HTTPR Specification

Draft Proposal

Andrew Banks, Jim Challenger, Paul Clarke, Doug Davis,
Richard P King, Francis Parr, Karen Witting

IBM

Version 1.1 2001-12-03

The authors acknowlege assistance from: Andrew Donoho, Tim Holloway, John Ibbotson, Stephen Todd

NOTICES

IBM may have patents or pending patent applications covering subject matter described in this specification. Non-exclusive licences to such patents are available on reasonable and non-discriminatory terms and conditions to those who respect IBM's intellectual property rights. In addition IBM owns a copyright on this specification. Inquiries regarding patent or copyright licences should be made, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore this statement may not apply to you.

If you supply any information to IBM relating to this specification, you agree that IBM may use or distribute any of that information in any way without incurring any obligation to you.

1. Introduction

1.1. Overview

HTTPR is a protocol for the reliable transport of messages from one application program to another over the Internet, even in the presence of failures either of the network or the agents on either end. It is layered on top of HTTP. Specifically, HTTPR defines how metadata and application messages are encapsulated within the payload of HTTP requests and responses. HTTPR also provides protocol rules which make it possible to ensure that each message is delivered to its destination application exactly once or is reliably reported as undelivered.

Messaging agents use the HTTPR protocol and some persistent storage capability to provide reliable messaging for application programs. This specification of HTTPR does *not* include the design of a messaging agent, *nor* does it say what storage mechanisms should be used by a messaging agent; it does specify what state information needs to be stored safely and when to store it, for a messaging agent to provide reliable delivery using HTTPR.

HTTP/1.1 serves as the base on which HTTPR builds. As such, all of the facilities of HTTP/1.1 (keep-alive, communication through proxies and firewalls, etc.) are available. One feature, the chunked transfer encoding, is especially convenient in the construction of batches of messages where the size of the entire batch is not known a priori. It should not be assumed, however, that this feature, nor any other feature, is actually being used on any particular occasion; any correct use of HTTP/1.1, as defined in RFC 2616, when used by one messaging agent, should be acceptable to any other messaging agent.

Layering HTTPR on HTTP in this way has the additional benefit that HTTPR can be used for reliable messaging with enterprises whose only presence on the Internet is a Web server behind a firewall admitting only Web-related traffic.

Given the asymmetries of HTTP (client connects to server, client sends request, server sends response), it will be convenient to use the terms *client* and *server* even though messaging agents may, in other senses, regard themselves as peers. The agent initiating an HTTPR interaction (the client) does so by sending a POST command, in the HTTP sense, including within its payload an HTTPR command, status information, and, for certain commands, a batch of messages. (A single message is handled as the special case of a batch with only one member.) The server sends back a response, whose payload includes status information and, if the client requested, a batch of messages intended for that client. The messages, and any accompanying meta-data, are uninterpreted bytes as far as HTTPR is concerned and are assigned no other meaning by it.

Messages flow from a source to a sink. Both clients and servers can be sources and sinks. When the client is the source it uses the PUSH command to send messages to the server. When the client is the sink it uses the PULL command to retrieve messages from the server. Each batch of messages is assigned an identifier by its source, which is sent along as HTTPR metadata with the batch. Correctly functioning messaging agents will, in accordance with the specification, store this identifier and the state of their processing of that batch of messages, in stable storage at the appropriate times. In the event of a failure, this information can be recovered from stable storage and used by the messaging agents, through specified interchanges of that state information, to resolve the status of the batch of messages, thereby achieving exactly-once delivery.

The HTTPR protocol places no constraints on the interface used by an application program to pass messages to its local agent implementing HTTPR. SOAP and JMS are two examples of application messaging interfaces for which reliable delivery using the HTTPR protocol can be provided, but we do not specify those, or any other. programming interfaces.

In addition to providing reliable messaging over a "single hop", HTTPR is intended to enable application programs to communicate reliably in a "multihop" environment. Specifically if two application programs are connected by a sequence of messaging agents, each agent uses the HTTPR protocol to exchange messages reliably with its immediately adjacent agents in the sequence and the intervening agents store and forward the messages reliably, then this use of HTTPR will provide reliable end-to-end messaging.

1.2. HTTPR commands

This section provides short, informal descriptions of the function and purpose of each of the HTTPR commands. Note that in addition to the specific information directly related to these commands, HTTPR requests and responses may contain information acknowledging or otherwise related to previous commands. The detailed description of each HTTPR command follows in section 4, "HTTPR Command Specification".

1.2.1. Get Responder Info

This command is used by an HTTPR client to inform an HTTPR server of the client's level of HTTPR protocol capabilities (size limits, etc.). The server is allowed to respond with reduced values that then become the agreed parameters for all following HTTPR interactions for this client/server pair. This sequence of interactions is an HTTPR session. HTTPR sessions are ended by a Report request with end-session indicated. Capability negotiation for HTTPR client/server pairs that are not using HTTPR sessions, is supported by allowing capability information to be included in the requests and responses of all the other commands - Push, Pull, Exchange, and Report.

1.2.2. Push

This command allows an HTTPR client to send a batch of one or more messages to an HTTPR server and get back a response indicating whether this message batch has been received and saved reliably. (The response may also/instead indicate status of previous batches. See the discussion of pipelining in section 1.5.) The batch of messages sent on a Push command is uniquely identified with a transaction identifier. (More precisely, the transaction identifier is unique across the sequence of related interactions between this client and server - defined more formally as a *channel* in subsection 1.4.)

1.2.3. Pull

This command allows an HTTPR client to ask an HTTPR server to send it any messages waiting for delivery to applications located at the client and also to report on the status of previously pulled batches of messages. The response to a pull command may be "empty" (if the server has nothing waiting for delivery to this client) or may include a batch of one or more messages. A

batch of messages returned in the response to a Pull command is uniquely identified with a transaction identifier.

1.2.4. Exchange

The Exchange command combines a Push and a Pull command into a single request. It allows an HTTPR client to send a batch of one or more messages to an HTTPR server. Unlike a Push command, Exchange also invites any waiting messages at the HTTPR server to be returned to the HTTPR client in the response along with the indicator of whether the sent messages were safely received and saved at the HTTPR server. Exchange can sometimes be used to get a single service request delivered and (if the serving process responds quickly enough) to get the reply returned to the initiating HTTPR client in a single HTTPR command flow. However, the HTTPR protocol does not assume any relationship in general between the outgoing and the returning message batches in an Exchange. Both outgoing and returning message batches in an exchange are uniquely identified with transaction identifiers generated by their respective sources.

1.2.5. Report

The Report command enables an HTTPR client to report to an HTTPR server exactly which batches of messages this client has received and saved (from that server); the response from the server allows the client to determine exactly what has been safely received by that server. Use of Report may be prompted either by a communications error, by the need to relieve the server's doubts about messages having been received by the client, or by the need to relieve the HTTPR server from having to continue saving state information about the last interactions with this HTTPR client.

The Report command request includes the unique transaction identifier of the last batch of messages successfully received by this HTTPR client from this server, and the transaction identifier of the last batch of messages sent by this client to this server.

The HTTPR server responds with the unique identifier of the last batch it tried to send to this HTTPR client and discards copies of messages now known to be safely delivered. The server also includes in its response the transaction identifier of the last batch of messages it received from the client. In response to subsequent Pull commands, the server will resend any messages shown by a preceding Report command to have been lost in transit to the client. In response to subsequent Push commands, the HTTPR server will reject any "late arriving" message batches, i.e. those with transaction identifiers showing them to have been sent by the client before it sent the Report, and therefore indicated by the server as not received in its Report response.

Similarly, the client, once it receives the Report response, will rollback its status information on any message batches sent but (according to the response to Report) not received at the HTTPR server.

1.3. HTTPR message structure and datastream

This subsection provides a high level overview of the structure of the HTTPR data stream. The complete and detailed description is provided in section 4, “HTTPR command flows” and section 6, “Header fields”.

The structure of the datastream for simple HTTPR command flows (flows where no HTTPR payload is included) is illustrated below in Figure 1.

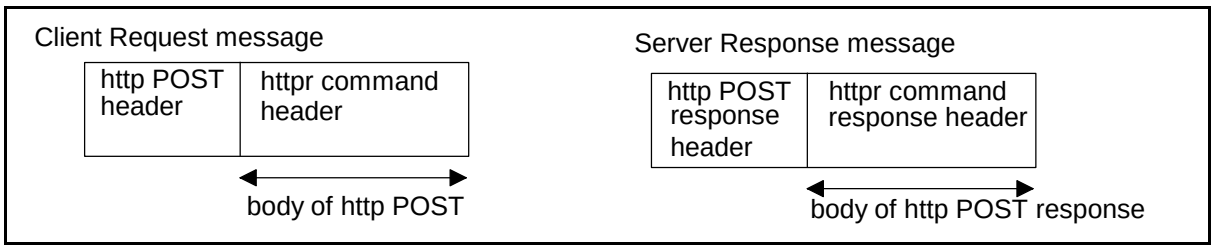


Figure 1: overview of datastream for simple HTTPR command flows

Each HTTPR command flows as the body of an HTTP POST request; the response from the server flows as the body of the POST response.

Some HTTPR commands and responses can also include a payload, which is typically a batch of one or more application messages. The general structure of an HTTPR command or response including a batch of application messages in its payload is illustrated in Figure 2 below.

Following the HTTPR command or response header is a batch of one or more HTTPR application message structures. Each application message structure represents a separate message from some sending application on its way to a specified destination application being passed over this HTTPR channel as one step in its path. The HTTPR client and server messaging agents are responsible for knowing that moving these application messages across this channel is getting each of them either to, or in a multihop path at least one step closer to, their destinations.

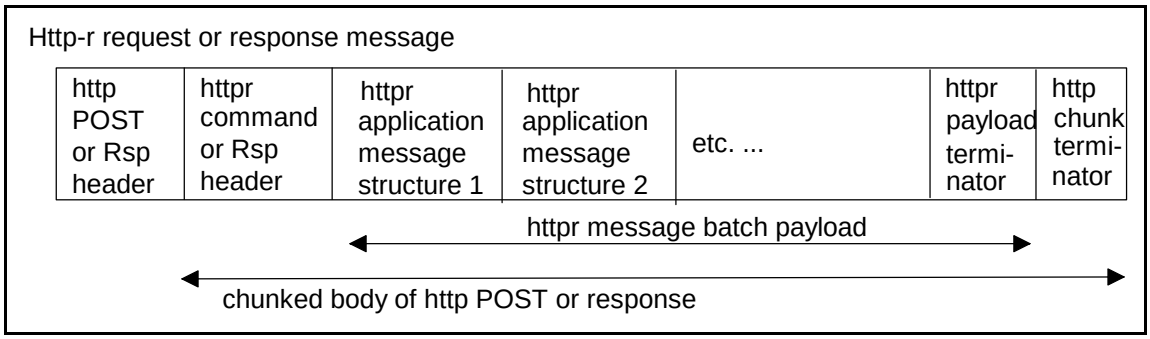


Figure 2. HTTPR command or response with a message batch payload

When an HTTPR command or response message includes a payload containing application messages, the size of the HTTPR message is open ended and potentially large. To ensure that this amount of data can be handled at the HTTPR endpoints and indeed at intermediate nodes in the network, HTTPR messages with payload are often sent using the HTTP chunked transfer encoding. Intermediate nodes in the network may rechunk this data as it passes through them to

meet their needs. Hence HTTPR requests and responses including a payload will usually be terminated with an HTTP data chunk terminator as specified in the HTTP protocol.

The sequence of HTTPR application message structures in a payload is terminated by an HTTPR payload terminator. This has the additional function of indicating *at the end of the payload* whether the sending agent detected some error condition during transmission of the messages which will require the entire message batch to be discarded at the receiver and resent.

Each HTTPR application message structure included in a message batch payload has the form illustrated in Figure 3 below.

The HTTPR application message header includes essential information such as the destination address of this application message and either the length of the application message data or an indication that it will be chunked at the HTTPR level. The application message data is an uninterpreted stream of bytes in which any values are allowed. If the size of an individual application message cannot be determined in advance of sending it (e.g. if it is generated by a stream and may be too large for the sending agent to hold it in memory at one time), it may be encoded as a sequence of HTTPR message data chunks each with their own length indication and with the sequence terminated by a null chunk. This HTTPR level of message data chunking may occur in addition to regular HTTP data chunking of the entire command or response message when, for example, the source of a particular message is a data stream whose size or origin is unknown to the HTTPR agent. (In such a case, the chunking of individual messages allows the receiver to parse the batch into individual messages.) The two levels of chunking will not interfere destructively.

