

Independent Submission
Internet-Draft
Intended status: Experimental
Expires: 2 October 2026

M. Baker
Independent
31 March 2026

Jason Transfer Protocol (JTP)
draft-baker-jtp-00

Abstract

This document specifies the Jason Transfer Protocol (JTP), a compact binary protocol for listing and transferring images over a reliable ordered byte stream. JTP is designed to be simple to implement and efficient to parse. Images are addressed by a content-derived 64-bit identifier computed using xxHash64. The protocol supports catalog enumeration, point retrieval by identifier, delta synchronization, and connection reuse via a keep-alive mechanism. Transport security may be provided by TLS with an ALPN protocol identifier of "jtp/1".

This document specifies the on-wire format of JTP version 1. It does not specify any particular implementation.

Discussion Venues

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

Source for this draft and an issue tracker can be found at [JTP-REPO].

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 2 October 2026.

Copyright Notice

Copyright (c) 2026 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document.

Table of Contents

1. Introduction	3
1.1. Requirements Language	4
1.2. Terminology	4
2. Protocol Overview	4
2.1. Connection Reuse (Keep-Alive)	5
3. Transport	5
3.1. TLS and Application-Layer Protocol Negotiation	6
4. Data Types	6
4.1. Unsigned Integers: u8, u16, u32, u64	6
4.2. Variable-Length Integer: varint(u32)	6
4.3. UTF-8 Strings	7
5. Identifiers	7
5.1. ImageID	7
6. Flags Field	7
6.1. File Type Codes	8
6.2. Compression (Bit 3)	9
6.3. Encryption (Bit 4)	9
6.4. Reserved Bits (Bits 5-7)	9
7. Requests	9
7.1. RequestFlags	9
7.2. LIST Request (ReqType = 1)	10
7.3. GET_BY_ID Request (ReqType = 0)	10
7.4. BATCH Request (ReqType = 2)	11
7.5. LIST_AND_GET Request (ReqType = 5)	11
8. Responses	12
8.1. LIST Response	12
8.2. Image Packet	13
8.3. BATCH Response	14
8.4. LIST_AND_GET Response	15
9. Error Handling	15
9.1. Structured ERROR Response	15
9.2. Error Codes	16
9.3. Legacy Error Signaling	16
10. Limits and Resource Considerations	16
11. Extensibility	17

12. Security Considerations	18
12.1. Transport Security	18
12.2. Content Integrity	18
12.3. Denial of Service	18
12.4. Filename Safety	18
12.5. Certificate Validation	19
13. IANA Considerations	19
14. References	19
14.1. Normative References	19
14.2. Informative References	20
Appendix A. Varint Encoding Example	20
Appendix B. Wire Format Summary	20
B.1. Request Formats	21
B.2. Response Formats	21
Author's Address	22

1. Introduction

The Jason Transfer Protocol (JTP) is a request/response binary protocol intended for efficient enumeration and retrieval of image files over a reliable, ordered byte stream such as TCP. JTP addresses images by a content-derived 64-bit hash (xxHash64), enabling content integrity checks and delta synchronization without additional metadata exchange.

JTP is motivated by use cases in which a client must efficiently synchronize a local image store against a remote server, acquiring only the subset of images it does not already possess (delta sync). The protocol is deliberately minimal: it provides catalog listing, targeted retrieval, batch delta sync, and a combined single-round-trip operation.

This specification defines:

- * The on-wire encoding of all request and response message types.
- * The data types, flag fields, and identifier derivation rules used by the protocol.
- * Transport requirements and optional TLS integration.
- * Error signaling, resource limits, and extensibility mechanisms.

1.1. Requirements Language

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

1.2. Terminology

Client An endpoint that initiates connections to a server and sends JTP requests.

Server An endpoint that accepts connections from clients and serves JTP responses.

ImageID A 64-bit content identifier derived from the raw bytes of an image file, computed as `xxHash64(image_bytes, seed=0)`.

Varint An unsigned LEB128 (Little-Endian Base 128) encoding of a 32-bit unsigned integer, as defined in Section 4.2.

Catalog The set of (ImageID, flags, filename, size) tuples returned by a LIST response.

Image Packet The wire encoding of a single image, comprising flags, length, ImageID, and raw data bytes.

Delta Sync The BATCH operation in which a client sends its set of known ImageIDs and the server returns only the images the client does not have.

