Workgroup:	Internet Engineering Task Force
Internet-Draft:	draft-royer-phoenix-00
Published:	17 February 2025
Intended Status:	Informational
Expires:	21 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 21 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:

The UTC value used in this specification is designed to be compatible with time_t on [POSIX] compliant systems. In POSIX systems, time_t is defined as an integer type used for representing time in seconds since the UNIX epoch, which is 00:00:00 UTC on January 1, 1970. And in this specification is 64-bits in size.

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

width = %d1-64

Where: "width" is the number of bits in the value. And must be an unsigned integer greater than zero. And is always expressed in decimal.

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:

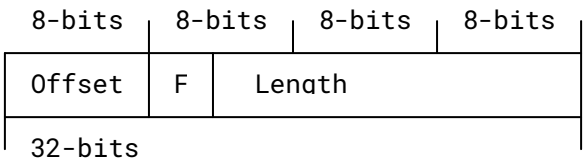


Figure 3: Packed Bit Example

3.3. Common Definitions

TYPE	Notes	XDR API
uint8_t	An 8-bit unsigned integer.	xdr_uint8_t()
uint16_t	A 16-bit unsigned integer.	xdr_uint16_t()
uint32_t	A 32-bit unsigned integer.	xdr_uint32_t()
uint64_t	A 64-bit unsigned integer.	xdr_uint64_t()
string	A string of UTF-8 characters.	xdr_string()
opaque	An array of 8-bit values that will not be XDR encoded or XDR decoded when tranfering the data over this protcol.	xdr_opaque()
Op	An 8-bit value. When the highest bit is one (1) it is a vendor specific Op. Otherwise, it is set to zero (0).	xdr_Op()
OpSet	An 8-bit value. Signifies the operation will set a value. This is used in a few places to signifiy the operation to apply to any included data.	xdr_Op() With the value cat to a (Op).
OpGet	An 8-bit value. Signifies the operation will get a value.	xdr_Op() With the value cat to a (Op).
OpUpdate	An 8-bit value. Signifies the operation will update an existing value.	xdr_Op() With the value cat to a (Op).
OpDelete	An 8-bit value. Signifies the operation will delete key/value pair.	xdr_Op() With the value cat to a (Op).
true	An 8-bit value. A value of true.	xdr_false()
false	An 8-bit value. A value of false.	xdr_true()
enabled	A true or false value.	xdr_enabled()

TYPE	Notes	XDR API
Unused7	A 7-bit value, set to zeros.	
Unused8	An 8-bit value, set to zeros.	
Unused16	A 16-bit value, set to zeros.	
Unused23	A 23-bit value, set to zeros.	
VENDOR_BIT	A 1 bit value, set to 1. It is placed in the highest bit position in the value.	
PHOENIX_BIT	A 1 bit value, set to 0. It is placed in the highest bit position in the value.	

Table 2: Common ABNF/XDR Mapping

3.3.1. Common Definitions - ABNF

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

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

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

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

                ; The number of octets in the the
                ;  associated object.
Length        = uint32_t

                ; 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        = Length *UTF8-Char

                ; Like string, except when the length is
                ; 0xffffffff it means the associated value has
                ; been deleted.
stringD       = (0xffffffff:32) / string

                ; The UTF8-Char is any valid and complete
                ; UTF-8 character.
                ;
UTF8-Char     = 1*4uint8_t

                ; 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        = Length *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 compliant systems.
UTC           = uint64_t

                ; The number of octets from the beginning of the
                ; associated object.
Offset        = uint32_t

                ; Key and Value. It uses stringD as the value
                ; to enable an indicator that the key part
                ; (string part) has been deleted.
KeyPair       = string stringD

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

                ; Setting a value.
OpSet         = %x00:8

                ; Getting a value.
OpGet         = %x01:8

                ; Updating an existing value.
OpUpdate      = %x02:8

                ; Deleting an existing value.
OpDelete      = %x03:8

                ; Any one of the the operations.
Op            = OpSet / OpGet / OpUpdate / OpDelete

true          = %x01:8

false         = %x00:8

enabled       = true   / false

unused8       = %x00:8

unused16      = %x00:16

unused24      = %x00:24

                ; The highest bit when, 1 means it is
                ; a vendor extension.
VENDOR_BIT    = %x01:1

                ; The highest bit when, 0 means it is
                ; a Phoenix command or future Phoenix command.
PHOENIX_BIT   = %x00:1
```

Figure 4: Common ABNF Definitions

3.3.2. Common Definitions - XDR

```
/**
 * The time in seconds since January 1 1970 in GMT.
 */
typedef uint64_t UTC;
/**
 * the number of octets from the beginning of
 * the associated object.
 */
typedef uint32_t Offset;
/**
 * The number of octets in the associated object.
 */
typedef uint32_t Length;
/**
 * When a stringD value is set to this,
 * The the associated value has been deleted.
 */
const Deleted = 0xffffffff;
/**
 * A stringD is a string, or Deleted (the associated
 * data has been deleted).
 */
union stringD switch (uint32_t LengthOrDeleted) {
  case Deleted:
    void;
  default:
    opaque String<>;
};
/**
 * A string Key and its associated Value.
 * It uses stringD as the value to enable an indicator that the
 * key part (string part) has been deleted.
 */
struct KeyPair {
    string Key<>;
    stringD Value<>;
};
/**
 * An array of KeyPair objects.
 */
typedef KeyPair KeyPairArray<>;
/**
 * A 1-bit value;
 * The highest bit in the value, 1 means it is
 * a vendor extension.
 */
const VENDOR_BIT = 0x1;
/**
 * A 1-bit value;
 * The highest bit in the value, 0 means it is
 * a Phoenix command or future Phoenix command.
 */
const PHOENIX_BIT = 0x0;
/**
 * Operations
 */
```

```
enum Op_e {
    OpSet = 0x00,
    OpGet = 0x01,
    OpUpdate = 0x02,
    OpDelete = 0x03
};
/**
 * Any one of OpSet, OpGet, OpUpdate, or OpDelete
 * cast to a (Op).
 */
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. The StringRef does not contain the string, it is a reference an existing string in a MIME object. A StringRef consists of two parts:

Name	Description	XDR API
Offset	The octet count to the start of the string with zero being the first octet in the message.	xdr_Offset()
Length	The length in octets of the string.	xdr_Length()
StringRef	A String reference object.	xdr_StringRef()

Table 3: StringRef ABNF/XDR Mapping

A StringRef over the wire is 8 octets in size.

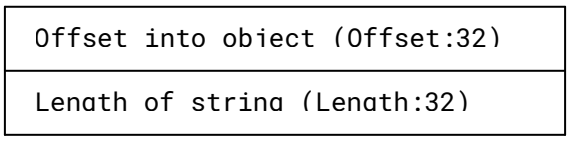


Figure 6: StringRef Format

ABNF:

3.4.1. StringRef ABNF

```
                ; A reference to the start and length of a string.
StringRef     = Offset Length
```

Figure 7: StringRef ABNF

3.4.2. StringRef XDR

The XDR definitions are:

```
/**
 * A reference to the start and length of a string.
 */
struct StringRef {
    Offset StartOffset;
    Length 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.

ADMIN
: 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.

AUTH
: 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 5.10).

  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 a CAPABILITY_PRE list to the server. Or the client sends 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 32-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
PacketHeader	The number of octets that follow this value that are part of this packet.	xdr_PacketHeader()

Table 4: Packet Header ABNF/XDR Mapping

5.1.1.1.1. Packet Header ABNF

ABNF:

```
                  ; The length of a packet.
PacketHeader    = Length
```

Figure 9: Packet Header ABNF

5.1.1.1.2. Packet Header XDR

XDR Definition:

