The Dojo Foundation

Request for Comments: not an RFC
Obsoletes: protocol.txt
Category: Standards Track

Alex Russell
Greg Wilkins
David Davis
Mark Nesbitt

Bayeux Protocol -- Bayeux 1.0draft1

Status of this Memo

This document specifies a protocol for the Internet community, and requests discussion and suggestions for improvement. This memo is written in the style and spirit of an IETF RFC but is not, as of yet, an official IETF RFC. Distribution of this memo is unlimited. This memo is written in UK English.

Copyright Notice

Abstract

Bayeux is a protocol for routing JSON encoded events between clients and servers in a publish subscribe model. The protocol is designed to overcome the client/server nature of the internet in general and specifically of HTTP to allow asynchronous messaging between all participants.

Table of Contents

1. Introduction

1.1. Purpose

Bayeux is a protocol for transporting asynchronous messages with low latency, primarily over HTTP. The messages are routed via named channels and can be delivered: server to client, client to server and client to client (via the server). The primary purpose of Bayeux is to implement responsive user interactions for web clients using Ajax and the server-push technique called Comet.

Bayeux seeks to reduce the complexity of developing Comet-driven applications by allowing implementers to more easily interoperate, solve common message distribution and routing problems, and provide mechanisms for incremental improvement and extension.

1.2. 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 RFC2119. 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."

1.3. Terminology

This specification uses a number of terms to refer to the roles played by participants in, and objects of, Bayeux communication:

client
A program that initiates communications. A HTTP client is a client that initiates TCP/IP connections for the purpose of sending HTTP requests. A Bayeux client initiates the Bayeux message exchange and will typically execute within a HTTP client, but it is likely to have Bayeux clients that execute within HTTP servers. Implementations may distinguish between Bayeux clients running within a HTTP client and Bayeux clients running within the HTTP server. Specifically server-side Bayeux clients MAY be privileged clients with access to private information about other clients (e.g. client IDs) and subscriptions.

server
An application program that accepts communications from clients. A HTTP server accepts TCP/IP connections in order to service HTTP requests by sending back HTTP responses. A Bayeux server accepts and responds to the message exchanges initiated by a Bayeux client.

request
An HTTP request message as defined by section 5 of RFC 2616

response
A HTTP response message as defined by section 6 of RFC 2616

message
A message is a JSON object exchanged between client and server for the purposed of implementing the Bayeux protocol as defined by sections 3, 4 and 5.

event
Application specific data that is sent over the Bayeux protocol

envelope
The transport specific message formate that wraps a standard Bayeux message.

channel
A named destination and/or source of events. Events are published to channels and subscribers to channels receive the published events.

connection
A connection is a communication link that is established either permanently or transiently, for the purposes of messages exchange. A client is connected if a link is established with the server, over which asynchronous events can be received.

JSON
JavaScript Object Notation(JSON) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999. JSON is described at http://www.json.org/

1.4. Overall Operation

1.4.1. HTTP

The HTTP protocol is a request/response protocol. A client sends a request to the server in the form of a request method, URI, and protocol version, followed by a MIME-like message containing request modifiers, client information, and optional body content over a connection with a server. The server responds with a status line, including the message's protocol version and a success or error code, followed by a MIME-like message containing server information, entity metainformation, and possible entity-body content.

The server may not initiate a connection with a client nor send an unrequested response to the client, thus asynchronous events cannot be delivered from server to client unless a previously issued request exists. In order to allow two way asynchronous communication, Bayeux supports the use of multiple HTTP connections between a client and server, so that previously issued requests are available to transport server to client messages.

The recommendation of section 8.1.4 of RFC 2616 is that a single user client SHOULD NOT maintain more than 2 connection with any server, thus the Bayeux protocol MUST NOT require any more than two HTTP requests to be simultaneously handled by a server in order to handle all application (Bayeux based or otherwise) running within a client.

1.4.2. Non HTTP Transports

While HTTP is the predominant transport protocol used on the internet, it is not intended that it will be the only transport for

Bayeux. Other transports that support a request/response paradigm may be used. However this document assumes HTTP for reasons of clarity. When non-HTTP connection-level transport mechanisms are employed, conforming Bayeux servers and clients MUST still conform to the semantics of the JSON messages outlined in this document.

Several of the "transport types" described in this document are distinguished primarily by how they wrap messages for delivery over HTTP and the sequence and content of the HTTP connections initiated by clients. While this may seem like a set of implementation concerns to observant readers, the difficulties of creating interoperable implementations without specifying these semantics fully is a primary motivation for the development of this specification. Were the deployed universe of servers and clients more flexible, it may not have been necessary to develop Bayeux.

Regardless, care has been taken in the development of this specification to ensure that future clients and servers which implement differing connection-level strategies and encodings may still evolve and continue to be conforming Bayeux implementations so long as they implement the JSON-based public/subscribe semantics outlined herein.

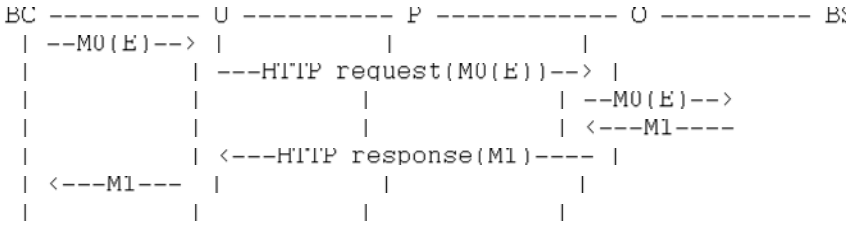
The rest of this document speaks as though HTTP will be used for message transport.