2. Protocol Overview

JTP is a request/response protocol. A single connection carries one or more request/response exchanges. A typical interaction proceeds as follows:

1. The client opens a connection (TCP or TLS-wrapped TCP) to the server.
2. The client transmits a LIST request (ReqType = 1). The server replies with a catalog frame containing (ImageID, Flags, Filename, Size) entries for every image it serves.
3. The client requests images using one of:

- * GET_BY_ID (ReqType = 0): explicit retrieval of up to 255 images by their ImageIDs.
 - * BATCH (ReqType = 2): delta sync in which the client provides all ImageIDs it already holds; the server returns only the missing images.
 - * LIST_AND_GET (ReqType = 5): a combined single-round-trip operation that returns all available images together with their metadata.
4. The server transmits zero or more image packets, each containing flags, byte length, ImageID, and raw (possibly compressed) image data.

2.1. Connection Reuse (Keep-Alive)

JTP supports connection reuse through a keep-alive mechanism. When enabled, multiple request/response exchanges may be performed over a single connection, amortizing connection establishment and TLS handshake costs.

- * If the keep-alive flag (bit 0 of RequestFlags) is set in a request, the server SHOULD keep the connection open after completing the response and await the next request.
- * If the keep-alive flag is not set, the server SHOULD close the connection after the response has been fully sent.
- * Servers MAY implement idle timeouts and close stale keep-alive connections unilaterally.
- * Clients SHOULD NOT assume keep-alive is honored. Clients MUST handle server-initiated connection closure gracefully at any point after a complete response has been received.

Legacy deployments MAY use a one-request-per-connection mode: the client opens a connection, sends exactly one request, reads the complete response, and then the connection is closed by either party.

3. Transport

JTP requires an ordered, reliable, full-duplex byte stream. The default and RECOMMENDED transport is TCP. JTP MAY be wrapped in TLS [RFC8446] to provide confidentiality and integrity for both request and response data.

No default TCP port is assigned by this specification. Deployments SHOULD document the port(s) they use. The reference implementation defaults to port 8443.

3.1. TLS and Application-Layer Protocol Negotiation

When TLS is used, servers MAY advertise the following ALPN [RFC7301] protocol identifier:

- * jtp/1

Clients that support ALPN SHOULD offer jtp/1 during the TLS handshake. A server that does not recognize the offered ALPN identifier MAY proceed without ALPN selection or abort the handshake.

JTP does not define certificate distribution. Deployments MAY use self-signed certificates, a locally trusted certificate authority, or a certificate issued by a public PKI. Clients SHOULD verify the server certificate according to the applicable PKI policy.

4. Data Types

4.1. Unsigned Integers: u8, u16, u32, u64

JTP uses unsigned integers of 8, 16, 32, and 64 bits. Unless otherwise specified, all multi-byte fixed-width integers are encoded in network byte order (big-endian).

4.2. Variable-Length Integer: varint(u32)

The varint(u32) type uses unsigned LEB128 (Little-Endian Base 128) encoding. LEB128 is described in the DWARF debugging standard and is in common use in binary protocols (e.g., WebAssembly, Protocol Buffers). The following properties apply to the JTP varint(u32) type:

- * Encodable range: 0 through 4,294,967,295 (0x00000000 through 0xFFFFFFFF), inclusive.
- * Encoded length: 1 to 5 bytes.
- * Each byte stores 7 data bits in bits 0 through 6. Bit 7 (0x80) is the continuation bit; a value of 1 indicates that additional bytes follow.

Canonical encoding: Implementations SHOULD produce the minimal (canonical) encoding, i.e., no unnecessary high-order zero groups. Receivers MAY reject non-canonical encodings as malformed.

Example: the value 4660 (0x00001234) encodes as the two-byte sequence 0xB4 0x24. Derivation: 4660 in binary is 0001 0010 0011 0100. Split into 7-bit groups from least significant: 0110100 (0x34) and 0100100 (0x24 without the continuation bit, giving 0x24 with bit 7 clear for the final byte). The first byte has bit 7 set: 0xB4. The second byte is 0x24.

4.3. UTF-8 Strings

Filenames in the catalog are encoded as UTF-8 [RFC3629] byte sequences. The protocol transmits an explicit byte-length prefix; no null terminator is used. Receivers SHOULD validate that filename bytes constitute well-formed UTF-8.

