

# HTTPR Specification

## *Draft Proposal*

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

IBM

Version 1.0 13th July 2001

The authors acknowledge assistance from: Andrew Donoho, Tim Holloway, John Ibbotson

## 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.

© Copyright IBM Corporation, 2001. All rights reserved.

# 1. Introduction and overview of HTTPR concepts

## 1.1. HTTPR objectives and relationship to HTTP

Httppr 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, httppr defines how metadata and application messages are encapsulated within the payload of http requests and responses. Httppr 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 undeliverable.

Messaging agents use the httppr protocol and some persistent storage capability to provide reliable messaging for application programs. This specification of httppr 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 httppr.

Http/1.1 serves as the base on which httppr builds. As such, all of the facilities of http/1.1 (SSL, 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 httppr on http in this way has the additional benefit that httppr 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 httppr interaction ( the client ) does so by sending a POST command, in the http sense, including with it a payload that identifies itself, specifies an httppr command, and, if the command asks the server to accept messages, includes 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 httppr is concerned and are assigned no other meaning by it.

Each batch is assigned an identifier by its sender (either client or server), which is sent along as httppr 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 http protocol places no constraints on the interface used by an application program to pass messages to its local messaging agent for reliable delivery to a partner application program. SOAP and JMS are two examples of application messaging interfaces for which reliable delivery using the http protocol can be provided.

In addition to providing reliable messaging over a “single hop”, http 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 http 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 http will provide reliable end-to-end messaging.

## **1.2. Reliable exactly-once delivery with HTTPR**

The http protocol allows multiple levels of delivery quality to be supported but the most important is *reliable exactly-once delivery*. With this quality of service each message will be reliably delivered to its destination application for processing exactly once or will be reliably reported as undeliverable.

The messages delivered to a target application from any one sending application should be received in exactly the same order as they were sent. However, end-to-end message ordering can be affected by factors which lie outside the scope of the http protocol. For example, ordering can be affected by the way in which messaging agents treat independent units of work or by the topology of the network of agents. A network of messaging agents which has multiple paths between a pair of endpoints would not normally preserve message ordering.

The agents transferring messages on behalf of the application have storage which they use to maintain state controlling the application message transfer. This state enables them to resume application message transfer reliably if the communications are interrupted, or if the agents themselves fail. The behavior of the persistent storage interface can also affect the end-to-end ordering of messages.

## **1.3. HTTPR command flows**

An http interaction consists of a sequence of exchanges between an http client and an http server. Each of these exchanges is an http command flow initiated at the http client. Each command flow consists of an http command flowing from the http client to the http server and the command response being returned to the http client. In general, the different http commands can be used in any sequence for any http client server pair.

This section provides short informal descriptions of the function and purpose of each of the http command flows. The detailed description of each http command flow follows in section 4, “HTTPR Command Flows”.

### **1.3.1. Push**

This command flow allows an http client to send a batch of one or more messages to an http server and get back a response indicating whether the message batch has been received and saved

reliably. 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 command flows between this client and server - defined more formally as a *channel* in subsection 1.5 ). A Push command may also include a piggy-backed acknowledgment reporting that a particular incoming batch of messages in the response to a preceding Pull or Exchange command was received safely by the client.

### **1.3.2. Pull**

This command allows an http server to ask an http client to send it any messages waiting for delivery to applications located at the client. The response to a pull command may be “empty” ( if the http server has nothing waiting in the requested class of service for delivery to this http 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. A Pull command may also include a piggy-backed acknowledgment reporting that a particular incoming batch of messages in the response to a preceding Pull or Exchange command was received safely by the client.

### **1.3.3. Exchange**

The Exchange command combines a Push and a Pull command into a single command flow. It allows an http client to send a batch of one or more messages to an http server. Unlike Push, it invites any waiting messages at the http server to be returned to the http client in the response along with the indicator of whether the sent messages were safely received and saved at the http 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 http client in a single http command flow. However, the http 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.

### **1.3.4. Resolve**

The Resolve command flow enables an http client to determine exactly what has been safely received at an http server after some communications error. The Resolve command includes the transaction identifier of the last message batch which this http client has sent to its partner http server. The http server saves this transaction identifier and responds to the Resolve command with the transaction identifier of the last message batch it has received safely from the http client. For exactly-once delivery, the http client will resend any message batches sent but (according to the response to Resolve) not received at the http server. After responding to a Resolve command, the http 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 Resolve, and reported by the server as not received in its response to the Resolve.

### **1.3.5. Report**

The Report command flow enables an http client to report to an http server exactly which batches of messages it has received and saved ( from that server ). Use of Report may be

prompted either by a communications error or by the need to relieve the http server from having to continue saving state information about the last interactions with this http client. The Report command includes the unique transaction identifier of the last batch of messages successfully received by this http client from its partner http server. The http server responds with the unique identifier of the last batch it tried to send to this http client and discards copies of messages now known to be safely delivered. After receiving the response to a Report, the http client will reject any “late arriving” batches of messages which have transaction identifiers showing that they were sent by the server before it responded to the Report command but have been delayed in transit from the http server. In response to subsequent Pull or Exchange command, the server will resend any messages shown by a preceding Report command to have been lost in transit to the client.

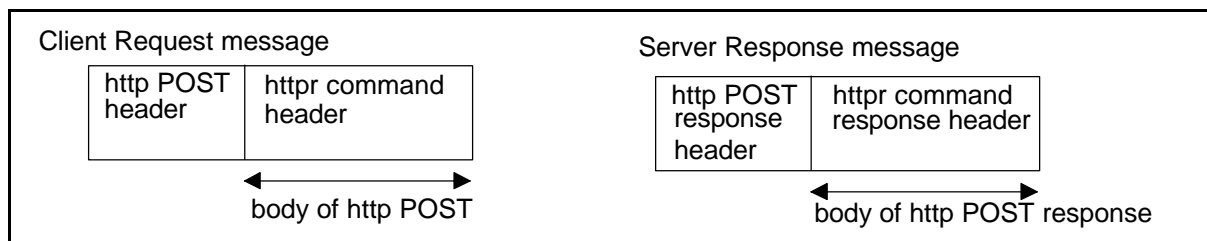
### 1.3.6. Get Responder Info

This command flow is used by an http client to inform an http server of its level of http protocol capabilities. The http-r server is allowed to respond with reduced capabilities which then become the agreed parameters for all following http command flows between this http client server pair. This sequence of interactions is an http session. Http sessions are ended by a Report command flow with end-session indicated. Capability negotiation for http client server pairs that are not using http sessions, is supported by allowing capability information to be included in the requests and responses of all the other commands - Push, Pull, Exchange, Resolve and Report.

