Workgroup:	Internet Engineering Task Force
Internet-Draft:	draft-royer-phoenix-00
Published:	11 February 2025
Intended Status:	Informational
Expires:	15 August 2025
Author:	DM. Royer
	RiverExplorer LLC

Phoenix: Lemonade Risen Again

Abstract

NOTE: This is just getting started, not ready for submission yet.

Email and MIME messages account for one the largest volumes of data on the internet. The transfer of these MIME message has not had a major updated in decades. Part of the reason is that it is very important data and altering it takes a great deal of care and planning.

This application transport can also transfer non-MIME data. It can be used as an XDR transport, or for opaque data (blobs of known or unknown data) transport.

Another major concern is security and authentication. This proposal allows for existing authentication to continue to work.

This is a MIME message transport that can facilitate the transfer of any kind of MIME message. Including email, calendaring, and text, image, or multimedia MIME messages. It can transfer multipart and simple MIME messages.

The POP and IMAP protocols are overly chatty and now that the Internet can handle 8-bit transfers, there is no need for the overly complex text handling of messages.

This proposal includes a sample implementation. (Github - Phoenix) Which also includes a gateway from this proposal to existing systems. Thunderbird and Outlook plugins are part of the sample implementation. A Linux, Windows DLL and .NET, and Android client library are part of the sample implementation

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

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."

This Internet-Draft will expire on 15 August 2025.

Copyright Notice

Table of Contents

1. Requirements Language

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

2. Introduction

On the Internet, just about everything is a MIME object and there are many ways to transport MIME. This document specifies a new application level MIME transport mechanism and protocol. This document does not specify any new or changed MIME types.

Transporting MIME objects is generally done in one of two ways: (1) Broadcasting, (2) Polling. Both methods often require some form of authentication, registration, and selecting of the desired material. These selection processes are essentially a form of remote folder management. In some cases you can only select what is provided, and in others you have some or a lot of control over the remote folders.

In addition to other functions, this specification defines a remote and local folder management. This remote folder management is common with many type of very popular protocols. This design started by looking at the very popular IMAP and POP protocols.

An additional task is transporting the perhaps very large MIME objects. Some MIME objects are so large that some devices may default to looking at only at parts of the MIME object. An example is an email message with one or more very large attachments, where the device may default to not download the large attachment without a specific request from the user.

Some objects are transported as blocks of data with a known and fixed size. These are often transported with some kind of search, get, and put commands. In effect these are folder and file commands

Other MIME objects are transported in streams of data with an unspecified size, such as streaming music, audio, or video. This specification describes how to use existing protocols to facilitate the data streaming. And again, these are folder and file commands.

A MIME object can be a simple object, or it may contain many multipart sections of small to huge size. These sections can be viewed as files in the containing MIME object.

By implementing this specification application developers can use the techniques to manage local and remote files and folders. Remote email or files are the same thing in this specification. The sections of MIME object with multipart sections are viewed as files in the MIME object. You can interact with the entire folder, or just the files within it.

MIME objects have meta data, and they are called headers. Files and folders have meta data, and they are called file attributes. This specification does not mandate any meta data. It does define some that may be used by implementations. Other related specifications do define some meta data that is consistant with existing protocols. This protocol allows for a consistent transport of existing meta data and MIME objects.

File and folder meta data is a complex task that can involve access control lists and permissions. This specification defines a mechanism to transport this meta data, it does not define the meta data.

And this specification provides for the ability to define both protocol extensions and the creating of finer control for specific commands that may evolve over time.

This examples compares current folder and file manipulations to how it can be used in this protocol with email.

- You can search for file names. You can search email for: sender, subject, and more.

- You can search for file contents. You can search for email message contents.
- You can create, delete, and modify files. You can create, delete, and modify email messages.
- You can create, delete, and modify folders. You can create, delete, and modify email folders.

What this specification defines:

- How to use existing authentication implementations or use new ones.
- This specification describes a standard way to perform file and folder operations that are remote to the application and agnostic to purpose of data being transported.
- Specifies a way to migrate from some existing protocols to Phoenix. Provides links to sample implementations.

3. ABNF, Notes, and Definitions

3.1. XDR TYPE - Meaning - Informative

The meaning of "XDR TYPE" in this specification refers to the "C" code API. For every "XDR TYPE" "foo", is a "C" code API of "xdr_foo(...)". The purpose of "XDR TYPE" is to guide implementors and is to be considered informative and not normative information.

In some cases the "xdr_foo(...)" API is part of system libraries, and in other cases it is the result of processing the normative XDR definition files provided in this specification with the tools in the sample implementation or open-source XDR "rpcgen / rpcgen++" [rpcgenopensource] tools. "rpcgen documentation" [rpcgendocs].

In most cases the "XDR TYPE" is used in a ".x" XDR definition file as the variable type. In some cases like "string", and "opaque", they must be defined as arrays. Any other exceptions are added to the informative description in this specifications when needed:

NAME	Description:	XDR API
string	Is an XDR array. See [rpcgendocs]. The ABNF definition for opaque is in Figure 4 "string MyVariableName<>;"	xdr_string()
opaque	Is an XDR array. See [rpcgendocs]. The ABNF definition for opaque is in Figure 4 "opaque MyOpaqueData<>;"	xdr_opaque()

Table 1: XDR string and opaque are arrays.

3.2. ABNF Number of bits in value

This specification adds some syntax to ABNF [RFC5234] to deal with bit width in a binary number.

Terminals may specify a bit width. That is the number of bits in the value.

Section 2.3 of Terminal Values [RFC5234] is within this specification defined to be:

b = binary / binary:width

d = decimal / decimal:width

x = hexadecimal / hexadecimal:width

Where: with is the number of bits in the value. And must be an unsigned integer greater than zero.

When the left side has a width: The number of bits on the left side must equal the number of bits on the right side.

The most significant values are placed to the left of lesser signification values in the rule:

In this example A Header is 32-bits in size and is composed of an 8-bit (Offset), 2-bit (Flags or F), and 22-bit (Length) value.

```
Header:32 = Offset:8 Flags:2 Length:22
```

Figure 1

Example pseudo code for the ABNF in Figure 1 could be:

```
// Header is a 32-bit unsigned integer.
// Offset is an 8-bit unsigned integer.
// Flags (F) is a 2-bit unsigned integer.
// Length is a 22-bit unsigned integer
//
Header = (Offset << 24) | (Flags << 22) | Length;
```

Figure 2

The pseudo code in Figure 2 shifts the 8-bit "Offset" over 24 bits to the left, then shifts the 2-bit value "Flags (F)" over 22 bits, then, places the lower 24-bits "Length" into the results. The result would be all three values into the one 32-bit result as illustrated in Figure 3:

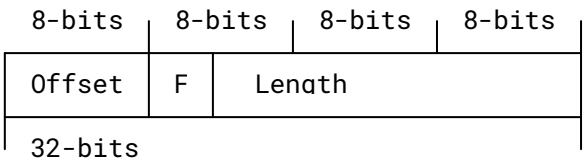


Figure 3: Packed Bit Example

3.3. Common Definitions

TYPE	Notes	XDR API
uint8_t	The associated "C" API function is xdr_uint8_t(). uint8_t is built into XDR.	xdr_uint8_t()
uint16_t	The associated "C" API function is xdr_uint16_t(). uint16_t is built into XDR.	xdr_uint16_t()
uint32_t	The associated "C" API function is xdr_uint32_t(). uint32_t is built into XDR.	xdr_uint32_t()
uint64_t	The associated "C" API function is xdr_uint64_t(). uint32_t is built into XDR.	xdr_uint64_t()
string	The associated "C" API function is xdr_string(). string is built into XDR.	xdr_string()
opaque	The associated "C" API function is xdr_opaque(). opaque is built into XDR.	xdr_opaque()
OpSet_t	Signifies the operation will set a value.	xdr_Op() With the value set to OpSet_t.
OpGet_t	Signifies the operation will get a value.	xdr_Op() With the value set to OpGet_t.
OpUpdate_t	Signifies the operation will update an existing value.	xdr_Op() With the value set to OpUpdate_t.

TYPE	Notes	XDR API
OpDelete_t	Signifies the operation will delete key/ value pair.	xdr_Op() With the value set to OpDelete_t.

Table 2: Common ABNF/XDR Mapping

3.3.1. Common Definitions - ABNF

```
                ; An 8-bit unsigned integer type
uint8_t       = 0x00-ff:8

                ; A 16-bit unsigned integer type
uint16_t      = 0x0000-ffff:16

                ; A 32-bit unsigned integer type
uint32_t      = 0x00000000-ffffffff:32

                ; A 64-bit unsigned integer type
uint64_t      = 0x0000000000000000-ffffffffffffffff:64

                ; This is a generic array of UTF-8 characters without
                ; any terminating character.
                ; They could be 1, 2, 3, or 4 octet UTF-8 characters.
                ; The implemention must ensure that complete characters
                ; are containd in the string.
                ;
                ; Specific uses in this or related specifications
                ; could limit the set of characters that could be in the
string.
                ;
                ; The uint32_t value is the total number of octets in the
string.
                ;
                ; The UTF8-Char is any valid and complete UTF-8 character.
                ;
string        = uint32_t *UTF8-Char

                ; This is a generic array of uint8_t values.
                ; The data in an opaque array is not altered in any
                ; way in the protocol. It is sent over the wire unaltered.
                ;
                ; The uint32_t value is the number of octets in the data.
                ;
opaque        = uint32_t *uint8_t

                ; The time in seconds since January 1st, 1970 GMT
                ; This is known as the epoch time on many systems.
                ; And time_t on POSIX systems.
UTC_t         = uint64_t

Offset_t      = uint32_t

Length_t      = uint32_t

                ; Key and Value
KeyPair       = string string

                ; Length is the number of KeyPair that follow.
KeyPairArray  = Length 1*KeyPair

OpSet_t       = 0x00:8

OpGet_t       = 0x01:8
```

```
OpUpdate_t   = 0x02:8

OpDelete_t   = 0x03:8

Op           = uint8_t
```

Figure 4: Common ABNF Definitions

3.3.2. Common Definitions - XDR

```
typedef uint64_t UTC_t;

typedef uint32_t Offset_t;

typedef uint32_t Length_t;

struct KeyPair {
    string Key<>;
    string Value<>;
};

typedef KeyPair KeyPairArray<>;

enum Op_e {
    OpSet_t = 0x00,
    OpGet_t = 0x01,
    OpUpdate_t = 0x02,
    OpDelete_t = 0x03
};
typedef uint8_t Op;
```

Figure 5: Common Definitions - XDR

3.4. StringRef

This protocol references strings in existing MIME objects by octet offset into the MIME object. This is is called a StringRef. All strings can be referenced by using a total of 8 octets. A StringRef consists of two parts:

The StringRef does not contain the string, it is a reference an existing string in a MIME object.

