

Internet Draft
expires in six months

Bill Janssen
Xerox PARC
1 August 1998

w3ng: Binary Wire Protocol for HTTP-ng

[<draft-janssen-httpng-wire-00.txt>](#)

Status of this Document

This document is an Internet-Draft. Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

To view the entire list of current Internet-Drafts, please check the "1id-abstracts.txt" listing contained in the Internet-Drafts Shadow Directories on ftp.is.co.za (Africa), ftp.nordu.net (Northern Europe), ftp.nis.garr.it (Southern Europe), munnari.oz.au (Pacific Rim), ftp.ietf.org (US East Coast), or ftp.isi.edu (US West Coast).

This document has been produced as part of the W3C HTTP-ng Activity (for current status, see "<http://www.w3.org/Protocols/HTTP-NG/Activity>"). This is work in progress and does not imply endorsement by, or the consensus of, either W3C or members of the HTTP-ng Protocol Design Working Group. We expect the document to evolve considerably as the project continues.

Distribution of this document is unlimited. Please send comments to the HTTP-NG mailing list at [<www-http-ng-comments@w3.org>](mailto:www-http-ng-comments@w3.org). Discussions are archived at "<http://www.w3.org/Protocols/HTTP-NG/>".

Please read the "HTTP-NG Short- and Longterm Goals" [[HTTP-ng-goals](#)] for a discussion of goals and requirements of a potential new generation of the HTTP protocol and how we intend to evaluate these goals.

Abstract

This document describes a binary 'on-the-wire' protocol to be used when sending HTTP-NG operation invocations or terminations across a network connection.

Table of Contents

1. Terminology and Syntax
2. Model of Operation
3. Global Issues

- 4. Utility Types
- 5. Messages
 - 5.1. Extension Headers
 - 5.2. Request Message
 - 5.2.1. Operation and Object Memoizing
 - 5.3. Reply Message
 - 5.4. InitializeConnection Message
 - 5.5. TerminateConnection Message
 - 5.6. DefaultCharset Message
- 6. Data Marshalling
- 7. Connection Exceptions
- 8. Security Considerations
- 9. References
- 10. Address of Author

1. Terminology and Syntax

Two data description languages are used in this document. The first is a pseudo-C syntax. It should be interpreted as C data structure layouts without any automatic padding to size boundaries, and allowing arbitrary bit-size limits on structs and unions as well as on ints and enums. The second is the data structure definition language defined in the XDR specification [[RFC 1832](#)]. Each use of pseudo-C and XDR is marked as to which language is being used.

This document uses a number of terms which are defined in the HTTP-ng Architecture Model document [HTTP-ng-arch]. It also references the type system defined in that document.

2. Model of Operation

This protocol assumes a particular model of operation based on conventional request/response messaging technology, with certain variations, as described in the HTTP-ng Architecture Model [HTTP-ng-arch]. The basic idea is that clients make use of services exported from a server by invoking operations on objects resident in that server. This model also assumes only hop-by-hop operation; proxying is supported at the application level.

The model used here assumes that operations are grouped into sets, the elements of which have a well-defined ordering; each operation set is called an "object type". It further assumes that an object type is identified by a UUID in the form of a URI; and that each operation in an object type can be identified with the ordinal number of the operation within the ordering of the elements of the object type. It assumes that every object has an "object ID", which also forms a unique identifier. It provides for the fact that instances of object types are grouped into "object groups"; an object group may contain any number of instances, even only a single instance. Object IDs always consist of a unique identifier for the object group of the instance, which we call the "group ID", along with a group-relative identifier we call the "instance handle". Note that grouping of instances is not required (i.e., every instance might define its own object group), but the protocol provides an efficiency optimization for the case where it is true.

The client is connected to the server by a "connection", which carries operation invocation requests from the client (known as the "caller") to the server (known as the "callee"), and operation results from the callee back to the caller. The connection has state associated with it, which allows the caller and callee to use shorthand notations for some of the data passed to the other party. Connections, in this model, are to a particular object group on the server; multiple connections can exist simultaneously between the same client and server, to different object groups; multiple connections to the same object group are also allowed. This protocol does not currently allow multiplexing of a single connection across multiple object groups; the HTTP-ng webmux transport layer is assumed to fulfill that function [HTTP-ng-webmux].

Two fundamental messages are defined by this protocol: the Request, which is used by the caller to invoke an operation on the callee, and the Reply, which is used to transfer the results of an operation from the callee to the caller. Every Reply message is associated with a particular Request message, but not every Request message has a Reply message associated with it. Connections are directional; operation invocation Requests always flow from the caller to the callee; Replies always flow from the callee to the caller. In addition to these messages, several control messages are defined for this protocol. These control messages are used to improve the efficiency and robustness of the connection. They are intended to be generated and consumed by the implementation of the wire protocol, and should have no direct effect on the applications using the protocol.