```
/**
 * The length of a packet.
 */
typedef Length PacketHeader;
```

Figure 10: Packet Header XDR

5.1.1.2. Packet Body (PacketBody)

The packet body is divided into three parts:

1. Sequence (SEQ)
2. Command (CMD): Which can be one of ADMIN_CMD, AUTH_CMD, FILE_CMD, or PROTO_CMD.
3. Payload (CmdPayload): The command specific data.

PacketBody details:

Name	Value	Description	XDR API
ADMIN_CMD	%x00:31	The payload is an administration command.	xdr_ADMIN_CMD()
AUTH_CMD	%x01:31	What follows in the payload, is an authentication command.	xdr_AUTH_CMD()
CMD	1 bit + 31 bits.	The 1-bit, VENDOR_BIT or PHOENIX_BIT followed by one of the 31-bit: ADMIN_CMD, AUTH_CMD, FILE_CMD, or PROTO_CMD. For a total of 32-bits. A command (CMD) is a unsigned integer that specifics a unique operation that describes and defines the data that follows. The highest bit in the 32-bit value is a VENDOR_BIT or PHOENIX_BIT. So CMD covers vendor and phoenix commands. -Phoenix CMD range is: %x00000000-7fffffff. -Vendor CMD range is: 80000000-fffffffe. -With %xffffffff reserved for future expansions.	xdr_Cmd()

Name	Value	Description	XDR API
CmdPayload	Variable	The Payload is whatever data follows the command. In some cases it is a blob of opaque data. In other cases it is a structured XDR set of data. See the specific CMD for details.	xdr_CmdPayload()
FILE_CMD	%x03:31	A FILE or FOLDER payload is included.	xdr_FILE_CMD()
PROTO_CMD	%x04:31	The payload includes one of many protocol packets.	xdr_PROTO_CMD()
NOT_SUPPORTED	%x2b:31	The endpoint is replying that the command sent with the SEQ value in this packet, was not a supported command or operation.	xdr_NOT_SUPPORTED()
SEQ	uint32_t	The Command SEQ is a uint32_t. 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()
PacketBody	Variable	The packet body.	xdr_PacketBody()

Table 5: Packet Body ABNF/XDR Mapping

A command conforming to this specification is not a vendor command. A command created by any vendor that implements vendor specific commands or operations is a vendor command. Vendor commands have the VENDOR_BIT set in the commands or operations.

If any operation in a command has the VENDOR_BIT set it may effect implementations that do not support that specific vendor operation. So caution must be used when creating vendor command operation extensions. If the vendor operation can not be performed by client conforming to this specification, then the command must also have the VENDOR_BIT set.

In this example the server is sending a CAPABILITY_PRE command telling the client that the server supports AUTHMD5 and some made up vendor authentication AUTH_Vendor_3. The AUTHMD5 does not have the VENDOR_BIT set, and AUTH_Vendor_e has the VENDOR_BIT set.

Clients or servers that are not using vendor specific extensions, can:

- Send a VENDOR_ID with the value as an empty string.
- Or set the string, and just do not send any command or command operations with extensions.

Clients that do not understand the string value in VENDOR_ID would ignore the commands and capabilities with the VENDOR_BIT set. Which is AUTH_Vendor_e in this example.

This is an example of a CAPABILITY_PRE being sent from the server to the client. A conforming client could ignore the AUTH_Vendor_3 and authenticate with AUTHMD5.

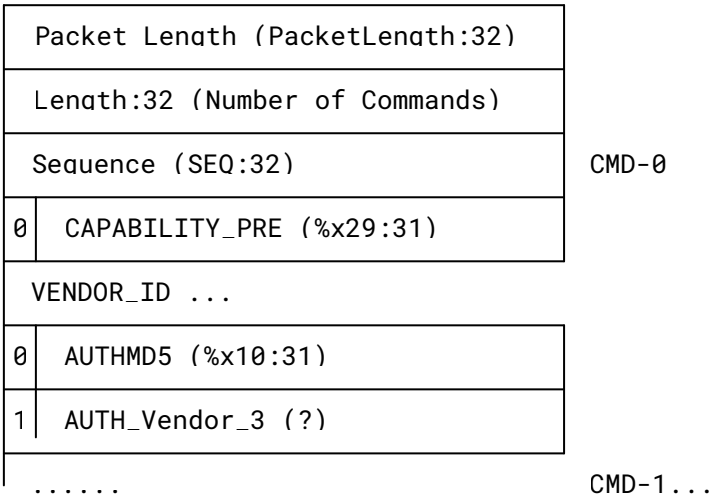


Figure 11: Packet Body Non-Vendor- Diagram

In this example, the server is telling the client that it is using a vendor specific AUTHMD5 and a vendor specific AUTH_Vendor_e only.

When when vendor specific extensions make the connection incompatible with implementations conforming to this specification, then it MUST also set the VENDOR_BIT in the command. In this example it is being set in the CAPABILITY_PRE command. A conforming client would then know that there are zero compatible authentication methods to this server. A client implementations that understand the contents of the string value for VENDOR_ID, may also understand these extensions.

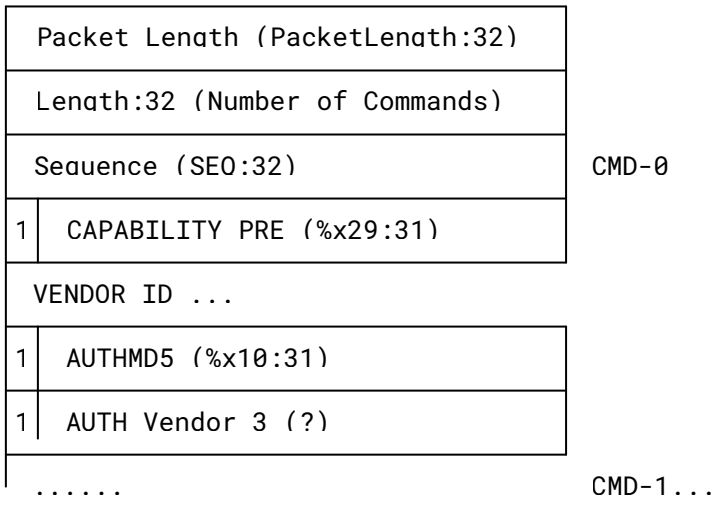


Figure 12: Packet Body Vendor - Diagram

Multiple commands may be sent in one packet. And PHOENIX and non-vendor (NON-PHOENIX) commands may be sent in one packet body by setting VENDOR_BIT (1) or PHOENIX_BIT (0) in the packet. This example shows two commands, one that is a PHOENIX command the other is a vendor command.

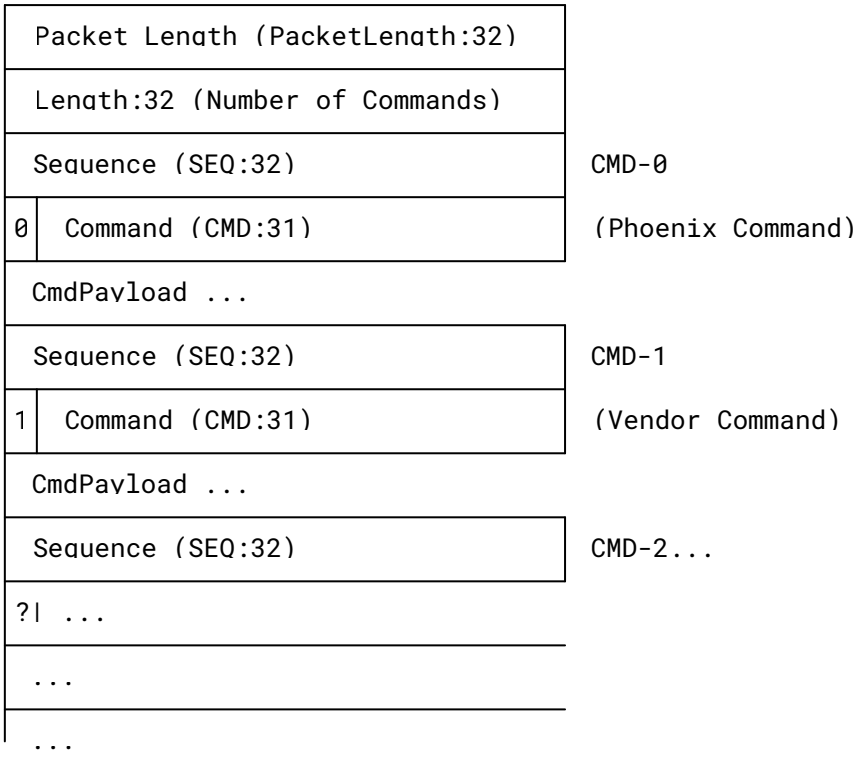


Figure 13: Packet Body Multiple Commands - Diagram

5.1.1.2.1. Packet Body ABNF

ABNF:

```
              ; The data that follows is ADMIN CMD data.
ADMIN_CMD  = %x00:31

              ; The data that follows is AUTH CMD data.
AUTH_CMD   = %x01:31

              ; The data that follows is FILE or FOLDER CMD data.
FILE_CMD   = %x02:31

              ; The data that follows is Phoenix PROTO CMD
              ; specific command data.
PROTO_CMD  = %x03:31

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

              ; Any command. A 32-bit value.
CMD        = (VENDOR_BIT / PHOENIX_BIT)
              (ADMIN_CMD / AUTH_CMD / FILE_CMD / PROTO_CMD)

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

              ; A Phoenix packet
CommandSet = SEQ CMD
              (AdminPayload
              / AdminReplyPayload
              / AuthPayload
              / AuthReplyPayload
              / FilePayload
              / FileReplyPayload
              / ProtoPayload
              / ProtoReplyPayload)

              ; Length: The number of CommandSet objects.
PacketBody = Length 1*CommandSet;
```

Figure 14: Packet Body ABNF

5.1.1.2.2. Packet Body XDR

XDR Definition:

```
/*
 * The ADMIN commands.
 * An XDR enum is 32-bits in size.
 * With the VENDOR_BIT set to zero, making it not a vendor command.
 */
enum CMD {
    /**
     * The data that follows is ADMIN CMD data.
     */
    ADMIN_CMD = 0x00,
    /**
     * The data that follows is AUTH CMD data.
     */
    AUTH_CMD = 0x01,
    /**
     * The data that follows is FILE or FOLDER CMD data.
     */
    FILE_CMD = 0x02,
    /**
     * The data that follows is Phoenix PROTO CMD data.
     */
    PROTO_CMD = 0x03
};
/**
 * Mask to check if CMD value in packet
 * is vendor extension.
 */
const CMD_VENDOR_MASK = 0x80000000;
/**
 * Define a SEQ (sequence) type.
 */
typedef uint32_t SEQ_t;
/**
 * A CMD payload is one of these types.
 * With Cmd set to a CMD value.
 */
union CmdPayload switch (CMD Cmd) {
  case ADMIN_CMD:
    AdminPayload AdminBodyPayload;
  case AUTH_CMD:
    AuthPayload AuthBodyPayload;
  case FILE_CMD:
    FilePayload FileBodyPayload;
  case PROTO_CMD:
    ProtoPayload ProtoBodyPayload;
};
/*
 * One comand.
 */
struct CommandSet {
    SEQ_t SEQ;
    CmdPayload Payload;
};
/**
 * A packet body.
 */
```

```
struct PacketBody {
  CommandSet Commands<>; /* XDR arrays start with the Length */
};
```

Figure 15: Packet Body XDR

5.1.1.2.3. Multiple Commands per Packet

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.

And like the original command, multiple replies may be in one packet.

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 a CAPABILITY_POST with an ADMIN capability in it, 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 any ADMIN capability in the CAPABILITY_PRE list.

Implementations that support any administration command will include an ADMIN capability in the CAPABILITY_POST list. An implementation may decide that only specified and authorized users may issue administrative commands and send only those authenticated users an 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 NOT_SUPPORTED command with its sequence number set to the sequence number from offending command.

5.2.2. Administration Command Payload

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, called an ADMIN Operation ID (AOID).

All of their XDR API is: xdr_AOID().

Command and Capability Name	AOID	Capability Description.	Command Description.
AOID	uint32_t	An Administrative Operation Identifier.	Holds the (VENDOR_BIT or PHOENIX_BIT) value and one of SERVER_CONFIGURE, SERVER_KICK_USER, SERVER_LOGS, SERVER_MANAGE_BANS, SERVER_SHUTDOWN, SERVER_VIEW_STATA, USER_CREATE, USER_CREATE, USER_DELETE, USER_LIST, USER_PERMISSIONS, USER_RENAME
SERVER_CONFIGURE	%x05:8	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.
SERVER_KICK_USER	%x06:8	May logout a user. And limit when they can use the server again.	The command to kick and limit a user.
SERVER_LOGS	%x07:8	May view the server logs.	The command to view server logs.

Command and Capability Name	AOID	Capability Description.	Command Description.
SERVER_MANAGE_BANS	%x08:8	May manage IP and user bans.	The command to manage ban users and IP addresses.
SERVER_SHUTDOWN	%x09:8	May shutdown the server.	The command to shutdown the server.
SERVER_VIEW_STATS	%x0a:8	May view server statistics.	The command to view statistics.
USER_CREATE	%x0b:8	May create a new user.	The command to create a Phoenix server user.
USER_DELETE	%x0c:8	May delete a user.	The command to delete a user.
USER_LIST	%x0d:8	May list users and their capabilities.	The command to list users.
USER_PERMISSIONS	%x0e:8	May update other users permissions.	The command to view and set user permissions.
USER_RENAME	%x0f:8	May rename a user.	The command to rename a user.

Table 6: Administration Comamnd Payload Operations

5.2.2.1. Administration Command Payload - Operations - ABNF