Name	Description	XDR API
Offset_t	The octet count to the start of the string with zero being the first octet in the message.	xdr_Offset_t()
Length_t	The length in octets of the string.	xdr_Length_t()
StringRef	A String reference object.	xdr_StringRef()

Table 3: StringRef ABNF/XDR Mapping

A StringRef over the wire is 8 octets in size.

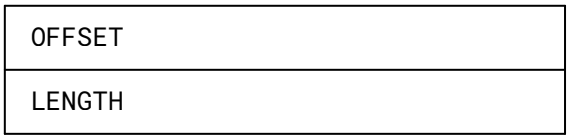


Figure 6: StringRef Format

ABNF:

3.4.1. StringRef ABNF

```
StringRef     = Offset_t Length_t
```

Figure 7: StringRef ABNF

3.4.2. StringRef XDR

The XDR definitions are:

```
struct StringRef {
    Offset_t StringStart;
    Length_t StringLength;
};
```

Figure 8: StringRef ABNF

4. Terms and Definition used in this proposal

The following is a list of terms with their definitions as used in this specification.

AdminCmd
: A general term for any administrative command. Administrative and auditing operations. This list includes commands for authorized users to configure, query logs, errors, possibly user activity.

AuthCmd
: A general term for any authentication command. Authentication and authorization operations. These operations authenticate users and verity their authorization access.

Body Part ID (BPID)
: A unique ID for a MIME Object. This is an unsigned 32-bit integer in network byte order that is assigned by the server and sent to the client on a successful folder open. This ID persists across connections. And as long as the MIME object does not get altered in any way, this ID is valid and persists across servers. It is the offset in octets from the beginning of the message to the start of the body part.

  See Index. (Section 4)

Command (CMD)
: A specific protocol operation, or command. They are broken down into, AdminCmd, AuthCmd, FileCmd, and ProtoCmd. These are called a CMD or command.

FileCmd
: A general term for any file or folder command. This include creating, getting, modifying, deleting, moving, and renaming files.

Folder ID (FolderID)
: A unique ID for a MIME folder. This is an unsigned 32-bit integer in network byte order that is assigned by the server and sent to the client on a successful folder open. This ID persists across connections to the same server. Once a folder has an ID, it never changes on a server as described in Folders (Section 5.7).

  See Index. (Section 4)

Index Operation Type (IndexOP)
: Header Index Operation. A command sent as part of a folder open command that tells the server which MIME headers it would like indexed.

  See Index. (Section 4)

Header Name ID (HID)
: And 8-bit unsigned integer the client has assiged to a specific header name. The client and server use the ID rather than passing the string value back and forth in indexes and other operations. It is not used in the MIME object.

  See Index. (Section 4)

HeaderName822
: A RFC822 or MIME header name. See Section 3.2 of [RFC0822]

HeaderID
: An offset into a MIME object where a specific header starts. As its position in a MIME object is unique, this value is also used as the offset to a specific header. As long as the MIME object does not change in any way this HeaderID persisists across connections and servers.

  See Index. (Section 4)

Header Value ID (HVID)
: Related to Header ID. An offset into a MIME object where a specific header value starts. As the position in a MIME object is unique, this value is also used as the HVID to a specific header value. As long as the MIME object does not change in any way this ID persisists across connections and servers.

  See Index. (Section 4)

  See Header ID. (Section 4)

Index
: This wire protocol transmits all or part of MIME objects. Various parts can be referenced by an offset into the object. This is an index into the MIME objects. A client may request an index be used when opening a folder.

  Note: None of these index values are guaranteed to persist across re-connections to the server, as other clients may have altered the contents.

List ID (LID)
: In operations that require a list or set of data. This LID uniquely identifies which list or set is in context.

Media Type
: Each MIME object has a media type that identifies the content of the object. This specification does not add, remove, or alter any MIME media type. This is represented in MIME objcects as the "Content-Type".

MIME
: This protocol transports MIME objects. This specification does not remove or alter any MIME objects;

  TODO - this link not valid. (Section 4)

Offset
: Unless otherwise specified, an offset is an unsigned 32-bit integer in network byte order.

Packet
: A packet is a blob of data that has a header (its length) followed by a Phoenix command with all of its values and parameters. Packets flow in both directions and asynchronously. Commands can be sent while still waiting for other replies. Each endpoint may send commands to the other endpoint without having to be prompted to send information.

Parameter
: Most commands have values that are associated with them. These values are called parameters. For example, the create folder command has the name of the new folder to be created as a parameter.

ProtoCmd
: A general term for all protocol commands. This also includes commands that do not fall into one of the other categories described here in this definitions section.

SEQ, Command Sequence, (CMDSEQ) or (SEQ)
: Each command has a unique identifier, a sequence number. All replies to a command include the same sequence number as the original command. In this way replies can be matched up with their original command.

SSL
: For the purpose of this specification, SSL is interchangeable with TLS. This document uses the term TLS. The sample implementation uses both SSL and TLS because the legacy UNIX, Linux, Windows, and OpenSSL code uses the term SSL in cases where is it TLS.

TLS
: A way of securely transporting data over the Internet.

  See [RFC8446]

XDR
: RFC-4506 specifies a standard and compatible way to transfer binary information. This protocol uses XDR to transmit a command, its values and any parameters and replies. The MIME data, the payload, is transported as XDR opaque, and is unmodified.

  Note: XDR transmits data in 32-bit chunks. An 8-bit value is transmitted with the lower 8-bits valid and the upper 24 bits set to zero. A 16-bit value is transmitted with the lower 16-bits valid and the upper 16 bits set to zero.

  So many of these protocol elements pack one or more of its parameters into one 32-bit value. As defined in each section. In many cases pseudo code is shown on how to pack the data and create the protocol element.

  See Section 3 of [RFC4506]

5. Commands

The endpoint that initiates the connection is called the client. The endpoint that is connected to, is called the server. The client is the protocol authority, and the server responds to client commands as configured or instructed by the client.

This section provides an overview of the basic commands. Each command has a detailed section in this specification.

When a command is sent to the remote endpoint and received, the remote endpoint determines if the connection is authenticated or authorized to perform the command. If not supported, or not authorized, a NotSupported command is send as a reply. The NotSupported command sent back has the same Sequence number that was in the original command.

Many commands are only valid after authentication.

When the client connects to a server it immediately sends its pre authentication capabilities to the server. Or an Auth command.

When the server gets a new connection followed by a pre authentication capability command, it immediately sends its pre authentication capabilities to the client.

When the client and server have had a relationship, the client may send an Auth Command to initiate the authorization and does not send its pre authentication capability list to the server. It then waits for the Auth reply from the server.

- If the client gets an Auth reply that is positive, it sends its post authentication capability list to the server.
- If the client gets an Auth reply that is negative, it sends its pre authentication capability list to the server.

When a servers first received packet is a Auth command, It processes the Auth command and sends the Auth reply.

- If the Auth reply is positive, then it also sends it post authentication capability list.
- If the Auth reply is negative, then it sends its pre authentication capability list to the client.

A server may automatically send its pre authentication capability list to the client upon initial connection. Or it may wait to see if it gets a pre authentication capability list, or an Auth command.

If the client sends an Auth command as its first packet, it may get the pre authentication capability from the server before the Auth reply. Simply process both.

5.1. Commands Overview - Packet and Reply

In addition to the protocols listed in this specification. Additional protocols and commands can be added in the future. They must follow the same framework listed here.

This protocol connects two endpoints over a network and facilitates the secure and authorized transfer of MIME objects.

This is a binary protocol. The payload can be anything, text or binary. This protocol was designed to reduce the number of back and forth requests and replies between the client and server. By using XDR as the format for transferring binary control information it is portable to any computer architecture. Appendix XXX has the rpcgen definition for the protocol defined in this specification.

After the connection is successful and authenticated, ether endpoint may send commands to the other endpoint. When the server initiates an unsolicited command, it could be a any kind of notification or message for the client side application or the user. It could be reporting errors or updates to previous client initiated commands.

All commands initiated from the client have even numbered command sequence numbers. All commands initiated from the server have odd numbered command sequence numbers.

Some commands expect a command reply. Other commands do not expect a command reply. An example of a command that expects a reply is the ping command. An example of a command that does not expect a reply is the keep-alive command. Conceptually there are two kinds of commands:

Directive commands: A directive type command expects the other endpoint to process the command and possibly reply with some results. An example could be: Send me an index of my emails in my InBox. The client would expect a result. Another example is a bye command, once sent, no reply is expected.

Request commands: A request type command may or might not have any reply. For example, a keep-alive command is a request to not timeout and has no reply. And a send new email notifications command would expect zero or more replies and it would not require them, as they might not happen.

These are not specific protocol entities, these concepts will be used to describe the expected behavior when one of these are transmitted.

5.1.1. Packet Overview

All commands are sent in a packet. A packet has two parts:

1. The packet header.
2. The packet body.

5.1.1.1. Packet Header

The packet header has one value, the total length of the packet body, and payload sent as an unsigned 64-bit integer in network byte order. The length does not include its own length. It is the total length that follows the length value.

Name	Description	XDR API
PacketLength	The nuber of octets that follow this value that are part of this packet.	xdr_PacketLength()

Table 4: Packet Header ABNF/XDR Mapping

5.1.1.1.1. Packet Header ABNF

ABNF:

```
PacketLength      = Length_t
```

Figure 9: Packet Header ABNF

5.1.1.1.2. Packet Header XDR

XDR Definition:

```
typedef Length_t PacketLength_t;
```

Figure 10: Packet Header XDR

5.1.1.2. Packet Body

The packet body is divided into three parts:

Name	Description	XDR API
ADMIN_CMD_t	The payload is an administration command.	xdr_ADMIN_CMD_t()
AUTH_CMD_t	What follows in the payload, is an authentication command.	xdr_AUTH_CMD_t()
FILE_CMD_t	A FILE or FOLDER payload is included.	xdr_FILE_CMD_t()
PROTO_CMD_t	The playload includes one of many protocol packets.	xdr_PROTO_CMD_t()
SEQ_t	The Command SEQ is a 32-bit unsigned integer sent in network byte order. This SEQ is an even number when initiated from the client, and an odd number when initiated from the server. The first SEQ value sent from the client is zero (0) and is incremented by two each time. The first SEQ value sent from the server is one (1) and is incremented by two each time. In the event an endpoint command SEQ reaches its maximum value, then its numbering starts over at zero (0) for the client and one (1) for the server. An implementation must keep track of outstanding commands and not accidentally re-issue the same SEQ that may still get replies from the other endpoint.	xdr_SEQ_t()