A Request message indicates two important elements, the "operation" and the "discriminant object", or discriminant; it also contains data values which are the input parameters to the operation. Each Request has an implicit connection-specific serial number associated with it; serial numbers begin with the value one (1), and have a maximum value of 16777215. When the maximum serial number of a connection has been reached, the connection must be terminated, and further operations must be invoked over a new connection.

A Reply message indicates the termination status of the operation, provides information about synchronization, and may contain data values which are output parameters or 'return values' from the operation. It contains an explicit serial number to indicate which Request it is a reply to. Replies may either indicate successful completion of the operation, or several different kinds of exceptional termination; if an exception is signalled, additional information is passed to indicate which of the possible exceptions for the operation was raised.

The model assumes that the messages are carried back and forth between the two parties by a "transport" subsystem. It requires that the transport subsystem be "reliable", "sequenced", and "message-oriented". By reliable, we mean that after a message is handed to the transport, the transport will either deliver it to the other party, or will signal an error if its reliable delivery cannot be ascertained. By sequenced, we mean that the transport will deliver messages to the other party in the same order in which the sender handed them to the transport. By message-oriented, we mean that the transport will provide indication of the beginning and ending of the

messages, without reference to any data encoded inside the message. An example of this type of transport would be the record marking defined in ONC RPC [[RFC 1831](#)] used with TCP/IP, or the HTTP-ng webmux transport layer [HTTP-ng-webmux] used with TCP/IP.

[3. Global Issues](#)

[3.1. Byte Order](#)

=====

All values use 'network standard' byte order, i.e. big-endian, because all Internet protocols use it. If in the future this becomes a problem for the Internet, this protocol will be affected by whatever solution is used to solve the problem in the wider Internet context. Note that the data marshalling format defined in XDR, which this protocol incorporates by reference, is also defined to be a big-endian protocol.

[3.2. Alignment and Padding](#)

=====

The marshalled form of each value begins on a 32-bit boundary. The marshalled form of each value is padded-after, if necessary, to the next 32-bit boundary. The padding bits may be either 0 or 1 in any combination.

[3.3. Marshalling Format](#)

=====

Marshalling is via the XDR format specified in the XDR specification [[RFC 1832](#)]. It could be argued that this format is inexcusably wasteful with certain value types, such as boolean (32 bits) or byte (32 bits), and that a 16-bit or 8-bit oriented format should be designed and used in its place. However, the argument of using an existing Internet standards-track marshalling format for this purpose, rather than inventing a new one, is a strong one; a new format should only be defined if measurement of the overhead shows gross waste, or if progress of the XDR specification slows unacceptably. Additionally, the simplicity of the XDR specification should allow correct implementations of it to be realized with a minimum of effort.

[3.4. Session Context](#)

=====

Unlike some previous protocols, this protocol is "session-oriented". That means that individual messages are sent in the context of a session, and are context-sensitive. This context-sensitivity allows session-wide compression. However, to support various kinds of marshalling architectures in implementations of this system, all marshalling can be done in a context-insensitive fashion, at the expense of sending additional bytes across the wire. However, unmarshalling implementations must always be capable of tracking and using context-sensitive information.

[4. Utility types](#)

The following data structures are defined in pseudo-C:

```
typedef enum {
    False = 0,
    True = 1
} Boolean;

typedef enum {
    InitializeConnection = 0,
    TerminateConnection = 1,
    DefaultCharset = 2
} ControlMsgType;

typedef enum {
    Success = 0,
    UserException = 1,           /* occurred during operation */
    SystemExceptionBefore = 2,  /* occurred before beginning operation */
    SystemExceptionAfter = 3    /* occurred after beginning operation */
} ReplyStatus;

typedef struct {
    Boolean cached_disc : 1;      /* True if cached object key */
    union {
        struct {
            Boolean cache_key : 1; /* True if both sides cache it */
            unsigned key_len : 13; /* length of key bytes */
        } uncached_key;
        unsigned cache_index : 14; /* cache index if cached */
    } v;
} DiscriminantID;

typedef struct {
    Boolean cached_op : 1;        /* True if cached id */
    union {
        struct {
            Boolean cache_operation : 1; /* True if should be cached */
            unsigned method_id : 13;     /* method index */
        } uncached_op_info;
        unsigned cache_index : 14;      /* cache index if "cached_op" set */
    } v;
} OperationID;

typedef enum {
    MangledMessage = 0,          /* bad protocol synchronization */
    ProcessFinished = 1,         /* sending party has `exitted' */
    ResourceManagement = 2,      /* transient close */
    WrongCallee = 3,            /* bad object group ID received */
    MaxSerialNumber = 4          /* the maximum serial number was used */
} TerminationCause;

typedef struct {
    unsigned major : 4;
    unsigned minor : 4;
} ProtocolVersion;

typedef unsigned Unused;
```

5. Messages

Only a few messages are defined. The ``InitializeConnection'` message is used by the caller to verify that it has connected to the right server, and that it is using the correct version of the wire protocol. The ``DefaultCharset'` message allows both sides to independently define a default value for string charsets. The ``Request'` message causes an operation to be started on the remote server. The ``Reply'` message is sent from the server to the client to inform it of the completion status of the operation, and to convey any result values. The ``TerminateConnection'` message allows either side to indicate graceful shutdown of a connection.