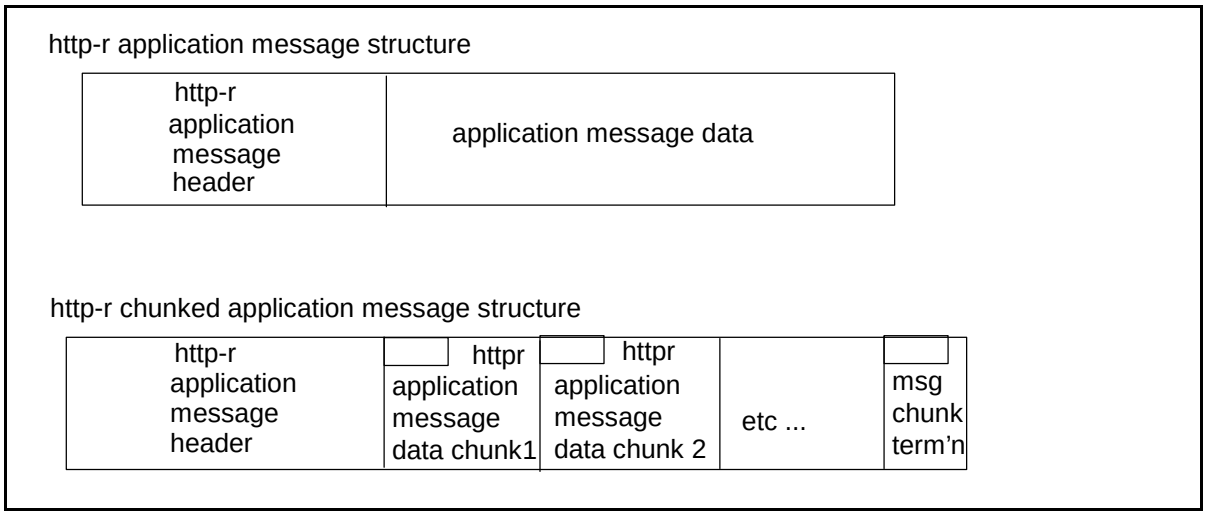


Figure 3. Datastream for HTTPR application message structure

Simple HTTPR agents may choose to use commands with the message batches containing a single message only. Support for message batches in the HTTPR protocol aids scalability and efficiency, bringing the transmission of several messages within the scope of a single unit of work being performed by the persistent stores of the sender, and correspondingly at the receiving end.

1.4. Identifying HTTPR clients, servers and channels.

The HTTPR client and HTTPR server participating in an HTTPR interaction are uniquely and globally identified by their URI's. However, at any point in time there could be several independent HTTPR conversations in progress between a particular HTTPR client and HTTPR server. Each of these conversations might represent a different quality of service and therefore need to be managed and recovered independently from the others. This specification defines the use and behavior of the HTTPR protocol over a single channel. An HTTPR server and client serialize their use of HTTPR command flows for any one channel. When multiple channels are in use between an HTTPR client and HTTPR server, each one follws the protocol rules independently.

Channels are uniquely identified by the ordered triplet:

```
< HTTPR client URI ;  channel identifier ;  HTTPR server URI >
```

This enables the channel identifier to be used to distinguish different qualities of service supported as separate channels between the same client and server. Inclusion of the client and server in the full channel name ensures that this name identifies the channel globally and uniquely. Note also that a channel has only one client initiating requests and one server responding to them. Hence a channel with A as a client and B as a server is necessarily distinct from a channel with A as a server and B as a client.

HTTPR channels, optionally, can be created dynamically without any preconfiguration, by a client sending an HTTPR command to a server and the server accepting it and responding (although some clients or servers may choose not to implement this feature, instead requiring explicit administrative control of channel creation). The channel will continue to exist as long as client and server maintain persistent records protecting the reliable transmission of messages across that channel. The Report command can be used by the client to indicate to the server that no memory of this channel need be retained.

1.5. Levels of HTTPR functionality and capability negotiation

Several functional levels of the HTTPR protocol are defined:

- sessionless
- simple session
- pipelined session

These levels of functionality and other parameters of the HTTPR implementation at the client and server such as timeout values and maximum message sizes are summarized in a *capability vector* which may be included in the HTTPR command and response headers.

Sessionless is the simplest mode for operating an HTTPR channel. In this mode each HTTPR command flow on the channel is independent and carries its own capability negotiation. The HTTPR client may include its capability vector in each command header. If the required capabilities are not supported in the HTTPR server, the command will be rejected. If the server

can accept the HTTPR command but wishes to process it with different, “lesser” capabilities, it processes the request but also returns its capability vector in the response. The intent is that the client should expect to use these lower capabilities on future commands on that channel.

In sessionless mode the command flows for a single channel are carried on a single TCP/IP connection or on a sequence of TCP/IP connections. These connections do not use HTTP pipelining nor do they overlap in any other sense.

Simple session mode HTTPR is a higher level capability in which a session spanning one or more TCP/IP connections is established between HTTPR client and HTTPR server, but pipelining of commands on the session is disallowed. In this mode the GetResponderInfo command is used to set up a session and agree on the exact capabilities to be used by client and server before any transfer or messages is attempted. The agreed upon capabilities will be in use for the lifetime of the session. The HTTPR client can terminate the session using a Report command, thereby allowing the server to free any space occupied by information regarding this session.

Pre-negotiating capabilities allows for better tuning of the communication properties of both the client and the server. It also simplifies error recovery. This provides potential for greater performance and scalability when the extra command flow can be amortized over significant amounts of message transfer traffic within the session.

At any point in time, any one HTTPR channel is either in session based use by exactly one session, or is available for sessionless use. Support of sessions by HTTPR implementations is optional.

In session based HTTPR (as in the sessionless case) the command flows on a single channel may flow over a single TCP/IP connection or over a sequence of TCP/IP connections. However, there is at most one TCP/IP connection asociated with the session at any time. Furthermore, in simple session mode, HTTP pipelining of the HTTPR commands on this connection is disallowed.

Pipelined session mode is the third and most sophisticated mode of HTTPR in which the restriction on HTTP pipelining of commands is removed. It provides potential for further scalability and performance. In this mode the maximum depth of the command pipeline is specified in the capability vector. In some error situations, the HTTPR protocol requires that the current session be terminated and a new session started for communication on this channel to continue. Start of a new session allows messages, which are being resent following an abort and resynchronization of HTTPR client and server, to be distinguished from old HTTPR commands in the HTTP pipeline behind the command which caused the abort.

1.6. Structure of the rest of this document

Section 2 defines the formal notational conventions used in the remainder of the specification.

Section 3 defines in detail common concepts used in several of the command flow definitions

Section 4 provides the formal definition and processing rules for each command flow.

Section 5 provides examples of simple command flows.

Section 6 defines the exact meaning of each of the fields appearing in HTTPR headers

Section 7 defines how to set message header fields for transport of a SOAP request over HTTPR.

Appendix A1 provides a glossary of frequently used terms.

2. Notation Conventions and Generic Grammar

2.1. Requirements

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 .

An implementation is not compliant if it fails to satisfy one or more of the MUST or REQUIRED level requirements for the protocols it implements. An implementation that satisfies all the MUST or REQUIRED level and all the SHOULD level requirements for its protocols is said to be "unconditionally compliant"; one that satisfies all the MUST level requirements but not all the SHOULD level requirements for its protocols is said to be "conditionally compliant."

2.2. Augmented BNF

All of the mechanisms specified in this document are described in both prose and an augmented Backus-Naur Form (BNF) similar to that used by RFC 822. Implementors will need to be familiar with the notation in order to understand this specification. The augmented BNF includes the following constructs:

name = definition

> The name of a rule is simply the name itself (without any enclosing "<" and ">") and is separated from its definition by the equal "=" character. White space is only significant in that indentation of continuation lines is used to indicate a rule definition that spans more than one line. Certain basic rules are in uppercase, such as SP, LWS, HT, CRLF, DIGIT, ALPHA, etc. Angle brackets are used within definitions whenever their presence will facilitate discerning the use of rule names.

"literal"

> Quotation marks surround literal text. Unless stated otherwise, the text is case-insensitive.

rule1 | rule2

> Elements separated by a bar ("|") are alternatives, e.g., "yes | no" will accept yes or no.

(rule1 rule2)

> Elements enclosed in parentheses are treated as a single element. Thus, "(elem (foo | bar) elem)" allows the token sequences "elem foo elem" and "elem bar elem".

*rule

> The character "*" preceding an element indicates repetition. The full form is "<n>*<m>element" indicating at least <n> and at most <m> occurrences of element. Default values are 0 and infinity so that "*(element)" allows any number, including zero; "1*element" requires at least one; and "1*2element" allows one or two.

[rule]

Square brackets enclose optional elements; "[foo bar]" is equivalent to "*1(foo bar)".

N rule

Exact repetition: "<n>(element)" is equivalent to "<n>*<n>(element)"; that is, exactly <n> occurrences of (element). Thus 2DIGIT is a 2-digit number, and 3ALPHA is a string of three alphabetic characters.

#rule

A construct "#" is defined, similar to "*", for defining lists of elements. The full form is "<n>#<m>element" indicating at least <n> and at most <m> elements, each separated by one or more commas (",") and OPTIONAL linear white space (LWS). This makes the usual form of lists very easy; a rule such as:

(*LWS element * (*LWS "," *LWS element))

can be shown as

1#element

Wherever this construct is used, null elements are allowed, but do not contribute to the count of elements present. That is, "(element), , (element) " is permitted, but counts as only two elements. Therefore, where at least one element is required, at least one non-null element MUST be present. Default values are 0 and infinity so that "#element" allows any number, including zero; "1#element" requires at least one; and "1#2element" allows one or two.

; comment

A semi-colon, set off some distance to the right of rule text, starts a comment that continues to the end of line. This is a simple way of including useful notes in parallel with the specifications.

implied *LWS

The grammar described by this specification is word-based. Except where noted otherwise, linear white space (LWS) can be included between any two adjacent words (token or quoted-string), and between adjacent words and separators, without changing the interpretation of a field. At least one delimiter (LWS and/or separators) MUST exist between any two tokens (for the definition of "token" below), since they would otherwise be interpreted as a single token.

2.3. Basic Rules

The following rules are used throughout this specification to describe basic parsing constructs. The US-ASCII coded character set is defined by ANSI X3.4-1986 .

OCTET = <any 8-bit sequence of data>

CHAR = <any US-ASCII character (octets 0 - 127)>

UPALPHA = <any US-ASCII uppercase letter "A".."Z">

LOALPHA = <any US-ASCII lowercase letter "a".."z">

ALPHA = UPALPHA | LOALPHA

DIGIT = <any US-ASCII digit "0".."9">

CTL = <any US-ASCII control character
(octets 0 - 31) and DEL (127)>

CR = <US-ASCII CR, carriage return (13)>

LF = <US-ASCII LF, linefeed (10)>

SP = <US-ASCII SP, space (32)>

HT = <US-ASCII HT, horizontal-tab (9)>

<"> = <US-ASCII double-quote mark (34)>

CRLF = CR LF

HTTP/1.1 header field values can be folded onto multiple lines if the continuation line begins with a space or horizontal tab. All linear white space, including folding, has the same semantics as SP. A recipient MAY replace any linear white space with a single SP before interpreting the field value or forwarding the message downstream.

LWS = [CRLF] 1*(SP | HT)

The TEXT rule is only used for descriptive field contents and values that are not intended to be interpreted by the message parser. Words of *TEXT MAY contain characters from character sets other than ISO-8859-1 only when encoded according to the rules of RFC 2047.

TEXT = <any OCTET except CTLs, but including LWS>

A CRLF is allowed in the definition of TEXT only as part of a header field continuation. It is expected that the folding LWS will be replaced with a single SP before interpretation of the TEXT value.

Hexadecimal numeric characters are used in several protocol elements.

HEX = "A" | "B" | "C" | "D" | "E" | "F" | "a" | "b" | "c" | "d" | "e" | "f" | DIGIT

Many HTTP/1.1 header field values consist of words separated by LWS or special characters. These special characters MUST be in a quoted string to be used within a parameter value (as defined in section 3.6).

token = 1*<any CHAR except CTLs or separators>

separators = "(" | ")" | "<" | ">" | "@"
| "," | ";" | ":" | "\" | <">
| "/" | "[" | "]" | "?" | "="
| "{" | "}" | SP | HT

Comments can be included in some HTTP header fields by surrounding the comment text with parentheses. Comments are only allowed in fields containing "comment" as part of their field value definition.

In all other fields, parentheses are considered part of the field value.

comment = "(" *(ctext | quoted-pair | comment) ")"

ctext = <any TEXT excluding "(" and ")">

A string of text is parsed as a single word if it is quoted using double-quote marks.

quoted-string = (<"> *(qdtext | quoted-pair) <">)

qdtext = <any TEXT except <">>

The backslash character ("\") MAY be used as a single-character quoting mechanism only within quoted-string and comment constructs.

quoted-pair = "\" CHAR

3. Design and Concepts

3.1. Relationship between HTTP and HTTPR

All HTTPR exchanges are carried as an HTTP POST request payload and its response. By definition, proxies MUST NOT cache the result of POST, unless, in HTTP/1.1, the server requests the proxy to cache it, by virtue of explicit indication that the response is publicly cacheable. No specific assumptions are made about the use of HTTP/1.1 headers.

An alternative approach would have been to use the framework for extending HTTP commands to expose the HTTPR specific request field. This would have exposed the protocol detail to web servers, proxies and firewalls in the web, requiring special versions of this software to be deployed to handle HTTPR. To avoid this exposure, and because it is anticipated that HTTPR will be a minority web protocol, this approach was not taken.

3.2. Uniform Resource Identifier

The messaging service represented by a particular agent (client or server) is identified in HTTPR requests using a URI constructed as follows:

```
"httpr://"[host[":"port]"/"] ServiceName
```

The intended recipient of each message is identified by another URI:

```
"httpr://"[host[":"port]"/"] ServiceName#Destination
```

where Destination is interpreted only by that messaging service (and its agents) identified by the portion of that URI to the left of the '#'.

3.3. Unit of Work

An agent sends a number of messages in the HTTP request or HTTP response. These are part of a single unit of work named by the "transactionid" .

```
"request:" "PUSH" Version CRLF
"transactionid:" Transactionid CRLF
```

Once the messages have been sent, the source is in doubt as to whether or not they have been received by the sink. The sink acknowledges receipt and notifies the source as to the outcome of the transaction by using the:

```
["outcome:" Outcome CRLF
"completed:" CompletedTransactionid CRLF]
```

fields. In making the explicit acknowledgment of the last completed transaction the sink is also notifying commitment of all earlier, unacknowledged transactions that have passed on this channel. See the section on Pipelining for more details on this subject.