Name	Description	XDR API
CMD_t	A command (CMD) is a unsigned integer that specifics a unique operation that describes and defines the data that follows. The command are specified in this specification.	xdr_Cmd_t()
CmdPayload	The Payload is whatever data follows the command. In some cases it is a blob of opaque data. In other caes it is a structured XDR set of data. See the specific CMD for details.	xdr_CmdPayload()
PacketBody	The packet body.	xdr_PacketBody()

Table 5: Packet Body ABNF/XDR Mapping

5.1.1.2.1. Packet Body ABNF

ABNF:

```
ADMIN_CMD_t  = 0x00:32
AUTH_CMD_t   = 0x01:32
FILE_CMD_t   = 0x02:32
PROTO_CMD_t  = 0x03:32

; Define a SEQ_t (sequence) type.
SEQ_t        = uint32_t

; Define a CMD_t (command) type.
CMD_t        = uint32_t

; Define a CmdPayload_t (blob of data) type.
; The length of CmdPayload is determined by the CMD.
CmdPayload_t = *uint8_t

PacketBody = SEQ_t CMD_t
             (AdminPayload / AuthPayload / FilePayload / PROTOPayload)
```

Figure 11: Packet Body ABNF

5.1.1.2.2. Packet Body XDR

XDR Definition:

```
enum CMD_t {
    ADMIN_CMD_t  = 0x00,
    AUTH_CMD_t   = 0x01,
    FILE_CMD_t   = 0x02,
    PROTO_CMD_t  = 0x03
};

/* Define a SEQ_t (sequence) type */
typedef uint32_t  SEQ_t;

union CmdPayload_t switch (CMD_t Cmd) {

    case ADMIN_CMD_t:
        AdminCmdPayload AdminPayload;

    case AUTH_CMD_t:
        AuthCmdPayload AuthPayload;


    case FILE_CMD_t:
        FileCmdPayload FilePayload;

    case PROTO_CMD_t:
        ProtoCmdPayload ProtoPayload;
};

struct PacketBody {
    SEQ_t      SEQ;
    CmdPayload_t Payload;
};
```

Figure 12: Packet Body XDR

5.1.2. Packet Reply Overview

All replies to a command are also a command packet. They contain the same command SEQ and command as the original packet. The endpoint recognizes it is a reply because:

- The command SEQ matches one that is waiting a reply.
- When the client gets an even numbered SEQ, it can only be a reply.
- When the server gets an odd numbered SEQ, it can only be a reply.

Some commands have zero to many replies. Each of these multiple replies contains the same SEQ as the original command. An example, the client sends a request to be notified when new email arrives and uses command SEQ 20. Each time a new email arrives, a reply will be sent from the server with a command SEQ of 20. And over time, the client may get many with a SEQ of 20 as new emails arrive on the server.

5.2. Administration Commands

Implementations are not required to implement any ADMIN command. A client will know the server supports one or more ADMIN commands when it gets its post authentication capability command from the server.

Administrative command can be used to configure, audit, and manage the remote endpoint. Administrative command can be used to configure, audit, and manage user access for the server implementation.

5.2.1. Administration Capability Definitions

Implementations MUST NOT send the ADMIN capability in the pre authorization CAPABILITY list.

Implementations that support any administration command MAY include ADMIN capability in the post authentication CAPABILITY list. An implementation may decide that only specified and authorized users may issue administrative commands and send only those authenticated users the ADMIN capability.

The ADMIN capability includes the list of ADMIN commands the user is allowed to perform. For example, if a user only has permission to only view user lists, then only the USER_LIST ADMIN capability will be provided.

The capability name is also the command name to use when invoking that capability.

When a user attempts to send a commmand they are not authorized to send, the remote endpoint will reply with a NotSupported command with its sequence number set to the sequence number from offending command.

5.2.2. Administration Operations

To simplify naming, the capability names and command/reply names are the same.

The following operations are defined for administration. Each is part of an ADMIN command or ADMIN reply. They each have a unique identifier, and ADMIN Operation ID (AOID).

All of their XDR API is: xdr_AOID().

Command and Capability Name	AOID	Capability Description.	Command Description.
SERVER_CONFIGURE	0x00:8	May configure the server. If sent with a VIEW_ONLY parameter, then the user may only view the configuration information.	The command to view and alter the server configuration information.

Command and Capability Name	AOID	Capability Description.	Command Description.
SERVER_KICK_USER	0x01:8	May logout a user. And limit when they can use the server again.	The command to kick and limit a user.
SERVER_LOGS	0x02:8	May view the server logs.	The command to view server logs.
SERVER_MANAGE_BANS	0x03:8	May manage IP and user bans.	The command to manage ban users and IP addresses.
SERVER_SHUTDOWN	0x04:8	May shutdown the server.	The command to shutdown the server.
SERVER_VIEW_STATS	0x05:8	May view server statistics.	The command to view statistics.
USER_CREATE	0x06:8	May create a new user.	The command to create a Phoenix server user.
USER_DELETE	0x07:8	May delete a user.	The command to delete a user.
USER_LIST	0x08:8	May list users and their capabilities.	The command to list users.
USER_PERMISSIONS	0x09:8	May update other users permissions.	The command to view and set user permissions.
USER_RENAME	0x0A:8	May rename a user.	The command to rename a user.

Table 6: Administration Operations List

5.2.2.1. Administration Operations - ABNF

```
AOID               = uint8_t

SERVER_CONFIGURE   = 0x00:8

SERVER_KICK_USER   = 0x01:8

SERVER_LOGS        = 0x02:8

SERVER_MANAGE_BANS = 0x03:8

SERVER_SHUTDOWN    = 0x04:8

SERVER_VIEW_STATS  = 0x05:8

USER_CREATE        = 0x06:8

USER_DELETE        = 0x07:8

USER_LIST          = 0x08:8

USER_PERMISSIONS   = 0x09:8

USER_RENAME        = 0x0A:8
```

Figure 13: Administration Operations - ABNF

5.2.2.2. Administration Operations - XDR

```
enum AOID_e {
    SERVER_CONFIGURE   = 0x00,
    SERVER_KICK_USER   = 0x01,
    SERVER_LOGS        = 0x02,
    SERVER_MANAGE_BANS = 0x03,
    SERVER_SHUTDOWN    = 0x04,
    SERVER_VIEW_STATS  = 0x05,
    USER_CREATE        = 0x06,
    USER_DELETE        = 0x07,
    USER_LIST          = 0x08,
    USER_PERMISSIONS   = 0x09,
    USER_RENAME        = 0x0A
};
typedef uint8_t AOID;
```

Figure 14: Administration Operations - XDR

5.2.2.3. Administration - SERVER_CONFIGURE (0x00:8)

Remote administration of servers is an essential feature in modern systems. An implementation that supports SERVER_CONFIGURE adds SERVER_CONFIGURE to the post authentication capability command sent from the server to the client. The server implementation only sends

this capability to authenticated and authorized users. Users can become authorized with the USER_PERMISSIONS command, or by server implementation specific configuration methods. Server specific configuration methods are out of scope for this specification.

Server specific configuration options are unique to each server implementation. This specification defines a method to set, update, delete, and view server configuration values.

It is suggested that client implementations send a post authentication implementation specific capability to authenticated and authorized users in order to identify themselves to their own server implementations. If the client does not send the correnct vendor ID information, then the server would not send its SERVER_CONFIGURE capability to the client.

SERVER_CONFIGURE sends and receives the configuration information in key + value pairs. The key and value are each a string.

This specification does not define any configuration information. It provides a common way to set, get, update, and delete them.

Multiple SERVER_CONFIGURE commands can be sent in the same ADMIN packet.

Name	Description	XDR API
ConfigOp	Indicates if the key/value pairs are to get, set, update, or be deleted.	xdr_ConfigOp()
OpSet	A set of key/value paris with the same ConfigOp. Allows the client to bundle multiple key/paris per ConfigOp.	xdr_OpSet()
ServerConfigure	The ADMIN operation that sets and gets server configuration information.	xdr_ServerConfigure();

Table 7: SERVER_CONFIGURE - ABNF/XDR Mapping

5.2.2.3.1. SERVER_CONFIGURE - ABNF

```
ConfigOp         = OpSet / OpGet / OpUpdate / UpDelete
OpSet            = ConfigOp Length_t 1*KeyPair
ServerConfigure  = SERVER_CONFIGURE Length_t 1*OpSet
```

Figure 15: SERVER_CONFIGURE - ABNF

5.2.2.3.2. SERVER_CONFIGURE - XDR

```
typedef uint8_t ConfigOp;

enum ConfigOp_e {
    OpSet_t,
    OpGet_t,
                OpUpdate_t,
                OpDelete_t
};

struct OpSet {
    ConfigOp   Op;     /* With the ConfigOp_e cast to a (ConfigOp)
    KeyValue     OpValues<>;
};

struct ServerConfigure {
    uint8_t   AOID;    /* With the APID_e value cast to a (AOID) */
    OpSet     SetValues<>;
};
```

Figure 16: SERVER_CONFIGURE - XDR

5.2.2.4. Administration - USER_KICK_USER

There can be only one user per connection. So the USER_KICK_USER ADMIN command has no data associated with it. When this ADMIN CMD is received by a client, it is being informed that the user is logged out and may not be able to login again.

5.2.2.5. Administration - SERVER_LOGS

Get information from the server about its implementation logs. All log information is related to the server implementation.

A SERVER_LOGS ADMIN CMD can be sent with no information, this causes the server to send all of the information for the last 24 hours.

There are 4 categories of information:

Category	Description
Error	These are anything the server implementation considers as an error.
Warning	A warning is any non critical information the server implementation thinks may be worth recording, and is not an error.
Information	This could be accounting information such as user login, logout records. Number of connections, or any other information the server implementation wishes to record. It does not include warning or error messages.

Table 8

Name	Description	XDR API
Category	Category	xdr_Category

Table 9

5.2.2.5.1. XXXX - ABNF

```
xxx
```

Figure 17: xxxx

5.2.2.5.2. XXXX - XDR

```
xxx
```

Figure 18: xxxx

5.2.2.6. Administration - SERVER_MANANGE_BANS

5.2.2.7. Administration - SERVER_SHUTDOWN

5.2.2.8. Administration - SERVER_VIEW_STATS

5.2.2.9. Administration - USER_CREATE

5.2.2.10. Administration - USER_DELETE

5.2.2.11. Administration - USER_LIST

5.2.2.12. Administration - USER_PERMISSIONS

5.2.2.13. Administration - USER_RENAME

5.3. Authentication Commands Summary

TODO

5.4. Calendar Commands Summary

These command are based on iCalendar and iTIp.

5.5. Capability Commands Summary

This section ...

5.6. EMail Commands Summary

These commands allow for the fetching and submission of EMail messages

5.7. File and Folder Commands Summary

The file operations (FileOp) have protocol names. Here are their protocol names and a breif description.