```
                      ; What follows is CONFIGURE operation.
SERVER_CONFIGURE    = %x00:8

                      ; What follows is a KICK USER operation.
SERVER_KICK_USER    = %x01:8

                      ; What follows is a LOGS operation.
SERVER_LOGS         = %x02:8

                      ; What follows is a BAN USER operation.
SERVER_MANAGE_BANS = %x03:8

                      ; What follows is a SERVER SHUTDOWN operation.
SERVER_SHUTDOWN     = %x04:8

                      ; What follows is a VIEW STATS operation.
SERVER_VIEW_STATS   = %x05:8

                      ; A USER CREATE (or enable) operation follows.
USER_CREATE         = %x06:8

                      ; A USER DELETE (or disable) operation follows.
USER_DELETE         = %x07:8

                      ; Get one or more USER info.
USER_LIST           = %x08:8

                      ; Give or take away USER permissions.
USER_PERMISSIONS    = %x09:8

                      ; Alter a users login name.
USER_RENAME         = %x0A:8

                      ; All ADMIN commands are 32-bit values.
AOID                = (VENDOR_BIT / PHOENIX_BIT) Unused23
                      (SERVER_CONFIGURE
                       / SERVER_KICK_USER
                       / SERVER_LOGS
                       / SERVER_MANAGE_BANS
                       / SERVER_SHUTDOWN
                       / SERVER_VIEW_STATS
                       / USER_CREATE
                       / USER_DELETE
                       / USER_LIST
                       / USER_PERMISSIONS
                       / USER_RENAME)

                      ; Where CmdPayload contents depend on the
                      ; the value of AOID.
AdminPayload        = AOID
                      (ServerConfigPayload
                      / ServerKickPayload
                      / ServerLogsPayload
                      / ServerBansPayload
                      / ServerShutdownPayload
                      / ServerStatsPayload
```

```
                      / UserCreatePayload
                      / UserDeletePayload
                      / UserListPayload
                      / UserPermissionsPayload
                      / UserRenamePayload)

                      ; Where CmdPayload contents depend on the
                      ; the value of AOID.
AdminReplyPayload  = AOID
                      (ServerConfigReplyPayload
                      / ServerKickReplyPayload
                      / ServerLogsReplyPayload
                      / ServerBansReplyPayload
                      / ServerShutdownReplyPayload
                      / ServerStatsReplyPayload
                      / UserCreateReplyPayload
                      / UserDeleteReplyPayload
                      / UserListReplyPayload
                      / UserPermissionsReplyPayload
                      / UserRenameReplyPayload)
```

Figure 16: Administration Command Payload - Operations - ABNF

5.2.2.2. Administration Command Payload - Operations - XDR

```
/**
 * ADMIN commands.
 * In XDR, enum values are 32-bit.
 * The high bit is zero, which means the PHOENIX_BIT is zero
 * in these values.
 */
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
};
/**
 * All ADMIN commands are sent as 32-bit values.
 */
typedef AOID_e AOID;
```

```
union AdminPayload switch(AOID_e Aoid) {
  case SERVER_CONFIGURE:
    ServerConfigPayload ConfigPayload;
  case SERVER_KICK_USER:
    ServerKickPayload KickPayload;
 case SERVER_LOGS:
    ServerLogsPayload LogsPayload;
  case SERVER_MANAGE_BANS:
    ServerBansPayload BansPayload;
  case SERVER_SHUTDOWN:
    ServerShutdownPayload ShutdownPayload;
  case SERVER_VIEW_STATS:
    ServerStatsPayload StatsPayload;
  case USER_CREATE:
    UserCreatePayload UCreatePayload;
  case USER_DELETE:
    UserDeletePayload UDeletePayload;
  case USER_LIST:
    UserListPayload UListPayload;
  case USER_PERMISSIONS:
    UserPermissionsPayload UPermissionsPayload;
  case USER_RENAME:
    UserRenamePayload URenamePayload;
};
union AdminReplyPayload switch(AOID_e Aoid) {
  case SERVER_CONFIGURE:
    ServerConfigReplyPayload ConfigPayload;
  case SERVER_KICK_USER:
  void; /* No reply allowd for SERVER_KICK_USER */
 case SERVER_LOGS:
    ServerLogsReplyPayload LogsPayload;
  case SERVER_MANAGE_BANS:
    ServerBansReplyPayload BansPayload;
  case SERVER_SHUTDOWN:
    ServerShutdownReplyPayload ShutdownPayload;
  case SERVER_VIEW_STATS:
    ServerStatsReplyPayload StatsPayload;
  case USER_CREATE:
    UserCreateReplyPayload UCreatePayload;
  case USER_DELETE:
    UserDeleteReplyPayload UDeletePayload;
  case USER_LIST:
    UserListReplyPayload UListPayload;
  case USER_PERMISSIONS:
    UserPermissionsReplyPayload UPermissionsPayload;
  case USER_RENAME:
    UserRenameReplyPayload URenamePayload;
};
```

Figure 17: Administration Command Payload - Operations - XDR

5.2.3. Administration - SERVER_CONFIGURE (%x05:8)

SERVER_CONFIGURE is optional and what can be configured is unique to specific implementations. Phoenix provides a way to transport key + value pairs to and from the server as a way to configure remote Phoenix servers.

A Phoenix server provides the SERVER_CONFIGURE in the CAPABILITY_POST list it provides to the client. This being sent, and the VENDOR_ID sent from the client allows the endpoints to determine if the client implementation is compatible to the servers implementation.

The keys and their values can be anything the server implementation supports. The content of configuration keys and configuration values is out of scope for this specification.

5.2.3.1. SERVER_CONFIGURE - Command

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.

A client implementation would generally only support its own matching server implementation. The client sends a VENDOR_ID capability with a string that the server implementation will check to see if it is a compatible client. The value is out of scope to this specification. If the client does not send the correct VENDOR_ID information, or the authentication user is not authorized, then the server would not send its SERVER_CONFIGURE capability back to the client in the CAPABILITY_POST list.

If a client or user is not authorized, then all SERVER_CONFIGURE ADMIN command will be rejected with a NotSupported Packet with the SEQ number the same as in the SERVER_CONFIGURE ADMIN command sent to the server.

The SERVER_CONFIGURE ADMIN command sends and receives the configuration information in key + value pairs. The key and value are each a string. The values used in the key or value are vendor specific and out of scope in this specification.

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. They are processed in the order sent.

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

Table 7: SERVER_CONFIGURE - ABNF/XDR Mapping

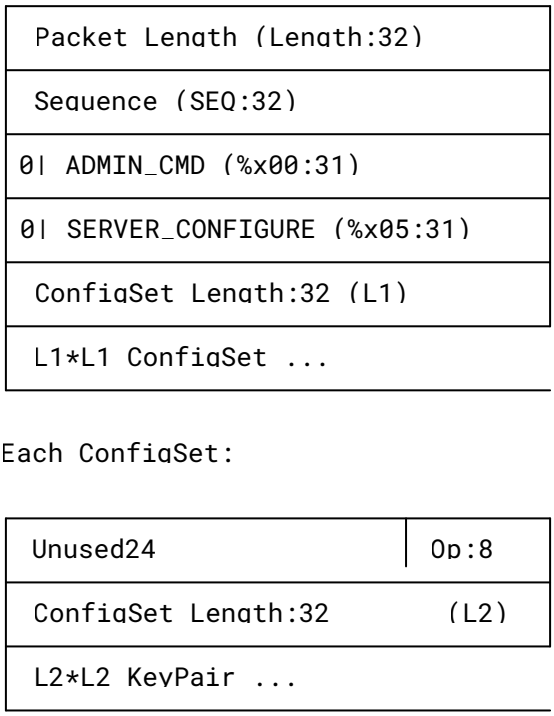


Figure 18: SERVER_CONFIGURE Command

5.2.3.1.1. SERVER_CONFIGURE - ABNF

```
                        ; Setting or updating a configuration value.
ConfigSet            = (Unused24:24 Op:8):32 Length 1*KeyPair

                        ; The SERVER_CONFIGURE command.
ServerConfigPayload  = SERVER_CONFIGURE Length 1*ConfigSet
```

Figure 19: SERVER_CONFIGURE - ABNF

5.2.3.1.2. SERVER_CONFIGURE - XDR

```
/**
 * An array of OpConfigSet values.
 */
struct ServerConfigPayload {
    AOID_e Aoid; /* Set to SERVER_CONFIGURE. */
    ConfigSet Values<>; /* XDR arrays start with a length. */
};
```

Figure 20: SERVER_CONFIGURE - XDR

5.2.3.2. SERVER_CONFIGURE - Reply

All SERVER_CONFIGURE operations result in a reply. It will include any OpGet as well as the new value for any OpSet or OpUpdate. And for OpDelete a reply will show the value has been deleted.

The reason for always returning the values is because sometimes not all configuration information can be altered. So the client will need to compare the desired outcome with the results.

The reply is a SERVER_CONFIGURE command with the same SEQ number that was sent. Followed by a list of KeyPair values. One KeyPair value for each unique key in the ConfigSet values in the SERVER_CONFIGURE command sent to the server. The list is unordered. And the list is the result of processing all ConfigSet values sent.

On any error the server can perform some or none of the operations. The specifics are implementation dependent and out of scope for this specification.

Empty Keys have a value length set to zero. These keys exist, and have empty content and have zero octets for the value.

Keys that have been deleted, have a value with the value length set to 0xffffffff:32 and and have zero octets for the value.

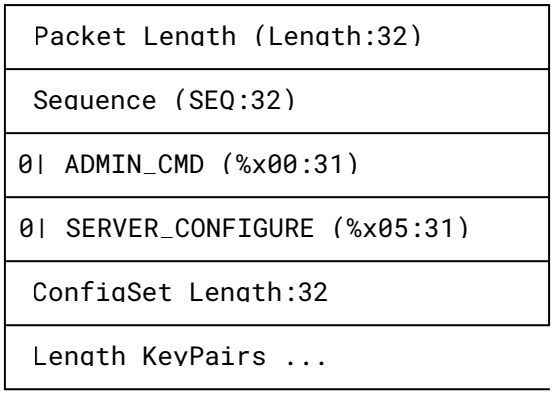


Figure 21: SERVER_CONFIGURE Reply

5.2.3.2.1. SERVER_CONFIGURE Reply - ABNF