If, for example, the client is the message source and it is no longer in doubt about its units of work because it has received an outcome for all transactionid's that it generated, the client sends:

```
["forget:" ForgetTransactionid CRLF]
```

as part of a REPORT request. It does this when it wishes to notify the server that it intends to remove all trace of a transaction identifier that it generated. Once the client receives the OK response it can forget all of its state associated with transactions for this channel prior to and including the ForgetTransactionid. The client can send a ForgetTransactionid in sessionless and simple session HTTPR, only when there are no transactions left in doubt on this channel.

Once a ForgetTransactionid has been sent, REPORT will not resolve that particular unit of work. If the ForgetTransactionid is not known to the server, the server should still respond OK, because this might be a repeat flow where an earlier response to the client was lost.

When the client is the sink for server originated messages, it indicates an outcome by flowing:

```
["outcome:" Outcome CRLF
"completed:" CompletedTransactionid CRLF]
```

as above. On receipt, in the server, as well as lifting the in doubt state for the CompletedTransactionid , the server may forget all of the state it has associated with the CompletedTransactionid. If the CompletedTransactionid is not known to the server it should still respond OK, because this might be a repeat flow where an earlier response was lost. The server may also forget all prior transactionid's that it generated.

The asymmetry in the behaviour of client agents versus server agents seen here is the result of the asymmetry inherent in HTTP. The client, by its very nature, is in control of when a server receives a request and of what that request is. Thus, the burden falls on the client to make those requests necessary to allow the appropriate communication of the state of transaction processing.

3.4. Resolution of In-Doubt Transactions

As soon as a client sends a batch of messages using a PUSH request, it is in doubt as to the status of that transaction. Only when a response is received from the server with an outcome of either COMMIT or ROLLBACK for that transaction is the client certain what to do with its copy of those messages. Whenever the client fails to get such a response (when, for example, the network connection to the server is broken before the response is received), the client MUST issue a REPORT request next (or a GET-RESPONDER-INFO request), to solicit information from the server on the status of that transaction. The client MUST NOT proceed with the sending of other messages until this problem has been fixed, as this could otherwise lead to their being received out of their proper order.

As soon as a server sends a batch of messages in the response to a PULL request, it is in doubt as to the status of its transaction. Those doubts will remain until the server receives an appropriate outcome for that transaction from the client. Whenever the client fails to get a response to a PULL request, the client MUST issue a REPORT request next, to provide infomation on what transaction the client *did* receive last, thereby allowing the server to deduce the status of any transactions that had been in doubt. The client MUST NOT proceed with the requesting for

other messages to be sent to it until this problem has been fixed, as the server could otherwise be lead to believe that the lost batch had actually been received, leading to message loss.

3.5. Message Ordering Assumption

The messages delivered to a target application from any one sending application should be received in exactly the same order as they were sent. In one transmission of a batch of messages from one agent to another over HTTPR, the sink agent will indeed see the messages arrive in the same order as the source agent sent them because they are flowing over a single TCP/IP connection and are therefore reliably ordered. If the agent is to present the messages to its application or forward them to other agents in the same order it will need to preserve the order when it stores them. To preserve ordering across a network of agents there must be only one path for the messages through the network, if there are multiple paths then ordering may be lost. For example, allowing two different channels to carry messages between the same source and sink could cause an arbitrary interleaving of messages that flow over those two paths. Since both storage mechanisms and the network configuration are outside the scope of the HTTPR standard, we simply claim that it is possible to build an order preserving messaging network with HTTPR.

3.6. Session Lifetime

A session consists of a sequence of requests and responses; they may flow over a single TCP/IP connection or over a sequence of TCP/IP connections. Sessions are begun by the “session: begin” on a GET-RESPONDER-INFO request and ended by the “session:end” fields. The session ends when the client indicates “session:end” in its REPORT request and receives the response, or when the server indicates “session:end” in its response to any client command.

Support of sessions is optional. The client is not obliged to use sessions; if the server sees a request that requires it to support sessions it can reply with a SESSION-NOT-SUPPORTED error. Not supporting session may simplify the implementation of the agent. Supporting sessions means that less data about the requester and responder identities and their capabilities needs to be carried as part of each request or response. Sessions must be used if pipelining is used.

The client and server retain the set of capabilities they have negotiated until the session ends; if they lose this information they MUST start a new session. Neither the client or server make any commitment to store the negotiated capabilities in a durable way and it is accepted that they will be lost if either of them terminates. If the server fails and loses the negotiated capabilities it should return

```
"error:" "528" "SESSION-IDENTIFIER-NOT-RECOGNISED" CRLF
"session:end" CRLF
```

in its response.

If a session is interrupted, for instance because the TCP/IP connection fails, the client SHOULD attempt to make another connection and then end the session in an orderly way so that the server knows that it need no longer retain the negotiated capabilities. Transaction state MUST be remembered after the session has ended; the agents SHOULD attempt to end sessions only when

they know the partner has no in-doubt transactions and when they know the partner has been able to forget all transaction state.

When the server receives:

```
session:begin
```

It returns a SessionId to the client, the client MUST include the SessionId in all requests that are part of the session so that the server is able to identify the session that the requests relate to. The server uses the SessionId to identify which session the request relates to. The client MUST NOT pipeline requests until it has established a SessionId.

If the sessions are not being used the server is obliged to return

```
"responder:" Responder CRLF
```

On every response it makes so that the client has the opportunity to check the responder identity.

3.7. Capabilities

The capabilities are a set of parameters that govern the way messages are exchanged between the client and server. They indicate limits on timeouts, message and batch sizes, and which commands will be used for message transmission. For example, some servers are only intended for the receiving of messages, and will negotiate a capability indicating that only PUSH is acceptable to it. The capabilities are negotiated using a GET-RESPONDER-INFO request, which MUST be the first request and response of each session. Capabilities in the response to GET-RESPONDER-INFO must be less than or equal to those proposed in the request. Capabilities last for the lifetime of the session. The client MUST NOT assume that the server has the same capabilities or identity as those used in any previous session, for example because the agent administrator may have changed them in the meantime. The client MAY NOT assume its new capabilities are in effect until it has seen a response to its request**.** The capabilities revert to the default values when the session ends or when a new one begins. If a TCP connection breaks without the end of the session being indicated, the client will start a new connection with the intention of indicating the session has ended. This behavior allows both parties to forget the negotiated capabilities. They may forget the negotiated capabilities at other times but they MUST start a new session if they do so.

3.8. Reconnection

If there is any request in progress and the server determines that a new request arrives for the same channel on a different TCP connection, it is an indication that there is something wrong with the first request. The server SHOULD attempt to terminate the first request and proceed with what it has determined to be the later request.

The client MUST NOT process a response to a previous request in another TCP connection once it has made a new connection, if a response does eventually reach it after it has started a new connection it MUST discard the response. This avoids a situation where two requests are outstanding (in separate TCP connections), and network or internal processing delays cause them to be reordered.

There can be multiple connections between a pair of agents as long as a separate channel is used for each one.

3.9. Pipelining

In order to support pipelining an agent must also support sessions. Several requests may be outstanding with the client waiting for a response, as described by HTTP pipelining. Even if persistent HTTP connections are being used the allowed depth of the pipeline may be reduced, by using the capabilities MaximumPipelineDepth, limiting the number of unacknowledged requests and in doubt units of work. The client MUST NOT send its next request message until it has received a response, thereby draining one request from the pipeline, and possibly until it has received a response that includes the "outcome:" and "completed:" fields, thereby draining one or more of the client's in-doubt units of work from the pipeline. If the client attempts to pipeline further requests, the server MAY reject them.

There is an additional constraint for pipelining in HTTPR beyond the constraint imposed by HTTP. The server MUST process requests for each session in the order that it receives them. The client MUST process the responses to its requests associated with each session in the same order that it makes the requests. This is necessary to preserve message sequencing in the event where one of the batches in the pipeline is rolled back.

A well-behaved server will acknowledge completed units of work to the client as soon as it has the opportunity. This will enhance the performance and scalability of HTTPR operation.

3.10. Data conversion

Data conversion is always the responsibility of the sink. The Encoding and Content-type fields describe the format of the incoming message and the sink must convert it into a form that is usable by its applications. The sink MAY choose not to convert a message it is simply forwarding to another agent. There can be a mixture of Encodings and Content-types in a payload.

3.11. Security

HTTPR may flow over SSL connections to achieve point to point authentication and privacy of the messages; no special considerations are necessary to achieve this level of security. If used appropriately, SSL can achieve mutual authentication of the source and sink, privacy and protection of the data exchanged. The HTTPR agent has to behave like any browser on the web. Either it negotiates a secure SSL connection to the agent hosted at ServiceName on port 443 or it gets a proxy to do this on its behalf. Any web server at the remote side would expect legal HTTP, as would a proxy.

Security can also be achieved by running HTTPR over IPSec (with HTTP and TCP in between, as usual). In addition to providing authentication and encryption, IPSec provides secure tunneling of private connections over the public internet without requiring any change to the HTTPR protocol or to the agents implementing it.

The HTTPR protocol does not make provision for end to end authentication and privacy where a message flows over a number of HTTPR links. However, end to end security can be enabled over HTTPR links by suitably encrypting the messages before they are given to the agents.

On the assumption that the HTTPR exchanges are authentic the agents may choose to impose their own access control over the resources used by using the UserId from the payload.

3.12. State Information

In order for an HTTPR agent to function correctly it must save state information persistently. The state information that must be stored can be divided into source information and sink information. Whether a client is acting as a message source (using PUSH) or a server is providing messages (responding to PULL), the same source state information must be maintained. Similarly, the same sink information must be maintained whether the client or server is receiving messages. We provide a description of one way to represent this information and, in terms of that representation, indicate when and how the information would be updated so that correct functioning can be achieved. This is not a definition of what must be done, but, rather, an illustrative suggestion. See, also, the examples in section 4, for diagrams showing how the updates of persistent storage relate to the sending and receiving of HTTPR requests and responses.

3.12.1. Source State Information

The source is responsible for persistently saving any messages that have not yet been safely received by the message sink as well as status information regarding the source's attempts to send messages. The status information can be represented as follows.

- **last-source-id**: the transaction-id for the last batch of messages sent by this source
- **indoubt-ids**: those transaction-ids not known to have been received by the sink
- **indoubt-messages**: which messages are in doubt, and which transaction-id relates to each

The source is free to use whatever transaction-id values it chooses, as long as they are strictly increasing (see section 6.1.8. Transactionid), but initially the source should set the last-source-id and last-ack-id values to 0, while last-ack-outcome starts as COMMIT.

As part of sending a batch of messages, a new transaction-id is generated and last-source-id is set to that value. The messages in the batch are marked as being part of that transaction and in doubt as to whether the sink has received them. All of these updates to persistent storage should be committed as a single transaction just before sending out the payload terminator that completes the batch transmission.

When the source receives an outcome from the sink, the related messages can be removed (for an outcome of COMMIT) or marked as available to be sent again (on ROLLBACK) or some other recovery action may be taken (in the case of INDOUBT). When using sessions, where there can be multiple indoubt transactions, if any messages related to smaller (earlier) transaction-ids are still in doubt, those messages can be removed, as though a COMMIT outcome had been received for them. (No ROLLBACK from the sink could have been lost, as that would violate the

session-breaking rules related to ROLLBACK.) Also, if there are any indoubt transactions with greater (more recent) ids, those should be marked as available again (on any ROLLBACK or on COMMIT as part of a REPORT).

The source will pass last-source-id to the sink when the status of a transaction needs to be determined. When the source is a client it would use the REPORT command and specify last-pushed-id = last-source-id. When the source is a server it would respond to the REPORT command setting last-pulled-id=last-source-id. These actions will allow the sink to avoid accepting late arriving transactions, since the source promises never to use any smaller transaction id value in the future.

3.12.2. Sink State Information

The sink is responsible for safely storing messages received and keeping sufficient state information to determine when a batch of messages has been lost or has arrived late. Aside from the messages themselves, this can be represented by the following variables.

- **last-received-id**: the transaction-id most recently received from the message source
- **last-received-outcome**: how that last transaction was disposed of here at the sink
- **last-sent-id**: the largest transaction-id known to have been used by the message source

Initially last-received-id and last-sent-id are set to 0, while last-received-outcome is COMMIT.

When the sink receives a batch of messages it verifies that the transaction-id assigned them by the source is greater than both the last-received-id and last-sent-id. The message are placed in persistent storage and last-received-id is updated to contain the newly received transaction-id. These updates to persistent storage would be commited before acknowledgement is returned to the sender, since the sender is already committed to the sending of these messages and they should be made available to applications on the receiving side as quickly as possible.

Of course, if the payload disposition is abort, the sink can abort these updates to persistent storage. This leaves the sink without the newly updated values in persistent storage. In the event of a failure of the sink, recovery to the previously saved state will allow a misbehaving source to incorrectly reissue a transaction-id without complaint from the sink. This causes no injury to the sink, or to any correctly operating source and, since this behavior is (nearly) indistinguishable from a sink that failed before processing the request in the first place, this is not a major concern.

last-sent-id is updated when the message source provides that information on a REPORT request (by a client source) or response (by a server source).

When a client acting as a source provides, as part of the REPORT command, a value for “forget” which matches last-received-id, then the server can forget all the values, effectively setting them back to their initial values.

4. Example message flow scenarios

This sections shows some typical HTTPR interactions. Messages between HTTPR agents will flow either from the client to the server, from the server to the client, or both.

4.1. Moving messages from the client to the server

The PUSH HTTPR command is used to send message from the client to the server. The client initiates a PUSH command by sending the command headers, along with a batch of messages, to the server. The server responds to the PUSH with status of the batch of messages sent. Under special, unusual, circumstances the server may not choose to respond to the PUSH command with the status of the messages sent.