Implementations are not required to support any or all of these commands.

Op Name	Brief Description.
FOLDER_CAPABILITY	When sent as a command, request the list of folder commands supported. When sent as a reply, includes the list of folder commands supported.
FOLDER_CREATE	Create a new folder. Also the name of the capability for this permission.
FOLDER_COPY	Copy a folder. Also the name of the capability for this permission.
FOLDER_DELETE	Delete a folder. Also the name of the capability for this permission.
FOLDER_RENAME	Rename a folder. Also the name of the capability for this permission.
FOLDER_METADATA	Get, set, and update information associated with the folder. File meta data is also returned with the FOLDER_OPEN command.
FOLDER_MOVE	Move a folder. Also the name of the capability for this permission.
FOLDER_OPEN	Open a folder and get information about the folder and files in the folder.
FOLDER_SHARE	Share a folder. Also the name of the capability for this permission.
FOLDER_LIST	List folders and files. Also the name of the capability for this permission.
FILE_CREATE	Create a new file. Also the name of the capability for this permission.
FILE_COPY	Copy a file. Also the name of the capability for this permission.
FILE_DELETE	Delete a file. Also the name of the capability for this permission.
FILE_RENAME	Rename a file. Also the name of the capability for this permission.
FILE_METADATA	Get, set, and update information associated with the folder. File meta data is also returned with the FOLDER_OPEN command.
FILE_MOVE	Move a file. Also the name of the capability for this permission.

Op Name	Brief Description.
FILE_SHARE	Share a file. Also the name of the capability for this permission.
FILE_GET	Get a file. Also the name of the capability for this permission.
FILE_MODIFY	Modify the contents of an existing file. Also the name of the capability for this permission.

Table 10: File and Folder Command List

5.7.1. File and Folder - FOLDER_CAPABILITY

5.7.2. File and Folder - FOLDER_CREATE

5.7.3. File and Folder - FOLDER_COPY

5.7.4. File and Folder - FOLDER_DELETE

5.7.5. File and Folder - FOLDER_RENAME

5.7.6. File and Folder - FOLDER_METADATA

5.7.7. File and Folder - FOLDER_MOVE

5.7.8. File and Folder - FOLDER_OPEN

5.7.9. File and Folder - FOLDER_LIST

5.7.10. File and Folder - FOLDER_SHARE

5.7.11. File and Folder - FILE_CREATE

5.7.12. File and Folder - FILE_COPY

5.7.13. File and Folder - FILE_DELETE

5.7.14. File and Folder - FILE_RENAME

5.7.15. File and Folder - FILE_METADATA

5.7.16. File and Folder - FILE_MOVE

5.7.17. File and Folder - FILE_SHARE

5.7.18. File and Folder - FILE_GET

5.7.19. File and Folder - FILE_MODIFY

5.8. KeepAlive Command Summary

The KeepAlive command is sent to the server from the client. It requests the server not time out. The server may honor or ignore the request.

The Phoenix protocol is designed to transfer data and a server may handle a small subsets of what is possible. Which is why the server decides what is an important command while determining idle timeout.

When the server sends the post authentication capabilities to the client, it includes an IdleTimeout capability that includes the number of seconds it allows for idle time. If no significant action has been taken by the client, as determined by the server, in that time the server may timeout and close the connection.

The KeepAlive command tells the server that the client wishes the server not to time out as long as a KeepAlive or other command is sent to the server before IdleTimeout seconds have passed.

An IdleTimeout capability can be a positive number, zero, or a negative number.

- A positive number is the maximum idle time in seconds before the server terminates the connection.
- When the IdleTimeout is zero (0), the server does not timeout.
- When the IdleTimeout is less than zero (< 0), it means it ignores KeepAlive and it will idle out in the absolute value of the IdleTimeout value in seconds. For example, a value of (-300) means it will ignore KeepAlive and timeout when the server determines nothing significant has happened in 5 minutes (300 seconds).

Servers that are not threaded or can not reply to simultaneous or overlapping commands, MUST set their IdleTimeout to zero (0) or a negative number.

Clients MUST NOT send KeepAlive commands to a server that has an IdleTimeout of zero (0) or negative (< 0).

Clients MUST NOT send KeepAlive commands to the server until at least 75% of the idle time has passed since the last command has been sent to the server.

A server may terminate a connection if the server implementation determines that KeepAlive commands are arriving to quickly.

5.9. Ping Command Summary

The ping command is only sent when the client implementation has determined it has waited too long for a command reply. The ping command is only initiated from the client. It is not valid for the server to send a ping command to a client.

The ping command MUST NOT be the first command sent to the server. It should only be sent when the client implementation determines it has waited too long for a reply.

If the server supports the ping command, then a PING capability is sent in the pre authentication capability command.

Sometimes servers are unavailable and can go down. A server could be down for maintenance, or in a shutdown mode. It might limit the number of simultaneous connections. It might be very busy. The packets might not be making it to the server because of network issues.

When a ping command is received by the server:

- When the server did not send PING capability in the post authentication capability list to the client. Then the server replies with a NotSupported packet with the sequence number the same as in the ping command.
- When the server has not yet received an authentication command, the server replies with a NotSupported packet with the sequence number the same as in the ping command.
- When the server has received an authentication command, and has not yet replied to an authentication command. Then the server sends a ping reply, with the same sequence number that was in the ping command. This could happen when the client implementation had determined it has waited too long for an authentication reply.
- When the client is authenticated, and when the server is available for processing commands. Then the server replies with a ping reply with the same sequence number. This could happen when the client implementation had determined it has waited too long for an expected reply.

If the server is alive and not available, the server will reply with a NotSupported command, with its sequence number set to the sequence number in the ping command.

If a connected and authenticated client has been waiting for a reply or for some other reason needs to determine if the server is still available. It can send a ping command. If the server is still available, it sends a ping reply. If it is no longer available for any reason, it sends a NotSupported reply.

A client MUST NOT send a ping command if it is waiting the results of a previously sent ping command.

A client MUST NOT send a ping command more frequently than 90% of the server timeout.

Clients and servers must give priority to ping commands. If possible, reply as soon as it receives the command.

The server MAY consider too many ping commands as a malfunctioning or malicious client and terminate the connection.

Servers that are not threaded or can not reply to simultaneous or overlapping commands, MUST NOT include PING in their post authentication capability command.

5.10. S/MIME Commands Summary

ToDo

6. Meta Data with Shared Objects

When a server implementation allows shared objects, the meta data returned to the client may be different depending on the authenticated user. Some users may have read only copies, other may be able to delete the object.

When a shared object is deleted, it is marked as deleted for only the user that issued the delete.

When a shared object is expunged, its access is removed for the user that issued the expunge. After all users have expunged the object, then it is removed by the server.

There are two kinds of expunge for shared objects. Forced and Delayed.

Server implementations must reject attempts to fetch or view a folder or file or any of its meta data when an expunge has started, and not yet completed.

- Forced:

  A forced expunge can be the result of security policies at the server, site, or administrators discretion. This also is how timed messages are deleted.

  In order for a shared object that is expunged to not force an immediate re-index for all clients, when the server gets a forced expunge, the server sends an expunge to all clients, where the client MUST immediately make the object not show to the user and MUST invalidate any file, cached, or memory copy of the data the client has control over. Then when convenient, the client can do a re-index of the folder. When a user is viewing the object when an expunge arrives, the client must inform the user that the data is no longer available and replace the user view of the data with an empty object or move the view to another object.

  Server implementations must prioritize forced expunge notices to the clients and immediately reject all attempts to read, view, copy, or access meta data.

- Delayed:

  The user is informed the MIME object is no longer available. The client implementation may continue to show the object. The client may copy the MIME object, unless tagged as NoCopy.

  The next time the client does an expunge the object will be expunged from the client.

  When a client application closes, all delayed expunges MUST occur at exit.

  When a client applications starts the client MUST check for delayed expunges that have not been processed and expunge them and not allow the user to see them.

7. Meta Data

In this specification a file and a MIME object are used interchangeably. Meta Data is data that is associated with the MIME object and not contained within the MIME object. Meta Data should never be stored in the MIME object as altering the MIME object would invalidate the index information and can invalidate digital signature and encryption information.

Meta Data for the folder and MIME objects is returned in a FOLDER_OPEN, FILE_OPEN, FILE_METADATA, or FOLDER_METADATA command. Meta Data can be set and updated by the client using FILE_METADATA or FOLDER_METADATA commands.

Most are 8-bit boolean values that are set to false (0x00) or true (0x01). A value that does not exists is the same as a false.

Meta data can be global to the object. That is once tagged (or not tagged) the attribute shows up for all users. Or it can be user specific meta data. User specific meta data does not show up for other users.

Many have the same or similar name and meaning as they do in IMAP [RFC9051].

7.1. Meta Data - Answered

This Meta Data only applies to files.

When true, the object has been replied to by the client. This has the same meaning as \Answered does in IMAP.

This value can be set and unset. This is user specific meta data because it also applies to shared folders and files.

7.2. Meta Data - Attributes

This object has been tagged with special attributes. It is a list of strings with matching values.

User defined attributes MUST start with "X-". These are not portable between implementations and no attempt should be made to copy these between implementations.

Non user defined attributes are described in other sections or specifications.

This can be user specific meta data or global meta data. See the specific attribute documentation.

7.3. Meta Data - Deleted

When true, this object has been marked as deleted and has not yet been expunged. This has the same meaning as \Deleted does in IMAP.

For shared objects, an expunge removes the user from shared access to the file. And the actual expunge is only processed when all shared users have expunged the object.

This value can be set and unset. This is user specific meta data because it also applies to shared folders and files.

7.4. Meta Data - Draft

This Meta Data only applies to files.

When true, this object is incomplete and not ready.

This has the same meaning as \Draft does in IMAP.

This value can be set and unset. This is user specific meta data.

7.5. Meta Data - Flagged

An object has been tagged as important. This is the same as the IMAP \Flagged value.

This value can be set and unset. This is user specific meta data because it also applies to shared folders and files.

7.6. Meta Data - Forwarded

This Meta Data only applies to files.

This has the same meaning as $Forwarded does in IMAP.

This value can be set and unset. This is user specific meta data because it also applies to shared folders and files.

7.7. Meta Data - Hide

With NotExpungable objects, the user may wish to not view the object. In these cases the attribute Hide can be set. The attribute does not effect the view of other users.

This value can be set and unset. This is user specific meta data because it also applies to shared folders and files.

7.8. Meta Data - Junk

This has the same meaning as $Junk does in IMAP.

This value can be set and unset. This is user specific meta data because it also applies to shared folders and files.

7.9. Meta Data - MDNSent

This Meta Data only applies to files.

This value can be set and unset. This has the same meaning as $MDNSent does in IMAP.

