Workgroup:	Internet Engineering Task Force
Internet-Draft:	draft-royer-phoenix-00
Published:	19 May 2025
Intended Status:	Standards Track
Expires:	20 November 2025
Author:	DM. Royer
	RiverExplorer LLC

Phoenix: Lemonade Risen Again

Abstract

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.

Security and authentication are always a serious issue with protocolls. This proposal allows for legacy authentication mechanisms to work, in some cases unaltered.

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 objects..

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

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

NOTICE: This is WORK IN PROGRESS

Just about everything transported on the net is a MIME object and there are many ways to transport them. This document specifies a new application level MIME transport mechanism and protocol. This is an attempt to unity the transfer protocol of MIME and other objects.

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

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

An additional consideration is transporting the perhaps very large objects. Some MIME objects are so large that the default for many devices is to not download the without a specific user request. Some large object can be downloaded and can be useful in pieces, other object types will need to be fully downloaded.

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 basically 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 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, or parts of a file.

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. This protocol allows for a consistent transport of basic access control information. You have access to read the data, or not. You can share the data, or not. You can create new remote objects, or not.

File and folder meta data can be a complex task that can involve access control lists, group lists, and various permissions. This protocol is not designed to manage complex access control lists.

The goal of this protocol is to provide a consistent transport mechanism. This is a transfer protocol to be used by applications, it is not a application. A streaming application might not implement some of the protocol. And an email application might not implement streaming.

What this specification defines:

- How to leverage existing authentication implementations or use new ones.
- 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 this Phoenix protocol. Provides links to sample implementations.
- A way to transport data between applications.

3. ABNF, Notes, and Definitions

3.1. CBOR TYPE - Meaning - Informative

Any specific computer programming language has list of data types it supports. And developers can manage those data types and bundle them into groups of data.

There are two very general data types discussed. in this specification.

- Application and computer data types. This is called application data types.
- CBOR over the wire data types. This is called CBOR data types.

A CBOR encoder takes application data and converts it to CBOR data and places them on an output stream.

And a CBOR decoder takes the CBOR data from an input stream and converts them to application data.

CBOR is a method of converting application data to and from a standard format.

Applications may need to specify the size of data. This is often done with bit size. And these bit sizes are often in multiples of 8-bits at a time.

CBOR data types are more flexible. CBOR encoding and decoding is based on data type and data value.

An application 64-bit unsigned integer value of 1, is encoded into an 8-bit CBOR unsigned integer. CBOR transfers values, using data types. Typical applications protocols transfer data types, that have values.

Application data types are not the same as CBOR data types. In this specification some diagrams and text may represent the data from the applications point of view using application data types. And other diagrams and text from the CBOR encoded point of view using CBOR data types. These will be noted in the text and diagrams.

3.1.1. Basic Data types.

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-bit unsigned integer. xxxxxxxx

Arrays are used in a lot of data flows. String are always an array of printable characters. Some data needs to be converted to a from network byte order. Data that does not need to be converted to and from network byte order is called 'opaque' data. A PNG file is an example of opaque data. The PNG data itself is defined so that it is readable with out any additional encoding.

In the CBOR Language Specification (CLS), a data type that is an array is fixed in size, or variable in size.

A fixed array has its size surrounded by square brackets '[' and ']'.

A variable array has its optional size surrounded by angle brackets '<' and '>'.

NAME	Description:	CBOR API / Data Type

string	• A string is always an array.
	• A string is an array of UTF-8 printable characters. With an optional maximum length.
	• A string is an array of printable characters. And each character may be 1, 2, 3, or 4 octets in size. The length specifies the number of printable characters in the string not the size of the array.
	• The octet count (size) of any string, could be the same as the number of characters (length), or the size could be up to 4 times larger. So strings have a size, and a length.
	• The length is the number of characters in the array.
	• The size is the number of octets in the array.
	• The length is always equal to or less than the size.
	
	// An array of printable characters, unlimited in size.
	string MyVariableName<>;
	
	// An array of printable characters, up to 25 in size.
	string MyVariableName2<25>;
		cbor_string()
		
		Major type 3 Text String

NAME	Description:	CBOR API / Data Type

opaque	• An opaque array is an array of 8-bit octets. And will not be network byte encoded or decoded.
	• The size and length of an opaque array are always the same value.
	
	// Always 42 in size.
	opaque MyOpaqueData[42];
	
	// Up to 42 in size.
	opaque MyOpaqueData<42>;
	
	// Unlimited in size.
	opaque MyOpaqueData<>;
		cbor_opaque()
		
		Major Type 2

Table 1: CLC strings and arrays.

Arrays in CBOR have a "<>" in them to indicate they are variable length arrays. A variable array may have a maximum length. All elements in the list are the same type.

Arrays in CBOR have a "[]" in them to indicate they are fixed length arrays. A fixed length array is an array that is always the same size and is not dependent on any run time or calculated values. It is a value that can be compiled into the code. All elements in the list are the same type.

An ObjectList is a variable or fixed size array of objects each of which may be a different type.

The opaque data type represents data that is not encoded in any way. A PNG file is an example that does not get network byte order translated, it is sent as an opaque blob of unaltered data.

A 'string' in CBOR represents a string of printable UTF-8 characters.

3.2. Number of bits in value

For application data types: In this specification terminal values may specify a bit width. Indicating the number of bits in the value. This is the number of bits in the application uses.

The application packs bits, sequences of bits into unsigned integer values padding the unused bits to zero. CBOR then treats them as unsigned integer values and encodes and decodes them normally.

The application may see the data 3 separate values. Place them into one 32-bit value, and send them to CBOR encoding.

This is done by placing a colon (:) after the application data type or variable name, followed by the decimal number of bits. And other application data types have the number of bits in the data type name. Some examples:

SomeDataType Length:21
: An example of a variable name, that needs 21-bits in the application.

uint_t F:5
: An unsigned integer, that has a value that is 5-bits wide. The application does not really care how it is stored. It cares that it needs 5-bits to represent the data.

In cases where the number of bits an application needs is not a multiple of 8 bits, the next larger application data type will be used. The value will be in the lower bits, and the unused upper bits will be zero.

In example Figure 1 'F' is 5-bits wide. Length is 21-bits wide. No application data type is 26 bits wide, so a 32-bit values is used, and 6-bits unused bits are set to zero.

Depending on how the application implements it:

- It could pass Length to be CBOR encoded, as an unsigned integer. Followed by F as an unsigned integer to be CBOR encoded. In this case the application would have to split the 32-bit value before sending. And the receiving application would have to reassemble the two unsigned integers back into the one 32-bit unsigned integer.
- Or it could pass the one 32-bit value as an unsigned integer to CBOR for encoding.

This is one example of why an application needs to specify exactly how the data is to be encoded. While at other times, just cares about the value.

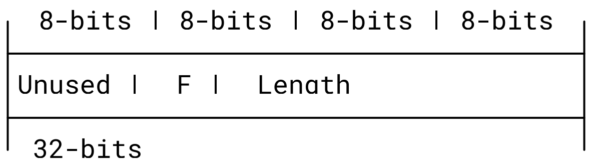

Figure 1: Packed Bit Example.

3.3. Common Definitions

In some cases the application may need to specify the number of bits for a value. In other cases the application might be able to just use an unsigned integer without regard for its size. However compilers and computer languages often need to know the size, or how much memory to allocate and use to hold variables at compile time.

Table Table 2 lists the names used in this specification for types and their widths.

TYPE	Notes	CBOR API
int_t	Any signed integer without any predefined number of bits.	cbor_int_t()
int8_t	An 8-bit signed integer.	cbor_int8_t()
int16_t	A 16-bit signed integer.	cbor_int16_t()
int32_t	A 32-bit signed integer.	cbor_int32_t()
int64_t	A 64-bit signed integer.	cbor_int64_t()
int?_t	Where ? is any multiple of 8. A ?-bit signed integer.	cbor_int?_t()
uint_t	Any unsigned integer without any predefined number of bits.	cbor_uint_t()
uint8_t	An 8-bit unsigned integer.	cbor_uint8_t()
uint16_t	A 16-bit unsigned integer.	cbor_uint16_t()
uint32_t	A 32-bit unsigned integer.	cbor_uint32_t()
uint64_t	A 64-bit unsigned integer.	cbor_uint64_t()
uint?_t	Where ? is any multiple of 8. A ?-bit unsigned integer.	cbor_uint?_t()
string	A string of UTF-8 characters.	cbor_string()
opaque	An array of 8-bit values that will not be CBOR encoded or CBOR decoded when transferring the data over this protocol.	cbor_opaque()
Op	An 8-bit unsigned value. When the highest bit is one (1) it is a vendor specific Op. Otherwise, it is set to zero (0) and is not a vendor specific operation.	cbor_Op()
OpSet	An 8-bit unsigned value. Signifies the operation will set a value.	cbor_OpSet()
OpGet	An 8-bit unsigned value. Signifies the operation will get a value.	cbor_OpGet()
OpUpdate	An 8-bit unsigned value. Signifies the operation will update an existing value.	cbor_OpUpdate()
OpDelete	An 8-bit unsigned value. Signifies the operation will delete a value.	cbor_OpDelete()

TYPE	Notes	CBOR API
true	An 8-bit unsigned value. A value of true.	cbor_false()
false	An 8-bit unsigned value. A value of false.	cbor_true()
VENDOR_BIT	A 1 bit value, set to 1. It is placed in the highest bit position in the value.	

Table 2: Phoenix Protocol types

3.3.1. Common Definitions - CBOR

```
/**
 * A value that can hold any signed
 * integer value is called an int_t.
 */
typedef integer int_t;

/**
 * An 8 bit signed integer
 * is called an int8_t.
 */
typedef integer:8 int8_t;

/**
 * An 16 bit signed integer
 * is called an int16_t.
 */
typedef integer:16 int16_t;

/**
 * An 32 bit signed integer
 * is called an int32_t.
 */
typedef integer:32 int32_t;

/**
 * An 64 bit signed integer
 * is called an int64_t.
 */
typedef integer:64 int64_t;

/**
 * A value that can hold any unsigned integer value
 * is called an uint_t.
 */
typedef unsigned_integer uint_t;

/**
 * An 8 bit signed integer
 * is called an uint8_t.
 */
typedef unsigned_integer:8 uint8_t;

/**
 * An 16 bit signed integer
 * is called an uint16_t.
 */
typedef unsigned_integer:16 int16_t;

/**
 * An 32 bit signed integer
 * is called an uint32_t.
 */
typedef unsigned_integer:32 int32_t;

/**
 * An 64 bit signed integer
 * is called an uint64_t.
 */
typedef unsigned_integer:64 int64_t;

/**
 * The time in seconds since January 1 1970 in GMT
 * is a 64-bit unsigned integer (uint64_t) and
 * is called UTC_t.
 */
typedef uint64_t UTC_t;

/**
 * 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;

/**
 * A single UTF-8 character.
 */
typedef uint8_t string;

/**
 * An array of UTF-8 character.
 * with no predefined length.
 */
typedef uint8_t string<>;

/**
 * Any 8-bit value without any
 * associated data type.
 */
typedef uint8_t opaque;

/**
 * An array of opaque values
 * with no predefined length.
 */
typedef uint8_t opaque<>;

/**
 * When a stringD value is set to this,
 * The the associated value has been deleted.
 */
const Deleted = 0xffffffff;

/**
 * A stringD is a string of printable UTF-8
 * characters,
 * -Or Deleted (the associated data has been deleted),
 * -Or is nullptr (not assigned).
 */
union stringD switch (uint32_t LengthOrDeleted) {
  case nullptr:
    void

  case Deleted:
    void;

  default:
    // An array of uint8_t characters
    // with no predefined length.
    //
    string Utf8Characters<>;
};

/**
 * An 8-bit value used to represent
 * true or false.
 */
typedef uint8_t bool;

/**
 * The value true.
 */
const bool true = 1;

/**
 * The value false.
 */
const bool false = 0;

/**
 * A string Key and its associated Value.
 * It uses stringD as the value to enable an indicator that the
 * key part (string part) may have been deleted.
 * And a stringD Value may be nullptr.
 */
struct KeyPair {
    string Key<>;
    stringD Value<>;
};

/**
 * An array of KeyPair objects
 * with an unspecified length.
 */
typedef KeyPair KeyPairArray<>;

/**
 * A 1-bit value;
 * The highest bit in the value, 1 means it is
 * a vendor extension.
 */
const VENDOR_BIT = 0x1;

/**
 * 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 2: Common Definitions - CBOR

3.4. Ref

This protocol references strings and other object in existing objects by octet offset into the object. This is is called a Ref. All references can be referenced by using a total of 8 octets. The Ref does not contain the value, it is a reference an existing value in an object. A Ref consists of two parts, Offset and length:

Name	Description	CBOR API
Offset	The octet count to the start of the value with zero being the first octet in the object.	cbor_Offset()
Length	The length in octets of the value.	cbor_Length()
Ref	A reference object.	cbor_Ref()

Table 3: Ref ABNF/CBOR Mapping

A Ref over the wire is 8 octets in size.

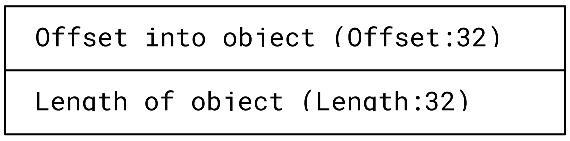

Figure 3: Ref Format

ABNF:

3.4.1. Ref ABNF