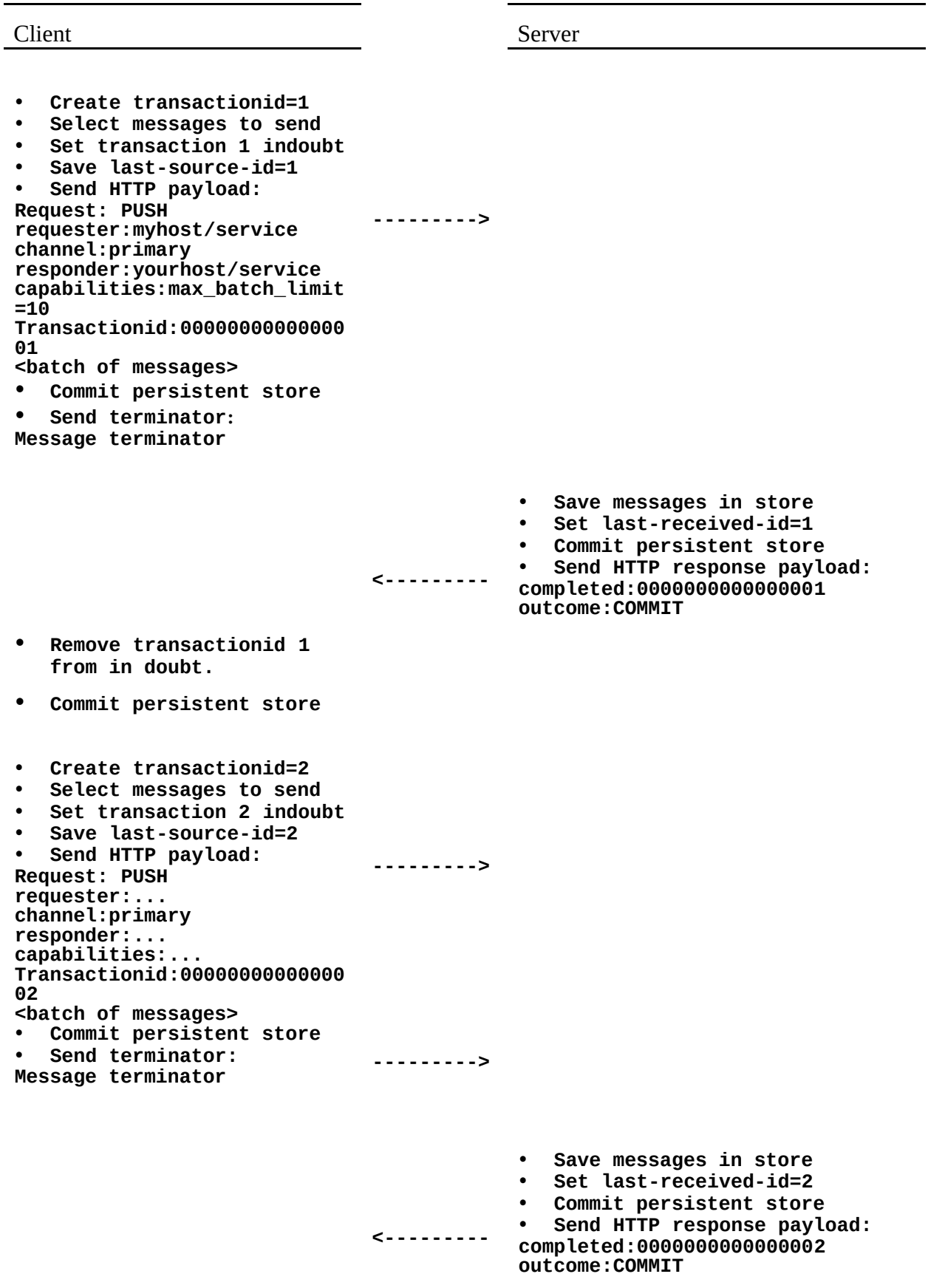
4.1.1. PUSH (without session)

In this interaction the client chooses to use a sessionless PUSH command to send a batch of messages.

In preparing to send the first batch of messages, the client assigns a unique transaction id (1), sets the messages to be sent to be associated with that transaction id, and puts the transaction id in doubt. After saving its state, with last-source-id = 1, the client sends the HTTPR command. Upon receiving the request the server reads in the batch of messages and saves them in its persistent mechanism. It also keeps a record of the last transaction id received from the client, last-received-id. Once this information is successfully saved in the persistent store, the server returns a response indicating that the messages have been committed. On subsequent requests the client assignes a new, unique, transaction id, saves updated information in its persistent store, and sends the request. The server updates its own information, including committing the messages, and responds with an indicator that the messages were successfully received.

When the client decides it will not be sending any more messages it may choose to send a REPORT, allowing the server to clean up its persistent store information associated with this channel.

The server will respond to the "forget" header by removing transaction information about messages received on this channel. If the client did not issue the REPORT with forget the server would be oblidged to keep the incoming transaction information indefinately.


Client
Server
• Create transactionid=1
• Select messages to send
• Set transaction 1 indoubt
• Save last-source-id=1
• Send HTTP payload:
Request: PUSH
requester:myhost/service
channel:primary
responder:yourhost/service
capabilities:max_batch_limit
=10
Transactionid:00000000000000
01
<batch of messages>
• Commit persistent store
• Send terminator:
Message terminator
--------->
• Save messages in store
• Set last-received-id=1
• Commit persistent store
• Send HTTP response payload:
completed:0000000000000001
outcome:COMMIT
<---------
• Remove transactionid 1
from in doubt.
• Commit persistent store
• Create transactionid=2
• Select messages to send
• Set transaction 2 indoubt
• Save last-source-id=2
• Send HTTP payload:
Request: PUSH
requester:...
channel:primary
responder:...
capabilities:...
Transactionid:00000000000000
02
<batch of messages>
• Commit persistent store
• Send terminator:
Message terminator
--------->
--------->
• Save messages in store
• Set last-received-id=2
• Commit persistent store
• Send HTTP response payload:
completed:0000000000000002
outcome:COMMIT
<---------

```
• Remove transaction 2 from
  indoubt
• Commit persistent store
• Detect that communication
  on this channel is
  complete                    --------->
• Send HTTP payload:
Request: REPORT
requester:...
channel:primary
reponder:..
forget:0000000000000002
```

```
                                          • Remove last-received-tid value
                                          • Remove channel information, if
                                            appropriate
                                          • Send HTTP response payload:
                              <---------  last-pulled-id: 0000000000000000
• Remove last-source-id
  value
```

4.2. Client initiates PUSH, using a session

The client uses simple session HTTPR to push messages to the server which makes them available for processing to applications located there. When the client has no more messages to send, it terminates the session.

Client		Server
GET-RESPONDER-INFO **requester:me** **channel:primary** **responder:you** **capabilities:xxx** **session:begin**	**---------->**	
	<----------	**completed:0000000000000000** **outcome:COMMIT** **sessionid:1**
• **Create transactionid=1** • **Select messages** • **Set transaction 1 indoubt** • **Save last-source-id=1** • **Send HTTP payload:** Request: PUSH sessionid: 1 transactionid: 1 <batch of messages>	**---------->**	
• **Commit persistent store** • **Send terminator:** Message terminator		
		• **Save messages in store**

```
                                        • Set last-received-id=1
                                        • Commit persistent store
                                        • Send HTTP response:
                          <----------   completed:1
                                        outcome:COMMIT
• Remove transaction 1 from
  in doubt state
• Commit persistent store

• Create transactionid=2
• Select messages
• Set transaction 2 indoubt
• Save last-source-id=2
• Send HTTP payload:
Request: PUSH             ---------->
sessionid:1
transactionid:0000000000000002
<batch of messages>
• Commit persistent store
• Send terminator:
Message terminator
                                        • Save messages to store
                                        • Set last-received-id=2
                                        • Commit persistent store
                          <----------   completed: 0000000000000002
                                        outcome:COMMIT
• Remove transaction 2 from
  in doubt
• Commit persistent store

• Detect that communication
  on this channel is complete
• Send HTTP payload:
Request: REPORT           ---------->
sessionid: 1
forget: 0000000000000002
sessionid:1
session:end
                                        • Remove last-received-id
                                          value
                                        • Remove channel information
                                        • Send HTTP response:
                          <----------   last-pulled-id:0000000000000000
• Remove last-source-id value
```

4.3. Client fails then restarts PUSH

Continuing an alternative path for the last example in the case where the first PUSH failed:

Client	Server
• Create transactionid=1 • Select messages • Set transaction 1 indoubt	

```
• Save last-source-id=1
• Send HTTP payload:
Request: PUSH                    ---------->
sessionid: 1
transactionid: 0000000000000001
<batch of messages>
• Commit persistent store
• Send terminator:
Message terminator               ------------>
                                                • Save messages
                                                • Set last-received-id=1
                                                • Commit persistent store
                                                • Send HTTP response:
                                 Failure<---    completed:0000000000000001
                                                outcome:COMMIT
```

After the failure the client, does not receive the expected response and so MUST establish whether the previous PUSH was committed or rolled back, before it can continue with other PUSH, PULL or EXCHANGE requests. The client chooses to start a new session to accomplish this, then it sends REPORT and ends the session.

Client		Server
(Transactionid=1 in doubt) **GET-RESPONDER-INFO** **requester:me** **channel:primary** **responder:you** **capabilities:xxx** **session:begin**	**---------->**	
	<----------	**completed:0000000000000001** **outcome:COMMIT** **sessionid:2**
• **Remove transaction 1 from indoubt (commit)** • **Send HTTP payload:** **Request: REPORT** **sessionid:2** **session:end** **forget:0000000000000001**	**---------->**	
		• **Remove last-received-tid value** • **Remove channel information** • **Send HTTP response payload:**
	<----------	**last-pulled-id: 0000000000000000**
• **Remove last-source-id value**		

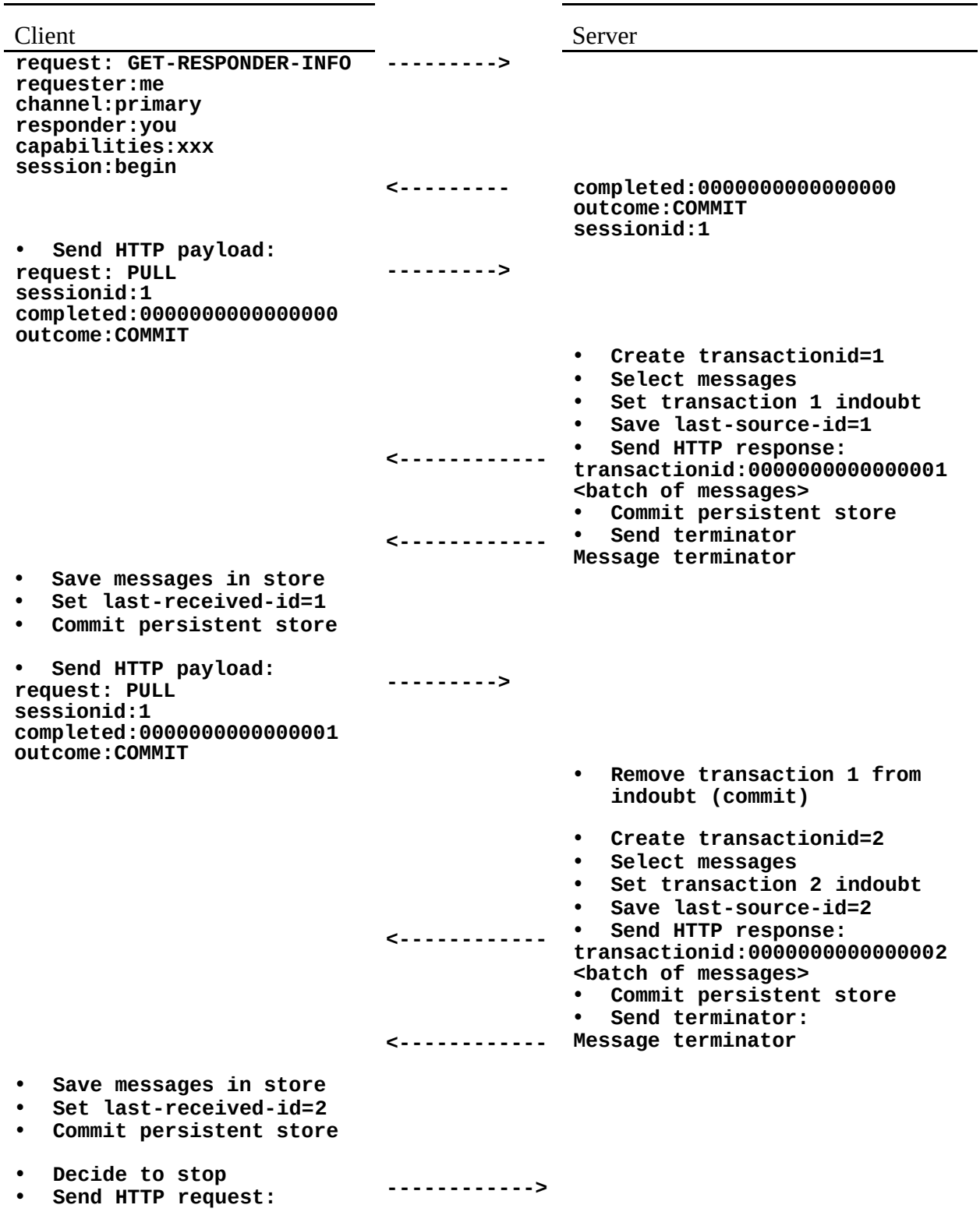
4.4. Client initiates PUSH, server rolls back.

