Network Working Group
Request for Comments: 3652
Category: Informational

S. Sun
S. Reilly
L. Lannom
J. Petrone
CNRI

November 2003

Handle System Protocol (ver 2.1) Specification

Status of this Memo

This memo provides information for the Internet community. It does not specify an Internet standard of any kind. Distribution of this memo is unlimited.

Copyright Notice

IESG Note

Several groups within the IETF and IRTF have discussed the Handle System and its relationship to existing systems of identifiers. The IESG wishes to point out that these discussions have not resulted in IETF consensus on the described Handle System, nor on how it might fit into the IETF architecture for identifiers. Though there has been discussion of handles as a form of URI, specifically as a URN, these documents describe an alternate view of how namespaces and identifiers might work on the Internet and include characterizations of existing systems which may not match the IETF consensus view.

Abstract

The Handle System is a general-purpose global name service that allows secured name resolution and administration over the public Internet. This document describes the protocol used for client software to access the Handle System for both handle resolution and administration. The protocol specifies the procedure for a client software to locate the responsible handle server of any given handle. It also defines the messages exchanged between the client and server for any handle operation.

Table of Contents

1. Overview

The Handle System provides a general-purpose, secured global name service for the Internet. It was originally conceived and described in a paper by Robert Kahn and Robert Wilensky [18] in 1995. The Handle System defines a client server protocol in which client software submits requests via a network to handle servers. Each request describes the operation to be performed on the server. The server will process the request and return a message indicating the result of the operation. This document specifies the protocol for client software to access a handle server for handle resolution and administration. It does not include the description of the protocol used to manage handle servers. A discussion of the management protocol is out of the scope of this document and will be made available in a separate document. The document assumes that readers are familiar with the basic concepts of the Handle System as introduced in the "Handle System Overview" [1], as well as the data model and service definition given in the "Handle System Namespace and Service Definition" [2].

The Handle System consists of a set of service components as defined in [2]. From the client's point of view, the Handle System is a distributed database for handles. Different handles under the Handle System may be maintained by different handle servers at different network locations. The Handle protocol specifies the procedure for a client to locate the responsible handle server of any given handle. It also defines the messages exchanged between the client and server for any handle operation.

Some key aspects of the Handle protocol include:

- The Handle protocol supports both handle resolution and administration. The protocol follows the data and service model defined in [2].
- A client may authenticate any server response based on the server's digital signature.

- A server may authenticate its client as handle administrator via the Handle authentication protocol. The Handle authentication protocol is a challenge-response protocol that supports both public-key and secret-key based authentication.

- A session may be established between the client and server so that authentication information and network resources (e.g., TCP connection) may be shared among multiple operations. A session key can be established to achieve data integrity and confidentiality.

- The protocol can be extended to support new operations. Controls can be used to extend the existing operations. The protocol is defined to allow future backward compatibility.

- Distributed service architecture. Support service referral among different service components.

- Handles and their data types are based on the ISO-10646 (Unicode 2.0) character set. UTF-8 [3] is the mandated encoding under the Handle protocol.

The Handle protocol (version 2.1) specified in this document has changed significantly from its earlier versions. These changes are necessary due to changes made in the Handle System data model and the service model. Servers that implement this protocol may continue to support earlier versions of the protocol by checking the protocol version specified in the Message Envelope (see section 2.2.1).

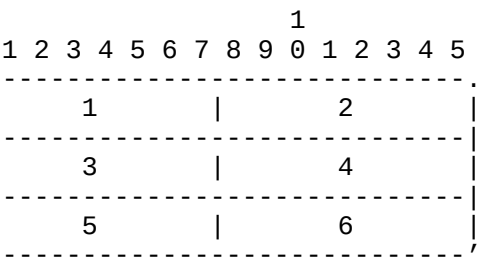
2. Protocol Elements

2.1. Conventions

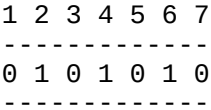
The following conventions are followed by the Handle protocol to ensure interoperability among different implementations.

2.1.1. Data Transmission Order

The order of transmission of data packets follows the network byte order (also called the Big-Endian [11]). That is, when a data-gram consists of a group of octets, the order of transmission of those octets follows their natural order from left to right and from top to bottom, as they are read in English. For example, in the following diagram, the octets are transmitted in the order they are numbered.



If an octet represents a numeric quantity, the left most bit is the most significant bit. For example, the following diagram represents the value 170 (decimal).



Similarly, whenever a multi-octet field represents a numeric quantity, the left most bit is the most significant bit and the most significant octet of the whole field is transmitted first.

2.1.2. Transport Layer

The Handle protocol is designed so that messages may be transmitted either as separate data-grams over UDP or as a continuous byte stream via a TCP connection. The recommended port number for both UDP and TCP is 2641.

UDP Usage

Messages carried by UDP are restricted to 512 bytes (not including the IP or UDP header). Longer messages must be fragmented into UDP packets where each packet carries a proper sequence number in the Message Envelope (see Section 2.2.1).

The optimum retransmission policy will vary depending on the network or server performance, but the following are recommended:

- The client should try other servers or service interfaces before repeating a request to the same server address.
- The retransmission interval should be based on prior statistics if possible. Overly aggressive retransmission should be avoided to prevent network congestion. The recommended retransmission interval is 2-5 seconds.

- When transmitting large amounts of data, TCP-friendly congestion control, such as an interface to the Congestion Manager [12], should be implemented whenever possible to avoid unfair consumption of the bandwidth against TCP-based applications. Details of the congestion control will be discussed in a separate document.

TCP Usage

Messages under the Handle protocol can be mapped directly into a TCP byte-stream. However, the size of each message is limited by the range of a 4-byte unsigned integer. Longer messages may be fragmented into multiple messages before the transmission and reassembled at the receiving end.

Several connection management policies are recommended:

- The server should support multiple connections and should not block other activities waiting for TCP data.

- By default, the server should close the connection after completing the request. However, if the request asks to keep the connection open, the server should assume that the client will initiate connection closing.

2.1.3. Character Case

Handles are character strings based on the ISO-10646 character set and must be encoded in UTF-8. By default, handle characters are treated as case-sensitive under the Handle protocol. A handle service, however, may be implemented in such a way that ASCII characters are processed case-insensitively. For example, the Global Handle Registry (GHR) provides a handle service where ASCII characters are processed in a case-insensitive manner. This suggests that ASCII characters in any naming authority are case-insensitive.

When handles are created under a case-insensitive handle server, their original case should be preserved. To avoid any confusion, the server should avoid creating any handle whose character string matches that of an existing handle, ignoring the case difference. For example, if the handle "X/Y" was already created, the server should refuse any request to create the handle "x/y" or any of its case variations.

2.1.4. Standard String Type: UTF8-String

Handles are transmitted as UTF8-Strings under the Handle protocol. Throughout this document, UTF8-String stands for the data type that consists of a 4-byte unsigned integer followed by a character string in UTF-8 encoding. The leading integer specifies the number of octets of the character string.

2.2. Common Elements

Each message exchanged under the system protocol consists of four sections (see Fig. 2.2). Some of these sections (e.g., the Message Body) may be empty depending on the protocol operation.

The Message Envelope must always be present. It has a fixed size of 20 octets. The Message Envelope does not carry any application layer information and is primarily used to help deliver the message. Content in the Message Envelope is not protected by the digital signature in the Message Credential.

The Message Header must always be present as well. It has a fixed size of 24 octets and holds the common data fields of all messages exchanged between client and server. These include the operation code, the response code, and the control options for each protocol operation. Content in the Message Header is protected by the digital signature in the Message Credential.

The Message Body contains data specific to each protocol operation. Its format varies according to the operation code and the response code in the Message Header. The Message Body may be empty. Content in the Message Body is protected by the digital signature in the Message Credential.

The Message Credential provides a mechanism for transport security for any message exchanged between the client and server. A non-empty Message Credential may contain the digital signature from the originator of the message or the one-way Message Authentication Code (MAC) based on a pre-established session key. The Message Credential may be used to authenticate the message between the client and server. It can also be used to check data integrity after its transmission.