1.4.3. Javascript

The majority of Bayeux clients will be implemented in JavaScript and will be running within the security framework of a client browser. For applications that need to communicate with multiple servers, the client implementation MUST adhere to the single origin policy for security.

1.4.4. Client to Server event delivery

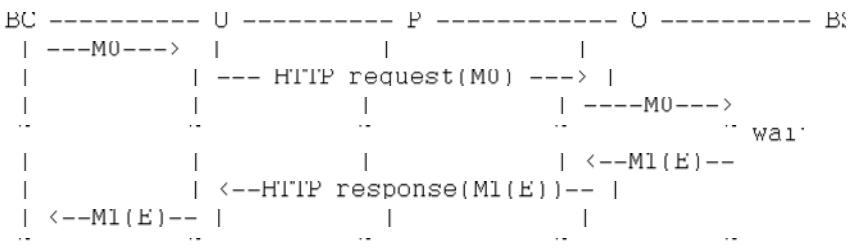
A Bayeux event is sent from the client to the server via a HTTP request initiated by a user agent and transmitted to an origin server via a chain of zero or more intermediaries (proxy, gateway or tunnel):



The figure above represents a Bayeux event E encapsulated in a Bayeux message M0 being sent from a Bayeux client BC to a Bayeux server BS via a HTTP request transmitted from a User Agent U to to an Origin server O via a proxy P. The HTTP response contains another Bayeux message M1 that will at least contain the protocol response to M0, but may contain other Bayeux events initiated on the server or on other clients.

1.4.5. Server to Client event delivery

A Bayeux event is sent from the server to the client via a HTTP response to a HTTP request sent in anticipation by a user agent and transmitted to an origin server via a chain of zero or more intermediaries (proxy, gateway or tunnel):



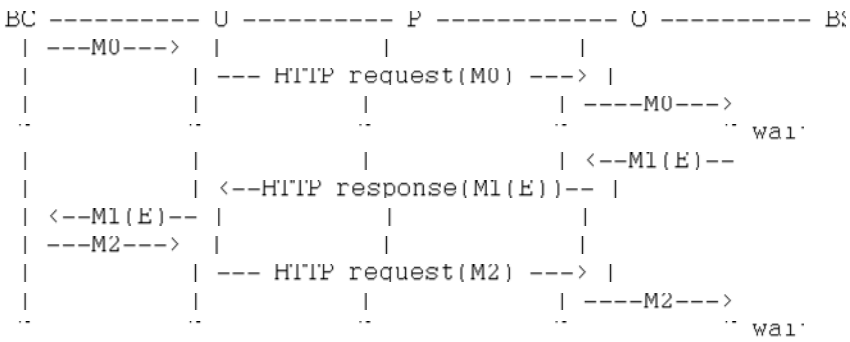
The figure above represents a Bayeux message M0 being sent from a Bayeux client BC to a Bayeux server BS via a HTTP request transmitted from a User Agent U to to an Origin server O via a proxy P. The message M0 is sent in anticipation of

a Bayeux event to be delivered from server to client and the Bayeux server waits for such an event before sending a response. A Bayeux event E is shown being delivered via Bayeux message M1 in the HTTP response. M1 may contain zero, one or more Bayeux events destined for the Bayeux client.

The transport used may terminate the HTTP response after delivery of M1 or use techniques to leave the response open and stream additional messages to the client.

1.4.5.i Polling transports

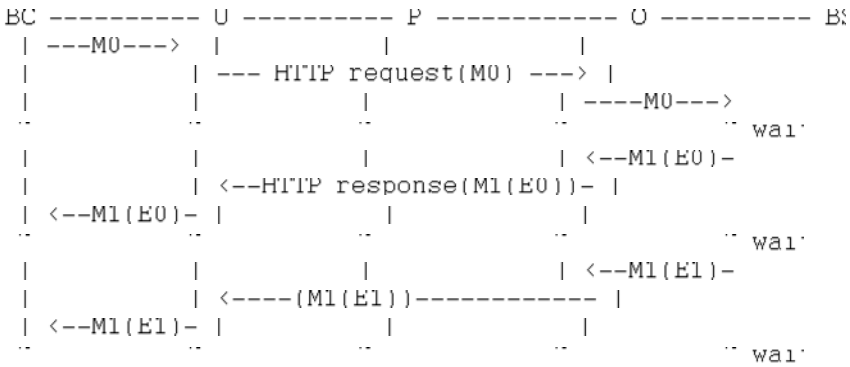
Polling transports will always terminate the HTTP response after sending all available Bayeux messages.



On receipt of the HTTP response containing M1, the Bayeux client issues a new Bayeux message M2 either immediately or after an interval in anticipation of more events to be delivered from server to client. Bayeux implementations are required to support a specific style of polling transport called "long polling" (see sec 6.1).