```
                    ; The SERVER_CONFIGURE Reply
ServerConfigReplyPayload  = SERVER_CONFIGURE Length 1*KeyPair
```

Figure 22: SERVER_CONFIGURE - ABNF

5.2.3.2.2. SERVER_CONFIGURE Reply - XDR

```
/**
 * Reply to SERVER_CONFIGURE
 */
struct ServerConfigReplyPayload {
    AOID_e ACmd; /* Set to SERVER_CONFIGURE */
    KeyPair ResultValues<>; /* XDR arrays start with a length */
};
```

Figure 23: SERVER_CONFIGURE - XDR

5.2.4. 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 or may be able to login again.

No specifics are given because the possible reasons are nearly unlimited and locale dependent. Any notifications the server may elect to provide is out of scope in this specification. The user may have to contact the service provider for details.

A SERVER_KICK_USER is only sent from the server to the client. If a server gets a SERVER_KICK_USER from a client, the server replies with a NotSupported with the sequence number the same as the invalid command.

There is no reply to a SERVER_KICK_USER. The client is no longer authenticated. The server may allow the client to re-authenticate, or the server may terminate the connection.

If the server allows the client to re-authenticate, the server next sends a CAPABILITY_PRE to the client

If the server is terminating the connection, the server then sends a SERVER_SHUTDOWN to the client.

The server may elect to send both packet sets in one packet to the server.

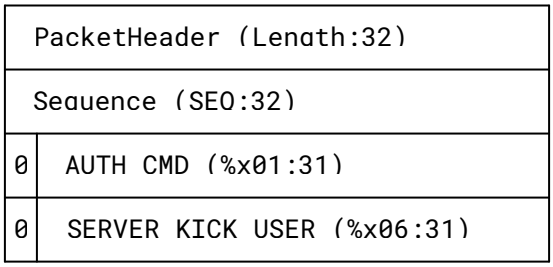


Figure 24: SERVER_KICK_USER

5.2.4.1. SERVER_KICK_USER - ABNF

```
ServerKickPayload  = (VENDOR_BIT / PHOENIX_BIT) SERVER_KICK_PAYLOAD
```

Figure 25: SERVER_KICK_USER - ABNF

5.2.4.2. SERVER_KICK_USER - XDR

```
struct ServerKickPayload
{
 AOID_e Kick; /* With Kick set to SERVER_KICK_USER */
};
```

Figure 26: SERVER_KICK_USER - XDR

5.2.5. 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 is sent with two time range values. The time values are a UTC object. The first is the start time, and the second is the end time.

When the start time is zero, it means start from the beginning of the logs.

When the end time is zero, up to now.

There are three categories of information and four request types.

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.
ALL	Only used in the request signifying all Error, Warning, and Information logs are to be sent.

Table 8

Name	Description	XDR API
Category	Category	xdr_Category

Table 9

5.2.5.1. XXXX - ABNF

```
xxx
```

Figure 27: xxxx

5.2.5.2. XXXX - XDR

```
xxx
```

Figure 28: xxxx

5.2.6. Administration - SERVER_MANANGE_BANS

Perform an Op on users, hosts, IP addresses, IP address ranges, networks.

TODO...

5.2.7. Administration - SERVER_SHUTDOWN

5.2.8. Administration - SERVER_VIEW_STATS

This ADMIN command allows authorized and authenticated users to view the system logs.

TODO ...

5.2.9. Administration - USER_CREATE

Create a new user.

TODO ...

5.2.10. Administration - USER_DELETE

TODO ...

5.2.11. Administration - USER_LIST

TODO ...

5.2.12. Administration - USER_PERMISSIONS

TODO ...

5.2.13. Administration - USER_RENAME

TODO ...

5.3. Authentication Commands Summary

The first thing a client must do, is authenticate with the server.

Some users may also be administrators. In those cases the user and client may wish to do further authentication steps. A user may wish to temporarily step up their authentication level to perform some operations, then step back down to do their personal operations.

A server may if it wishes include AUTH capabilities in the CAPABILITY_POST command. It can decide which authenticated users can or must use additional authentication.

- Some user accounts can be user only, without administrative abilities of any kind. Their CAPABILITY_POST list will not include any ADMIN capabilities.
- Some user accounts can be administrative only, and are limited to CAPABILITY_POST actions that only include ADMIN capabilities.
- Some user accounts can be a normal user, with the ability to step up their account to be able to do administrative actions. While in stepped up mode, they are not able to act as their original user account. These users will get one or more AUTH capabilities in their CAPABILITY_POST list.

- And some user account can be allowed all operations. This protocol does not limit users. This protocol enables user permissions to be configured.

5.3.1. Authentication - ABNF

```
AUTHANONYMOUS    = %x26
AUTHMD5          = %x10
AUTHCERT         = %x11
AUTHCERT_TLS     = %x27
AUTHCERT_USER    = %x28

AuthPayload      = (AuthAnonymousPayload
                 / AuthMD5Payload
                 / AuthCertPayload
                 / AuthCertTlsPayload
                 / AuthCertUserPayload)

; There is no AUTH... REPLY payload.
; An authentication replies with:
;
;   CAPABILITY_POST: Authentication passed.
;
;   CAPABILITY_PRE:  Authentication failed.
;
;   NotSupported:    Authentication method not supported.
```

Figure 29: Authentication - ABNF

5.3.2. Authentication - XDR

```
enum Auth_e {
    AUTHANONYMOUS = 0x26,
    AUTHMD5 = 0x10,
    AUTHCERT = 0x11,
    AUTHCERT_TLS = 0x27,
    AUTHCERT_USER = 0x28
};
```

```
union AuthPayload switch(Auth_e Auth) {
  case AUTHANONYMOUS:
    AuthAnonymousPayload AnonymousPayload;
  case AUTHMD5:
    AuthMD5Payload MD5Payload;
 case AUTHCERT:
    AuthCertPayload CertPayload;
  case AUTHCERT_TLS:
    AuthCertTlsPayload CertTlsPayload;
  case AUTHCERT_USER:
    AuthCertUserPayload CertUserPayload;
};
/**
 * There is no AUTH... REPLY payload.
 * An authentication replies with:
 *
 * CAPABILITY_POST: Authentication passed.
 *
 * CAPABILITY_PRE: Authentication failed.
 *
 * NotSupported:   Authentication method not supported.
 */
```

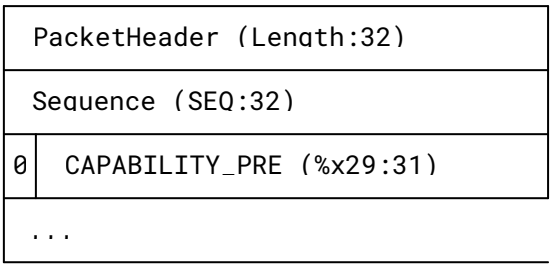
Figure 30: Authentication - XDR

5.3.3. Authentication Failure

If the authentication fails, then the server replies back with a CAPABILITY_PRE with the sequence number the same as in the authentication request.

Or The server replies with a NOT_SUPPORTED when the authentication method is not supported, and with the sequence number the same as in the authentication request.

Failed authentication:



Failed authentication, method not supported:

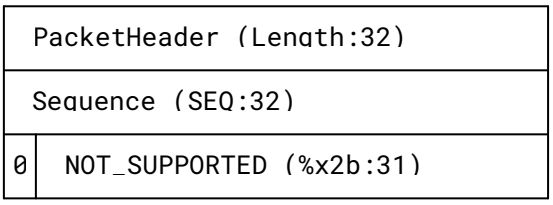


Figure 31: AUTHCERT_USER

5.4. Authentication - ANONYMOUS

ANONYMOUS is both a capability and a command.

5.4.1. Authentication - AUTHANONYMOUS - Capability

When sent as a capability, a true or false value follows. When true, it means that anonymous login is supported. When false, it means that anonymous login is not supported.

The highest bit is set to zero (0) which indicates this is a Phoenix defined capability, and not a vendor created and known capability. Followed by the 31-bit capability value.

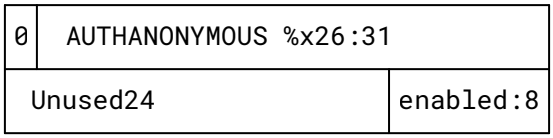


Figure 32: Capability - AUTHANONYMOUS

5.4.2. Authentication - AUTHANONYMOUS - Command

Once the connection is made the server sends its CAPABILITY_PRE list to the client. If AUTHANONYMOUS is included in that list, then the client may initiate an AUTHANONYMOUS login.

If the client has already had a relationship with the server, then the client may send the AUTHANONYMOUS command to the server before receiving the servers capability list, and only if AUTHANONYMOUS had been successful in the past, to that same server.

If the server does not support (or no longer supports) an AUTHANONYMOUS command, it will reply with a NotSupported packet with the sequence number the same as in the AUTHANONYMOUS login request.

After the server receives an AUTHANONYMOUS, and if it supports it, it allows the connection and considers the user a valid anonymous user. Then the server replies with a CAPABILITY_POST command. When the client gets the CAPABILITY_POST command, it knows the AUTHANONYMOUS was successful.

On failure the server replies with a AUTHANONYMOUS packet, with the sequence number the same that was in the ANONYMOUS command it received.

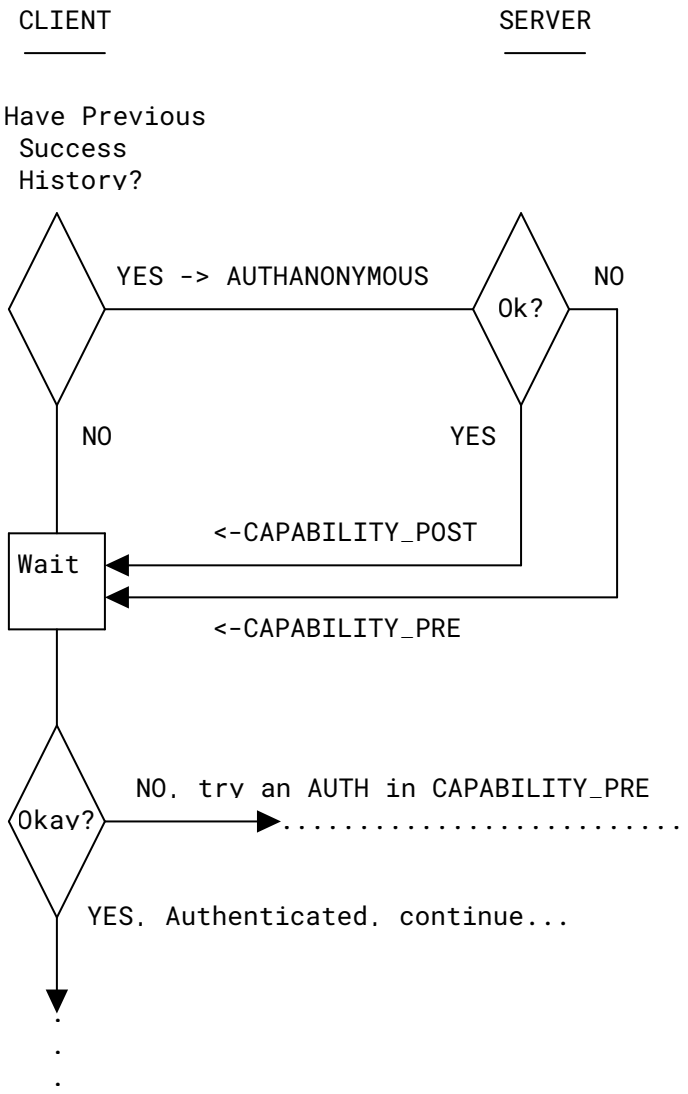


Figure 33: AUTHANONYMOUS - Login Flow

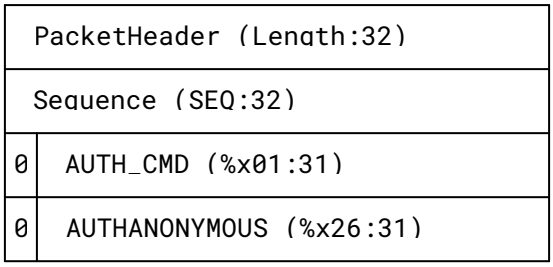


Figure 34: Capability - AUTHANONYMOUS

5.4.2.1. Authentication - ANONYMOUS - ABNF