```
          ; A reference to the start and length of a object.
Ref     = Offset Length
```

Figure 4: Ref ABNF

3.4.2. Ref CBOR

The CBOR definitions are:

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

Figure 5: Ref 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]

CBOR

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

Note: CBOR 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. The client then waits for the Auth reply from the server.

- If the client gets a successful Auth reply, then the client sends its post authentication capability list to the server.
- If the client get an unsuccessful Auth reply, then the client sends its pre authentication capability list to the server followed by another Auth command. Upon too many retries, the server or client may terminate the connection.

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

- If the Auth was successful, then the server sends a post authentication capability list.
- If the Auth was unsuccessful, then the server sends its pre authentication capability list to the client.

When the server gets a new connection, it waits for a packet from the client. It will be a pre authentication capability packet, or a authentication packet. When the server has an unauthenticated connection, it only accepts two kinds of packets:

- A pre authentication capability packet. Which is replied to by the server with its pre authentication packet.
- An authentication packet.

Upon too many retries, the server or client may terminate the connection.

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 and other objects.

This is a binary protocol encoded using CBOR streaming. 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 CBOR as the format for transferring binary control information it is portable to any computer architecture.

The basic connection starts in one of two modes.

- An account is connecting to a server for the first time and does not know which authentication methods the server supports.
- An account has authenticated to a server in the past or the client is already aware of a valid authentication protocol to use.

When connection is made, the server waits for a packet from the client. It will be one of two kinds.

- A pre authentication capability packet (CAPABILITY_PRE).
- An authentication packet. (AUTH...).

If the first packet the server receives is a pre authentication capability packet, The server examines the clients packet and determines what authentication options to present to the client. The server then sends back its pre authentication capability packet which includes the supported authentication methods. Then the client, using the information from the server starts the authentication process. On a successful authentication, the server sends the client a post authentication capability packet with a new sequence number.

If the first packet the server receives is any of the supported authentication packets, then the server processes the authentication packet. On failure to authenticate, the server sends the client a pre authentication capability packet with the same sequence number that was in the authentication attempt and includes the servers supported authentication methods. On a successful authentication, the server sends the client a post authentication capability packet with a new sequence number.

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 CBOR unsigned integer. The length does not include its own length. It is the total length that follows the length value.

Name	Description	CBOR API
PacketHeader	The number of octets that follow this value that are part of this packet.	cbor_PacketHeader()

Table 4: Packet Header ABNF/CBOR Mapping

5.1.1.1.1. Packet Header ABNF

ABNF:

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

Figure 6: Packet Header ABNF

5.1.1.1.2. Packet Header CBOR

CBOR Definition:

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

Figure 7: Packet Header CBOR

5.1.1.2. Packet Body (PacketBody)

The packet body is divided into four parts:

1. Number of commands that are in the packet.
2. Sequence (SEQ)
3. Command (CMD):
4. Payload (CmdPayload): The command specific data.

PacketBody details:

Name	Value	Description	CBOR API
Length	An unsigned integer	The number of CMD objects in this packet.	cbor_Length()
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. Over the wire it is a CBOR unsigned integer. 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.	cbor_SEQ()

Name	Value	Description	CBOR API
CMD	1 bit + 31 bits.	A 31-bit value that is a Phoenix compliant command or a command with the VENDOR_BIT set (a vendor command). 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 the VENDOR_BIT. CMD covers vendor and phoenix commands. • Phoenix CMD range is: %x00000000-7fffffff. • Vendor CMD range is: 80000000-fffffffe. • With %xffffffff reserved. All vendor commands are followed by another 32-bit Length value indicating how may octets follow the Length that are in the payload. This is so that compliant implementation that do not support the vendor extensions know how many octets to skip to find the next command or end of packet.	cbor_Cmd()
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 CBOR set of data. See the specific CMD for details.	cbor_CmdPayload()

Table 5: Packet Body ABNF/CBOR 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. And vendor commands MUST have a Length value that follows the command that indicates how many octets follow the length. This is so that implementations that do not understand the vendor extension can skip that many more octets to find the next command or operation.

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.

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_3 in this example.

Figure 8 is an example of a CAPABILITY_PRE being sent from the server to the client. A client that is conforming to this specification and does not support any vendor extensions, would ignore the AUTH_Vendor_3 and authenticate with AUTHMD5. NOTE: This is how the application sees the data, this is not the over the wire data as it is not in CBOR format.

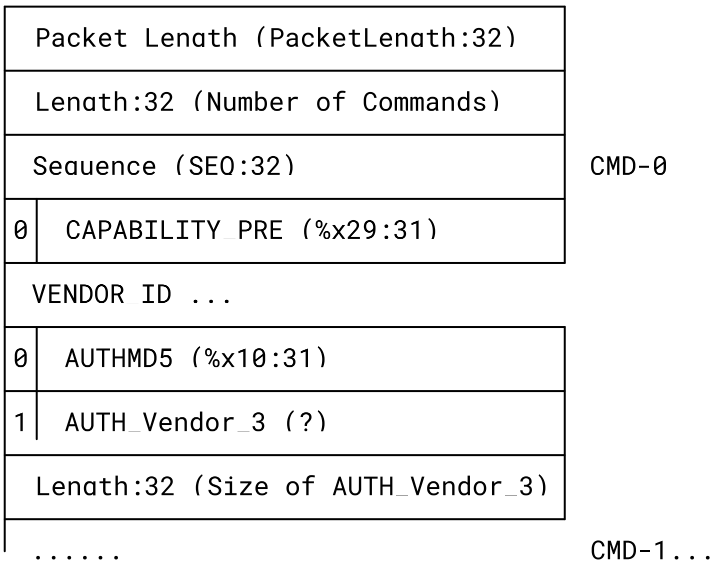

Figure 8: Packet Body Non-Vendor- Diagram

In example Figure 9, 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.

After each entry that has the VENDOR_BIT set, the next value must be a Length indicating how many octets of data follow the Length, even when zero.

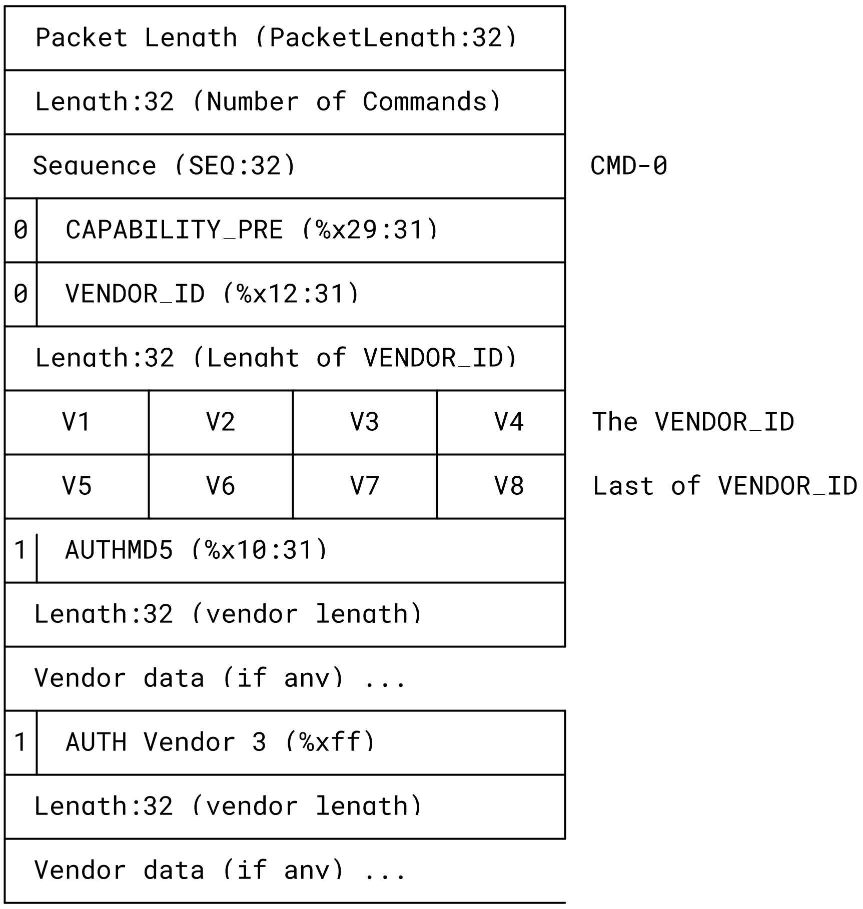

Figure 9: Packet Body Vendor - Diagram

And the matching data for Figure 9 could be:

```
+-------------------------------+
|           %x2c:32             |  44 octets Follow
+-------------------------------+
|           %x01:32             |  1 Command
+-------------------------------+
|           %x03:32             |  SEQ
+-+-----------------------------+
|0|         %x29:31             |  CAPABILITY_PRE
+-+-----------------------------+
|0|         %0x12:31            |  VENDOR_ID
+-+-----------------------------+
|           %0x08               |  Length of VENDOR_ID
+-------+-------+-------+-------+
|  V1   |  V2   |  V3   |  V4   |  The VENDOR_ID
+-------+-------+-------+-------+
|  V5   |  V6   |  V7   |  V8   |  Last of VENDOR_ID
+-------+-------+-------+-------+
|       %x80000029:32           |  Vendor AUTH_MD5
+-------------------------------+
|           %x00:32             |  Zero data
+-------------------------------+
|       %x80abc123:32           |  AUTH_Vendor_3
+-------------------------------+
|           %x00:32             |  Zero data
+-------------------------------+
```

Figure 10: Packet Body Vendor - Diagram Data

Note that the CAPABILITY_PRE in Figure 9 and Figure 10 has the VENDOR_BIT set, so it is followed by the number of octets in the vendor extension.

Multiple commands may be sent in one packet. And PHOENIX and VENDOR commands may be sent in one packet body by setting VENDOR_BIT to (1). This example shows two commands and the start of a third, one that is a PHOENIX command the other is a vendor command.

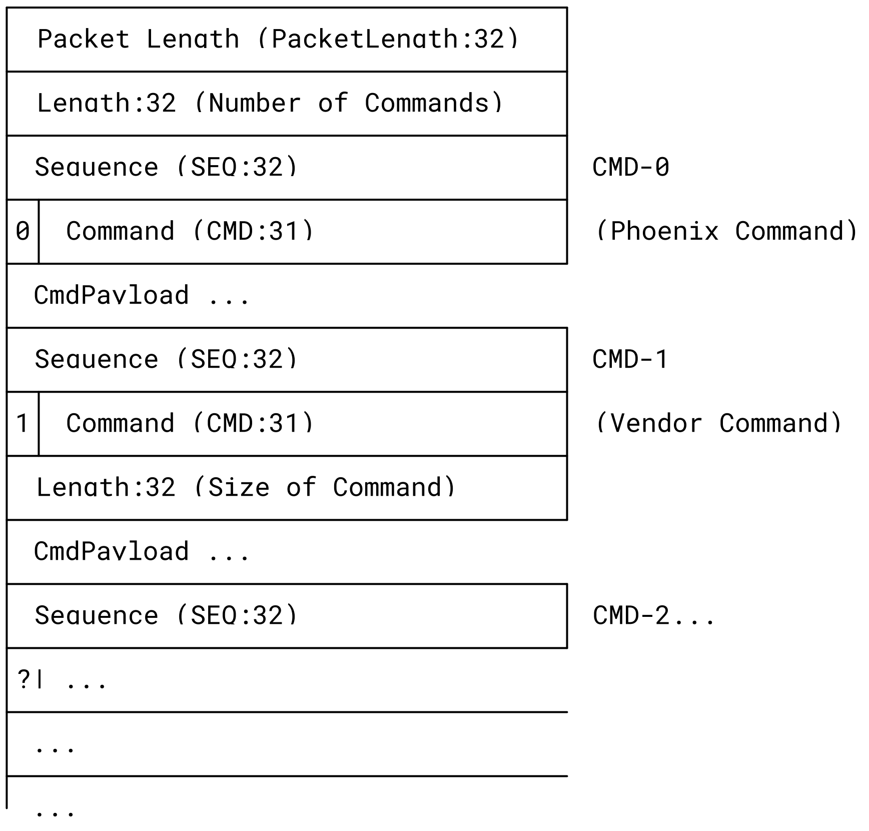

Figure 11: Packet Body Multiple Commands - Diagram

5.1.1.2.1. Packet Body ABNF

ABNF:

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

OneCommand   = AUTHANONYMOUS AuthAnonymousPayload
             / AUTHCERT_TLS AuthCertTlsPayload
             / AUTHCERT_USER AuthCertUserPayload
             / AUTHMD5 AuthMD5Payload
             / AUTHMD5 AuthMD5ReplyPayload
             / CAPABILITY_PRE CapabilityPrePayload
             / FILE_CREATE FileCreatePayload
             / FILE_COPY FileCopyPayload
             / FILE_DELETE FileDeletePayload
             / FILE_RENAME FileRenamePayload
             / FILE_METADATA FileMetaDataPayload
             / FILE_MOVE FileMovePayload
             / FILE_SHARE FileSharePayload
             / FILE_GET FileGetPayload
             / FILE_MODIFY FileModifyPayload
             / FOLDER_CREATE FolderCreatePayload
             / FOLDER_COPY FolderCopyPayload
             / FOLDER_DELETE FolderDeletePayload
             / FOLDER_RENAME FolderRenamePayload
             / FOLDER_METADATA FolderMetaDataPayload
             / FOLDER_MOVE FolderMovePayload
             / FOLDER_OPEN FolderOpenPayload
             / FOLDER_SHARE FolderSharePayload
             / FOLDER_LIST FolderListPayload
             / NOT_SUPPORTED
             / SERVER_CONFIGURE ServerConfigurePayload
             / SERVER_KICK_USER
             / SERVER_LOGS ServerLogsPayload
             / SERVER_MANAGE_BANS ServerManageBansPayload
             / SERVER_SHUTDOWN ServerShutdownPayload
             / SERVER_VIEW_STATS ServerViewStatsPayload
             / USER_CREATE UserCreatePayload
             / USER_DELETE UserDeletePayload
             / USER_LIST UserListPayload
             / USER_PERMISSIONS UserPermissionsPayload
             / USER_RENAME UserRenamePayload
             / RESERVED_CMD

OneReply     = AUTHANONYMOUS AuthAnonymousReplyPayload
             / AUTHCERT_TLS AuthCertTlsReplyPayload
             / AUTHCERT_USER AuthCertUserReplyPayload
             / AUTHMD5 AuthMD5ReplyPayload
             / FILE_CREATE FileCreateReplyPayload
             / FILE_COPY FileCopyReplyPayload
             / FILE_DELETE FileDeleteReplyPayload
             / FILE_RENAME FileRenameReplyPayload
             / FILE_METADATA FileMetaDataReplyPayload
             / FILE_MOVE FileMoveReplyPayload
             / FILE_SHARE FileShareReplyPayload
             / FILE_GET FileGetReplyPayload
             / FILE_MODIFY FileModifyReplyPayload
             / FOLDER_CREATE FolderCreateReplyPayload
             / FOLDER_COPY FolderCopyReplyPayload
             / FOLDER_DELETE FolderDeleteReplyPayload
             / FOLDER_RENAME FolderRenameReplyPayload
             / FOLDER_METADATA FolderMetaDataReplyPayload
             / FOLDER_MOVE FolderMoveReplyPayload
             / FOLDER_OPEN FolderOpenReplyPayload
             / FOLDER_SHARE FolderShareReplyPayload
             / FOLDER_LIST FolderListReplyPayload
             / SERVER_LOGS ServerLogsReplyPayload
             / SERVER_MANAGE_BANS ServerManageBansReplyPayload
             / SERVER_SHUTDOWN ServerShutdownReplyPayload
             / SERVER_VIEW_STATS ServerViewStatsReplyPayload
             / USER_CREATE UserCreateReplyPayload
             / USER_DELETE UserDeleteReplyPayload
             / USER_LIST UserListReplyPayload
             / USER_PERMISSIONS UserPermissionsReplyPayload
             / USER_RENAME UserRenameReplyPayload
             / RESERVED_CMD

RESERVED_CMD = %xffffffff

             ; VendorDefined are only valid when both the
             ; client and server agree on compatible
             ; VENDOR_ID values.
             ;
             ; Where:
             ; Length is the length of data that follows
             ; *uint8_t Is an array of Length 8-bit values.
             ;
VendorDefined = %x80000000-fffffffe Length *uint8_t

Command = SEQ (Command / VendorDefined)
CommandReply = SEQ (CommandReply / VendorDefined)

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

             ; Length: The number of CommandReply objects.
PacketReply = Length 1*CommandReply
```

