Workgroup: MASQUE
Internet-Draft:
draft-ietf-masque-h3-datagram-05
Published: 25 October 2021
Intended Status: Standards Track
Expires: 28 April 2022
Authors: D. Schinazi
Google LLC
L. Pardue
Cloudflare

Using Datagrams with HTTP

Abstract

The QUIC DATAGRAM extension provides application protocols running over QUIC with a mechanism to send unreliable data while leveraging the security and congestion-control properties of QUIC. However, QUIC DATAGRAM frames do not provide a means to demultiplex application contexts. This document describes how to use QUIC DATAGRAM frames when the application protocol running over QUIC is HTTP/3. It associates datagrams with client-initiated bidirectional streams and defines an optional additional demultiplexing layer. Additionally, this document defines how to convey datagrams over prior versions of HTTP.

Discussion Venues

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

Discussion of this document takes place on the MASQUE WG mailing list (masque@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/masque/.

Source for this draft and an issue tracker can be found at https://github.com/ietf-wg-masque/draft-ietf-masque-h3-datagram.

Status of This Memo

This Internet-Draft will expire on 28 April 2022.

Table of Contents

1. Introduction

The QUIC DATAGRAM extension [DGRAM] provides application protocols running over QUIC [QUIC] with a mechanism to send unreliable data while leveraging the security and congestion-control properties of QUIC. However, QUIC DATAGRAM frames do not provide a means to demultiplex application contexts. This document describes how to use QUIC DATAGRAM frames when the application protocol running over QUIC is HTTP/3 [H3]. It associates datagrams with client-initiated bidirectional streams and defines an optional additional demultiplexing layer. Additionally, this document defines how to convey datagrams over prior versions of HTTP.

This document is structured as follows:

* Section 2 presents core concepts for multiplexing across HTTP versions.
  - Section 2.1 defines datagram contexts, an optional end-to-end multiplexing concept scoped to each HTTP request. Whether contexts are in use is defined in Section 6.
  - Section 2.2 defines datagram formats, which are scoped to contexts. Formats communicate the format and encoding of datagrams sent using the associated context.
  - Contexts are identified using a variable-length integer. Requirements for allocating identifier values are detailed in Section 2.3.
* Section 3 defines how QUIC DATAGRAM frames are used with HTTP/3. Section 5 defines an HTTP/3 setting that endpoints can use to advertise support of the frame.
* Section 4 introduces the Capsule Protocol and the "data stream" concept. Data streams are initiated using special-purpose HTTP

requests, after which Capsules, an end-to-end message, can be sent.

-The following Capsule types are defined, together with guidance for defining new types:

oDatagram registration capsules Section 4.4.1

oDatagram close capsule Section 4.4.2

oDatagram capsules Section 4.4.3

1.1. Conventions and Definitions

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

2. Multiplexing

When running over HTTP/3, multiple exchanges of datagrams need the ability to coexist on a given QUIC connection. To allow this, HTTP datagrams contain two layers of multiplexing. First, the QUIC DATAGRAM frame payload starts with an encoded stream identifier that associates the datagram with a given QUIC stream. Second, datagrams optionally carry a context identifier (see Section 2.1) that allows multiplexing multiple datagram contexts related to a given HTTP request. Conceptually, the first layer of multiplexing is per-hop, while the second is end-to-end.

When running over HTTP/2, the first level of demultiplexing is provided by the HTTP/2 framing layer. When running over HTTP/1, requests are strictly serialized in the connection, therefore the first layer of demultiplexing is not needed.

2.1. Datagram Contexts

Within the scope of a given HTTP request, contexts provide an additional demultiplexing layer. Contexts determine the encoding of datagrams, and can be used to implicitly convey metadata. For example, contexts can be used for compression to elide some parts of the datagram: the context identifier then maps to a compression context that the receiver can use to reconstruct the elided data.

While stream IDs are a per-hop concept, context IDs are an end-to-end concept. In other words, if a datagram travels through one or more intermediaries on its way from client to server, the stream ID

will most likely change from hop to hop, but the context ID will remain the same. Context IDs are opaque to intermediaries.

Contexts are OPTIONAL to implement for both endpoints. Intermediaries do not require any context-specific software to enable contexts. When contexts are supported by the implementation, their use is optional and can be selected on each stream. Endpoints inform their peer of whether they wish to use contexts via the Sec-Use-Datagram-Contexts HTTP header, see Section 6.

When contexts are used, they are identified within the scope of a given request by a numeric value, referred to as the context ID. A context ID is a 62-bit integer (0 to 2^{62}-1).

2.2. Datagram Formats

When an endpoint registers a datagram context (or the lack of contexts), it communicates the format (i.e., the semantics and encoding) of datagrams sent using this context. This is acccomplished by sending a Datagram Format Type as part of the datagram registration capsule, see Section 4.4.1. This type identifier is registered with IANA (see Section 9.4) and allows applications that use HTTP Datagrams to indicate what the content of datagrams are. Registration capsules carry a Datagram Format Additional Data field which allows sending some additional information that would impact the format of datagrams.