This value can be set and unset. This is user specific meta data because it also applies to shared folders and files.

7.10. Meta Data - NoCopy

When true, this MIME object can not be copied.

This value can be set and unset by the owner of the file or folder. This value can not be unset by non owners. This is global meta data.

7.11. Meta Data - NotJunk

This has the same meaning as $NotJunk does in IMAP.

This value can be set and unset. This is user specific meta data because it also applies to shared folders and files.

7.12. Meta Data - NotExpungable

The mime object can not be marked for delete or expunged. It could be because it is an historical record that will never be expunged, or other reason.

A client implementation could use the Hide attribute to not show the object to the user.

This value can be set and unset by the owner of the file or folder. This value can not be unset by non owners. This is global meta data.

7.13. Meta Data - Phishing

This has the same meaning as $Phishing does in IMAP.

This value can be set and unset. This is user specific meta data because it also applies to shared folders and files.

7.14. Meta Data - ReadOnly

The MIME object associated with this attribute can not be altered, deleted, moved, or renamed. It can be copied, unless the NoCopy meta tag is also applied.

This value can be set and unset by the owner of the file or folder. This value can not be unset by non owners. This is global meta data.

Setting of this to false may fail if the file or folder is stored on read-only media. When the file or folder is stored on read-only media, this MUST BE set to true.

7.15. Meta Data - Shared

The MIME object associated with this attribute is shared and is also often tagged with the ReadOnly meta data tag.

This value can not be set and unset by the owner.

If copying of the file or folder is allowed, then the shared attribute is removed when copied.

This file or folder will only be expunged when all of the users with shared access have deleted and expunged it.

7.16. Meta Data - Seen

This has the same meaning as \Seen does in IMAP.

This value can be set and unset. This is user specific meta data because it also applies to shared folders and files.

7.17. Meta Data - MDNData Attribute

This Meta Data Attribute is only visible to the owner of the object for which MDN has been set.

This is a list of recipients email address that that are on the distribution list effected by the MDN.

7.17.1. MDNRecord

The format of an MDNRecord:

Name	XDR Type	Description
MDNSent	uint64_t	The UTC timestamp as a 64-bit unsigned integer in network byte order of when the MDN reply was sent.
MDNListCount	uint32_t	A 32-bit unsigned integer in network byte order indicating how many were on the distribution list for this MDN.

Table 11

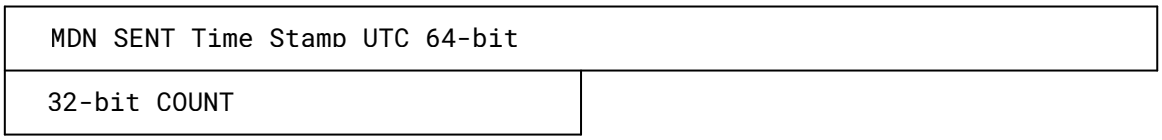


Figure 19: MDNRecord

7.17.1.1. MDNRecord - ABNF

ABNF:

```
MDNListCount = Length_t

MDNRecord    = MDNSent MDNListCount 1*MDNRecord
```

7.17.1.2. MDNRecord - XDR

XDR:

```
MDNListCount = Length_t

MDNRecord    = MDNSent MDNListCount 1*MDNRecord
```

Followed by MDNListCount MDNEntry's.

7.17.2. MDNEntry

Name	XDR Type	Description
UTC	uint32_t	The UTC timestamp as a 64-bit unsigned integer in network byte order of when the MDN reply was received. Set to zero if not received.
EMail Length	uint32_t	The number of octets in the email address that follows. Not including any terminating zero.
EMailAddress	string	A string of the associated email address of the user that has or has not returned the MDN.

Table 12: MDMEntry ABNF/XDR Mapping

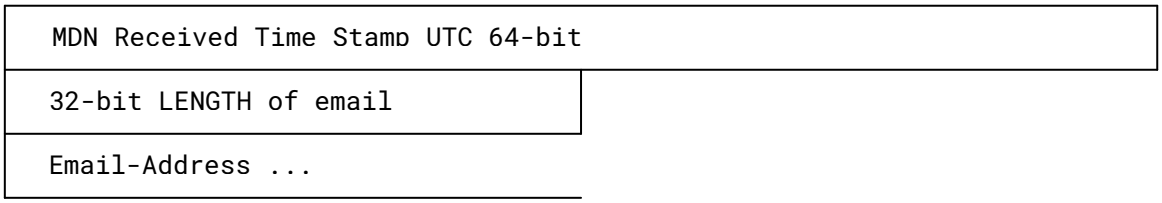


Figure 20: MDNEntry

7.17.2.1. MDNEntry ABNF

ABNF:

```
MDNEntry  = UTC_t string
```

Figure 21: MDNEntry - ABNF

7.17.2.2. MDNEntry - XDR

XDR:

```
struct MDMEntry
{
    UTC_t  Received;
    string EMail<>;
};
```

Figure 22: MDNEntry - XDR

8. Index

8.1. Interested Headers

Some implementation may wish to specify which MIME headers it wants to get in the index supplied by the server. This is done as part of the folder selection command which can supply a list of desired headers. Or it can specify a list ID that has already been transmitted. When none are supplied, no header index values will be returned.

This list can be the same for all folders, or unique to specific folders. The client generates a list of interested headers and sends an Interested Headers list or list ID to the server when selecting a folder.

8.1.1. List ID (LID)

A List ID (LID) is a unsigned integer ranging from 0 to 254. It is used in requests and replies to refer to the interested headers list. A client can have up to 254 (LID 0 to 254) lists per connection. The value 255 is reserved for expansion.

Restrictions:

- The list IDs are unique to the connection and do not persist across connections.
- No two lists can have the same ID within a connection.

ABNF:

```
LID    = uint8_t;
```

8.1.2. Index Operation (IndexOP)

An Index Operation (IndexOP) has only one of two values:

- IndexOPDefine = 0

  Used to define a list of body MIME object, and Body Part, interesting headers the client cares about. When the LID is already defined, then this redefines it. When the LID is not already defined, it creates a new list. The results will come back as an Folder-Index in a successful FOLDER_OPEN reply.

- IndexOPUse = 1

  This indicates that LID is an existing list number to use. LID has previously been defined in this session. The results will come back as an Folder-Index in a successful FOLDER_OPEN reply.

ABNF:

```
IndexOpDefine   = 0x00:8

IndexOpUse      = 0x01:8

IndexOp         = IndexOpDefine / IndexOpUse
```

8.1.3. Header ID (HID)

A Header ID (HID) is an unsigned integer ranging from 0 to 254. The client assigned the HID value to a header name, then the client and server references it by HID in packets and replies. A client can have up to 254 interested headers per connection. The value 255 is reserved for expansion.

ABNF:

```
HID    = uint8_t;
```

8.1.4. Lists

The client sends a list to the server as part of a FOLDER_OPEN. One of the parameters to a FOLDER_OPEN is an interested header list. A successfule reply to a FOLDER_OPEN will include indexes into the MIME object for the desired header values.

The list can be defined in the same packet. Or it can use an already defined list. Or it can not request any header indexes by defining or using a list that has zero entries.

Figure 23, shows the interisted header list prefix. This interisted header list prefix is followed by zero or more SingleEntry objects.

Name	XDR Type	Description
IndexOP	uint8_t	One of IndexOpDefine or IndexOpUse
IndexOPDefine	uint8_t	Define or redefine a list.
IndexOPUse	uint8_t	Use an already defined list.
LID	uint8_t	LID is the list ID of the list that client is defining. With 255 reserved for expansion.

Name	XDR Type	Description
HDRCNT	uint32_t	HdrCnt is set by the client to the number of headers in the list. With 0xffffff reserved for expansion.

Table 13

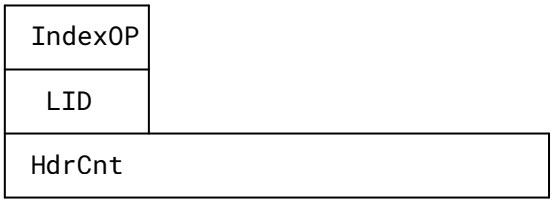


Figure 23: Interest Header List Prefix

ABNF:

```
HdrCnt          = uint32_t

Interest-Header = IndexOp LID HdrCnt
```

8.1.4.1. Interested Headers - Single Entry

Following interest header list prefix data is zero or more of these single header entries. One sent for each HdrCnt in the prefix. This list informs the server the HID value that will be used for each interested header in the index that the server replies with. As shown in Figure 24, where:

Name	XDR Type	Description
HID	uint8_t	HID is the client assigned unique header ID for the named header. This is an 8-bit unsigned integer.
HEADER NAME	StringRef	THE HEADER NAME is the characters that make up the MIME header name that is interesting without including any terminating zero (0).

Table 14

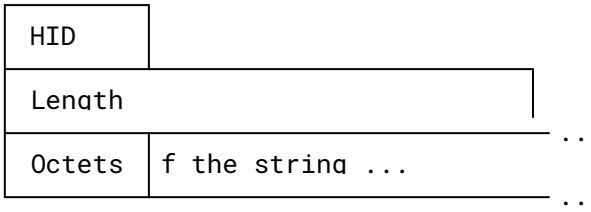


Figure 24: Setting the Interest List - Contents

ABNF:

```
SingleHeader = HID StringRef
```

8.1.4.2. Interested Headers - Use Existing List

When the IndexOP flag is set to one (1) then it is followed by an existing list ID number.

LID, the list ID of an already transmitted list to be used.

This is sent as a 32-bit unsigned integer in network byte order.

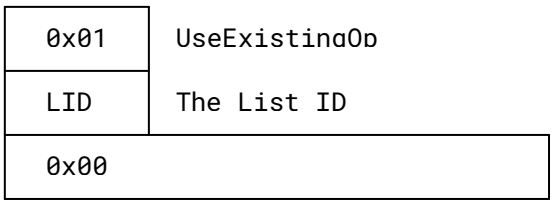


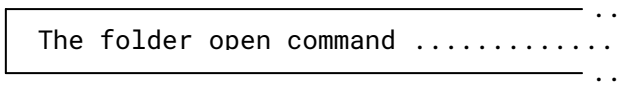
Figure 25: Using Existing Header Index by List ID (LID)

ABNF:

```
UseExistingOp      = 0x01:8

UseExistingList:32 = UseExistingOp LID 0x00:8
```

8.1.4.3. Example: Setting the Interested Header List

This is an example of the client sending an interesting header list to the server. The client is asking for the index values for the following MIME headers (1) From, and (2) Subject. And for the following Body part headers (1) Content-Type.