1.4.5.ii Streaming transports

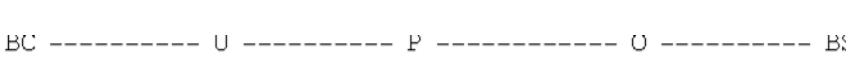
Some Bayeux transports use a streaming technique (also called a forever response) that allows multiple messages to be sent over the same HTTP response:

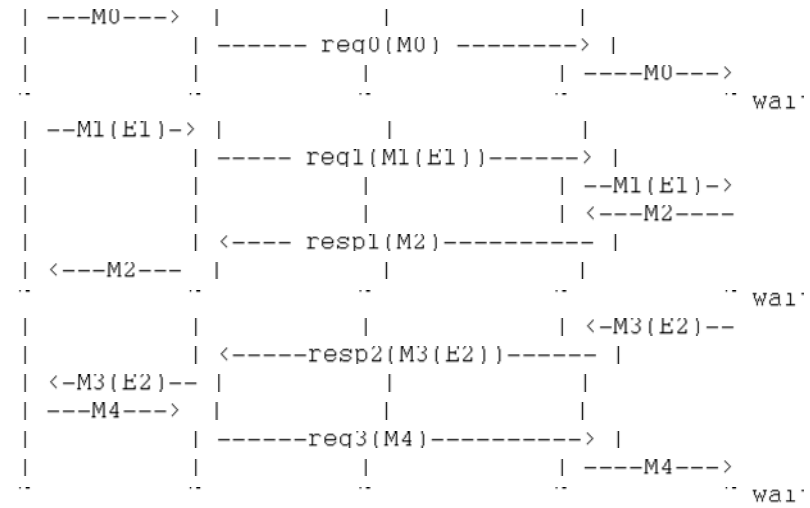


Streaming techniques avoid the latency and extra messaging of anticipatory requests, but are subject to the implementation of user agents and proxies as they requires incomplete HTTP responses to be delivered to the Bayeux client.

1.4.6. Two connection operation

In order to achieve bi-directional communications, a Bayeux client will use two HTTP connections to a Bayeux server so that both server to client and client to server messaging may occur asynchronously:

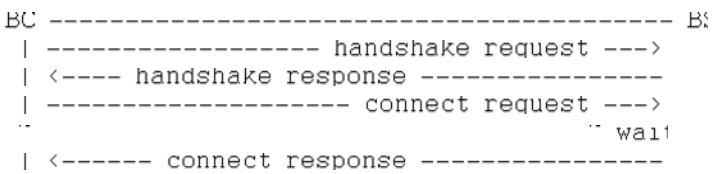




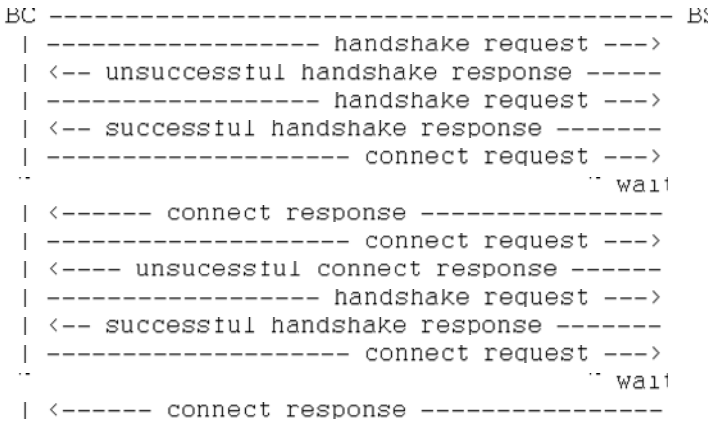
HTTP requests req0 and req1 are sent on different TCP/IP connections, so that the response to req1 may be sent before the response to req0. Implementations MUST control HTTP pipelining so that req1 does not get queued behind req0 and thus enforce an ordering of responses.

1.4.7. Connection Negotiation

Bayeux connections are negotiated between client and server with handshake messages that allow the connection type, authentication and other parameters to be agreed upon between the client and the server.



Connection negotiation may be iterative and several handshake messages may be exchanged before a successful connection is obtained. Servers may also request connection renegotiation by sending an unsuccessful connect response with advice to reconnect with a handshake message.



1.4.8. Unconnected operation

OPTIONALLY, messages can be sent without a prior handshake (see 5.1 Publish event messages).

```
| -------------------- message request ----->
| <---- message response ------------------
```

This pattern is often useful when implementing non-browser clients for Bayeux servers. These clients often simply wish to address messages to other clients which the Bayeux server may be servicing, but do not wish to listen for events themselves.

1.5 State Tables

1.5.1 Client State

```
-------------++------------+-------------+----------- +-----------
State/Event  || handshake  | Timeout     | Successful | Disconnect
             ||  request   |             |   connect  |  request
             ||   sent     |             |  response  |   sent
-------------++------------+-------------+----------- +-----------
UNCONNECTED  || CONNECTING | UNCONNECTED |
CONNECTING   ||            | UNCONNECTED | CONNECTED  | UNCONNECTED
CONNECTED    ||            | UNCONNECTED |            | UNCONNECTED
-------------++------------+-------------+------------+-----------
```

2. Protocol values

2.1. Common Elements

The characters used for Bayeux names and identifiers are defined by the BNF definitions:

```
alpha    = lowalpha | upalpha

lowalpha = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i"
         | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r"
         | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"

upalpha  = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I"
         | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R"
         | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"

digit    = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7"
         | "8" | "9"

alphanum = alpha | digit

mark     = "-" | "_" | "!" | "~" | "(" | ")" | "$" | "@"

string   = *( alphanum | mark | " " | "/" | "*" | "." )

token    = ( alphanum | mark ) *( alphanum | mark )

integer  = digit *( digit )
```

2.2. Channel

Channels are identified by names that are styled as the absolute path component of a URI without parameters as defined by RFC2396.

```
channel_name     = "/"  channel_segments
channel_segments = channel_segment *( "/" channel_segment
channel_segment  = token
```

The channel name consists of an initial "/" followed by an optional sequence of path segments separated by a single slash "/" character. Within a path segment, the character "/" is reserved.