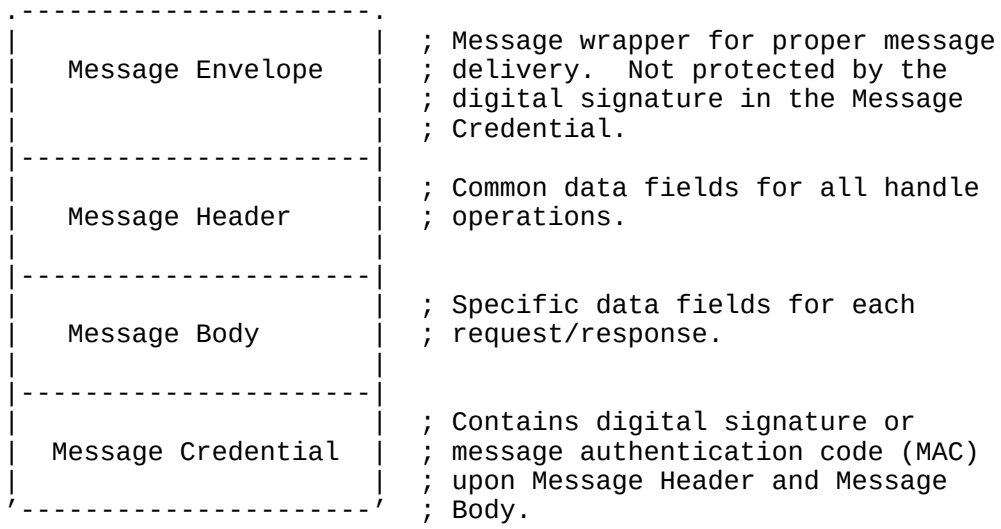
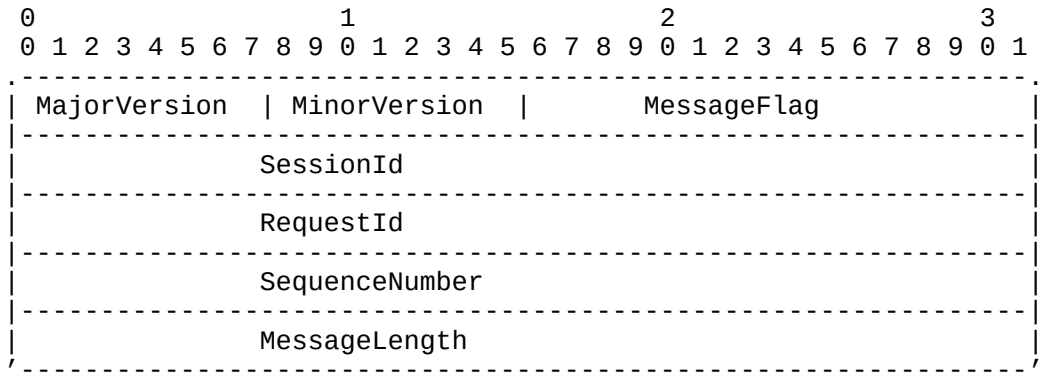


Fig 2.2: Message format under the Handle protocol

2.2.1. Message Envelope

Each message begins with a Message Envelope under the Handle protocol. If a message has to be truncated before its transmission, each truncated portion must also begin with a Message Envelope.

The Message Envelope allows the reassembly of the message at the receiving end. It has a fixed size of 20 octets and consists of seven fields:



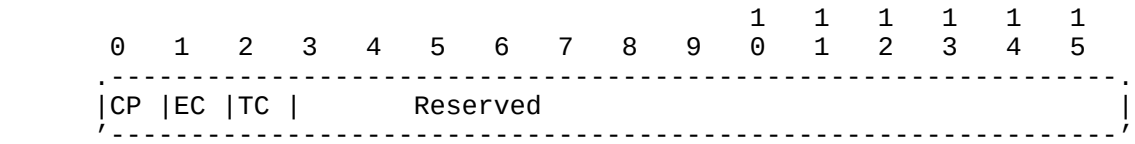
2.2.1.1. <MajorVersion> and <MinorVersion>

The <MajorVersion> and <MinorVersion> are used to identify the version of the Handle protocol. Each of them is defined as a one-byte unsigned integer. This specification defines the protocol version whose <MajorVersion> is 2 and <MinorVersion> is 1.

<MajorVersion> and <MinorVersion> are designed to allow future backward compatibility. A difference in <MajorVersion> indicates major variation in the protocol format and the party with the lower <MajorVersion> will have to upgrade its software to ensure precise communication. An increment in <MinorVersion> is made when additional capabilities are added to the protocol without any major change to the message format.

2.2.1.2. <MessageFlag>

The <MessageFlag> consists of two octets defined as follows:



Bit 0 is the CP (ComPressed) flag that indicates whether the message (excluding the Message Envelope) is compressed. If the CP bit is set (to 1), the message is compressed. Otherwise, the message is not compressed. The Handle protocol uses the same compression method as used by the FTP protocol[8].

Bit 1 is the EC (EnCrypted) flag that indicates whether the message (excluding the Message Envelope) is encrypted. The EC bit should only be set under an established session where a session key is in place. If the EC bit is set (to 1), the message is encrypted using the session key. Otherwise the message is not encrypted.

Bit 2 is the TC (TrunCated) flag that indicates whether this is a truncated message. Message truncation happens most often when transmitting a large message over the UDP protocol. Details of message truncation (or fragmentation) will be discussed in section 2.3.

Bits 3 to 15 are currently reserved and must be set to zero.

2.2.1.3. <SessionId>

The <SessionId> is a four-byte unsigned integer that identifies a communication session between the client and server.

Session and its <SessionId> are assigned by a server, either upon an explicit request from a client or when multiple message exchanges are expected to fulfill the client's request. For example, the server will assign a unique <SessionId> in its response if it has to authenticate the client. A client may explicitly ask the server to set up a session as a virtually private communication channel like SSL [4]. Requests from clients without an established session must have their <SessionId> set to zero. The server must assign a unique non-zero <SessionId> for each new session. It is also responsible for terminating those sessions that are not in use after some period of time.

Both clients and servers must maintain the same <SessionId> for messages exchanged under an established session. A message whose <SessionId> is zero indicates that no session has been established.

The session and its state information may be shared among multiple handle operations. They may also be shared over multiple TCP connections as well. Once a session is established, both client and server must maintain their state information according to the <SessionId>. The state information may include the stage of the conversation, the other party's authentication information, and the session key that was established for message encryption or authentication. Details of these are discussed in section 3.8.

2.2.1.4. <RequestId>

Each request from a client is identified by a <RequestId>, a 4-byte unsigned integer set by the client. Each <RequestId> must be unique from all other outstanding requests from the same client. The <RequestId> allows the client to keep track of its requests, and any response from the server must include the correct <RequestId>.

2.2.1.5. <SequenceNumber>

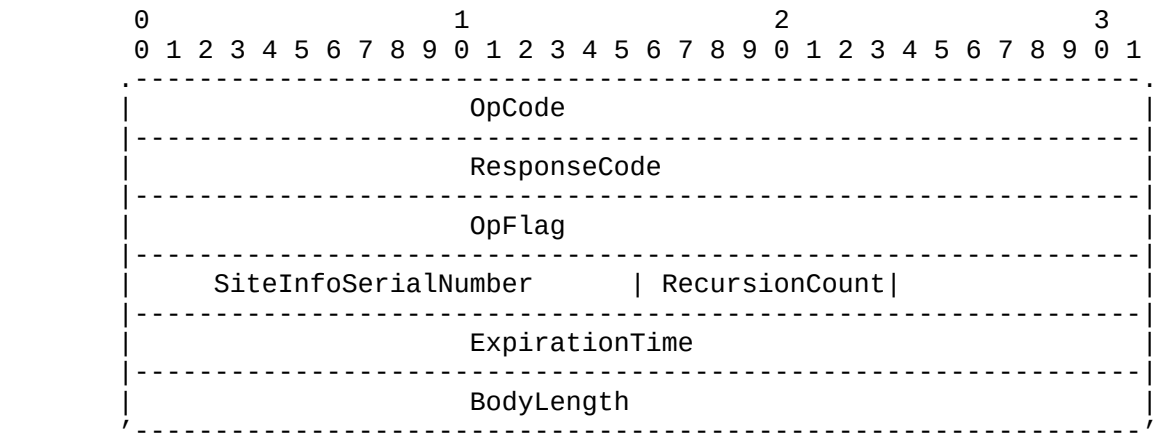
Messages under the Handle protocol may be truncated during their transmission (e.g., under UDP). The <SequenceNumber> is a 4-byte unsigned integer used as a counter to keep track of each truncated portion of the original message. The message recipient can reassemble the original message based on the <SequenceNumber>. The <SequenceNumber> must start with 0 for each message. Each truncated message must set its TC flag in the Message Envelope. Messages that are not truncated must set their <SequenceNumber> to zero.

2.2.1.6. <MessageLen>

A 4-byte unsigned integer that specifies the total number of octets of any message, excluding those in the Message Envelope. The length of any single message exchanged under the Handle protocol is limited by the range of a 4-byte unsigned integer. Longer data can be transmitted as multiple messages with a common <RequestId>.

2.2.2. Message Header

The Message Header contains the common data elements among any protocol operation. It has a fixed size of 24 octets and consists of eight fields.



Every message that is not truncated must have a Message Header. If a message has to be truncated for its transmission, the Message Header must appear in the first truncated portion of the message.

