Internet Engineering Task Force
Internet-Draft
Intended status: Standards Track
Expires: 1 October 2026

A. Luis Bustamante
Internet of Thinger SL

30 March 2026

Internet of Things Message Protocol (IOTMP)
draft-bustamante-iotmp-00

Abstract

IOTMP (Internet of Things Message Protocol) is a compact, binary, application-layer protocol for bidirectional communication between IoT devices and servers over persistent connections. It provides a resource-oriented model with native support for remote procedure calls, real-time data streaming, and API introspection. IOTMP uses PSON (Packed Sensor Object Notation) as its data encoding format, achieving significantly lower wire overhead than text-based alternatives. The protocol is designed for constrained devices with limited memory and bandwidth, while remaining expressive enough for complex IoT applications. IOTMP operates over TCP, TLS, or WebSocket transports and supports symmetric operation where both clients and servers may expose and invoke resources.

Status of This Memo

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

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 1 October 2026.

Copyright Notice

Table of Contents

1. Introduction

1.1. Purpose

IOTMP (Internet of Things Message Protocol) is an application-layer protocol designed for efficient, bidirectional communication between IoT devices and servers. It provides a resource-oriented, request-response model with native support for real-time data streaming, optimized for constrained devices with minimal memory and bandwidth.

1.2. Design Goals

* *Compactness:* Minimize wire overhead through binary encoding (PSON) and varint-based framing.
* *Bidirectional RPC:* Enable both client-to-server and server-to-client remote procedure calls over a single persistent connection.
* *Resource Orientation:* Model device capabilities as named resources with typed I/O, enabling auto-discovery and introspection.
* *Native Streaming:* Support periodic and event-driven data streams without additional protocol layers.
* *Transport Agnostic:* Operate over TCP, TLS, or WebSocket transports.
* *Embedded-Friendly:* Fit within the memory and processing constraints of microcontrollers (e.g., ESP32, Arduino, Zephyr-based devices).

1.3. Rationale for PSON

IOTMP uses PSON (Packed Sensor Object Notation) as its data encoding format rather than an existing binary encoding such as CBOR [RFC8949]. The primary motivation is architectural: PSON and IOTMP share the same encoding primitives, eliminating the need for two independent encoding systems on constrained devices.

Protocol-Encoding Integration. IOTMP’s message framing layer (Section 7) and PSON use identical encoding primitives: a tag-byte structure where type bits and an inline value are packed into a single byte, and varint encoding for variable-length integers. A single encoder/decoder implementation on a constrained device serves both the protocol framing layer (message fields) and the application data layer (payloads). Adopting an external encoding like CBOR would require a second, independent type system alongside the protocol’s own field encoding -- two tag formats, two integer encodings, two sets of type definitions -- increasing code size and implementation complexity for no functional benefit.

This integration is particularly valuable on the most constrained targets (Cortex-M0/M0+, 2-16 KB RAM), where every kilobyte of code matters. A complete IOTMP + PSON implementation shares encoding routines between framing and payload, keeping the combined footprint under 400 lines of C.

Comparison with CBOR. In terms of wire size, PSON and CBOR produce comparable encodings for typical IoT data. Neither format has a significant size advantage over the other. The differences are in scope and complexity: CBOR is a general-purpose encoding designed to cover a wide range of use cases, including semantic tags (CBOR type 6, with hundreds of IANA-registered values), indefinite-length encoding, half-precision floats, and extensible simple values. PSON deliberately omits these features, which add decoder complexity without benefit for IoT applications. This reduced scope, combined with the shared primitives with IOTMP, is what makes PSON the appropriate choice for this protocol -- not a claim of superior compression.

PSON is specified in a companion document: [PSON].

1.4. Scope

This specification defines:

* The message framing format
* The set of message types and their semantics
* The PSON binary encoding format
* The connection lifecycle (authentication, keepalive, disconnection)
* The resource model and streaming semantics

This specification does NOT define:

* Transport-level security configuration (deferred to TLS)
* Application-level authorization policies
* Specific resource naming conventions

1.5. Applicability

IOTMP is designed for scenarios where IoT devices maintain persistent connections with a server (broker) and require bidirectional communication. The protocol is most appropriate when:

* *Devices need to be both controlled and observed* over the same connection -- reading sensor data, invoking actions, and streaming telemetry without separate protocols for each pattern.

* *Low wire overhead matters* -- constrained devices on cellular, satellite, or low-bandwidth links where every byte counts.
* *The server needs to initiate operations on the device* -- pushing configuration, invoking remote procedures, or starting/stopping data streams, without the device polling.
* *Real-time interaction is required* -- interactive sessions (terminals, proxies), live dashboards, or event-driven streaming with sub-second latency.
* *API introspection is valuable* -- auto-discovery of device capabilities, schema-based validation, and automatic UI or API generation from resource descriptions.

IOTMP is NOT the best fit when:

* *One-to-many broadcast is the primary pattern* -- MQTT's publish/subscribe model is better suited for fan-out messaging to many subscribers on a topic.
* *Devices cannot maintain persistent connections* -- battery-powered sensors that wake briefly to send a single reading are better served by CoAP over UDP, which supports connectionless operation.
* *Interoperability with existing ecosystems is required* -- deployments that must integrate with existing MQTT or LwM2M infrastructure should use those protocols unless a bridge is acceptable.
* *The application is purely request-response with no device state* -- plain HTTP may be simpler when there is no need for persistent connections, streaming, or server-initiated operations.

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.

* *Client:* An IoT device or application that initiates a connection to a server.
* *Server:* The endpoint that accepts client connections and manages device state.
* *Resource:* A named endpoint on a client that can be invoked (read, written, or called) by the server.
* *Stream:* A continuous flow of data from a resource, initiated by a START_STREAM message and terminated by STOP_STREAM.

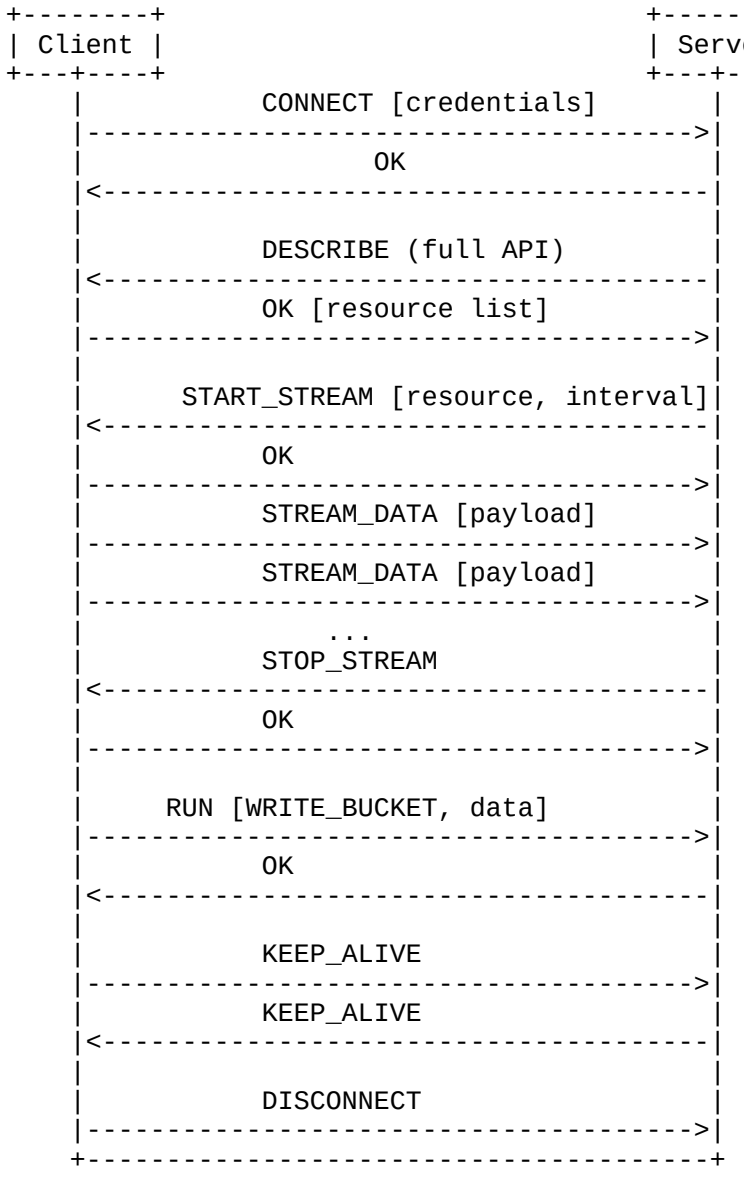
* *Stream ID:* A 16-bit unsigned integer used to correlate requests with responses and to identify active streams. The ID space is partitioned: client-initiated requests use even IDs (0, 2, 4, ...), server-initiated requests use odd IDs (1, 3, 5, ...).
* *PSON:* Packed Sensor Object Notation -- the binary encoding format used for structured data within IOTMP messages.
* *Varint:* A variable-length integer encoding where each byte uses 7 bits for data and 1 bit (MSB) as a continuation flag.

3. Protocol Overview

IOTMP operates over a persistent, full-duplex connection. After transport establishment, the client authenticates via a CONNECT/OK handshake. Once authenticated, either side can send messages at any time.

The protocol supports five interaction patterns:

1. *Request-Response:* A RUN or DESCRIBE message is sent with a Stream ID; the peer responds with OK or ERROR carrying the same Stream ID.
2. *Streaming:* A START_STREAM message initiates periodic or event-driven STREAM_DATA messages from a resource. STOP_STREAM terminates the stream.
3. *Fire-and-Forget:* DISCONNECT messages require no response.
4. *Keepalive:* The client periodically sends a KEEP_ALIVE message; the server MUST echo it back. This mechanism does not use Stream IDs. The server determines client liveness by monitoring incoming messages of any type.
5. *Server-Initiated RPC:* The server MAY send RUN messages to the client to read properties, set values, or invoke device resources.



4. Transport Layer

4.1. Supported Transports

IOTMP is designed to operate over reliable, ordered, byte-stream transports:

Transport	Default Port	Description
TCP	25204	Unencrypted TCP connection
TLS over TCP	25206	TLS-encrypted TCP connection
WebSocket	443	WebSocket upgrade over HTTPS, for NAT traversal

Table 1

Implementations SHOULD support TLS. Unencrypted TCP SHOULD only be used in trusted networks or during development.

4.2. Connection Establishment

1. The client establishes a transport-level connection (TCP handshake, TLS negotiation, or WebSocket upgrade).
2. Upon successful transport connection, the client MUST send a CONNECT message as the first IOTMP message.
3. The server MUST respond with OK or ERROR.
4. No other message types SHALL be sent by the client before receiving a successful CONNECT response.

4.3. WebSocket Transport

When operating over WebSocket, the following requirements apply:

* The client MUST request the subprotocol "iotmp" in the Sec-WebSocket-Protocol header during the WebSocket handshake.
* The server MUST confirm the "iotmp" subprotocol in the handshake response. If the server does not confirm it, the client MUST close the connection.
* Each WebSocket message MUST contain exactly one complete IOTMP message. IOTMP messages MUST NOT be fragmented across multiple WebSocket frames.
* The WebSocket message type MUST be binary (opcode 0x02).

4.4. Byte Ordering

All multi-byte numeric values (floats, doubles) MUST be encoded in *little-endian* byte order. Varint encoding is byte-order independent by design.

Rationale: Traditional Internet protocols use big-endian ("network byte order"), a convention established in the 1980s when the dominant networking hardware (Motorola 68000, SPARC, IBM mainframes) was big-endian. Today, virtually all microcontrollers and processors used in IoT are little-endian: ARM Cortex-M (the dominant embedded architecture), ESP32 (Xtensa), RISC-V, and x86. No widely deployed IoT microcontroller uses big-endian as its native byte order.

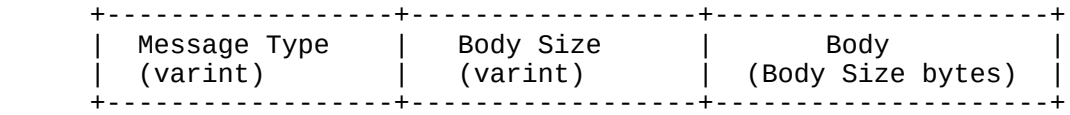
On the most constrained ARM cores (Cortex-M0/M0+), which lack a hardware byte-reversal instruction (REV), swapping a 32-bit float requires 4 instructions per value. For streaming telemetry -- the most common IoT workload -- this overhead is incurred on every sample. Little-endian encoding allows constrained devices to write float and double values directly from memory to the wire with no conversion.

This approach only affects [IEEE754] floats and doubles. Integers use varint encoding, which is byte-order independent. The same design choice is made by Protocol Buffers [ProtocolBuffers], which encodes fixed-width floats and doubles in little-endian regardless of platform.

5. Message Framing

Every IOTMP message on the wire consists of a *header* followed by a *body*.

5.1. Frame Format



* *Message Type:* A varint encoding one of the defined message type values Section 6.
* *Body Size:* A varint indicating the number of bytes in the body. A value of 0 indicates an empty body (used for KEEP_ALIVE).
* *Body:* A sequence of tagged fields Section 7, encoded as Body Size bytes.

5.2. Varint Encoding

IOTMP uses Protocol Buffers-style variable-length integer encoding:

* Each byte uses bits [6:0] for data and bit [7] as a continuation flag.
* If bit [7] is set, more bytes follow.

* If bit [7] is clear, this is the last byte.
* Least significant group comes first (little-endian byte order).

Examples:

Decimal	Hex	Varint Bytes (hex)
0	0x00	00
1	0x01	01
127	0x7F	7F
128	0x80	80 01
300	0x012C	AC 02
16384	0x4000	80 80 01

Table 2

Implementations MUST NOT encode varints longer than 4 bytes, representing values up to 2^28 - 1 (268,435,455). A receiver that encounters a varint that does not terminate within 4 bytes MUST treat it as a decode error and close the connection.