Figure 12: Packet Body ABNF

5.1.1.2.2. Packet Body CBOR

CBOR Definition:

```
/**
 * Mask to check if CMD value in packet
 * is vendor extension.
 */
const CMD_VENDOR_MASK = 0x80000000;
struct VendorDefined {
 uint32_t VendorCommand; /* 0x80000000-fffffffe */
 opaque Data<>; /* XDR arrays start with a length. */
};
/**
 * A CMD payload is one of these types.
 * With Cmd set to a CMD value.
 */
union OneCommand switch (CMD_e Cmd) {
 case AUTHANONYMOUS:
  AuthAnonymousPayload AuthAnonymousCmd;
 case AUTHCERT_TLS:
  AuthCertTlsPayload AuthCertTlsCmd;
 case AUTHCERT_USER:
  AuthCertUserPayload AuthCertUserCmd;
 case AUTHMD5:
  AuthMD5Payload AuthMD5Cmd;
 case BYE:
  ByePayload ByeCmd;
 case CAPABILITY_PRE:
  CapabilityPayload CapabilityPreCmd;
 case CAPABILITY_POST:
  CapabilityPayload CapabilityPostCmd;
 case FILE_CREATE:
  FileCreatePayload FileCreateCmd;
 case FILE_COPY:
  FileCopyPayload FileCopyCmd;
 case FILE_DELETE:
  FileDeletePayload FileDeleteCmd;
 case FILE_RENAME:
  FileRenamePayload FileRenameCmd;
 case FILE_METADATA:
  FileMetaDataPayload FileMetaDataCmd;
 case FILE_MOVE:
  FileMovePayload FileMoveCmd;
 case FILE_SHARE:
  FileSharePayload FileShareCmd;
 case FILE_GET:
  FileGetPayload FileGetCmd;
 case FILE_MODIFY:
  FileModifyPayload FileModifyCmd;
 case FOLDER_CREATE:
  FolderCreatePayload FolderCreateCmd;
 case FOLDER_COPY:
  FolderCopyPayload FolderCopyCmd;
 case FOLDER_DELETE:
  FolderDeletePayload FolderDeleteCmd;
 case FOLDER_RENAME:
  FolderRenamePayload FolderRenameCmd;
 case FOLDER_METADATA:
  FolderMetaDataPayload FolderMetaDataCmd;
 case FOLDER_MOVE:
  FolderMovePayload FolderMoveCmd;
 case FOLDER_OPEN:
  FolderOpenPayload FolderOpenCmd;
 case FOLDER_SHARE:
  FolderSharePayload FolderShareCmd;
 case FOLDER_LIST:
  FolderListPayload FolderListCmd;
 case NOT_SUPPORTED:
  void;
 case RESERVED_CMD:
  void;
 case SERVER_CONFIGURE:
  ServerConfigurePayload ServerConfigCmd;
 case SERVER_KICK_USER:
  void;
 case SERVER_LOGS:
  ServerLogsPayload ServerLogsCmd;
 case SERVER_MANAGE_BANS:
  ServerManageBansPayload ServerBansCmd;
 case SERVER_SHUTDOWN:
  ServerShutdownPayload ServerShutdownCmd;
 case SERVER_VIEW_STATS:
  ServerStatsPayload ServerStatsCmd;
 case USER_CREATE:
  UserCreatePayload UserCreateCmd;
 case USER_DELETE:
  UserDeletePayload UserDeleteCmd;
 case USER_LIST:
  UserListPayload UserListCmd;
 case USER_PERMISSIONS:
  UserPermissionsPayload UserPermissionsCmd;
 case USER_RENAME:
  UserRenamePayload UserRenameCmd;
 default:
  VendorDefined Vendor;
};
/**
 * A CMDReply payload is one of these types.
 * With Cmd set to a CMD value.
 */
union OneReply switch (CMD_e Cmd) {
 case AUTHANONYMOUS:
  void; /* Replies with CAPABILITY_PRE or CAPABILITY_POST */
 case AUTHCERT_TLS:
  AuthCertTlsReplyPayload AuthCertTlsCmd;
 case AUTHCERT_USER:
  AuthCertUserReplyPayload AuthCertUserCmd;
 case AUTHMD5:
  void;
 case BYE:
  ByeReplyPayload ByeCmd;
 case FILE_CREATE:
  FileCreateReplyPayload FileCreateCmd;
 case FILE_COPY:
  FileCopyReplyPayload FileCopyCmd;
 case FILE_DELETE:
  FileDeleteReplyPayload FileDeleteCmd;
 case FILE_RENAME:
  FileRenameReplyPayload FileRenameCmd;
 case FILE_METADATA:
  FileMetaDataReplyPayload FileMetaDataCmd;
 case FILE_MOVE:
  FileMoveReplyPayload FileMoveCmd;
 case FILE_SHARE:
  FileShareReplyPayload FileShareCmd;
 case FILE_GET:
  FileGetReplyPayload FileGetCmd;
 case FILE_MODIFY:
  FileModifyReplyPayload FileModifyCmd;
 case FOLDER_CREATE:
  FolderCreateReplyPayload FolderCreateCmd;
 case FOLDER_COPY:
  FolderCopyReplyPayload FolderCopyCmd;
 case FOLDER_DELETE:
  FolderDeleteReplyPayload FolderDeleteCmd;
 case FOLDER_RENAME:
  FolderRenameReplyPayload FolderRenameCmd;
 case FOLDER_METADATA:
  FolderMetaDataReplyPayload FolderMetaDataCmd;
 case FOLDER_MOVE:
  FolderMoveReplyPayload FolderMoveCmd;
 case FOLDER_OPEN:
  FolderOpenReplyPayload FolderOpenCmd;
 case FOLDER_SHARE:
  FolderShareReplyPayload FolderShareCmd;
 case FOLDER_LIST:
  FolderListReplyPayload FolderListCmd;
 case RESERVED_CMD:
  void;
 case SERVER_CONFIGURE:
  ServerConfigureReplyPayload ServerConfigCmd;
 case SERVER_LOGS:
  ServerLogsReplyPayload ServerLogsCmd;
 case SERVER_MANAGE_BANS:
  ServerManageBansReplyPayload ServerManageBansCmd;
 case SERVER_SHUTDOWN:
  ServerShutdownReplyPayload ServerShutdownCmd;
 case SERVER_VIEW_STATS:
  ServerStatsReplyPayload ServerViewStatsCmd;
 case USER_CREATE:
  UserCreateReplyPayload UserCreateCmd;
 case USER_DELETE:
  UserDeleteReplyPayload UserDeleteCmd;
 case USER_LIST:
  UserListReplyPayload UserListCmd;
 case USER_PERMISSIONS:
  UserPermissionsReplyPayload UserPermissionsCmd;
 case USER_RENAME:
  UserRenameReplyPayload UserRenameCmd;
 default:
  VendorDefined Vendor;
};
/*
 * One command.
 */
```

```
struct Command {
 SEQ_t Sequence;
 OneCommand Payload;
};
/*
 * One Reply
 */
struct CommandReply {
 SEQ_t Sequence;
 OneReply Payload;
};
/**
 * A packet body.
 */
struct PacketBody {
  Command Commands<>; /* XDR arrays start with the Length */
};
/**
 * A packet reply.
 */
struct PacketReply {
  CommandReply Commands<>; /* XDR arrays start with the Length */
};
```

Figure 13: Packet Body CBOR

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 CMD.

All of their CBOR API is: cbor_CMD().

Name	CMD	Capability Description.	Command Description.
	CMD	An Administrative Operation Identifier.	Holds the (VENDOR_BIT or PHOENIX_BIT) value and one of ADMIN commands described in this section.
SERVER_CONFIGURE	%x05:8	May configure the server. A reply may have zero to many of the reply values set to READ_ONLY indicating the client may not alter them	The command to view and alter the server configuration information.

Name	CMD	Capability Description.	Command Description.
SERVER_KICK_USER	%x06:8	logs out a user. And may 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.
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

5.2.3. 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	CBOR API
Op	Indicates if the key/value pairs are to get, set, update, or be deleted.	cbor_Op()
ConfigSet	A set of key/value pairs with the same Op. Allows the client to bundle multiple key/pairs per Op.	cbor_ConfigSet()
ServerConfigure	The ADMIN operation that sets and gets server configuration information.	cbor_ServerConfigure();

Table 7: SERVER_CONFIGURE - ABNF/CBOR Mapping

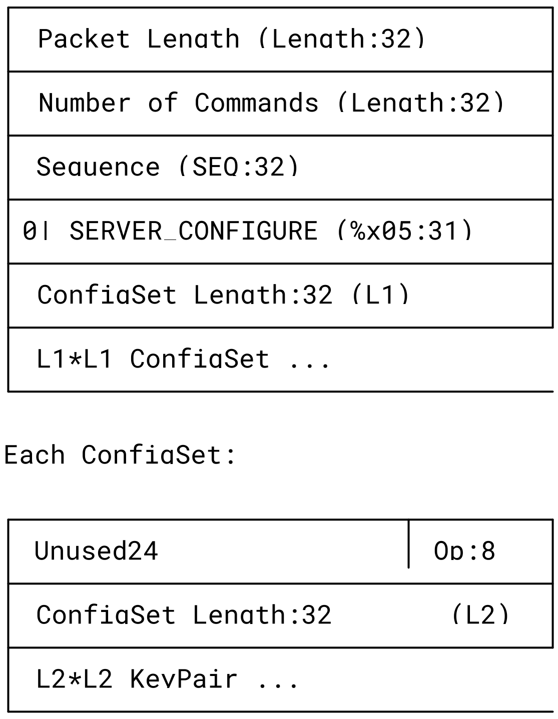

Figure 14: SERVER_CONFIGURE Command

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 15: SERVER_CONFIGURE - ABNF

5.2.3.1.2. SERVER_CONFIGURE - CBOR

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

Figure 16: SERVER_CONFIGURE - CBOR

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.

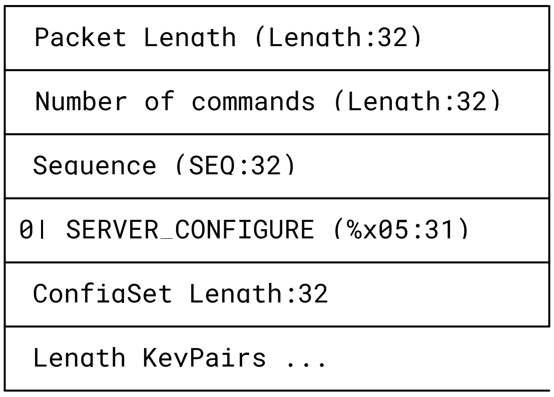
Packet Lenath (Lenath:32)
Number of commands (Lenath:32)
Sequence (SEQ:32)
0| SERVER_CONFIGURE (%x05:31)
ConfiaSet Lenath:32
Lenath KevPairs ...