Channel names commencing with "/meta/" are reserved protocol use. Example non-meta channel names are:

/foo
/foo/bar
/foo-bar/(foobar)

2.2.1 Channel Globbing

A set of channels may be specified with a channel globbing pattern:

```
channel_pattern  = *( "/" channel_segment ) "/" wild_car
wild_card = "*" | "**"
```

The channel patterns support only trailing wildcards of either "*" to match a single segment or "**" to match multiple segments. Example channel patterns are:

/foo/*
Matches /foo/bar and /foo/boo. Does not match /foo, /foobar or /foo/bar/boo.

/foo/**
Matches /foo/bar, /foo/boo and /foo/bar/boo. Does not match /foo, /foobar or /foobar/boo

2.2.2 Meta Channel

The channels within the "/meta/" segment are the channels used by the Bayeux protocol itself. Local server-side Bayeux clients MAY, and remote Bayeux clients SHOULD NOT, subscribe to meta channels. Messages published to meta channels MUST NOT be distributed to remote clients by Bayeux servers. A server side handler of a meta channel MAY publish response messages that are delivered only to the client that sent the original request message. If a message published to a meta channel contains an id field, then any response messages delivered to the client MUST contain an id field with the same value.

2.2.3 Service Channel

The channels within the "/service/" channel segement are special channels designed to assist request/response style messaging. Messages published to nservice channels are not distributed to any remote Bayeux clients. Handlers of service channels MAY deliver response messages to the client that published the request message. Servers SHOULD NOT record any subscriptions they receive for service channels. If a message published to a meta channel contains an id field, then any response messages SHOULD contain an id field with the same value or a value derived from the request id. Request response operations are described in detail in section 9.

2.3. Version

A protocol version is a integer followed by an optional "." separated sequence of alphanumeric elements:

```
version         = integer *( "." version_element
version_element = alphanum *( alphanum | "-" | "_"
```

Versions are compared element by element, applying normal alphanumeric comparison to each element.

2.4. Client ID

A client ID is an random, non predictable sequence of alpha numeric characters:

```
clientId   =   alphanum *( alphanum
```

Client IDs are generated by the server and SHOULD be created with a strong random algorithm that contains at least 128 truly random bits. Servers MUST ensure that client IDs are unique and SHOULD attempt to avoid reuse of client IDs. Client IDs are encoded for delivery as JSON strings.

2.5 Messages

Bayeux messages are JSON encoded objects that contain an unordered sequence of name value pairs representing fields and values. Values may be a simple strings, numbers, boolean values, or complex JSON encoded objects. A Bayeux message MUST contain one and only one channel field which determines the type of the message and the allowable fields.

All Bayeux messages SHOULD be encapsulated in a JSON array so that multiple messages may be transported together. A Bayeux client or server MUST accept either array of messages and MAY accept a single message. The JSON message or array of messages is itself often encapsulated in transport specific formatting and encodings. Below is an example Bayeux message in a JSON array representing an event sent from a client to a server:

```
[
  "channel": "/some/name",
  "clientId": "83js73jsh29sjd92"
  "data": { "myapp" : "specific data", value: 100
]
```

3. Message Field Definitions

3.1. channel

The channel message field MUST be included in every Bayeux message to specify the source or destination of the message. In a request, the channel specifies the destination of the message, and in a response it specifies the source of the message.

3.2. version

The version message field MUST be included in messages to/from the "/meta/handshake" channel to indicate the protocol version expected by the client/server.

3.3. minimumVersion

The minimumVersion message field MAY be included in messages to/from the "/meta/handshake" channel to indicate the oldest protocol version that can be handled by the client/server.

3.4. supportedConnectionTypes

The supportedConnectionTypes field is included in messages to/from the "/meta/handshake" channel to allow clients and servers to reveal the transports that are supported. The value is an array of strings, with each string representing a transport name. Defined connection types include:

long-polling

messages are sent to the server as the 'message' parameter of a application/x-www-form-urlencoded encoded POST request. Messages are sent to the client as unencapsulated body content of a POST response. This transport is defined in section [XXX] of this memo.

callback-polling
: messages are sent to the server as the 'message' parameter of a url encoded GET request. Responses are sent wrapped in a JavaScript callback in order to facilitate delivery. As specified by the JSON-P pseudo-protocol, the name of the callback to be triggered is passed to the server via the 'jsonp' GET parameter. In the absence of such a parameter, the name of the callback defaults to 'jsonpcallback'.

iframe
: OPTIONAL transport using the document content of a hidden iframe element.

flash
: OPTIONAL transport using the capabilities of a browser flash plugin.

All server and client implementations MUST support the "long-polling" connection type and SHOULD support "callback-polling". All other connection types are OPTIONAL.

3.5. clientId

The clientId message field uniquely identifies a client to a Bayeux server. The clientId message field MUST be included in every message sent to the server except for a messages sent to the "/meta/handshake" channel and a publish message (see 5.1 Publish event messages). The clientId field MUST be returned in every message response except for a failed handshake request and is OPTIONAL in a message delivery message.

3.6. advice

The advice field provides a way for servers to inform clients of their preferred mode of client operation so that in conjunction with server-enforced limits, Bayeux implementations can prevent resource exhaustion and inelegant failure modes.

The advice field is a JSON map containing general and transport specific values that indicate modes of operation, timeouts and other potential transport specific parameters. Fields may occur either in the top level of an advice or within a transport specific section.