5.3. Maximum Message Size

Implementations MUST support messages up to at least 32,768 bytes (32 KB). Implementations MAY support larger messages. Both sides MAY declare their maximum message size using the "ms" parameter: the client in the CONNECT message and the server in the OK response Section 9.2. Each side MUST NOT send messages exceeding the peer’s declared maximum. If "ms" is not declared, the default of 32,768 bytes is assumed. A receiver that encounters a message exceeding its maximum size SHOULD close the connection.

IOTMP does not define message-level fragmentation. For data transfers that exceed the maximum message size, applications SHOULD use streaming Section 11: a START_STREAM opens a persistent channel over which arbitrarily large data can be sent as a sequence of STREAM_DATA messages, each within the negotiated size limit.

6. Message Types

The Message Type field identifies the purpose of each message:

Value	Name	Direction	Description
0x00	RESERVED	--	Reserved for future use. MUST NOT be sent.
0x01	OK	Both	Successful response to a request.
0x02	ERROR	Both	Error response to a request.
0x03	CONNECT	Client -> Server	Authentication request with credentials.
0x04	DISCONNECT	Both	Graceful connection termination.
0x05	KEEP_ALIVE	Client -> Server (echo: Server -> Client)	Connection liveness probe. Body MUST be empty.
0x06	RUN	Both	Execute a resource on the peer.
0x07	DESCRIBE	Both	Request resource metadata/API description.
0x08	START_STREAM	Both	Begin streaming data from a resource.
0x09	STOP_STREAM	Both	Stop an active stream.
0x0A	STREAM_DATA	Both	Carry stream payload data.

Table 3

Values 0x0B through 0xFF are reserved for future use.

6.1. Response Correlation

Messages that expect a response (CONNECT, RUN, DESCRIBE, START_STREAM, STOP_STREAM) MUST include a Stream ID field. The response (OK or ERROR) MUST carry the same Stream ID to correlate with the request.

Note: The Stream ID serves two roles depending on the message type. For one-time request-response exchanges (RUN, DESCRIBE), the Stream ID is a short-lived correlation identifier that is released when the response is received. For streaming operations (START_STREAM), the same Stream ID becomes a long-lived identifier for the stream, used in subsequent STREAM_DATA and STOP_STREAM messages until the stream is terminated. In both cases, the lifecycle rules in Section 6.2 apply.

Request Timeout: If a response (OK or ERROR) is not received within a reasonable time, the sender SHOULD consider the request failed and release the Stream ID. Implementations SHOULD use a default timeout of 30 seconds for RUN, DESCRIBE, START_STREAM, and STOP_STREAM requests. The CONNECT message has a separate timeout (RECOMMENDED: 10 seconds, Section 14.5). Implementations MAY use longer timeouts for specific resources that are known to require extended processing.

6.2. Stream ID Partitioning

Since IOTMP is a symmetric protocol where both sides may initiate requests concurrently, the Stream ID space is partitioned to avoid collisions:

* *Client-initiated requests* MUST use *even* Stream IDs (0, 2, 4, ..., 65534).
* *Server-initiated requests* MUST use *odd* Stream IDs (1, 3, 5, ..., 65535).

This partitioning ensures that responses (OK, ERROR) and stream data (STREAM_DATA) can be unambiguously correlated with the originating request, even when both sides send requests simultaneously with the same numeric value.

Stream ID Lifecycle:

A Stream ID is considered *active* from the moment the request message is sent until:

* For request-response (RUN, DESCRIBE): the corresponding OK or ERROR is received, OR
* For streams (START_STREAM): the stream is terminated by STOP_STREAM and its OK response.

An endpoint MUST NOT reuse a Stream ID that is currently active. Once a Stream ID is released, it MAY be reused for a new request. Implementations SHOULD favor low-valued Stream IDs to minimize varint encoding overhead.

A receiver that encounters a request with a Stream ID from the wrong partition (e.g., a client-initiated message with an odd Stream ID) MUST respond with ERROR (400) and SHOULD log the violation.

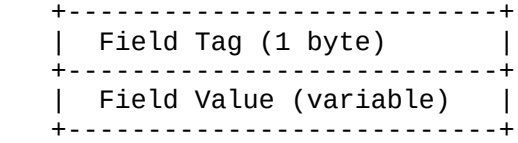
6.3. Message Direction

IOTMP is a symmetric protocol. After authentication, both sides MAY send any message type except CONNECT and KEEP_ALIVE. The directional constraints are:

* *Client -> Server only:* CONNECT (authentication request), KEEP_ALIVE (liveness probe).
* *Server -> Client only (in response):* KEEP_ALIVE echo Section 9.4.
* *Both directions:* OK, ERROR, DISCONNECT, RUN, DESCRIBE, START_STREAM, STOP_STREAM, STREAM_DATA.

7. Message Fields

The message body consists of zero or more tagged fields. Each field is encoded as:



7.1. Field Tag Encoding

The field tag is a single byte combining the field number and wire type:

Tag = (field_number << 3) | wire_type

* *field_number* (bits [7:3]): Identifies which field this is (5 bits, values 0-31).

* *wire_type* (bits [2:0]): Determines how the field value is encoded.

7.2. Wire Types

Value	Name	Description
0x00	varint	Value is a varint-encoded unsigned integer.
0x01	bytes	Value is a varint-encoded length followed by raw bytes. The content is opaque to the protocol and MAY carry any application-defined binary format.
0x02	pson	Value is PSON encoded Section 8.

Table 4

Wire types 0x03-0x07 are reserved for future use.

7.3. Defined Fields

Field Number	Name	Allowed Wire Types	Description
0x01	STREAM_ID	varint	16-bit stream identifier for request/response correlation.
0x02	PARAMETERS	varint, pson	Request parameters (e.g., server operation code, stream interval).
0x03	PAYLOAD	pson, bytes	Primary data payload (credentials, resource data, error info).
0x04	RESOURCE	varint, pson	Resource identifier (string name or 16-bit hash).

Table 5

Field 0x00 is reserved and MUST NOT be used. Reserving field zero allows zero-initialized field variables to be distinguished from valid field numbers and serves as a sentinel value for error detection -- a convention shared with Protocol Buffers and other binary protocols. Fields 0x05-0x07 are reserved for future use. A maximum of 8 fields (0x00-0x07) are supported per message.

7.4. Field Presence

Fields are OPTIONAL. A receiver MUST handle messages with any subset of fields present. The field_mask is implicit in the encoding -- a field is present if and only if its tag appears in the body. Fields MAY appear in any order within the message body. A receiver MUST NOT assume a specific field ordering.

7.5. Field Encoding Rules

* *STREAM_ID:* MUST be encoded as varint (wire type 0x00). Values range from 0 to 65535.
* *PARAMETERS:* When carrying a simple integer (e.g., a server operation code or stream interval), SHOULD be encoded as varint. When carrying structured data (e.g., CONNECT parameters), MUST be encoded as pson.
* *PAYLOAD:* MUST be encoded as pson. Implementations MAY use wire type bytes (0x01) to carry opaque binary data in application-defined formats.
* *RESOURCE:* When carrying a resource name (string), MUST be encoded as pson. When carrying a resource hash (unsigned integer), SHOULD be encoded as varint for compactness; implementations MAY also encode it as pson. Receivers MUST accept both wire types for integer values. See Section 10.6 for resource hashing.

A formal CDDL grammar defining all message structures and field requirements is provided in Appendix B.

8. Data Encoding: PSON

IOTMP uses PSON (Packed Sensor Object Notation) as its data encoding format. PSON is a compact, self-describing binary encoding that efficiently represents the data types commonly found in IoT applications: integers, floating-point numbers, booleans, strings, binary data, arrays, and key-value maps.

PSON is specified in a separate document: [PSON]. This section provides a brief summary; the PSON specification is the normative reference for encoding and decoding rules.

8.1. Summary

Each PSON value begins with a tag byte that encodes both the wire type (3 bits) and an inline value (5 bits). Small values (0-30) are encoded in a single byte. Eight wire types are defined:

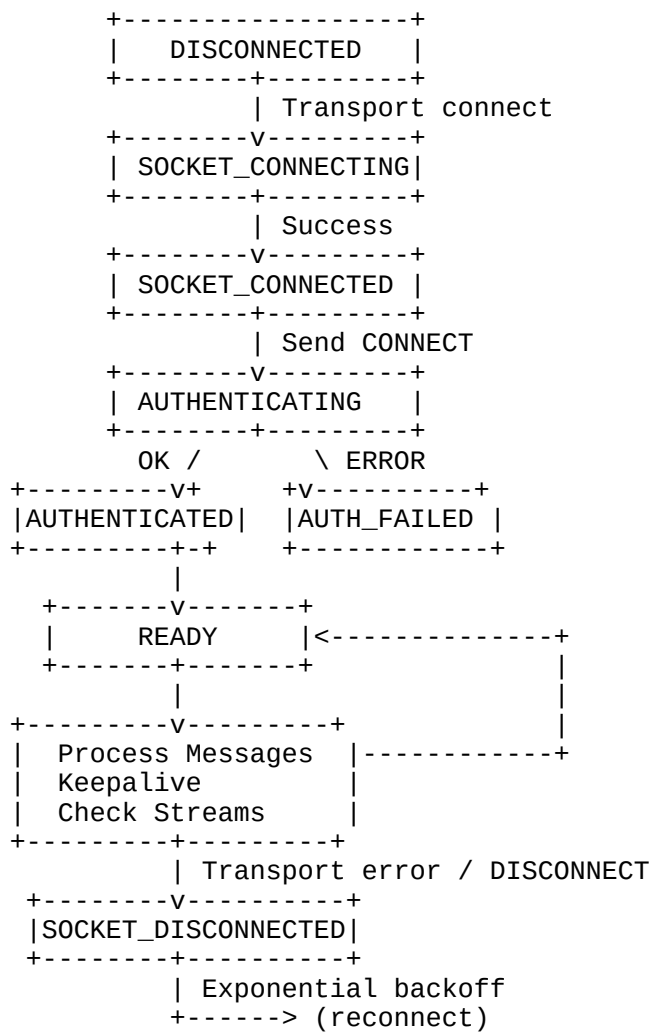
Wire Type	Name	Description
0	unsigned_t	Unsigned integer
1	signed_t	Negative integer (stored as absolute value)
2	floating_t	IEEE 754 float or double
3	discrete_t	Boolean (false/true) or null
4	string_t	UTF-8 string
5	bytes_t	Raw binary data
6	map_t	Key-value map (keys are strings)
7	array_t	Ordered array

Table 6

For the complete encoding rules, wire format, and examples, see [PSON].

9. Connection Lifecycle

9.1. State Machine



9.2. CONNECT Message

The client MUST send a CONNECT message as the first message after transport establishment.

CONNECT Fields:

Field	Content
STREAM_ID	An even 16-bit value for response correlation (client partition, Section 6.2).
PAYLOAD	Authentication data. Format depends on the authentication type ("at" parameter).
PARAMETERS	(OPTIONAL) A PSON map with connection options.

Table 7

Connection Parameters (PARAMETERS field):

Key	Type	Description	Default
"v"	uint	Protocol version. Current version: 1.	1
"ka"	uint	Keepalive interval in seconds. Max: 1800.	60
"at"	uint	Authentication type (see below).	0
"ms"	uint	Maximum message size in bytes the client can accept. Min: 1024.	32768

Table 8

Authentication Types:

The "at" parameter determines the format and interpretation of the PAYLOAD field:

Code	Name	PAYLOAD Format
0	Credentials	A PSON array of 3 strings: [namespace, device_id, credential]. The namespace identifies the organizational scope (account, project, or tenant) under which the device is registered. The device_id uniquely identifies the device within the namespace. The credential is the shared secret.
1	Token	A PSON string containing a bearer token (e.g., JWT, API key, or opaque token). The server determines the device identity from the token claims or by lookup.
2	Certificate	Device identity is established via the client certificate presented during the TLS handshake (mTLS). This type MUST only be used over TLS with client certificate authentication. PAYLOAD is OPTIONAL: when the certificate identifies only the namespace (e.g., via CN or SAN), the PAYLOAD MUST be a PSON array of 2 strings [namespace, device_id] so the server can identify the specific device. When the certificate identifies both namespace and device, the PAYLOAD MAY be absent or empty. If PAYLOAD is present, the server MUST verify that the namespace matches the certificate identity and MUST reject the connection with ERROR (401) on mismatch. See Section 14.3.1 for security considerations on fleet certificates.

Table 9

Authentication types 3-255 are reserved for future use (e.g., OAuth 2.0 device flow, challenge-response). A server that receives an unrecognized authentication type MUST respond with ERROR (400).

If the "at" parameter is omitted, type 0 (Credentials) is assumed and the PAYLOAD MUST follow the Credentials format.

Server Response:

* *OK:* Authentication succeeded. The connection enters the READY state. The Stream ID used in the CONNECT message is released upon receiving this response and MAY be reused. The server MAY include a PARAMETERS field with a PSON map containing its own connection parameters (e.g., "ms" to declare the server's maximum message size). The client MUST NOT send messages exceeding the server's declared maximum.
* *ERROR:* Authentication failed or version not supported. The Stream ID is released. The server SHOULD close the connection.

Extensibility of Connection Parameters:

The PARAMETERS maps in both the CONNECT and OK messages are extensible. Implementations MUST ignore unknown keys in PARAMETERS and process known keys normally. Future versions of this protocol or application profiles MAY define additional keys for capability negotiation (e.g., supported authentication types, maximum concurrent streams, or protocol extensions). This allows the connection handshake to evolve without breaking backward compatibility.

9.3. Version Negotiation

The "v" parameter in the CONNECT message indicates the protocol version the client wishes to use.