5.1. Extension Headers

=====

This protocol uses a mechanism called an "extension header" to provide for extensibility and tailorability. Features such as transaction contexts or global thread identifiers may be implemented via this mechanism. An extension header is name-value pair, where the name is a UUID expressed as a URI, and the value is an HTTP-ng pickle (see [HTTP-ng-arch]) value. This name-value pair is then expressed as an XDR value of the type ``ExtensionHeader'` as described below. Each request message and reply message may contain a sequence of extension headers, expressed as a value of the XDR type ``ExtensionHeaderList'`.

```
/* XDR */
struct {
    string      name<0xFFFF>; /* URI for extension header */
    opaque      value<>;      /* Pickle containing value of header */
} ExtensionHeader;
typedef ExtensionHeader ExtensionHeaderList<>;
```

5.2. ``Request'` Message

=====

Request header (pseudo-C):

```
typedef struct {
    Boolean control_msg : 1; /* == FALSE */
    Boolean ext_hdr_present : 1; /* True if ext hdr list present */
    OperationID operation_id : 15; /* identifies operation */
    DiscriminantID object_key : 15; /* identifies discriminant */
} RequestMsgHeader /* 4 bytes total */
```

The actual message consists of the following sections:

```
[ `RequestMsgHeader' ]
[ extension header list, if any ]
[ XDR `string' containing object type ID of object type defining
operation, if not cached ]
[ bytes of OBJECT_KEY, if not cached, padded to 4 byte boundary ]
[ explicit input parameter values, if any, padded to a 4 byte boundary ]
```

The OPERATION_ID contains either a connection-specific 14-bit cache index, or a 13-bit method id (the zero-based ordinal position of the method in the definition of the object type in which the operation is

defined) of the operation. If the method id is given, an additional value, an XDR `string' value containing the object type ID of the object type in which the operation is defined, is also passed. This means that this protocol will not support interfaces in which object types have more than 8192 methods directly defined.

The OBJECT_KEY is either a 14-bit connection-specific cache index, or the length of a variable length octet sequence of 8192 or fewer bytes containing the service-point-relative name for the object (the INSTANCE-HANDLE of the URL). The object key value of `{ False, False, 0 }', normally a zero byte variable length object key, is reserved for use by the protocol. The OBJECT_KEY is marshalled onto the transport as an XDR value of type `fixed-length opaque data', where the length is that specified in the `v.key_len' field of the OBJECT_KEY.

5.2.1 Operation and Object Memoizing

Callers may reduce the size of messages by memoizing operation IDs and object IDs that are passed in the connection. This is done by the caller setting the `cache_key' (for object IDs) or `cache_operation' (for operation IDs) bit in the `DiscriminantID' or `OperationID' struct when the object key or operation ID is first sent. Each side must then assign the next available index to that object or operation. The space of operations is separate from the space of object ids, so that a total of 16383 possible values is available for memoizing of discriminant objects, and 16383 different possible values for memoizing of operations.

Note that the index is passed implicitly, so both sides of the connection must synchronize their use of indices.

A shared set of indices may be loaded into the connection by some mechanism before any messages are sent. This specification does not define a mechanism for doing so.

5.3. `Reply' Message

=====

Reply header (pseudo-C):

```
typedef struct {
    Boolean control_msg : 1;          /* == FALSE */
    Boolean ext_hdr_present : 1;      /* True if ext hdr list present */
    ReplyStatus : 2;
    Unused_reply_1 : 4;
    unsigned serial_no : 24;          /* serial # from Request */
} ReplyMsgHeader;                   /* 4 bytes total */
```

The actual message consists of the following fields:

```
[ `ReplyMsgHeader' ]
[ extension header list, if any ]
[ exception ID (32-bit unsigned), if any ]
[ explicit output parameter values, if any, padded to 4 byte boundary ]
```

5.4. `InitializeConnection' Message

=====

InitializeConnection header (pseudo-C):

```
typedef struct {
    Boolean control_msg : 1;          /* == TRUE */
    ControlMsgType msg_type : 3;      /* == InitializeConnection */
    Unused verify_1 : 4;
    ProtocolVersion version : 8;      /* what version of the protocol? */
    unsigned objgroup_id_len : 16;    /* length of object group ID */
} InitializeConnectionMsgHeader;
```

The actual message consists of the following fields:

[`InitializeConnectionMsgHeader']

[`objgroup_id_len'-length object group ID for supposed callee, padded to 4-byte boundary]

This message is sent from caller to callee as the first message of the connection. It is used to pass the object group ID of the connection from client to server, so that both sides understand what the omitted prefix portion of discriminant IDs is. If the object group ID received by the callee is not the correct object group ID for the callee (i.e., the callee has objects which do not have that prefix in their object IDs), the callee should terminate the connection, with the appropriate reason. The object group ID is passed as an XDR `fixed-length opaque data' value of the length specified in `objgroup_id_len'.