Figure 17: SERVER_CONFIGURE Reply

5.2.3.2.1. SERVER_CONFIGURE Reply - ABNF

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

Figure 18: SERVER_CONFIGURE - ABNF

5.2.3.2.2. SERVER_CONFIGURE Reply - CBOR

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

Figure 19: SERVER_CONFIGURE - CBOR

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.

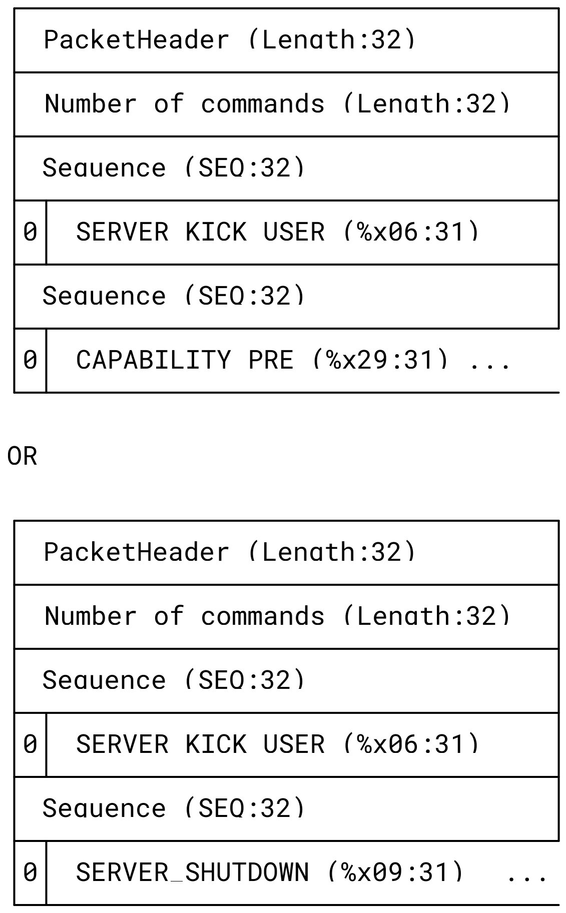
PacketHeader (Length:32)
Number of commands (Length:32)
Sequence (SEQ:32)
0
SERVER_KICK_USER (%x06:31)
Sequence (SEQ:32)
0
CAPABILITY_PRE (%x29:31) ...
OR
PacketHeader (Length:32)
Number of commands (Length:32)
Sequence (SEQ:32)
0
SERVER_KICK_USER (%x06:31)
Sequence (SEQ:32)
0
SERVER_SHUTDOWN (%x09:31) ...

Figure 20: SERVER_KICK_USER

5.2.4.1. SERVER_KICK_USER - ABNF

```
ServerKickPayload  = PHOENIX_BIT:1 SERVER_KICK_PAYLOAD:31
```

Figure 21: SERVER_KICK_USER - ABNF

5.2.4.2. SERVER_KICK_USER - CBOR

```
struct ServerKickPayload
{
 /*
         * With Kick set to SERVER_KICK_USER.
         * And the VENDOR_BIT not set.
         */
 CMD_e Kick;
};
```

Figure 22: SERVER_KICK_USER - CBOR

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	API
Category	Category	cbor_Category

Table 9

5.2.5.1. XXXX - ABNF

```
xxx
```

Figure 23: xxxx

5.2.5.2. XXXX - CBOR

```
xxx
```

Figure 24: xxxx

5.2.6. Administration - SERVER_MANANGE_BANS

The SERVERM_MANAGE_BANS command allows administrators to ban, or block, specific users hosts, IP addresses, IP address ranges networks.

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. This would be done on separate connection.

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 administrative capabilities.
- Some user accounts can be administrative only, and are limited to CAPABILITY_POST actions that only include administrative 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 administrative 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 process

The client initiates the authentication process. When the client makes a connection to a server it takes one of two paths, depending on [Figure 25 (A)]:

- If it has never authenticated to this server using the current account.[Figure 25 (L)]:
- If its last connection had a successful authentication to this server using the current account. [Figure 25 (B)]:

Part of the authentication process is determining which authentication process a server requires. So upon initial connection the client sends a CAPABILITY_PRE packet that includes all of the authentication methods it is willing to use. It is an ordered list with the more desirable ones at the front of the list, and the least desirable ones at the end if the list. And if the client wishes to do any vendor specific operations, then it must also include the VENDOR_ID string in the initial packet.

Similarly, the CAPABILITY_PRE packet sent from the server includes the same information. The server is the authentication mechanism authority.

When a client sends a VENDOR_ID command in its CAPABILITY_PRE command, then the server MUST reply back in its CAPABILITY_PRE reply, its VENDOR_ID (It is valid for it to be empty). (See VendorID (Section 5.14))

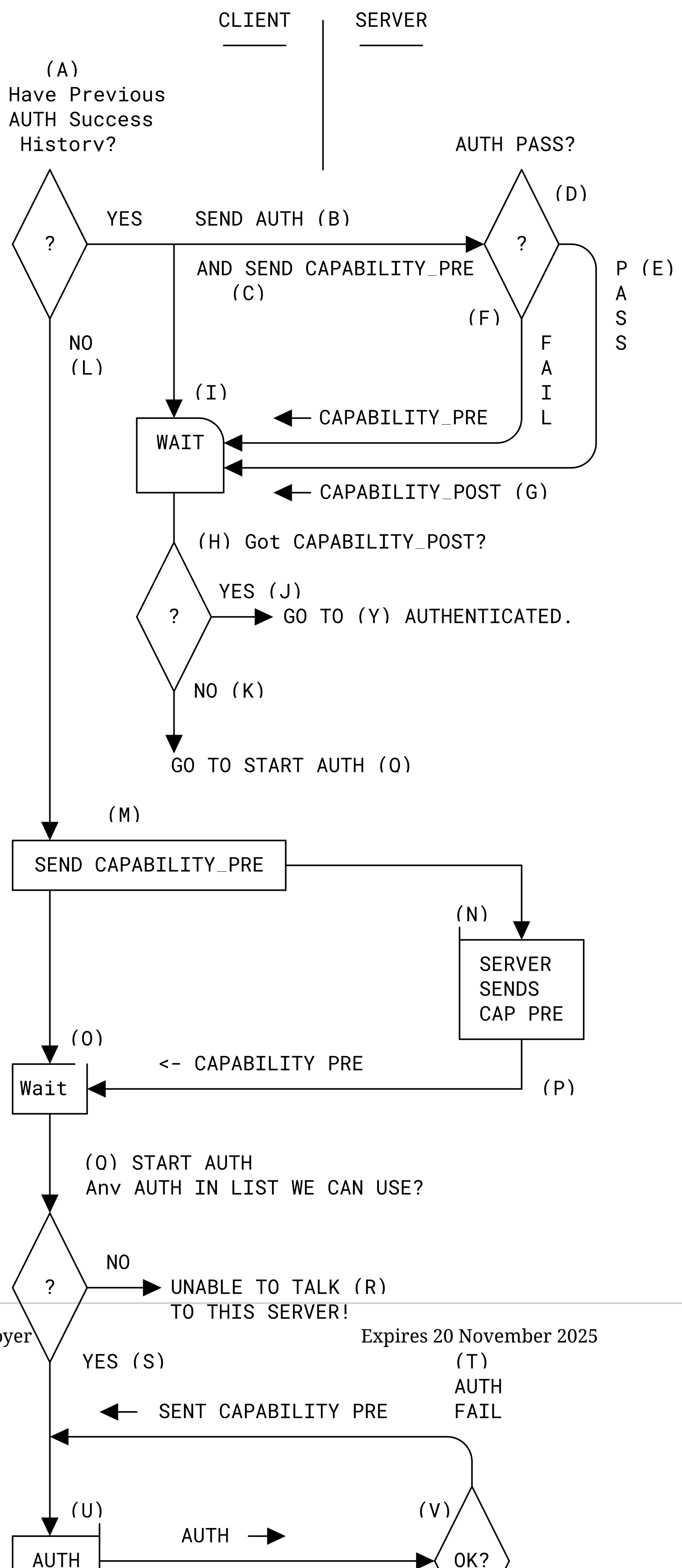
CLIENT
SERVER
(A)
Have Previous
AUTH Success
History?
AUTH PASS?
?
YES
SEND AUTH (B)
?
(D)
AND SEND CAPABILITY_PRE
(C)
P (E)
A
S
S
(F)
NO
(L)
F
A
I
L
(I)
CAPABILITY_PRE
WAIT
CAPABILITY_POST (G)
(H) Got CAPABILITY_POST?
YES (J)
?
GO TO (Y) AUTHENTICATED.
NO (K)
GO TO START AUTH (Q)
(M)
SEND CAPABILITY_PRE
(N)
SERVER
SENDS
CAP PRE
(O)
<- CAPABILITY PRE
Wait
(P)
(Q) START AUTH
Any AUTH IN LIST WE CAN USE?
NO
?
UNABLE TO TALK (R)
TO THIS SERVER!
YES (S)
(T)
AUTH
FAIL
SENT CAPABILITY PRE
(U)
(V)
AUTH
AUTH
OK?

Figure 25: Authentication Overview

5.3.2. Authenticating With Successful History

When a client has had a successful connection to the server using the current client login name, then the client sends the AUTH command and appends as a second command in the same packet, the clients CAPABILITY_PRE command as shown in Figure 25 (B & C). Then the client waits for a server reply at Figure 25 (I):

When the server gets this dual command packet Figure 25 at (D) and attempts the authentication process. If the authentication passes,[Figure 25 (E)]: the server sends a CAPABILITY_POST [Figure 25 (G)] command to the client as the okay reply.

If the authentication fails, then the server sends the client its first CAPABILITY_PRE command. [Figure 25 (F)]

The client waits for a reply from the server at Figure 25 (I).

When the client gets a CAPABILITY_PRE [Figure 25 (F)] it knows the authentication failed, so the client goes to start a normal authentication process [Figure 25 (K) & (Q)]

If the received command was a CAPABILITY_POST [Figure 25 (G)], then the authentication passed and the client is authenticated. And starts any post authentication work. [Figure 25 (J & Y)]

5.3.3. Authenticating With No History

When there is not successful history between the client and server, the authentication process starts at [Figure 25 (M)] with the client sending its CAPABILITY_PRE command to the server. Then waits for a server reply [Figure 25 (O)]

As soon as the server gets its authentication CAPABILITY_PRE at [Figure 25 (N)], the server evaluates the contents of the CAPABILITY_PRE from the client, matches it to the servers configured ability, and sends a CAPABILITY_PRE packet back to the client. [Figure 25 (P)] This CAPABILITY_PRE packet contains the authentication method(s) that the client must use to authenticate with the server. A CAPABILITY_PRE command from the server is only sent after receiving one from the client. So the server only sends the common authentication method using a priority configured into the server.

The servers CAPABILITY_PRE command may contain zero or more authentication methods. When more than one, all must be used. For example, the server could require a client certificate (AUTHCERT_TLS), and a user MD5 login (AUTHMD5). So both would be sent back, in this example. When the servers CAPABILITY_PRE command does not have any authentication methods, the server is telling the client that no authentication is possible. When this happens the server then terminates the connection. And the client then terminates the connection.

5.3.4. 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. And includes the supported authentication methods.

After too many retries, the client or server may terminate the connection.

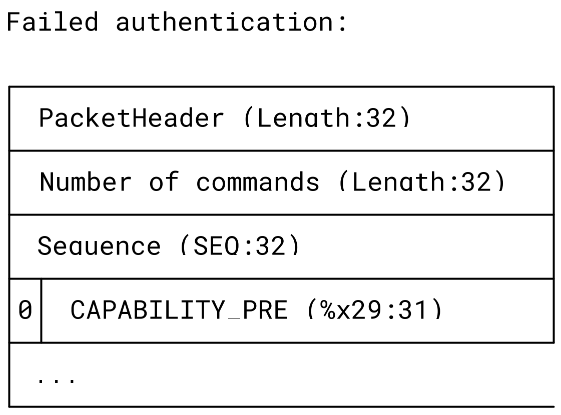

Figure 26: AUTHCERT_USER

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.

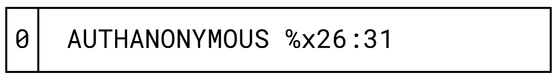

Figure 27: Capability - AUTHANONYMOUS

5.4.2. Authentication - AUTHANONYMOUS - Command

Once the connection is made the client sends its CAPABILITY_PRE list, or an authentication to the server..

If the client has already had a relationship with the server, then the client may send the AUTHANONYMOUS command to the server. 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 CAPABILITY_PRE packet with the same sequence number the same as in the AUTHANONYMOUS login request. And include the authorized authentication methods.

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. The client may then send any CAPABILITY_POST items to the server.