```
AuthAnonymous  = (VENDOR_BIT / PHOENIX_BIT) AUTHANONYMOUS
```

Figure 35: Authentication - ANONYMOUS - ABNF

5.4.2.2. Authentication - ANONYMOUS - XDR

```
struct AuthAnonymousPayload
{
 AOID_e Anonymous; /* Set to AUTHANONYMOUS (%x26) */
};
```

Figure 36: Authentication - ANONYMOUS - XDR

5.5. Authentication - Certificate

There are two kinds of AUTHCERT.

- Authentication by TLS certificate at connection time. This is called an AUTHCERT_TLS.
- Authentication challenge and response after connection time. This is called an AUTHCERT_USER.

5.5.1. Authentication - Certificate - Capability

When the server sends the AUTHCERT capability to the client, it is followed by two "enabled" values. One for AUTHCERT_TLS and the other for AUTHCERT_USER.

5.5.2. AUTHCERT_TLS

When the client connects to the server, it uses a pre authorized digital certificate for the TLS connection.

The certificate itself could be sufficient. Or the server may look into the contents of the client public certificate supplied at TLS connection time for information to help it determine the level of trust, including none.

The server could be configured to accept self-signed certificates, or it may be configured to verify a certificate chain to a root certificate it trusts. Or some combination.

A server could be configured to only allow AUTHCERT_TLS from a subset of IP addresses or networks.

When the client successfully authenticates using AUTHCERT_TLS, then the server replies with a CAPABILITY_POST command to the client. And no CAPABILITY_PRE command is sent by the server.

When a client fails the AUTHCERT_TLS, then the server sends a CAPABILITY_PRE command to the client. The client can then proceed with other authentication methods that were provided in the capability list supplied by the server.

When a client gets a CAPABILITY_POST command from the server after connection, without having sent any authentication commands, the the client knows it has been authenticated with AUTHCERT_TLS.

Clients expecting an AUTHCERT_TLS must wait for the CAPABILITY_POST or CAPABILITY_PRE command before continuing with client operations with associated folders and files.

When a client that was not expecting an AUTHCERT_TLS gets a CAPABILITY_POST after connection and did not get a CAPABILITY_PRE, then the client know they are authenticated using the supplied TLS certificate.

5.5.3. AUTHCERT_USER

This authentication method still requires a valid TLS connection certificate, as it does with all connections. It also requires that the client send a public certificate to the server as a separate authentication step for the user.

This type of login could be used when traveling or the server requires more control over security. The users certificates could be under the control of the users company, and easier to create and revoke than traditional certificate sources.

In order for AUTHCERT_USER to work, the server MUST already have the users public certificate. This could have been setup by a servers implementation configuration files, or from a previous successful non-AUTHCERT_USER connection where the client informed the server of the users public certificate.

To authenticate with a AUTHCERT_USER, the client sends a AUTHCERT_USER command with a clear text token over the TLS connection, followed by a the both the secret login name and password encrypted with the users private certificate.

The token could be one time, or reusable. The server implementation is the authority on the token and token usage. It could be possible that the user never knows the actual login and password. They could be installed on a device for the user. They, perhaps, go to a web page, or other method to get the token. Then perhaps enter their personal password to allow access to the certificates and secret login information that is installed and already encrypted on the client.

The server decrypts the users login name and password with the users public certificate selected from the clear text token sent.

If the decrypted user login and password match what is expected, then the authentication is successful and the server replies with a CAPABILITY_POST command.

5.5.4. Certificate Management

A Phoenix server may use AUTHCERT_USER authentication. When it does, it needs a way for the user, if authorized to upload their public certificate to the server. This can be enabled or disabled by server configuration on the site, per user, or any other rules implemented in the server.

User certificate management can only be used after the user has authenticated with the server.

5.6. Authentication - MD5

AUTHMD5 is both a capability and a command.

5.6.1. Authentication - MD5 - Capability

When sent as a capability, a true or false value follows. When true, it means that AUTHMD5 is supported. When false, it means that AUTHMD5 is not supported.

The highest bit is set to zero (0) which indicates this is a Phoenix defined capability, and not a vendor created and known capability. Followed by the 31-bit capability value.

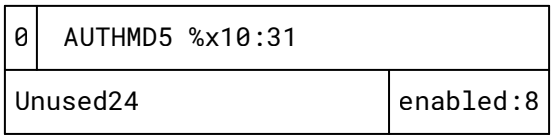


Figure 37: Capability - AUTHMD5

5.6.2. Authentication - MD5 - Command

Once the connection is made the server sends its pre authentication capability list to the client. If AUTHMD5 is included in that list, then the client may initiate an AUTHMD5 login.

If the client has already had a relationship with the server, then the client may send the AUTHMD5 command to the server before receiving the servers capability list, and only if AUTHMD5 had been successful in the past, to that server.

If the server does not support (or no longer supports) an MD5 command, it will reply with a NotSupported packet with the sequence number the same as in the MD5 login request.

After the server receives an AUTHMD5, and if it supports it, then it attempts to verify the provided information. One of two replies are possible, success, or failure.

On failure the server replies with a AUTHMD5 packet, with the sequence number the same that was in the AUTHMD5 command it received. With the login and password fields empty and their lengths set to zero.

On success the server replies with a post authentication capability command.

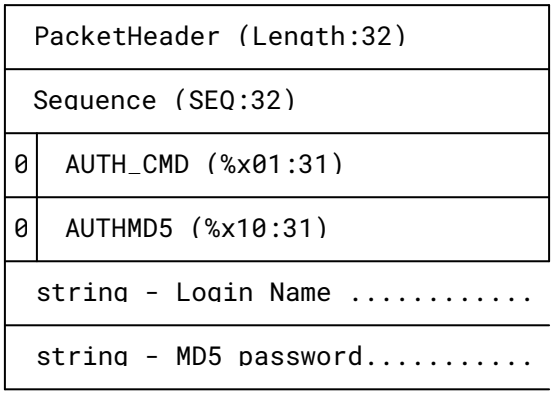


Figure 38: Capability - AUTHMD5

5.6.2.1. Authentication - MD5 - ABNF

```
Login       = string

Md5Password = string

AuthMD5     = (VENDOR_BIT / PHOENIX_BIT) AUTHMD5 Login Md5Password
```

Figure 39: Authentication - MD5 - ABNF

5.6.2.2. Authentication - MD5 - XDR