5. Identifiers

5.1. ImageID

An ImageID is a 64-bit unsigned integer computed from the raw bytes of an image file using the xxHash64 non-cryptographic hash function with a seed value of zero:

```
ImageID = xxHash64(image_bytes, seed = 0)
```

The xxHash64 algorithm is defined in the xxHash specification [xxHash]. Implementors MUST use seed value 0.

On the wire, ImageID is transmitted as a u64 in big-endian byte order. When rendered in human-readable form (e.g., log output), the RECOMMENDED representation is the 16-character lowercase hexadecimal encoding of the 8 big-endian bytes.

ImageID provides content integrity verification but is NOT a cryptographic message authentication code (MAC). An adversary with the ability to modify transmitted data can also forge an ImageID. Cryptographic integrity of image content requires TLS or an equivalent channel security mechanism (see Section 12).

6. Flags Field

Both catalog entries and image packets carry a one-octet Flags field. The bit assignments are:

Bits	Mask	Name	Description
0-2	0x07	FileType	Image file type code (see Section 6.1)
3	0x08	Compressed	1 = image data is Zstd compressed
4	0x10	Encrypted	Reserved for future encryption support; MUST be 0
5-7	0xE0	(reserved)	MUST be 0 unless defined by a future extension

Table 1

6.1. File Type Codes

The three-bit FileType sub-field (bits 0-2) encodes the image format:

Code	Format
0	PNG
1	JPEG (JFIF / Exif)
2	WebP
3	BMP
4	GIF
5	Reserved
6	Reserved
7	Unknown / Other

Table 2

If the file type cannot be determined, senders SHOULD use code 7 (Unknown / Other).

6.2. Compression (Bit 3)

When bit 3 of the Flags field is set, the image data in the enclosing image packet (see Section 8.2) is compressed using Zstandard (Zstd) [RFC8878] at an implementation-defined compression level. Receivers MUST decompress the data before use or integrity verification.

Integrity verification via xxHash64 (see Section 5.1) MUST be performed against the decompressed data.

If a receiver does not support Zstd decompression and the Compressed bit is set, the receiver SHOULD treat the packet as an error and SHOULD close the connection or send an UnsupportedFeature ERROR response (see Section 9.2).

6.3. Encryption (Bit 4)

Bit 4 is reserved for future encryption support. Senders MUST set this bit to 0 in the current version of the protocol. Receivers that encounter this bit set SHOULD treat the packet as an error.

6.4. Reserved Bits (Bits 5-7)

Bits 5 through 7 are reserved. Senders MUST set reserved bits to 0. Receivers MAY reject messages containing non-zero reserved bits as malformed.

7. Requests

Every JTP request begins with a one-octet ReqType field that identifies the request type. Requests that support connection reuse carry a second one-octet RequestFlags field immediately following ReqType.

7.1. RequestFlags

Bit	Name	Description
0	keep-alive	1 = request connection keep-alive after response
1-7	(reserved)	MUST be 0 unless defined by a future extension

Table 3

Servers MUST reject requests that have any reserved RequestFlags bit set to 1 by either closing the connection or transmitting an InvalidRequest ERROR response (see Section 9.2).

7.2. LIST Request (ReqType = 1)

The LIST request asks the server to return a complete catalog of the images it currently serves. It carries no additional payload beyond the fixed two-octet header.

Field	Type	Size (octets)	Description
ReqType	u8	1	Value: 1
RequestFlags	u8	1	Bit 0 = keep-alive

Table 4

The server responds with a LIST response as defined in Section 8.1.

7.3. GET_BY_ID Request (ReqType = 0)

The GET_BY_ID request asks the server to return the image data for a specified set of ImageIDs.

Field	Type	Size (octets)	Description
ReqType	u8	1	Value: 0
RequestFlags	u8	1	Bit 0 = keep-alive
Count	u8	1	Number of ImageIDs (N)
ImageID[0..N-1]	u64	8 x N	Requested IDs, big-endian

Table 5

Semantics:

- * N MAY be zero, in which case no ImageID fields are present and the server returns zero image packets.
- * N MUST NOT exceed 255 (the maximum value of a u8).

- * Servers MAY silently ignore ImageIDs that are not present in the catalog.

Response framing: The GET_BY_ID response does not include an explicit top-level count. Clients SHOULD read exactly N image packets (as defined in Section 8.2). Servers that silently skip unknown IDs will therefore produce fewer than N image packets; clients MUST be prepared for this. If the keep-alive flag is set, the connection remains open after the last image packet.