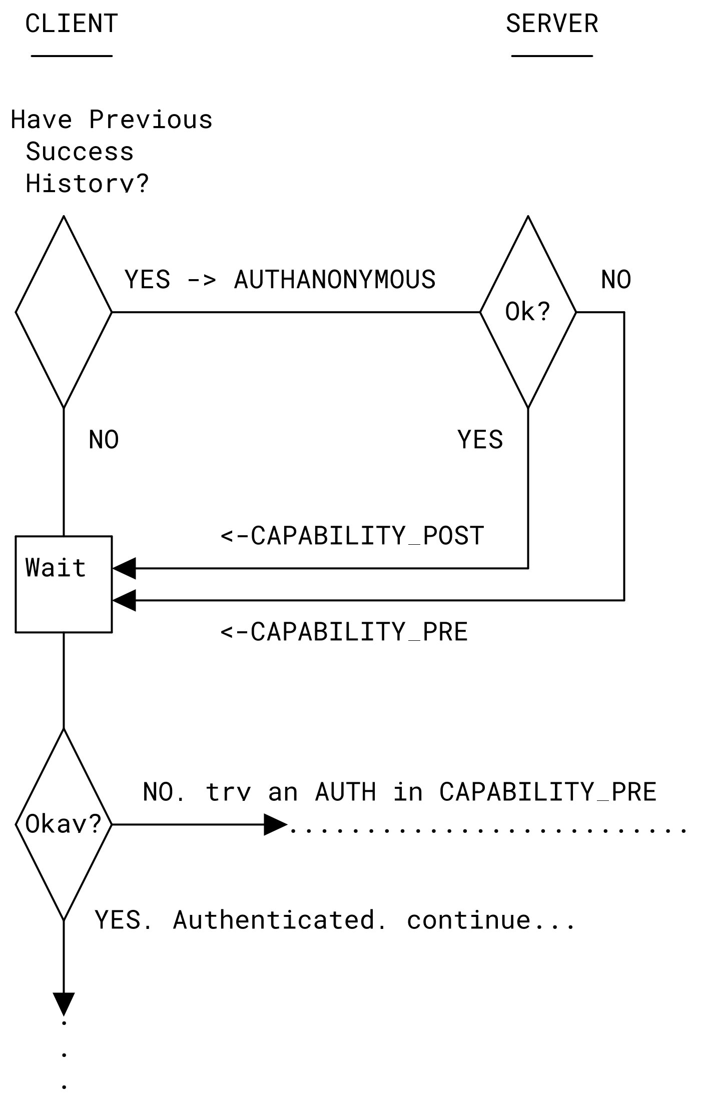
CLIENT
SERVER
Have Previous Success Historv?
YES -> AUTHANONYMOUS
Ok?
NO
NO
YES
<-CAPABILITY_POST
Wait
<-CAPABILITY_PRE
NO. trv an AUTH in CAPABILITY_PRE
Okav?
YES. Authenticated. continue...

Figure 28: AUTHANONYMOUS - Login Flow

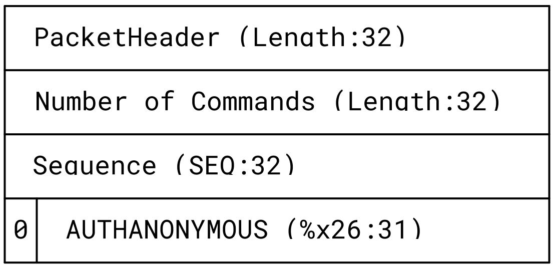

Figure 29: Capability - AUTHANONYMOUS

5.4.2.1. Authentication - ANONYMOUS - ABNF

```
AuthAnonymous  = PHOENIX_BIT:1 AUTHANONYMOUS:31
```

Figure 30: Authentication - ANONYMOUS - ABNF

5.4.2.2. Authentication - ANONYMOUS - CBOR

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

Figure 31: Authentication - ANONYMOUS - CBOR

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.

0	AUTHMD5 %x10:31

Figure 32: 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.

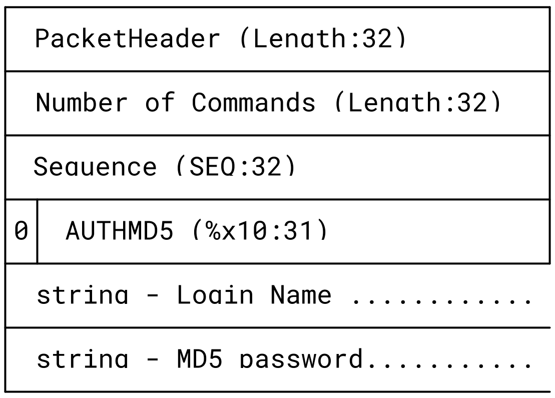
PacketHeader (Length:32)
Number of Commands (Length:32)
Sequence (SEQ:32)
0
AUTHMD5 (%x10:31)
string - Login Name ....
string - MD5 password..........

Figure 33: Capability - AUTHMD5

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

5.6.2.1. Authentication - MD5 - ABNF

```
Login       = string

Md5Password = string

AuthMD5     = PHOENIX_BIT:1 AUTHMD5:31 Login Md5Password
```

Figure 34: Authentication - MD5 - ABNF

5.6.2.2. Authentication - MD5 - CBOR

```
struct AuthMD5Payload
{
 string AccountName<>;
 string Md5Password<>;
};
```

Figure 35: Authentication - MD5 - CBOR

5.7. Calendar Commands Summary

These command are based on iCalendar and iTIp.

5.8. Capability Commands Summary

The purpose of this protocol is to facilitate the transfer of MIME objects, not to define how they are used. Capabilities allow each endpoint to ensure the other endpoint is capable of transferring the desired content and optionally allow control of the other endpoint.

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 unsigned 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 the Capability Index (Appendix C) for a complete list in this specification.

These are not a negotiation. Each sends their abilities to the other.

Capabilities from the server are sent once or twice. Optionally one before the user is authenticated (CAPABILITY_PRE), and once after (CAPABILITY_POST). If a user logs out and stays connected, then the process starts over with the server assuming a new client just connected.

Capabilities from the client may be sent, once, or twice per authentication process. The client sends a CAPABILITY_PRE with its connection to a server. 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.

NOTE: Vendor specific capabilities MUST include a Length value after the capability value. This is because there is no way a non compatible implementation could calculate the length of the data that would follow it in order to find the next capability or command in the packet.

Table Table 10 lists some CAPABILITY_PRE capabilities.

Some capabilities have data associated with them, others do not.

5.8.1. Capability - CAPABILITY_PRE

Pre authentication capabilities are sent before authentication.

When the client connects to a server it always sends a CAPABILITY_PRE as soon as the connection is established. This packet contains all of the authentication methods supported by the client. It is an ordered list with the most preferred at the start of the list and the lesser preferred at the end of the list.

The CAPABILITY_PRE that the client sends to the server in this specification includes an optional VENDOR_ID. Vendors may add new client to server capabilities as long as they set the VENDOR_BIT in the command and are implemented to understand that not all server will understand their vendor specific extensions.

xxx

Name	Value	Value Type	Description
AUTHANONYMOUS	%x26:31		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		Authenticate by providing an account name and an MD5 password.
AUTHCERT_TLS	%x27:31	CMD_e	Authenticate using the connections TLS certificate.
AUTHCERT_USER	%x28:31	CMD_e	Like AUTHMD5, an account and password are provided in the payload and they are encrypted with a prearranged certificate. The server must already have the accounts public certificate.
VENDOR_ID	%x12:31	string	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 or other vendor specific abilities. The specific string value is determined by the server implementation and is out of scope for this specification.

Table 10: Capabilities - CAPABILITY_PRE

5.8.2. Capability - CAPABILITY_POST

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

A client or server may or might not also send an additional CAPABILITY_POST command as account permissions change.

For example after authentication an administrator could give the current user more permissions. At that time the server would send a new CAPABILITY_POST to the client. Or perhaps the client needs to update the server with a new VENDOR_ID and CAPABILITY_POST after a license key is installed. Or perhaps remove capabilities after business hours.

Name	Value	Description
x	x1	x2

Table 11: Capabilities - CAPABILITY_POST

5.9. EMail Commands Summary

These commands allow for the fetching and submission of EMail messages

5.10. File and Folder Commands Summary

Remote systems have files and folders that are accessible. And have files and folders that are not accessible. This section covers only these files and folders for which the authenticated users has at least read only access to full access and control.

Read only files and folders could be historical archives, news, email, calendar, or marketing information. Anything that the owner of the files and folders wants to share and keep unaltered.

The file operations (FileOp) here are a descriptive category of several over the wire commands, there is no 'FileOp' command. Implementations are not required to support any or all of these commands.

Folders might be nested. That is one folder may contain one or more folders. Each of which may contain folders. Much like a computer folder structure. Only folders and files that are read-only or are read-write are returned in a list to the client.

Each file and folder has an ID. A FOLDER_LIST command returns a list of files and folders, each with their ID. Most FileOps use this ID and not the name. These IDs are guaranteed to not change during a session. The server should attempt to preserve IDs across sessions.

Op Name	Value in hex	Brief Description.
FOLDER_CAPABILITY Section 5.10.1	%x13	When sent as a command, request the list of folder commands supported. When FOLDER_CAPABILITY is sent as a reply, this includes a CBOR type 4 array of folder commands supported along with any of their optional or mandatory parameters. Only when an administrator changes the users access while a client is active will this list change during a session. The client will be notified of any changes. See Notifications (Section 7).
FOLDER_CREATE Section 5.10.2	%x14	Create a new folder. The reply will be success and folder ID, or a reason the creation was denied. Also the name of the capability for this permission. This indicates that user has the capability to create folders. It does not guarantee they can create a specific folder or folder hierarchy.
FOLDER_COPY Section 5.10.3	%x15	Copy a folder. The reply will be success or a reason the copy was denied. Also the name of the capability for this permission. This indicates that user has the capability to copy folders. It does not guarantee they can copy any specific folder to any specific location available to the user.
FOLDER_DELETE Section 5.10.4	%x16	Delete a folder. The reply will be success and new folder ID, or a reason the delete was denied. Also the name of the capability for this permission. This indicates that user has the capability to delete folders. It does not guarantee they can delete all folders for which they have access
FOLDER_RENAME Section 5.10.5	%x17	Rename a folder. The reply will be success or a reason the rename was denied. The folder ID will not be changed. Also the name of the capability for this permission. This indicates that user has the capability to rename folders. It does not guarantee they can rename any specific folder for which the user has access.

Op Name	Value in hex	Brief Description.
FOLDER_METADATA Section 5.10.6	%x18	Get, set, and update information associated with the folder by ID. The reply will be success and metadata permissions, or a reason the operation was denied. All folders have a meta data object associated with them, and the list may be empty. Some file meta data is also returned with the FOLDER_OPEN command. This is not a CAPABILITY request or reply.
FOLDER_MOVE Section 5.10.7	%x19	Move a folder. The reply will be success or a reason the move was denied. The folder ID will not be changed. Also the name of the capability for this permission. This indicates that user has the capability to move folders. It does not guarantee they can move any specific folder for which the user has access.
FOLDER_OPEN Section 5.10.8	%x1a	Open an existing folder and get information about the folder, its ID, and files in the folder. The reply will be success or a reason the open was denied. All users can open for at least read only access any folder they they were provided in a FOLDER_LIST reply. This is not a CAPABILITY reply or command.
FOLDER_SHARE Section 5.10.10	%x1b	Share a folder. Share to a list or anyone. The reply will be a success or a reason the share was denied. Also the name of the capability for this permission. This indicates that user has the capability to share folders. It does not guarantee they can share any specific folder for which the user has access.

Op Name	Value in hex	Brief Description.
FOLDER_LIST Section 5.10.9	%x1c	List folders and optional files available to the user. The command may include zero or more folders ID's. The top most folder has no name, and is an empty string. The top most folder is the default folder name. SEE XREF-TODO. When provided, the reply includes a list of files within the folder. The content of each index entry depends on the type of file. SEE XREF-TODO. All clients may issue a FOLDER_LIST command. The result could be empty. This is not a CAPABILITY command or reply. Only shared folders have their contents altered during a session without the client initiating the changes. The client is notified of shared folder changes to folders and files at runtime that were not initiated by the client. See Section 7
FILE_CREATE Section 5.10.11	%x1d	Create a new file. The reply will be success and a new file ID, or a reason the create was denied. Also the name of the capability for this permission. This indicates that user has the capability to create files. It does not guarantee they can create any file for which they have folder access.
FILE_COPY Section 5.10.12	%x1e	Copy a file. The reply will be success and the new file ID, or a reason the copy was denied. Also the name of the capability for this permission. This indicates that user has the capability to copy files. It does not guarantee they can copy any file for which they have folder access.
FILE_DELETE Section 5.10.13	%x1f	Delete a file. The reply will be success or a reason the delete was denied. Also the name of the capability for this permission. This indicates that user has the capability to delete files. It does not guarantee they can delete any file for which they have folder access.
FILE_RENAME Section 5.10.14	%x20	Rename a file. The reply will be success or a reason the rename was denied. The file ID will not change. Also the name of the capability for this permission. This indicates that user has the capability to rename files. It does not guarantee they can rename any file for which they have folder access.

Op Name	Value in hex	Brief Description.
FILE_METADATA Section 5.10.15	%x21	Get, set, and update information associated with the file. The reply will be success or a reason the operation was denied. All files have a meta data object associated with them, and the list may be empty. Some file meta data is also returned with the FOLDER_OPEN command. This is not a CAPABILITY request or reply. All users have access to the meta data for files they have access to. The list of meta data returned may be restricted depending on the authenticated user.
FILE_MOVE Section 5.10.17	%x22	Move a file. The reply will be success or a reason the move was denied. The file ID will not change. Also the name of the capability for this permission. This indicates that user has the capability to move files. It does not guarantee they can move any file for which they have folder access.
FILE_SHARE Section 5.10.18	%x23	Share a file. The reply will be success or a reason the share was denied. Also the name of the capability for this permission. This indicates that user has the capability to share files. It does not guarantee they can share any file for which they have folder access.
FILE_GET Section 5.10.19	%x24	Get file contents. All files provided in FOLDER_LIST command are available to the client and may be read only. The reply will be the file contents, with optional index information for MIME objects. Or a reason for the failure. This is not a CAPABILITY request or reply. All users may get the file contents for any file provided to the in the FOLDER_LIST reply.