* If the "v" parameter is omitted, the server MUST assume protocol version 1.
* If the server supports the requested version, it MUST proceed with authentication and respond with OK or ERROR based on the credentials.
* If the server does not support the requested version, it MUST respond with an ERROR message with status code 400 and SHOULD include the supported versions in the PAYLOAD. The server MUST then close the connection.

Version Mismatch Response Example:

```
ERROR Message:
  STREAM_ID:  (mirrors CONNECT)
  PARAMETERS: 400
  PAYLOAD:    {"error": "Unsupported protocol version",
               "supported": [1]}
```

The client MAY retry the connection using a version from the supported list.

9.4. Keepalive

The keepalive mechanism verifies connection liveness for both sides. KEEP_ALIVE is a client-initiated message with an empty body (Body Size = 0). The server echoes each KEEP_ALIVE back, allowing the client to confirm reachability. The server determines client liveness by monitoring incoming traffic -- any message from the client (KEEP_ALIVE, STREAM_DATA, RUN, or any other type) resets the server's inactivity timer.

This asymmetric design simplifies server implementation: the server does not need to maintain per-connection keepalive send timers -- it only needs a single inactivity timeout per connection. This is the same model used by [MQTT] (PINGREQ/PINGRESP) and WebSocket (Ping/Pong).

Client behavior:

* The client MUST send a KEEP_ALIVE message if no other message has been sent to the server within the negotiated keepalive interval (default: 60 seconds). If the client sends other messages (RUN, STREAM_DATA, etc.) within the interval, it MAY skip the KEEP_ALIVE for that period, since the server treats any incoming message as proof of liveness.
* If the client does not receive any message from the server (KEEP_ALIVE echo, STREAM_DATA, OK, RUN, or any other message type) within the keepalive interval plus a tolerance margin (RECOMMENDED: 15 seconds), it SHOULD consider the connection lost and initiate reconnection Section 9.7.

Server behavior:

* Upon receiving a KEEP_ALIVE message, the server MUST respond with a KEEP_ALIVE message (echo). This enables the client to verify that the server is still reachable.
* The server MUST NOT initiate KEEP_ALIVE messages. The server's role is limited to echoing client-initiated KEEP_ALIVE messages.
* The server SHOULD consider a client disconnected if no message of any type is received within the keepalive interval plus a tolerance margin (RECOMMENDED: 15 seconds). The server MUST then close the connection and release all associated resources (streams, Stream IDs).

9.5. Disconnection

Either side MAY send a DISCONNECT message to initiate graceful shutdown. Upon receiving DISCONNECT, the peer SHOULD close the transport connection. No response is expected.

DISCONNECT Fields:

Field	Content
PARAMETERS	(OPTIONAL) Status code (varint).
PAYLOAD	(OPTIONAL) Additional information (PSON-encoded).

Table 10

A DISCONNECT message with no fields indicates a normal graceful shutdown.

9.6. Server Redirect

A server MAY redirect a client to a different server in two scenarios:

At connection time: The server responds to CONNECT with an ERROR message containing a redirect status code and the target server information in the PAYLOAD.

During an active connection: The server sends a DISCONNECT message with a redirect status code and the target server in the PAYLOAD. The client SHOULD close the current connection and connect to the indicated server.

Redirect Status Codes:

Code	Meaning	Client Behavior
301	Moved Permanently	Client MUST update its stored server address and connect to the new server for all future connections.
307	Temporary Redirect	Client SHOULD connect to the indicated server now, but MUST retain the original server address for future connections.

Table 11

Redirect Payload:

The PAYLOAD MUST be a PSON map containing the target server information:

Key	Type	Description	Required
"host"	string	Hostname or IP address of the target.	Yes
"port"	uint	Port number of the target.	No

Table 12

If "port" is omitted, the client SHOULD use the same port as the current connection.

Redirect Example (at connection time):

```
Client -> Server:  CONNECT [credentials]
Server -> Client:  ERROR
                    STREAM_ID:  (mirrors CONNECT)
                    PARAMETERS: 307
                    PAYLOAD:    {"host": "server2.example.com",
                                 "port": 25206}
```

Redirect Example (during active connection):

```
Server -> Client:  DISCONNECT
                    PARAMETERS: 301
                    PAYLOAD:    {"host": "new-server.example.com"}
```

9.7. Reconnection

Clients SHOULD implement automatic reconnection with exponential backoff:

* Initial delay: 5 seconds.
* Doubling on each failed attempt.
* Maximum delay: 60 seconds.
* Reset to initial delay upon successful authentication.

10. Resource Model

10.1. Overview

IOTMP models capabilities as *resources* -- named endpoints that can be invoked by the peer. Both clients and servers MAY expose resources. Each resource has a defined I/O type that determines how data flows through it.

10.2. I/O Types

Type	Code	Data Flow	Description
none	0	--	No callback registered.
run	1	-> (trigger)	Action with no data input or output.
input	2	Caller -> Resource	Receives data (actuator, setter).
output	3	Resource -> Caller	Produces data (sensor, getter).
input_output	4	Bidirectional	Accepts input and produces output (property).

Table 13

10.3. Resource Invocation (RUN Message)

Either side MAY send a RUN message to invoke a resource on the peer:

1. The receiver looks up the resource by the RESOURCE field value.
2. The receiver executes the resource callback, passing PAYLOAD as input and collecting output.
3. The receiver responds with OK (including any output in PAYLOAD) or ERROR.

RUN Fields:

Field	Content
STREAM_ID	Request correlation identifier.
RESOURCE	Resource name (string) or resource hash (unsigned integer, Section 10.6).
PAYLOAD	(OPTIONAL) Input data for the resource.

Table 14

10.4. Resource Description (DESCRIBE Message)

The DESCRIBE message enables API introspection. Either peer can request:

1. *Full API:* DESCRIBE with no RESOURCE field -> the receiving peer responds with a map of all resource names and their I/O types.
2. *Single Resource:* DESCRIBE with a RESOURCE field -> the receiving peer responds with the resource’s input/output data and optional schema information.

10.4.1. Description Versioning

DESCRIBE responses MUST include a "v" (version) field at the root level to indicate the description format version. The current version is *1*.

Version	Description
1	Resource descriptions with optional [JSON-Schema] support for input/output validation and introspection.

Table 15

Receivers that encounter an unrecognized version SHOULD ignore unknown fields and interpret the response on a best-effort basis. The version field enables forward-compatible evolution of the description format as the protocol evolves (e.g., adding richer metadata, new keywords, or alternative schema languages in future versions).

10.4.2. Full API Description

A DESCRIBE with no RESOURCE field returns a map containing the description version and a "res" field with all resource names and their metadata.

```
{
  "v": 1,
  "res": {
    "temperature": {
      "fn": 3,
      "description": "Room temperature sensor"
    },
    "led": {"fn": 2, "description": "Status LED control"},
    "relay": {"fn": 4},
    "reboot": {"fn": 1}
  }
}
```

The "res" field is a PSON map where each key is a resource name and each value is a resource descriptor. This separation ensures that protocol-level fields (such as "v") do not collide with resource names.

Each resource entry MUST include a "fn" field with the I/O type code Section 10.2. Each resource entry MAY include a "description" field with a human-readable string describing the resource's purpose.

10.4.3. Single Resource Description

A DESCRIBE with a RESOURCE field returns the resource's input/output information. The response contains "in" and/or "out" fields (depending on the resource's I/O type), each structured as an object with the following fields:

Field	Required	Description
value	YES	Current or sample data from the resource callback.
schema	NO	A [JSON-Schema] object describing the expected data structure.

Table 16

Example -- Sensor with input and output (I/O type input_output):

```
{
  "v": 1,
  "in": {
    "value": {"brightness": 128},
    "schema": {
      "type": "object",
      "properties": {
        "brightness": {
          "type": "integer",
          "minimum": 0,
          "maximum": 255,
          "description": "LED brightness level"
        }
      }
    }
  },
  "out": {
    "value": {"celsius": 22.5, "fahrenheit": 72.5},
    "schema": {
      "type": "object",
      "properties": {
        "celsius": {
          "type": "number",
          "minimum": -40,
          "maximum": 125,
          "description": "Temperature in Celsius"
        },
        "fahrenheit": {
          "type": "number",
          "minimum": -40,
          "maximum": 257,
          "description": "Temperature in Fahrenheit"
        }
      }
    }
  }
}
```

Example -- Output-only sensor (I/O type output):

```
{
  "v": 1,
  "out": {
    "value": {
      "latitude": 40.4168,
      "longitude": -3.7038,
      "altitude": 650.0
    },
    "schema": {
      "type": "object",
      "properties": {
        "latitude": {
          "type": "number",
          "minimum": -90,
          "maximum": 90
        },
        "longitude": {
          "type": "number",
          "minimum": -180,
          "maximum": 180
        },
        "altitude": {
          "type": "number",
          "description": "Altitude in meters"
        }
      }
    }
  }
}
```

Example -- Input-only actuator (I/O type input):

```
{
  "v": 1,
  "in": {
    "value": {"on": false},
    "schema": {
      "type": "object",
      "properties": {
        "on": {"type": "boolean", "description": "Relay state"}
      }
    }
  }
}
```

Example -- Minimal description without schema:

```
{
  "v": 1,
  "out": {
    "value": {"celsius": 22.5, "fahrenheit": 72.5}
  }
}
```

The "schema" field, when present, MUST be a valid [JSON-Schema] object. Implementations SHOULD use the following JSON Schema keywords where applicable:

Keyword	Purpose
type	Data type (boolean, integer, number, string, object, array).
properties	Named fields within an object.
items	Element schema for arrays.
description	Human-readable field description.
minimum	Lower bound for numeric values.
maximum	Upper bound for numeric values.
enum	Array of allowed values.
readOnly	Field cannot be written.
writeOnly	Field cannot be read.
default	Default value.
required	Array of required property names.

Table 17

The "schema" field is OPTIONAL. When absent, receivers SHOULD infer data types from the sample values in "value". Resources that only have output (e.g., sensors) include only "out"; resources that only have input (e.g., actuators) include only "in"; resources with no data (type run) MAY omit both.

Servers MAY use the schema information to validate incoming data before forwarding it to the device, generate user interfaces automatically, or produce API documentation compatible with standards such as OpenAPI.

10.5. Stream Echo

When a resource with one or more active streams receives input via a RUN message, the resource owner (typically the client) SHOULD immediately send a STREAM_DATA message with the updated resource state on *each* active stream for that resource. This enables real-time synchronization -- for example, when a server sets a property value via RUN, all dashboards subscribed to that resource’s stream receive the updated state without waiting for the next periodic sample.

Behavior:

* Stream echo MUST occur *after* the resource callback has executed and the RUN response (OK or ERROR) has been sent. The echo reflects the resource state after the operation.
* The STREAM_DATA message MUST use the Stream ID of the active stream, not the Stream ID of the RUN request.
* If the resource has multiple active streams (from different server requests or stream configurations), the resource owner MUST send a separate STREAM_DATA on each active stream.
* If compact mode Section 11.4 is active on a stream, the echo STREAM_DATA MUST follow the compact encoding rules (PSON array with positional values).
* Stream echo is enabled by default. Implementations MAY allow disabling echo on a per-resource basis if the application does not require immediate synchronization.

Example:

```
S -> C:  START_STREAM
           (Stream 0x01, resource: "relay", interval: 5000)
C -> S:  OK (Stream 0x01)
C -> S:  STREAM_DATA (Stream 0x01, {"on": false})
           <- initial state

S -> C:  RUN (Stream 0x03, resource: "relay",
           payload: {"on": true})
C -> S:  OK (Stream 0x03)
C -> S:  STREAM_DATA (Stream 0x01, {"on": true})
           <- echo (immediate)

  ... 5 seconds later ...
C -> S:  STREAM_DATA (Stream 0x01, {"on": true})
           <- periodic sample
```

In this example, the server has an active stream (Stream ID 0x01) observing the "relay" resource. When the server changes the relay state via RUN (Stream ID 0x03), the client sends the RUN response (OK) first, then immediately echoes the updated state on the active stream (Stream ID 0x01). This ensures that any dashboard or application consuming the stream sees the change in real time, without waiting for the next periodic sample.

10.6. Resource Hashing

As an optimization for high-frequency interactions, IOTMP supports identifying resources by a numeric hash of their name instead of the full string. This eliminates the need for prior DESCRIBE exchanges or state synchronization between peers.

The RESOURCE field in any message (RUN, DESCRIBE, START_STREAM, STOP_STREAM) accepts both:

* A *PSON string*: the resource name. Always works, self-documenting. Required for resources with dynamic path parameters (e.g., paths containing /).
* A *PSON unsigned integer*: a 16-bit hash of the resource name. Compact (1-3 bytes in varint), requires no prior negotiation.

The receiver MUST support both forms. When a PSON unsigned integer is received, the receiver computes the hash of each of its defined resources and matches the incoming value. When a PSON string is received, the receiver looks up the resource by name.

Hash Function:

Implementations MUST use FNV-1a (Fowler-Noll-Vo) truncated to 16 bits:

```
hash = 0x811C9DC5          // FNV offset basis (32-bit)
for each byte b in name:
    hash = hash XOR b
    hash = hash * 0x01000193  // FNV prime (32-bit)
return hash AND 0xFFFF      // truncate to 16 bits
```

This function is simple to implement on constrained devices and provides good distribution across typical IoT resource names.

Reference Implementation (C):

```
uint16_t fnv1a_16(const char *name) {
    uint32_t hash = 0x811C9DC5;
    while (*name) {
        hash ^= (uint8_t)*name++;
        hash *= 0x01000193;
    }
    return (uint16_t)(hash & 0xFFFF);
}
```

Test Vectors:

Resource Name	FNV-1a (32-bit)	Truncated (16-bit)	Hex
"temperature"	0xE9F2A935	0xA935	A935
"humidity"	0x25C5B9A0	0xB9A0	B9A0
"led"	0x406AEACA	0xEACA	EACA
"relay"	0x16B481C2	0x81C2	81C2
"reboot"	0x87729FB8	0x9FB8	9FB8

Table 18

Example:

```
RUN with string:   RESOURCE = "temperature"  (12 bytes in PSON)
RUN with hash:     RESOURCE = 0xA935         (3 bytes in PSON)
```

Both identify the same resource. The hash form saves 9 bytes per message.