7.4. BATCH Request (ReqType = 2)

The BATCH request implements delta synchronization. The client provides the complete set of ImageIDs it already holds; the server responds with only those images the client does not have.

Field	Type	Size (octets)	Description
ReqType	u8	1	Value: 2
RequestFlags	u8	1	Bit 0 = keep-alive
HaveCount	varint(u32)	1-5	Number of ImageIDs (N)
ImageID[0..N-1]	u64	8 x N	IDs the client already has

Table 6

Semantics:

- * The server computes the set difference (server catalog) minus (client's HaveSet) and returns exactly those images.
- * Servers SHOULD reject BATCH requests in which HaveCount exceeds 1,000,000, responding with an InvalidRequest ERROR.

The server responds with a BATCH response as defined in Section 8.3.

7.5. LIST_AND_GET Request (ReqType = 5)

The LIST_AND_GET request retrieves all available images in a single round trip, without a prior LIST exchange. The server returns all images together with their ImageIDs; no separate catalog is needed.

Field	Type	Size (octets)	Description
ReqType	u8	1	Value: 5
RequestFlags	u8	1	Bit 0 = keep-alive

Table 7

No additional payload. The server responds with a LIST_AND_GET response as defined in Section 8.4.

8. Responses

Each response type begins with a four-octet ASCII magic header that allows receivers to identify the response type and detect framing errors.

8.1. LIST Response

The LIST response carries the server's image catalog. It begins with the fixed frame:

Field	Type	Size (octets)	Description
Header	bytes	4	ASCII "JTPL" (0x4A 0x54 0x50 0x4C)
Count	u16	2	Number of catalog entries (N)
Entry[0..N-1]	-	variable	Repeated N times (see below)

Table 8

Each catalog entry has the following structure:

Field	Type	Size (octets)	Description
ImageID	u64	8	Content identifier, big-endian
Flags	u8	1	File type and feature flags (see Section 6)
NameLen	u16	2	Byte length of Filename
Filename	bytes	NameLen	UTF-8 basename of the image file
Size	varint(u32)	1-5	Byte length of image data in the corresponding image packet

Table 9

Notes:

- * Size is the byte count of the data field that will appear in an image packet for this ImageID. If the Compressed flag is set, Size reflects the compressed data length.
- * Filenames are informational only. Clients SHOULD NOT trust or use path components from filenames. Clients SHOULD sanitize filenames before using them as local filesystem paths.

8.2. Image Packet

Image packets are the common unit of image delivery used by the GET_BY_ID, BATCH, and LIST_AND_GET responses.

Field	Type	Size (octets)	Description
Flags	u8	1	File type and feature flags (see Section 6)
Length	varint(u32)	1-5	Byte length of Data
ImageID	u64	8	Content identifier, big-endian
Data	bytes	Length	Raw (possibly compressed) image bytes

Table 10

Integrity verification: After receiving an image packet (and decompressing if the Compressed flag is set), receivers SHOULD verify:

ImageID == xxHash64(Data, seed = 0)

If verification fails, receivers SHOULD discard the data and SHOULD treat the condition as a transmission error. Continuing to process corrupt data is NOT RECOMMENDED.

8.3. BATCH Response

Field	Type	Size (octets)	Description
Header	bytes	4	ASCII "JTPB" (0x4A 0x54 0x50 0x42)
MissingCount	varint(u32)	1-5	Number of missing images (M)
Packet[0..M-1]	-	variable	M image packets (see Section 8.2)

Table 11

The client reads exactly M image packets following the header.

8.4. LIST_AND_GET Response

Field	Type	Size (octets)	Description
Header	bytes	4	ASCII "JTPG" (0x4A 0x54 0x50 0x47)
Count	u16	2	Number of images (N)
Packet[0..N-1]	-	variable	N image packets (see Section 8.2)

Table 12

The client reads exactly N image packets. Because each image packet contains the ImageID, no separate catalog is required.