## 1.4. HTTPR message structure and datastream

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

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

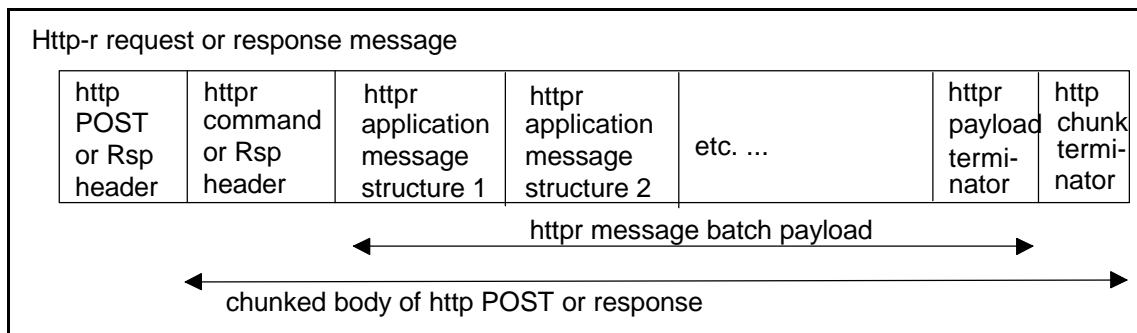


**Figure 1: overview of datastream for simple http command flows**

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

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

Following the http request or response header is a batch of one or more http 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 http channel as one step in its path. The http 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. Http command or response with a message batch payload**

When an http request or response message includes a payload containing application messages, the size of the http message is open ended and potentially large. To ensure that this amount of data can be handled at the http endpoints and indeed at intermediate nodes in the network, http messages with payload are 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 http requests and responses including a payload will usually be terminated with an http data chunk terminator as specified in the http protocol.

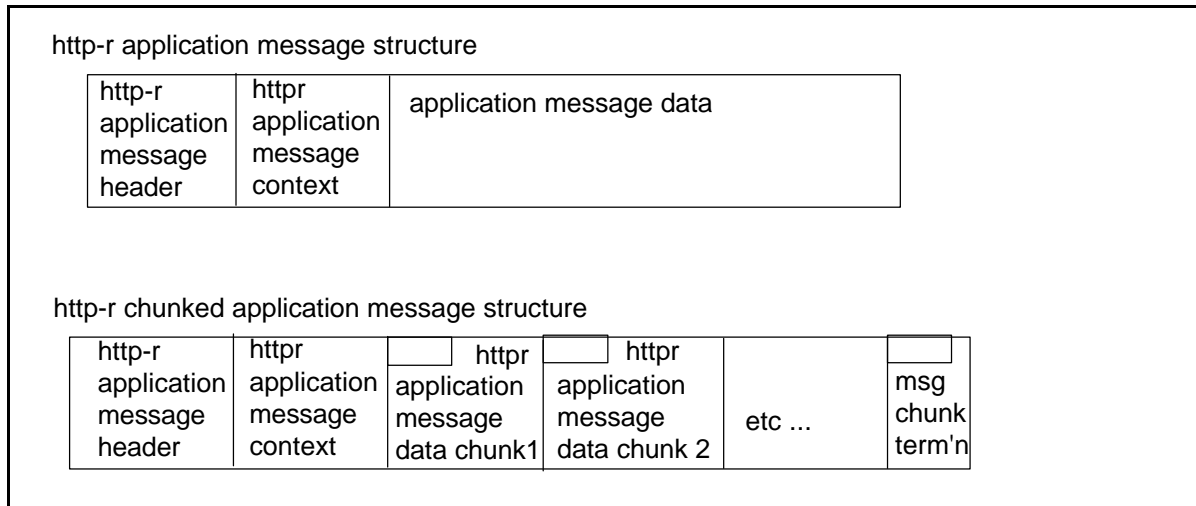
The sequence of http application message structures in a payload is terminated by an http payload terminator. This has the additional function of indicating *at the end of the payload* whether the sending agent detected some error condition during transmission of the messages which will require the entire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

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

The http 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 http level. The application message data is an uninterpreted stream of bytes in which any character values are allowed. If the individual application message is too large for the sending or receiving agent to hold it in memory at one time, it may be encoded as a sequence of http message data chunks each with their own length indication and with the sequence terminated by a null chunk. This http level of message data chunking may occur in addition to regular http data chunking of the entire command or response message. The two levels of chunking will not interfere destructively.

The http application message structure also includes a message context header with meta-data about this application message which will enable messaging agents to present it correctly or to transform it for the end-user application. Http defines a set of message context fields. Additional

message context fields may be inserted to meet the needs of messaging agents. These fields will always be forwarded along with the associated application message.



**Figure 3. Datastream for http application message structure**

Simple http agents may choose to use commands with the message batches containing a single message only. Support for message batches in the http protocol aids scalability and efficiency.

### 1.5. Identifying HTTPR clients, servers and channels.

The http client and http server participating in an http interaction are uniquely and globally identified by their URI's. However, at any point in time there could be several independent http conversations in progress between a particular http client and http 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 http protocol over a single channel. An http server and client serialize their use of http command flows for any one channel. When multiple channels are in use between an http client and http server, each one follows the protocol rules independently.

Channels are uniquely identified by the ordered triplet:

```
< http client URI ; channel identifier ; http 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.

Http channels can be created dynamically without any preconfiguration by a client sending an http command to a server and the server accepting it and responding. 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.

## 1.6. Levels of HTTPR functionality and capability negotiation

Several functional levels of the http protocol are defined:

sessionless

simple session

pipelined session

These levels of functionality and other parameters of the http 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 http command and response headers.

*Sessionless* is the simplest mode for operating an http channel. In this mode each http command flow on the channel is independent and carries its own capability negotiation. The http client may include its capability vector in each command header. If this request requires capabilities not supported in the http server, the command will be rejected. If the server can accept the http command but wishes to process it with different, “lesser” capabilities, it returns its capability vector in the http command response header. 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 http is a higher level capability in which a session is established between http client and http 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 http client can terminate the session using a Report command.

Pre-negotiating capabilities allows for better tuning of the communications properties at both client and 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 http channel is either in session based use by exactly one session, or is available for sessionless use. Support of sessions by http implementations is optional.

In session based http ( 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 associated with the session at any time. Furthermore, in simple session mode, http pipelining of the http commands on this connection is disallowed.

*Pipelined session* mode is the third and most sophisticated mode of httptr 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 httptr 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 httptr client and server, to be distinguished from old httptr commands in the http pipeline behind the command which caused the abort.

## **1.7. 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 httptr headers

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

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 = ( "<" \*(qdtex | quoted-pair ) "<" )  
qdtex = <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. Common concepts, design issues and rationale

### 3.1. Relationship between HTTP and HTTPR

All http 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 http 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 http. To avoid this exposure, and because it is anticipated that http will be a minority web protocol, this approach was not taken.

### 3.2. Uniform Resource Identifier

The client identifies the server intended to receive an http command by using a URI constructed as follows.

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

If host is not given, the local host is used. If port is empty or not given then 80 is used. ServiceName identifies the agent that will respond to the http requests carrying http.

The sink of each individual message, to be interpreted by the agent at the message's final destination as it sees fit, is named in the target-uri: field of the payload as follows.

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

Destination is not otherwise interpreted and is assigned no meaning by http

### 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 they 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 flowed on this channel. See the section on Pipelining for more details on this subject.

If 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 flows:

```
[ "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 before it ends a session. 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 server only needs to retain the last client generated transaction it returned an outcome for. The client can send a ForgetTransactionid in sessionless and simple session http, only when there are no transactions left in doubt on this channel. With pipelined session http, the client can send ForgetTransactionid when transaction to be forgotten is resolved (known not to be in doubt) by both parties.

Once an ForgetTransactionid has flowed, RESOLVE 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 was lost. The client and server may also forget all prior transactionid's that the client generated.

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 to take place.

### **3.4. Resolution of In Doubt Transactions**

Occasionally the PUSH, PULL and EXCHANGE requests will not lift the in doubt status of a transaction in the source, either because an INDOUBT outcome is received or because the request or response containing the Outcome is lost.

If the client sends or receives an INDOUBT outcome or fails to receive a response to an http request it MUST flow a RESOLVE or GET-RESPONDER-INFO request next, and attempt to remove the in doubt status of the unit of work. If the client chooses to use GET-RESPONDER-INFO, does not send an Outcome and CompletedTransactionid, but has previously sent a Transactionid and not forgotten it, then it MUST follow the GET-RESPONDER-INFO request with a RESOLVE request that does contain the Outcome and CompletedTransactionid. By doing so, all units of work that have not yet been seen are clearly identified, which allows them to be correctly handled (rejected) if they should arrive at some later date.

### 3.5. Message ordering assumption

In one transmission of a batch of messages from one agent to another over http, 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 http standard, we simply claim that it is possible to build an order preserving messaging network with http.

### 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 flagged as beginning by using the "session: begin" on a GET-RESPONDER-INFO request and ending by using "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 session 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 state 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.

The recipient SHOULD rollback any uncommitted transactions if a connection terminates before the http request message is complete.

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 http:// 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 http:// 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**

Http:// may flow over https 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 https can achieve mutual authentication of the source and sink, privacy and protection of the data exchanged. The http:// 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.

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

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

## 4. HTTPR Command Flows

This chapter describes the http 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 http commands may flow in any sequence on any single http connection. Each http command flow consists of a client request message sent from the client to the responder and replied to with a server response message. The request message flows as an http POST; the response message flows as an http POST response. Command flows on a single http 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.

### 4.1. 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 RESOLVE 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.

## 4.2. 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 RESOLVE 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. This behavior is preferable to the server simply waiting until it knows the outcome of the transaction, because the client will now also know that there is a problem at the server and that there will probably be some delay while the problem is resolved.

### 4.3. 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.

#### 4.4. 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.

## 4.5. REPORT

This flow 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 or EXCHANGE, about which the server may still be in doubt. 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 http 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 or EXCHANGE 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
["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]
["error:" ErrorNumber ErrorText CRLF]
*[ProductSpecificField CRLF]

CRLF
```

To avoid duplication of messages, 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.

#### 4.6. RESOLVE

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** or **EXCHANGE** request being received, or when an **INDOUBT** Outcome was returned.. No payload messages may be sent in either direction until the indoubt state has been removed.

```
"POST" /ServiceName "HTTP/1.1" CRLF
["Host:"host[:port]" CRLF]
["Content-length:" Length]

CRLF

"request:" "RESOLVE" Version CRLF
(("requester:" Requester CRLF
"channel:" Channel CRLF
["responder:" Responder CRLF]
["capabilities:" Capabilities CRLF])
```

```

|"sessionid:" SessionId CRLF)
  "last-pushed-id:" LastPushedId 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 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 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 prevent the duplication of messages, once a RESOLVE 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.

## 4.7. 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 http 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 http 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
[hhttpr-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.