```
struct AuthMD5 {
    uint32_t Cmd; /* Set to AUTHMD5 */
    string Login<>;
    string Md5Password<>;
};
```

Figure 40: Authentication - MD5 - XDR

5.7. Calendar Commands Summary

These command are based on iCalendar and iTIp.

5.8. Capability Commands Summary

Capabilities are attributes of both a client and server implementation. Some may provide a superset or subset when compared to other implementations. This can be done to split workload or just because they specialize in specific operations.

A capability is a 31-bit integer. Plus a 1-bit identifier signifying if it is a Phoenix capability or vendor specific capability, for a total of 32-bits.

This specification describes several capabilities. Some are described in other sections, and some are described in this section. See appendix XREF-TODO for a complete list in this specification.

Capabilities from the server arrive twice. Once before the user is authenticated (CAPABILITY_PRE), and once after (CAPABILITY_POST).

Capabilities from the client may be sent, zero, once, or twice per authentication process. Optionally once included with the AUTH command packet (CAPABILITY_PRE), to inform the server of any required vendor or authentication specific information. And optionally once after the user is authenticated (CAPABILITY_POST).

There is no requirement that a server provides an authentication method for any client. There is no requirement that a server provide any non-vendor capability for any client. They could be configured to only allow vendor specific authentication or only vendor specific commands because they are not servers open to the public and require non standard authentication methods, or only from clients providing a correct CAPABILITY_PRE value.

Table Table 10 lists the CAPABILITY_PRE capabilities.

Name	Value	Data Type	Required or Optional	Description
AUTHANONYMOUS	%x26:31	Boolean	Optional	No authentication required. An example usage could be a shared company bulletin board where most employees had view only access to the messages. And perhaps the server only allowed company local IP addresses to use this authentication method.
AUTHMD5	%x10:31	Boolean	Required	Authenticate by providing an MD5 password. If not supported, the value must be false. When supported, the value must be true.
AUTHCERT	%x11:31	Boolean	Optional	Authenticate by providing a certificate. This can be used for humans, and can be used by automated connections. Not being provided is the same as false. May have a true or false value as needed.
VENDOR_ID	%x12:31	string	Optional	VENDOR_ID includes a vendor ID string that can be used to help the server determine if it will send a post-authentication ADMIN capability. The specific string value is determined by the server implementation and is out of scope for this specification.

Table 10: Capabilities - CAPABILITY_PRE

Table Table 11 lists the CAPABILITY_POST capabilities. Post-Authentication capabilities are sent to the client after a user authenticates.

Name	Value	Description
x	x1	x2

Table 11: Capabilities - CAPABILITY_POST

5.8.1. Capability - VENDOR_ID

When sent, the VENDOR_ID capability is accompanied by a string. This string is unique and defined by the server implementation or instance.

When a server gets a VENDOR_ID capability, it compares it to what it expects. When they match, then after the user is authenticated the server can then determine if it will send the SERVER_CONFIGURE capability to the client.

It would be expected that any client sending its VENDOR_ID capability to the server is expecting the possibility of receiving a SERVER_CONFIGURE capability back.

The purpose of the VENDOR_ID capability and its value is to help ensure that any SERVER_CONFIGURE commands are compatible between the client and server.

5.9. EMail Commands Summary

These commands allow for the fetching and submission of EMail messages

5.10. 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	Valule	Brief Description.
FOLDER_CAPABILITY	%x13:32	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	%x14:32	Create a new folder. Also the name of the capability for this permission.
FOLDER_COPY	%x15:32	Copy a folder. Also the name of the capability for this permission.
FOLDER_DELETE	%x16:32	Delete a folder. Also the name of the capability for this permission.
FOLDER_RENAME	%x17:32	Rename a folder. Also the name of the capability for this permission.
FOLDER_METADATA	%x18:32	Get, set, and update information associated with the folder. File meta data is also returned with the FOLDER_OPEN command.

Op Name	Valule	Brief Description.
FOLDER_MOVE	%x19:32	Move a folder. Also the name of the capability for this permission.
FOLDER_OPEN	%x1a:32	Open a folder and get information about the folder and files in the folder.
FOLDER_SHARE	%x1b:32	Share a folder. Also the name of the capability for this permission.
FOLDER_LIST	%x1c:32	List folders and files. Also the name of the capability for this permission.
FILE_CREATE	%x1d:32	Create a new file. Also the name of the capability for this permission.
FILE_COPY	%x1e:32	Copy a file. Also the name of the capability for this permission.
FILE_DELETE	%x1f:32	Delete a file. Also the name of the capability for this permission.
FILE_RENAME	%x20:32	Rename a file. Also the name of the capability for this permission.
FILE_METADATA	%x21:32	Get, set, and update information associated with the folder. File meta data is also returned with the FOLDER_OPEN command.
FILE_MOVE	%x22:32	Move a file. Also the name of the capability for this permission.
FILE_SHARE	%x23:32	Share a file. Also the name of the capability for this permission.
FILE_GET	%x24:32	Get a file. Also the name of the capability for this permission.
FILE_MODIFY	%x25:32	Modify the contents of an existing file. Also the name of the capability for this permission.

Table 12: File and Folder Command List

5.11. 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.

5.12. 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.

5.13. 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 (%x00) or true (%x01). 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 13

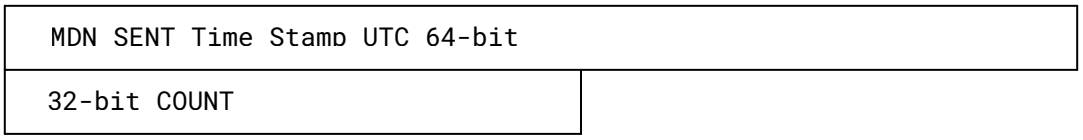


Figure 41: MDNRecord

7.17.1.1. MDNRecord - ABNF

ABNF:

```
                ; The number of MDN emails in the list.
MDNListCount = Length

                ; A list of all MDN records for the associated object.
MDNRecord    = MDNSent MDNListCount 1*MDNRecord
```

7.17.1.2. MDNRecord - XDR

XDR:

```
                ; The number of MDN emails in the list.
MDNListCount = Length

                ; A list of all MDN records for the associated object.
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 14: MDMEntry ABNF/XDR Mapping

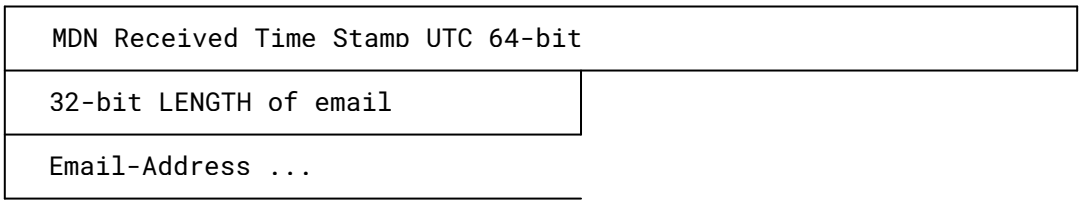


Figure 42: MDNEntry

7.17.2.1. MDNEntry ABNF

ABNF:

```
            ; A single MDN Entry. When (or zero) and the email
MDNEntry  = UTC string
```

Figure 43: MDNEntry - ABNF

7.17.2.2. MDNEntry - XDR

XDR:

```
/**
 * A single MDN entry, when (or zero) and the email.
 */
struct MDMEntry
{
    UTC Received;
    string EMail<>;
};
```

Figure 44: 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   = %x00:8

IndexOpUse      = %x01:8

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 45, 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.

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

Table 15

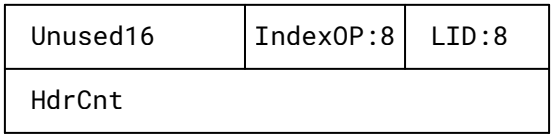


Figure 45: Interest Header List Prefix

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 46, 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 16

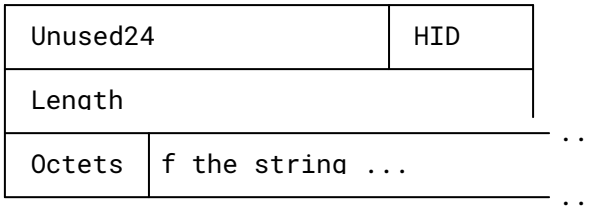


Figure 46: Setting the Interest List - Contents

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.

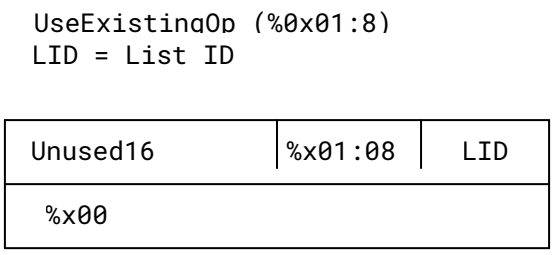


Figure 47: Using Existing Header Index by List ID (LID)

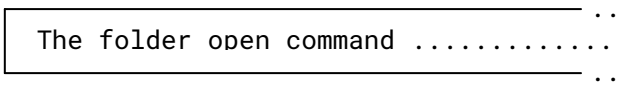
ABNF:

```
UseExistingOp      = %x01:8

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

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

Figure 48: Example Setting a List

Where:

- (a): A 32-bit unsigned integer in network byte order as described in Figure 45.

  The first 8-bits are zero.

  The IndexOP of zero, which means defining a list.

  And in this example two (%x02) 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 46.

  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 46.

  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 (%xc): '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:

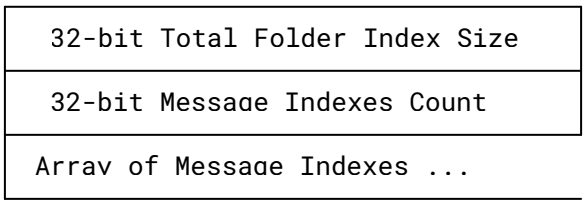


Figure 49: Folder Index

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.

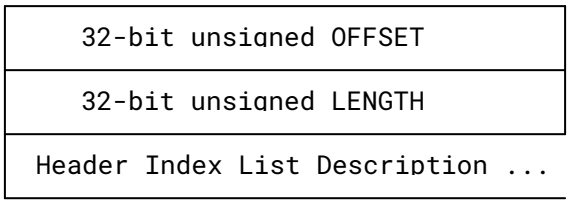


Figure 50: Message Index

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
```