This is different from the Message Envelope, which appears in each truncated portion of the message.

2.2.2.1. <OpCode>

The <OpCode> stands for operation code, which is a four-byte unsigned integer that specifies the intended operation. The following table lists the <OpCode>s that MUST be supported by all implementations in order to conform to the base protocol specification. Each operation code is given a symbolic name that is used throughout this document for easy reference.

Op_Code	Symbolic Name	Remark
0	OC_RESERVED	Reserved
1	OC_RESOLUTION	Handle query
2	OC_GET_SITEINFO	Get HS_SITE values
100	OC_CREATE_HANDLE	Create new handle
101	OC_DELETE_HANDLE	Delete existing handle
102	OC_ADD_VALUE	Add handle value(s)
103	OC_REMOVE_VALUE	Remove handle value(s)
104	OC_MODIFY_VALUE	Modify handle value(s)
105	OC_LIST_HANDLE	List handles
106	OC_LIST_NA	List sub-naming authorities
200	OC_CHALLENGE_RESPONSE	Response to challenge
201	OC_VERIFY_RESPONSE	Verify challenge response
300 : 399	{ Reserved for handle server administration }	
400	OC_SESSION_SETUP	Session setup request
401	OC_SESSION_TERMINATE	Session termination request
402	OC_SESSION_EXCHANGEKEY	Session key exchange

A detailed description of each of these <OpCode>s can be found in section 3 of this document. In general, clients use the <OpCode> to tell the server what kind of handle operation they want to accomplish. Response from the server must maintain the same <OpCode> as the original request and use the <ResponseCode> to indicate the result.

2.2.2.2. <ResponseCode>

The <ResponseCode> is a 4-byte unsigned integer that is given by a server to indicate the result of any service request. The list of <ResponseCode>s used in the Handle protocol is defined in the following table. Each response code is given a symbolic name that is used throughout this document for easy reference.

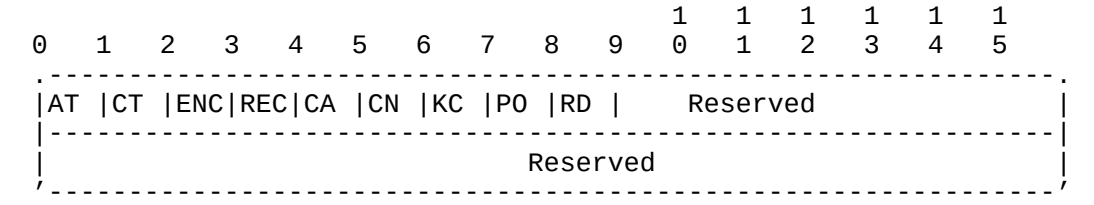
Res. Code	Symbolic Name	Remark
0	RC_RESERVED	Reserved for request
1	RC_SUCCESS	Success response
2	RC_ERROR	General error
3	RC_SERVER_BUSY	Server too busy to respond
4	RC_PROTOCOL_ERROR	Corrupted or unrecognizable message
5	RC_OPERATION_DENIED	Unsupported operation
6	RC_RECUR_LIMIT_EXCEEDED	Too many recursions for the request
100	RC_HANDLE_NOT_FOUND	Handle not found
101	RC_HANDLE_ALREADY_EXIST	Handle already exists
102	RC_INVALID_HANDLE	Encoding (or syntax) error
200	RC_VALUE_NOT_FOUND	Value not found
201	RC_VALUE_ALREADY_EXIST	Value already exists
202	RC_VALUE_INVALID	Invalid handle value
300	RC_EXPIRED_SITE_INFO	SITE_INFO out of date
301	RC_SERVER_NOT_RESP	Server not responsible
302	RC_SERVICE_REFERRAL	Server referral
303	RC_NA_DELEGATE	Naming authority delegation takes place.
400	RC_NOT_AUTHORIZED	Not authorized/permitted
401	RC_ACCESS_DENIED	No access to data
402	RC_AUTHEN_NEEDED	Authentication required
403	RC_AUTHEN_FAILED	Failed to authenticate
404	RC_INVALID_CREDENTIAL	Invalid credential
405	RC_AUTHEN_TIMEOUT	Authentication timed out
406	RC_UNABLE_TO_AUTHEN	Unable to authenticate
500	RC_SESSION_TIMEOUT	Session expired
501	RC_SESSION_FAILED	Unable to establish session
502	RC_NO_SESSION_KEY	No session yet available
503	RC_SESSION_NO_SUPPORT	Session not supported
504	RC_SESSION_KEY_INVALID	Invalid session key
900	RC_TRYING	Request under processing
901	RC_FORWARDED	Request forwarded to another server
902	RC_QUEUED	Request queued for later processing

Response codes under 10000 are reserved for system use. Any message with a response code under 10000 but not listed above should be treated as an unknown error. Response codes above 10000 are user defined and can be used for application specific purposes.

Detailed descriptions of these <ResponseCode>s can be found in section 3 of this document. In general, any request from a client must have its <ResponseCode> set to 0. The response message from the server must have a non-zero <ResponseCode> to indicate the result. For example, a response message from a server with <ResponseCode> set to RC_SUCCESS indicates that the server has successfully fulfilled the client's request.

2.2.2.3. <OpFlag>

The <OpFlag> is a 32-bit bit-mask that defines various control options for protocol operation. The following figure shows the location of each option flag in the <OpFlag> field.



AT - AuThoritative bit. A request with the AT bit set (to 1) indicates that the request should be directed to the primary service site (instead of any mirroring sites). A response message with the AT bit set (to 1) indicates that the message is returned from a primary server (within the primary service site).

CT - CerTified bit. A request with the CT bit set (to 1) asks the server to sign its response with its digital signature. A response with the CT bit set (to 1) indicates that the message is signed. The server must sign its response if the request has its CT bit set (to 1). If the server fails to provide a valid signature in its response, the client should discard the response and treat the request as failed.

ENC - ENCryption bit. A request with the ENC bit set (to 1) requires the server to encrypt its response using the pre-established session key.

REC - RECursive bit. A request with the REC bit set (to 1) asks the server to forward the query on behalf of the client if the request has to be processed by another handle server. The server may honor the request by forwarding the request to the appropriate handle server and passing on any result back to the client. The server may also deny any such request by sending a response with <ResponseCode> set to RC_SERVER_NOT_RESP.

CA - Cache Authentication. A request with the CA bit set (to 1) asks the caching server (if any) to authenticate any server response (e.g., verifying the server's signature) on behalf of the client. A response with the CA bit set (to 1) indicates that the response has been authenticated by the caching server.

CN - ContiNuous bit. A message with the CN bit set (to 1) tells the message recipient that more messages that are part of the same request (or response) will follow. This happens if a request (or response) has data that is too large to fit into any single message and has to be fragmented into multiple messages.

KC - Keep Connection bit. A message with the KC bit set requires the message recipient to keep the TCP connection open (after the response is sent back). This allows the same TCP connection to be used for multiple handle operations.

PO - Public Only bit. Used by query operations only. A query request with the PO bit set (to 1) indicates that the client is only asking for handle values that have the PUB_READ permission. A request with PO bit set to zero asks for all the handle values regardless of their read permission. If any of the handle values require ADMIN_READ permission, the server must authenticate the client as the handle administrator.

RD - Request-Digest bit. A request with the RD bit set (to 1) asks the server to include in its response the message digest of the request. A response message with the RD bit set (to 1) indicates that the first field in the Message Body contains the message digest of the original request. The message digest can be used to check the integrity of the server response. Details of these are discussed later in this document.

All other bits in the <OpFlag> field are reserved and must be set to zero.

In general, servers must honor the <OpFlag> specified in the request. If a requested option cannot be met, the server should return an error message with the proper <ResponseCode> as defined in the previous section.

2.2.2.4. <SiteInfoSerialNumber>

The <SiteInfoSerialNumber> is a two-byte unsigned integer. The <SiteInfoSerialNumber> in a request refers to the <SerialNumber> of the HS_SITE value used by the client (to access the server). Servers can check the <SiteInfoSerialNumber> in the request to find out if the client has up-to-date service information.

When possible, the server should fulfill a client's request even if the service information used by the client is out-of-date. However, the response message should specify the latest version of service information in the <SiteInforSerialNumber> field. Clients with out-of-date service information can update the service information from the Global Handle Registry. If the server cannot fulfill a client's request due to expired service information, it should reject the request and return an error message with <ResponseCode> set to RC_EXPIRED_SITE_INFO.