5.5. `TerminateConnection' Message

=====

TerminateConnection header (pseudo-C):

```
typedef struct {
    Boolean control_msg : 1;          /* == TRUE */
    ControlMsgType msg_type : 3;      /* == TerminateConnection */
    TerminationCause cause : 4;       /* why connection terminated */
    unsigned serial_no : 24;          /* last request processed/sent */
} TerminateConnectionMsgHeader;
```

The actual message consists simply of the header; it provides for graceful connection shutdown. It is sent either from the caller to the callee, or from the callee to the caller, and informs the other party that it is cancelling the connection, for one of these reasons:

1. A badly formatted message has arrived from the other party, and protocol synchronization is believed lost, or, the caller has sent a `InitializeConnection' message with the wrong major version for the protocol;
2. This party (process, thread, whatever) is going away, and the other party should not attempt to reconnect to it;
3. This connection is being terminated due to active resource management; the other party should attempt to reconnect if it needs to - this reason is typically only useful from callee to caller;
4. The caller has sent a `InitializeConnection' message with the

wrong object group ID;

5. The caller has used the maximum serial number available for this connection.

The ``serial_no'` field contains the serial number of the last message completely processed by the caller (when ``TerminateConnection'` is sent from caller to callee), or the serial number of the last message sent by the callee (when sent from callee to caller). No further messages should be sent on the connection by a sender of a ``TerminateConnection'` message after it has been sent, or by a receiver of ``TerminateConnection'` message after it has been received.

5.6. ``DefaultCharset'` Message

DefaultCharset header (pseudo-C):

```
typedef struct {
    Boolean control_msg : 1;      /* == TRUE */
    ControlMsgType msg_type : 3;  /* == DefaultCharset */
    Unused bits_12: 12;          /* unused */
    unsigned charset_mibenum : 16; /* default charset */
} DefaultCharsetMsgHeader;
```

This message is sent by either side of a connection to establish a default charset for subsequent messages sent by that side of the connection. The charset defines how string values are marshalled as octet sequences. The default charset defines the default marshalling, unless overridden by an explicit charset in a string value. Each side of the connection may establish a default charset independently of the other side of the connection; the default charset only applies to string values in messages coming from that side. A new value of the default charset may be established at any time by sending another ``DefaultCharset'` message.

6. Data Marshalling

This section defines how values of the type specified in the HTTP-ng type system [HTTP-ng-arch] are marshalled into a Request or Reply message.

The data value format used for parameters is the XDR format specified in [RFC 1832]. However, we extend the XDR specification with one additional type, called "flagged variable-length opaque data". It is similar to XDR's regular variable-length opaque data, except that the high-order bit of the length field is used as a flag bit, instead of being part of the length. This means that flagged variable-length opaque data can only carry opaque data of lengths less than or equal to $(2^{*31})-1$.

	0	1	2	3	4	5	...		
flag	length n			byte0 byte1 ...		n-1	0	...	0
bit	31 bits			n bytes		r bytes		(where (n+r) mod 4 = 0)	

6.1. Boolean Type

=====

Values of type ``BOOLEAN'` are passed as XDR ``bool'`.

6.2. Enumeration Types

=====

Values of enumeration types are passed as XDR ``enum'`. Each enumeration value is assigned its ordinal value as it appears in the declaration of the enumeration type, starting with the value ``one'`.

6.3. Numeric Types

=====

7.3.1. Fixed-point Types

Values of fixed-point types are passed by passing the value of the numerator. We define a number of special cases for efficient marshalling of common integer types, as well as a general case for passing values of fixed-point types that are not covered by the special cases.

Special cases:

- * 32-bit integer: Fixed-point values with a minimum-numerator value greater than or equal to -2147483648 and with a minimum numerator value less than or equal to 2147483647 are passed as XDR ``integer'`.
- * 32-bit unsigned integer: Fixed-point values with a minimum-numerator value greater than or equal to 0 and with a maximum numerator less than or equal to 4294967295 are passed as XDR ``unsigned integer'`.
- * 64-bit integer: Fixed-point values with a minimum numerator value greater than or equal to -9223372036854775808 and with a maximum numerator less than or equal to 9223372036854775807 are passed as XDR ``hyper integer'`.
- * 64-bit unsigned integer: Fixed-point values with a minimum-numerator value greater than or equal to 0 and with a maximum numerator value less than or equal to 18446744073709551615 are passed as XDR ``unsigned hyper integer'`.

General case:

The numerator of the value is passed as XDR ``flagged variable-length opaque data'`, with the bytes of the data containing the value expressed as a base-256 number, in big-endian order; that is, with the most significant digit of the value first. The flag bit is used to carry the sign; the flag bit is 0 for a positive number or zero, and 1 for a negative number.

7.3.2. Floating-point Types

We define a number of special cases for efficient marshalling of common floating-point types, as well as a general case for passing values of floating-point types that are not covered by the special cases.