Unless otherwise specified in sections 5 and 6, any Bayeux response message may contain an advice field. Advice received always superceeds any previous received advice.

An example advice field is

```
"advice": {
   "reconnect": "retry",
   "interval": 1000,
   "callback-polling" : {
      "reconnect": "handshake'
```

3.6.1. reconnect advice

The reconnect advice field is a string that indicates how the client should act in the case of a failure to connect. Defined reconnect values are:

retry
: a client MAY attempt to reconnect with a /meta/connect after the interval (as defined by "interval" advice or client-default backoff), and with the same credentials.

handshake
: the server has terminated any prior connection status and the client MUST reconnect with a /meta/handshake. A client MUST NOT automatically retry if handshake advice has been received.

none

hard failure for the connect attempt. Do not attempt to reconnect at all. A client MUST respect reconnect advice of none and MUST NOT automatically retry or handshake.

Any client that does not implement all defined values of reconnect MUST NOT automatically retry or handshake.

3.6.2. interval advice

An integer representing the minimum period in milliseconds for a client to delay subsequent requests to the /meta/connect channel. A negative period indicates that the message should not be retried.

A client MUST implement interval support, but a client MAY exceed the interval provided by the server. A client SHOULD implement a backoff strategy to increase the interval if requests to the server fail without new advice being received from the server.

3.6.3. multiple-clients advice

This is a boolean field, which if true indicates that the server has detected multiple Bayeux client instances running within the same HTTP client.

3.6.4. hosts advice

This is an array of strings field, which if present indicates a list of host names or IP addresses that MAY be used as alternate servers with which the client may connect. If a client receives advice to re-handshake and the current server is not included in a supplied hosts list, then the client SHOULD try the hosts in order until a successful connection is establish. Advice received during handshakes with hosts in the list supercedes any previously received advice.

3.7. connectionType

The connectionType message field specifies the type of transport the client requires for communication. The connectionType message field MUST be included in request messages to the "/meta/connect" channel. Connection types are defined in section 4.7.

3.8. id

An id field MAY be included in any Bayeux message with an alpha numeric value:

```
id   =   alphanum *( alphanum
```

Generation of IDs is implementation specific and may be provided by the application. Messages published to /meta/** and /service/** SHOULD have id fields that are unique within the the connection.

Messages sent in response to messages delivered to /meta/** channels MUST use the same message id as the request message.

Messages sent in response to messages delivered to /service/** channels SHOULD use the same message id as the request message or an id derived from the request message id.

3.9. timestamp

The timestamp message field SHOULD be specified in the following ISO 8601 profile: All times SHOULD be sent in GMT time.

```
YYYY-MM-DDThh:mm:ss.ss
```

A timestamp message field is OPTIONAL in all Bayeux messages.

3.10. data

The data message field is an arbitrary JSON object that contains event information. The data field MUST be included in publish requests, and a Bayeux server MUST include the data field in an event delivery message.

3.11. connectionId

The connectionId field was used during development of the Bayeux protocol and its use is now deprecated.

3.12. successful

The successful boolean message field is used to indicate success or failure and MUST be included in responses to the "/meta/handshake", "/meta/connect", "/meta/subscribe","/meta/unsubscribe", "/meta/disconnect", and publish channels.

3.13. subscription

The subscription message field specific the channels the client wishes to subscribe to or unsubscribe from. The subscription message field MUST be included in requests and responses to/from the "/meta/subscribe" or "/meta /unsubscribe" channels.

3.14. error

The error message field is OPTIONAL on any Bayeux response. The error message field MAY indicate the type of error that occurred when a request returns with a false successful message. The error message field should be sent as a string in the following format:

```
error           = error_code ":" error_args ":" error_message
                | error_code ":" ":" error_message
error_code       = digit digit digit
error_args       = string *( "," string
error_message    = strin
```

Example error strings are:

```
401::No client Id
402:xj3sjdsjdsjad:Unknown Client Id
403:xj3sjdsjdsjad,/foo/bar:Subscription denie
404:/foo/bar:Unknown Channe
```

Need to provide list of codes

3.15. ext

An ext field MAY be included in any Bayeux message. Its value SHOULD be a JSON map with top level names distinguished by implementation names (eg. "org.dojo.Bayeux.field").

The contents of ext may be arbitrary values that allow extensions to be negotiated and implemented between server and client implementations.

4. Meta Message Definitions

4.1. handshake

4.1.1. handshake Request

A Bayeux client initiates a connection negotiation by sending a message to the "/meta/handshake" channel. For same domain connections, the Handshake requests MUST be sent to the server as the 'message' parameter of an application/x-www-form-urlencoded encoded POST request. For cross domain connections, the Handshake request MUST be sent to the server as a url encoded GET request with the jsonp parameter set for callback-polling.

A handshake request MUST contain the message fields:

channel
value "/meta/handshake"

version
The version of the protocol supported by the client.

supportedConnectionTypes
An array of the connection types supported by the client for the purposes of the connection being negotiated. This list MAY be a subset of the connection types actually supported if the client wishes to negotiate a specific connection type.

A handshake request MAY contain the message fields:

minimumVersion
ext
id

A client SHOULD NOT send any other message in the request with a handshake message. A server MUST ignore any other message sent in the same request as a handshake message. An example handshake request is:

```
"channel": "/meta/handshake"
"version": "1.0"
"minimumVersion": "1.0beta"
"supportedConnectionTypes": ["long-polling", "callback-polling", "iframe"
```

4.1.2. handshake Response

A Bayeux server MUST respond to a handshake request with a handshake response message in the body content of the response. For cross domain connections that have the 'jsonp' parameter set, the message body may be encapsulated in a jsonp callback method.