Op Name	Value in hex	Brief Description.
FILE_MODIFY Section 5.10.16	%x25	Modify the contents of an existing file. The reply will be success or a reason the modify was denied. The reply will be a success with the contents updated, or a reason for the denial. All client may issue FILE_MODIFY commands. The contents of the folder index might have a read only attribute for any file. A user may not modify the contents of read only files. This is not a CAPABILITY request or reply.

Table 12: File and Folder Command List

5.10.1. Folder - FOLDER_CAPABILITY

As part of the successful authentication reply, the server sends the client a list of FileOps allowed by the client. This list includes zero or more folder capabilities the authenticated client is allowed to issue.

The only time this list changes is if an administrator alters the authenticated client access while the session is active. The client will be notified of any changes. See Notifications (Section 7). Unless the client receives such a notification, there is no need for the client to reissue a FOLDER_CAPABILITY request.

5.10.1.1. FOLDER_CAPABILITY Command (%x13) - Request

A FOLDER_CAPABILITY request does not have any parameters. It is a request for all of the folder commands available to the authenticated user

5.10.1.1.1. FOLDER_CAPABILITY Request ABNF

This is transmitted as a CBOR major type 0, unsigned integer with a hex value of %x13.

```
FOLDER_CAPABILITY_REQUEST:8 = %13:8
```

5.10.1.1.2. FOLDER_CAPABILITY Request Example

A FOLDER_CAPABILITY request has the vendor bit set to zero. This is transmitted as a CBOR major type 0, unsigned integer with a hex value of %x13. Figure 36 is a Phoenix packet payload content value for making a FOLDER_CAPABILITY request. This is in the payload part of a Phoenix packet.

```
CBOR Binary Bit Value:

+---------------+
|0|0|0|1|0|0|1|1|
+---------------+
```

Figure 36: FOLDER_CAPABILITY over the wire

5.10.1.2. FOLDER_CAPABILITY Reply (%x13)

The reply is in a Phoenix packet reply payload. The reply is a CBOR major type 4 array. Each element of the array is a FOLDER_CAPABILITY with any optional or mandatory parameters.

Figure 37 is an example of a session with only read-only access. In this example, the user can open folders, list folders, get the meta data for folders, get the metadata for files, and get files, and one vendor specific capability.

```
CBOR Binary Bit values:

An ARRAY and it SIZE.
(Array 0x80, size 8 entries)
A CBOR array with a value of 8 entries.
+---------------+
|1|0|0|0|1|0|0|0| (%x88)
+---------------+

Entry 1 the
FOLDER_OPEN capability. (0x1a)
A CBOR unisgned int with a value of %x1a.
+---------------+
|0|0|0|1|1|0|0|0| A uint8_t value follows.
+---------------+
|0|0|0|1|1|0|1|0| FOLDER_OPEN (%x1a)
+---------------+

Entry 2 the
FOLDER_LIST capability (0x1c)
A CBOR unisgned int with a value of %x1c.
+---------------+
|0|0|0|1|1|0|0|0| A uint8_t value follows.
+---------------+
|0|0|0|1|1|1|0|0| FOLDER_LIST (%x1c)
+---------------+

Entry 3 the
FOLDER_METADATA capability (0x18)
A CBOR unisgned int with a value of %x18.
+---------------+
|0|0|0|1|1|0|0|0| A uint8_t value follows.
+---------------+
|0|0|0|1|1|0|0|0| FOLDER_METADATA (%x18)
+---------------+

Entry 4 the
FOLDER_METADATA_PARAMETERS
A CBOR unisgned int with a value of %x02.
+---------------+
|0|0|0|0|0|0|1|0| The Get bit set.
+---------------+

Entry 5 the
FILE_METADATA capability (0x21)
A CBOR unisgned int with a value of %x21.
+---------------+
|0|0|0|1|1|0|0|0| A uint8_t value follows.
+---------------+
|0|0|1|0|0|0|0|1| FOLDER_METADATA (%x21)
+---------------+

Entry 6 the
FILE_METADATA_PARAMETERS
A CBOR unisgned int with a value of %x02.
+---------------+
|0|0|0|0|0|0|1|0| The Get bit set.
+---------------+

Entry 7 the
FILE_GET capability (%x24)
A CBOR unisgned int with a value of %x21.
+---------------+
|0|0|0|1|1|0|0|0| A uint8_t value follows.
+---------------+
|0|0|1|0|0|0|0|1| FOLDER_METADATA (%x21)
+---------------+

Entry 8
A Vendor specific capability (%x80000002)
A CBOR unisgned int.
The vendor bit is set (the high bit),
plus the value (%x02) results in (%x80000002).
The last zero part is because ALL vendor commands
MUST include the size of any related data,
even when zero.
+---------------+
|0|0|0|1|1|0|1|0| A uint32_t value follows.
+---------------+---------------+---------------+---------------+
|1|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|1|0|
+---------------+---------------+---------------+---------------+
|0|0|0|0|0|0|0|0| A CBOR unsigned int 0 (size).
+---------------+
```

Figure 37

The example FOLDER_CAPABILITY reply in Figure 37 contains the folder commands the client may issue:

```
ExampleCapabilityRepy
        = CBOR_FIXED_SIZE_ARRAY with 8 entries
          FOLDER_OPEN
          FOLDER_LIST
          FOLDER_METADATA FOLDER_METATDATA_PARAMETERS
          FILE_METADATA FILE_METADATA_PARAMETERS
          FILE_GET
          A VENDOR specific command
```

5.10.1.2.1. CBOR

```
// The FOLDER_METADATA and FILE_METADATA
// capabilities return the OpPermissions
// allowed in the session.
//
// Four bits, each one set or not set.
// When the bit is set, the client
// is allowed to perform the operation
// on the meta data (or not when not set).
//
// NOTE: These are operations on the
// meta data, not the file or folder data.
//
BitMask:4 OpPermission {
 OpSetBitMask     = 0x01,
 OpGetBitMask     = 0x02,
 OpUpdateBitMask  = 0x04,
 OpDeleteBitMask  = 0x08
};

// Each entry in FOLDER_CAPABILITY reply is
// one of these enumerated REPLY values.
//
enum FolderCapabilityReply_e {
  FOLDER_CREATE_REPLY = FOLDER_CREATE,
  FOLDER_COPY_REPLY = FOLDER_COPY,
  FOLDER_RENAME_REPLY = FOLDER_RENAME,
  FOLDER_METADATA_REPLY = FOLDER_METADATA,
  FOLDER_MOVE_REPLY = FOLDER_MOVE,
  FOLDER_OPEN_REPLY = FOLDER_OPEN,
  FOLDER_SHARE_REPLY = FOLDER_SHARE,
  FOLDER_LIST_REPLY = FOLDER_LIST,
  FILE_CREATE_REPLY = FILE_CREATE,
  FILE_COPY_REPLY = FILE_COPY,
  FILE_DELETE_REPLY = FILE_DELETE,
  FILE_RENAME_REPLY = FILE_RENAME,
  FILE_METADATA_REPLY = FILE_METADATA,
  FILE_MOVE_REPLY = FILE_MOVE,
  FILE_SHARE_REPLY = FILE_SHARE,
  FILE_GET_REPLY = FILE_GET,
  FILE_MODIFY_REPLY = FILE_MODIFY,
  VENDOR_FOLDER_REPLY = VENDOR_COMMAND
};

// In this version, each FOLDER CAPABILITY command REPLY
// has (1) no additional data (void), (2) or it
// has meta data permission values (OpPermissions).
//
union FolderCapabilityReplyEntry (FolderCapabiltyReply_e Cmd) {
  OpPermission  Permissions;
  void;
};

// The FOLDER_CAPABILITY reply:
// A variable list of capabilities in the reply.
// And is sent back as a CBOR major type 4 array.
//
```

```
struct FolderCapabilityReply
    FolderCapabilityReplyEntry Capabilities<>;
};
```

Figure 38: CBOR Definition

5.10.1.2.2. Code Example

```
/**
 * Example code to extract the data and process
 * the FOLDER_CAPABILITY reply.
 */
 void
 ProcessFileFolderCapabilityReply(Session & Session,
                                  PhoenixPacket & Packet)
 {
   PhoenixPayload * Payload = Packet.GetPayload();

   // Default to no supported capabilities.
   //
   Session->DisableAllCapabilities();

   if (Payload != nullptr && Payload.size() > 0) {
     Capability                  * OneCapability;
     Capability::const_iterator    It;

     for (It = Payload->cbegin(); It != Payload->cend(); It++) {
       OneCapability = *It;

       if (VendorBitSet(OneCapability) {
         Session.AddVendorCapability(OneCapability)

       } else {
           switch (OneCapability->Name) {

           case FOLDER_CREATE:
             Session->CanCreateFolders(true);
             break;

           case FOLDER_COPY:
             Session->CanCopyFolders(true);
             break;

           case FOLDER_DELETE:
             Session->CanDeleteFolders(true);
             break;

           case FOLDER_RENAME:
             Session->CanRenameFolders(true);
             break;

           case FOLDER_METADATA:
             OpPermissions Allowed = OneCapability->GetOpPermissions());

             if (Allowed & OpSetBitMask) {
               Session->CanSetFolderMetaData(true);
             }
             if (Allowed & OpGetBitMask) {
               Session->CanGetFolderMetaData(true);
             }
             if (Allowed & OpUpdateBitMask) {
               Session->CanUpdateFolderMetaData(true);
             }
             if (Allowed & OpDeleteBitMask) {
               Session->CanDeleteFolderMetaData(true);
             }
             break;

           case FOLDER_MOVE:
             Session->CanMoveFolders(true);
             break;

           case FOLDER_OPEN:
             Session->CanOpenFolders(true);
             break;

           case FOLDER_SHARE:
             Session->CanShareFolders(true);
             break;

           case FOLDER_LIST:
             Session->CanListFolders(true);
             break;

           case FILE_CREATE:
             Session->CanCreateFiles(true);
             break;

           caes FILE_COPY:
             Session->CanCopyFiles(true);
             break;

           case FILE_DELETE:
             Session->CanDeleteFiles(true);
             break;

           case FILE_RENAME:
             Session->CanRenameFiles(true);
             break;

           case FILE_METADATA:
             OpPermissions Allowed = OneCapability->GetOpPermissions());

             if (Allowed & OpSetBitMask) {
               Session->CanSetFileMetaData(true);
             }
             if (Allowed & OpGetBitMask) {
               Session->CanGetFileMetaData(true);
             }
             if (Allowed & OpUpdateBitMask) {
               Session->CanUpdateFileMetaData(true);
             }
             if (Allowed & OpDeleteBitMask) {
               Session->CanDeleteFileMetaData(true);
             }
             break;

           case FILE_MOVE:
             Session->CanMoveFiles(true);
             break;

           case FILE_SHARE:
             Session->CanShareFiles(true);
             break;

           case FILE_GET:
             Session->CanGetFiles(true);
             break;

           case FILE_MODIFY
             Session->CanModifyFiles(true);
             break;

           default:
             ProcessError("Unknown FOLDER_CAPABILITY reply.");
             break;
       }
     }
   }
 }
```

5.10.2. File and Folder - FOLDER_CREATE

5.10.3. File and Folder - FOLDER_COPY

5.10.4. File and Folder - FOLDER_DELETE

5.10.5. File and Folder - FOLDER_RENAME

5.10.6. File and Folder - FOLDER_METADATA

5.10.7. File and Folder - FOLDER_MOVE

5.10.8. File and Folder - FOLDER_OPEN

5.10.9. File and Folder - FOLDER_LIST

When a client successfully authenticates, part of the success reply is a list of folders the client has access to.

All changes to files, folder, or access not performed by the client will cause a notification to be sent to the client. (See Notifications (Section 7))

A FOLDER_LIST command returns an unordered array of File and Folder entries (FF entry). Each entry indicates if the item is a file or folder, its name, its ID (FFID), if it is shared with other users, any meta data, and if the user has read-only or read-write access

The meta data has two parts.

- Folder and File Meta data associated with the folder or file.
- User Meta data associated with the folder or file.

Folder and File Meta data contains persistent information about the folder or file. Such as its size, index information, and other persistent information important to the application.

User Meta data contains information about if the contents has been viewed and other application specific information unique to each user.

The FOLDER_LIST command has optional parameters. The default is to get just the file and folder index (FF).

- The client can ask for meta data for folders.
- The client can ask for meta data for files.
- The client can ask MIME index information for files.

5.10.9.1. A File and Folder (FF) entry

FF = Name FFID Shared:1 RW:1 string Name<>; uint64_t FFID; bits Shared:1 bits RW:1

5.10.9.2. FOLDER_LIST - Reply

5.10.10. File and Folder - FOLDER_SHARE

5.10.11. File and Folder - FILE_CREATE

5.10.12. File and Folder - FILE_COPY

5.10.13. File and Folder - FILE_DELETE

5.10.14. File and Folder - FILE_RENAME

5.10.15. File and Folder - FILE_METADATA

5.10.16. File and Folder - FILE_MOVE

5.10.17. File and Folder - FILE_MOVE

5.10.18. File and Folder - FILE_SHARE

5.10.19. File and Folder - FILE_GET

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.

A ping command must not be sent before a successful authentication.

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 CAPABILITY_POST 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. The server ignores the PING command.
- When the connection is not authenticated, The server ignores the PING command.
- 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. If the server is ignoring PING commands, a reply will never happen.

A client MUST NOT send a ping command more frequently than 90% of the SERVER_TIMEOUT value that the server sent in the CAPABILITY_POST command.

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

With servers that support PING, clients MUST NOT send any other command while wating for the PING reply.