2.2.2.5. <RecursionCount>

The <RecursionCount> is a one-byte unsigned integer that specifies the number of service recursions. Service recursion happens if the server has to forward the client's request to another server. Any request directly from the client must have its <RecursionCount> set to 0. If the server has to send a recursive request on behalf of the client, it must increment the <RecursionCount> by 1. Any response from the server must maintain the same <RecursionCount> as the one in the request. To prevent an infinite loop of service recursion, the server should be configurable to stop sending a recursive request when the <RecursionCount> reaches a certain value.

2.2.2.6. <ExpirationTime>

The <ExpirationTime> is a 4-byte unsigned integer that specifies the time when the message should be considered expired, relative to January 1st, 1970 GMT, in seconds. It is set to zero if no expiration is expected.

2.2.2.7. <BodyLength>

The <BodyLength> is a 4-byte unsigned integer that specifies the number of octets in the Message Body. The <BodyLength> does not count the octets in the Message Header or those in the Message Credential.

2.2.3. Message Body

The Message Body always follows the Message Header. The number of octets in the Message Body can be determined from the <BodyLength> in the Message Header. The Message Body may be empty. The exact format of the Message Body depends on the <OpCode> and the <ResponseCode> in the Message Header. Details of the Message Body under each <OpCode> and <ResponseCode> are described in section 3 of this document.

For any response message, if the Message Header has its RD bit (in <OpFlag>) set to 1, the Message Body must begin with the message digest of the original request. The message digest is defined as follows:

```
<RequestDigest> ::= <DigestAlgorithmIdentifier>
                    <MessageDigest>
```

where

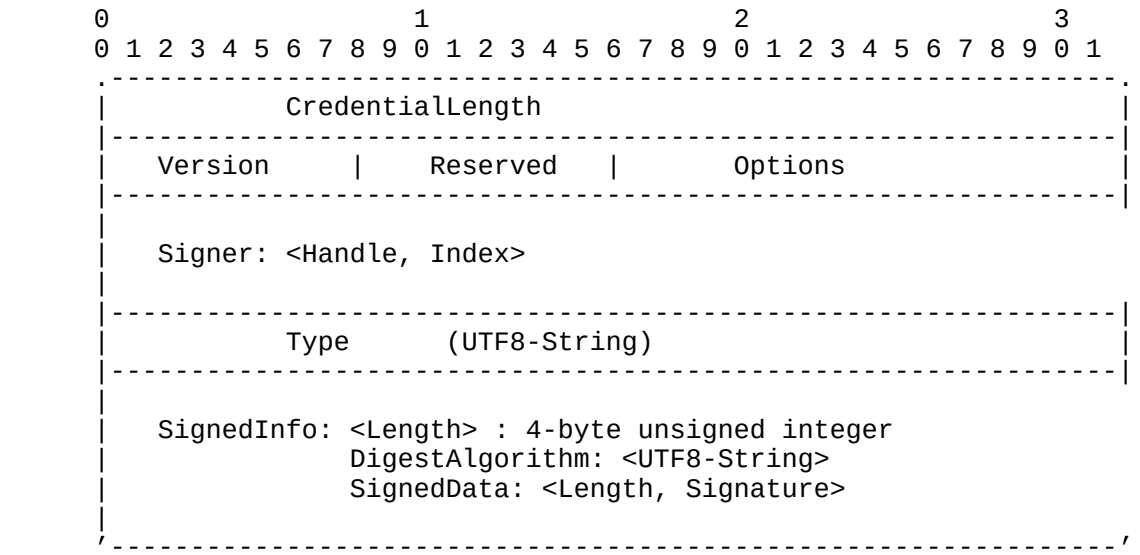
<DigestAlgorithmIdentifier>
An octet that identifies the algorithm used to generate the message digest. If the octet is set to 1, the digest is generated using the MD5 [9] algorithm. If the octet is set to 2, SHA-1 [10] algorithm is used.

<MessageDigest>
The message digest itself. It is calculated upon the Message Header and the Message Body of the original request. The length of the field is fixed according to the digest algorithm. For MD5 algorithm, the length is 16 octets. For SHA-1, the length is 20 octets.

The Message Body may be truncated into multiple portions during its transmission (e.g., over UDP). Recipients of such a message may reassemble the Message Body from each portion based on the <SequenceNumber> in the Message Envelope.

2.2.4. Message Credential

The Message Credential is primarily used to carry any digital signatures signed by the message issuer. It may also carry the Message Authentication Code (MAC) if a session key has been established. The Message Credential is used to protect contents in the Message Header and the Message Body from being tampered with during transmission. The format of the Message Credential is designed to be semantically compatible with PKCS#7 [5]. Each Message Credential consists of the following fields:



where

<CredentialLength>
A 4-byte unsigned integer that specifies the number of octets in the Message Credential. It must be set to zero if the message has no Message Credential.

<Version>
An octet that identifies the version number of the Message Credential. The version number specified in this document is zero.

<Reserved>
An octet that must be set to zero.

<Options>
Two octets reserved for various cryptography options.

```
<Signer> ::= <HANDLE>
             <INDEX>
```

A reference to a handle value in terms of the <HANDLE> and the <INDEX> of the handle value. The handle value may contain the public key, or the X.509 certificate, that can be used to validate the digital signature.

<Type>
A UTF8-String that indicates the type of content in the <SignedInfo> field (described below). It may contain HS_DIGEST if <SignedInfo> contains the message digest, or HS_MAC if <SignedInfo> contains the Message Authentication Code (MAC). The <Type> field will specify the signature algorithm identifier if <SignedInfo> contains a digital signature. For example, with the <Type> field set to HS_SIGNED_PSS, the <SignedInfo> field will contain the digital signature generated using the RSA-PSS algorithm [16]. If the <Type> field is set to HS_SIGNED, the <SignedInfo> field will contain the digital signature generated from a DSA public key pair.

```
<SignedInfo> ::=  <Length>
                  <DigestAlgorithm>
                  <SignedData>
```

where

<Length>
A 4-byte unsigned integer that specifies the number of octets in the <SignedInfo> field.

<DigestAlgorithm>
A UTF8-String that refers to the digest algorithm used to generate the digital signature. For example, the value "SHA-1" indicates that the SHA-1 algorithm is used to generate the message digest for the signature.

```
<SignedData> ::=  <LENGTH>
                  <SIGNATURE>
```

where

<LENGTH>
A 4-byte unsigned integer that specifies the number of octets in the <SIGNATURE>.

<SIGNATURE>
Contains the digital signature or the MAC over the Message Header and Message Body. The syntax and semantics of the signature depend on the <Type> field

and the public key referenced in the <Signer> field. For example, if the <Type> field is "HS_SIGNED" and the public key referred to by the <Signer> field is a DSA [6] public key, the signature will be the ASN.1 octet string representation of the parameter R and S as described in [7]. If the <Signer> field refers to a handle value that contains a X.509 certificate, the signature should be encoded according to RFC 3279 and RFC 3280 [14, 15].

The Message Credential may contain the message authentication code (MAC) generated using a pre-established session key. In this case, the <Signer> field must set its <HANDLE> to a zero-length UTF8-String and its <INDEX> to the <SessionId> specified in the Message Envelope. The <Signature> field must contain the MAC in its <SIGNATURE> field. The MAC is the result of the one-way hash over the concatenation of the session key, the <Message Header>, the <MessageBody>, and the session key again.

The Message Credential in a response message may contain the digital signature signed by the server. The server's public key can be found in the service information used by the client to send the request to the server. In this case, the client should ignore any reference in the <Signer> field and use the public key in the service information to verify the signature.

The Message Credential can also be used for non-repudiation purposes. This happens if the Message Credential contains a server's digital signature. The signature may be used as evidence to demonstrate that the server has rendered its service in response to a client's request.

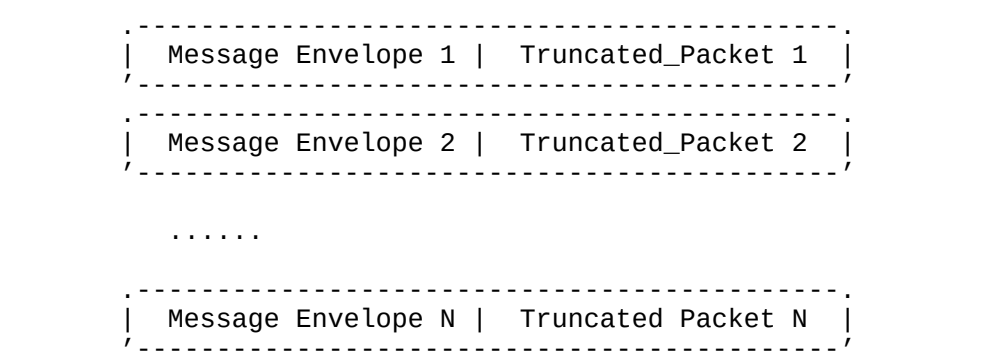
The Message Credential provides a mechanism for safe transmission of any message between the client and server. Any message whose Message Header and Message Body complies with its Message Credential suggests that the message indeed comes from its originator and assures that the message has not been tampered with during its transmission.