9. Error Handling

9.1. Structured ERROR Response

Servers that wish to provide machine-readable error information MAY transmit a structured ERROR response in place of the expected response frame:

Field	Type	Size (octets)	Description
Header	bytes	4	ASCII "JTPE" (0x4A 0x54 0x50 0x45)
ErrorCode	u8	1	Numeric error code (see Section 9.2)
MessageLen	u16	2	Byte length of Message
Message	bytes	MessageLen	UTF-8 human-readable error description

Table 13

9.2. Error Codes

Code	Name	Description
1	NotFound	One or more requested resources were not found
2	InvalidRequest	The request was malformed or violated a protocol constraint
3	ServerError	An internal server error occurred
4	UnsupportedFeature	A requested feature is not supported by this server
5	RateLimited	The request was refused because a rate limit was exceeded

Table 14

9.3. Legacy Error Signaling

Servers MAY signal errors by closing the TCP connection or terminating the TLS session without sending a structured ERROR response. Clients MUST handle the following conditions as request failure:

- * Unexpected TCP or TLS connection closure (EOF) before the response has been fully received.
- * A response magic header that does not match any known value ("JTPL", "JTPB", "JTPG", "JTPE").
- * A reserved Flags or RequestFlags bit set to 1.
- * A varint that is non-canonical or encodes a value exceeding 0xFFFFFFFF.
- * Any other decoding error that prevents the message from being parsed as specified in this document.

10. Limits and Resource Considerations

JTP implementations SHOULD defend against resource exhaustion attacks arising from large field values. In particular:

- * varint(u32) values that, when used as allocation sizes, would exhaust available memory SHOULD be rejected. Implementations MAY impose per-field upper bounds lower than 4,294,967,295.
- * Oversized NameLen values (e.g., values that would require reading more bytes than remain in the anticipated message) SHOULD be rejected.
- * Large Count or HaveCount values SHOULD be checked before allocating memory proportional to them.

Because the Length and Size fields are encoded as varint(u32), the maximum single image data payload supported by this framing is 4,294,967,295 octets ($2^{32} - 1$, approximately 4 GiB). Implementations MAY enforce stricter per-image size limits appropriate to their deployment context.

Servers SHOULD reject BATCH requests in which HaveCount exceeds 1,000,000 by transmitting an InvalidRequest ERROR response.

11. Extensibility

JTP version 1 is designed to accommodate future evolution without breaking existing implementations:

- * Unassigned ReqType values (currently: 3, 4, and values 6-255) are reserved. Future documents MAY define their semantics. Servers receiving an unrecognized ReqType SHOULD respond with an UnsupportedFeature ERROR and close the connection.
- * Reserved Flags bits (bits 5-7) and reserved RequestFlags bits (bits 1-7) MUST remain 0 in version 1 messages. Future extensions MAY assign meaning to these bits via a new specification. Receivers MUST be able to handle (or reject) messages with previously undefined bits set.
- * The Encrypted flag (Flags bit 4) is reserved for a future encryption layer definition. This specification does not define the encryption scheme; a future document will do so.

A future versioning scheme for the protocol as a whole MAY be introduced through one or more of the following mechanisms:

- * A new ALPN protocol identifier (e.g., jtp/2) negotiated during TLS handshake.
- * A new ReqType value designated for capability negotiation.

- * Explicit version magic fields in a revised framing layer.

12. Security Considerations

12.1. Transport Security

Deployments SHOULD use TLS [RFC8446] to protect JTP connections against passive eavesdropping and active tampering. Without TLS, both image content and ImageIDs are transmitted in the clear, and an on-path attacker may modify image data or inject fabricated responses.

12.2. Content Integrity

The xxHash64-derived ImageID provides a non-cryptographic integrity check against accidental corruption (e.g., transmission bit errors). It does NOT provide security against a malicious actor, who can compute a valid xxHash64 over any chosen payload. Applications that require cryptographic integrity assurance MUST rely on TLS record-layer integrity or a separate authenticated mechanism.

12.3. Denial of Service

Servers SHOULD validate and bound all count and size fields before allocating memory or performing I/O proportional to those values. Specific recommendations:

- * Reject BATCH requests with HaveCount exceeding 1,000,000.
- * Enforce maximum image sizes appropriate to the deployment.
- * Enforce keep-alive idle timeouts to reclaim connection resources from inactive clients.
- * Apply request rate limiting (e.g., using the RateLimited error code) to mitigate abusive clients.

12.4. Filename Safety

Filenames in the catalog are informational. Clients MUST NOT use catalog filenames as filesystem paths without first sanitizing them. In particular, clients MUST strip or reject path separators, relative path components (e.g., ".." sequences), and any characters not valid in the target filesystem. Failure to do so may allow a malicious server to write files to unintended locations (path traversal).

12.5. Certificate Validation

When TLS is used, clients SHOULD validate the server's certificate against a trusted trust anchor appropriate to the deployment. The use of self-signed certificates or a local CA is acceptable in controlled environments but introduces risks if the trust anchor is compromised or mis-deployed.