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

5.14. Command - VENDOR_ID

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

When a server gets a VENDOR_ID command, 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 command to the server is expecting the possibility of receiving a SERVER_CONFIGURE capability back.

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

Over the life of the connection, the VENDOR_ID command and value can change and would need to be sent again. It might update the value after authentication. Or after some action has been performed.

For example, a server may perform one set of vendor specific operations during working hours, and a different set after hours. It may or might not also send an additional CAPABILITY_PRE command as account permissions change.

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. Notifications

TODO

8. 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].

8.1. 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.

8.2. 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.

8.3. 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.

8.4. 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.

8.5. 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.

8.6. 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.

8.7. 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.

8.8. 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.

8.9. 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.

8.10. 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.

User interfaces MUST NOT allow the MIME object to be copied. They MUST disable any copy/paste for the object in the user interface. The user interface may elect to display an indicator to the user that what they are viewing is read only.

8.11. 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.

8.12. 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.

8.13. 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.

8.14. 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.

8.15. 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.

8.16. 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.

8.17. 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.

8.17.1. MDNRecord

The format of an MDNRecord:

Name	CBOR 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.

Name	CBOR Type	Description
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

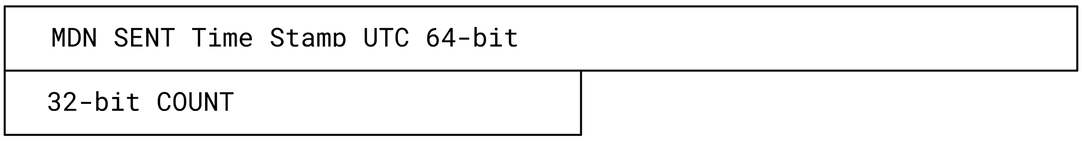

Figure 39: MDNRecord

8.17.1.1. 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
```

8.17.1.2. MDNRecord - CBOR

CBOR:

```
                ; 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.

8.17.2. MDNEntry

Name	CBOR 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.

Name	CBOR Type	Description
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/CBOR Mapping

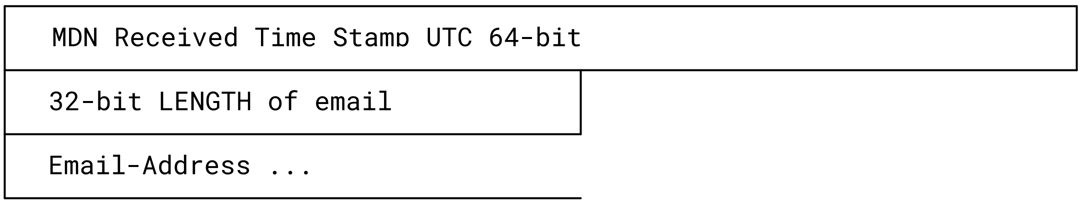

Figure 40: MDNEntry

8.17.2.1. MDNEntry ABNF

ABNF:

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

Figure 41: MDNEntry - ABNF

8.17.2.2. MDNEntry - CBOR

CBOR:

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

Figure 42: MDNEntry - CBOR

9. Index

9.1. 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.

9.1.1. 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;
```

9.1.2. 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
```

9.1.3. 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;
```

9.1.4. 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 43, shows the interisted header list prefix. This interisted header list prefix is followed by zero or more SingleEntry objects.

Name	CBOR Type	Description
IndexOP	uint8_t	One of IndexOpDefine or IndexOpUse
IndexOPDefine	uint8_t	Define or redefine a list.
IndexOPUse	uint8_t	Use an already defined list.
LID	uint8_t	LID is the list ID of the list that client is defining. With 255 reserved for expansion.
HDRCNT	uint32_t	HdrCnt is set by the client to the number of headers in the list. With %xffffff reserved for expansion.

Table 15

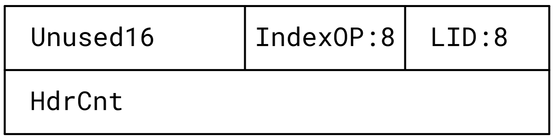

Figure 43: Interest Header List Prefix

ABNF:

```
HdrCnt          = uint32_t

Interest-Header = IndexOp LID HdrCnt
```

9.1.4.1. 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 44, where:

Name	CBOR 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

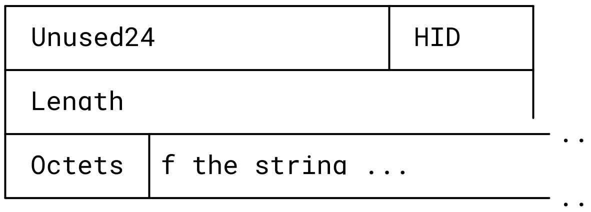

Figure 44: Setting the Interest List - Contents

ABNF:

```
SingleHeader = HID StringRef
```

9.1.4.2. 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.

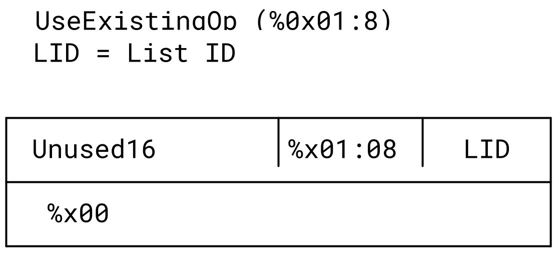

Figure 45: Using Existing Header Index by List ID (LID)

ABNF:

```
UseExistingOp      = %x01:8

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

9.1.4.3. 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.

```
      +------------------------------------+..
      | The folder open command ............
      +------------------------------------+..

   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 46: Example Setting a List

Where:

- (a): A 32-bit unsigned integer in network byte order as described in Figure 43.

  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 44.

  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 44.

  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'.

9.2. 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.

9.2.1. 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:

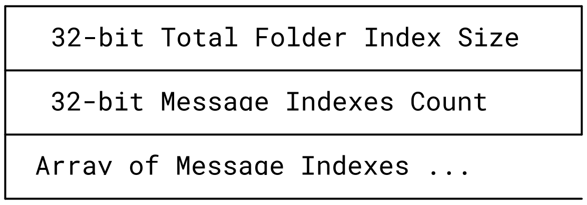
32-bit Total Folder Index Size
32-bit Message Indexes Count
Array of Message Indexes ...

Figure 47: 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:

9.2.2. 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.

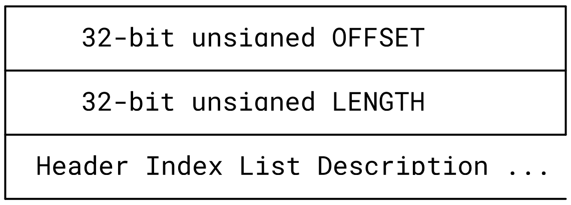

Figure 48: Message Index

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:

9.2.3. 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
```

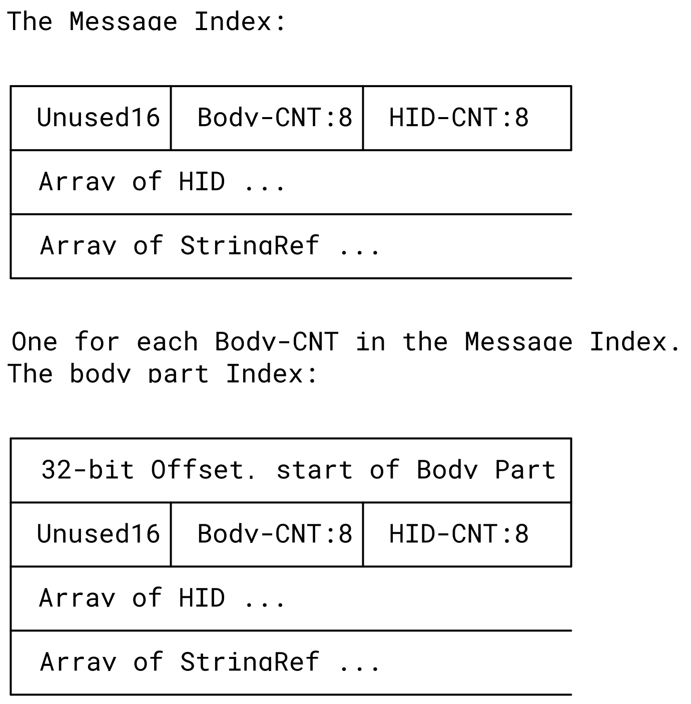

Figure 49: Header Index

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.

9.2.4. 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 50: Header Index

9.2.5. 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: multiplar/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 51: Header And Body Part Index

10. IANA Considerations

NOTE: This will be filled in with instructions and procedures to expand capabilities and commands.

11. Security Considerations

Robust digital certificate control. Especially with AUTHCERT_TLS.

MUCH MORE TODO ...

12. References

12.1. 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>.

[RFC8949] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, RFC 8949, DOI 10.17487/RFC8949, December 2020, <https://www.rfc-editor.org/info/rfc8949>.

[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>.

12.2. Informative References

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

Appendix A. Administrative Enumerated Binary Values

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

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

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

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

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

Decimal Value	Command / Capability Name	Brief Description.
x	USER_CERT	Manage a users certificate.
x	USER_CREATE	When sent in a capability reply USER_CREATE informs the user that they have permission to create users. When sent as a command the USER_CREATE instructs the other endpoint to create a named user.
x	USER_DELETE	Delete a user.
x	USER_LIST	List users and their capabilities.
x	USER_PERMISSIONS	Update user permissions.
x	USER_RENAME	Rename a user.

Decimal Value	Command / Capability Name	Brief Description.
x	USER_RESET	Used to coordinate resetting a users authentication information.
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
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

```
                 ; Any signed integer value
                 ;
int_t          = ; any positive or negative size or length

                 ; An 8-bit signed integer value
                 ;
int8_t         = -128 to 127

                 ; A 16-bit signed integer type
                 ;
int16_t        = 32,768 to 32,767

                 ; A 32-bit signed integer type
                 ;
int32_t        = 2,147,483,648 to 2,147,483,647

                 ; A 64-bit signed integer type
                 ;
int64_t        = -9,223,372,036,854,775,808
                   to 9,223,372,036,854,775,807

                 ; Any unsigned integer type
                 ;
uint_t         = ; any positive size or length

                 ; An 8-bit unsigned integer type
                 ;
uint8_t        = 0-255

                 ; A 16-bit unsigned integer type
                 ;
uint16_t       = 0-65535

                 ; A 32-bit unsigned integer type
                 ;
uint32_t       = 0-4294967295

                 ; A 64-bit unsigned integer type
                 ;
uint64_t       = 0-18446744073709551615

                 ;
cbor_int_t     = ; Any signed integer up to 64 bits.

                 ;
cbor_uint_t    = ; Any unsigned integer up to 64 bits.

                 ; 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.
                  ; nullptr indicates there is no value.
                  ;
KeyPair       = string (stringD | nullptr)

                  ; 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

                ; Boolean true.
                ;
true          = %x01:8

                ; Boolean false.
                ;
false         = %x00:8

                ;
enabled       = true   / false

                ; 8-bits set to zero.
                ; Used for padding larger data types.
                ;
unused8       = %x00:8

                ; 16-bits set to zero.
                ; Used for padding larger data types.
                ;
unused16      = %x00:16

                ; 24-bits set to zero.
                ; Used for padding larger data types.
                ;
unused24      = %x00:24
```

```
           ; A reference to the start and length of a object.
Ref     = Offset Length
```

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

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

OneCommand   = AUTHANONYMOUS AuthAnonymousPayload
             / AUTHCERT_TLS AuthCertTlsPayload
             / AUTHCERT_USER AuthCertUserPayload
             / AUTHMD5 AuthMD5Payload
             / AUTHMD5 AuthMD5ReplyPayload
             / CAPABILITY_PRE CapabilityPrePayload
             / FILE_CREATE FileCreatePayload
             / FILE_COPY FileCopyPayload
             / FILE_DELETE FileDeletePayload
             / FILE_RENAME FileRenamePayload
             / FILE_METADATA FileMetaDataPayload
             / FILE_MOVE FileMovePayload
             / FILE_SHARE FileSharePayload
             / FILE_GET FileGetPayload
             / FILE_MODIFY FileModifyPayload
             / FOLDER_CREATE FolderCreatePayload
             / FOLDER_COPY FolderCopyPayload
             / FOLDER_DELETE FolderDeletePayload
             / FOLDER_RENAME FolderRenamePayload
             / FOLDER_METADATA FolderMetaDataPayload
             / FOLDER_MOVE FolderMovePayload
             / FOLDER_OPEN FolderOpenPayload
             / FOLDER_SHARE FolderSharePayload
             / FOLDER_LIST FolderListPayload
             / NOT_SUPPORTED
             / SERVER_CONFIGURE ServerConfigurePayload
             / SERVER_KICK_USER
             / SERVER_LOGS ServerLogsPayload
             / SERVER_MANAGE_BANS ServerManageBansPayload
             / SERVER_SHUTDOWN ServerShutdownPayload
             / SERVER_VIEW_STATS ServerViewStatsPayload
             / USER_CREATE UserCreatePayload
             / USER_DELETE UserDeletePayload
             / USER_LIST UserListPayload
             / USER_PERMISSIONS UserPermissionsPayload
             / USER_RENAME UserRenamePayload
             / RESERVED_CMD