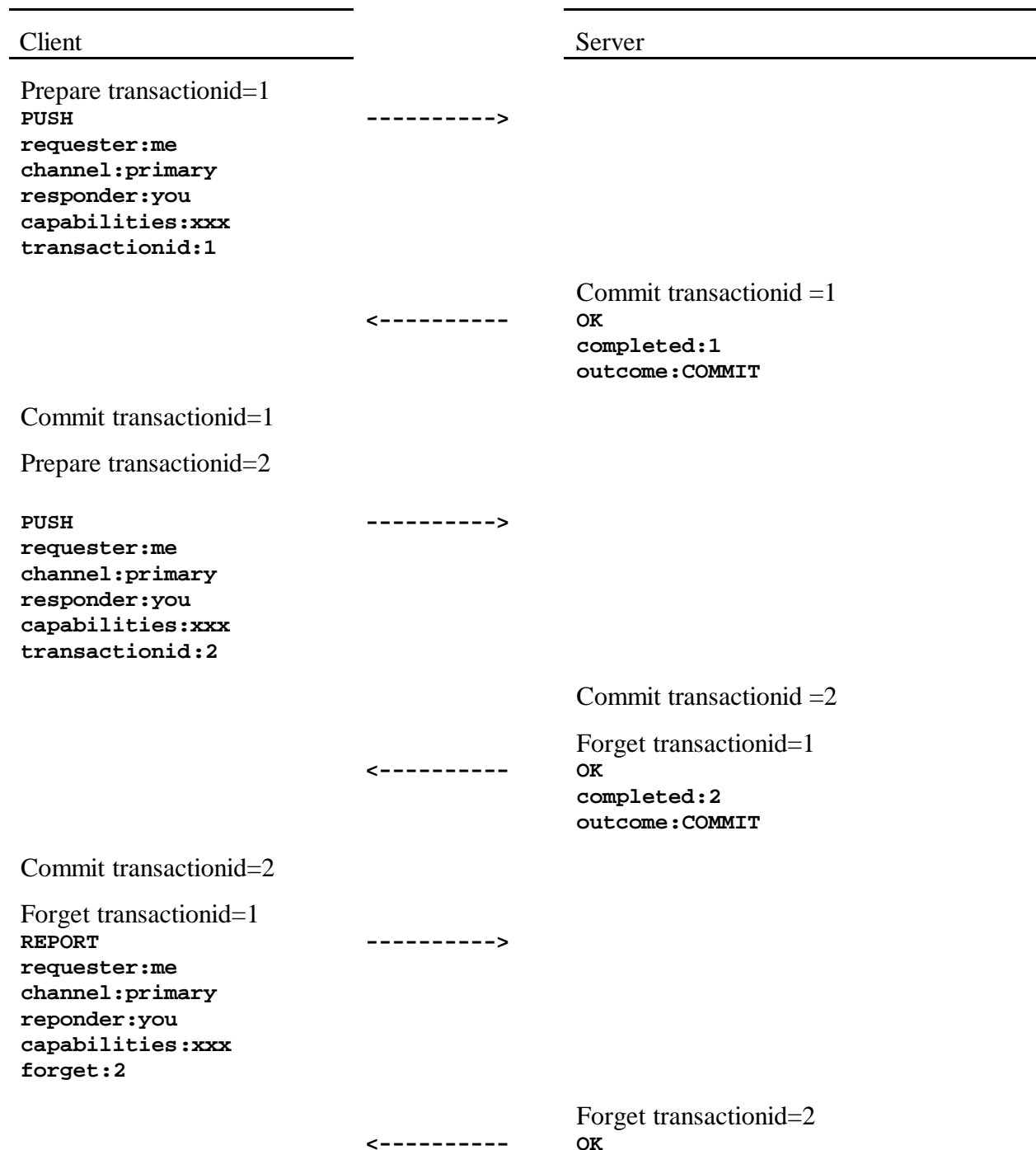
```
"error:" "530" "HTTP-R-VERSION-NOT-SUPPORTED" CRLF
"session:end" CRLF
```

## 5. Example message flow scenarios

Http messages shown in bold. Agent activity shown normal.

### 5.1. Client initiates PUSH

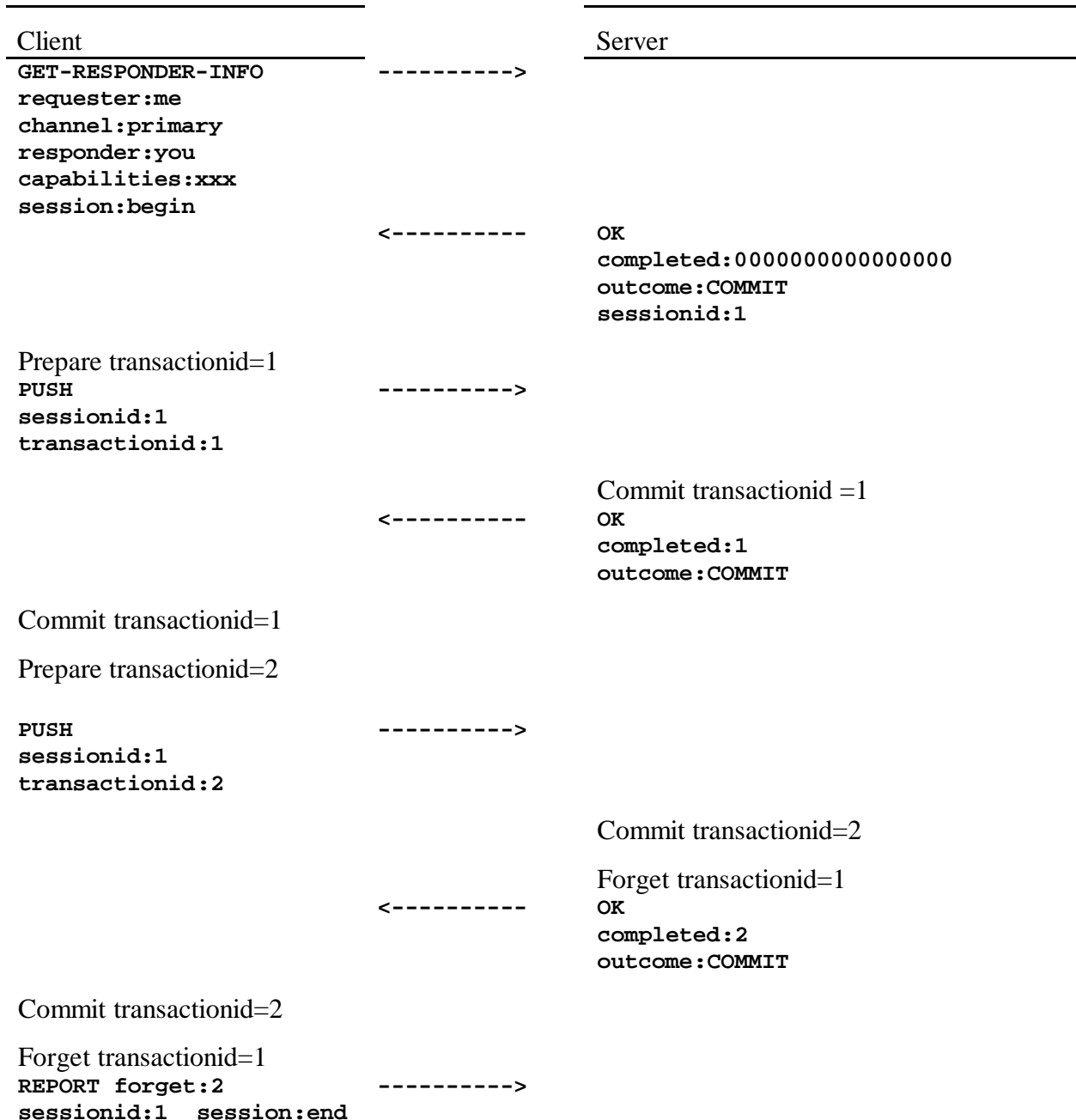
The client uses sessionless httptr to push messages to a server, which makes them available processing to applications located there. When the client has no more messages to send it issues a final REPORT.