2.3. Message Transmission

A large message may be truncated into multiple packets during its transmission. For example, to fit the size limit of a UDP packet, the message issuer must truncate any large message into multiple UDP packets before its transmission. The message recipient must reassemble the message from these truncated packets before further processing. Message truncation must be carried out over the entire

message except the Message Envelope. A new Message Envelope has to be inserted in front of each truncated packet before its transmission. For example, a large message that consists of

may be truncated into:



where the "Truncated_packet 1", "Truncated_packet 2", ..., and "Truncated_packet N" result from truncating the Message Header, the Message Body and the Message Credential. Each "Message Envelope i" (inserted before each truncation) must set its TC flag to 1 and maintain the proper sequence count (in the <SequenceNumber>). Each "Message Envelope i" must also set its <MessageLength> to reflect the size of the packet. The recipient of these truncated packets can reassemble the message by concatenating these packets based on their <SequenceNumber>.

3. Handle Protocol Operations

This section describes the details of each protocol operation in terms of messages exchanged between the client and server. It also defines the format of the Message Body according to each <OpCode> and <ResponseCode> in the Message Header.

3.1. Client Bootstrapping

3.1.1. Global Handle Registry and its Service Information

The service information for the Global Handle Registry (GHR) allows clients to contact the GHR to find out the responsible service components for their handles. The service information is a set of HS_SITE values assigned to the root handle "0.NA/0.NA" and is also

called the root service information. The root service information may be distributed along with the client software, or be downloaded from the Handle System website at http://www.handle.net.

Changes to the root service information are identified by the <SerialNumber> in the HS_SITE values. A server at GHR can find out if the root service information used by the client is outdated by checking the <SerialNumber> in the client's request. The client should update the root service information if the <ResponseCode> of the response message is RC_EXPIRED_SITE_INFO. Clients may obtain the most up-to-date root service information from the root handle. The GHR must sign the root service information using the public key specified in the outdated service information (identified in the client's request) so that the client can validate the signature.

3.1.2. Locating the Handle System Service Component

Each handle under the Handle System is managed by a unique handle service component (e.g., LHS). For any given handle, the responsible service component (and its service information) can be found from its naming authority handle. Before resolving any given handle, the client needs to find the responsible service component by querying the naming authority handle from the GHR.

For example, to find the responsible LHS for the handle "1000/abc", client software can query the GHR for the HS_SITE (or HS_SERV) values assigned to the naming authority handle "0.NA/1000". The set of HS_SITE values provides the service information of the LHS that manages every handle under the naming authority "1000". If no HS_SITE values are found, the client can check if there is any HS_SERV value assigned to the naming authority handle. The HS_SERV value provides the service handle that maintains the service information for the LHS. Service handles are used to manage the service information shared by different naming authorities.

It is possible that the naming authority handle requested by the client does not reside at the GHR. This happens when naming authority delegation takes place. Naming authority delegation happens when a naming authority delegates an LHS to manage all its child naming authorities. In this case, the delegating naming authority must contain the service information, a set of HS_NA_DELEGATE values, of the LHS that manages its child naming authorities.

All top-level naming authority handles must be registered and managed by the GHR. When a server at the GHR receives a request for a naming authority that has been delegated to an LHS, it must return a message with the <ResponseCode> set to RC_NA_DELEGATE, along with the

HS_NA_DELAGATE values from the nearest ancestor naming authority. The client can query the LHS described by the HS_NA_DELAGATE values for the delegated naming authority handle. In practice, the ancestor naming authority should make itself available to any handle server within the GHR, by replicating itself at the time of delegation. This will prevent any cross-queries among handle servers (within a service site) when the naming authority in query and the ancestor naming authority do not hash into the same handle server.

3.1.3. Selecting the Responsible Server

Each handle service component is defined in terms of a set of HS_SITE values. Each of these HS_SITE values defines a service site within the service component. A service site may consist of a group of handle servers. For any given handle, the responsible handle server within the service component can be found following this procedure:

1. Select a preferred service site.

   Each service site is defined in terms of an HS_SITE value. The HS_SITE value may contain a <Description> or other attributes (under the <AttributeList>) to help the selection. Clients must select the primary service site for any administrative operations.

2. Locate the responsible server within the service site.

   This can be done as follows: Convert every ASCII character in the handle to its upper case. Calculate the MD5 hash of the converted handle string according to the <HashOption> given in the HS_SITE value. Take the last 4 bytes of the hash result as a signed integer. Modulo the absolute value of the integer by the <NumOfServer> given in the HS_SITE value. The result is the sequence number of the <ServerRecord> listed in the HS_SITE value. For example, if the result of the modulation is 2, the third <ServerRecord> listed in the <HS_SITE> should be selected. The <ServerRecord> defines the responsible handle server for the given handle.

3.2. Query Operation

A query operation consists of a client sending a query request to the responsible handle server and the server returning the query result to the client. Query requests are used to retrieve handle values assigned to any given handle.

3.2.1. Query Request

The Message Header of any query request must set its <OpCode> to OC_RESOLUTION (defined in section 2.2.2.1) and <ResponseCode> to 0.

The Message Body for any query request is defined as follows:

```
<Message Body of Query Request>  ::=  <Handle>
                                      <IndexList>
                                      <TypeList>
```

where

<Handle>
A UTF8-String (as defined in section 2.1.4) that specifies the handle to be resolved.

<IndexList>
A 4-byte unsigned integer followed by an array of 4-byte unsigned integers. The first integer indicates the number of integers in the integer array. Each number in the integer array is a handle value index and refers to a handle value to be retrieved. The client sets the first integer to zero (followed by an empty array) to ask for all the handle values regardless of their index.

<TypeList>
A 4-byte unsigned integer followed by a list of UTF8-Strings. The first integer indicates the number of UTF8-Strings in the list that follows. Each UTF8-String in the list specifies a data type. This tells the server to return all handle values whose data type is listed in the list. If a UTF8-String ends with the '.' (0x2E) character, the server must return all handle values whose data type is under the type hierarchy specified in the UTF8-String. The <TypeList> may contain no UTF8-String if the first integer is 0. In this case, the server must return all handle values regardless of their data type.

If a query request does not specify any index or data type and the PO flag (in the Message Header) is set, the server will return all the handle values that have the PUBLIC_READ permission. Clients can also send queries without the PO flag set. In this case, the server will return all the handle values with PUBLIC_READ permission and all the handle values with ADMIN_READ permission. If the query requests a specific handle value via the value index and the value does not have PUBLIC_READ permission, the server should accept the request (and authenticate the client) even if the request has its PO flag set.

If a query consists of a non-empty <IndexList> but an empty <TypeList>, the server should only return those handle values whose indexes are listed in the <IndexList>. Likewise, if a query consists of a non-empty <TypeList> but an empty <IndexList>, the server should only return those handle values whose data types are listed in the <TypeList>.

When both <IndexList> and <TypeList> fields are non-empty, the server should return all handle values whose indexes are listed in the <IndexList> AND all handle values whose data types are listed in the <TypeList>.

3.2.2. Successful Query Response

The Message Header of any query response must set its <OpCode> to OC_RESOLUTION. A successful query response must set its <ResponseCode> to RC_SUCCESS.

The message body of the successful query response is defined as follows:

```
<Message Body of Successful Query Response> ::= [<RequestDigest>]
                                                <Handle>
                                                <ValueList>
```

where

<RequestDigest>
Optional field as defined in section 2.2.3.

<Handle>
A UTF8-String that specifies the handle queried by the client.

<ValueList>
A 4-byte unsigned integer followed by a list of handle values. The integer specifies the number of handle values in the list. The encoding of each handle value follows the specification given in [2] (see section 3.1). The integer is set to zero if there is no handle value that satisfies the query.

3.2.3. Unsuccessful Query Response

If a server cannot fulfill a client's request, it must return an error message. The general format for any error message from the server is specified in section 3.3 of this document.

For example, a server must return an error message if the queried handle does not exist in its database. The error message will have an empty message body and have its <ResponseCode> set to RC_HANDLE_NOT_FOUND.

Note that a server should NOT return an RC_HANDLE_NOT_FOUND message if the server is not responsible for the handle being queried. It is possible that the queried handle exists but is managed by another handle server (under some other handle service). When this happens, the server should either send a service referral (see section 3.4) or simply return an error message with <ResponseCode> set to RC_SERVER_NOT_RESP.

The server may return an error message with <ResponseCode> set to RC_SERVER_BUSY if the server is too busy to process the request. Like RC_HANDLE_NOT_FOUND, an RC_SERVER_BUSY message also has an empty message body.