13. IANA Considerations

This document has no IANA actions at this time. A future revision of this specification MAY request registration of the following:

- * The ALPN Protocol ID jtp/1 in the "TLS Application-Layer Protocol Negotiation (ALPN) Protocol IDs" registry [RFC7301].
- * A TCP port number for JTP in the "Service Name and Transport Protocol Port Number Registry".
- * A registry for JTP ReqType values.
- * A registry for JTP Error Codes.

14. References

14.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", also known as BCP 14, BCP 14, RFC 2119, March 1997, <<https://www.rfc-editor.org/rfc/rfc2119>>.
- [RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003, <<https://www.rfc-editor.org/rfc/rfc3629>>.
- [RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, July 2014, <<https://www.rfc-editor.org/rfc/rfc7301>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, May 2017, <<https://www.rfc-editor.org/rfc/rfc8174>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, August 2018, <<https://www.rfc-editor.org/rfc/rfc8446>>.

[RFC8878] Collet, Y. and M. Kucherawy, Ed., "Zstandard Compression and the 'application/zstd' Media Type", RFC 8878, February 2021, <<https://www.rfc-editor.org/rfc/rfc8878>>.

14.2. Informative References

[JTP-REPO] Matt, "punctuations/jtp: High-performance binary protocol for efficient image transfer over TCP", 2026, <<https://github.com/punctuations/jtp>>.

[xxHash] Collet, Y., "xxHash - Extremely Fast Non-Cryptographic Hash Algorithm", 2023, <<https://cyan4973.github.io/xxHash/>>.

Appendix A. Varint Encoding Example

This appendix illustrates the unsigned LEB128 (`varint(u32)`) encoding used by JTP for the value 4660 (0x00001234).

Step 1: Represent the value in binary (14 significant bits):

4660 = 0001 0010 0011 0100 (binary)

Step 2: Group into 7-bit chunks from least significant to most significant (padding to a multiple of 7 if necessary):

Chunk 0 (least significant): 011 0100 = 0x34

Chunk 1 (most significant): 010 0100 = 0x24

Step 3: Set the continuation bit (bit 7 = 0x80) on all but the final chunk:

Byte 0: 0x34 | 0x80 = 0xB4 (more bytes follow)

Byte 1: 0x24 = 0x24 (final byte, continuation bit clear)

Encoded result: 0xB4 0x24 (2 bytes).

Decoding: multiply Chunk 1 by 2^7 (128) and add Chunk 0: $36 * 128 + 52 = 4608 + 52 = 4660$.

Appendix B. Wire Format Summary

The following diagrams provide a compact summary of each message format. All fields are transmitted left-to-right, top-to-bottom as written. Fields labelled "var" have variable length as described in the corresponding section.

B.1. Request Formats

LIST (ReqType = 1):

ReqType	ReqFlags
(0x01)	(u8)

GET_BY_ID (ReqType = 0):

ReqType	ReqFlags	Count	ImageID[0..N-1]
(0x00)	(u8)	(u8)	N x u64 big-endian

BATCH (ReqType = 2):

ReqType	ReqFlags	HaveCount	ImageID[0..N-1]
(0x02)	(u8)	varint(u32)	N x u64 big-endian

LIST_AND_GET (ReqType = 5):

ReqType	ReqFlags
(0x05)	(u8)

B.2. Response Formats

LIST response ("JTPL"):

"JTPL"	Count	Entries
4 bytes	(u16)	N x catalog entry (var)

Catalog entry:

ImageID	Flags	NameLen	Filename	Size
(u64)	(u8)	(u16)	NameLen	var

Image packet (used in GET_BY_ID, BATCH, LIST_AND_GET responses):

Flags	Length	ImageID	Data
(u8)	(var)	(u64)	Length bytes

BATCH response ("JTPB"):

"JTPB"	MissingCount	Images
4 bytes	varint(u32)	M x image pkt

LIST_AND_GET response ("JTPG"):

"JTPG"	Count	Images
4 bytes	(u16)	N x image pkt

ERROR response ("JTPE"):

"JTPE"	ErrorCode	MessageLen	Message
4 bytes	(u8)	(u16)	var

Author's Address

Matthew Baker
 Independent
 Canada
 Email: hey@mattt.space
 URI: <https://github.com/punctuations/jtp>