Successful handshake response

A successful handshake responses MUST contain the message fields:

channel
value "/meta/handshake"

version

supportedConnectionTypes
The connection types supported by the server for the purposes of the connection being negotiated. This list MAY be a subset of the connection types actually supported if the server wishes to negotiate a specific connection type. This list MUST contain at list one element in common with the supportedConnectionType provided in the handshake request. If there are no connectionTypes in common, the handshake response MUST be unsuccessful.

clientId
A newly generated unique clientId.

successful
value true

A successful handshake response MAY contain the message fields:

minimumVersion
advice

ext
id
same value as request message id

authSuccessful
Value true, this field may be included to support prototype client implementations that required the authSuccessful field

An example successful handshake response is:

```
"channel": "/meta/handshake"
"version": "1.0"
"minimumVersion": "1.0beta"
"supportedConnectionTypes": |"long-polling","callback-polling"|
"clientId": "Un1q31d3nt1f13r"
"successful": true
"authSuccessful": true
"advice": { "reconnect": "retry"
```

Unsuccessful handshake response

An unsuccessful handshake response MUST contain the message fields:

channel
value "/meta/handshake"

successful
value false

error
a string with the description of the reason for the failure.

An unsuccessful handshake response MAY contain the message fields:

supportedConnectionTypes
The connection types supported by the server for the purposes of the connection being negotiated. This list MAY be a subset of the connection types actually supported if the server wishes to negotiate a specific connection type.

advice
version
minimumVersion
ext
id
same value as request message id

An example unsuccessful handshake response is:

```
"channel": "/meta/handshake"
"version": "1.0"
"minimumVersion": "1.0beta"
"supportedConnectionTypes": |"long-polling","callback-polling"|
"successful": false
"error": "Authentication failed"
"advice": { "reconnect": "none"
```

For complex connection negotiations, multiple handshake messages may be exchanged between the Bayeux client and server. The handshake response will set the "successful" field to false until the handshake processs is complete. The advice and ext fields may be used to communicate additional information needed to complete the handshake process. An unsuccessful handshake response with reconnect advice of "handshake" is used to continue the connection negotiation. An

unsuccessful handshake response with reconnect advice of "none" is used to terminate connection negotiations.

4.2. connect

4.2.1. connect Request

After a Bayeux client has discovered the server's capabilities with a handshake exchange, a connection is established by sending a message to the "/meta/connect" channel. This message may be transported over any of the transports indicated as supported by the server in the handshake response.

A connect request MUST contain the message fields:

channel
value "/meta/connect"

clientId
The client ID returned in the handshake response

connectionType
The connection type used by the client for the purposes of this connection.

A connect request MAY contain the message fields:

ext
id

A client MAY send other messages in the same HTTP request with a connection message. A server MUST handle any other message sent in the same request as a connect message after the handling of the connect message is complete.

An example connect request is:

```
"channel": "/meta/connect"
"clientId": "Un1q31d3nt1f13r"
"connectionType": "long-polling"
```

A transport MUST maintain one and only one outstanding connect message. When a HTTP response that contains a /meta/connect response terminates, the client MUST wait at least the interval specified in the last received advice before following the advice to reestablish the connection

4.2.2. connect Response

A Bayeux server MUST respond to a connect request with a connect response message over the same transport as used for the request.

A Bayeux server MAY wait to respond until there are event messages available in the subscribed channels for the client that need to be delivered to the client.

A connect responses MUST contain the message fields:

channel
value "/meta/connect"

successful
boolean indicating the success or failure of the connection

clientId
The negotiated client ID

A connect response MAY contain the message fields:

error
advice
ext
id
same value as request message id
timestamp

An example connect response is:

```
"channel": "/meta/connect"
"successful": true
"error": ""
"clientId": "Un1q31d3nt1f13r"
"timestamp": "12:00:00 1970"
"advice": { "reconnect": "retry"
```

The client MUST maintain only a single outstanding connect message. If the server does not have a current outstanding connect and a connect is not received within a configured timeout, then the server SHOULD act as if a disconnect message has been received.

4.4. disconnect

4.4.1. disconnect Request

When a connected client wishes to cease operation it should send a request to the "/meta/disconnect" channel for the server to remove any client-related state. The server SHOULD release any waiting meta message handlers. Bayeux client applications should send a disconnect request when the user shuts down a browser window or leaves the current page. A Bayeux server SHOULD not rely solely on the client sending a disconnect message to remove client-related state information because a disconnect message might not be sent from the client or the disconnect request might not reach the server.

A disconnect request MUST contain the message fields:
channel
value "/meta/disconnect"
clientId
The client ID returned in the handshake response

A disconnect request MAY contain the message fields:

ext
id
An example disconnect request is:

```
"channel": "/meta/disconnect"
"clientId": "Un1q31d3nt1f13r'
```

4.4.2. disconnect Response

A Bayeux server MUST respond to a disconnect request with a disconnect response.

A disconnect response MUST contain the message fields:

channel
value "/meta/disconnect"

clientId
The client ID returned in the handshake response

successful
boolean value indicated the success or failure of the disconnect request

A disconnect response MAY contain the message fields:

error
ext
id
same value as request message id

An example disconnect response is:

```
{
  "channel": "/meta/disconnect",
  "clientId": "Un1q31d3nt1f13r",
  "successful": true
}
```

4.5. subscribe

4.5.1. subscribe Request

A connected Bayeux client may send subscribe messages to register interest in a channel and to request that messages published to the subscribe channel are delivered to the client.