Servers should return an RC_ACCESS_DENIED message if the request asks for a specific handle value (via the handle value index) that has neither PUBLIC_READ nor ADMIN_READ permission.

A handle Server may ask its client to authenticate itself as the handle administrator during the resolution. This happens if any handle value in query has ADMIN_READ permission, but no PUBLIC_READ permission. Details of client authentication are described later in this document.

3.3. Error Response from Server

A handle server will return an error message if it encounters an error when processing a request. Any error response from the server must maintain the same <OpCode> (in the message header) as the one in the original request. Each error condition is identified by a unique <ResponseCode> as defined in section 2.2.2.2 of this document.

The Message Body of an error message may be empty. Otherwise it consists of the following data fields (unless otherwise specified):

```
<Message Body of Error Response from Server> ::= [<RequestDigest>]
                                                  <ErrorMessage>
                                                 [ <IndexList> ]
```

where

<RequestDigest>
Optional field as defined in section 2.2.3.

<ErrorMessage>
A UTF8-String that explains the error.

<IndexList>
An optional field. When not empty, it consists of a 4-byte unsigned integer followed by a list of handle value indexes. The first integer indicates the number of indexes in the list. Each index in the list is a 4-byte unsigned integer that refers to a handle value that contributed to the error. An example would be a server that is asked to add three handle values, with indexes 1, 2, and 3, and handle values with indexes of 1 and 2 already in existence. In this case, the server could return an error message with <REsponseCode> set to RC_VALUE_ALREADY_EXIST and add index 1 and 2 to the <IndexList>. Note that the server is not obligated to return the complete list of handle value indexes that may have caused the error.

3.4. Service Referral

A handle server may receive requests for handles that are managed by some other handle server or service. When this happens, the server has the option to either return a referral message that directs the client to the proper handle service, or simply return an error message with <ResponseCode> set to RC_SERVER_NOT_RESP. Service referral also happens when ownership of handles moves from one handle service to another. It may also be used by any local handle service to delegate its service into multiple service layers.

The Message Header of a service referral must maintain the same <OpCode> as the one in the original request and set its <ResponseCode> to RC_SERVICE_REFERRAL.

The Message Body of any service referral is defined as follows:

```
<Message Body of Service Referral> ::= [ <RequestDigest> ]
                                         <ReferralHandle>
                                       [ <ValueList> ]
```

where

<RequestDigest>
Optional field as defined in section 2.2.3.

<ReferralHandle>
A UTF8-String that identifies the handle (e.g., a service handle) that maintains the referral information (i.e., the service information of the handle service in which this refers). If the <ReferralHandle> is set to "0.NA/0.NA", it is referring the client to the GHR.

<ValueList>
An optional field that must be empty if the <ReferralHandle> is provided. When not empty, it consists of a 4-byte unsigned integer, followed by a list of HS_SITE values. The integer specifies the number of HS_SITE values in the list.

Unlike regular query responses that may consist of handle values of any data type, a service referral can only have zero or more HS_SITE values in its <ValueList>. The <ReferralHandle> may contain an empty UTF8-String if the HS_SITE values in the <ValueList> are not maintained by any handle.

Care must be taken by clients to avoid any loops caused by service referrals. It is also the client's responsibility to authenticate the service information obtained from the service referral. A client should always use its own copy of the GHR service information if the <ReferralHandle> is set to "0.NA/0.NA".

3.5. Client Authentication

Clients are asked to authenticate themselves as handle administrators when querying for any handle value with ADMIN_READ but no PUBLIC_READ permission. Client authentication is also required for any handle administration requests that require administrator privileges. This includes adding, removing, or modifying handles or handle values.

Client authentication consists of multiple messages exchanged between the client and server. Such messages include the challenge from the server to the client to authenticate the client, the challenge-response from the client in response to the server's challenge, and

the verification request and response message if secret key authentication takes place. Messages exchanged during the authentication are correlated via a unique <SessionId> assigned by the server. For each authentication session, the server needs to maintain the state information that includes the server's challenge, the challenge-response from the client, as well as the original client request.

The authentication starts with a response message from the server that contains a challenge to the client. The client must respond to the challenge with a challenge-response message. The server validates the challenge-response, either by verifying the digital signature inside the challenge-response, or by sending a verification request to another handle server (herein referred to as the verification server), that maintains the secret key for the administrator. The purpose of the challenge and the challenge-response is to prove to the server that the client possesses the private key (or the secret key) of the handle administrator. If the authentication fails, an error response will be sent back with the <ResponseCode> set to RC_AUTHEN_FAILED.

Upon successful client authentication, the server must also make sure that the administrator is authorized for the request. If the administrator has sufficient privileges, the server will process the request and send back the result. If the administrator does not have sufficient privileges, the server will return an error message with <ResponseCode> set to RC_NOT_AUTHORIZED.

The following sections provide details of each message exchanged during the authentication process.

3.5.1. Challenge from Server to Client

The Message Header of the CHALLENGE must keep the same <OpCode> as the original request and set the <ResponseCode> to RC_AUTH_NEEDED. The server must assign a non-zero unique <SessionId> in the Message Envelope to keep track of the authentication. It must also set the RD flag of the <OpFlag> (see section 2.2.2.3) in the Message Header, regardless of whether the original request had the RD bit set or not.

The Message Body of the server's CHALLENGE is defined as follows:

```
<Message Body of Server's Challenge> ::=  <RequestDigest>
                                          <Nonce>
```

where

<RequestDigest>
Message Digest of the request message, as defined in section 2.2.3.

<Nonce>
A 4-byte unsigned integer followed by a random string generated by the server via a secure random number generator. The integer specifies the number of octets in the random string. The size of the random string should be no less than 20 octets.

Note that the server will not sign the challenge if the client did not request the server to do so. If the client worries about whether it is speaking to the right server, it may ask the server to sign the <Challenge>. If the client requested the server to sign the <Challenge> but failed to validate the server's signature, the client should discard the server's response and reissue the request to the server.

3.5.2. Challenge-Response from Client to Server

The Message Header of the CHALLENGE_RESPONSE must set its <OpCode> to OC_CHALLENGE_RESPONSE and its <ResponseCode> to 0. It must also keep the same <SessionId> (in the Message Envelope) as specified in the challenge from the server.

The Message Body of the CHALLENGE_RESPONSE request is defines as follows:

```
<Message Body of CHALLENGE_RESPONSE> ::=  <AuthenticationType>
                                          <KeyHandle>
                                          <KeyIndex>
                                          <ChallengeResponse>
```

where

<AuthenticationType>
A UTF8-String that identifies the type of authentication key used by the client. For example, the field is set to "HS_SECKEY" if the client chooses to use a secret key for its authentication. The field is set to "HS_PUBKEY" if a public key is used instead.

<KeyHandle>
A UTF8-String that identifies the handle that holds the public or secret key of the handle administrator.

<KeyIndex>
A 4-byte unsigned integer that specifies the index of the handle value (of the <KeyHandle>) that holds the public or secret key of the administrator.

<ChallengeResponse>
Contains either the Message Authentication Code (MAC) or the digital signature over the challenge from the server. If the <AuthenticationType> is "HS_SECKEY", the <ChallengeResponse> consists of an octet followed by the MAC. The octet identifies the algorithm used to generate the MAC. For example, if the first octet is set to 0x01, the MAC is generated by

```
MD5_Hash(<SecretKey> + <ServerChallenge> + <SecretKey>)
```

where the <SecretKey> is the administrator's secret key referenced by the <KeyHandle> and <KeyIndex>. The <ServerChallenge> is the Message Body portion of the server's challenge. If the first octet in the <ChallengeResponse> is set to 0x02, the MAC is generated using

```
SHA-1_Hash(<SecretKey> + <ServerChallenge> + <SecretKey>)
```

A more secure approach is to use HMAC [17] for the <ChallengeResponse>. The HMAC can be generated using the <SecretKey> and <ServerChallenge>. A <ChallengeResponse> with its first octet set to 0x11 indicates that the HMAC is generated using the MD5 algorithm. Likewise, a <ChallengeResponse> with its first octet set to 0x12 indicates that the HMAC is generated using the SHA-1 algorithm.

If the <AuthenticationType> is "HS_PUBKEY", the <ChallengeResponse> contains the digital signature over the Message Body portion of the server's challenge. The signature is generated in two steps: First, a one-way hash value is computed over the blob that is to be signed. Second, the hash value is signed using the private key. The signature consists of a UTF8-String that specifies the digest algorithm used for the signature, followed by the signature over the server's challenge. The <KeyHandle> and

<KeyIndex> refers to the administrator's public key that can be used to verify the signature.