For example, a protocol which proxies IP packets can define a Datagram Format Type which represents an IP packet. The corresponding Datagram Format Additional Data field would be empty. An extension to such a protocol that wishes to compress IP addresses could define a distinct Datagram Format Type and exchange two IP addresses via the Datagram Format Additional Data field. Then any datagrams with that type would contain the IP packet with addresses elided.

2.3. Context ID Allocation

Implementations of HTTP Datagrams that support datagram contexts MUST provide a context ID allocation service. That service will allow applications co-located with HTTP to request a unique context ID that they can subsequently use for their own purposes. The HTTP implementation will then parse the context ID of incoming HTTP Datagrams and use it to deliver the frame to the appropriate application context.

Even-numbered context IDs are client-initiated, while odd-numbered context IDs are server-initiated. This means that an HTTP client implementation of the context ID allocation service MUST only provide even-numbered IDs, while a server implementation MUST only

provide odd-numbered IDs. Note that, once allocated, any context ID can be used by both client and server - only allocation carries separate namespaces to avoid requiring synchronization. Additionally, note that the context ID namespace is tied to a given HTTP request: it is possible for the same numeral context ID to be used simultaneously in distinct requests.

3. HTTP/3 DATAGRAM Format

When used with HTTP/3, the Datagram Data field of QUIC DATAGRAM frames uses the following format (using the notation from the "Notational Conventions" section of [QUIC]):

```
HTTP/3 Datagram {
  Quarter Stream ID (i),
  [Context ID (i)],
  HTTP Datagram Payload (..),
}
```

Figure 1: HTTP/3 DATAGRAM Format

Quarter Stream ID: A variable-length integer that contains the value of the client-initiated bidirectional stream that this datagram is associated with, divided by four (the division by four stems from the fact that HTTP requests are sent on client-initiated bidirectional streams, and those have stream IDs that are divisible by four). The largest legal QUIC stream ID value is 2^{62}-1, so the largest legal value of Quarter Stream ID is 2^{62}-1 / 4. Receipt of a frame that includes a larger value MUST be treated as a connection error of type FRAME_ENCODING_ERROR.

Context ID: A variable-length integer indicating the context ID of the datagram (see Section 2.1). Whether or not this field is present depends on whether datagram contexts are in use on this stream, see Section 6. If this QUIC DATAGRAM frame is reordered and arrives before the receiver knows whether datagram contexts are in use on this stream, then the receiver cannot parse this datagram and the receiver MUST either drop that datagram silently or buffer it temporarily.

HTTP Datagram Payload: The payload of the datagram, whose semantics are defined by individual applications. Note that this field can be empty.

Intermediaries parse the Quarter Stream ID field in order to associate the QUIC DATAGRAM frame with a stream. If an intermediary receives a QUIC DATAGRAM frame whose payload is too short to allow parsing the Quarter Stream ID field, the intermediary MUST treat it as an HTTP/3 connection error of type H3_GENERAL_PROTOCOL_ERROR. The

Context ID field is optional and whether it is present or not is decided end-to-end by the endpoints, see Section 6. Therefore intermediaries cannot know whether the Context ID field is present or absent and they MUST ignore any HTTP/3 Datagram fields after the Quarter Stream ID.

Endpoints parse both the Quarter Stream ID field and the Context ID field in order to associate the QUIC DATAGRAM frame with a stream and context within that stream. If an endpoint receives a QUIC DATAGRAM frame whose payload is too short to allow parsing the Quarter Stream ID field, the endpoint MUST treat it as an HTTP/3 connection error of type H3_GENERAL_PROTOCOL_ERROR. If an endpoint receives a QUIC DATAGRAM frame whose payload is long enough to allow parsing the Quarter Stream ID field but too short to allow parsing the Context ID field, the endpoint MUST abruptly terminate the corresponding stream with a stream error of type H3_GENERAL_PROTOCOL_ERROR.

Endpoints MUST NOT send HTTP/3 datagrams unless the corresponding stream's send side is open. On a given endpoint, once the receive side of a stream is closed, incoming datagrams for this stream are no longer expected so the endpoint can release related state. Endpoints MAY keep state for a short time to account for reordering. Once the state is released, the endpoint MUST silently drop received associated datagrams.

If an HTTP/3 datagram is received and its Quarter Stream ID maps to a stream that has not yet been created, the receiver SHALL either drop that datagram silently or buffer it temporarily while awaiting the creation of the corresponding stream.

4. Capsules

This specification introduces the Capsule Protocol. The Capsule Protocol is a sequence of type-length-value tuples that allows endpoints to reliably communicate request-related information end-to-end, even in the presence of HTTP intermediaries.

4.1. Capsule Protocol

This specification defines the "data stream" of an HTTP request as the bidirectional stream of bytes that follow the headers in both directions. In HTTP/1.x, the data stream consists of all bytes on the connection that follow the blank line that concludes either the request header section, or the 2xx (Successful) response header section. In HTTP/2 and HTTP/3, the data stream of a given HTTP request consists of all bytes sent in DATA frames with the corresponding stream ID. The concept of a data stream is

particularly relevant for methods such as CONNECT where there is no HTTP message content after the headers.