A subscribe request MUST contain the message fields:

channel
value "/meta/subscribe"

clientId
The client ID returned in the handshake response

subscription
a channel name or a channel pattern or an array of channel names and channel patterns.

A subscribe request MAY contain the message fields:

ext
id

An example subscribe request is:

```
{
  "channel": "/meta/subscribe",
  "clientId": "Un1q31d3nt1f13r",
  "subscription": "/foo/**"
}
```

4.5.2. subscribe Response

A Bayeux server MUST respond to a subscribe request with a subscribe response message.

A Bayeux server MAY send event messages for the client in the same HTTP response as the subscribe response,

including events for the channels just subscribed to.

A subscribe responses MUST contain the message fields:

channel
value "/meta/subscribe"

successful
boolean indicating the success or failure of the subscribe

clientId
The negotiated client ID

subscription
a channel name or a channel pattern or an array of channel names and channel patterns.

A subscribe response MAY contain the message fields:

error
advice
ext
id
same value as request message id

timestamp

An example successful subscribe response is:

```
{
  "channel": "/meta/subscribe",
  "clientId": "Un1q31d3nt1f13r",
  "subscription": "/foo/**",
  "successful": true,
  "error": ""
}
```

An example failed subscribe response is:

```
{
  "channel": "/meta/subscribe",
  "clientId": "Un1q31d3nt1f13r",
  "subscription": "/bar/baz",
  "successful": false,
  "error": "403:/bar/baz:Permission Denied"
}
```

4.6. unsubscribe

4.6.1. unsubscribe Request

A connected Bayeux client may send unsubscribe messages to cancel interest in a channel and to stop published message delivery from the server to the unsubscribe channel.

A unsubscribe request MUST contain the message fields:

channel
value "/meta/unsubscribe"

clientId
The client ID returned in the handshake response

subscription
a channel name or a channel pattern or an array of channel names and channel patterns.

A unsubscribe request MAY contain the message fields:

ext

id

An example unsubscribe request is:

```
[
  {
    "channel": "/meta/unsubscribe"
    "clientId": "Un1q31d3nt1f13r"
    "subscription": "/foo/**"
  }
]
```

4.6.2. unsubscribe Response

A Bayeux server MUST respond to a unsubscribe request with a unsubscribe response message. A Bayeux server MAY send event messages for the client in the same HTTP response as the unsubscribe response, including events for the channels just unsubscribed to as long as the event was processed before the unsubscribe request.

A unsubscribe responses MUST contain the message fields:

channel
value "/meta/unsubscribe"

successful
boolean indicating the success or failure of the unsubscribe operation

clientId
The negotiated client ID

subscription
a channel name or a channel pattern or an array of channel names and channel patterns.

A unsubscribe response MAY contain the message fields:

error
advice
ext
id
same value as request message id

timestamp

An example unsubscribe response is:

```
[
  {
    "channel": "/meta/unsubscribe"
    "clientId": "Un1q31d3nt1f13r"
    "subscription": "/foo/**"
    "successful": true
    "error": ""
  }
]
```

5. Event Message Definitions

Application events are published in event messages sent from a Bayeux client to a Bayeux server and are delivered in event messages sent from a Bayeux server to a Bayeux client.

5.1. Publish event messages

5.1.1. publish Request

A Bayeux client can publish events on a channel by sending event messages. An event message MAY be sent in new HTTP request or it MAY be sent in the same HTTP request as any message other than a handshake meta message.

A publish message may be sent from an unconnected client (that has not performed handshaking and thus does not have a client ID). It is OPTIONAL for a server to accept unconnected publish requests and they should apply server specific authentication and authorization before doing so.

A publish event message MUST contain the message fields:

channel

data
The message as an arbitrary JSON object

A publish event message MAY contain the message fields:

clientId
The negotiated client ID

id
A unique ID for the message generated by the client

ext

An example event message is:

```
"channel": "/some/channel"
"clientId": "Unlq3ld3ntltl3r"
"data": "some application string or JSON object"
"id": "some unique message id'
```

5.1.2. publish Response

A Bayeux server MAY respond to a publish event message with a publish event acknowlegement.

A publish event message MUST contain the message fields:

channel

successful
boolean indicating the success or otherwise of the publish

A publish event response MAY contain the message fields:

clientId
The negotiated client ID

id

error

ext

An example event message is:

```
"channel": "/some/channel"
"clientId": "Unlq3ld3ntltl3r"
"data": "some application string or JSON object"
"id": "some unique message id'
```

5.2. Deliver Event messages

Event messages are delivered to clients if the client is subscribed to the channel of the event message. Event messages may be sent to the client in the same HTTP response as any other message other than a meta handshake response. If a Bayeux server has multiple HTTP requests from the same client, the server SHOULD deliver all available messages in the HTTP response that will be sent immediately in preference to waking a waiting connect meta message handler. Event message delivery is not acknowledged by the client.

A deliver event message MUST contain the message fields:

channel

data
The message as an arbitrary JSON object

A deliver event response MAY contain the message fields:

id

Unique message ID from the publisher

clientId
The client ID of the publisher

ext
advice

An example event deliver message is:

```
"channel": "/some/channel"
"data": "some application string or JSON object"
"id": "some unique message id'
```

6. Transports

6.1. long-polling

"Long-polling" is a polling transport that attempts to minimize both latency in server-client message delivery, and the processing/network resources required for the connection. In "traditional" polling, servers send and close responses to requests immediately, even when there are no events to deliver, and worst-case latency is the polling delay between each client request. Long-polling server implementations attempt to hold open each request until there are events to deliver; the goal is to always have a pending request available to use for delivering events as they occur, thereby minimizing the latency in message delivery. Increased server load and resource starvation are addressed by using the reconnect and interval advice fields to throttle clients, which in the worst-case degenerate to traditional polling behaviour.