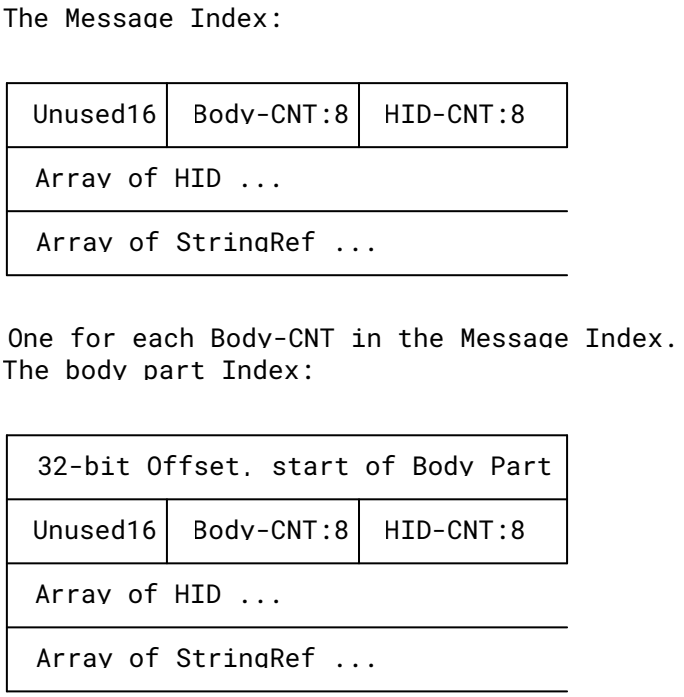


Figure 51: Header Index

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 %x00:8)
                       / (%x00:8 %x00:8)

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 %x00:8)
                       / (HID:8 HID:8 %x00:8 %x00:8)
                       / (HID:8 %x00:8 %x00:8 %x00:8)
```

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.
```

7	0	0	2	CNT.BodyCnt.From.To.
2	2	5	3	To.To.Date.Message-ID.
1	4	0	0	Subject.Content-Type.pads 0
6				OFFSET: Doug@example.com
16				LENGTH: 16
28				OFFSET: Notices@example.com
19				LENGTH: 19
49				OFFSET: Supervisors@example.com
23				LENGTH: 23
74				OFFSET: Dave@example.com
16				LENGTH: 16
98				OFFSET: 06 Feb ...
31				LENGTH: 31
142				OFFSET: <732er ....>
50				LENGTH: 50
204				OFFSET: A new draft ...
42				LENGTH: 42
249				OFFSET: Content/Type
10				LENGTH: 10

Figure 52: 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--
```

5	2	8	4	CNT. Body Cnt. From. To.
12	9	3	0	Subject.MIME.Content-Type
6				OFFSET: User@example.com
16				LENGTH: 16
28				OFFSET: User2@example.com
17				LENGTH: 17
56				OFFSET: This is the subject
39				LENGTH: 39
111				OFFSET: 1.0
3				LENGTH: 3
130				OFFSET: multipar/alternative
50				LENGTH: 50

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

206				Offset to start of Body Part
1	0	3	0	CNT. BodyCnt.Content-Type.pad
220				OFFSET to: Content/Type
27				LENGTH of: 27

Then the second body part:

361				Offset to start of Body Part
1	0	3	0	CNT.BodyCnt.Content-Type.pad
376				OFFSET: Content/Type
26				LENGTH: 26

Figure 53: Header And Body Part Index

9. IANA Considerations

This memo includes no request to IANA. [CHECK]

10. Security Considerations

Robust digital certificate control. Especially with AUTHCERT_TLS.

MORE TODO ...

11. References

11.1. Normative References

[POSIX] IEEE, "1003.1-2024 - IEEE/Open Group Standard for Information Technology--Portable Operating System Interface (POSIX™) Base Specifications", June 2024, <https://ieeexplore.ieee.org/servlet/opac?punumber=10555527>.

[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

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

[rpcgendocs] Unknown Author, "rpcgen Protocol Compiler", January 2025, <https://docs.oracle.com/cd/E37838_01/html/E61058/rpcgenpguide-12915.html>.

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

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.

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

Table 17

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: %xffffffff

Table 18

Appendix C. Capability - Index

Below is a comprehensive list of capability values defined in this specification.

Some capabilities are only sent pre-authentication (P), some are sent only post-authentication (O), and some can be sent pre-authentication or post-authentication, or both (B).

Name	Value	Data Type	Required or Optional	Pre (P), Post (O), or Both (B)	Defined in
AUTHMD5	%x10:31	Boolean	Required	P	Section 5.8

Name	Value	Data Type	Required or Optional	Pre (P), Post (O), or Both (B)	Defined in
AUTHCERT	%x11:31	Boolean	Optional	P	Section 5.8

Table 19: Capability - Index

Appendix D. 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.
xxx	FILE_xxx	xxx.
4294967296	Reserved for future expansion.	4294967296 has a hex value of: %xffffffff

Table 20

Appendix E. 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: %xffffffff

Table 21

Appendix F. Complete ABNF

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

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

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

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

                ; The number of octets in the the
                ;  associated object.
Length        = uint32_t

                ; 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        = Length *UTF8-Char

                ; Like string, except when the length is
                ; 0xffffffff it means the associated value has
                ; been deleted.
stringD       = (0xffffffff:32) / string

                ; The UTF8-Char is any valid and complete
                ; UTF-8 character.
                ;
UTF8-Char     = 1*4uint8_t

                ; 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        = Length *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 compliant systems.
UTC           = uint64_t

                ; The number of octets from the beginning of the
                ; associated object.
Offset        = uint32_t

                ; Key and Value. It uses stringD as the value
                ; to enable an indicator that the key part
                ; (string part) has been deleted.
KeyPair       = string stringD

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

                ; Setting a value.
OpSet         = %x00:8

                ; Getting a value.
OpGet         = %x01:8

                ; Updating an existing value.
OpUpdate      = %x02:8

                ; Deleting an existing value.
OpDelete      = %x03:8

                ; Any one of the the operations.
Op            = OpSet / OpGet / OpUpdate / OpDelete

true          = %x01:8

false         = %x00:8

enabled       = true   / false

unused8       = %x00:8

unused16      = %x00:16

unused24      = %x00:24

                ; The highest bit when, 1 means it is
                ; a vendor extension.
VENDOR_BIT    = %x01:1

                ; The highest bit when, 0 means it is
                ; a Phoenix command or future Phoenix command.
PHOENIX_BIT   = %x00:1
```

```
                 ; A reference to the start and length of a string.
StringRef     = Offset Length
```

```
                  ; The length of a packet.
PacketHeader    = Length
```

```
              ; The data that follows is ADMIN CMD data.
ADMIN_CMD  = %x00:31

              ; The data that follows is AUTH CMD data.
AUTH_CMD   = %x01:31

              ; The data that follows is FILE or FOLDER CMD data.
FILE_CMD   = %x02:31

              ; The data that follows is Phoenix PROTO CMD
              ; specific command data.
PROTO_CMD  = %x03:31

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

              ; Any command. A 32-bit value.
CMD        = (VENDOR_BIT / PHOENIX_BIT)
              (ADMIN_CMD / AUTH_CMD / FILE_CMD / PROTO_CMD)

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

              ; A Phoenix packet
CommandSet = SEQ CMD
              (AdminPayload
              / AdminReplyPayload
              / AuthPayload
              / AuthReplyPayload
              / FilePayload
              / FileReplyPayload
              / ProtoPayload
              / ProtoReplyPayload)

              ; Length: The number of CommandSet objects.
PacketBody = Length 1*CommandSet;
```

```
                      ; What follows is CONFIGURE operation.
SERVER_CONFIGURE    = %x00:8

                      ; What follows is a KICK USER operation.
SERVER_KICK_USER    = %x01:8

                      ; What follows is a LOGS operation.
SERVER_LOGS         = %x02:8

                      ; What follows is a BAN USER operation.
SERVER_MANAGE_BANS = %x03:8

                      ; What follows is a SERVER SHUTDOWN operation.
SERVER_SHUTDOWN     = %x04:8

                      ; What follows is a VIEW STATS operation.
SERVER_VIEW_STATS   = %x05:8

                      ; A USER CREATE (or enable) operation follows.
USER_CREATE         = %x06:8

                      ; A USER DELETE (or disable) operation follows.
USER_DELETE         = %x07:8

                      ; Get one or more USER info.
USER_LIST           = %x08:8

                      ; Give or take away USER permissions.
USER_PERMISSIONS    = %x09:8

                      ; Alter a users login name.
USER_RENAME         = %x0A:8

                      ; All ADMIN commands are 32-bit values.
AOID                = (VENDOR_BIT / PHOENIX_BIT) Unused23
                      (SERVER_CONFIGURE
                       / SERVER_KICK_USER
                       / SERVER_LOGS
                       / SERVER_MANAGE_BANS
                       / SERVER_SHUTDOWN
                       / SERVER_VIEW_STATS
                       / USER_CREATE
                       / USER_DELETE
                       / USER_LIST
                       / USER_PERMISSIONS
                       / USER_RENAME)

                      ; Where CmdPayload contents depend on the
                      ; the value of AOID.
AdminPayload        = AOID
                      (ServerConfigPayload
                      / ServerKickPayload
                      / ServerLogsPayload
                      / ServerBansPayload
                      / ServerShutdownPayload
                      / ServerStatsPayload
                      / UserCreatePayload
```

```
                      / UserDeletePayload
                      / UserListPayload
                      / UserPermissionsPayload
                      / UserRenamePayload)

                      ; Where CmdPayload contents depend on the
                      ; the value of AOID.
AdminReplyPayload  = AOID
                      (ServerConfigReplyPayload
                      / ServerKickReplyPayload
                      / ServerLogsReplyPayload
                      / ServerBansReplyPayload
                      / ServerShutdownReplyPayload
                      / ServerStatsReplyPayload
                      / UserCreateReplyPayload
                      / UserDeleteReplyPayload
                      / UserListReplyPayload
                      / UserPermissionsReplyPayload
                      / UserRenameReplyPayload)
```

```
                        ; Setting or updating a configuration value.
ConfigSet            = (Unused24:24 Op:8):32 Length 1*KeyPair

                        ; The SERVER_CONFIGURE command.
ServerConfigPayload  = SERVER_CONFIGURE Length 1*ConfigSet
```

```
                    ; The SERVER_CONFIGURE Reply
ServerConfigReplyPayload  = SERVER_CONFIGURE Length 1*KeyPair
```

```
ServerKickPayload  = (VENDOR_BIT / PHOENIX_BIT) SERVER_KICK_PAYLOAD
```

```
                ; The number of MDN emails in the list.
MDNListCount = Length

                ; A list of all MDN records for the associated object.
MDNRecord    = MDNSent MDNListCount 1*MDNRecord
```

```
            ; A single MDN Entry. When (or zero) and the email
MDNEntry  = UTC string
```

```
Login       = string

Md5Password = string

AuthMD5     = (VENDOR_BIT / PHOENIX_BIT) AUTHMD5 Login Md5Password
```

```
AUTHANONYMOUS    = %x26
AUTHMD5          = %x10
AUTHCERT         = %x11
AUTHCERT_TLS     = %x27
AUTHCERT_USER    = %x28

AuthPayload      = (AuthAnonymousPayload
                 / AuthMD5Payload
                 / AuthCertPayload
                 / AuthCertTlsPayload
                 / AuthCertUserPayload)

; There is no AUTH... REPLY payload.
; An authentication replies with:
;
;   CAPABILITY_POST: Authentication passed.
;
;   CAPABILITY_PRE:  Authentication failed.
;
;   NotSupported:    Authentication method not supported.
```

```
AuthAnonymous  = (VENDOR_BIT / PHOENIX_BIT) AUTHANONYMOUS
```