Applicability:

Resource hashing is best suited for simple resource names without dynamic parameters. Resources with path-based parameters (e.g., fs/home/config.txt) SHOULD use the string form, since the path contains dynamic segments that cannot be pre-hashed.

Collision Handling:

With a 16-bit hash space (65,536 values), the probability of at least one collision follows the birthday problem. For typical IoT devices with small resource sets, the probability is very low:

Resources	P(collision)
10	< 0.1%
20	_(0.3%)
30	_(0.7%)
50	_(1.9%)
100	_(7.3%)

Table 19

For constrained devices with fewer than 30 resources, hash-based identification is safe for practical purposes. Implementations MUST verify at device startup that no two resource names produce the same hash. If a collision is detected, the implementation MUST fall back to string-based identification for the affected resources and SHOULD emit a diagnostic warning (e.g., via log output or assertion). If resources are added dynamically at runtime, the implementation MUST check for hash collisions against all existing resources before enabling hash-based identification for the new resource.

Unmatched Hash Handling:

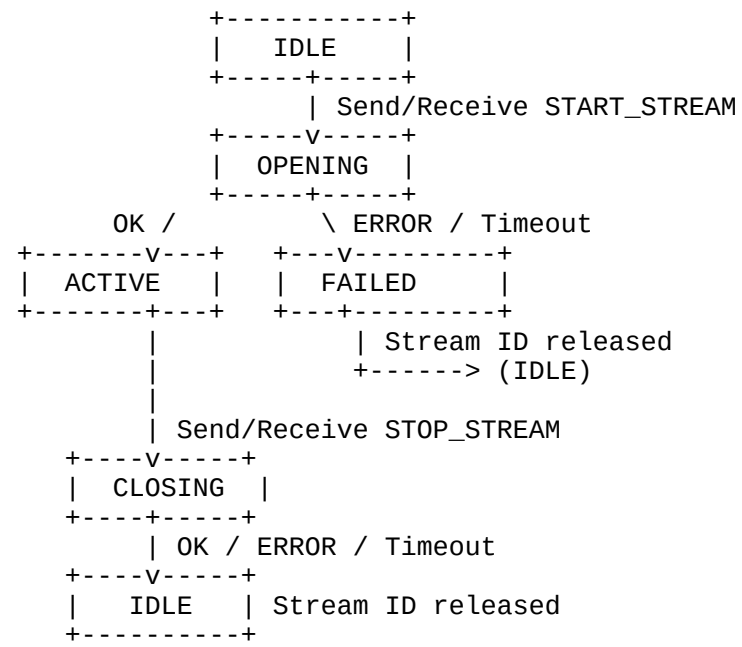
If a receiver receives a PSON unsigned integer in the RESOURCE field and no defined resource matches the hash value, the receiver MUST respond with ERROR (404). This is consistent with the behavior for unrecognized string resource names Section 15.3.

11. Streaming

11.1. Stream Lifecycle

11.1.1. Stream State Machine

Each stream is identified by a Stream ID and transitions through the following states:



State Definitions:

State	Description
IDLE	No active stream for this Stream ID. The ID is available for reuse.
OPENING	START_STREAM has been sent; awaiting OK or ERROR response.
ACTIVE	Stream is established. STREAM_DATA messages are being sent by the resource owner.
CLOSING	STOP_STREAM has been sent; awaiting OK or ERROR response.
FAILED	START_STREAM was rejected (ERROR) or timed out. Stream ID is released.

Table 20

Transition Rules:

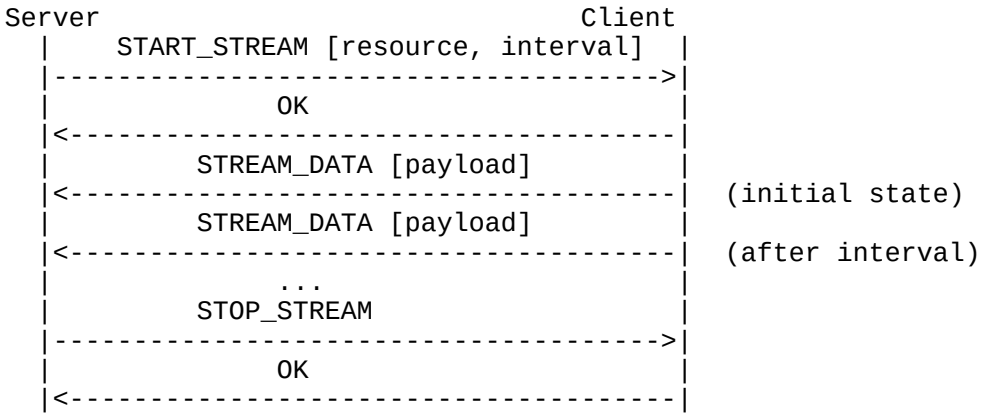
* *IDLE -> OPENING:* A START_STREAM message is sent or received. The Stream ID is now active and MUST NOT be reused until released.
* *OPENING -> ACTIVE:* An OK response is received for the START_STREAM. The resource owner MUST send an initial STREAM_DATA immediately Section 11.2.
* *OPENING -> FAILED -> IDLE:* An ERROR response is received, or the request times out Section 6.1. The Stream ID is released.
* *ACTIVE -> CLOSING (initiator):* The stream initiator sends a STOP_STREAM to terminate the stream.
* *ACTIVE -> CLOSING (resource owner):* The resource owner MAY also send a STOP_STREAM with the stream's Stream ID to terminate a stream it did not initiate. This allows the resource owner to signal that the resource is no longer available (e.g., a terminal session has ended, a sensor has failed, or the device is running low on resources). The stream initiator MUST respond with OK and release the Stream ID.
* *CLOSING -> IDLE:* An OK or ERROR response is received for the STOP_STREAM request. The Stream ID is released. If no response is received within the request timeout, the stream is considered closed and the Stream ID is released.
* *Any state -> IDLE (transport loss):* If the transport connection is lost, all streams are immediately terminated and all Stream IDs are released Section 15.3.

Edge Cases:

Condition	Required Behavior
STREAM_DATA received while in OPENING state	Receiver SHOULD buffer or discard; sender MUST NOT send STREAM_DATA before receiving OK.
START_STREAM received for an already ACTIVE Stream ID	Receiver MUST respond with ERROR (409 Conflict).
STOP_STREAM received for a Stream ID not in ACTIVE state	Receiver MUST respond with ERROR (409 Conflict).
STOP_STREAM rejected with ERROR	Initiator MUST still consider the stream terminated and release the Stream ID.
RUN received targeting a resource with an ACTIVE stream	Resource executes normally. If stream echo is enabled Section 10.5, the client MUST send a STREAM_DATA with the updated state on all ACTIVE streams for that resource.

Table 21

11.1.2. Interaction Diagram



11.2. START_STREAM Message

Field	Content
STREAM_ID	Identifies this stream. Used in subsequent STREAM_DATA/STOP.
RESOURCE	Name of the resource to stream.
PARAMETERS	(OPTIONAL) Stream configuration. See below.

Table 22

PARAMETERS Encoding:

When PARAMETERS is a simple unsigned integer (varint), it represents the stream interval in milliseconds. A value of 0 indicates event-driven streaming (no periodic sampling).

When PARAMETERS is a PSON map, it MAY contain the following fields:

Key	Type	Description	Default
"i"	uint	Stream interval in milliseconds. 0 = event-driven.	0
"cm"	bool	Compact mode. Request compact encoding for STREAM_DATA.	false

Table 23

For backward compatibility, a server that only needs to set the interval MAY send PARAMETERS as a plain varint. A client MUST accept both encodings.

Upon receiving START_STREAM:

1. The client registers the stream (Stream ID -> resource + interval + encoding mode).
2. The client evaluates whether the resource supports compact mode (see Section 11.4).

3. The client responds with OK. If compact mode was activated, the OK MUST include a PARAMETERS field as a PSON map with "cm": true. If compact mode is not active, the "cm" field SHOULD be omitted.
4. The client immediately sends one STREAM_DATA with the current resource state.
5. If an interval is specified, the client sends STREAM_DATA at that interval.

11.3. STREAM_DATA Message

Field	Content
STREAM_ID	The stream identifier from START_STREAM.
PAYLOAD	The resource output data, PSON-encoded.

Table 24

STREAM_DATA messages do NOT require a response.

11.4. Compact Encoding Mode

Compact encoding mode reduces per-message overhead for resources that consistently return PSON maps with the same set of keys. It eliminates the repetition of string keys in every STREAM_DATA message after the first.

11.4.1. Negotiation

1. The server requests compact mode by setting "cm": true in the START_STREAM PARAMETERS. If "cm" is absent or false, the server does not want compact mode and the client MUST NOT activate it.
2. The client evaluates whether the resource output is suitable for compaction (i.e., it produces a PSON map with a stable set of keys).
3. If the server requested compact mode and the resource supports it, the client activates it for this stream and confirms by including "cm": true in the OK response PARAMETERS. If the resource does not support compact mode, the client omits "cm" from the OK response and sends all STREAM_DATA messages as full PSON maps.
4. The server MUST check the OK response for "cm": true before assuming compact encoding is active. If "cm" is absent from the OK response, the server MUST decode all STREAM_DATA messages as standard PSON values.

11.4.2. Schema Establishment

The *first STREAM_DATA* message on a compact stream MUST be a full PSON map with string keys. This message establishes the *key order* (schema) for all subsequent messages on this stream.

```
First STREAM_DATA:
  PAYLOAD: {"temperature": 23.5, "humidity": 60}
           (PSON map -- schema)
```

Both the client and server MUST store the key order from this first map. The key order is determined by the iteration order of the PSON map fields.

11.4.3. Compact STREAM_DATA

All subsequent STREAM_DATA messages on this stream MUST encode the PAYLOAD as a *PSON array* with values in the same positional order as the keys in the schema message.

```
Subsequent STREAM_DATA:
  PAYLOAD: [23.6, 61]
           (PSON array -- values only)
```

The server reconstructs the full map by matching each array position to the corresponding key from the schema.

11.4.4. Rules

* The resource owner MUST NOT change the structure of the output (add, remove, or reorder keys at any nesting level) during the lifetime of a compact stream. If the output structure changes, the resource owner MUST terminate the stream with STOP_STREAM. The stream initiator MAY then open a new stream to renegotiate the schema.
* If a value is absent or unavailable for a given key, the client MUST encode null at that position to maintain alignment.
* The schema is valid for the lifetime of the stream. It is reset only when the stream is stopped (STOP_STREAM) and a new stream is started (START_STREAM).
* If the resource output is not a PSON map (e.g., a single number, boolean, string, or binary), compact mode has no effect. The client SHOULD send the value as-is.
* The PSON array MUST contain exactly the same number of elements as keys in the schema. If the array length does not match the schema length, the receiver SHOULD treat the STREAM_DATA as malformed and MAY close the stream with STOP_STREAM.

11.4.5. Recursive Compaction

Compact mode applies recursively to nested PSON maps. When the schema message contains a map value nested within the top-level map, that nested map is also converted to a positional array in subsequent compact messages. This process applies at all nesting levels:

* A value that is a PSON map in the schema message becomes a PSON array in compact messages, with its own positional order derived from the schema.
* A value that is a scalar (number, boolean, string, null) or a PSON array remains encoded as-is.
* The null placeholder rule applies at every nesting level for absent values.

Distinguishing arrays from compacted maps:

The receiver MUST store the type of each value from the schema message (the first STREAM_DATA). In subsequent compact messages, when the receiver encounters a PSON array at a given position, it checks the schema to determine the original type:

* If the schema value at that position was a *PSON map*, the array is a compacted map and MUST be expanded using the stored key order.
* If the schema value at that position was a *PSON array*, the array is a real array and MUST be kept as-is. Real arrays MAY change in length between messages -- they are not subject to compaction or positional alignment.

This distinction is unambiguous because the schema message provides the complete type information for every value at every nesting level.

Example -- Mixed maps and arrays:

Schema message (first STREAM_DATA):

```
  PAYLOAD: {"temperature": 23.5,
            "tags": ["indoor", "sensor"],
            "location": {"lat": 40.4168, "lon": -3.7038}}
```

Compact messages (subsequent STREAM_DATA):

```
  PAYLOAD: [23.6, ["indoor", "active", "new"], [40.4200, -3.7035]]
```

The receiver reconstructs the full structure using the schema:

* Position 0: "temperature" -- scalar in schema, value is 23.6 (kept as-is).

* Position 1: "tags" -- array in schema, value is ["indoor", "active", "new"] (kept as-is, length may vary).
* Position 2: "location" -- map in schema, value is [40.4200, -3.7035] (expanded to {"lat": 40.4200, "lon": -3.7035} using stored key order).

11.4.6. Example

```
Server -> Client: START_STREAM
  STREAM_ID: 0x00A1
  RESOURCE:  "environment"
  PARAMETERS: {"i": 5000, "cm": true}

Client -> Server: OK
  STREAM_ID: 0x00A1
  PARAMETERS: {"cm": true}

Client -> Server: STREAM_DATA (schema message)
  STREAM_ID: 0x00A1
  PAYLOAD:   {"temperature": 23.5, "humidity": 60, "pressure": 1013}
             (PSON map: 38 bytes)

Client -> Server: STREAM_DATA (compact)
  STREAM_ID: 0x00A1
  PAYLOAD:   [23.6, 61, 1013]
             (PSON array: 10 bytes -- 74% smaller)

Client -> Server: STREAM_DATA (compact)
  STREAM_ID: 0x00A1
  PAYLOAD:   [23.7, 62, 1014]
             (PSON array: 10 bytes)
```

11.4.7. Wire Efficiency

Compact mode is most effective when:

* The resource returns a map with many keys (more string bytes eliminated per sample).
* The stream has many samples (the schema overhead is amortized over all samples).
* Keys are long (e.g., "temperature", "humidity" vs. short keys like "t", "h").