6.1.1 long-polling request messages

Messages are sent to the server as the body of a POST, encoded either as "application/x-www-form-urlencoded" or as "text/json". If sent as form encoded, the Bayeux messages are sent as the "message" parameter in one of the following forms as:

- Single valued and contain a single Bayeux message
- Single valued and contain an array of Bayeux message
- Multi valued and contain a several individual Bayeux message
- Multi valued and contain a several arrays of Bayeux message
- Multi valued and contain a mix of individual Bayeux messages and arrays of Bayeux message

6.1.2 long-polling response messages

Messages are sent to the client as unencapsulated body content of a POST response with content type "text/json" or "text/json-comment-filtered".

6.2. callback-polling

6.2.1 callback-polling request messages

Messages are sent to the server either using POST requests as per long-polling transport or as the 'message' URL parameter of a GET request.

6.2.2 callback-polling response messages

Messages are sent to the client as JavaScript function call returned for script source GET requests. The function called will be determined by the 'jsonp' field of any associated request messages, or 'jsonpcallback' if not specified. The called function will be passed a JSON array of Bayeux messages.

7. Security

7.1. Authentication

Bayeux may be used with:

- No authentication
- Container supplied authentication (eg BASIC auth or cookie managed session based authentication)
- Bayeux extension authentication that exchanges authentication credentials and tokens within Bayeux messages ext fields

For Bayeux authentication, no algorithm is specified for generating or validating security credentials or token. This version of the protocol only defines that the ext field may be used to exchange authentication challenges, credentials, and tokens and that the advice field may be used to control multiple iterations of the exchange.

The connection negotiation mechanism may be used to negotiate authentication or request re-authentication.

7.2. Ajax Hijacking

The Ajax hijacking vulnerability is when an attacking web site uses a script tag to execute JSON content obtained from an Ajax server. The Bayeux protocol is not vulnerable to this style of attack as cookies are not used for authentication and a valid client ID is needed before private client data is returned. The use of POST requests further protects against this style of attack.

To futher protect against this class of attack, it is RECOMMENDED that Bayeux clients and servers support JSON comment filtered format, where the outer JSON array is enclosed in a comment that will prevent execution of the JSON without explicit handling of the comment characters. An example of a commented message is:

```
/*
[
    "channel": "/some/channel"
    "data": "some application string or JSON object"
    "id": "some unique message id"
]*/
```

An boolean ext field of json-comment-filtered should be sent by both client and server in its meta handshake exchange to indicate that comment-filtering is supported. If it is, subsequent messages (including meta connect) should be so commented, and the content-type should be text/json-comment-filtered.

An example handshake request with JSON comment support:

```
[
    "channel": "/meta/handshake"
    "version": "1.0"
    "minimumVersion": "1.0"
    "supportedConnectionTypes": ["long-polling", "callback-polling", "iframe"]
    "ext": { "json-comment-filtered": true
]
```

An example handshake response with JSON comment support:

```
[
    "channel": "/meta/handshake"
    "version": "1.0"
    "minimumVersion": "1.0"
    "supportedConnectionTypes": ["long-polling"]
    "clientId": "Un1q31d3nt1f13r"
    "successful": true
    "authSuccessful": true
    "advice": { "reconnect": "retry"
    "ext": { "json-comment-filtered": true
```

8. Multi frame operation

Current HTTP client implementations are RECOMMENDED to allow only two connections between a client and a server. This presents a problem when multiple instances of the Bayeux client are operating in multiple tabs or windows of the same browser instance. The two connection limit can be consumed by outstanding connect meta messages from each tab or window and thus prevent other messages from being delivered in a timely fashion.

8.1 Server Multi frame detection

It is RECOMMENDED that Bayeux server implementations use the cookie "Bayeux_HTTP_ID" to identify a HTTP client and to thus detect multiple Bayeux clients running within the same HTTP client. Once detected, the server SHOULD not wait for messages in connect and SHOULD use the advice interval mechanism to establish traditional polling.

8.2 Client Multi frame handling

It is RECOMMENDED that Bayeux client implementations use client side persistence or cookies to detect multiple intances of Bayeux clients running within the same HTTP client. Once detected, the user MAY be offered the option to disconnect all but one of the clients. It MAY be possible for client implementations to use client side persistence to share a Bayeux client instance.

9. Request / Response operation with service channels

The publish/subscribe paradigm that is directly supported by the Bayeux protocol is difficult to use to efficiently implement the request/response paradigm between a client and a server. The /service/** channel space has been designated as a special channel space to allow efficient transport of application request and responses over Bayeux channels. Messages published to service channels are not distributed to other Bayeux clients so these channels can be used for private requests between a Bayeux client and a server side handlers.

A trivial example would be an echo service, that sent any message received from a client back to that client unaltered. Bayeux clients would subscribe the the /service/echo channel, but the Bayeux server would not need to record this subscription. When a client publishes a message to the /service/echo channel, it will be delivered only to server-side subscribers (in an implementation depedent fashion). The server side handler for the echo service would handle each message received by publishing a response directly to the client regardless of any subscription. As the client has subscribed to /service/echo, the response message will be routed correctly within the client to the appropriate application handler.