Special cases:

- * IEEE single: floating point types matching the IEEE 32-bit floating-point format (that is, with the parameters significand-size=24, exponent-base=2, maximum-exponent-value=127, minimum-exponent-value=-126, has-Not-A-Number=TRUE, has-Infinity=TRUE, denormalized-value-allowed=TRUE, and has-signed-zero=TRUE) are passed as XDR `floating-point'.
- * IEEE double: floating point types matching the IEEE 64-bit floating-point format (that is, with the parameters significand-size=53, exponent-base=2, maximum-exponent-value=1023, minimum-exponent-value=-1022, has-Not-A-Number=TRUE, has-Infinity=TRUE, denormalized-value-allowed=TRUE, and has-signed-zero=TRUE) are passed as XDR `double-precision floating-point'.
- * Intel extended double: floating point types matching the Intel IEEE floating-point-compliant extended double floating-point format (that is, with the parameters significand-size=64, exponent-base=2, maximum-exponent-value=16383, minimum-exponent-value=-16382, has-Not-A-Number=TRUE, has-Infinity=TRUE, denormalized-value-allowed=TRUE, and has-signed-zero=TRUE), are passed as a 12-byte value of XDR `fixed-length opaque data', containing the floating-point value in the format specified in the UNIX System V Application Binary Interface Intel 386 Processor Supplement (Intel ABI) document: the 63 bits of the fraction occupy the first 7 bytes in little-endian order plus the low seven bits of the eighth byte; the 1 bit explicit leading significand bit occupies the high-order bit of the eighth byte; the 15 bits of the exponent occupy the ninth byte and the low-order bits of the tenth byte, in little-endian order; the sign bit occupies the high-order bit of the tenth byte; the eleventh and twelfth bytes are unused, and should contain zero values.
- * SPARC & PowerPC extended double: floating point types matching the XDR quadruple-precision floating-point format (that is, with the parameters significand-size=113, exponent-base=2, maximum-exponent-value=16383, minimum-exponent-value=-16382, has-Not-A-Number=TRUE, has-Infinity=TRUE, denormalized-value-allowed=TRUE, and has-signed-zero=TRUE), which is the form of extended double floating-point used by PowerPC and SPARC processors, are passed as XDR `quadruple-precision floating-point'.

General case:

Values of floating-point types not matching the special cases identified above are passed as a value of the XDR struct type `GeneralFloatingPointValue', which has the following definition:

```
/* XDR */
```

```
enum { Normal = 1, NotANumber = 2, Infinity = 3 } FloatingPointValueType;
struct {
    flagged opaque FixedPointSignAndSignificand<>;
    flagged opaque FixedPointExponent<>;
} NormalFloatingPointValue;
union switch (FloatingPointValueType disc) {
    case Normal: NormalFloatingPointValue value;
    case NotANumber: void;
    case Infinity: void;
} GeneralFloatingPointValue;
```

The two fields of the 'NormalFloatingPointValue' struct each contain an on-the-wire representation of a fixed-point value of the fixed-point type (denominator=1, no-minimum-numerator, no-maximum-numerator). The 'FixedPointSignAndSignificand' field contains the sign of the floating-point value as the sign, and the actual significand as the absolute value of the fixed-point value. The 'FixedPointExponent' field contains the exponent of the floating-point value.

6.4. String Types

Each string value sent in this protocol has a "charset" [[RFC 2278](#)] associated with it, identified by the charset's IANA-assigned MIBEnum value. Each side of a session may establish a "default charset" by sending the 'DefaultCharset' message. String values that use the default character set do not contain explicit charset information; string values that use a charset other than the default charset contain the MIBEnum value for the charset, along with the bytes of the string.

We send a string value as a value of XDR 'flagged variable-length opaque data'. If the flag bit is 1, the first two bytes of the string value are the MIBEnum of the charset, high-order byte first; the remaining bytes are the bytes of the string. If the flag bit is 0, the bytes of opaque data simply contain the bytes of the string; the charset is the default charset for the session. It is a marshalling error to send a string value with a flag bit of 0 over a session for which no default charset has been established. To avoid context-sensitivity in marshalling a string, it is always valid to marshal a string with an explicit charset value, even if the charset value is the same as the default charset for the session. When marshalling a string into a pickle, the charset should always be explicitly included.

6.5. Sequence Types

Values of sequence types are passed as XDR 'variable-length arrays', with one exception: Sequences of any fixed-point type with a minimum numerator greater than or equal to 0, and a maximum numerator less than or equal to 255, are passed as XDR 'variable-length opaque data', with one numerator value per octet.

6.6. Array Types

Values of array types are passed as XDR 'fixed-length arrays', with

one exception: Arrays of any fixed-point type with a minimum numerator greater than or equal to 0, and a maximum numerator less than or equal to 255, are passed as XDR 'fixed-length opaque data', with one numerator value per octet. Values of array types are passed as XDR 'fixed-length arrays', with one exception:

6.7. Record Types

=====

Values of record types are passed as XDR 'struct'.

6.8. Union Types

=====

Values of union types are passed as XDR 'union', with the union discriminant being the zero-based ordinal value for the encapsulated value's type.