Definitions of new HTTP Methods or of new HTTP Upgrade Tokens can state that their data stream uses the Capsule Protocol. If they do so, that means that the contents of their data stream uses the following format (using the notation from the "Notational Conventions" section of [QUIC]):

```
Capsule Protocol {
  Capsule (..) ...,
}
```

Figure 2: Capsule Protocol Stream Format

```
Capsule {
  Capsule Type (i),
  Capsule Length (i),
  Capsule Value (..),
}
```

Figure 3: Capsule Format

Capsule Type: A variable-length integer indicating the Type of the capsule. Endpoints that receive a capsule with an unknown Capsule Type MUST silently skip over that capsule.

Capsule Length: The length of the Capsule Value field following this field, encoded as a variable-length integer. Note that this field can have a value of zero.

Capsule Value: The payload of this capsule. Its semantics are determined by the value of the Capsule Type field.

4.2. Requirements

If the definition of an HTTP Method or HTTP Upgrade Token states that it uses the capsule protocol, its implementations MUST follow the following requirements:

* A server MUST NOT send any Transfer-Encoding or Content-Length header fields in a 2xx (Successful) response. If a client receives a Content-Length or Transfer-Encoding header fields in a successful response, it MUST treat that response as malformed.

* A request message does not have content.

* A successful response message does not have content.

*Responses are not cacheable.

4.3. Intermediary Processing

Intermediaries MUST operate in one of the two following modes:

Pass-through mode: In this mode, the intermediary forwards the data stream between two associated streams without any modification of the data stream.

Participant mode: In this mode, the intermediary terminates the data stream and parses all Capsule Type and Capsule Length fields it receives.

Each Capsule Type determines whether it is opaque or transparent to intermediaries in participant mode: opaque capsules are forwarded unmodified while transparent ones can be parsed, added, or removed by intermediaries. Intermediaries MAY modify the contents of the Capsule Data field of transparent capsule types.

Unless otherwise specified, all Capsule Types are defined as opaque to intermediaries. Intermediaries MUST forward all received opaque CAPSULE frames in their unmodified entirety. Intermediaries MUST NOT send any opaque CAPSULE frames other than the ones it is forwarding. All Capsule Types defined in this document are opaque, with the exception of the datagram capsules, see Section 4.4.3. Definitions of new Capsule Types MAY specify that the newly introduced type is transparent. Intermediaries MUST treat unknown Capsule Types as opaque.

Intermediaries respect the order of opaque CAPSULE frames: if an intermediary receives two opaque CAPSULE frames in a given order, it MUST forward them in the same order.

Endpoints which receive a Capsule with an unknown Capsule Type MUST silently drop that Capsule.

4.4. Capsule Types

4.4.1. The Datagram Registration Capsules

This document defines the REGISTER_DATAGRAM and REGISTER_DATAGRAM_CONTEXT capsules types, known collectively as the datagram registration capsules (see Section 9.3 for the value of the capsule types). The REGISTER_DATAGRAM capsule is used by endpoints to inform their peer of the encoding and semantics of all datagrams associated with a stream. The REGISTER_DATAGRAM_CONTEXT capsule is used by endpoints to inform their peer of the encoding and semantics of all datagrams associated with a given context ID on this stream.

```
Datagram Registration Capsule {
  Type (i) = REGISTER_DATAGRAM or REGISTER_DATAGRAM_CONTEXT,
  Length (i),
  [Context ID (i)],
  Datagram Format Type (i),
  Datagram Format Additional Data (..),
}
```

Figure 4: REGISTER_DATAGRAM_CONTEXT Capsule Format

Context ID: A variable-length integer indicating the context ID to register (see Section 2.1). This field is present in REGISTER_DATAGRAM_CONTEXT capsules but not in REGISTER_DATAGRAM capsules. If a REGISTER_DATAGRAM capsule is used on a stream where datagram contexts are in use, it is associated with context ID 0. REGISTER_DATAGRAM_CONTEXT capsules MUST NOT carry context ID 0 as that context ID is conveyed using the REGISTER_DATAGRAM capsule.

Datagram Format Type: A variable-length integer that defines the semantics and encoding of the HTTP Datagram Payload field of datagrams with this context ID, see Section 2.2.

Datagram Format Additional Data: This field carries additional information that impact the format of datagrams with this context ID, see Section 2.2.

Note that these registrations are unilateral and bidirectional: the sender of the capsule unilaterally defines the semantics it will apply to the datagrams it sends and receives using this context ID. Once a context ID is registered, it can be used in both directions.

Endpoints MUST NOT send HTTP Datagrams until they have either sent or received a datagram registration capsule with the same Context ID. However, reordering can cause HTTP Datagrams to be received with an unknown Context ID. Receipt of such HTTP datagrams MUST NOT be treated as an error. Endpoints SHALL drop the HTTP Datagram silently, or buffer it temporarily while awaiting the corresponding datagram registration capsule. Intermediaries SHALL drop the HTTP Datagram silently, MAY buffer it, or forward it on immediately.

Endpoints MUST NOT register the same Context ID twice on the same stream. This also applies to Context IDs that have been closed using a CLOSE_DATAGRAM_CONTEXT capsule. Clients MUST NOT register server-initiated Context IDs and servers MUST NOT register client-initiated Context IDs. If an endpoint receives a REGISTER_DATAGRAM_CONTEXT capsule that violates one or more of these requirements, the endpoint MUST abruptly terminate the corresponding stream with a stream error of type H3_GENERAL_PROTOCOL_ERROR.