Appendix G. Complete XDR

```
/**
 * The time in seconds since January 1 1970 in GMT.
 */
typedef uint64_t UTC;
/**
 * the number of octets from the beginning of
 * the associated object.
 */
typedef uint32_t Offset;
/**
 * The number of octets in the associated object.
 */
typedef uint32_t Length;
/**
 * When a stringD value is set to this,
 * The the associated value has been deleted.
 */
const Deleted = 0xffffffff;
/**
 * A stringD is a string, or Deleted (the associated
 * data has been deleted).
 */
union stringD switch (uint32_t LengthOrDeleted) {
  case Deleted:
    void;
  default:
    opaque String<>;
};
/**
 * A string Key and its associated Value.
 * It uses stringD as the value to enable an indicator that the
 * key part (string part) has been deleted.
 */
struct KeyPair {
    string Key<>;
    stringD Value<>;
};
/**
 * An array of KeyPair objects.
 */
typedef KeyPair KeyPairArray<>;
/**
 * A 1-bit value;
 * The highest bit in the value, 1 means it is
 * a vendor extension.
 */
const VENDOR_BIT = 0x1;
/**
 * A 1-bit value;
 * The highest bit in the value, 0 means it is
 * a Phoenix command or future Phoenix command.
 */
const PHOENIX_BIT = 0x0;
/**
 * Operations
 */
enum Op_e {
```

```
    OpSet = 0x00,
    OpGet = 0x01,
    OpUpdate = 0x02,
    OpDelete = 0x03
};
/**
 * Any one of OpSet, OpGet, OpUpdate, or OpDelete
 * cast to a (Op).
 */
typedef uint8_t Op;
```

```
/**
 * A reference to the start and length of a string.
 */
struct StringRef {
    Offset StartOffset;
    Length StringLength;
};
```

```
/**
 * The length of a packet.
 */
typedef Length PacketHeader;
```

```
/*
 * The ADMIN commands.
 * An XDR enum is 32-bits in size.
 * With the VENDOR_BIT set to zero, making it not a vendor command.
 */
enum CMD {
    /**
     * The data that follows is ADMIN CMD data.
     */
    ADMIN_CMD = 0x00,
    /**
     * The data that follows is AUTH CMD data.
     */
    AUTH_CMD = 0x01,
    /**
     * The data that follows is FILE or FOLDER CMD data.
     */
    FILE_CMD = 0x02,
    /**
     * The data that follows is Phoenix PROTO CMD data.
     */
    PROTO_CMD = 0x03
};
/**
 * Mask to check if CMD value in packet
 * is vendor extension.
 */
const CMD_VENDOR_MASK = 0x80000000;
/**
 * Define a SEQ (sequence) type.
 */
typedef uint32_t SEQ_t;
/**
 * A CMD payload is one of these types.
 * With Cmd set to a CMD value.
 */
union CmdPayload switch (CMD Cmd) {
  case ADMIN_CMD:
    AdminPayload AdminBodyPayload;
  case AUTH_CMD:
    AuthPayload AuthBodyPayload;
  case FILE_CMD:
    FilePayload FileBodyPayload;
  case PROTO_CMD:
    ProtoPayload ProtoBodyPayload;
};
/*
 * One comand.
 */
struct CommandSet {
    SEQ_t SEQ;
    CmdPayload Payload;
};
/**
 * A packet body.
 */
struct PacketBody {
```

```
  CommandSet Commands<>; /* XDR arrays start with the Length */
};
```

```
/**
 * ADMIN commands.
 * In XDR, enum values are 32-bit.
 * The high bit is zero, which means the PHOENIX_BIT is zero
 * in these values.
 */
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
};
/**
 * All ADMIN commands are sent as 32-bit values.
 */
typedef AOID_e AOID;
```

```
union AdminPayload switch(AOID_e Aoid) {
  case SERVER_CONFIGURE:
    ServerConfigPayload ConfigPayload;
  case SERVER_KICK_USER:
    ServerKickPayload KickPayload;
 case SERVER_LOGS:
    ServerLogsPayload LogsPayload;
  case SERVER_MANAGE_BANS:
    ServerBansPayload BansPayload;
  case SERVER_SHUTDOWN:
    ServerShutdownPayload ShutdownPayload;
  case SERVER_VIEW_STATS:
    ServerStatsPayload StatsPayload;
  case USER_CREATE:
    UserCreatePayload UCreatePayload;
  case USER_DELETE:
    UserDeletePayload UDeletePayload;
  case USER_LIST:
    UserListPayload UListPayload;
  case USER_PERMISSIONS:
    UserPermissionsPayload UPermissionsPayload;
  case USER_RENAME:
    UserRenamePayload URenamePayload;
};
union AdminReplyPayload switch(AOID_e Aoid) {
  case SERVER_CONFIGURE:
    ServerConfigReplyPayload ConfigPayload;
  case SERVER_KICK_USER:
  void; /* No reply allowd for SERVER_KICK_USER */
 case SERVER_LOGS:
    ServerLogsReplyPayload LogsPayload;
  case SERVER_MANAGE_BANS:
    ServerBansReplyPayload BansPayload;
  case SERVER_SHUTDOWN:
    ServerShutdownReplyPayload ShutdownPayload;
  case SERVER_VIEW_STATS:
    ServerStatsReplyPayload StatsPayload;
  case USER_CREATE:
    UserCreateReplyPayload UCreatePayload;
  case USER_DELETE:
    UserDeleteReplyPayload UDeletePayload;
  case USER_LIST:
    UserListReplyPayload UListPayload;
  case USER_PERMISSIONS:
    UserPermissionsReplyPayload UPermissionsPayload;
  case USER_RENAME:
    UserRenameReplyPayload URenamePayload;
};
```

```
/**
 * An array of OpConfigSet values.
 */
struct ServerConfigPayload {
    AOID_e Aoid; /* Set to SERVER_CONFIGURE. */
    ConfigSet Values<>; /* XDR arrays start with a length. */
};
```

```
/**
 * Reply to SERVER_CONFIGURE
 */
struct ServerConfigReplyPayload {
    AOID_e ACmd; /* Set to SERVER_CONFIGURE */
    KeyPair ResultValues<>; /* XDR arrays start with a length */
};
```

```
struct ServerKickPayload
{
 AOID_e Kick; /* With Kick set to SERVER_KICK_USER */
};
```

```
struct ServerLogsPayload
{
 int foo;
};
struct ServerLogsReplyPayload
{
 int foo;
};
```

```
struct ServerBansPayload
{
 int foo;
};
struct ServerBansReplyPayload
{
 int foo;
};
```

```
struct ServerShutdownPayload
{
 int foo;
};
struct ServerShutdownReplyPayload
{
 int foo;
};
```

```
struct UserCreatePayload
{
 int foo;
};
struct UserCreateReplyPayload
{
 int foo;
};
```

```
struct UserDeletePayload
{
 int foo;
};
struct UserDeleteReplyPayload
{
 int foo;
};
```

```
struct UserListPayload
{
 int foo;
};
struct UserListReplyPayload
{
 int foo;
};
```

```
struct UserPermissionsPayload
{
 int foo;
};
struct UserPermissionsReplyPayload
{
 int foo;
};
```

```
struct UserRenamePayload
{
 int foo;
};
struct UserRenameReplyPayload
{
 int foo;
};
```

```
struct ServerStatsPayload
{
 int foo;
};
struct ServerStatsReplyPayload
{
 int foo;
};
```

```
/**
 * The number of emails in the MDN record set.
 */
typedef Length MDNListCount;
/**
 * When the MDN was sent.
 */
typedef UTC MDNSent;
/**
 * A list of all MDN records for the associated object.
 */
struct MDNRecord {
    MDNSent TimeSent;
    /* The first item in an XDR array, is its size (MDNListCount) */
    MDNRecord Entries<>;
};
```

```
/**
 * A single MDN entry, when (or zero) and the email.
 */
struct MDMEntry
{
    UTC Received;
    string EMail<>;
};
```

```
struct AuthMD5 {
    uint32_t Cmd; /* Set to AUTHMD5 */
    string Login<>;
    string Md5Password<>;
};
```

```
struct AuthAnonymousPayload
{
 AOID_e Anonymous; /* Set to AUTHANONYMOUS (%x26) */
};
```

```
enum Auth_e {
    AUTHANONYMOUS = 0x26,
    AUTHMD5 = 0x10,
    AUTHCERT = 0x11,
    AUTHCERT_TLS = 0x27,
    AUTHCERT_USER = 0x28
};
```

```
union AuthPayload switch(Auth_e Auth) {
  case AUTHANONYMOUS:
    AuthAnonymousPayload AnonymousPayload;
  case AUTHMD5:
    AuthMD5Payload MD5Payload;
 case AUTHCERT:
    AuthCertPayload CertPayload;
  case AUTHCERT_TLS:
    AuthCertTlsPayload CertTlsPayload;
  case AUTHCERT_USER:
    AuthCertUserPayload CertUserPayload;
};
/**
 * There is no AUTH... REPLY payload.
 * An authentication replies with:
 *
 * CAPABILITY_POST: Authentication passed.
 *
 * CAPABILITY_PRE: Authentication failed.
 *
 * NotSupported:   Authentication method not supported.
 */
```

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+208-806-1358
Email: DouglasRoyer@gmail.com
URI: https://DougRoyer.US