OneReply     = AUTHANONYMOUS AuthAnonymousReplyPayload
             / AUTHCERT_TLS AuthCertTlsReplyPayload
             / AUTHCERT_USER AuthCertUserReplyPayload
             / AUTHMD5 AuthMD5ReplyPayload
             / FILE_CREATE FileCreateReplyPayload
             / FILE_COPY FileCopyReplyPayload
             / FILE_DELETE FileDeleteReplyPayload
             / FILE_RENAME FileRenameReplyPayload
             / FILE_METADATA FileMetaDataReplyPayload
             / FILE_MOVE FileMoveReplyPayload
             / FILE_SHARE FileShareReplyPayload
             / FILE_GET FileGetReplyPayload
             / FILE_MODIFY FileModifyReplyPayload
             / FOLDER_CREATE FolderCreateReplyPayload
             / FOLDER_COPY FolderCopyReplyPayload
             / FOLDER_DELETE FolderDeleteReplyPayload
```

```
              / FOLDER_RENAME FolderRenameReplyPayload
              / FOLDER_METADATA FolderMetaDataReplyPayload
              / FOLDER_MOVE FolderMoveReplyPayload
              / FOLDER_OPEN FolderOpenReplyPayload
              / FOLDER_SHARE FolderShareReplyPayload
              / FOLDER_LIST FolderListReplyPayload
              / SERVER_LOGS ServerLogsReplyPayload
              / SERVER_MANAGE_BANS ServerManageBansReplyPayload
              / SERVER_SHUTDOWN ServerShutdownReplyPayload
              / SERVER_VIEW_STATS ServerViewStatsReplyPayload
              / USER_CREATE UserCreateReplyPayload
              / USER_DELETE UserDeleteReplyPayload
              / USER_LIST UserListReplyPayload
              / USER_PERMISSIONS UserPermissionsReplyPayload
              / USER_RENAME UserRenameReplyPayload
              / RESERVED_CMD

RESERVED_CMD = %xffffffff

              ; VendorDefined are only valid when both the
              ; client and server agree on compatible
              ; VENDOR_ID values.
              ;
              ; Where:
              ; Length is the length of data that follows
              ; *uint8_t Is an array of Length 8-bit values.
              ;
VendorDefined = %x80000000-fffffffe Length *uint8_t

Command = SEQ (Command / VendorDefined)
CommandReply = SEQ (CommandReply / VendorDefined)

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

              ; Length: The number of CommandReply objects.
PacketReply = Length 1*CommandReply
```

```
                        ; 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  = PHOENIX_BIT:1 SERVER_KICK_PAYLOAD:31
```

```
               ; 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     = PHOENIX_BIT:1 AUTHMD5:31 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  = PHOENIX_BIT:1 AUTHANONYMOUS:31
```

```
FOLDER_CAPABILITY_REPLY = *1FOLDER_CREATE
                          *1FOLDER_COPY
                          *1FOLDER_RENAME
                          *1FOLDER_METATATA
                          *1FOLDER_MOVE
                          *1FOLDER_OPEN
                          *1FOLDER_SHARE
                          *1FOLDER_LIST
                          *1FILE_CREATE
                          *1FILE_COPY
                          *1FILE_DELETE
                          *1FILE_RENAME
                          *1FILE_METADATA
                          *1FILE_MOVE
                          *1FILE_SHARE
                          *1FILE_GET
                          *1FILE_MODIFY
                          *VENDOR_FOLDER_CAPABILITY
```

```
FOLDER_CAPABILITY_REQUEST:8 = %13:8
```

Appendix G. Complete XDR

```
/**
 * A value that can hold any signed
 * integer value is called an int_t.
 */
typedef integer int_t;

/**
 * An 8 bit signed integer
 * is called an int8_t.
 */
typedef integer:8 int8_t;

/**
 * An 16 bit signed integer
 * is called an int16_t.
 */
typedef integer:16 int16_t;

/**
 * An 32 bit signed integer
 * is called an int32_t.
 */
typedef integer:32 int32_t;

/**
 * An 64 bit signed integer
 * is called an int64_t.
 */
typedef integer:64 int64_t;

/**
 * A value that can hold any unsigned integer value
 * is called an uint_t.
 */
typedef unsigned_integer uint_t;

/**
 * An 8 bit signed integer
 * is called an uint8_t.
 */
typedef unsigned_integer:8 uint8_t;

/**
 * An 16 bit signed integer
 * is called an uint16_t.
 */
typedef unsigned_integer:16 int16_t;

/**
 * An 32 bit signed integer
 * is called an uint32_t.
 */
typedef unsigned_integer:32 int32_t;

/**
 * An 64 bit signed integer
 * is called an uint64_t.
 */
typedef unsigned_integer:64 int64_t;

/**
 * The time in seconds since January 1 1970 in GMT
 * is a 64-bit unsigned integer (uint64_t) and
 * is called UTC_t.
 */
typedef uint64_t UTC_t;

/**
 * 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;

/**
 * A single UTF-8 character.
 */
typedef uint8_t string;

/**
 * An array of UTF-8 character.
 * with no predefined length.
 */
typedef uint8_t string<>;

/**
 * Any 8-bit value without any
 * associated data type.
 */
typedef uint8_t opaque;

/**
 * An array of opaque values
 * with no predefined length.
 */
typedef uint8_t opaque<>;

/**
 * When a stringD value is set to this,
 * The the associated value has been deleted.
 */
const Deleted = 0xffffffff;

/**
 * A stringD is a string of printable UTF-8
 * characters,
 * -Or Deleted (the associated data has been deleted),
 * -Or is nullptr (not assigned).
 */
union stringD switch (uint32_t LengthOrDeleted) {
  case nullptr:
    void

  case Deleted:
    void;

  default:
    // An array of uint8_t characters
    // with no predefined length.
    //
    string Utf8Characters<>;
};

/**
 * An 8-bit value used to represent
 * true or false.
 */
typedef uint8_t bool;

/**
 * The value true.
 */
const bool true = 1;

/**
 * The value false.
 */
const bool false = 0;

/**
 * A string Key and its associated Value.
 * It uses stringD as the value to enable an indicator that the
 * key part (string part) may have been deleted.
 * And a stringD Value may be nullptr.
 */
struct KeyPair {
    string Key<>;
    stringD Value<>;
};

/**
 * An array of KeyPair objects
 * with an unspecified length.
 */
typedef KeyPair KeyPairArray<>;

/**
 * A 1-bit value;
 * The highest bit in the value, 1 means it is
 * a vendor extension.
 */
const VENDOR_BIT = 0x1;

/**
 * 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;
```

```
/**
 * Mask to check if CMD value in packet
 * is vendor extension.
 */
const CMD_VENDOR_MASK = 0x80000000;
struct VendorDefined {
 uint32_t VendorCommand; /* 0x80000000-fffffffe */
 opaque Data<>; /* XDR arrays start with a length. */
};
/**
 * A CMD payload is one of these types.
 * With Cmd set to a CMD value.
 */
union OneCommand switch (CMD_e Cmd) {
 case AUTHANONYMOUS:
  AuthAnonymousPayload AuthAnonymousCmd;
 case AUTHCERT_TLS:
  AuthCertTlsPayload AuthCertTlsCmd;
 case AUTHCERT_USER:
  AuthCertUserPayload AuthCertUserCmd;
 case AUTHMD5:
  AuthMD5Payload AuthMD5Cmd;
 case BYE:
  ByePayload ByeCmd;
 case CAPABILITY_PRE:
  CapabilityPayload CapabilityPreCmd;
 case CAPABILITY_POST:
  CapabilityPayload CapabilityPostCmd;
 case FILE_CREATE:
  FileCreatePayload FileCreateCmd;
 case FILE_COPY:
  FileCopyPayload FileCopyCmd;
 case FILE_DELETE:
  FileDeletePayload FileDeleteCmd;
 case FILE_RENAME:
  FileRenamePayload FileRenameCmd;
 case FILE_METADATA:
  FileMetaDataPayload FileMetaDataCmd;
 case FILE_MOVE:
  FileMovePayload FileMoveCmd;
 case FILE_SHARE:
  FileSharePayload FileShareCmd;
 case FILE_GET:
  FileGetPayload FileGetCmd;
 case FILE_MODIFY:
  FileModifyPayload FileModifyCmd;
 case FOLDER_CREATE:
  FolderCreatePayload FolderCreateCmd;
 case FOLDER_COPY:
  FolderCopyPayload FolderCopyCmd;
 case FOLDER_DELETE:
  FolderDeletePayload FolderDeleteCmd;
 case FOLDER_RENAME:
  FolderRenamePayload FolderRenameCmd;
 case FOLDER_METADATA:
  FolderMetaDataPayload FolderMetaDataCmd;
 case FOLDER_MOVE:
  FolderMovePayload FolderMoveCmd;
 case FOLDER_OPEN:
  FolderOpenPayload FolderOpenCmd;
 case FOLDER_SHARE:
  FolderSharePayload FolderShareCmd;
 case FOLDER_LIST:
  FolderListPayload FolderListCmd;
 case NOT_SUPPORTED:
  void;
 case RESERVED_CMD:
  void;
 case SERVER_CONFIGURE:
  ServerConfigurePayload ServerConfigCmd;
 case SERVER_KICK_USER:
  void;
 case SERVER_LOGS:
  ServerLogsPayload ServerLogsCmd;
 case SERVER_MANAGE_BANS:
  ServerManageBansPayload ServerBansCmd;
 case SERVER_SHUTDOWN:
  ServerShutdownPayload ServerShutdownCmd;
 case SERVER_VIEW_STATS:
  ServerStatsPayload ServerStatsCmd;
 case USER_CREATE:
  UserCreatePayload UserCreateCmd;
 case USER_DELETE:
  UserDeletePayload UserDeleteCmd;
 case USER_LIST:
  UserListPayload UserListCmd;
 case USER_PERMISSIONS:
  UserPermissionsPayload UserPermissionsCmd;
 case USER_RENAME:
  UserRenamePayload UserRenameCmd;
 default:
  VendorDefined Vendor;
};
/**
 * A CMDReply payload is one of these types.
 * With Cmd set to a CMD value.
 */
union OneReply switch (CMD_e Cmd) {
 case AUTHANONYMOUS:
  void; /* Replies with CAPABILITY_PRE or CAPABILITY_POST */
 case AUTHCERT_TLS:
  AuthCertTlsReplyPayload AuthCertTlsCmd;
 case AUTHCERT_USER:
  AuthCertUserReplyPayload AuthCertUserCmd;
 case AUTHMD5:
  void;
 case BYE:
  ByeReplyPayload ByeCmd;
 case FILE_CREATE:
  FileCreateReplyPayload FileCreateCmd;
 case FILE_COPY:
  FileCopyReplyPayload FileCopyCmd;
 case FILE_DELETE:
  FileDeleteReplyPayload FileDeleteCmd;
 case FILE_RENAME:
  FileRenameReplyPayload FileRenameCmd;
 case FILE_METADATA:
  FileMetaDataReplyPayload FileMetaDataCmd;
 case FILE_MOVE:
  FileMoveReplyPayload FileMoveCmd;
 case FILE_SHARE:
  FileShareReplyPayload FileShareCmd;
 case FILE_GET:
  FileGetReplyPayload FileGetCmd;
 case FILE_MODIFY:
  FileModifyReplyPayload FileModifyCmd;
 case FOLDER_CREATE:
  FolderCreateReplyPayload FolderCreateCmd;
 case FOLDER_COPY:
  FolderCopyReplyPayload FolderCopyCmd;
 case FOLDER_DELETE:
  FolderDeleteReplyPayload FolderDeleteCmd;
 case FOLDER_RENAME:
  FolderRenameReplyPayload FolderRenameCmd;
 case FOLDER_METADATA:
  FolderMetaDataReplyPayload FolderMetaDataCmd;
 case FOLDER_MOVE:
  FolderMoveReplyPayload FolderMoveCmd;
 case FOLDER_OPEN:
  FolderOpenReplyPayload FolderOpenCmd;
 case FOLDER_SHARE:
  FolderShareReplyPayload FolderShareCmd;
 case FOLDER_LIST:
  FolderListReplyPayload FolderListCmd;
 case RESERVED_CMD:
  void;
 case SERVER_CONFIGURE:
  ServerConfigureReplyPayload ServerConfigCmd;
 case SERVER_LOGS:
  ServerLogsReplyPayload ServerLogsCmd;
 case SERVER_MANAGE_BANS:
  ServerManageBansReplyPayload ServerManageBansCmd;
 case SERVER_SHUTDOWN:
  ServerShutdownReplyPayload ServerShutdownCmd;
 case SERVER_VIEW_STATS:
  ServerStatsReplyPayload ServerViewStatsCmd;
 case USER_CREATE:
  UserCreateReplyPayload UserCreateCmd;
 case USER_DELETE:
  UserDeleteReplyPayload UserDeleteCmd;
 case USER_LIST:
  UserListReplyPayload UserListCmd;
 case USER_PERMISSIONS:
  UserPermissionsReplyPayload UserPermissionsCmd;
 case USER_RENAME:
  UserRenameReplyPayload UserRenameCmd;
 default:
  VendorDefined Vendor;
};
/*
 * One command.
 */
struct Command {
```

```
 SEQ_t Sequence;
 OneCommand Payload;
};
/*
 * One Reply
 */
struct CommandReply {
 SEQ_t Sequence;
 OneReply Payload;
};
/**
 * A packet body.
 */
struct PacketBody {
  Command Commands<>; /* XDR arrays start with the Length */
};
/**
 * A packet reply.
 */
struct PacketReply {
  CommandReply Commands<>; /* XDR arrays start with the Length */
};
```

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

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

```
struct ServerKickPayload
{
 /*
         * With Kick set to SERVER_KICK_USER.
         * And the VENDOR_BIT not set.
         */
 CMD_e Kick;
};
```

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

```
struct ServerManageBansPayload
{
 int foo;
};
struct ServerManageBansReplyPayload
{
 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 AuthMD5Payload
{
 string AccountName<>;
 string Md5Password<>;
};
```

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

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

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