6.9. Pickle Type

=====

A pickle is passed as an XDR 'variable-length opaque data', containing the type ID of the pickled value's type, followed by the XDR-marshalled pickled value. To save pickle space for common value types used in metadata, we define a packed format for the type ID marshalling. A type ID is marshalled into a pickle as a 32-bit header, in an XDR 'unsigned integer', possibly followed by an XDR 'fixed-length opaque data', containing the string form of the type ID of the pickled type. The header has the following internal structure:

```
/* Pseudo-C */
typedef struct {
    unsigned          version : 8;
    PickleTypeKind    type_kind : 8;
    unsigned          type_id_len : 16;
} TypeIDHeader;
```

The 'version' field gives the version number of the pickle format; the 'type_kind' field contains a value from the enum

```
/* Pseudo-C */
typedef enum {
    TypeKind_unconstrained = 0, /* anything not covered by other type kind */
    TypeKind_boolean = 1, /* BOOLEAN */
    TypeKind_s8 = 2, /* FIXED-POINT DENOM=1 MIN-NUM=-128 MAX-NUM=127 */
    TypeKind_s16 = 3, /* FIXED-POINT DENOM=1 MIN-NUM=-32768 MAX-NUM=32768 */
    TypeKind_s32 = 4, /* FIXED-POINT DENOM=1 MIN-NUM=-2147483648 MAX-NUM=2147483648 */
    TypeKind_s64 = 5, /* FIXED-POINT DENOM=1 MIN-NUM=-9223372036854775808 MAX-NUM=9223372036854775807 */
    TypeKind_u8 = 6, /* FIXED-POINT DENOM=1 MIN-NUM=0 MAX-NUM=255 */
    TypeKind_u16 = 7, /* FIXED-POINT DENOM=1 MIN-NUM=0 MAX-NUM=65535 */
    TypeKind_u32 = 8, /* FIXED-POINT DENOM=1 MIN-NUM=0 MAX-NUM=4294967295 */
    TypeKind_u64 = 9, /* FIXED-POINT DENOM=1 MIN-NUM=0 MAX-NUM=18446744073709551615 */
    TypeKind_ieee_float32 = 10, /* FLOATING-POINT SIGNIFICAND-SIZE=24 EXPONENT-SIZE=8
                                MAXIMUM-EXPONENT-VALUE=127 MINIMUM-EXPONENT-VALUE=-126
                                HAS-NOT-A-NUMBER=TRUE HAS-INFINITY=TRUE
                                DENORMALIZED-VALUE-ALLOWED=TRUE HAS-SIG-NAL=TRUE */
    TypeKind_ieee_float64 = 11, /* FLOATING-POINT SIGNIFICAND-SIZE=53 EXPONENT-SIZE=11
                                MAXIMUM-EXPONENT-VALUE=1023 MINIMUM-EXPONENT-VALUE=-1022
                                HAS-NOT-A-NUMBER=TRUE HAS-INFINITY=TRUE
                                DENORMALIZED-VALUE-ALLOWED=TRUE HAS-SIGNAL=TRUE */
}
```

```

HAS-NOT-A-NUMBER=TRUE HAS-INFINITY=TRUE
DENORMALIZED-VALUE-ALLOWED=TRUE HAS-SIG
TypeKind_i_default_str = 12, /* STRING LANGUAGE="i-default" */
TypeKind_object = 13,      /* local or remote object */
...
/* other types like Date, etc, should be added here... */
...
} PickleTypeKind;

```

If the value of ``type_kind'` is ``TypeKind_unconstrained'`, the value of ``type_kind_len'` is the length of a value of XDR type ``fixed-length opaque data'`, containing the full string type ID of the type, which immediately follows the header. Otherwise, no ``opaque data'` is marshalled.

For the purposes of marshalling, pickles have no default charset; this means that strings marshalled into a pickle should always contain an explicit charset. Pickles should be considered a single "message" for the purposes of marshalling aliased reference types.

6.10. Reference Types

6.10.1. Optional Types

Optional types are passed as XDR ``optional-data'`.

6.10.2. Aliased Types

The scope of aliasing in this protocol is the message, as in Java RMI, rather than the call, as in DCE RPC. That is, aliasing occurs only within the context of a single invocation or result, rather than across a full invocation-result pair. For the purposes of marshalling, a pickle scope should be considered a single message scope.

Each unique value of an aliased type that is marshalled is assigned a 32-bit unsigned integer value, unique in the scope of aliasing, called its "aliased identifier". This identifier is marshalled as an XDR ``unsigned integer'`. If the aliased value has not previously been sent in this scope, its value is then marshalled as a value of its base type would be. Note that this means that the full value of every aliased type is sent only once in a scope; subsequent occurrences send only the aliased identifier.

[XXX - how to handle overflow of aliased value cache?]

6.11. Object Types

An instance of an object type is passed as the state of the object type, which also contains information about the actual type of the value. For remote object types, this state is followed by the object identifier, and optionally information about how the instance may be contacted.