If datagrams contexts are not in use, the client is responsible for choosing the datagram format and informing the server via a REGISTER_DATAGRAM capsule. Servers MUST NOT send the REGISTER_DATAGRAM capsule. If a client receives a REGISTER_DATAGRAM capsule, the client MUST abruptly terminate the corresponding stream with a stream error of type H3_GENERAL_PROTOCOL_ERROR.

4.4.2. The Datagram Close Capsule

The CLOSE_DATAGRAM_CONTEXT capsule (see Section 9.3 for the value of the capsule type) allows an endpoint to inform its peer that it will no longer send or parse received datagrams associated with a given context ID.

```
CLOSE_DATAGRAM_CONTEXT Capsule {
  Type (i) = CLOSE_DATAGRAM_CONTEXT,
  Length (i),
  Context ID (i),
  Close Code (i),
  Close Details (..),
}
```

Figure 5: CLOSE_DATAGRAM_CONTEXT Capsule Format

Context ID: The context ID to close.

Close Code: The close code allows an endpoint to provide additional information as to why a datagram context was closed. Section 4.4.2.1 defines a set of codes, the circumstances under which an implementation sends them, and how receivers react.

Close Details: This is meant for debugging purposes. It consists of a human-readable string encoded in UTF-8.

Note that this close is unilateral and bidirectional: the sender of the frame unilaterally informs its peer of the closure. Endpoints can use CLOSE_DATAGRAM_CONTEXT capsules to close a context that was initially registered by either themselves, or by their peer. Endpoints MAY use the CLOSE_DATAGRAM_CONTEXT capsule to immediately reject a context that was just registered using a REGISTER_DATAGRAM_CONTEXT capsule if they find its Datagram Format Type field to be unacceptable.

After an endpoint has either sent or received a CLOSE_DATAGRAM_CONTEXT frame, it MUST NOT send any HTTP Datagrams with that Context ID. However, due to reordering, an endpoint that receives an HTTP Datagram with a closed Context ID MUST NOT treat it as an error, it SHALL instead drop the HTTP Datagram silently.

Endpoints MUST NOT close a Context ID that was not previously registered. Endpoints MUST NOT close a Context ID that has already been closed. If an endpoint receives a CLOSE_DATAGRAM_CONTEXT capsule that violates one or more of these requirements, the endpoint MUST abruptly terminate the corresponding stream with a stream error of type H3_GENERAL_PROTOCOL_ERROR.

4.4.2.1. Close Codes

Close codes are intended to allow implementations to react differently when they receive them - for example, some close codes require the receiver to not open another context under certain conditions.

This specification defines the close codes below. Their numeric values are in Section 9.5. Extensions to this mechanism MAY define new close codes and they SHOULD state how receivers react to them.

NO_ERROR: This indicates that a context was closed without any action specified for the receiver.

UNKNOWN_FORMAT: This indicates that the sender does not know how to interpret the datagram format type associated with this context. The endpoint that had originally registered this context MUST NOT try to register another context with the same datagram format type on this stream.

DENIED: This indicates that the sender has rejected the context registration based on its local policy. The endpoint that had originally registered this context MUST NOT try to register another context with the same datagram format type and datagram format data on this stream.

RESOURCE_LIMIT: This indicates that the context was closed to save resources. The recipient SHOULD limit its future registration of resource-intensive contexts.

Receipt of an unknown close code MUST be treated as if the NO_ERROR code was present. Close codes are registered with IANA, see Section 9.5.

4.4.3. The Datagram Capsules

This document defines the DATAGRAM and DATAGRAM_WITH_CONTEXT capsules types, known collectively as the datagram capsules (see Section 9.3 for the value of the capsule types). These capsules allow an endpoint to send a datagram frame over an HTTP stream. This is particularly useful when using a version of HTTP that does not support QUIC DATAGRAM frames.

```
Datagram Capsule {
  Type (i) = DATAGRAM or DATAGRAM_WITH_CONTEXT,
  Length (i),
  [Context ID (i)],
  HTTP Datagram Payload (..),
}
```

Figure 6: DATAGRAM Capsule Format

Context ID: A variable-length integer indicating the context ID of the datagram (see Section 2.1). This field is present in DATAGRAM_WITH_CONTEXT capsules but not in DATAGRAM capsules. If a DATAGRAM capsule is used on a stream where datagram contexts are in use, it is associated with context ID 0. DATAGRAM_WITH_CONTEXT capsules MUST NOT carry context ID 0 as that context ID is conveyed using the DATAGRAM capsule.

HTTP Datagram Payload: The payload of the datagram, whose semantics are defined by individual applications. Note that this field can be empty.

Datagrams sent using the datagram capsule have the exact same semantics as datagrams sent in QUIC DATAGRAM frames. In particular, the restrictions on when it is allowed to send an HTTP Datagram and how to process them from Section 3 also apply to HTTP Datagrams sent and received using the datagram capsules.