The client uses simple session HTTPR to push messages to the server but the server's resource manager's disk is full so the Push request is aborted.

Client		Server
• Create transactionid=1 • Select messages • Set transaction 1 indoubt • Save last-source-id=1 • Send HTTP payload: request: PUSH sessionid:1 transactionid:0000000000000001 <batch of messages> • Commit persistent store • Send terminator Message terminator	---------->	(Disk is full.)
	<----------	• Rollback persistent store • Send HTTP response: completed:0000000000000001 outcome:ROLLBACK
• Restore indoubt messages • Remove transaction 1 • Create transactionid=2 • Set last-source-id=2 • Send HTTP payload: request: PUSH sessionid: 1 transactionid:0000000000000002 <more or less same messages> • Commit persistent store • Send terminator: Message terminator	---------->	(Disk becomes empty.)
	<----------	• Save messages • Set last-received-id=2 • Commit persistent store • Send HTTP response: Completed:0000000000000002 outcome:COMMIT
• Remove transaction 2 from indoubt (commit) • Detect that communication on this channel is complete • Send HTTP payload: request: REPORT sessionid:1 session:end forget:0000000000000002	---------->	
	<----------	• Remove last-received-tid • Remove channel info • Send HTTP response: last-pulled-id: 0000000000000000
• Remove last-source-id value		

4.5. Client initiates PULL

The client uses simple session HTTPR to pull messages from the server and make them available to for processing to applications located at the client.



```
request: REPORT
sessionid:1
session:end
completed:0000000000000002
outcome:COMMIT
                                                  • Remove transaction 2 from
                                                    indoubt (commit)
                                                  • Remove last-source-id value
                                  <------------   • Send HTTP response:
                                                  last-pulled-id:0000000000000000

• Remove last-received-tid
  value
```

The CompletedTransactionid flowed in the PULL message indicates to the server that it can now forget the state of that transaction and remove any messages associated with it. This will be the transactionid received as the response to a previous PULL. The client may discard the state of the transaction once it receives a new transactionid from the server in response to a PULL request.

The client flows a REPORT request on receipt of a response from a previous PULL without delay. This is so that the responder is not left in doubt as to whether the messages have been received, for any longer than necessary.

4.6. Client fails then restarts PULL

Client		Server
• Send HTTP payload: request: PULL sessionid:1	------------>	
		• Create transactionid=1 • Select messages • Set transaction 1 indoubt • Save last-source-id=1
	<------------	• Send HTTP response: transactionid:0000000000000001 <batch of messages>
		• Commit persistent store
	<------------	• Send terminator: Message terminator
• Save messages in store • Set last-received-id=1 • Commit persistent store		
• Send HTTP payload: request: PULL sessionid: 1 completed:0000000000000001	------------------>	
		• Remove transaction 1 from indoubt (commit) • Create transactionid=2 • Select messages • Set transaction 2 indoubt • Save last-source-id=2 • Send HTTP response:

```
                                <------------   transactionid:0000000000000002
                                                <batch of messages>
                                                •  Commit persistent store
                                                •  Send terminator:
                                Failure<---     Message terminator
```

After the failure the client sends the last PULL request again.

Client		Server
		(Transactionid=2 Still In Doubt.)
GET-RESPONDER-INFO requester:me channel:primary responder:you capabilities:xxx session:begin	---------->	
	<----------	completed:0000000000000000 outcome:COMMIT sessionid:2
request: REPORT sessionid:2 completed:0000000000000001 outcome:COMMIT	---------->	
	<----------	• Restore transaction 2's messages • Remove transaction 2 • Commit persistent store • Send HTTP response: last-pulled-id:0000000000000002
• Set last-sent-id=2 • Send HTTP request: request: PULL sessionid:2	---------->	
	<------------	• Create transactionid=3 • Select (same) messages • Set transaction 3 indoubt • Send HTTP response: transactionid:0000000000000003 <(same) batch of messages> • Commit persistent store • Send terminator:
	<------------	Message terminator
• Save messages in store • Set last-received-id=3 • Commit persistent store • Decide to stop • Send HTTP request: request: REPORT sessionid:2 completed:0000000000000003 outcome:COMMIT	------------>	
		• Remove transaction 3 from indoubt (commit)

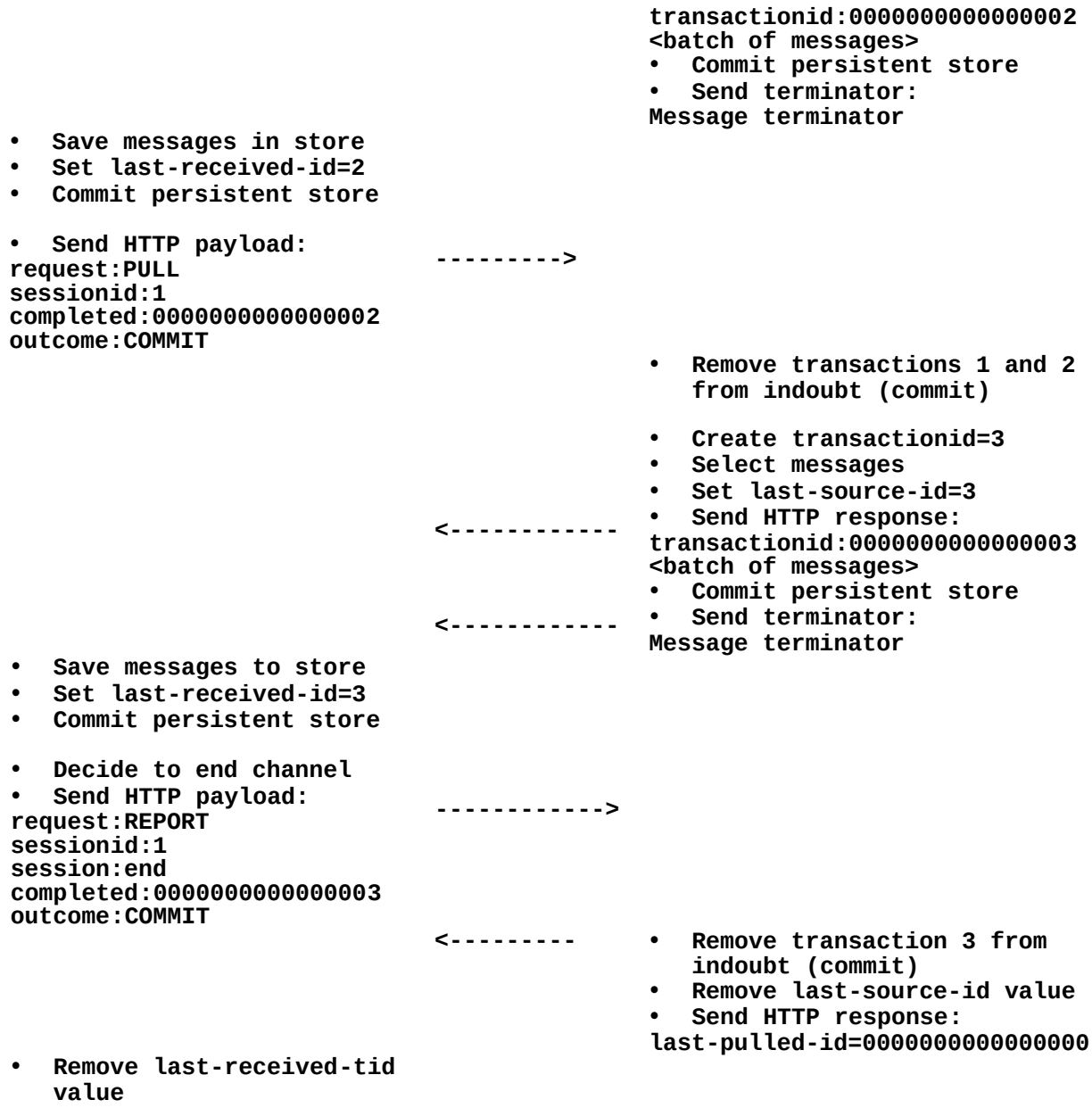
```
                                                    •  Remove last-source-id value
                                                    •  Send HTTP response:
                                <----------         last-pulled-id:0000000000000000
                                                    session:end
•  Remove last-received-id
   value
```

Note that because the unit of work associated with transactionid=2 was rolled back and then prepared again, the new transactionid is set to 3, underscoring the fact that the new payload may not be identical to the one that was sent at the earlier failure.

4.7. Client initiates pipelined PULL

The client uses pipelines session HTTPR to pull messages from the server and make them available for processing to applications located at the client . It uses pipelining in order to achieve improved throughput so that the flow of messages from the server is not interrupted by the need to wait for client requests to arrive. All of the flows here must be completed as one pipelined sequence of HTTP requests and responses, in a single unbroken TCP connection. This is to prevent any possibility of requests and responses being missing or out of sequence. When the client se it does not wish to receive any more messages, it terminates the session allowing the server to forget the last transactionid.

Client		Server
request:GET-RESPONDER-INFO requester:me channel:primary responder:you capabilities:xxx session:begin	--------->	
	<---------	completed:0000000000000000 outcome:COMMIT sessionid:1
request:PULL sessionid:1	--------->	
request:PULL sessionid:1	--------->	
		• Create transactionid=1 • Select messages • Set transaction 1 indoubt • Save last-source-id=1 • Send HTTP response:
	<---------	transactionid:0000000000000001 <batch of messages>
		• Commit persistent store • Send terminator: Message terminator
• Save messages in store • Set last-received-id=1 • Commit persistent store		
		• Create transactionid=2 • Select messages • Set transaction 2 indoubt • Save last-source-id=2
	<---------	• Send HTTP response:



The COMMIT of transactionid=1 in the client is implied when the CompletedTransactionid=2 request is processed in the server.

4.8. Client initiates EXCHANGE

The client uses simple session HTTPR and an Exchange command. This is equivalent to first pushing messages to the server, which makes them available for processing to applications located there, and then pulling any available reply messages to make them available for processing to applications located back at the client. The pulled reply messages may be generated by application level processing of messages pushed to the server in the first part of the

Exchange flow, or they may be any other messages from other sources waiting to be returned to the client on this channel. The client sets its batch size and batch interval to restrict how long it waits for reply messages and to limit how many messages it expects.

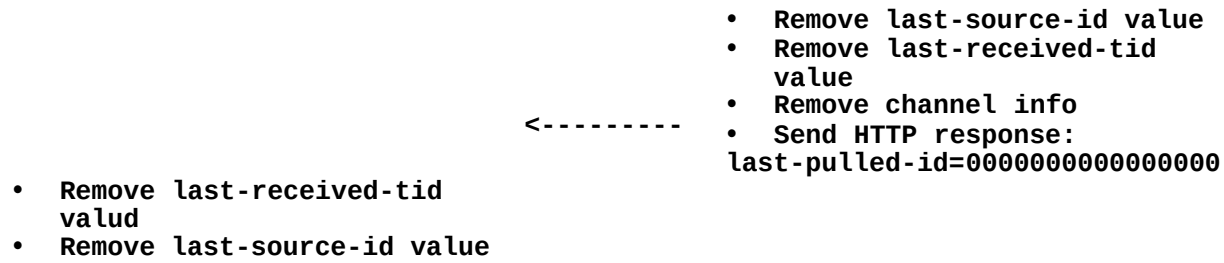
Client		Server
GET-RESPONDER-INFO **requester:me** **channel:primary** **responder:you** **capabilities:xxx** **session:begin**	**---------->**	
	<---------	**completed:0000000000000000** **outcome:COMMIT** **sessionid:1**
• **Create transactionid=1** • **Select messages** • **Set transaction 1 indoubt** • **Send HTTP payload:** **request:EXCHANGE** **sessionid:1** **transactionid:0000000000000001** **<batch of messages>** • **Commit persistent store** • **Send terminator:** **Message terminator**	**---------->**	
	<----------	• **Save messages to store** • **Set last-received-id=1** • **Commit persistent store** • **Create transactionid=101** • **Select messages** • **Set transaction 101 indoubt** • **Save last-source-id=101** • **Send HTTP response:** **transactionid:0000000000000101** **completed:0000000000000001** **outcome:COMMIT** **<batch of messages>** • **Commit persistent store** • **Send terminator:** **Message terminator**
• **Remove transaction 1 from indoubt (commit)** • **Save messages to store** • **Set last-received-id=101** • **Commit persistent store** • **Create transactionid=2** • **Select messages** • **Set transaction 2 indoubt** • **Save last-source-id=2** • **Send HTTP payload:** **request:EXCHANGE** **sessionid:1** **transactionid:0000000000000002** **completed:0000000000000101** **outcome:COMMIT** **<batch of messages>**	**---------->**	

```
• Commit persistent store
• Send terminator:              ---------->
Messages terminator

                                                • Remove transaction 101 from
                                                  indoubt (commit)
                                                • Save messages to store
                                                • Set last-received-id=2
                                                • Commit persistent store
                                                • Create transactionid=102
                                                • Select messages
                                                • Set transaction 102 indoubt
                                                • Save last-source-id=102
                                                • Send HTTP response:
                                <----------     transactionid:0000000000000102
                                                completed:0000000000000002
                                                outcome:COMMIT
                                                <batch of messages>
                                                • Commit persistent store
                                                • Send terminator:
                                <----------     Message terminator
• Remove transaction 2 from     ---------->
  indoubt (commit)
• Save messages in store
• Set last-received-id=102
• Commit persistent store

• Decide to stop
• Send HTTP request:
request:REPORT
sessionid:1
session:end
completed:0000000000000102
outcome:COMMIT
forget:0000000000000002

                                                • Remove transaction 102 from
                                                  indoubt (commit)
                                                • Remove last-source-id value
                                                • Remove last-received-tid
                                                  value
                                                • Remove channel information
                                                • Send HTTP response:
                                <----------     last-pulled-id:0000000000000000
• Remove last-source-id value
• Remove last-received-tid
  value
```

4.9. Client PUSH followed by PULL

The client PUSHes a message to the server and then issues a PULL request to receive the application message it expects as a result of the one it just PUSHed.