Scenario	Normal (bytes/ sample)	Compact (bytes/sample)	Savings
2 sensors (temp + humidity)	35	14	60%
8 sensors (environmental station)	83	27	67%

Table 25

For 100 samples of 2 sensors:

Mode	Total bytes	vs MQTT 3.1.1	vs MQTT v5 (alias)
Normal	_(3,816)	40% savings	11% savings
Compact	_(1,421)	78% savings	67% savings

Table 26

11.4.8. Related Work

The problem of eliminating repeated keys in serialized data has been addressed by several existing standards and technologies. IOTMP's compact mode differs from these approaches in how schema negotiation is integrated into the streaming lifecycle.

CBOR Packed [RFC9543] defines a mechanism for reducing repetition in CBOR by using shared value tables. Frequently occurring strings or values are stored in a table and referenced by index. This is a general-purpose approach that works on any CBOR data but requires an explicit packing/unpacking step and does not integrate with a streaming protocol.

SenML [RFC8428] is a data format for sensor measurements that reduces repetition through base values and delta encoding (e.g., a base name shared across multiple records). SenML is widely used in CoAP and LwM2M but operates at the data model level, not the transport level, and each record still carries field identifiers.

Protocol Buffers [ProtocolBuffers] and Apache Avro use pre-shared schema files (.proto, .avsc) to eliminate field names entirely from the wire format. This achieves excellent compactness but requires the schema to be agreed upon at build time, making it unsuitable for dynamic IoT resources whose structure may vary across firmware versions or device types.

MQTT Sparkplug defines a metric registration mechanism over MQTT where metrics are registered with numeric aliases on first publish, and subsequent messages use aliases instead of full metric names. This is the closest existing approach to IOTMP's compact mode.

IOTMP's compact mode combines dynamic schema establishment (from the first STREAM_DATA message), recursive compaction (nested maps become nested arrays), and protocol-integrated negotiation (via START_STREAM parameters) into a single mechanism that requires no pre-shared schema files, no external registries, and no additional protocol messages beyond the standard streaming lifecycle.

11.5. STOP_STREAM Message

Field	Content
STREAM_ID	The stream identifier to stop.

Table 27

Either side MAY send STOP_STREAM to terminate an active stream. The stream initiator sends STOP_STREAM when it no longer needs the data. The resource owner sends STOP_STREAM when the resource is no longer available (e.g., a terminal session has ended or the device cannot sustain the stream).

Upon receiving STOP_STREAM:

1. The receiver removes the stream registration and compact schema state (if any).
2. The receiver clears the stream ID on the resource.
3. The receiver responds with OK.

11.6. Multiple Concurrent Streams

A client MUST support multiple simultaneous streams. Different resources MAY have different stream intervals and encoding modes (normal or compact). The Stream ID space is partitioned between client and server Section 6.2, providing 32,768 IDs per side -- ample for concurrent streams on a single connection.

11.7. Flow Control

IOTMP does not define an application-level flow control mechanism (such as credit-based windows). Flow control is delegated to the transport layer: TCP’s built-in backpressure naturally throttles the sender when the receiver cannot consume data fast enough.

If a device cannot accept a new stream due to resource constraints (memory, CPU, or maximum stream limit), it MUST respond to the START_STREAM request with ERROR (429). Similarly, if a device cannot execute a RUN request due to overload, it MAY respond with ERROR (429).

The server controls the data rate of each stream through the interval parameter in START_STREAM. Implementations SHOULD use appropriate intervals to avoid overwhelming constrained devices.

IOTMP does not define application-level delivery guarantees (such as QoS levels). Request-response exchanges (RUN, DESCRIBE, START_STREAM, STOP_STREAM) provide implicit delivery confirmation through OK/ERROR responses. Streaming data (STREAM_DATA) is ephemeral by nature -- each sample supersedes the previous one -- so retransmission of individual samples is unnecessary. Connection-level reliability is delegated to the transport layer (TCP/TLS).

12. Extensibility

12.1. Symmetric Resource Model

IOTMP is a symmetric protocol: both clients and servers MAY expose resources. After authentication, either side can invoke resources on the peer using RUN, DESCRIBE, START_STREAM, and STOP_STREAM messages. The resource namespace is defined by each implementation.

This symmetry enables diverse use cases without protocol-level changes:

* A client can expose sensor resources that the server reads or streams.

* A server can expose storage, notification, or inter-device communication resources that clients invoke via RUN.
* Both sides discover available resources through DESCRIBE.

12.2. Application Profiles

The specific resources exposed by a server or client are outside the scope of this specification. Implementations MAY define application profiles that specify well-known resource names, their expected I/O types, and PAYLOAD formats.

12.3. Bidirectional Stream Channels

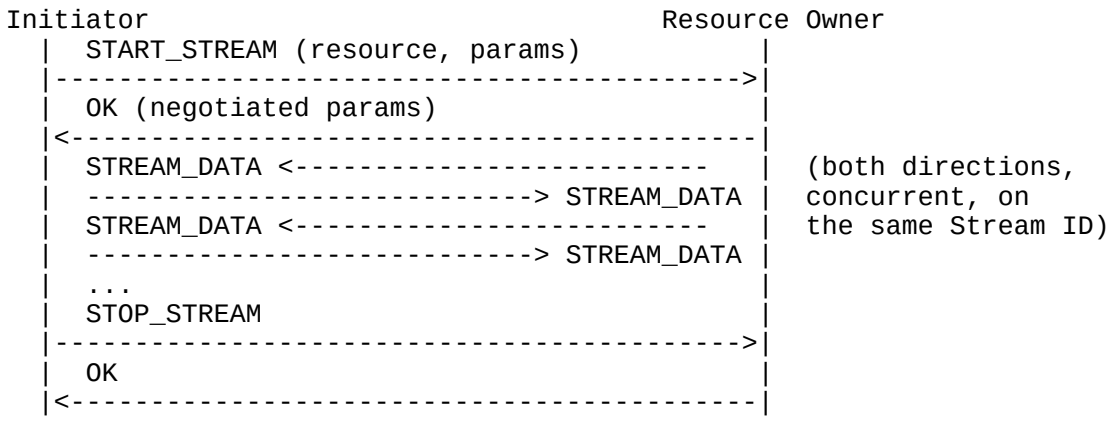
While Section 11 describes streaming primarily in the context of periodic telemetry (resource owner sends STREAM_DATA at regular intervals), IOTMP streams are general-purpose bidirectional data channels. Once a stream is active, *both sides* MAY send STREAM_DATA messages on the same Stream ID simultaneously. This enables a wide range of application patterns beyond periodic sampling, using only existing protocol primitives.

This bidirectional capability has been validated in production implementations supporting remote terminals, file transfers of multiple gigabytes, TCP proxy tunneling, and firmware updates -- all over the same IOTMP connection used for sensor telemetry, with no protocol modifications.

12.3.1. Channel Establishment

All bidirectional stream use cases follow the same lifecycle:

1. The stream initiator sends START_STREAM with a resource name and application-specific parameters.
2. The resource owner responds with OK (optionally confirming negotiated parameters).
3. Both sides exchange STREAM_DATA messages on the Stream ID for the duration of the session.
4. Either side terminates the channel with STOP_STREAM.



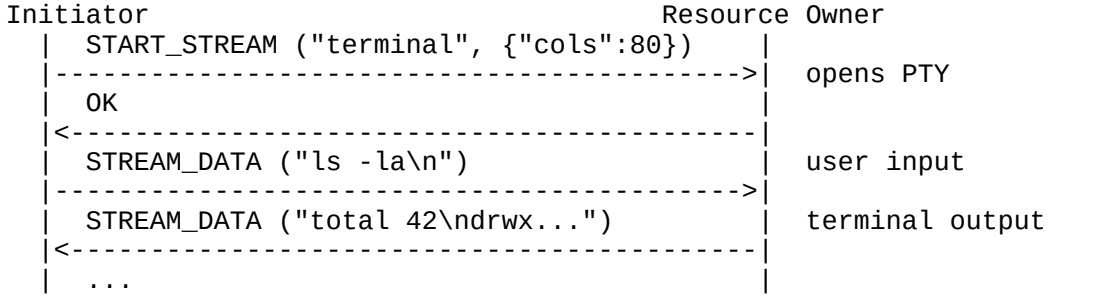
The content and semantics of the STREAM_DATA messages are defined by the application profile for each resource. The protocol layer treats them identically regardless of the use case.

12.3.2. Use Case: Interactive Sessions (Terminal, Proxy)

For interactive sessions, STREAM_DATA carries raw binary data in both directions simultaneously. This includes:

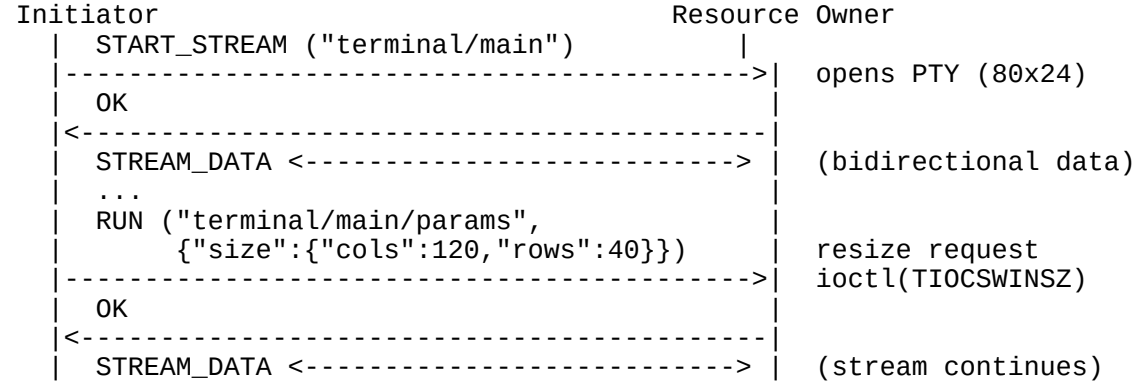
* *Remote terminal:* A PTY-backed shell session where user input flows as binary STREAM_DATA from initiator to resource owner, and terminal output flows back. START_STREAM parameters specify initial terminal dimensions (columns, rows).
* *TCP proxy tunneling:* A transparent relay where STREAM_DATA carries raw TCP bytes between the initiator and a local TCP endpoint on the device (e.g., a router admin panel, PLC interface, or database port). START_STREAM parameters specify the target host and port.

In both cases, data flows continuously in both directions with no application-level acknowledgment -- TCP backpressure provides sufficient flow control for interactive traffic.



Updating stream parameters at runtime:

Some interactive sessions require runtime parameter updates without interrupting the data flow (e.g., resizing a terminal window). Since START_STREAM cannot be re-sent on an active Stream ID, the recommended application-level pattern is to expose a companion resource that accepts parameter updates via RUN:



The companion resource (e.g., terminal/main/params) is a regular IOTMP resource with I/O type input that updates the session configuration. This pattern keeps the protocol simple -- no new message types or stream renegotiation mechanisms are needed -- while allowing applications to define whatever runtime parameters they require.

12.3.3. Use Case: Bulk Data Transfer (Files, Firmware)

For large data transfers (files, firmware images, logs), one side sends data chunks as binary STREAM_DATA and the other side sends acknowledgments as PSON maps in the opposite direction. This provides application-level flow control suitable for multi-gigabyte transfers.

Data flow:

* The *sender* transmits data as a sequence of STREAM_DATA messages, each carrying a PSON binary value within the negotiated message size limit.
* The *receiver* periodically sends ACK messages as STREAM_DATA in the reverse direction. Each ACK is a PSON map: {"ack": <sequence_number>, "bytes": <confirmed_bytes>}.
* The stream initiator sends STOP_STREAM when the transfer is complete.

Application-level flow control:

The recommended approach uses a byte-based sliding window:

* The stream initiator specifies a window_size in the START_STREAM parameters -- the maximum number of unacknowledged bytes allowed in flight.
* The sender tracks bytes_in_flight (bytes sent minus bytes acknowledged). When bytes_in_flight reaches window_size, the sender pauses until an ACK is received.
* The ACK frequency is adaptive: implementations SHOULD send ACKs approximately every 1% of total transfer size, bounded between one chunk size and 1 MB.

Recommended defaults:

Parameter	Default	Description
chunk_size	64 KB	Size of each binary STREAM_DATA payload. Confirmed by resource owner in OK response.
window_size	2 MB	Maximum unacknowledged bytes in flight.
ack_timeout	10 s	Timeout waiting for an ACK before aborting the transfer.

Table 28

Bandwidth limiting: Implementations MAY limit transfer rate by introducing controlled delays between chunks (sender-side) or between ACKs (receiver-side). Delaying ACKs naturally throttles the sender through the flow control window.

Small data optimization: For data that fits within a single message (less than or equal to the negotiated maximum message size), implementations SHOULD use an inline RUN request/response instead of opening a stream, avoiding the START_STREAM/STOP_STREAM overhead.

12.3.4. Use Case: Command Execution

For operations that are self-contained (short-lived, bounded output), a simple RUN request/response is sufficient. The client exposes a resource that accepts a command as input and returns the result (stdout, stderr, exit code) in the OK payload. No streaming is needed.

```
S -> C:  RUN ("cmd",
           {"exec":"uname","args":["-a"],"timeout":10})
C -> S:  OK  ({"stdout": "Linux device 5.15.0 ...",
           "stderr": "", "exit_code": 0})
```

12.3.5. Design Principle

These use cases illustrate a key IOTMP design property: the protocol provides a small set of general-purpose primitives (RUN for one-shot operations, START_STREAM/STREAM_DATA for persistent channels) that application profiles combine to address diverse requirements. No protocol-level changes are needed to support new use cases -- only the definition of new resource names and their data formats.