The datagram capsules are transparent to intermediaries, meaning that intermediaries MAY parse them and send datagram capsules that they did not receive. This allows an intermediary to reencode HTTP Datagrams as it forwards them: in other words, an intermediary MAY send a datagram capsule to forward an HTTP Datagram which was received in a QUIC DATAGRAM frame, and vice versa.

Note that while datagram capsules are sent on a stream, intermediaries can reencode HTTP Datagrams into QUIC DATAGRAM frames over the next hop, and those could be dropped. Because of this, applications have to always consider HTTP Datagrams to be unreliable, even if they were initially sent in a capsule.

If an intermediary receives an HTTP Datagram in a QUIC DATAGRAM frame and is forwarding it on a connection that supports QUIC DATAGRAM frames, the intermediary SHOULD NOT convert that HTTP Datagram to a DATAGRAM capsule. If the HTTP Datagram is too large to fit in a DATAGRAM frame (for example because the path MTU of that QUIC connection is too low or if the maximum UDP payload size advertised on that connection is too low), the intermediary SHOULD drop the HTTP Datagram instead of converting it to a DATAGRAM capsule. This preserves the end-to-end unreliability characteristic

that methods such as Datagram Packetization Layer Path MTU Discovery (DPLPMTUD) depend on [RFC8899]. An intermediary that converts QUIC DATAGRAM frames to datagram capsules allows HTTP Datagrams to be arbitrarily large without suffering any loss; this can misrepresent the true path properties, defeating methods such a DPLPMTUD.

5. The H3_DATAGRAM HTTP/3 SETTINGS Parameter

Implementations of HTTP/3 that support HTTP Datagrams can indicate that to their peer by sending the H3_DATAGRAM SETTINGS parameter with a value of 1. The value of the H3_DATAGRAM SETTINGS parameter MUST be either 0 or 1. A value of 0 indicates that HTTP Datagrams are not supported. An endpoint that receives the H3_DATAGRAM SETTINGS parameter with a value that is neither 0 or 1 MUST terminate the connection with error H3_SETTINGS_ERROR.

Endpoints MUST NOT send QUIC DATAGRAM frames until they have both sent and received the H3_DATAGRAM SETTINGS parameter with a value of 1.

When clients use 0-RTT, they MAY store the value of the server's H3_DATAGRAM SETTINGS parameter. Doing so allows the client to send QUIC DATAGRAM frames in 0-RTT packets. When servers decide to accept 0-RTT data, they MUST send a H3_DATAGRAM SETTINGS parameter greater than or equal to the value they sent to the client in the connection where they sent them the NewSessionTicket message. If a client stores the value of the H3_DATAGRAM SETTINGS parameter with their 0-RTT state, they MUST validate that the new value of the H3_DATAGRAM SETTINGS parameter sent by the server in the handshake is greater than or equal to the stored value; if not, the client MUST terminate the connection with error H3_SETTINGS_ERROR. In all cases, the maximum permitted value of the H3_DATAGRAM SETTINGS parameter is 1.

5.1. Note About Draft Versions

[[RFC editor: please remove this section before publication.]]

Some revisions of this draft specification use a different value (the Identifier field of a Setting in the HTTP/3 SETTINGS frame) for the H3_DATAGRAM Settings Parameter. This allows new draft revisions to make incompatible changes. Multiple draft versions MAY be supported by either endpoint in a connection. Such endpoints MUST send multiple values for H3_DATAGRAM. Once an endpoint has sent and received SETTINGS, it MUST compute the intersection of the values it has sent and received, and then it MUST select and use the most recent draft version from the intersection set. This ensures that both endpoints negotiate the same draft version.

6. The Sec-Use-Datagram-Contexts HTTP Header

Endpoints indicate their support for datagram contexts by sending the Sec-Use-Datagram-Contexts header with a value of ?1. If the header is missing or has a value different from ?1, that indicates that its sender does not wish to use datagram contexts. Endpoints that wish to use datagram contexts SHALL send the Sec-Use-Datagram-Contexts header with a value of ?1 on requests and responses that use the capsule protocol.

"Sec-Use-Datagram-Contexts" is an Item Structured Header [RFC8941]. Its value MUST be a Boolean, its ABNF is:

```
Sec-Use-Datagram-Contexts = sf-boolean
```

The REGISTER_DATAGRAM_CONTEXT, DATAGRAM_WITH_CONTEXT, and CLOSE_DATAGRAM_CONTEXT capsules as refered to as context-related capsules. Endpoints which do not wish to use contexts MUST NOT send context-related capsules, and MUST silently ignore any received context-related capsules.

Both endpoints unilaterally decide whether they wish to use datagram contexts on a given stream. Contexts are used on a given stream if and only if both endpoints indicate they wish to use them on this stream. Once an endpoint has received the HTTP request or response, it knows whether datagram contexts are in use on this stream or not.

Conceptually, when datagram contexts are not in use on a stream, all datagrams use context ID 0, which is client-initiated. This means that the client chooses the datagram format for all datagrams when datagram contexts are not in use.

If datagram contexts are not in use on a stream, endpoints MUST NOT send context-related capsules to the peer on that stream. Clients MAY optimistically send context-related capsules before learning whether the server wishes to support datagram contexts or not.