Immediately followed by:

	IndexOP	LID		2 MIME Object Headers included.
(a)	0x00	0x00	0x00	0x02	Start of MIME object List
(b)	0x00	0x000004			Header (0), Length (4)
(c)	0x46	0x72	0x6f	0x6d	"From"
(d)	0x01	0x000007			Header (1), Length (7)
(e)	0x53	0x75	0x62	0x6a	"Subject"
	0x65	0x63	0x74	0x00	
(f)	0x00	0x00	0x00	0x01	Start of Body Part List
(g)	0x02	0x00000c			Body Header (2), Length (12)
(h)	0x43	0x6f	0x6e	0x74	"Content-Type"
	0x65	0x6e	0x74	0x2d	
	0x74	0x79	0x70	0x74	

Figure 26: Example Setting a List

Where:

- (a): A 32-bit unsigned integer in network byte order as described in Figure 23.

  The first 8-bits are zero.

  The IndexOP of zero, which means defining a list.

  And in this example two (0x02) MIME object headers are requested to be indexed, "From", and "Subject".

- (b): A 32-bit unsigned integer in network byte order as described in Figure 24.

  The header ID that the client and server will use to to identify the "From" header name will be zero (0) in this example.

  The length of the string "From" is four (4) and its length is the lower 24 bits of this entry.

- (c) A series of 8-bit unsigned values packed into one or more 32-bit unsigned integers in network byte order.

Each 8-bit value is the value of the letters in "From". As "From" is a multiple of 32-bits, no padding is done.

- (d): A 32-bit unsigned integer in network byte order as described in Figure 24.

  The header ID that the client and server will use to to identify the "Subject" header name will be one (1) in this example.

  The length of the string "Subject" is seven (7) and its length is the lower 24 bits of this entry.

- (e) A series of 8-bit unsigned values packed into one or more 32-bit unsigned integers in network byte order.

  Each 8-bit value is the value of the letters in "Subject".

  As The length of "Subject" is not a multiple of 32-bits, the remaining bits are ignored. Shown as zero in this example.

- (f) The two MIME objects headers are done, start of Body Part headers, and there is one (1) of them. IndexOP and LID are not used here.
- (g) The second header will be identified as three (3). The first body part header is 12 octets long (0xc): 'Content-Type'.
- (h) The value of the characters for 'Content-Type'.
- (i) The rest of the value of the characters for 'Content-Type'.

8.2. MIME Folder Index

In this specification, a MIME folder is also called a folder. And can be files containing MIME objects on a disk that have a defined order, or sequence of MIME objects in one file.

A folder index is a summary of the contents of a MIME folder. It may include the basic header information. It does include location information provided as the octet count to the start of the beginning of the related target data.

- An index is an unsigned 32-bit integer in network byte order.
- A length is an unsigned 32-bit integer in network byte order.

For example, if a MIME folder contains 100 MIME messages, then the folder index will have 100 message indexes. Each message will have header indexes for the interested headers. Each message index will contain 1 or more body part indexes. Each body part will have header indexes with zero (0) or more entries.

8.2.1. Folder Index Header

A folder index consists of:

- The entire length of the index as a 32-bit unsigned integer in network byte order of what follows this value. Allowing the recipient of this index to do one read and process later.
- The number of message indexes in this folder index. As an unsigned 32-bit integer in network byte order.

The index header is 8 octets, that is followed by the each message index:

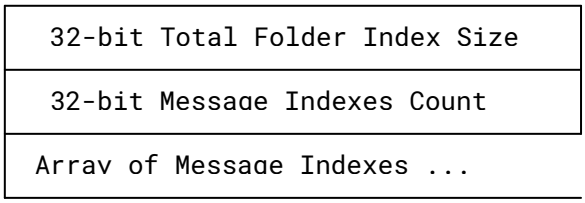


Figure 27: Folder Index

ABNF:

```
FolderIndexHeader = FolderIndexSize:32 MessageCount:32 ArrayOfMsgIndex
```

The header is followed by an array of message indexes. They are an ordered list of references to each message. In the order they appear in the folder:

8.2.2. Message Index

- A 32-bit unsigned integer in network byte order that is the offset into the folder of the message. A Message offset is unique in a MIME folder, it is used both as an offset into the MIME folder, and as a unique ID within a MIME folder for a message.
- An a length of the message as a 32-bit unsigned integer in network byte order.

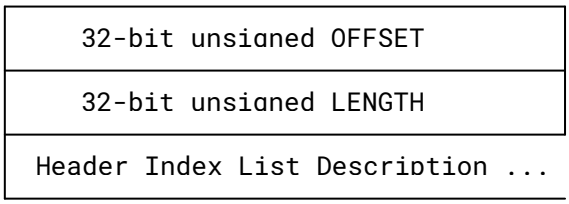


Figure 28: Message Index

ABNF:

```
MessageLength     = uint32_t
OffsetIntoFolder = uint32_t
MessageIndex     = OfffsetIntoFolder MessageLength ArrayOfHeaderIndex
ArrayOfMsgIndex = *MessageIndex
```

For each message index is an ordered list of interested headers. The interested header list is assignable by the client and body part indexes. It consists of offsets to the interested headers and associated value. Each interested header can be indexed with nine (9) octets. and consists of:

8.2.3. Header Index

- ID-CNT: A count of matched headers. Only matched headers will be included. If they are not included, no such header existed in the object.
- The number of body parts in this object. An unsigned 8-bit number. With MIME, body parts may contain body parts.

  Any MIME preamble and epilogue are not counted as body parts A preamble, if it exists, can be easily be calculated as it starts as the first octet after the header area. And the epilogue, if it exists, can be calculated as starting as the first octet after the last MIME boundary.
- Followed by an array of ID-CNT 8-bit client assigned HID values that matched. Padded to round up to 32-bits. The unused bits are ignored and shown as zero in this specification.

A single header index consists of the list description, followed by the index values. There are two header indexes in each Message index.

1. The first is for the MIME object itself.
2. The second is for the objects Body Parts. This part will not exists exist when it is an RFC-822 style message or has no body parts. Followed by the header index. This second part also include an offset to the start of the body part itself in the MIME object.

A list description is one 8-bit result count, followed by the list of matching header ID's (HID).

If the list description is not a multiple of 32-bits then padding is added and the extra are ignored and shown as zero in this specification.

```
-Meta-Data-: Seen,Answered,$NotJunk
```

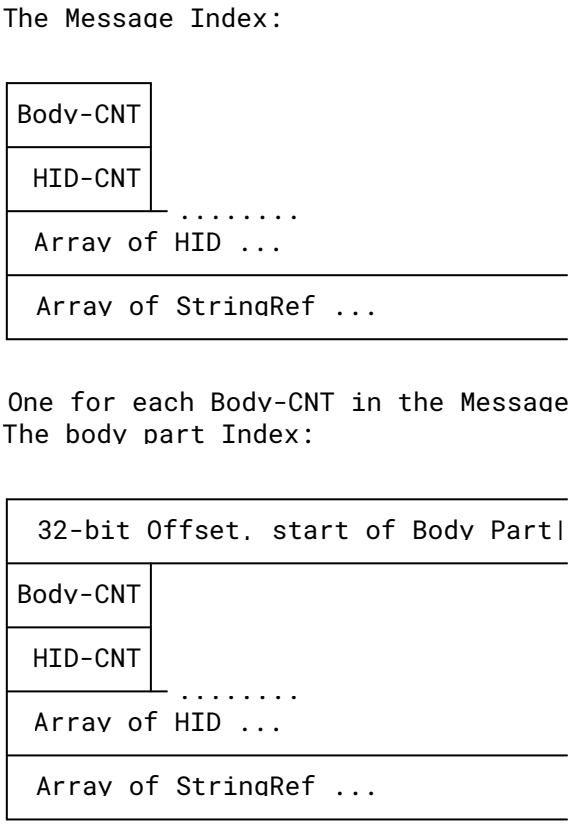


Figure 29: Header Index

ABNF:

```
HeaderIndex          = HeaderIndexHeader:32 *ArrayOfHID
                       *StringRef *BodyPartIndex

                       ; One HID (HeaderID) for each match header
                       ; in the LID provided. Padded out to multiples
                       ; of 32-bits.
HeaderIndexHeader:32 = ID-CNT:8 Body-Count:8
                       / (HID HID)
                       / (HID 0x00:8)
                       / (0x00:8 0x00:8)

ArrayOfHid           = *HIDEntry

BodyPartIndex        = BodyPartOffset:32 HeaderIndexHeader:32 *StringRef
                       *BodyPartIndex

ID-CNT:8               ; The number of headers found in the
                       ; MIME object and requested in the interested
                       ; header list.

Body-CNT:8             ; The number of body parts in the object

                       ; Padded out to multiples of 32-bits.
HidEntry             = (HID:8 HID:8 HID:8 HID:8)
                       / (HID:8 HID:8 HID:8 0x00:8)
                       / (HID:8 HID:8 0x00:8 0x00:8)
                       / (HID:8 0x00:8 0x00:8 0x00:8)
```

Where:

HeaderIndex: The header index starts with a 32-bit unsigned integer in network byte order, the HeaderIndex:32.

HeaderIndex:32: Contains 0, 1, or 2 HID values. They are in the order found in the object.

ArrayIfHID: Keeps repeating until all of the headers in the list have been found in the message. The last one pads with zeros when needed.

BodyPartIndex: When the object has body parts, there will be a BodyPartIndex for each body part, in the order they are in the object. The first 32-bits are the offset to the start of the body part. This does not include any boundary.

Body parts themselves may contain body parts, they are recursively included as needed.

8.2.4. Header Index Example 1

For example, if the client requested MIME object indexes for the "From", "Subject", "To", "Message-ID", "Content-Type", "MIME-Version", and "Date" header values.