Client		Server
GET-RESPONDER-INFO **requester:me** **channel:primary** **responder:you** **capabilities:xxx** **session:begin**	**--------->**	
	<---------	**completed:0000000000000000** **outcome:COMMIT** **sessionid:1**
• **Create transactionid=1** • **Select messages** • **Set transaction 1 indoubt** • **Save last-source-id=1** • **Send HTTP payload:** **request:PUSH** **sessionid:1** **transactionid=0000000000000001**	**--------->**	
	<--------	• **Save messages to store** • **Set last-received-id=1** • **Commit persistent store** • **Send HTTP response:** **completed:0000000000000001** **outcome:COMMIT**
• **Remove transaction id from indoubt (commit)**		
• **Send HTTP payload:** **request:PULL** **sessionid:1**	**--------->**	
	<---------	• **Create transactionid=101** • **Select messages** • **Set transaction 101 indoubt** • **Save last-source-id=101** • **Send HTTP response:** **transactionid:0000000000000101** **<batch of messages>** • **Commit persistent store** • **Send terminator:** **Message terminator**
• **Save messages in store** • **Set last-received-id=101** • **Commit persistent store**	**--------->**	
• **Decide to stop** • **Send HTTP payload:** **request:REPORT** **sessionid:1** **session:end** **completed:0000000000000101** **outcome:COMMIT** **forget:0000000000000001**		
		• **Remove transaction 101 from indoubt (commit)**


• Remove last-source-id value
• Remove last-received-tid value
• Remove channel info
• Send HTTP response: last-pulled-id=0000000000000000
<---------
• Remove last-received-tid valud
• Remove last-source-id value

5. HTTPR Command Specification

This chapter describes the HTTPR commands, giving their request and response structures. Details on the definition and meaning of individual headers fields will be found in Section 6, “Header Fields”.

The HTTPR commands may be used in any sequence on any single HTTPR channel. Each HTTPR interaction consists of a client request sent from the client to the server and a response sent from the server to the client. The HTTPR request is passed in the entity body of an HTTP POST; the HTTPR response goes in the entity body of the corresponding HTTP response. HTTPR requests related to a single HTTPR channel may pass over the same TCP connection or on a series of TCP connections, except where this is limited by specific pipelining or session requirements.

In the following, the use of the HTTP command POST and related HTTP request and response headers are shown. The explicit use of these headers does not imply that no other HTTP headers may be used; we have only included those headers relevant to the demonstration of how HTTPR relates to HTTP. In the case of transfer-encoding, we have included that header in preference to content-length on certain requests and responses not because it is necessary for the HTTPR protocol, but because it is the more likely to be used, given that those requests/responses are amenable to dynamic construction on the fly.

5.1. GET-RESPONDER-INFO

Used to begin a session, to agree on capabilities, and to determine the responder identity associated with a URL. No message payload is carried in either direction. If it is used, it is always the first flow of a session; subsequent flows of the session will carry the sessionid returned in the response to this command. For channels not using GET-RESPONDER-INFO (and, hence, being sessionless), each command flow MUST include, in the request, the full identification of the channel and the client MUST wait to receive the servers response before sending another request on the same TCP connection.

As with the REPORT request, an Outcome and CompletedTransactionid can be exchanged to allow in doubt messages sent on this channel in a previous session to be resolved. However the client may defer sending this information until a later stage if it wants to discover the server identity in the returned Responder field first.

- Client request message.

```
"POST" /ServiceName "HTTP/1.1" CRLF
["Host:"host[":port]" CRLF]
["Content-length:" length CRLF]
CRLF
"request:GET-RESPONDER-INFO" Version CRLF
"requester:" Requester CRLF
```

```
"channel:" Channel CRLF
["responder:" Responder CRLF]
"session:begin" CRLF
["capabilities:" Capabilities CRLF]
["agent-type:" AgentType CRLF]
["outcome:" Outcome CRLF
 "completed:" CompletedTransactionid CRLF]
*[ProductSpecificField CRLF]
CRLF
```

- Server response message.

```
"HTTP/1.1" "200" "OK" CRLF
CRLF
HTTPR/1.0 CRLF
"responder:" Responder CRLF[ "session:end" CRLF
 |"sessionid:" SessionId CRLF]
["capabilities:" Capabilities CRLF]
["agent-type:" AgentType CRLF]
"outcome:" Outcome CRLF
"completed:" CompletedTransactionid CRLF
["error:"  ErrorNumber  ErrorText  CRLF]
*[ProductSpecificField CRLF]
CRLF
```

The response will include a SessionId unless there is an error, in which case it will, instead, indicate the session is ended. Capabilities will be included in the response if the server decided to negotiate them down from the defaults or, if present on the request, from those proposed by the client. If the server does not support the Session capability, it will both indicate that the session is ended and return capabilities showing support only for Sessionless interaction.

5.2. PUSH

The client wishes to transfer application messages to the server but does not want to receive any application messages in return.

- Client request message.

```
"POST" /ServiceName "HTTP/1.1" CRLF
["Host:"host[":port]" CRLF]
```

```
["Transfer-Encoding:" "chunked" CRLF]
CRLF
1*HEX CRLF                          ; size of 1st HTTP chunk
"request:" "PUSH" Version CRLF
(("requester:" Requester CRLF
 "channel:" Channel CRLF
 ["responder:" Responder CRLF]
 ["capabilities:" Capabilities CRLF])
|"sessionid:" SessionId CRLF)
"transactionid:" Transactionid CRLF
["outcome:" Outcome CRLF
 "completed:" CompletedTransactionid CRLF]
["agent-type:" AgentType CRLF]
["error:"  ErrorNumber  ErrorText  CRLF]
[ProductSpecificField CRLF]
CRLF
*Payload
Terminator
0 CRLF
CRLF
```

Inclusion of an outcome on the request is to allow the client to indicate arrival of messages received in response to previous PULL or EXCHANGE commands.

- Server response message.

```
"HTTP/1.1" "200" "OK" CRLF
["Content-length:" length CRLF]
CRLF
["responder:" Responder CRLF]
["session:end" CRLF]
["outcome:" Outcome CRLF
 "completed:" CompletedTransactionid CRLF]
["agent-type:" AgentType CRLF]
["error:"  ErrorNumber  ErrorText  CRLF]
*[ProductSpecificField CRLF]
CRLF
```

The Outcome indicates that the server has committed, rolled back, or does not know what happened to the unit of work the client sent. It may be omitted only when requests are being pipelined within a session. If the server did not return an outcome of in doubt, the client may commit or rollback the messages, as indicated by the outcome, and forget the CompletedTransactionid. All of the prior units of work, if any had been pipelined and left unmentioned in previous responses, are assumed by the client to have been committed in the server.

If an INDOUBT outcome has been returned in the response (caused, for example, by a failure of its database subsystem), the server MUST also end the session, if any, without processing any further requests for this channel. The client may choose to restart the communications and restart the message flow, using the REPORT request to try to determine the outcome of the transaction. This is necessary to maintain message sequencing, as the client must now reestablish which was the last unit of work that was committed by the server. This behavior is preferable to the server simply waiting until it knows the outcome of the transaction, because the client will now also know immediately that there is a problem at the server and that there will probably be some delay while the problem is resolved.

5.3. PULL

The client is inviting the server to send it messages. In the request the client may acknowledge receipt and commitment, rollback of messages it received from the server in prior requests. The acknowledgment of a unit of work implies commitment of all of the previous unacknowledged units of work on this channel. If the client wishes to notify the server of in doubt state it should use a REPORT request to achieve this.

- Client request message

```
"POST" /ServiceName "HTTP/1.1" CRLF
["Host:"host[":port]" CRLF]
["Content-length:" Length]
CRLF
"request:" "PULL" Version CRLF
(("requester:" Requester CRLF
 "channel:" Channel CRLF
 ["responder:" Responder CRLF]
 ["capabilities:" Capabilities CRLF])
|"sessionid:" SessionId CRLF)
["outcome:" Outcome CRLF
 "completed:" CompletedTransactionid CRLF]
["agent-type:" AgentType CRLF]
```

```
["error:"  ErrorNumber  ErrorText  CRLF]
*[ProductSpecificField CRLF]
CRLF
```

Inclusion of an outcome on the request is to allow the client to indicate arrival of messages received in response to previous PULL or EXCHANGE commands.

- Server response when it has messages to return to the client

```
"HTTP/1.1" "200" "OK" CRLF
["Transfer-Encoding:" "chunked" CRLF]
CRLF
1*HEX CRLF                          ; size of 1st chunk
["responder:" Responder CRLF]
["session:end" CRLF]
"transactionid:" Transactionid CRLF
["outcome:" Outcome CRLF
 "completed:" CompletedTransactionid CRLF]
["agent-type:" AgentType CRLF]
["error:"  ErrorNumber  ErrorText  CRLF]
*[ProductSpecificField CRLF]
CRLF
*Payload
Terminator
0 CRLF
CRLF
```

- Or, if the Server has no messages to return:

```
"HTTP/1.1" "200" "OK" CRLF
["Content-length:" length] CRLF
["responder:" Responder CRLF]
["session:end" CRLF]
["outcome:" Outcome CRLF
 "completed:" CompletedTransactionid CRLF]
["agent-type:" AgentType CRLF]
["error:"  ErrorNumber  ErrorText  CRLF]
*[ProductSpecificField CRLF]
CRLF
```

Inclusion of outcome on the response is only optional when commands are pipelined within a session.

5.4. EXCHANGE

The client wishes to transfer messages to the server and wishes to receive messages. It is a PULL piggybacked with a PUSH; all comments about those commands therefore apply here.

- Client request message.

```
"POST" /ServiceName "HTTP/1.1" CRLF
["Host:"host[":port]" CRLF]
[Transfer-Encoding:" "chunked" CRLF]
CRLF
1*HEX CRLF                          ; size of 1st chunk
"request:" "EXCHANGE" Version CRLF
(("requester:" Requester CRLF
 "channel:" Channel CRLF
 ["responder:" Responder CRLF]
 ["capabilities:" Capabilities CRLF])
|"sessionid:" SessionId CRLF)
"transactionid:" Transactionid CRLF
["outcome:" Outcome CRLF
 "completed:" CompletedTransactionid CRLF]
["agent-type:" AgentType CRLF]
["error:"  ErrorNumber  ErrorText  CRLF]
*[ProductSpecificField CRLF]
CRLF
*Payload
Terminator
0 CRLF
CRLF
```

- Server response message

```
"HTTP/1.1" "200" "OK" CRLF
["Transfer-Encoding:" "chunked" CRLF]
CRLF
1*HEX CRLF                          ; size of 1st chunk
["responder:" Responder CRLF]
["session:end" CRLF]
"transactionid:" Transactionid CRLF
["outcome:" Outcome CRLF
 "completed:" CompletedTransactionid CRLF]
["agent-type:" AgentType CRLF]
["error:"  ErrorNumber  ErrorText  CRLF]
*[ProductSpecificField CRLF]
CRLF
*Payload
Terminator
0 CRLF
CRLF
```

Or if the Server has no messages to return

```
"HTTP/1.1" "200" "OK" CRLF
["Content-length:" length] CRLF
["responder:" Responder CRLF]
["session:end" CRLF]
["outcome:" Outcome CRLF
 "completed:" CompletedTransactionid CRLF]
["agent-type:" AgentType CRLF]
["error:"  ErrorNumber  ErrorText  CRLF]
*[ProductSpecificField CRLF]
CRLF
```

If a completed transaction outcome other than COMMIT is returned then the session MUST also be ended.

5.5. REPORT

This command is used when the client does not wish to send or receive messages but does wish to notify the server about the state of transactions related to messages previously received from the server as a result of a PULL, about which the server may still be in doubt. (Since EXCHANGE is both a PUSH and a PULL, all references here to PULL also apply to the PULL part of an EXCHANGE.) REPORT is also sent if the client is in doubt that messages sent in a previous request were received. This might be when a previous connection failed without a response to a PUSH request being received, or when an INDOUBT Outcome was returned.. (Since EXCHANGE is both a PUSH and a PULL, all references here to PUSH also apply to the PUSH part of EXCHANGE.)

REPORT is sent by the client as soon as possible after it has received a payload, if it does not wish to send or receive a new payload. This allows the responder to complete its transaction and not remain in doubt for longer than necessary. After the client has received a response with no HTTPR error indication, the client may forget all state associated with the CompletedTransactionid.

This request is also sent after a PULL request when the requester does not want to commit the messages it received. Once a REPORT request with an INDOUBT outcome has been sent, the responder MUST also terminate the session, if any. If there is a session in progress, the requester MUST not process any further, pipelined responses to requests on this channel (particularly PULL requests that carry a payload) it may have outstanding, and MUST, instead, start a new session. This is necessary to maintain application message ordering, because the payload in the transaction that is in doubt must be resolved before new payloads can be committed.

It may be necessary for an agent to respond INDOUBT if its resource manager fails to return from its COMMIT or ROLLBACK instruction. In this case the agent does not know the outcome of the transaction and may wish to communicate this to the requester rather than blocking or failing itself.

This flow is also used, typically as the last flow in a session when the client wishes to allow itself and the server to forget the CompletedTransactionid. After this flow has been exchanged neither side need retain any transaction state.