This allows a client to optimistically use extensions that rely on datagram contexts without knowing a priori whether the server supports them, and without incurring a latency cost to negotiate extension support. In this scenario, the client would send its request with the Sec-Use-Datagram-Contexts header set to ?1, and register two datagram contexts: the main context would use context ID 0 and the extension context would use context ID 2. The client then sends a REGISTER_DATAGRAM capsule to register the main context, and a REGISTER_DATAGRAM_CONTEXT to register the extension context. The client can then immediately send DATAGRAM capsules to send main

datagrams and DATAGRAM_WITH_CONTEXT capsules to send extension datagrams.

- If the server wishes to use datagram contexts, it will set Sec-Use-Datagram-Contexts to ?1 on its response and correctly parse all the received capsules.

- If the server does not wish to use datagram contexts (for example if the server implementation does not support them), it will not set Sec-Use-Datagram-Contexts to ?1 on its response. It will then parse the REGISTER_DATAGRAM and DATAGRAM capsules without datagram contexts being in use on this stream, and parse the main datagrams correctly while silently dropping the extension datagrams. Once the client receives the server's response, it will know datagram contexts are not in use, and then will be able to send HTTP Datagrams via the QUIC DATAGRAM frame.

Extensions MAY define a different mechanism to communicate whether contexts are in use, and they MAY do so in a way which is opaque to intermediaries.

7. Prioritization

Data streams (see Section 4.1) can be prioritized using any means suited to stream or request prioritization. For example, see Section 11 of [PRIORITY].

Prioritization of HTTP/3 datagrams is not defined in this document. Future extensions MAY define how to prioritize datagrams, and MAY define signaling to allow endpoints to communicate their prioritization preferences.

8. Security Considerations

Since this feature requires sending an HTTP/3 Settings parameter, it "sticks out". In other words, probing clients can learn whether a server supports this feature. Implementations that support this feature SHOULD always send this Settings parameter to avoid leaking the fact that there are applications using HTTP/3 datagrams enabled on this endpoint.

9. IANA Considerations

9.1. HTTP/3 SETTINGS Parameter

This document will request IANA to register the following entry in the "HTTP/3 Settings" registry:

Setting Name	Value	Specification	Default
H3_DATAGRAM	0xffd277	This Document	0

Table 1: New HTTP/3 Settings

9.2. HTTP Header Field Name

This document will request IANA to register the following entry in the "HTTP Field Name" registry:

Field Name: Sec-Use-Datagram-Contexts

Template: None

Status: provisional (permanent if this document is approved)

Reference: This document

Comments: None

9.3. Capsule Types

This document establishes a registry for HTTP capsule type codes. The "HTTP Capsule Types" registry governs a 62-bit space. Registrations in this registry MUST include the following fields:

Type: A name or label for the capsule type.

Value: The value of the Capsule Type field (see Section 4.1) is a 62-bit integer.

Reference: An optional reference to a specification for the type. This field MAY be empty.

Registrations follow the "First Come First Served" policy (see Section 4.4 of [IANA-POLICY]) where two registrations MUST NOT have the same Type.

This registry initially contains the following entries:

Capsule Type	Value	Specification
REGISTER_DATAGRAM_CONTEXT	0xff37a1	This Document
REGISTER_DATAGRAM	0xff37a2	This Document
CLOSE_DATAGRAM_CONTEXT	0xff37a3	This Document
DATAGRAM_WITH_CONTEXT	0xff37a4	This Document
DATAGRAM	0xff37a5	This Document

Table 2: Initial Capsule Types Registry Entries

Capsule types with a value of the form 41 * N + 23 for integer values of N are reserved to exercise the requirement that unknown

capsule types be ignored. These capsules have no semantics and can carry arbitrary values. These values MUST NOT be assigned by IANA and MUST NOT appear in the listing of assigned values.

9.4. Datagram Format Types

This document establishes a registry for HTTP datagram format type codes. The "HTTP Datagram Format Types" registry governs a 62-bit space. Registrations in this registry MUST include the following fields:

Type: A name or label for the datagram format type.

Value: The value of the Datagram Format Type field (see Section 2.2) is a 62-bit integer.

Reference: An optional reference to a specification for the parameter. This field MAY be empty.

Registrations follow the "First Come First Served" policy (see Section 4.4 of [IANA-POLICY]) where two registrations MUST NOT have the same Type nor Value.

This registry is initially empty.

Datagram format types with a value of the form 41 * N + 17 for integer values of N are reserved to exercise the requirement that unknown datagram format types be ignored. These format types have no semantics and can carry arbitrary values. These values MUST NOT be assigned by IANA and MUST NOT appear in the listing of assigned values.

9.5. Context Close Codes

This document establishes a registry for HTTP context close codes. The "HTTP Context Close Codes" registry governs a 62-bit space. Registrations in this registry MUST include the following fields:

Type: A name or label for the close code.

Value: The value of the Close Code field (see Section 4.4.2) is a 62-bit integer.

Reference: An optional reference to a specification for the parameter. This field MAY be empty.

Registrations follow the "First Come First Served" policy (see Section 4.4 of [IANA-POLICY]) where two registrations MUST NOT have the same Type nor Value.