Assume this is an RFC-822 message with no body parts. So the body part header index has a count of zero (0). And the HID values assigned by the client when opening the folder are:

- From: 0

- Subject: 1
- To: 2
- Message-ID: 3
- Content-Type: 4
- Data: 5
- MIME-Version: 6

In the Message each line is terminated with a carriage return and line feed:

```
From: Doug@example.com
To: Notices@example.com, Supervisors@example.com, Dave@example.com
Date: Thu, 06 Feb 2025 20:29:35 +0000
MessageID: <7324e0b9-f6dc-3c9b-a02f-0b2b824e863c@example.com>
Subject: A new draft of Phoenix has been published.
Content-Type: text/plain

A new draft has been published.
```

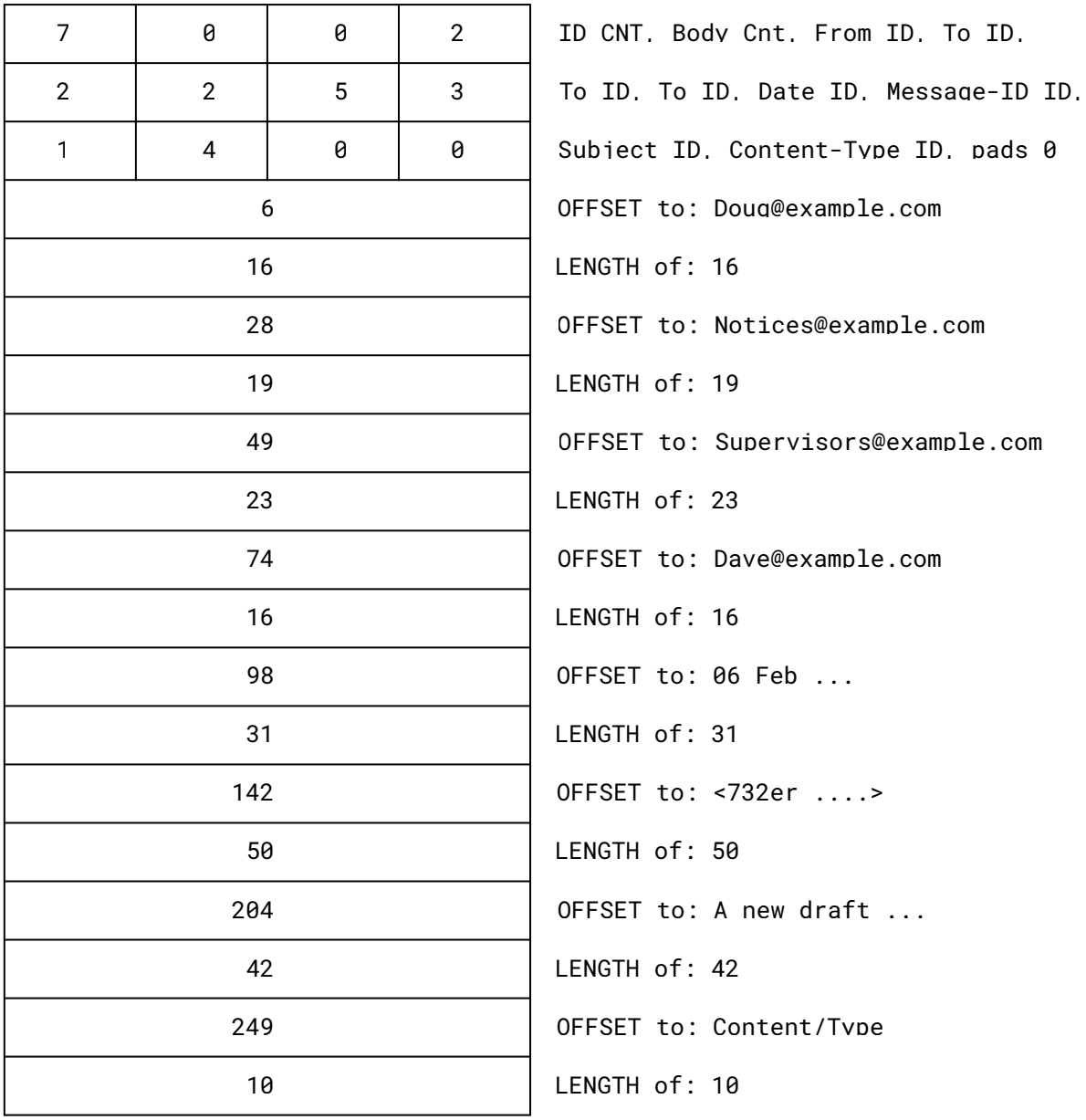


Figure 30: Header Index

8.2.5. Header Index Example 2

For example, if the client requested MIME object indexes for the "From", "Subject", "To", "MIME-Version", and "Content-Type". header values.

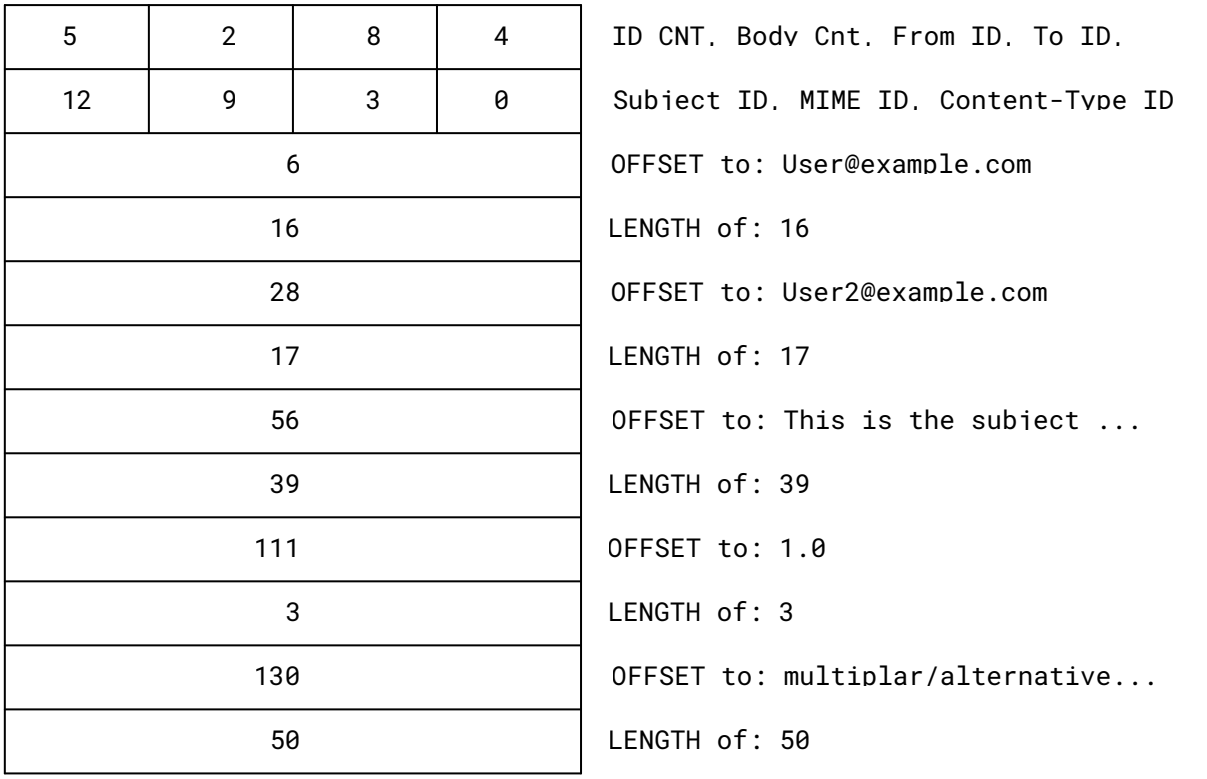
And when the folder was opened, the client asked for the "Content-Type" header.

Assume this is a MIME message with two body parts. So the body part header index has a count of two (2). And the HID values assigned by the client when opening the folder are:

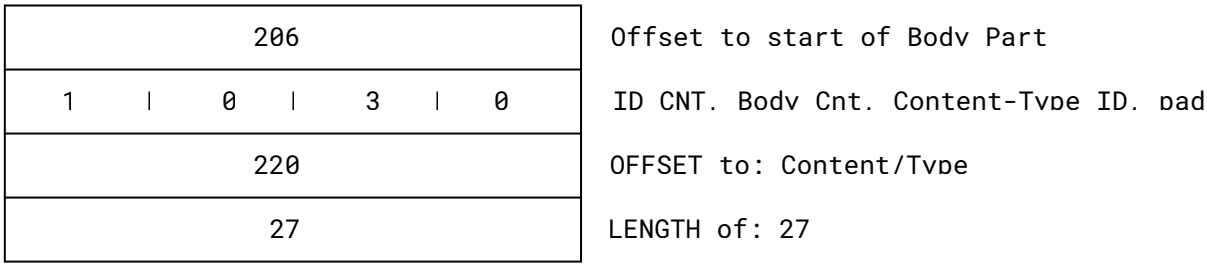
- From: 8
- Subject: 12
- To: 4
- Content-Type: 3
- MIME-Version: 9

In the Message each line is terminated with a carriage return and line feed:

```
From: User@example.com
To: User2@example.com
Subject: This is the subject of a sample message
MIME-Version: 1.0
Content-Type: multipart/alternative; boundary="XXXXboundary text"

--XXXXboundary text
Content-Type: text/plain; charset="utf-8"
Content-Transfer-Encoding: quoted-printable

This is the body text of a sample message

--XXXXboundary text
Content-Type: text/html; charset="utf-8"
Content-Transfer-Encoding: quoted-printable

This is the body text of a sample message.
--XXXXboundary text--
```



Next is the data for the first body part.
This one body part has no body parts, so its Body Cnt is zero.



Then the second body part:

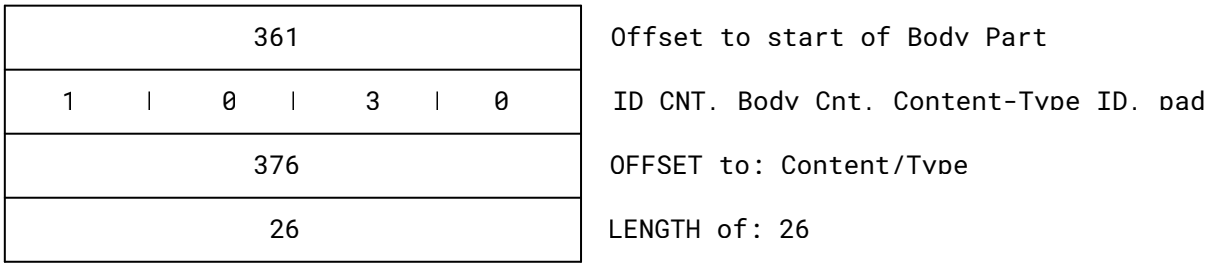


Figure 31: Header And Body Part Index

9. IANA Considerations

This memo includes no request to IANA. [CHECK]

10. Security Considerations

This document should not affect the security of the Internet. [CHECK]

11. References

11.1. Normative References

[RFC0822] Crocker, D., "STANDARD FOR THE FORMAT OF ARPA INTERNET TEXT MESSAGES", STD 11, RFC 822, DOI 10.17487/RFC0822, August 1982, <https://www.rfc-editor.org/info/rfc822>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC4506] Eisler, M., Ed., "XDR: External Data Representation Standard", STD 67, RFC 4506, DOI 10.17487/RFC4506, May 2006, <https://www.rfc-editor.org/info/rfc4506>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

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

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

[RFC9051] Melnikov, A., Ed. and B. Leiba, Ed., "Internet Message Access Protocol (IMAP) - Version 4rev2", RFC 9051, DOI 10.17487/RFC9051, August 2021, <https://www.rfc-editor.org/info/rfc9051>.