Forget transactionid=2

## 5.2. Client initiates PUSH, using a session

The client uses simple session http 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.



```

                                Forget transactionid=2
                                OK
                                <-----
Forget transactionid=2

```

### 5.3. Client fails then restarts PUSH

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

Client	Server
Prepare transactionid=1	
<b>PUSH</b> transactionid:1 sessionid:1	
	Commit transactionid=1 OK completed:1 outcome:COMMIT
	Failure<---

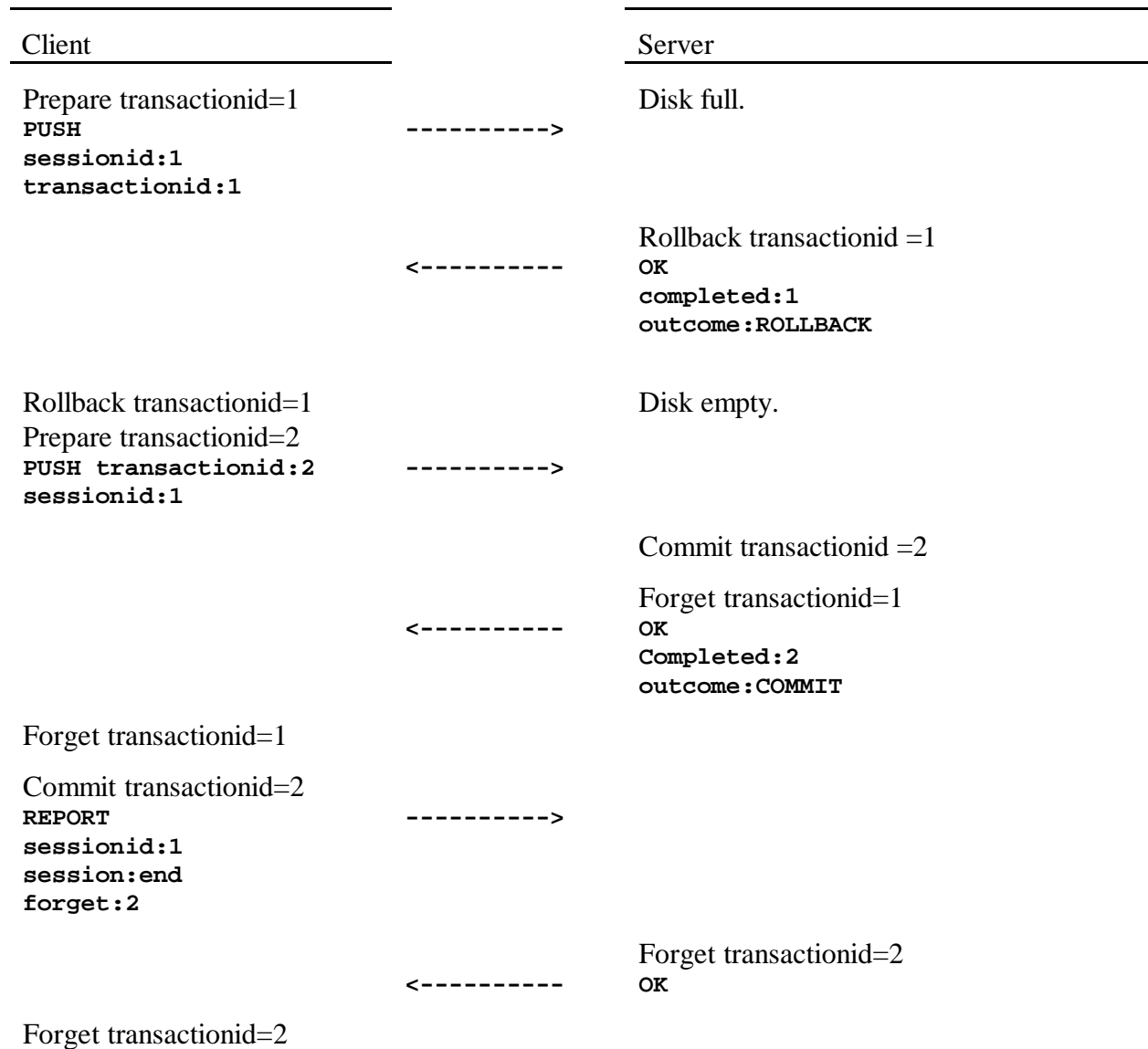
After the failure the client, does not receive its expected response and so MUST establish whether the previous PUSH was committed or rolled back, before it can continue with other PUSH, PULL or EXCHANGE requests. The client chooses to start a new session to accomplish this, then it sends REPORT and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

Client	Server
Transactionid=1 in doubt GET-RESPONDER-INFO requester:me channel:primary responder:you capabilities:xxx session:begin	
	OK completed:1 outcome:COMMIT sessionid:2
Commit transactionid=1 REPORT sessionid:2 session:end forget:1	
	Forget transactionid=1 OK