12.4. Example: Client Invoking a Server Resource

```
RUN Message:
  STREAM_ID:  0x1234
  RESOURCE:   "storage/sensor_data" (PSON string)
  PAYLOAD:    {"temperature": 25.3, "humidity": 60} (PSON map)

Response:
  OK
  STREAM_ID: 0x1234
```

12.5. Example: Server Invoking a Client Resource

```
RUN Message:
  STREAM_ID:  0x5679
  RESOURCE:   "led" (PSON string)
  PAYLOAD:    true (PSON boolean)

Response:
  OK
  STREAM_ID: 0x5679
```

13. Error Handling

13.1. ERROR Message

An ERROR message is sent in response to a failed request. It carries the same Stream ID as the request.

Field	Content
STREAM_ID	Mirrors the request Stream ID.
PARAMETERS	(OPTIONAL) Status code (varint).
PAYLOAD	(OPTIONAL) Error details (PSON-encoded).

Table 29

13.2. Status Codes

The PARAMETERS field in OK and ERROR messages MAY carry a numeric status code. Implementations SHOULD use HTTP status codes as defined in [RFC9110].

Rationale: IOTMP reuses HTTP status codes rather than defining a separate code space for two reasons. First, IOTMP's resource-oriented model (named resources with read/write/invoke operations) maps naturally to HTTP semantics, making HTTP status codes directly applicable: a missing resource is 404, a malformed request is 400, a timeout is 408. Second, IOTMP is designed to support transparent HTTP bridging, where an HTTP gateway translates between HTTP requests and IOTMP messages. Reusing HTTP status codes enables this bridge to pass status codes through without a translation table, preserving the original error semantics end-to-end.

If the PARAMETERS field is absent or zero, a generic success (for OK) or generic error (for ERROR) is assumed.

The following status codes are commonly used:

Code	Name	Usage
200	OK	Successful operation (implicit if omitted in OK).
301	Moved Permanently	Server redirect: client must update stored address Section 9.6.
307	Temporary Redirect	Server redirect: client should connect to indicated server Section 9.6.
400	Bad Request	Malformed request or invalid parameters.
401	Unauthorized	Authentication required or credentials invalid.
403	Forbidden	Insufficient permissions for the requested action.
404	Not Found	Resource does not exist.
408	Request Timeout	Operation timed out.
409	Conflict	Resource state conflict (e.g., already exists).
413	Content Too Large	Payload exceeds allowed size.
429	Too Many Requests	Rate limit exceeded.
500	Internal Server Error	Unexpected error during processing.

Table 30

Implementations MAY use other HTTP status codes as defined in [RFC9110]. Receivers that encounter an unrecognized status code SHOULD treat it according to the class of the code (2xx = success, 3xx = redirect, 4xx = client error, 5xx = server error).

13.3. Error Payload

When the PAYLOAD field is present in an ERROR message, it SHOULD be a PSON map containing an "error" key with a human-readable error description:

```
ERROR Message:
  STREAM_ID:  0x1234
  PARAMETERS: 404
  PAYLOAD:    {"error": "Resource 'sensor' does not exist"}
```

Implementations MAY include additional keys in the error payload for application-specific diagnostics.

13.4. OK Message with Status Code

An OK message MAY also carry a status code in the PARAMETERS field to provide more specific success information. If omitted, a generic 200 (OK) is assumed.

13.5. Connection-Level Errors

* *Transport failure:* If the transport connection is lost, all active streams are terminated. The client SHOULD attempt reconnection with exponential backoff.
* *Message too large:* If a received message exceeds the maximum size, the receiver MUST close the connection.
* *Invalid message type:* If a receiver encounters an unknown message type, it SHOULD ignore the message.
* *Decode failure:* If a message body cannot be decoded, the receiver MUST close the connection.

14. Security Considerations

This section follows the guidelines of [RFC3552] for security considerations.

14.1. Threat Model

IOTMP is designed for client-server communication between IoT devices and infrastructure servers (brokers). The following threat model identifies the primary attack surfaces and the protocol's defenses.

Actors:

* *Device (Client):* A constrained IoT device connecting to a broker over a potentially untrusted network.

* *Server (Broker):* An infrastructure server managing device connections, authentication, and resource routing.
* *External Attacker:* An entity with network access who can observe, inject, or modify traffic between the device and the server.
* *Compromised Device:* A legitimate device whose credentials or firmware have been compromised.

Threat Analysis:

Threat	Attack Vector	Impact	Mitigation
Eavesdropping	Passive observation of unencrypted traffic	Credential theft, data leakage	TLS encryption Section 14.2
Credential Replay	Attacker captures and replays a valid CONNECT message	Unauthorized access, device impersonation	TLS prevents replay; servers SHOULD reject duplicate CONNECT on same connection
Man-in-the-Middle	Attacker intercepts and modifies messages in transit	Data manipulation, command injection	TLS with server certificate validation; mTLS for mutual authentication
Device Impersonation	Attacker uses stolen or guessed credentials	Unauthorized data injection, control of resources	Strong unique credentials per device; rate limiting on authentication; credential rotation Section 14.3
Message Injection	Attacker injects crafted IOTMP messages into a connection	Unauthorized resource invocation, data corruption	TLS integrity protection; servers MUST validate Stream ID correlation
Message Tampering	Modification of messages in	Altered sensor data,	TLS message integrity

	transit	modified commands	
Denial of Service (Connection Flood)	Attacker opens many TCP connections without authenticating	Server resource exhaustion	CONNECT handshake timeout; per-IP connection limits Section 14.5
Denial of Service (Message Flood)	Compromised device sends excessive messages	Server CPU/ memory exhaustion	Rate limiting; maximum message size enforcement Section 14.5
Stream Exhaustion	Client opens maximum number of streams without closing them	Server memory exhaustion	Per-device stream limits; idle stream timeout Section 14.6
Malformed Payload	Crafted PSON data with extreme nesting, oversized lengths	Stack overflow, memory exhaustion on constrained devices	PSON validation; nesting depth limits; length bounds checking Section 14.6
Resource Enumeration	Attacker uses DESCRIBE to discover all device capabilities	Information disclosure; aids targeted attacks	Authorization on DESCRIBE; TLS to prevent passive discovery Section 14.8
Unauthorized Resource Access	Authenticated device invokes resources beyond its scope	Privilege escalation, unauthorized control	Per-resource authorization; least privilege principle Section 14.4

Table 31

14.2. Transport Security

IOTMP itself does not provide encryption or integrity protection at the application layer. Implementations MUST rely on TLS (version 1.2 or later, as specified in [RFC8446]) for:

* *Confidentiality:* Encryption of credentials and application data.
* *Server Authentication:* Verification of the server's identity via X.509 certificates.
* *Mutual Authentication:* (OPTIONAL) Verification of the client's identity via client certificates (mTLS).
* *Message Integrity:* Protection against message tampering and replay.

Unencrypted connections (port 25204) SHOULD only be used in isolated, trusted networks or during development. Production deployments MUST use TLS.

Implementations SHOULD support TLS 1.3 [RFC8446] for improved handshake performance (1-RTT vs 2-RTT) and stronger cipher suites. TLS 1.2 MAY be supported for compatibility with constrained devices that lack TLS 1.3 implementations.

14.3. Authentication

The CONNECT message transmits authentication data in the PAYLOAD field. The format of this data depends on the authentication type ("at" parameter, Section 9.2). For type 0 (Credentials), the payload contains namespace, device ID, and credential in cleartext. For type 1 (Token), the payload contains a bearer token. Without TLS, this data is vulnerable to eavesdropping.

Implementations MUST use TLS when transmitting over untrusted networks.

Implementations MUST:

* Reject CONNECT messages with empty, malformed, or missing authentication data.
* Respond with ERROR (401) when authentication credentials are invalid, expired, or do not match any registered device.
* Close the connection after a failed CONNECT response (ERROR). The client MUST NOT retry authentication on the same transport connection.
* Respond with ERROR (400) if the "at" value is not supported by the server.

Implementations SHOULD:

* Use strong, unique credentials per device (type 0) or short-lived tokens (type 1).
* Store credentials securely on the device (e.g., in a secure element or encrypted storage).
* Implement rate limiting on authentication attempts to mitigate brute-force attacks (RECOMMENDED: maximum 3 attempts per minute per source IP).
* Support credential rotation without requiring device firmware updates.
* Log failed authentication attempts for security monitoring.
* Prefer certificate-based authentication (type 2) or token-based authentication (type 1) over plain credentials (type 0) in production deployments.

The "at" parameter is extensible. Additional authentication types MAY be defined in separate specifications.

14.3.1. Certificate Authentication Security

When using authentication type 2 (Certificate), deployments may use either per-device certificates or fleet certificates (a single certificate shared across multiple devices within a namespace). Each approach has different security properties:

Per-device certificates: The certificate's subject (CN or SAN) identifies both the namespace and the device. The server extracts the full device identity from the certificate. If a PAYLOAD is present, the server MUST verify that the declared namespace and device_id match the certificate identity and MUST reject the connection with ERROR (401) on mismatch. This is the RECOMMENDED approach for production deployments.

Fleet certificates: The certificate identifies only the namespace (e.g., a shared CA per organizational scope). The device MUST send a PAYLOAD with [namespace, device_id] so the server can identify the specific device. In this model, any device holding the fleet certificate can claim any device_id within the namespace. Deployments using fleet certificates SHOULD be aware that compromise of a single device's certificate allows impersonation of any other device in the same namespace. To mitigate this risk, operators SHOULD:

* Limit the scope of fleet certificates to the smallest practical namespace.
* Implement server-side device registries that reject unknown device_ids even when the certificate is valid.
* Monitor for anomalous device_id claims (e.g., a single certificate claiming multiple device_ids in rapid succession).

* Prefer per-device certificates when the PKI infrastructure supports it.

14.4. Authorization

Authorization (which operations a device may perform) is outside the scope of this specification and is determined by the server implementation. However, implementations SHOULD:

* Follow the principle of least privilege: devices SHOULD only be authorized to access the resources they require.
* Enforce authorization on all message types, including DESCRIBE, RUN, START_STREAM, and STOP_STREAM.
* Support per-resource access control (e.g., device X may read resource Y but not write to it).
* Apply authorization checks on both client-to-server and server-to-client operations in symmetric mode.

14.5. Denial of Service

Connection-level mitigations:

* Servers SHOULD enforce a maximum time for the CONNECT handshake (RECOMMENDED: 10 seconds). If the client does not send a valid CONNECT message within this time, the server MUST close the connection.
* Servers SHOULD limit the number of concurrent connections per account or IP address (RECOMMENDED: maximum 100 connections per IP).
* Servers SHOULD implement connection rate limiting to prevent SYN flood attacks (RECOMMENDED: maximum 10 new connections per second per IP).
* Servers SHOULD close idle connections that have not sent any message within twice the negotiated keepalive interval.

Message-level mitigations:

* Implementations MUST enforce the negotiated maximum message size. A receiver that encounters a message exceeding its maximum MUST close the connection.
* Servers SHOULD implement per-device message rate limiting (RECOMMENDED: configurable, default 100 messages per second).
* Implementations MUST validate the Message Type field. Unknown message types SHOULD be ignored; malformed framing (e.g., varint that does not terminate) MUST cause connection closure.

14.6. Resource Exhaustion

* Servers SHOULD limit the number of concurrent streams per device (RECOMMENDED: maximum 256 active streams).
* Servers SHOULD implement idle stream detection and close streams that have not produced data within a configurable timeout.
* Implementations SHOULD validate PSON data structures to prevent deeply nested or excessively large payloads from consuming excessive processing time or memory. A maximum nesting depth of 16 levels is RECOMMENDED for constrained devices.
* Implementations MUST validate that varint encoding terminates within 4 bytes (representing values up to 2^28 - 1). A varint that exceeds this limit MUST be treated as a decode error and the connection MUST be closed.

14.7. Secure Credential Storage

Devices operating in physically accessible environments face the risk of credential extraction through hardware attacks (JTAG, flash dumping, side-channel analysis). Implementations SHOULD:

* Store credentials in hardware secure elements or trusted execution environments when available.
* Use derived keys rather than storing master credentials on devices.
* Support remote credential revocation so that compromised devices can be disabled without physical access.

14.8. Privacy

* Device identifiers, resource names, and DESCRIBE responses are visible to any entity with access to the network traffic unless TLS is used. TLS MUST be used when privacy of device metadata is required.
* Implementations SHOULD minimize the exposure of device metadata in resource names and DESCRIBE responses when operating over untrusted networks.
* The DESCRIBE message can reveal the full API surface of a device, including resource names, I/O types, data schemas, and human-readable descriptions. Servers MUST enforce authorization on DESCRIBE requests and SHOULD allow operators to restrict which clients can perform API discovery.
* Servers SHOULD NOT log secrets (credentials, tokens). For type 0 (Credentials), implementations SHOULD log only the namespace and device ID for audit purposes.

15. Conformance Requirements

15.1. Client Conformance

A conformant IOTMP client implementation MUST:

* Support TCP and TLS transports.
* Send a CONNECT message as the first message after transport establishment.
* Support all message types defined in Section 6.
* Correctly encode and decode PSON values as defined in [PSON].
* Support at least one concurrent stream.
* Implement keepalive as defined in Section 9.4.
* Implement reconnection with exponential backoff as defined in Section 9.7.
* Support messages up to at least 32,768 bytes (or the negotiated maximum).
* Include a Stream ID in all request messages (CONNECT, RUN, DESCRIBE, START_STREAM, STOP_STREAM).
* Use even Stream IDs for all client-initiated requests Section 6.2.
* Not reuse a Stream ID that is currently active.
* Close the connection upon receiving an ERROR response to CONNECT.