6.11.1. Parameter Type Versus Actual Type

When marshalling the state of an object, it's important to distinguish two important types of the value: the "parameter type", which is the type that both sides of the session expect the value to have, and the "actual type" of the value, which is the most-derived type of the object, and may be a subtype of the parameter type. If the actual type is different from the parameter type, extra information must be passed along with the value to allow the receiver to properly distinguish the type and its associated data. However, if the actual type is the same as the parameter type, some of this information can be omitted.

6.11.2. Passing the Actual Type Information

We try to pass the type information of the object type as the type ID of the most-derived-type of the object. However, for proper unmarshalling of local object types, we also need to pass additional type IDs. Type information is thus passed according to the following rules:

1. If the parameter type of the object is sealed, both sides already know the most-derived-type ID of the instance, and know that the actual type must be the same as the parameter type. In this case, the type ID is passed as XDR `void'.
2. Otherwise, type information is passed as a value of the following XDR union type `GeneralTypeInfo':

```
/* XDR */
enum { FormalType = 1, RemoteMSTID = 2, LocalTypeTree = 3 }
ObjectTypeInformation;
union switch (ObjectTypeInformation disc) {
  case FormalType: void; /* passed implicitly */
  case RemoteMSTID: opaque<0xFFFF>; /* contains mdt type ID */
  case LocalTypeTree: VALUE OF HTTP-NG.TYPEIDTREENODEREF;
                                /* full inheritance tree */
} GeneralTypeInfo;
```

That is:

1. If the actual type of the object is the same as the parameter type, again both sides know it, and the type information is passed implicitly.
2. If the object type inherits from `HTTP-ng.RemoteObjectBase', the type ID of the most-derived type of the object is passed as a value of XDR `variable-length opaque data' containing the type ID.
3. If the object type does not inherit from `HTTP-ng.RemoteObjectBase', and is thus a local object type, the full type inheritance hierarchy of the type is passed as a value of `HTTP-ng.TypeIDTreeNodeRef'.

6.11.3. Passing the State Attributes

The state attributes are marshalled in one of two ways:

1. If the actual type of the instance is the same as the parameter type, the state of each of the types of the object are passed by walking the supertype inheritance tree of the instance in a depth-first order, passing the value of each attribute of any particular state in the order in which they are defined, as if each state formed an XDR 'structure' with the attributes as the components of the structure. The value of each attribute is marshalled directly according to the type of the attribute.
2. If the actual type of the instance is a subtype of the parameter type, the receiver has to be able to handle state for types it has no knowledge of. To allow for this, the state of each type is passed as an encapsulation. That is, the state of the instance is passed as a sequence of XDR 'structure' values, each containing the state for one of the types of the instance. Types of the instance which have no associated state do not appear in this sequence. An XDR expression of the sequence would be the following:

```
/* XDR */
struct {
    opaque type_id<0xFFFF>;
    opaque state<>;
} TypeState;
typedef TypeState StateSequence<>;
```

The type_id field contains the type ID for that type of the the object value. The variable-length opaque data field state contains the values of the attributes of the state marshalled as an XDR 'structure', where the components of the structure are the attributes of the state.

6.11.4. Passing the Object ID and Contact Info

In the case of a remote object type, the object group ID, instance handle and contact info for the value are passed as a value of the following XDR structure type 'RemoteObjectInfo':

```
/* XDR */
typedef string ContactInfo<0xFFFF>;
struct {
    opaque objgroup_id<>;
    opaque instance_handle<>;
    ContactInfo cinfos<>;
} RemoteObjectInfo;
```

where objgroup_id is a identifier for the server which supports the desired object, and instance_handle is a server-relative name for the object. The cinfos field contains zero or more pieces of information about the way in which the object needs to be contacted, including information such as whether various transport layers are involved.

6.11.5. Syntax of Cinfo Strings

[Note: this cinfo syntax is poorly defined. An extensible more conventionally URI-based scheme should replace this at some point.]

Each cinfo string has the form described below (where brackets indicate optionality, an <ALPHANUMERIC-ID> is an identifier composed of ASCII lowercase alphabetic and numeric characters, beginning with a lowercase alphabetic character, and a <NON-UNDERSCORE-STRING> is any string of ASCII characters not containing the underscore character '_'):

```
<cinfo> := <pinfo> '@' <tinfo-stack>

<pinfo> := <scheme> [ '_' <parms> ]

<scheme> := <ALPHANUMERIC-ID>

<parms> := <parm> [ '_' <parms> ]

<parm> := <NON-UNDERSCORE-STRING>

<tinfo-stack> := <tinfo> [ '=' <tinfo-stack> ]

<tinfo> := <scheme> [ '_' <parms> ]
```

* 6.11.5.1. Syntax of `w3ng' Pinfo

The current syntax of the pinfo string for the `w3ng' wire protocol is

```
<scheme> := 'w3ng'

<parms> := <major-version> [ '.' <minor-version> ]
```

where `<major-version>' and `<minor-version>' are numbers between 0 and 15. If the `<minor-version>' is not specified, it defaults to 0.