Forget transactionid=1

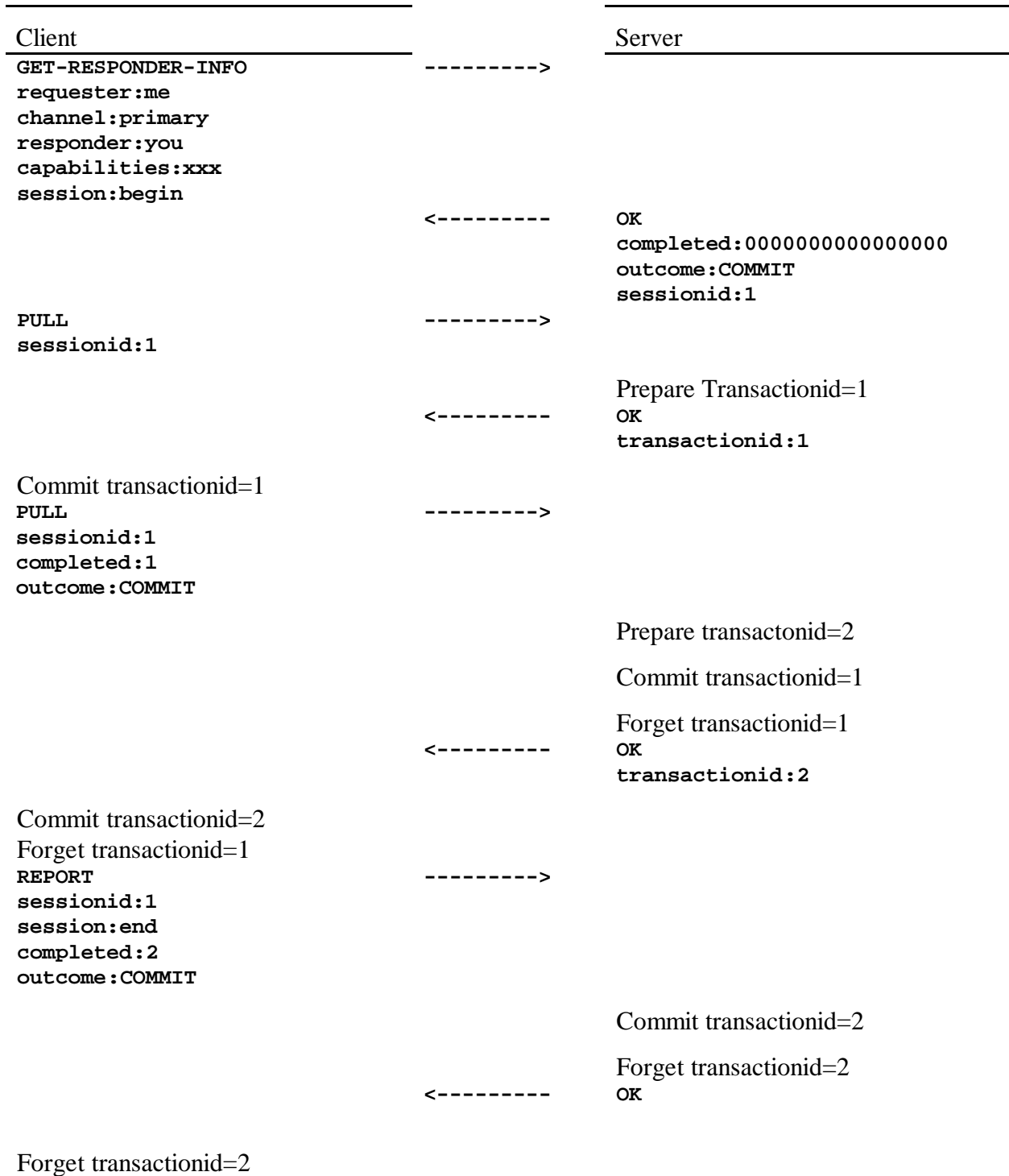
#### 5.4. Client initiates PUSH, server rolls back.

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



#### 5.5. Client initiates PULL

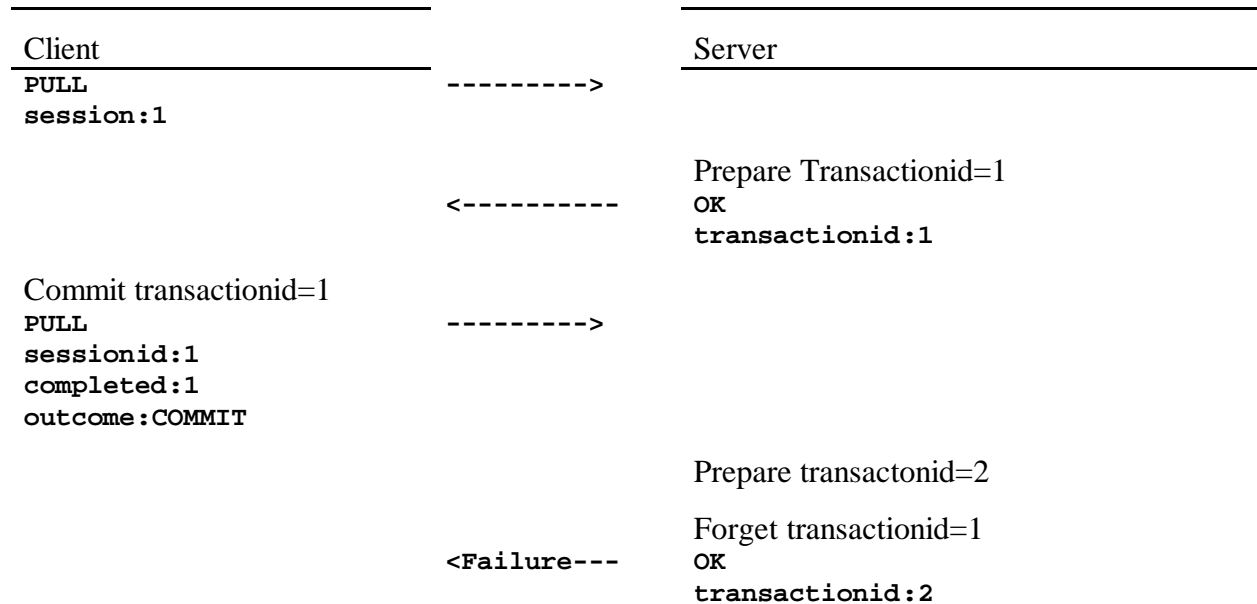
The client uses simple session httptr to pull messages from the server and make them available to for processing to applications located at the client.



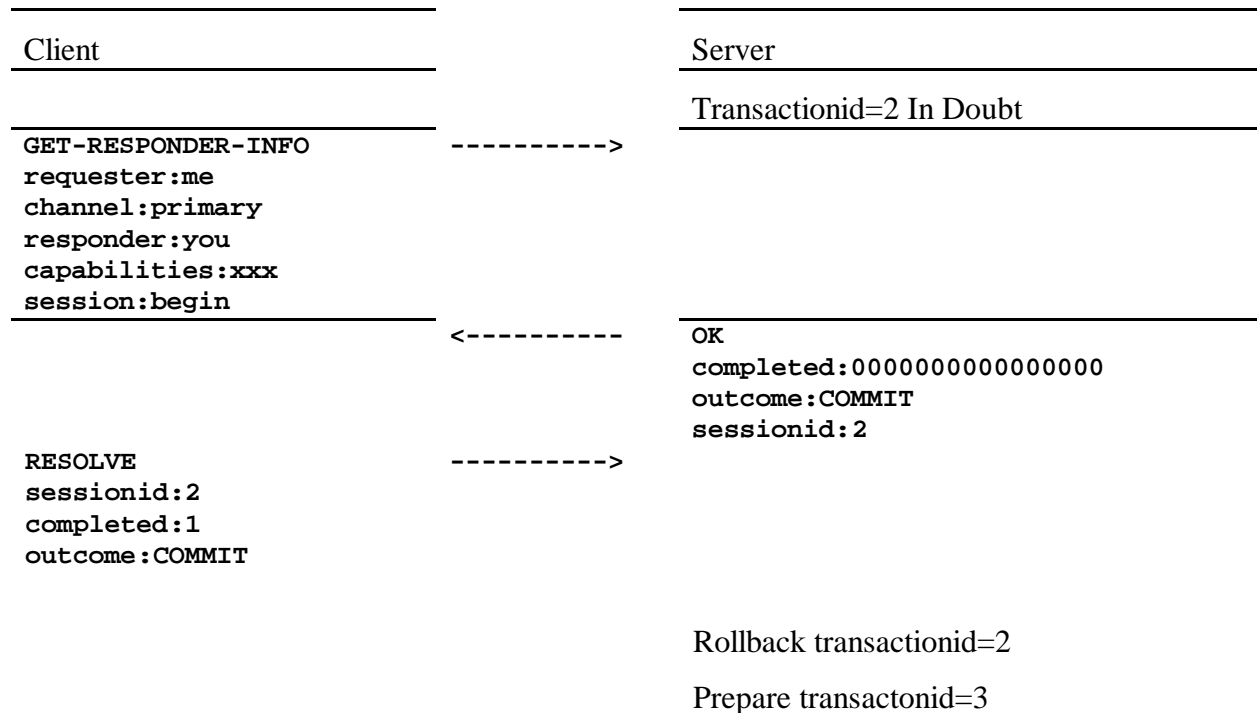
The CompletedTransactionid flowed in the PULL message indicates to the server that it can now forget the state of that transaction and remove any messages associated with it. This will be the transactionid received as the response to a previous PULL. The client may discard the state of the transaction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