A conformant client SHOULD:

* Support WebSocket transport with the "iotmp" subprotocol.
* Support multiple concurrent streams.
* Support compact streaming mode Section 11.4.
* Support DESCRIBE with [JSON-Schema] Section 10.4.

15.2. Server Conformance

A conformant IOTMP server implementation MUST:

* Support TCP and TLS transports.
* Process CONNECT messages and respond with OK or ERROR.
* Support all message types defined in Section 6.
* Correctly encode and decode PSON values as defined in [PSON].
* Respect the client’s declared maximum message size ("ms" parameter).
* Use odd Stream IDs for all server-initiated requests Section 6.2.
* Validate that client-initiated requests use even Stream IDs.
* Echo KEEP_ALIVE messages as defined in Section 9.4.
* Implement keepalive timeout detection.
* Enforce the CONNECT handshake timeout Section 14.5.
* Enforce maximum message size Section 14.5.
* Validate the Message Type field and close the connection on malformed framing.
* Enforce authorization on all resource operations.

A conformant server SHOULD:

* Support WebSocket transport with the "iotmp" subprotocol.
* Support server redirect Section 9.6.
* Include status codes in OK and ERROR responses Section 13.2.
* Support compact streaming mode negotiation Section 11.4.
* Validate incoming data against [JSON-Schema] when available from DESCRIBE Section 10.4.

15.3. Connection Lifecycle Conformance

The following rules govern connection state transitions and edge cases:

Stream Recovery on Reconnect:

When a transport connection is lost, all active streams are terminated. Stream state (Stream IDs, compact mode schemas) is NOT preserved across connections. After reconnection:

1. The client MUST send a new CONNECT message.
2. The server MUST re-establish any desired streams by sending new START_STREAM messages.
3. Both sides MUST NOT reference Stream IDs from the previous connection.

In-Flight Message Handling:

* If a request message (RUN, DESCRIBE, START_STREAM, STOP_STREAM) has been sent but no response received when the connection drops, the request is considered failed. The client SHOULD NOT assume the operation was executed.
* If a response (OK, ERROR) is lost due to connection failure, the server MAY have already executed the operation. The client MAY safely re-send the request after reconnection, provided the operation is idempotent.

Idempotency Guidance:

IOTMP's request-response model (RUN -> OK/ERROR) provides implicit delivery confirmation for the common case: if the client receives OK, the operation succeeded; if it receives ERROR, it failed. The only ambiguous scenario occurs when the connection drops after the server executes the operation but before the client receives the response.

IOTMP does not define protocol-level delivery guarantees (such as QoS levels) because its request-response model already covers the vast majority of cases. For the rare connection-loss scenario, the recommended approach is to design resource operations as *absolute state transitions* rather than relative or toggling operations:

Pattern	Example	Safe to retry?
Absolute state (RECOMMENDED)	{"relay": true}	Yes -- setting the same state twice has no additional effect.
Relative/ toggle (AVOID)	{"action": "toggle"}	No -- retrying may reverse the intended state.

Table 32

Resources that follow this pattern are inherently idempotent: re-executing the same RUN after reconnection produces the same result whether the original request was executed or not. This eliminates the need for protocol-level deduplication mechanisms.

Additionally, if the client needs to verify the outcome of an ambiguous request after reconnection, it can query the current resource state by sending a RUN (for output resources) or DESCRIBE to the resource. This allows the client to confirm whether the previous operation took effect before deciding whether to retry.

For the rare cases where non-idempotent operations are unavoidable, application profiles MAY implement their own deduplication (e.g., including a unique request identifier in the PAYLOAD that the receiver checks against recently processed requests).

Invalid State Handling:

Condition	Required Behavior
Client sends a message before CONNECT	Server MUST close the connection.
Client sends CONNECT after authentication	Server MUST respond with ERROR (400) and close the connection.
Message received with unknown message type	Receiver SHOULD ignore the message.
Message received with unknown field number	Receiver MUST ignore the unknown field and process known fields.
STREAM_DATA received for unknown Stream ID	Receiver SHOULD ignore the message.
RUN/DESCRIBE received for non-existent resource	Receiver MUST respond with ERROR (404).
Message exceeds negotiated maximum size	Receiver MUST close the connection.
Varint does not terminate within 4 bytes	Receiver MUST close the connection.
Keepalive timeout exceeded	Server MUST close the connection.
Request with Stream ID from wrong partition Section 6.2	Receiver MUST respond with ERROR (400).
Request with a Stream ID that is already active	Receiver MUST respond with ERROR (409).

Table 33

15.4. Conformance Test Vectors

The following test vectors allow implementations to verify correct encoding and decoding. Each vector specifies the input, the expected wire encoding (hex), and the total byte count.

15.4.1. KEEP_ALIVE

Input: KEEP_ALIVE message (empty body).

Expected encoding: 05 00
Total: 2 bytes

* 05: Message type KEEP_ALIVE (varint 5).
* 00: Body size 0 (varint 0).

15.4.2. CONNECT

Input: CONNECT (auth type 0) with credentials ["acme1", "device1", "secret123"] (namespace, device_id, credential), Stream ID = 42 (client-initiated, even).

Expected encoding:
03 1C 08 2A 1A E3 85 61 63 6D 65 31 87 64 65 76
69 63 65 31 89 73 65 63 72 65 74 31 32 33
Total: 30 bytes

Decoding verification:

* 03: Message type CONNECT.
* 1C: Body size 28.
* 08: Field tag (STREAM_ID, varint). 2A: Stream ID = 42.
* 1A: Field tag (PAYLOAD, pson).
* E3: PSON array, 3 elements.
* 85 61 63 6D 65 31: PSON string "acme1" (namespace, 5 bytes).
* 87 64 65 76 69 63 65 31: PSON string "device1" (device_id, 7 bytes).
* 89 73 65 63 72 65 74 31 32 33: PSON string "secret123" (credential, 9 bytes).

15.4.3. OK Response

Input: OK response to Stream ID 42, no status code, no payload.

Expected encoding: 01 02 08 2A
Total: 4 bytes

* 01: Message type OK.
* 02: Body size 2.
* 08: Field tag (STREAM_ID, varint). 2A: Stream ID = 42.

15.4.4. RUN with Resource Name

Input: RUN resource "led" with payload {"on": true}, Stream ID = 100 (client-initiated, even).

Expected encoding:
06 0D 08 64 22 83 6C 65 64 1A C1 82 6F 6E 61
Total: 15 bytes

Decoding verification:

* 06: Message type RUN.
* 0D: Body size 13.
* 08 64: STREAM_ID = 100.
* 22 83 6C 65 64: RESOURCE = PSON string "led" (3 bytes).
* 1A C1 82 6F 6E 61: PAYLOAD = PSON map {1 entry: "on" -> true}.

15.4.5. RUN with Resource Hash

Input: RUN resource hash 0x1A2B (FNV-1a hash of a resource name), no payload, Stream ID = 7 (server-initiated, odd).

Expected encoding:
06 05 08 07 20 AB 34
Total: 7 bytes

Decoding verification:

* 06: Message type RUN.
* 05: Body size 5.
* 08 07: STREAM_ID = 7.
* 20 AB 34: RESOURCE = varint 6699 (0x1A2B).

15.4.6. ERROR with Status Code and Payload

Input: ERROR for Stream ID 42, status 404, payload {"error": "Not found"}.

Expected encoding:
02 17 08 2A 10 94 03 1A C1 85 65 72 72 6F 72 89
4E 6F 74 20 66 6F 75 6E 64
Total: 25 bytes

Decoding verification:

* 02: Message type ERROR.
* 17: Body size 23.
* 08 2A: STREAM_ID = 42.

* 10 94 03: PARAMETERS (varint) = 404.
* 1A C1 85 65 72 72 6F 72 89 4E 6F 74 20 66 6F 75 6E 64: PAYLOAD = PSON map {"error": "Not found"}.

15.4.7. START_STREAM with Compact Mode

Input: START_STREAM resource "temperature", interval 5000ms, compact mode, Stream ID = 161 (server-initiated, odd).

Expected encoding:
08 1B 08 A1 01 12 C2 81 69 1F 88 27 82 63 6D 61
22 8B 74 65 6D 70 65 72 61 74 75 72 65
Total: 29 bytes

Decoding verification:

* 08: Message type START_STREAM.
* 1B: Body size 27.
* 08 A1 01: STREAM_ID = 161 (varint).
* 12 C2 81 69 1F 88 27 82 63 6D 61: PARAMETERS (pson map) = {"i": 5000, "cm": true}.
* 22 8B 74 65 6D 70 65 72 61 74 75 72 65: RESOURCE = PSON string "temperature" (11 bytes).

16. IANA Considerations

16.1. Port Number Registration

This specification requests the assignment of the following TCP port numbers from IANA’s Service Name and Transport Protocol Port Number Registry:

Service Name	Port	Transport	Description	Reference
iotmp	25204	TCP	IOTMP over TCP (unencrypted)	This document
iotmps	25206	TCP	IOTMP over TLS	This document

Table 34

Registration details:

* *Service Name:* iotmp / iotmps
* *Transport Protocol:* TCP

* *Assignee:* Alvaro Luis Bustamante / Internet of Thinger SL
* *Contact:* alvaro@thinger.io
* *Description:* Internet of Things Message Protocol -- a binary application-layer protocol for bidirectional communication between IoT devices and servers.
* *Reference:* This document
* *Assignment Notes:* Port 25204 is for unencrypted connections (development and trusted networks only). Port 25206 is for TLS-encrypted connections (production use). Implementations SHOULD default to port 25206 (TLS).

16.2. WebSocket Subprotocol Registration

This specification requests registration of the following entry in the IANA WebSocket Subprotocol Name Registry, as defined in [RFC6455], Section 11.5:

* *Subprotocol Identifier:* iotmp
* *Subprotocol Common Name:* Internet of Things Message Protocol
* *Subprotocol Definition:* This document
* *Reference:* This document

When operating over WebSocket, the client MUST include "iotmp" in the Sec-WebSocket-Protocol header during the handshake. The server MUST confirm the subprotocol in the response. All WebSocket messages MUST use binary opcode (0x02) and each frame MUST contain exactly one complete IOTMP message Section 4.3.

16.3. Message Type Registry

IANA is requested to create a new registry entitled "IOTMP Message Types" in a new "Internet of Things Message Protocol (IOTMP)" registry group. The registry contains the following columns: Value, Name, and Reference.

New registrations in the range 0x0B-0xFF require Standards Action [RFC8126].

Initial values:

Value	Name	Reference
0x00	RESERVED	This document
0x01	OK	This document
0x02	ERROR	This document
0x03	CONNECT	This document
0x04	DISCONNECT	This document
0x05	KEEP_ALIVE	This document
0x06	RUN	This document
0x07	DESCRIBE	This document
0x08	START_STREAM	This document
0x09	STOP_STREAM	This document
0x0A	STREAM_DATA	This document

Table 35

16.4. Authentication Type Registry

IANA is requested to create a new registry entitled "IOTMP Authentication Types" in the "Internet of Things Message Protocol (IOTMP)" registry group. The registry contains the following columns: Code, Name, PAYLOAD Format, and Reference.

New registrations in the range 3-255 require Specification Required [RFC8126].

Initial values:

Code	Name	PAYLOAD Format	Reference
0	Credentials	PSON array: [namespace, device_id, credential]	This document
1	Token	PSON string (bearer token)	This document
2	Certificate	Optional PSON array: [namespace, device_id] or absent	This document

Table 36

16.5. Field Number Registry

IANA is requested to create a new registry entitled "IOTMP Field Numbers" in the "Internet of Things Message Protocol (IOTMP)" registry group. The registry contains the following columns: Number, Name, Allowed Wire Types, and Reference.

New registrations in the range 0x05-0x07 require Standards Action [RFC8126].

Initial values:

Number	Name	Allowed Wire Types	Reference
0x00	RESERVED	--	This document
0x01	STREAM_ID	varint	This document
0x02	PARAMETERS	varint, pson	This document
0x03	PAYLOAD	pson, bytes	This document
0x04	RESOURCE	varint, pson	This document

Table 37

16.6. Wire Type Registry

IANA is requested to create a new registry entitled "IOTMP Wire Types" in the "Internet of Things Message Protocol (IOTMP)" registry group. The registry contains the following columns: Value, Name, Description, and Reference.

New registrations in the range 0x03-0x07 require Standards Action [RFC8126].

Initial values:

Value	Name	Description	Reference
0x00	varint	Variable-length unsigned integer	This document
0x01	bytes	Length-prefixed raw bytes	This document
0x02	pson	PSON-encoded value	This document

Table 38

16.7. Media Type

The application/pson media type is defined in [PSON].

17. References

17.1. Normative References

[IEEE754] IEEE, "IEEE Standard for Floating-Point Arithmetic", IEEE 754-2019, 2019.

[JSON-Schema] Wright, A., Andrews, H., Hutton, B., and G. Dennis, "JSON Schema: A Media Type for Describing JSON Documents", Work in Progress, Internet-Draft, draft-bhutton-json-schema-01, June 2022, <https://json-schema.org/specification>.

[PSON] Bustamante, A.L., "PSON: Packed Sensor Object Notation", Work in Progress, Internet-Draft, draft-bustamante-pson-00, March 2026, <https://datatracker.ietf.org/doc/html/draft-bustamante-pson-00>.

[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>.

[RFC6455] Fette, I. and A. Melnikov, "The WebSocket Protocol", RFC 6455, DOI 10.17487/RFC6455, December 2011, <https://www.rfc-editor.org/info/rfc6455>.

[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>.

[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/info/rfc8446>.

[RFC9110] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/RFC9110, June 2022, <https://www.rfc-editor.org/info/rfc9110>.

17.2. Informative References

[LwM2M] Open Mobile Alliance, "Lightweight Machine to Machine Technical Specification", OMA TS-LightweightM2M_Core-V1_2.

[MQTT] OASIS, "MQTT Version 5.0", OASIS Standard, March 2019, <https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html>.