- Client request message

```
"POST" /ServiceName "HTTP/1.1" CRLF
["Host:"host[":port]" CRLF]
["Content-length:" Length]
CRLF
"request:" "REPORT" Version CRLF
(("requester:" Requester CRLF
 "channel:" Channel CRLF
 ["responder:" Responder CRLF]
```

```
 ["capabilities:" Capabilities CRLF])
|"(sessionid:" SessionId CRLF
 ["session:end" CRLF]))
["outcome:" Outcome CRLF
 "completed:" CompletedTransactionid CRLF]
"last-pushed-id:" LastPushedId CRLF
["forget:" ForgetTransactionid CRLF]
["agent-type:" AgentType CRLF]
["error:"  ErrorNumber  ErrorText  CRLF]
*[ProductSpecificField CRLF]
CRLF
```

- Server response message.

```
"HTTP/1.1" "200" "OK" CRLF
["Content-length:" length CRLF]
["responder:" Responder CRLF]
["session:end" CRLF]
 "last-pulled-id:" LastPulledId CRLF
["agent-type:" AgentType CRLF]
"outcome:" Outcome CRLF
"completed:" CompletedTransactionid CRLF
["error:"  ErrorNumber  ErrorText  CRLF]
*[ProductSpecificField CRLF]
CRLF
```

The response's CompletedTransactionid MUST be returned as 00000000 00000000 if there is no last transaction known at the server for this channel. This might be because this is the first request ever on this channel or the first request after the server has forgotten the state of completed transactions on this channel.

To avoid duplication of messages that are sent by the server, the client MUST NOT process any responses to any other outstanding requests once it sends a REPORT request and until it receives the response. A PULL response with a transactionid less than or equal to the LastPulledId must then be rejected by the client, as the server will retransmit the affected messages, in batches with new, larger transactionids, at the next opportunity, given the client's REPORT of having failed to receive those messages.

To prevent the duplication of messsages that are sent by the client, once a REPORT request has been processed, the server MUST NOT accept requests with transactionids that are less than or equal to the LastPushedId sent by the client. The client will, having seen the server's claim not to have received those transactions, retransmit the affected messages at the next opportunity with new, larger transactionids.

5.6. Error responses

The requester and responder both need a capability to report errors. These errors are carried in the error field of both the requests and replies, according to whether the client or server detected an error.

If the server cannot handle the capabilities of the client, not even by negotiating them down then the following error is returned. Any transaction associated with the request is rolled back.

```
"["error:" "510" "INCOMPATIBLE" CRLF]
"[outcome:" "ROLLBACK" CRLF
 "completed:" CompletedTransactionid CRLF]
"session:end" CRLF
[payload to explain why we don't like or know the capabilities ]
```

If the server does not recognise the name of the responder to be itself then the following error is returned. Any transaction associated with the request is rolled back.

```
"error:" "511" "RESPONDER-INVALID" CRLF
"[outcome:" "ROLLBACK" CRLF
 "completed:" CompletedTransactionid CRLF]
"session:end" CRLF
```

If the server does not recognise the name of the channel then the following error is returned. Any transaction associated with the request is rolled back.

```
"error:" "512" "CHANNEL-INVALID" CRLF
"[outcome:" "ROLLBACK" CRLF
 "completed:" CompletedTransactionid CRLF]
"session:end" CRLF
```

If the server’s resource manager is unavailable the following error is returned. Any transaction associated with the request is rolled back.

```
"error" "513" "RESOURCE-MANAGER-UNAVAILABLE" CRLF
"[outcome:" "ROLLBACK" CRLF
 "completed:" CompletedTransactionid CRLF]
"session:end" CRLF
```

If the server’s resource manager is terminating the following error is returned. Any transaction associated with the request is rolled back.

```
"error:" "514" "RESOURCE-MANAGER-TERMINATING" CRLF
"[outcome:" "ROLLBACK" CRLF
 "completed:" CompletedTransactionid CRLF]
"session:end" CRLF
```

If the server’s resource manager is unable to store the message the following error is returned. Any transaction associated with the request is rolled back.

```
"error:" "515" "RESOURCE-MANAGER-CAN-NOT-STORE" CRLF
"[outcome:" "ROLLBACK" CRLF
 "completed:" CompletedTransactionid CRLF]
"session:end" CRLF
```

If the server’s session was ended at the request of the administrator the following error is returned.

```
"error:" "516" "ADMINISTRATOR-CLOSED" CRLF"
[outcome:" Outcome CRLF
 "completed:" CompletedTransactionid CRLF]"session:end" CRLF
```

If the server had no messages to return to the client within the DisconnectInterval the following error is returned.

```
"error:" "517" "DISCONNECT-TIMEOUT-EXPIRED" CRLF
"session:end" CRLF
```

If the server's resource manager is unable to store the message because it cannot identify the sink resource the following error is returned. Any transaction associated with the request is rolled back.

```
"error:" "518" "SINK-NOT-KNOWN" CRLF
"[outcome:" "ROLLBACK" CRLF
 "completed:" CompletedTransactionid CRLF]
"session:end" CRLF
```

If the server discovers that the request does not begin with the characters "POST" `"httpr:"` the following error is returned. This error is likely to occur if some program other than an HTTPR agent attempts to communicate with the responder. The requester will not interpret any of the data in the request so there will be no transaction assumed to be associated with the request.

```
"error:" "519" "NOT-HTTP-R" CRLF"session:end" CRLF
```

If the request received in the server begins with the characters POST" `"httpr:"` but the server believes that the protocol that follows does not meet the HTTPR specification the following error is returned. Any transaction associated with the request is rolled back.

```
"error:" "520" "HTTP-R-PROTOCOL-ERROR" CRLF
"[outcome:" "ROLLBACK" CRLF
 "completed:" CompletedTransactionid CRLF]
"session:end" CRLF
```

If the request contains a message longer than the MaximumMessageSize in the currently negotiated capabilities the following error is returned. Any transaction associated with the request is rolled back.

```
"error:" "521" "MAXIMUM-MESSAGE-SIZE-EXCEEDED" CRLF
"[outcome:" "ROLLBACK" CRLF
 "completed:" CompletedTransactionid CRLF]
"session:end" CRLF
```

If the request contains a more messages than the MaximumBatchSize in the currently negotiated capabilities the following error is returned. Any transaction associated with the request is rolled back.

```
"error" "522" "MAXIMUM-BATCH-SIZE-EXCEEDED" CRLF
"[outcome:" "ROLLBACK" CRLF
 "completed:" CompletedTransactionid CRLF]
"session:end" CRLF
```

If the request would cause the number of in doubt or un forgotten transactions to exceed the MaximumPipelineDepth in the currently negotiated capabilities the following error is returned. Any transaction associated with the request is rolled back.

```
"error:" "523" "MAXIMUM-PIPELINE-DEPTH-EXCEEDED" CRLF
"[outcome:" "ROLLBACK" CRLF
 "completed:" CompletedTransactionid CRLF]
"session:end" CRLF
```

If the request contains a flow other than one in the currently negotiated capabilities the following error is returned. Any transaction associated with the request is rolled back.

```
"error:" "524" "INVALID-FLOW" CRLF
"[outcome:" "ROLLBACK" CRLF
 "completed:" CompletedTransactionid CRLF]
"session:end" CRLF
```

If in the processing of the request the agent is denied access to a resource it needs for security reasons then the following error is returned. Any transaction associated with the request is rolled back.

```
"error:" "525" "AGENT-SECURITY" CRLF
"[outcome:" "ROLLBACK" CRLF
 "completed:" CompletedTransactionid CRLF]
"session:end" CRLF
```

If the server cannot process a message because the Userid associated with a message causes it to be denied access to a resource it needs the following error is returned. Any transaction associated with the request is rolled back.

```
"error:" "526" "USERID-SECURITY" CRLF
"[outcome:" "ROLLBACK" CRLF
 "completed:" CompletedTransactionid CRLF]
"session:end" CRLF
```

If the session is terminated at the request of a user module the following error is returned.

```
"error:" "527" "USER-MODULE-CLOSED" CRLF
[hpttpr-user-module-data:" UserModuleData CRLF]
"session:end" CRLF
```

If the SessionId is not recognised by the server the following error is returned.

```
"error:" "528" "SESSION-IDENTIFIER-NOT-RECOGNISED" CRLF
"session:end" CRLF
```

If the server receives an out-of-sequence, old transaction identifier and discards the associated payload.

```
"error:" "529" "OUT-OF-SEQUENCE-TRANSACTION-DISCARDED" CRLF
["outcome:" "ROLLBACK" CRLF
 "completed:" CompletedTransactionid CRLF ]
"session:end" CRLF
```

If the HTTPR Version is not supported by the server, the following error is returned.

```
"error:" "530" "HTTP-R-VERSION-NOT-SUPPORTED" CRLF
"session:end" CRLF
```

6. Header Fields

Implementations of this protocol SHOULD describe how the fields in the message flows are affected by the application interfaces exposed to the application writer.

6.1. Request Header Fields

6.1.1. Version

```
Version = "HTTPR/1.0"
```

Identifies the version of the HTTPR protocol being used.

6.1.2. Requester

The identity of the client agent making the request. This MUST be constant for all time and SHOULD uniquely identify the agent. An agent MAY assume multiple Requester identities. As the Requester field will be used by servers to determine which messages to send to this client, it SHOULD have the form

```
Requester = "httpr:"["//"host[":port]]"/"ServiceName
```

although HTTPR does not require the contents of the Requester field to have this interpretation.

6.1.3. Channel

The name that the agent associates with a particular stream of HTTPR flows. This may be used to construct a class of service for a particular set of messages; for example, large messages may flow over one channel while small ones flow over another.

6.1.4. Responder

```
Responder = "httpr:"["//"host[":port]]"/"ServiceName
```

The identity of the server agent receiving the request, i.e. the agent that did not initiate contact. It is contacted at the corresponding HTTP URL

```
"HTTP:"["//"host[":port]]"/"ServiceName
```

This Responder value MUST be constant for all time for a particular agent and SHOULD uniquely identify the agent. An agent MAY assume multiple Responder identities. The responder MUST check that it is the intended responder and return an error indication if it is not.

6.1.5. Capabilities

In the request, this is the list of attributes that the client supports; if a capability is not specified in the client's list, then the default value is assumed. In the response, this is the list of attributes that the server will accept and use for this session; if a capability is not specified then the value sent or default assumed by the client is used. The each capability in a response MUST be less than or equal to the corresponding value specified in the request.

```
Capabilities = *Capability
*[ProductSpecificField]

Capability = ["idle_session_interval="IdleSessionInterval]
             |["empty_batch_delay="EmptyBatchDelay]
             |["max_latency="MaxLatency]
             |["max_wait_next="MaxWaitNext]
             |["max_wait_batch="MaxWaitBatch]
             |["maximum_message_size="MaximumMessageSize]
             |["maximum_batch_size="MaximumBatchSize]
             |["maximum_pipeline_depth="MaximumPipelineDepth]
             |["flows="Flows]
             |["session_support="SessionSupport]
```

If the request contains a ProductSpecificField as a capability, the client MUST assume that it is not supported, if it is absent from the response. A ProductSpecificField may not be included in a response if it is not already present in the request.

```
IdleSessionInterval = 1*DIGIT
```

The number of seconds that the server SHOULD keep on hand the information about a session and thereby allow the session to remain active without a new request being submitted by the client. The default value is 10 seconds. The response value (if any) MUST be less than or equal to the request value.

```
EmptyBatchDelay = 1*DIGIT
```

When a PULL request is received, the server may not have any messages intended for this client on hand. The EmptyBatchDelay is the number of milliseconds that the server should wait for the first message to appear (from whatever message queuing subsystem, or other message source, it might be using). If, after this length of time, there are still no messages, the server MUST complete its response immediately, indicating that no messages are available. The default value is 10 seconds. The response value (if any) MUST be less than or equal to the request value.

```
MaxLatency = 1*DIGIT
```

When a PULL request is received, as soon as the server has found one messages intended for this client, the server can start sending the response payload. If the server cannot fill a batch because it has run out of messages, it can wait for more to appear, but the server MUST complete the payload by the time MaxLatency milliseconds have elapsed since the batch was begun with that first message. The default value is 100 milliseconds. The response value (if any) MUST be less than or equal to the request value.

```
MaxWaitNext = 1*DIGIT
```

When a PULL request is received, as soon as the server has found one messages intended for this client, the server can start sending the response payload. If the server cannot fill a batch because it has run out of messages, it can wait for more to appear, but the server MUST complete the payload once MaxWaitNext milliseconds have elapsed since the last message was placed in the batch. The default value is 100 milliseconds. The response value (if any) MUST be less than or equal to the request value.

```
MaxWaitBatch = 1*DIGIT
```

When a PULL request is received, whether there are messages or not, the server MUST complete the payload once MaxWaitBatch milliseconds have elapsed since the request arrived. The default value is 100 milliseconds. The response value (if any) MUST be less than or equal to the request value.

```
MaximumMessageSize = 1*DIGIT
```

The largest number of bytes that MAY comprise a single message. The default value is 100 000 000 bytes. Implementations that do not support messages of unlimited size SHOULD use MaximumMessageSize to avoid the unnecessary transmission of messages that it knows will be rejected a priori. The response value (if any) MUST be less than or equal to the request value.

```
MaximumBatchSize = 1*DIGIT
```

The largest number of messages that MAY flow in a single request or response. The default value is 10 messages. The response value (if any) MUST be less than or equal to the request value.

```
MaximumPipelineDepth = 1*DIGIT
```

The maximum number of outsanding requests that the client can make without having received the corresponding response from the server. The default value is 1 request. The response value (if any) MUST be less than or equal to the request value.

```
Flows= 1*Flow
Flow = PUSH|PULL|EXCHANGE
```

The particular requests that the client and server are willing to make. The delimiter for the list of Flows is “+” not “,”.

```
SessionSupport= 1*SessionLevel
SessionLevel= SESSIONLESS|SESSION
```

An agent MUST support either SESSIONLESS or SESSION, but need not support both. The default value is SESSIONLESS. The delimiter for the list is “+” not “,”.