### 5.6. Client fails then restarts PULL



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



```

<----- OK
PULL ----->
sessionid:2
<----- OK
transactionid:3

Commit transactionid=3

Forget transactionid=1
REPORT ----->
sessionid:2
completed:3
outcome:COMMIT

Commit transactionid=3

Forget transactionid=3
OK
session:end
<-----

```

Forget transactionid=3

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.

### 5.7. Client initiates pipelined PULL

The client uses pipelines session http 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
GET-RESPONDER-INFO -----> requester:me channel:primary responder:you capabilities:xxx session:begin	<----- OK completed:0000000000000000 outcome:COMMIT sessionid:1
PULL -----> sessionid:1	
PULL ----->	

```

sessionid:1
                                     Prepare Transactionid=1
                                     OK
                                     transactionid:1
                                     <-----
Commit transactionid=1
                                     Prepare transactonid=2
                                     OK
                                     transactionid:2
                                     <-----
Commit transactionid=2
PULL
                                     ----->
sessionid:1
completed:2
outcome:COMMIT
                                     Prepare transactonid=3
                                     Commit transactionid=1,2
                                     Forget transactionid=1,2
                                     OK
                                     transactionid:3
                                     <-----
Commit transactionid=3
Forget transactionid=1,2
REPORT
                                     ----->
sessionid:1
session:end
completed:3
outcome:COMMIT
Forget transactionid=3
                                     Forget transactionid=3
                                     OK
                                     <-----

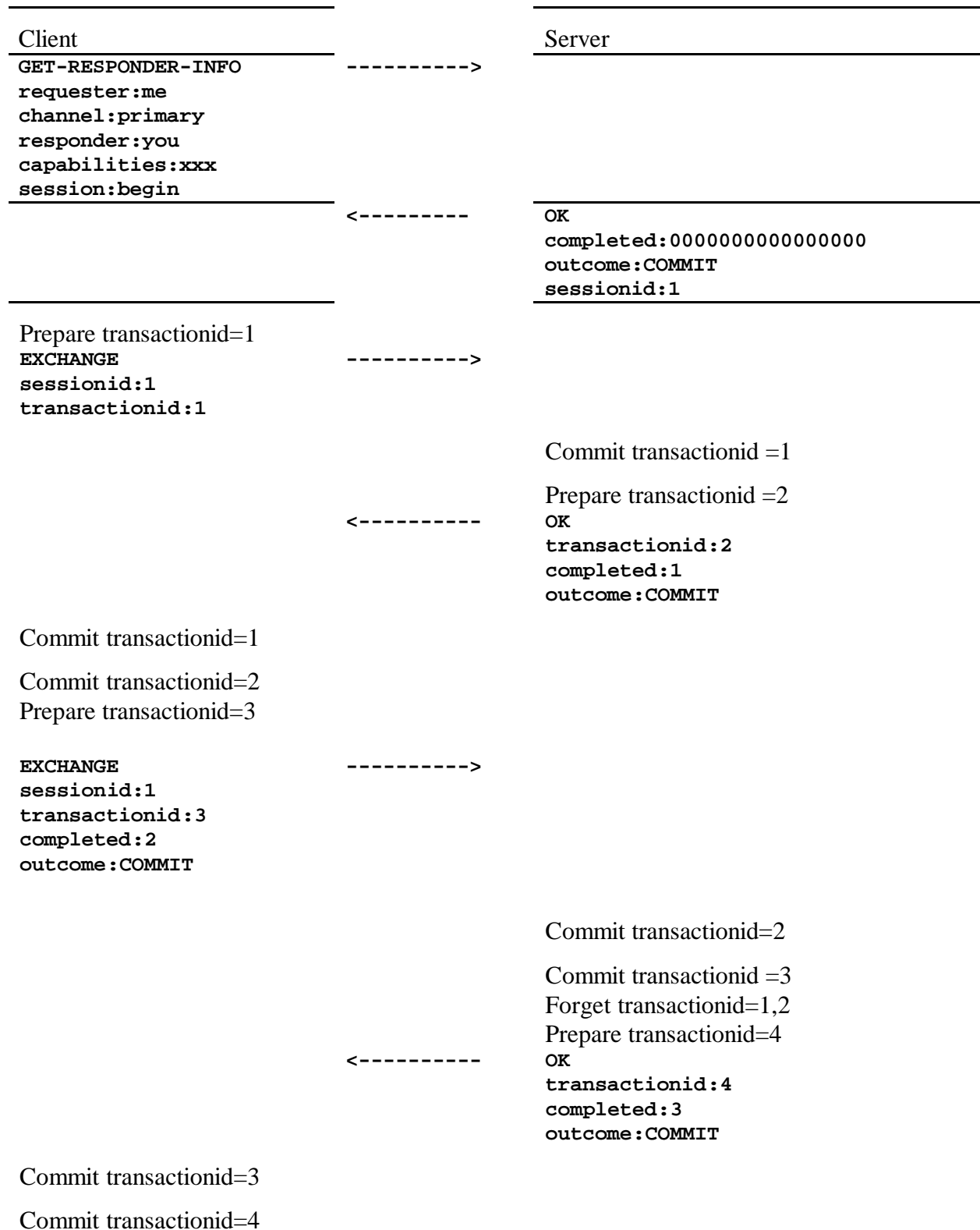
```