This registry initially contains the following entries:

Context Close Code	Value	Specification
NO_ERROR	0xff78a0	This Document
UNKNOWN_FORMAT	0xff78a1	This Document
DENIED	0xff78a2	This Document
RESOURCE_LIMIT	0xff78a3	This Document

Table 3: Initial Context Close Code Registry Entries

Context close codes with a value of the form 41 * N + 19 for integer values of N are reserved to exercise the requirement that unknown context close codes be treated as NO_ERROR. These values MUST NOT be assigned by IANA and MUST NOT appear in the listing of assigned values.

10. References

10.1. Normative References

[DGRAM] Pauly, T., Kinnear, E., and D. Schinazi, "An Unreliable Datagram Extension to QUIC", Work in Progress, Internet-Draft, draft-ietf-quic-datagram-06, 5 October 2021, <https://datatracker.ietf.org/doc/html/draft-ietf-quic-datagram-06>.

[H3] Bishop, M., "Hypertext Transfer Protocol Version 3 (HTTP/3)", Work in Progress, Internet-Draft, draft-ietf-quic-http-34, 2 February 2021, <https://datatracker.ietf.org/doc/html/draft-ietf-quic-http-34>.

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

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

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/

RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

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

[RFC8941] Nottingham, M. and P-H. Kamp, "Structured Field Values for HTTP", RFC 8941, DOI 10.17487/RFC8941, February 2021, <https://www.rfc-editor.org/rfc/rfc8941>.

10.2. Informative References

[PRIORITY] Oku, K. and L. Pardue, "Extensible Prioritization Scheme for HTTP", Work in Progress, Internet-Draft, draft-ietf-httpbis-priority-07, 25 October 2021, <https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-priority-07>.

[RFC8899] Fairhurst, G., Jones, T., Tüxen, M., Rüngeler, I., and T. Völker, "Packetization Layer Path MTU Discovery for Datagram Transports", RFC 8899, DOI 10.17487/RFC8899, September 2020, <https://www.rfc-editor.org/rfc/rfc8899>.

Appendix A. Examples

A.1. CONNECT-UDP

In this example, the client does not support any CONNECT-UDP nor HTTP Datagram extensions, and therefore has no use for datagram contexts on this stream.

```
Client                                                  Server

STREAM(44): HEADERS              -------->
  :method = CONNECT
  :protocol = connect-udp
  :scheme = https
  :path = /target.example.org/443/
  :authority = proxy.example.org:443

STREAM(44): DATA                 -------->
  Capsule Type = REGISTER_DATAGRAM
  Datagram Format Type = UDP_PAYLOAD
  Datagram Format Additional Data = ""

DATAGRAM                         -------->
  Quarter Stream ID = 11
  Payload = Encapsulated UDP Payload

           <--------  STREAM(44): HEADERS
                        :status = 200

/* Wait for target server to respond to UDP packet. */

           <--------  DATAGRAM
                        Quarter Stream ID = 11
                        Payload = Encapsulated UDP Payload
```

A.2. CONNECT-UDP with Delayed Timestamp Extension

In these examples, the client supports a CONNECT-UDP Timestamp Extension, which uses a different Datagram Format Type that carries a timestamp followed by the encapsulated UDP payload.

A.2.1. With Delay

In this instance, the client prefers to wait a round trip to learn whether the server supports datagram contexts.

```
Client                                                  Server

STREAM(44): HEADERS            -------->
  :method = CONNECT
  :protocol = connect-udp
  :scheme = https
  :path = /target.example.org/443/
  :authority = proxy.example.org:443
  Sec-Use-Datagram-Contexts = ?1

           <--------  STREAM(44): HEADERS
                        :status = 200
                        Sec-Use-Datagram-Contexts = ?1

STREAM(44): DATA               -------->
  Capsule Type = REGISTER_DATAGRAM_CONTEXT
  Context ID = 0
  Datagram Format Type = UDP_PAYLOAD
  Datagram Format Additional Data = ""

DATAGRAM                        -------->
  Quarter Stream ID = 11
  Context ID = 0
  Payload = Encapsulated UDP Payload

           <--------  DATAGRAM
                        Quarter Stream ID = 11
                        Context ID = 0
                        Payload = Encapsulated UDP Payload

STREAM(44): DATA               -------->
  Capsule Type = REGISTER_DATAGRAM_CONTEXT
  Context ID = 2
  Datagram Format Type = UDP_PAYLOAD_WITH_TIMESTAMP
  Datagram Format Additional Data = ""

DATAGRAM                        -------->
  Quarter Stream ID = 11
  Context ID = 2
  Payload = Encapsulated UDP Payload With Timestamp
```