[ProtocolBuffers] Google, "Protocol Buffers Encoding", <https://protobuf.dev/programming-guides/encoding/>.

[RFC3552] Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", BCP 72, RFC 3552, DOI 10.17487/RFC3552, July 2003, <https://www.rfc-editor.org/info/rfc3552>.

[RFC7252] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained Application Protocol (CoAP)", RFC 7252, DOI 10.17487/RFC7252, June 2014, <https://www.rfc-editor.org/info/rfc7252>.

[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/info/rfc8126>.

[RFC8428] Jennings, C., Shelby, Z., Arkko, J., Keranen, A., and C. Bormann, "Sensor Measurement Lists (SenML)", RFC 8428, DOI 10.17487/RFC8428, August 2018, <https://www.rfc-editor.org/info/rfc8428>.

[RFC8610] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610, June 2019, <https://www.rfc-editor.org/info/rfc8610>.

[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/info/rfc8949>.

[RFC9543] Farrel, A., Ed., Drake, J., Ed., Rokui, R., Homma, S., Makhijani, K., Contreras, L., and J. Tantsura, "A Framework for Network Slices in Networks Built from IETF Technologies", RFC 9543, DOI 10.17487/RFC9543, March 2024, <https://www.rfc-editor.org/info/rfc9543>.

Appendix A. Wire Format Examples

A.1. KEEP_ALIVE Message

```
05 00
|  +- Body Size: 0 (varint)
+---- Message Type: 0x05 = KEEP_ALIVE (varint)
```

Total: *2 bytes*.

A.2. CONNECT Message

For auth type 0 credentials ["acme1", "device1", "secret123"] (namespace, device_id, credential) with Stream ID 42:

```
03                          # Message Type: CONNECT (varint)
1C                          # Body Size: 28 (varint)
08                          # STREAM_ID (field=1, wire=varint)
2A                          # Stream ID: 42 (varint)
1A                          # PAYLOAD (field=3, wire=pson)
E3                          # PSON: array of 3 elements
  85 61 63 6D 65 31         # string "acme1" (namespace, len=5)
  87 64 65 76 69 63 65 31   # string "device1" (device_id, len=7)
                            # PSON: string "secret123"
  89 73 65 63 72 65 74 31 32 33  # (credential, len=9)
```

Total: *30 bytes*.

A.3. RUN Message (Resource Invocation)

Peer asks for resource "temperature":

```
06                          # Message Type: RUN (varint)
0F                          # Body Size: 15 (varint)
08                          # STREAM_ID (field=1, wire=varint)
2A                          # Stream ID: 42
22                          # RESOURCE (field=4, wire=pson)
                            # PSON: string "temperature" (len=11)
8B 74 65 6D 70 65 72 61 74 75 72 65
```

Total: *17 bytes*.

A.4. OK Response with Payload

Response with {"temperature": 25.3}:

```
01                          # Message Type: OK (varint)
15                          # Body Size: 21 (varint)
08                          # STREAM_ID (field=1, wire=varint)
2A                          # Stream ID: 42
1A                          # PAYLOAD (field=3, wire=pson)
C1                          # PSON: map with 1 entry
                            # key: "temperature" (len=11)
  8B 74 65 6D 70 65 72 61 74 75 72 65
                            # value: float 25.3 (IEEE 754, LE)
  40 66 66 CA 41
```

Total: *23 bytes*.

A.5. ERROR Response with Status Code

Response with 404 status and error message:

```
   02                          # Message Type: ERROR (varint)
   20                          # Body Size: 32 (varint)
   08                          # STREAM_ID (field=1, wire=varint)
   2A                          # Stream ID: 42
   10                          # PARAMETERS (field=2, wire=varint)
   94 03                       # Status code: 404 (varint)
   1A                          # PAYLOAD (field=3, wire=pson)
   C1                          # PSON: map with 1 entry
     85 65 72 72 6F 72         # key: "error" (len=5)
                               # value: "Resource not found" (len=18)
     92 52 65 73 6F 75 72 63 65 20 6E 6F 74 20
     66 6F 75 6E 64
```

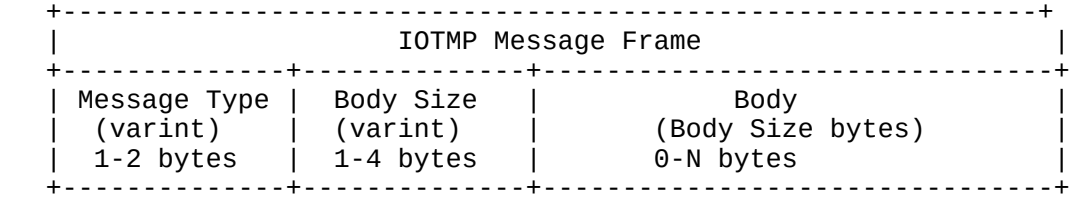
Total: *34 bytes*.

Appendix B. Formal Message Grammar (CDDL)

This appendix provides a formal definition of IOTMP messages using CDDL (Concise Data Definition Language, [RFC8610]). This grammar is normative and defines the *logical structure* of all IOTMP messages -- the required fields, their types, and their constraints.

Note on scope: CDDL is designed for CBOR-based protocols, but is used here as a structural description language. The CDDL definitions describe the semantic structure of each message (which fields exist, their types, and cardinality), NOT the wire encoding. The actual wire format uses IOTMP's own framing Section 5, field tag encoding Section 7, and PSON data encoding Section 8, which differ from CBOR. Implementations MUST follow the wire format defined in Section 5, Section 7, and Section 8; the CDDL grammar serves as a complementary formal reference for message structure validation.

B.1. Message Frame



B.2. Field Tag Encoding

```
 +--------------------------------------+
 |          Field Tag (1 byte)          |
 +------------------+-------------------+
 |  Field Number    |   Wire Type       |
 |  (bits 7-3)      |   (bits 2-0)      |
 |  5 bits (0-31)   |   3 bits (0-7)    |
 +------------------+-------------------+

 Wire Type values:
   0x00 = varint   (variable-length unsigned integer)
   0x01 = bytes    (length-prefixed raw bytes)
   0x02 = pson     (PSON-encoded value)

 Defined Field Tags:
   0x08 = STREAM_ID   (field 1, wire type varint)
   0x10 = PARAMETERS  (field 2, wire type varint)
   0x12 = PARAMETERS  (field 2, wire type pson)
   0x1A = PAYLOAD     (field 3, wire type pson)
   0x19 = PAYLOAD     (field 3, wire type bytes)
   0x20 = RESOURCE    (field 4, wire type varint -- resource hash)
   0x22 = RESOURCE    (field 4, wire type pson -- resource name or hash)
```

B.3. Varint Encoding

```
   Single-byte varint (values 0-127):
   +---+-----------+
   | 0 |  value    |
   |   | (7 bits)  |
   +---+-----------+

   Multi-byte varint (values >= 128):
   +---+-----------+ +---+-----------+     +---+-----------+
   | 1 | bits 0-6  | | 1 | bits 7-13 | ... | 0 | bits N-N+6|
   +---+-----------+ +---+-----------+     +---+-----------+
    MSB = 1: more      MSB = 1: more        MSB = 0: last
    bytes follow        bytes follow         byte
```

B.4. CDDL Definitions

```
   ; ============================================================
   ; IOTMP Message Grammar -- CDDL (RFC 8610)
   ; ============================================================

   ; --- Top-level frame ---

   iotmp-frame = (
     message-type: message-type-id,
     body-size:    uint,
```

```
    body:         bstr .size body-size,
  )

  message-type-id = &(
    RESERVED:     0x00,
    OK:           0x01,
    ERROR:        0x02,
    CONNECT:      0x03,
    DISCONNECT:   0x04,
    KEEP_ALIVE:   0x05,
    RUN:          0x06,
    DESCRIBE:     0x07,
    START_STREAM: 0x08,
    STOP_STREAM:  0x09,
    STREAM_DATA:  0x0A,
  )

  ; --- Field definitions ---

  stream-id   = uint .size 2     ; 0-65535 (even=client, odd=server)
  status-code = uint              ; HTTP status code (RFC 9110)
  resource-id = tstr / uint .size 2
                                  ; string name or 16-bit FNV-1a hash

  ; --- Per-message field requirements ---

  ; CONNECT (0x03): Client -> Server
  connect-body = {
    stream_id:   stream-id,
    ? payload:   connect-payload,
    ? parameters: {
      ? "v":  uint .default 1,          ; protocol version
      ? "ka": uint .le 1800 .default 60, ; keepalive (seconds)
      ? "at": auth-type .default 0,     ; authentication type
      ? "ms": uint .ge 1024 .default 32768,  ; max message size (bytes)
    },
  }

  auth-type = &(
    credentials: 0,  ; PAYLOAD = [ns, device_id, cred]
    token:       1,  ; PAYLOAD = bearer token string
    certificate: 2,  ; PAYLOAD optional: [ns, device_id]
                     ; or absent
  )

  connect-payload = credentials-payload
                  / token-payload
                  / certificate-payload
```

```
credentials-payload = [
  namespace: tstr, device_id: tstr, credential: tstr
]
token-payload       = tstr      ; bearer token (JWT, API key, etc.)
certificate-payload = [namespace: tstr, device_id: tstr]
                    / nil / empty

; OK (0x01): Response to any request
ok-body = {
  stream_id:    stream-id,
  ? parameters: status-code / pson-value,
                ; status code or structured data
  ? payload:    pson-value,
}

; ERROR (0x02): Error response to any request
error-body = {
  stream_id:    stream-id,
  ? parameters: status-code,
  ? payload: { "error": tstr, * tstr => any },
             ; error message + optional diagnostics
}

; DISCONNECT (0x04)
disconnect-body = {
  ? parameters: status-code,    ; redirect code (301, 307)
  ? payload:    pson-value,          ; redirect target or reason
}

; KEEP_ALIVE (0x05)
keep-alive-body = empty             ; body MUST be empty (size = 0)

; RUN (0x06): Execute a resource
run-body = {
  stream_id:  stream-id,
  resource:   resource-id,
  ? payload:  pson-value,            ; input data for the resource
}

; DESCRIBE (0x07): Request resource metadata
describe-body = {
  stream_id:  stream-id,
  ? resource: resource-id,
              ; absent = full API, present = single resource
}

; START_STREAM (0x08): Begin streaming
start-stream-body = {
```

```
    stream_id:    stream-id,
    resource:     resource-id,
    ? parameters: uint / {        ; interval or structured params
      ? "i":  uint,              ; interval in ms (0 = event-driven)
      ? "cm": bool .default false,     ; compact mode request
    },
  }

  ; STOP_STREAM (0x09): End streaming
  stop-stream-body = {
    stream_id: stream-id,
  }

  ; STREAM_DATA (0x0A): Stream payload
  stream-data-body = {
    stream_id: stream-id,
    payload:   pson-value,   ; resource data (map or compact arr)
  }

  ; --- DESCRIBE response structures ---

  ; Full API DESCRIBE response (no RESOURCE in request)
  full-api-describe-response = {
    "v": uint,                          ; description format version
    "res": {               ; resource map (not protocol fields)
      * resource-name => {
        "fn": io-type-code,             ; I/O type code
        ? "description": tstr,          ; human-readable description
      },
    },
  }

  resource-name = tstr
  io-type-code = &(
    none:         0,
    run:          1,
    input:        2,
    output:       3,
    input_output: 4,
  )

  ; Single Resource DESCRIBE response (RESOURCE present in request)
  single-resource-describe-response = {
    "v":   uint,
    ? "in":  io-descriptor,
    ? "out": io-descriptor,
  }
```

```
io-descriptor = {
  "value":   pson-value,             ; current/sample data
  ? "schema": json-schema,     ; JSON Schema definition (optional)
}

json-schema = {                      ; subset of JSON Schema keywords
  ? "type":        schema-type,
  ? "properties":  { * tstr => json-schema },
  ? "items":       json-schema,
  ? "description": tstr,
  ? "minimum":     number,
  ? "maximum":     number,
  ? "enum":        [+ any],
  ? "readOnly":    bool,
  ? "writeOnly":   bool,
  ? "default":     any,
  ? "required":    [+ tstr],
}

schema-type = "boolean" / "integer" / "number"
            / "string" / "object" / "array"

; --- Generic types ---

pson-value = any          ; any PSON-encoded value (see PSON)
empty = nil                          ; zero-length body
```

B.5. Field Requirements per Message Type

The following table summarizes which fields are required (R), optional (O), conditional (C), or not used (--) for each message type:

Message Type	STREAM_ID	PARAMETERS	PAYLOAD	RESOURCE
OK	R	O	O	--
ERROR	R	O	O	--
CONNECT	R	O	C	--
DISCONNECT	--	O	O	--
KEEP_ALIVE	--	--	--	--
RUN	R	--	O	R
DESCRIBE	R	--	--	O
START_STREAM	R	O	--	R
STOP_STREAM	R	--	--	--
STREAM_DATA	R	--	R	--

Table 39

C = Conditional: CONNECT PAYLOAD is required for authentication types 0 (Credentials) and 1 (Token). For type 2 (Certificate), PAYLOAD is optional: required when the certificate identifies only the namespace, and may be absent when the certificate identifies both namespace and device.

Appendix C. PSON vs JSON Size Comparison

For a detailed size comparison between PSON and JSON for typical IoT payloads, see Section 10 of [PSON]. In summary, PSON achieves 40-75% size reduction for typical IoT data patterns.

Appendix D. Protocol Comparisons

Detailed protocol comparisons between IOTMP and other IoT protocols [MQTT], [RFC7252], [LwM2M] are available as separate companion documents.

Appendix E. Revision History

Version	Date	Changes
0.1	2026-03-30	Initial public draft.

Table 40

Author’s Address

Alvaro Luis Bustamante
Internet of Thinger SL
Spain
Email: alvaro@thinger.io