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

## 5.8. Client initiates EXCHANGE

The client uses simple session http 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.



```

Forget transactionid=1,2
REPORT ----->
sessionid:1
session:end
completed:4
outcome:COMMIT
forget:3

Commit transactionid=4
Forget transactionid=3,4
OK <-----

Forget transactionid=3,4

```

### 5.9. 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		
	<-----	OK
		completed:0000000000000000
		outcome:COMMIT
		sessionid:1
Prepare transactionid=1		
PUSH	----->	
sessionid:1		
transactionid=1		
	<-----	Commit transactionid =1
		OK
		completed:1
		outcome:COMMIT
Commit transactionid=1		
PULL	----->	
sessionid:1		
		Prepare transactonid=2
	<-----	OK
		transactionid:2

Commit transactionid=2

REPORT

sessionid:1

session:end

completed:2

outcome:COMMIT

forget:1

----->

Commit transactionid=2

Forget transactionid=1,2

OK

<-----

Forget transactionid=1,2

## 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 http 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 http 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 http 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. This MUST be constant for all time 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 = [ "disconnect_interval="DisconnectInterval ]
             | [ "batch_interval="BatchInterval ]
             | [ "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.

```
DisconnectInterval = 1*DIGIT
```

The number of seconds that the session **MAY** remain active for without a new request being flowed by the client. The default value is 10 seconds. The response value (if any) **MUST** be less than or equal to the request value.

```
BatchInterval = 1*DIGIT
```

When a PULL request is received, the server may not have any messages intended for this client on hand, or may not have enough to fill a batch. The `BatchInterval` is the number of milliseconds that the server should wait for more messages to appear (from whatever message queuing subsystem, or other message source, it might be using) before completing the payload. After this length of time the server **MUST** complete its response, if the payload contains at least one message. 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 outstanding 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 http 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 http. They enable product specific data to be exchanged that is not defined within the http 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 SHOULD 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 httptr 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 and the message context followed by the message data.

The message header is parsed by the agent however the message context may simply be copied to the next link in the chain of agents in a multi hop connection. The MessageHeader contains information useful for each hop in the connection between ultimate source and the ultimate sink, the MessageContext and MessageData remain unchanged.

```
Payload=MessageHeader
    CRLF                               ;End of the message header
    MessageContext
    CRLF                               ;End of message context
    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]
    *[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. It is recommended that agents be able to parse the Destination constructed as QueueName”@”QueueMangerName amongst other formats.

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 of the message flow to the sink. It may be used by the sink agent to determine if the message is acceptable and can use the resources it needs.

### 6.2.7. ProductSpecificMessageField

ProductSpecificMessageField is any field compliant with the http format rules that is not defined in the version of httptr 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 httptr. They enable product specific data to be exchanged that is not defined within the httptr 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 some 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 httptr field in preference to a product specific field, as this would inhibit interoperability.

### 6.2.8. MessageContext

```
MessageContext = [ "encoding:" Encoding CRLF ]
                  [ "reply-uri:" ReplyUri CRLF ]
                  [ "message-id:" MessageId CRLF ]
                  [ "correlation-id:" CorrelationId CRLF ]
                  [ "put-time:" PutTime CRLF ]
                  [ "expiry:" Expiry CRLF ]
                  [ "report-options:" ReportOptions CRLF ]
                  [ "content-type:" ContentType CRLF ]

                  *[ProductSpecificMessageField CRLF]
```

If the agent is not the ultimate destination for a message in the payload because it does not recognise the ServiceName in the TargetUri, all ProductSpecificFields in the MessageContext MUST be stored and forwarded intact to the next agent in the multi-hop configuration.

### 6.2.9. 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

### 6.2.10. ReplyUri:

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

### 6.2.11. MessageId:

token

A non unique identifier of the message.

### 6.2.12. CorrelationId:

token

Another non unique identifier of the message. Where a reply is to be generated by an application, the message-id of the request is often copied by the application into the correlation identifier of the 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 to distinguish replies.

HTTPR is not aware of the request/reply relationship. There is no http 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 http EXCHANGE sequences.

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

### 6.2.13. PutTime:

```
http-r-date
```

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

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

All http 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.

### 6.2.14. 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

### 6.2.15. ReportOptions:

```
ReportOptions = "COA"           ;confirm on arrival
                [ ",", "COD" ]   ;confirm on delivery
```

Confirm on arrival means that the agent receiving the application message **MUST** send a `???`COA report message`???` to the ReplyUri `???` Why not a separate ReportUri`???` if it is not forwarding the message to another TargetUri.

Confirm on delivery means that the agent receiving the application message **MUST** send a `???` COD report message `???` to the ReplyUri when an application processes the message.

### 6.2.16. ContentType

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

### 6.2.17. 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 http being re-chunked for HTTP. However, no confusion should result, as dechunking is applied first at the HTTP level, and then, separately, at the http level.

### 6.2.18. 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.

### 6.2.19. 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.

### 6.2.20. 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 http message.

#### **6.2.21. ErrorText**

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

## Appendix

### A1. Glossary

<i>agent</i>	The software transferring the message payload on behalf of the application, also the resource manager for transactions.
<i>application message</i>	The message being transferred as the application sees it, as distinct from the http message.
<i>capabilities</i>	A vector describing the http protocol capabilities and parameters requested or agreed to on this command flow or session
<i>channel</i>	An independent http conversation globally uniquely identified by the triplet: <client URI, channel identifier, server URI >.
<i>client</i>	The agent initiating the communication. The agent sending the http request message.
<i>command flow</i>	The basic unit of http 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.
<i>commit</i>	The action of permanently recording that the payload has been received, or sent.
<i>connection</i>	The TCP/IP communications used to carry the requests and responses. This has the same lifetime as the TCP/IP socket.
<i>forget</i>	The point at which the agent is no longer required to have knowledge of the transaction associated with a payload.
<i>http</i>	HyperText Transfer Protocol.
<i>http payload</i>	The application messages being transferred, together with their http message header and context information.
<i>multi-hop</i>	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.
<i>prepare</i>	The action of permanently recording that a payload is in doubt as to whether the partner agent has received it or not.
<i>request</i>	The http request message sent by the client.
<i>resource manager</i>	The software that stores persistent state and manages transactions.
<i>response</i>	The http response message sent by the server.
<i>server</i>	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.