* 6.11.5.2. Syntax of `w3mux' Tinfo

The current syntax of the tinfo string for the `w3mux' transport layer is

```
<scheme> := 'w3mux'

<parms> := <channel> '_' <endpoint>
```

where `<channel>' is a protocol ID number [MUX], and `<endpoint>' is a UUID string for an endpoint. The size of the `<endpoint>' string must be less than 1000 bytes.

* 6.11.5.3. Syntax of `tcp' Tinfo

The current syntax of the tinfo string for the `tcp' transport layer is

```
<scheme> := 'tcp'

<parms> := <host> '_' <port>
```

where `<host>' is string of less than 1000 bytes indicating the IP address or hostname of the remote machine, and `<port>' is the TCP

port on which the host is listening.

7. Connection Exceptions

We define a number of exceptions, as defined in [HTTP-ng-arch], which may be signalled 'across the wire' (between compatibility domains) as a result of any operation invocation, and are called "connection exceptions". Each connection exception has a specified "exception code", and may also have output parameters associated with it. This set of exceptions may not be the same as the set of system exceptions built into any implementation of the HTTP-ng architecture; the implementation is responsible for mapping from its internal set of exceptions to that supported by the wire protocol.

7.1. 'UnknownProblem'

=====

Exception code: 0

Output parameters: None

An unknown problem occurred.

7.2. 'ImplementationLimit'

=====

Exception code: 1

Output parameters: None

The request could not be properly addressed because of some implementation resource limit on the callee side.

7.3. 'SwitchConnectionCinfo'

=====

Exception code: 2

Output parameters: NEW-CINFO : value of a string type with language "i-default"

This exception requests the caller to upgrade the connection protocol and transport information to the cinfo specified as the argument, and re-try the call. This is the equivalent of the 'UPGRADE' message in HTTP 1.1, and the 'RELOCATE_REPLY' message in CORBA GIOP.

7.4. 'Marshal'

=====

Exception code: 3

Output parameters: None

A marshalling problem was encountered.

7.5. 'NoSuchObjectType'

=====

Exception code: 4

Output parameters: None

The object type of the operation was unknown at the server.

7.6. `NoSuchMethod'

=====

Exception code: 5

Output parameters: None

The object type of the operation was known at the server, but did not contain the indicated method.

7.7. `NoSuchObject'

=====

Exception code: 6

Output parameters: None

The specified discriminant object was not available at the server.

7.8. `InvalidType'

=====

Exception code: 7

Output parameters: None

The object specified by the discriminant did not participate in the type specified in the operation.

7.9. `Rejected'

=====

Exception code: 8

Output parameters: REASON : value of a string type with language "i-default"

The server refused to process the request. It may return a string giving a reason for the rejection.

7.10. `OperationOrDiscriminantCacheOverflow'

=====

Exception code: 9

Output parameters: None

The request caused the receiver's cache of operations or discriminants to overflow. The sender may retry the request with uncached operation and discriminant values; subsequent requests should not cache any additional operation or discriminant values, but may continue to use previously successfully cached values.

8. Security

This protocol assumes that security provisions are made either at some level above it, typically in the application interfaces, or at some level below it, typically by use of a secure transport mechanism.

It contains no protocol-level mechanisms for providing or assuring any of the concerns normally related to security.

9. References

[HTTP-ng-arch]: HTTP-ng Architecture Model. (See
`http://www.w3.org/TR/WD-HTTP-NG-architecture'.)

[HTTP-ng-goals]: HTTP-ng Short- and Long-term Goals. (See
`http://www.w3.org/TR/WD-http-ng-goals'.)

[HTTP-ng-webmux]: HTTP-ng WEBMUX Protocol Specification. (See
`http://www.w3.org/TR/WD-mux'.)

[[RFC 1831](#)]: [RFC 1831](#), RPC: Remote Procedure Call Protocol Specification Version 2; R. Srinivasan, August 1995. (See
`http://info.internet.isi.edu:80/in-notes/rfc/files/rfc1831.txt'.)

[[RFC 1832](#)]: [RFC 1832](#), XDR: External Data Representation Standard; R. Srinivasan, August 1995. (See
`http://info.internet.isi.edu:80/in-notes/rfc/files/rfc1832.txt'.)

[RFC 2277] [RFC 2277](#), IETF Policy on Character Sets and Languages; H. Alvestrand, January 1998. (See
`http://info.internet.isi.edu:80/in-notes/rfc/files/rfc2277.txt'.)

[RFC 2278] [RFC 2278](#), IANA Charset Registration Procedures; N. Freed & J. Postel, January 1998. (See
`http://info.internet.isi.edu:80/in-notes/rfc/files/rfc2278.txt'.)

10. Address of Author

Bill Janssen

Mail: Xerox Palo Alto Research Center
3333 Coyote Hill Rd
Palo Alto, CA 94304

Phone: (650) 812-4763

FAX: (650) 812-4777

Email: janssen@parc.xerox.com

HTTP: <http://www.parc.xerox.com/istl/members/janssen/>