Example:

```
capabilities:disconnect_interval=15,batch_interval=31,max_message_size=1
0000,batch_size=5,flows=PUSH+PULL,session_support=SESSION CRLF
```

6.1.6. SessionId

```
1*12(ALPHA|DIGIT)
```

Identifies the session uniquely in the server. The server generates a unique session identifier for each new session and returns it to the client. The client includes the session identifier in each request it sends to the server. The server must allocate SessionIds that it knows are unique for all time, one way to achieve this is to use a time stamp for part of the SessionId.

6.1.7. AgentType

The type of agent installed in the client if this is sent in a request. The agent installed in the server if this is sent in a response. It is RECOMMENDED that this string be of the form.

Organisation”.”ProductName

Example:

```
"ibm.MQSeries"
```

This SHOULD be part of the first request and first response in the lifetime of a connection, to serve as documentation that will be useful in the event of a failure.

6.1.8. Transactionid

```
16HEX
```

Identifies the unit of work. The transactionid MUST be unique until such time as both parties agree to forget all previous transaction state. Successive values of transactionid used on a channel MUST form a strictly increasing sequence. It was decided to place the transactionid in the request headers instead of in the terminator of the payload for purposes of documentation. A transmission that is interrupted will, thereby, certainly contain a transactionid if it contains any part of the payload, which may aid administrators during problem determination. This may lead to the inclusion of a transactionid when none of the messages in the payload have a “class of service” of assured or reliable, in which case the transactionid is irrelevant, and the sink need not remember it.

The Transactionid MUST NOT be equal to 0000000000000000 (16 zeros).

6.1.9. CompletedTransactionid

```
16HEX
```

The transactionid which was received in conjunction with a previous payload and which has now been committed. The recipient may assume that all prior transactions that are still in doubt have also been committed.

The reserved value 0000000000000000 (16 zeros) MUST be used if there is no transaction which has just been committed, for instance because this is the first PULL request.

6.1.10. ForgetTransactionid

```
16HEX
```

The transactionid of a unit of work, generated by the client, that is no longer in doubt and which the client wishes to forget.

6.1.11. LastPushedId

```
16HEX
```

The largest transactionid of any unit of work to have been generated by the client.

6.1.12. LastPulledId

```
16HEX
```

The largest transactionid of any unit of work to have been generated by the server.

6.1.13. ProductSpecificField

ProductSpecificField is any field compliant with the HTTP format rules that is not defined in the version of HTTPR being used. Product Specific fields SHOULD begin with the characters “app-” in order to avoid a conflict with names that might be used in future versions of HTTPR. They enable product specific data to be exchanged that is not defined within the HTTPR protocol. Product specific fields may not be blank otherwise it would indicate a delimiter for message header etc.

Example:

```
app-ibm-mqseries-accountingtoken: 00000001
```

The sink MUST ignore these fields if it does not understand their meaning. Product owners SHOULD document all of the product specific fields that their products generate and interpret on receipt. Product owners SHOULD use an HTTPR field in preference to a product specific field, as this would inhibit interoperability.

6.2. Payload and Message Header Fields

The payload is the message header followed by the message data.

The message header is parsed by the agent to find information useful to this receiving agent. However, some fields may be relevant only to the ultimate message sink and are simply be copied unchanged to the next link in the chain of agents in a multi-hop connection.

```
Payload=MessageHeader
        CRLF                        ;End of the message header
        MessageData
        CRLF
```

6.2.1. MessageHeader

```
MessageHeader = ("message-size:" MessageSize CRLF |
                  "message-encoding: chunked")
                 ["target-uri:" TargetUri CRLF]
                 ["class-of-service:" ClassOfService CRLF]
                 ["priority:" Priority CRLF]
                 ["user-id:" UserId CRLF]
                 ["encoding:" Encoding CRLF]
                 ["reply-uri:" ReplyUri CRLF]
                 ["message-id:" MessageId CRLF]
                 ["correlation-id:" CorrelationId CRLF]
                 ["put-time:" PutTime CRLF]
                 ["expiry:" Expiry CRLF]
                 ["content-type: ContentType CRLF]
                *[ProductSpecificMessageField CRLF]
```

6.2.2. MessageSize

MessageSize = 1*DIGIT

The number of bytes in the MessageData itself, excluding the CRLF following the MessageData.

6.2.3. TargetUri

The destination for the message.

```
TargetUri = "httpr:"["//"host[":"port]]"/"ServiceName"#"Destination
```

Host specifies the network address of the agent or its proxy and port is the port, default 80. The ServiceName identifies the agent. Destination identifies the sink for the message as interpreted by the agent. There are no specific rules for interpreting the Destination. Providers SHOULD document how the Destination is interpreted. Among other possible formats, one way a messaging system might construct Destination is QueueName“@”QueueMangerName.

Examples:

```
Target-uri: httpr:QM1#/QueueQuery@QM1 CRLF
target-uri: httpr://gateway.org1.com/soapAgent#SOAPQ@QM1_SERVER CRLF
```

6.2.4. ClassOfService

```
ClassOfService = "assured"          ;once and only once
                |"reliable"         ;at least once
                |"datagram"         ;at most once
```

Default = "datagram"

Once-and-only-once delivery requires full transaction coordination between sender and receiver. The agent where messages originate will be in doubt as to whether they have arrived for some period during the transfer, messages in this state would not normally visible to applications.

At least once delivery allows lazy confirmation from receiver.

Datagram or at most once delivery requires no coordination between sender and receiver.

6.2.5. Priority

The priority of the message. Agents should make their best effort to transfer higher priority messages before lower priority messages.

```
Priority = DIGIT
```

Default = 4.

6.2.6. UserId:

```
token
```

The user identifier of the user that originally created the message, or the user identifier of the user who has assumed ownership since it was created. This field may be used by the source agent to determine authority of the message to flow to the sink. It may also/instead be used by the sink agent to determine if the message is acceptable and can use the resources it needs.

6.2.7. Encoding

```
Encoding = [EncodingIntegerType]
           [","EncodingFloatType]
           [","EncodingDecimalType]
```

```
EncodingIntegerType = "integer-normal"
                      |"integer-reversed"
```

The default is integer-normal.

```
EncodingFloatType =   "float-ieee-normal"
                    | "float-ieee-reversed"
                    | "float-s390"
```

The default is float-ieee-normal.

```
EncodingDecimalType=  "decimal-normal"
                     |"decimal-reversed"
```

The default is decimal-normal.

```
The default encoding
```

6.2.8. ReplyUri:

```
ReplyUri = "httpr:"["//"host[":"port]]"/"ServiceName"#"Destination
```

6.2.9. MessageId:

```
token
```

An identifier of the message provided by the sending application. See the discussion of CorrelationId (next) for further details.

6.2.10. CorrelationId:

```
token
```

Another identifier associated with the message. Where a reply is to be generated by a receiving application, the message-id of the request is often copied by that application into the correlation identifier of its reply. It is the responsibility of the application generating the request, or the agent acting on its behalf, to ensure that message identifiers contain adequate information so that the correlation-ids can correctly distinguish replies.

HTTPR is not aware of the request/reply relationship. There is no HTTPR header information to indicate that a particular message is a request expecting reply, though this might be inferred from the presence of the reply-uri: field. HTTPR does not specify any relationship between the application request/reply and HTTPR EXCHANGE sequences.

Where the HTTPR agent and application are tightly coupled, an application request and resulting reply may flow within a single HTTP flow using HTTPR EXCHANGE. Transactional requirements for loosely coupled HTTPR agent and application will almost certainly involve multiple HTTP flows.

6.2.11. PutTime:

```
http-r-date
```

The time of day when the messages was created. For example:

```
06 Nov 1994 08:49:37
```

All HTTPR date/time stamps MUST be represented in Greenwich Mean Time (GMT), without exception. For the purposes of HTTP, GMT is exactly equal to UTC (Coordinated Universal Time). http-r-date is case sensitive and MUST NOT include additional LWS beyond that specifically included as SP in the grammar.

```
http-r-date = date1 SP time

date1        = 2DIGIT SP month SP 4DIGIT

               ; day month year (e.g., 02 Jun 1982)

time         = 2DIGIT ":" 2DIGIT ":" 2DIGIT

               ; 00:00:00 - 23:59:59

month        = "Jan" | "Feb" | "Mar" | "Apr"
             | "May" | "Jun" | "Jul" | "Aug"
             | "Sep" | "Oct" | "Nov" | "Dec"
```

Note: HTTP requirements for the date/time stamp format apply only to their usage within the protocol stream. Clients and servers are not required to use these formats for user presentation, request logging, etc.

6.2.12. Expiry

After this number of seconds have elapsed the message need not be presented to an application. The total time MAY exclude time spent in transmission of the message on the communications link.

```
Expiry = 1*DIGIT
```

6.2.13. ContentType

```
Pcf/pcf;charset=iso-8859-4
```

6.2.14. ProductSpecificMessageField

ProductSpecificMessageField is any field compliant with the HTTP format rules that is not defined in the version of HTTPR being used. Product Specific message fields SHOULD begin with the characters “app-” in order to avoid a conflict with names that might be used in future versions of HTTPR. They enable product specific data to be exchanged that is not defined within the HTTPR protocol. Product specific fields may not be blank otherwise it would indicate a delimiter for message header etc.

Example:

```
app-ibm-mqseries-correlid: 00000001
```

The sink SHOULD ignore these fields if it does not understand their meaning, but MUST save them, as they may be intended for same later consumer of the message. Product owners SHOULD document all of the product specific fields that their products generate and interpret on receipt. Product owners SHOULD use an HTTPR field in preference to a product specific field, as this would inhibit interoperability.

6.2.15. MessageData

The application message data is an uninterpreted sequence of bytes unless the chunked message encoding is used. The encoding of the message data is, in that case, exactly as specified in RFC 2616, in section 3.6.1, for an HTTP/1.1 chunked encoding:

```
Chunked-Body   = *chunk
                 last-chunk
                 trailer
                 CRLF

chunk          = chunk-size [ chunk-extension ] CRLF
                 chunk-data CRLF
chunk-size     = 1*HEX
last-chunk     = 1*("0") [ chunk-extension ] CRLF

chunk-extension= *( ";" chunk-ext-name [ "=" chunk-ext-val ] )
chunk-ext-name = token
chunk-ext-val  = token | quoted-string
chunk-data     = chunk-size(OCTET)
trailer        = *(entity-header CRLF)
```

As these chunks are passing over an HTTP/1.1 session, there is the possibility of double chunking, with messages chunked for HTTPR being re-chunked for HTTP. However, no confusion should result, as dechunking is applied first at the HTTP level, and then, separately, at the HTTPR level.

6.2.16. Terminator

The final part of the message following the payloads.

```
Terminator = "payload-disposition:" PayloadDisposition CRLF
PayloadDisposition =  "last"
                     |"abort"
```

This is the last part of the HTTP message body and indicates that the payload is valid and can be used by the recipient. Once the terminator has been received the recipient may proceed to process or abort the request or response.

Last means that the recipient may attempt to commit the contents of the message where the class of service is assured.

Abort means that the recipient MUST discard all of the request response flow. The response to a payload where the terminator is “abort” MUST contain

```
"outcome:"  "ROLLBACK"
```

Similarly the sink MUST discard the payload if the connection is broken before the terminator is received.

6.2.17. Outcome

```
Outcome = "COMMIT"
         |"ROLLBACK"
         |"INDOUBT"
```

COMMIT means the sink has received and permanently recorded the payload and any other state it might need.

ROLLBACK means the sink has been unable to receive the payload.

INDOUBT means the sink is uncertain as to what it did with the payload, perhaps because there was a partial system failure during the processing.

6.2.18. ErrorNumber

```
ErrorNumber = 1*DIGIT
```

The number that uniquely identifies the error as listed above. Either the client or server may generate a single error on each HTTPR message.

6.2.19. ErrorText

English text that describes the error being reported as described above.

Appendix

A1. Glossary

agent	The software transferring the message payload on behalf of the application, also the resource manager for transactions.
application message	The message being transferred as the application sees it, as distinct from the HTTP message.
capabilities	A vector describing the HTTPR protocol capabilities and parameters requested or agreed to on this command flow or session
channel	An independent HTTPR conversation globally uniquely identified by the triplet: <client URI, channel identifier, server URI >.
client	The agent initiating the communication. The agent sending the HTTP request message.
command flow	The basic unit of HTTPR interaction; a request message from client to server and a response message from server to client. The request flows as the body of an HTTP POST; the response as its POST response.
commit	The action of permanently recording that the payload has been received, or sent.
connection	The TCP/IP communications used to carry the requests and responses. This has the same lifetime as the TCP/IP socket.
forget	The point at which the agent is no longer required to have knowledge of the transaction associated with a payload.
HTTP	HyperText Transfer Protocol.
HTTPR payload	The application messages being transferred, togrether with their HTTPR message header information.
multi-hop	A series of Agents that a message passes through, each acts as a source for the next agent in the chain. Messages are stored at each Agent and then passed unaltered to the next Agent.
prepare	The action of permanently recording that a payload is in doubt as to whether the partner agent has received it or not.
request	The HTTPR request message sent by the client.
resource manager	The software that stores persistent state and manages transactions.
response	The HTTPR response message sent by the server.
server	The agent accepting the communication initiated by the client. The agent responding to the request message sent by the client.

session	A uniquely identified, grouped sequence of command flows which use a fixed, prenegotiated set of capabilities.
sink	The agent with the application messages after the transfer.
source	The agent with the application messages before the transfer.