11.2. Informative References

[rpcgenopensource] Unknown Author, "rpcgen++ Open Source Tool", January 1983, <https://github.com/RiverExplorer/Phoenix/tree/main/rpcgen%2B%2B-src>.

[rpcgendocs] Unknown Author, "rpcgen Protocol Compiler", January 2025, <https://www.ibm.com/docs/en/aix/7.3?topic=call-rpcgen-protocol-compiler>.

[PhoenixImplementation] Royer, D., "Phoenix Sample Implementation", 2025, <https://github.com/RiverExplorer/Phoenix>.

Appendix A. Administrative Enumerated Binary Values

Phoenix is a binary protocol. Each value is sent as an unsigned 32-bit integer in xdr format.

The values for the commands are arbitrary and were assigned as created. There is no plan or origination to the numbers. There is no priority or superiority to any value. The table is sorted by name, not value.

The values are not unique. They are only unique within the context in which they are used.

Some of these values are reused for other commands. For example USER_CREATE is both an (a) AUTH capability reply informing the user that they have permission to create a user with the (b) USER_CREATE command.

Some values may be reused if they are parameter arguments to other commands. For example xxxxxx.

Decimal Value	Command / Capability Name	Brief Description.
x	USER_CERT	Manage a users certificate.
x	USER_CREATE	When sent in a capability reply USER_CREATE informs the user that they have permission to create users. When sent as a command the USER_CREATE instructs the other endpoint to create a named user.
x	USER_DELETE	Delete a user.
x	USER_LIST	List users and their capabilities.
x	USER_PERMISSIONS	Update user permissions.
x	USER_RENAME	Rename a user.
x	USER_RESET	Used to coordinate resetting a users authentication information.
4294967296	Reserved for future expansion.	4294967296 has a hex value of: 0xffffffff

Table 15

Appendix B. Authentication Enumerated Binary Values

Phoenix is a binary protocol. Each value is sent as an unsigned 32-bit integer in xdr format.

The values for the commands are arbitrary and were assigned as created. There is no plan or origination to the numbers. There is no priority or superiority to any value. The table is sorted by name, not value.

The values are not unique. They are only unique within the context in which they are used.

Some of these values are reused for other commands. For example USER_CREATE is both an (a) AUTH capability reply informing the user that they have permission to create a user with the (b) USER_CREATE command.

Some values may be reused if they are parameter arguments to other commands. For example xxxxxx.

Decimal Value	Command / Capability Name	Brief Description.
x	AUTH_TODO	xxx.
xxx	AUTH_xxx	xxx.
4294967296	Reserved for future expansion.	4294967296 has a hex value of: 0xffffffff

Table 16

Appendix C. File and Folder Enumerated Binary Values

Phoenix is a binary protocol. Each value is sent as an unsigned 32-bit integer in xdr format.

The values for the commands are arbitrary and were assigned as created. There is no plan or origination to the numbers. There is no priority or superiority to any value. The table is sorted by name, not value.

The values are not unique. They are only unique within the context in which they are used.

Some of these values are reused for other commands. For example USER_CREATE is both an (a) AUTH capability reply informing the user that they have permission to create a user with the (b) USER_CREATE command.

Some values may be reused if they are parameter arguments to other commands. For example xxxxxx.

Decimal Value	Command / Capability Name	Brief Description.
x	FILE_TODO	xxx.

Decimal Value	Command / Capability Name	Brief Description.
xxx	FILE_xxx	xxx.
4294967296	Reserved for future expansion.	4294967296 has a hex value of: 0xffffffff

Table 17

Appendix D. Protocol Enumerated Binary Values

Phoenix is a binary protocol. Each value is sent as an unsigned 32-bit integer in xdr format.

The values for the commands are arbitrary and were assigned as created. There is no plan or origination to the numbers. There is no priority or superiority to any value. The table is sorted by name, not value.

The values are not unique. They are only unique within the context in which they are used.

Some of these values are reused for other commands. For example USER_CREATE is both an (a) AUTH capability reply informing the user that they have permission to create a user with the (b) USER_CREATE command.

Some values may be reused if they are parameter arguments to other commands. For example xxxxxx.

Decimal Value	Command / Capability Name	Brief Description.
x	PROTO_TODO	xxx.
xxx	PROTO_xxx	xxx.
4294967296	Reserved for future expansion.	4294967296 has a hex value of: 0xffffffff

Table 18

Appendix E. Complete ABNF

```
                ; An 8-bit unsigned integer type
uint8_t       = 0x00-ff:8

                ; A 16-bit unsigned integer type
uint16_t      = 0x0000-ffff:16

                ; A 32-bit unsigned integer type
uint32_t      = 0x00000000-ffffffff:32

                ; A 64-bit unsigned integer type
uint64_t      = 0x0000000000000000-ffffffffffffffff:64

                ; This is a generic array of UTF-8 characters without
                ; any terminating character.
                ; They could be 1, 2, 3, or 4 octet UTF-8 characters.
                ; The implemention must ensure that complete characters
                ; are containd in the string.
                ;
                ; Specific uses in this or related specifications
                ; could limit the set of characters that could be in the
string.
                ;
                ; The uint32_t value is the total number of octets in the
string.
                ;
                ; The UTF8-Char is any valid and complete UTF-8 character.
                ;
string        = uint32_t *UTF8-Char

                ; This is a generic array of uint8_t values.
                ; The data in an opaque array is not altered in any
                ; way in the protocol. It is sent over the wire unaltered.
                ;
                ; The uint32_t value is the number of octets in the data.
                ;
opaque        = uint32_t *uint8_t

                ; The time in seconds since January 1st, 1970 GMT
                ; This is known as the epoch time on many systems.
                ; And time_t on POSIX systems.
UTC_t         = uint64_t

Offset_t      = uint32_t

Length_t      = uint32_t

                ; Key and Value
KeyPair       = string string

                ; Length is the number of KeyPair that follow.
KeyPairArray  = Length 1*KeyPair

OpSet_t       = 0x00:8

OpGet_t       = 0x01:8

OpUpdate_t    = 0x02:8
```

```
OpDelete_t   = 0x03:8

Op           = uint8_t
```

```
StringRef     = Offset_t Length_t
```

```
PacketLength      = Length_t
```

```
ADMIN_CMD_t  = 0x00:32
AUTH_CMD_t   = 0x01:32
FILE_CMD_t   = 0x02:32
PROTO_CMD_t  = 0x03:32

; Define a SEQ_t (sequence) type.
SEQ_t        = uint32_t

; Define a CMD_t (command) type.
CMD_t        = uint32_t

; Define a CmdPayload_t (blob of data) type.
; The length of CmdPayload is determined by the CMD.
CmdPayload_t = *uint8_t

PacketBody = SEQ_t CMD_t
             (AdminPayload / AuthPayload / FilePayload / PROTOPayload)
```

```
AOID               = uint8_t

SERVER_CONFIGURE   = 0x00:8

SERVER_KICK_USER   = 0x01:8

SERVER_LOGS        = 0x02:8

SERVER_MANAGE_BANS = 0x03:8

SERVER_SHUTDOWN    = 0x04:8

SERVER_VIEW_STATS  = 0x05:8

USER_CREATE        = 0x06:8

USER_DELETE        = 0x07:8

USER_LIST          = 0x08:8

USER_PERMISSIONS   = 0x09:8

USER_RENAME        = 0x0A:8
```

```
ConfigOp         = OpSet / OpGet / OpUpdate / UpDelete
OpSet            = ConfigOp Length_t 1*KeyPair
ServerConfigure  = SERVER_CONFIGURE Length_t 1*OpSet
```

```
MDNListCount = Length_t
MDNRecord    = MDNSent MDNListCount 1*MDNRecord
```

```
MDNEntry  = UTC_t string
```

Appendix F. Complete XDR

```
typedef uint64_t UTC_t;

typedef uint32_t Offset_t;

typedef uint32_t Length_t;

struct KeyPair {
    string Key<>;
    string Value<>;
};

typedef KeyPair KeyPairArray<>;

enum Op_e {
    OpSet_t = 0x00,
    OpGet_t = 0x01,
    OpUpdate_t = 0x02,
    OpDelete_t = 0x03
};
typedef uint8_t Op;
```

```
struct StringRef {
    Offset_t StringStart;
    Length_t StringLength;
};
```

```
typedef Length_t PacketLength_t;
```

```
enum CMD_t {
    ADMIN_CMD_t  = 0x00,
    AUTH_CMD_t   = 0x01,
    FILE_CMD_t   = 0x02,
    PROTO_CMD_t  = 0x03
};

/* Define a SEQ_t (sequence) type */
typedef uint32_t  SEQ_t;

union CmdPayload_t switch (CMD_t Cmd) {

    case ADMIN_CMD_t:
        AdminCmdPayload AdminPayload;

    case AUTH_CMD_t:
        AuthCmdPayload AuthPayload;


    case FILE_CMD_t:
        FileCmdPayload FilePayload;

    case PROTO_CMD_t:
        ProtoCmdPayload ProtoPayload;
};

struct PacketBody {
    SEQ_t      SEQ;
    CmdPayload_t Payload;
};
```

```
enum AOID_e {
    SERVER_CONFIGURE   = 0x00,
    SERVER_KICK_USER   = 0x01,
    SERVER_LOGS        = 0x02,
    SERVER_MANAGE_BANS = 0x03,
    SERVER_SHUTDOWN    = 0x04,
    SERVER_VIEW_STATS  = 0x05,
    USER_CREATE        = 0x06,
    USER_DELETE        = 0x07,
    USER_LIST          = 0x08,
    USER_PERMISSIONS   = 0x09,
    USER_RENAME        = 0x0A
};
typedef uint8_t AOID;
```

```
typedef uint8_t ConfigOp;

enum ConfigOp_e {
    OpSet_t,
    OpGet_t,
                OpUpdate_t,
                OpDelete_t
};

struct OpSet {
    ConfigOp   Op;     /* With the ConfigOp_e cast to a (ConfigOp)
    KeyValue     OpValues<>;
};

struct ServerConfigure {
    uint8_t   AOID;    /* With the APID_e value cast to a (AOID) */
    OpSet     SetValues<>;
};
```

```
typedef Length_t MDNListCount;

typedef UTC_t MDNSent;

struct MDNRecord {
    MDNSent   TimeSent;

    /* The first item in an XDR array, is its size (MDNListCount) */
    MDNRecord Entries<>;
};
```

```
struct MDMEntry
{
    UTC_t  Received;
    string EMail<>;
};
```

Acknowledgments

Contributors

Thanks to all of the contributors. [REPLACE]

Author's Address

Doug Royer
RiverExplorer LLC
848 N. Rainbow Blvd #1120
Las Vegas, Nevada 89107
United States of America
Phone: 1+714-989-6135
Email: DouglasRoyer@gmail.com
URI: https://DougRoyer.US