Handle administrators are defined in terms of HS_ADMIN values assigned to the handle. Each HS_ADMIN value defines the set of privileges granted to the administrator. It also provides the reference to the authentication key that can be used to authenticate the administrator. The reference can be made directly if the <AdminRef> field of the HS_ADMIN value refers to the handle value that holds the authentication key. Indirect reference to the authentication key can also be made via administrator groups. In this case, the <AdminRef> field may refer to a handle value of type HS_VLIST. An HS_VLIST value defines an administrator group via a list of handle value references, each of which refers to the authentication key of a handle administrator.

For handles with multiple HS_ADMIN values, the server will have to check each of those with sufficient privileges to see if its <AdminRef> field matches the <KeyHandle> and <KeyIndex>. If no match is found, but there are administrator groups defined, the server must check if the <KeyHandle> and <KeyIndex> belong to any of the administrator groups that have sufficient privileges. An administrator group may contain another administrator group as a member. Servers must be careful to avoid infinite loops when navigating these groups.

If the <KeyHandle> and <KeyIndex> are not referenced by any of the HS_ADMIN values, or the administrator group that has sufficient privileges, the server will return an error message with <ResponseCode> set to RC_NOT_AUTHORIZED. Otherwise, the server will continue to authenticate the client as follows:

If the <AuthenticationType> is "HS_PUBKEY", the server will retrieve the administrator's public key based on the <KeyHandle> and <KeyIndex>. The public key can be used to verify the <ChallengeResponse> against the server's <Challenge>. If the <ChallengeResponse> matches the <Challenge>, the server will continue to process the original request and return the result. Otherwise, the server will return an error message with <ResponseCode> set to RC_AUTHENTICATION_FAILED.

If the <AuthenticationType> is "HS_SECKEY", the server will have to send a verification request to the verification server; that is, the handle server that manages the handle referenced by the <KeyHandle>. The verification request and its response are defined in the following sections. The verification server will verify the <ChallengeResponse> against the <Challenge> on behalf of the handle server.

3.5.3. Challenge-Response Verification-Request

The message header of the VERIFICATION_REQUEST must set its <OpCode> to OC_VERIFY_CHALLENGE and the <ResponseCode> to 0.

The message body of the Verification-Request is defined as follows:

```
<Message Body of VERIFICATION_REQUEST> ::=  <KeyHandle>
                                            <KeyIndex>
                                            <Challenge>
                                            <ChallengeResponse>
```

where

<KeyHandle>
A UTF8-String that refers to the handle that holds the secret key of the administrator.

<KeyIndex>
A 4-byte unsigned integer that is the index of the handle value that holds the secret key of the administrator.

<Challenge>
The message body of the server's challenge, as described in section 3.5.1.

<ChallengeResponse>
The <ChallengeResponse> from the client in response to the server's <Challenge>, as defined in section 3.5.2.

Any Challenge-Response Verification-Request must set its CT bit in the message header. This is to ensure that the verification server will sign the Verification-Response as specified in the next section.

3.5.4. Challenge-Response Verification-Response

The Verification-Response tells the requesting handle server whether the <ChallengeResponse> matches the <Challenge> in the Verification-Request.

The Message Header of the Verification-Response must set its <ResponseCode> to RC_SUCCESS whether or not the <ChallengeResponse> matches the <Challenge>. The RD flag in the <OpFlag> field should also be set (to 1) since the <RequestDigist> will be mandatory in the Message Body.

The Message Body of the Verification-Response is defined as follows:

```
<Challenge-Response Verification-Response>
                            ::= <RequestDigest>
                                <VerificationResult>
```

where

<RequestDigest>
Contains the message digest of the Verification-Request.

<VerificationResult>
An octet that is set to 1 if the <ChallengeResponse> matches the <Challenge>. Otherwise it must be set to 0.

The verification server may return an error with <ResponseCode> set to RC_AUTHEN_FAILED if it cannot perform the verification (e.g., the <KeyHandle> does not exist, or the <KeyHandle> and <KeyIndex> refer to an invalid handle value). When this happens, the server that performs the client authentication should relay the same error message back to the client.

3.6. Handle Administration

The Handle System protocol supports a set of handle administration functions that include adding, deleting, and modifying handles or handle values. Before fulfilling any administration request, the server must authenticate the client as the handle administrator that is authorized for the administrative operation. Handle administration can only be carried out by the primary handle server.

3.6.1. Add Handle Value(s)

Clients add values to existing handles by sending ADD_VALUE requests to the responsible handle server. The Message Header of the ADD_VALUE request must set its <OpCode> to OC_ADD_VALUE.

The Message Body of the ADD_VALUE request is encoded as follows:

```
<Message Body of ADD_VALUE Request> ::=  <Handle>
                                         <ValueList>
```

where

<Handle>
A UTF8-String that specifies the handle.

<ValueList>
A 4-byte unsigned integer followed by a list of handle values. The integer indicates the number of handle values in the list.

The server that receives the ADD_VALUE request must first authenticate the client as the administrator with the ADD_VALUE privilege. Upon successful authentication, the server will proceed to add each value in the <ValueList> to the <Handle>. If successful, the server will return an RC_SUCCESS message to the client.

Each ADD_VALUE request must be carried out as a transaction. If adding any value in the <ValueList> raises an error, the entire operation must be rolled back. For any failed ADD_VALUE request, none of the values in the <ValueList> should be added to the <Handle>. The server must also send a response to the client that explains the error. For example, if a value in the <ValueList> has the same index as one of the existing handle values, the server will return an error message that has the <ResponseCode> set to RC_VALUE_ALREADY_EXISTS.

ADD_VALUE requests can also be used to add handle administrators. This happens if the <ValueList> in the ADD_VALUE request contains any HS_ADMIN values. The server must authenticate the client as an administrator with the ADD_ADMIN privilege before fulfilling such requests.

An ADD_VALUE request will result in an error if the requested handle does not exist. When this happens, the server will return an error message with <ResponseCode> set to RC_HANDLE_NOT_EXIST.

3.6.2. Remove Handle Value(s)

Clients remove existing handle values by sending REMOVE_VALUE requests to the responsible handle server. The Message Header of the REMOVE_VALUE request must set its <OpCode> to OC_REMOVE_VALUE.

The Message Body of any REMOVE_VALUE request is encoded as follows:

```
<Message Body of REMOVE_VALUE Request> ::=  <Handle>
                                            <IndexList>
```

where

<Handle>
A UTF8-String that specifies the handle whose value(s) needs to be removed.

<IndexList>
A 4-byte unsigned integer followed by a list of handle value indexes. Each index refers to a handle value to be removed from the <Handle>. The integer specifies the number of indexes in the list. Each index is also encoded as a 4-byte unsigned integer.

The server that receives the REMOVE_VALUE request must first authenticate the client as the administrator with the REMOVE VALUE privilege. Upon successful authentication, the server will proceed to remove the handle values specified in the <IndexList> from the <Handle>. If successful, the server will return an RC_SUCCESS message to the client.

Each REMOVE_VALUE request must be carried out as a transaction. If removing any value specified in the <IndexList> raises an error, the entire operation must be rolled back. For any failed REMOVE_VALUE request, none of values referenced in the <IndexList> should be removed from the <Handle>. The server must also send a response to the client that explains the error. For example, attempts to remove any handle value with neither PUB_WRITE nor ADMIN_WRITE permission will result in an RC_ACCESS_DENIED error. Note that a REMOVE_VALUE request asking to remove a non-existing handle value will not be treated as an error.

REMOVE_VALUE requests can also be used to remove handle administrators. This happens if any of the indexes in the <IndexList> refer to an HS_ADMIN value. Servers must authenticate the client as an administrator with the REMOVE_ADMIN privilege before fulfilling such requests.

3.6.3. Modify Handle Value(s)

Clients can make modifications to an existing handle value by sending MODIFY_VALUE requests to the responsible handle server. The Message Header of the MODIFY_VALUE request must set its <OpCode> to OC_MODIFY_VALUE.

The Message Body of any MODIFY_VALUE request is defined as follows:

```
<Message Body of MODIFY_VALUE Response> ::= <Handle>
                                            <ValueList>
```

where

<Handle>
A UTF8-String that specifies the handle whose value(s) needs to be modified.

<ValueList>
A 4-byte unsigned integer followed by a list of handle values. The integer specifies the number of handle values in the list. Each value in the <ValueList> specifies a handle value that will replace the existing handle value with the same index.

The server that receives the MODIFY_VALUE request must first authenticate the client as an administrator with the MODIFY_VALUE privilege. Upon successful authentication, the server will proceed to replace those handle values listed in the <ValueList>, provided each handle value has PUB_WRITE or ADMIN_WRITE permission. If successful, the server must notify the client with an RC_SUCCESS message.