A.3. Successful Optimistic

In this instance, the client does not wish to spend a round trip waiting to learn whether the server supports datagram contexts. It registers its context optimistically in such a way that the server will react well whether it supports contexts or not. In this case, the server does support datagram contexts.

```
Client                                                  Server

STREAM(44): HEADERS            -------->
  :method = CONNECT
  :protocol = connect-udp
  :scheme = https
  :path = /target.example.org/443/
  :authority = proxy.example.org:443
  Sec-Use-Datagram-Contexts = ?1

STREAM(44): DATA               -------->
  Capsule Type = REGISTER_DATAGRAM
  Datagram Format Type = UDP_PAYLOAD
  Datagram Format Additional Data = ""

STREAM(44): DATA               -------->
  Capsule Type = DATAGRAM
  Payload = Encapsulated UDP Payload

           <--------  STREAM(44): HEADERS
                        :status = 200
                        Sec-Use-Datagram-Contexts = ?1

/* Datagram contexts are in use on this stream */

           <--------  DATAGRAM
                        Quarter Stream ID = 11
                        Context ID = 0
                        Payload = Encapsulated UDP Payload

STREAM(44): DATA               -------->
  Capsule Type = REGISTER_DATAGRAM_CONTEXT
  Context ID = 2
  Datagram Format Type = UDP_PAYLOAD_WITH_TIMESTAMP
  Datagram Format Additional Data = ""

DATAGRAM                       -------->
  Quarter Stream ID = 11
  Context ID = 2
  Payload = Encapsulated UDP Payload With Timestamp
```

A.4. Optimistic but Unsupported

In this instance, the client does not wish to spend a round trip waiting to learn whether the server supports datagram contexts. It registers its context optimistically in such a way that the server will react well whether it supports contexts or not. In this case, the server does not support datagram contexts.

```
Client                                                  Server

STREAM(44): HEADERS             -------->
  :method = CONNECT
  :protocol = connect-udp
  :scheme = https
  :path = /target.example.org/443/
  :authority = proxy.example.org:443
  Sec-Use-Datagram-Contexts = ?1

STREAM(44): DATA                -------->
  Capsule Type = REGISTER_DATAGRAM
  Datagram Format Type = UDP_PAYLOAD
  Datagram Format Additional Data = ""

STREAM(44): DATA                -------->
  Capsule Type = DATAGRAM
  Payload = Encapsulated UDP Payload

            <--------  STREAM(44): HEADERS
                         :status = 200

/* Datagram contexts are not in use on this stream */

            <--------  DATAGRAM
                         Quarter Stream ID = 11
                         Payload = Encapsulated UDP Payload

DATAGRAM                        -------->
  Quarter Stream ID = 11
  Payload = Encapsulated UDP Payload
```

A.5. CONNECT-IP with IP compression

```
Client                                                Server

STREAM(44): HEADERS            -------->
  :method = CONNECT
  :protocol = connect-ip
  :scheme = https
  :path = /
  :authority = proxy.example.org:443
  Sec-Use-Datagram-Contexts = ?1

           <--------  STREAM(44): HEADERS
                        :status = 200
                        Sec-Use-Datagram-Contexts = ?1

/* Exchange CONNECT-IP configuration information. */

STREAM(44): DATA                --------->
  Capsule Type = REGISTER_DATAGRAM_CONTEXT
  Context ID = 0
  Datagram Format Type = IP_PACKET
  Datagram Format Additional Data = ""

DATAGRAM                        -------->
  Quarter Stream ID = 11
  Context ID = 0
  Payload = Encapsulated IP Packet

/* Endpoint happily exchange encapsulated IP packets */
/* using Quarter Stream ID 11 and Context ID 0.      */

DATAGRAM                        -------->
  Quarter Stream ID = 11
  Context ID = 0
  Payload = Encapsulated IP Packet

/* After performing some analysis on traffic patterns, */
/* the client decides it wants to compress a 2-tuple.  */


STREAM(44): DATA                --------->
  Capsule Type = REGISTER_DATAGRAM_CONTEXT
  Context ID = 2
  Datagram Format Type = COMPRESSED_IP_PACKET
  Datagram Format Additional Data = "192.0.2.6,192.0.2.7"

DATAGRAM                        -------->
  Quarter Stream ID = 11
  Context ID = 2
  Payload = Compressed IP Packet
```

A.6. WebTransport

```
Client                                                Server

STREAM(44): HEADERS            -------->
  :method = CONNECT
  :scheme = https
  :method = webtransport
  :path = /hello
  :authority = webtransport.example.org:443
  Origin = https://www.example.org:443

STREAM(44): DATA                -------->
  Capsule Type = REGISTER_DATAGRAM
  Datagram Format Type = WEBTRANSPORT_DATAGRAM
  Datagram Format Additional Data = ""

           <--------  STREAM(44): HEADERS
                        :status = 200

/* Both endpoints can now send WebTransport datagrams. */
```


Acknowledgments

The DATAGRAM context identifier was previously part of the DATAGRAM frame definition itself, the authors would like to acknowledge the authors of that document and the members of the IETF MASQUE working group for their suggestions. Additionally, the authors would like to thank Martin Thomson for suggesting the use of an HTTP/3 SETTINGS parameter. Furthermore, the authors would like to thank Ben Schwartz for writing the first proposal that used two layers of indirection.

Authors' Addresses

David Schinazi
Google LLC
1600 Amphitheatre Parkway
Mountain View, California 94043,
United States of America

Email: dschinazi.ietf@gmail.com

Lucas Pardue
Cloudflare

Email: lucaspardue.24.7@gmail.com