Each MODIFY_VALUE request must be carried out as a transaction. If replacing any value listed in the <ValueList> raises an error, the entire operation must be rolled back. For any failed MODIFY_VALUE request, none of values in the <ValueList> should be replaced. The server must also return a response to the client that explains the error. For example, if a MODIFY_VALUE requests to remove a handle value that has neither PUB_WRITE nor ADMIN_WRITE permission, the server must return an error message with the <ResponseCode> set to RC_ACCESS_DENIED. Any MODIFY_VALUE request to replace non- existing handle values is also treated as an error. In this case, the server will return an error message with <ResponseCode> set to RC_VALUE_NOT_FOUND.

MODIFY_VALUE requests can also be used to update handle administrators. This happens if both the values in the <ValueList> and the value to be replaced are HS_ADMIN values. Servers must authenticate the client as an administrator with the MODIFY_ADMIN privilege before fulfilling such a request. It is an error to replace a non-HS_ADMIN value with an HS_ADMIN value. In this case, the server will return an error message with <ResponseCode> set to RC_VALUE_INVALID.

3.6.4. Create Handle

Clients can create new handles by sending CREATE_HANDLE requests to the responsible handle server. The Message Header of any CREATE_HANDLE request must set its <OpCode> to OC_CREATE_HANDLE.

The Message Body of any CREATE_HANDLE request is defined as follows:

```
<Message Body of CREATE_HANDLE Response> ::= <Handle>
                                             <ValueList>
```

where

<Handle>
A UTF8-String that specifies the handle.

<ValueList>
A 4-byte unsigned integer followed by a list of handle values. The integer indicates the number of handle values in the list. The <ValueList> should at least include one HS_ADMIN value that defines the handle administrator.

Only naming authority administrators with the CREATE_HANDLE privilege are allowed to create new handles under the naming authority. The server that receives a CREATE_HANDLE request must authenticate the client as the administrator of the corresponding naming authority handle and make certain that the administrator is authorized to create handles under the naming authority. This is different from the ADD_VALUE request where the server authenticates the client as an administrator of the handle. Upon successful authentication, the server will proceed to create the new handle and add each value in the <ValueList> to the new <Handle>. If successful, the server will return an RC_SUCCESS message to the client.

Each CREATE_HANDLE request must be carried out as a transaction. If any part of the CREATE_HANDLE process fails, the entire operation can be rolled back. For example, if the server fails to add values in the <ValueList> to the new handle, it must return an error message without creating the new handle. Any CREATE_HANDLE request that asks to create a handle that already exists will be treated as an error. In this case, the server will return an error message with the <ResponseCode> set to RC_HANDLE_ALREADY_EXIST.

CREATE_HANDLE requests can also be used to create naming authorities. Naming authorities are created as naming authority handles at the GHR. Before creating a new naming authority handle, the server must authenticate the client as the administrator of the parent naming authority. Only administrators with the CREATE_NA privilege are allowed to create any sub-naming authority. Root level naming authorities may be created by the administrator of the root handle "0.NA/0.NA".

3.6.5. Delete Handle

Clients delete existing handles by sending DELETE_HANDLE requests to the responsible handle server. The Message Header of the DELETE_HANDLE request must set its <OpCode> to OC_DELETE_HANDLE.

The Message Body of any DELETE_HANDLE request is defined as follows:

<Message Body of DELETE_HANDLE Request> ::= <Handle>

where

<Handle>
A UTF8-String that specifies the handle.

The server that receives the DELETE_HANDLE request must first authenticate the client as the administrator with the DELETE_HANDLE privilege. Upon successful authentication, the server will proceed to delete the handle along with any handle values assigned to the handle. If successful, the server will return an RC_SUCCESS message to the client.

Each DELETE_HANDLE request must be carried out as a transaction. If any part of the DELETE_HANDLE process fails, the entire operation must be rolled back. For example, if the server fails to remove any handle values assigned to the handle (before deleting the handle), it must return an error message without deleting the handle. This may happen if the handle contains a value that has neither PUB_WRITE nor ADMIN_WRITE permission. In this case, the server will return an error message with the <ResponseCode> set to RC_PERMISSION_DENIED. A DELETE_HANDLE request that asks to delete a non-existing handle will also be treated as an error. The server will return an error message with the <ResponseCode> set to RC_HANDLE_NOT_EXIST.

DELETE_HANDLE requests can also be used to delete naming authorities. This is achieved by deleting the corresponding naming authority handle on the GHR. Before deleting a naming authority handle, the server must authenticate the client as the administrator of the naming authority handle. Only administrators with the DELETE_NA privilege are allowed to delete the naming authority. Root level naming authorities may be deleted by the administrator of the root handle "0.NA/0.NA".

3.7. Naming Authority (NA) Administration

The Handle System manages naming authorities via naming authority handles. Naming authority handles are managed by the GHR. Clients can change the service information of any naming authority by changing the HS_SITE values assigned to the corresponding naming authority handle. Creating or deleting naming authorities is done by creating or deleting the corresponding naming authority handles. Root level naming authorities may be created or deleted by the administrator of the root handle "0.NA/0.NA". Non-root-level naming authorities may be created by the administrator of its parent naming authority.

For example, the administrator of the naming authority handle "0.NA/10" may create the naming authority "10.1000" by sending a CREATE_HANDLE request to the GHR to create the naming authority handle "0.NA/10.1000". Before fulfilling the request, the server at the GHR must authenticate the client as the administrator of the parent naming authority, that is, the administrator of the naming authority handle "0.NA/10". The server must also make sure that the administrator has the NA_CREATE privilege.

The Handle protocol also allows clients to list handles or sub-naming authorities under a naming authority. Details of these operations are described in the following sections.

3.7.1. List Handle(s) under a Naming Authority

Clients send LIST_HANDLE requests to handle servers to get a list of handles under a naming authority. The Message Header of the LIST_HANDLE request must set its <OpCode> to OC_LIST_HANDLE.

The Message Body of any LIST_HANDLE request is defined as follows:

```
<Message Body of LIST_HANDLE Request> ::= <NA_Handle>

   where

      <NA_Handle>
      A UTF8-String that specifies the naming authority handle.
```

To obtain a complete list of the handles, the request must be sent to every handle server listed in one of the service sites of the responsible handle service. Each server within the service site will return its own list of handles under the naming authority. The Message Body of a successful LIST_HANDLE response (from each handle server) is defined as follows:

```
<Message Body of LIST_HANDLE Response>  ::=  <Num_Handles>
                                             <HandleList>
```

where

<Num_Handles>
Number of handles (managed by the handle server) under the naming authority.

<HandleList>
A list of UTF8-Strings, each of which identify a handle under the naming authority.

The LIST_HANDLE request may potentially slow down the overall system performance. A handle service (or its service site) has the option of whether or not to support such request. The server will return an RC_OPERATION_DENIED message if LIST_HANDLE is not supported. The server that receives a LIST_HANDLE request should authenticate the client as a naming authority administrator with the LIST_HANDLE privilege before fulfilling the request.

3.7.2. List Sub-Naming Authorities under a Naming Authority

Clients send LIST_NA requests to handle servers to get a list of sub-naming authorities under a naming authority. The Message Header of the LIST_NA request must set its <OpCode> to OC_LIST_NA.

The Message Body of any LIST_NA request is defined as follows:

```
<Message Body of LIST_HANDLE Request> ::= <NA_Handle>
```

where

<NA_Handle>
A UTF8-String that specifies the naming authority handle.

To obtain a complete list of the sub-naming authorities, the request must be sent to every handle server listed in any one of the service sites of the GHR. Each server within the service site will return its own list of sub-naming authority handles under the given naming authority. The Message Body of a successful LIST_NA response (from each handle server) is defined as follows:

```
<Message Body of LIST_HANDLE Response> ::=  <Num_Handles>
                                            <HandleList>
```

where

<Num_Handles>
Number of handles (managed by the handle server) under the naming authority.

<HandleList>
A list of UTF8-Strings, each of which identifies a sub-naming authority user-specified naming authority.

LIST_NA requests must be sent to servers under the GHR that manages all the naming authority handles. The LIST_NA request may potentially slow down the overall system performance, especially the GHS. A server (or service sites) under the GHR has the option of whether or not to support such requests. The server will return an RC_OPERATION_DENIED message if LIST_NA is not supported. The server that receives a LIST_HANDLE request should authenticate the client as a naming authority administrator with the LIST_NA privilege before fulfilling the request.

3.8. Session and Session Management

Sessions are used to allow sharing of authentication information or network resources among multiple protocol operations. For example, a naming authority administrator may authenticate itself once through the session setup, and then register multiple handles under the session.

A client may ask the server to establish a session key and use it for subsequent requests. A session key is a secret key that is shared by the client and server. It can be used to authenticate or encrypt any message exchanged under the session. A session is encrypted if every message exchanged within the session is encrypted using the session key.

Sessions may be established as the result of an explicit OC_SESSION_SETUP request from a client. A server may also automatically setup a session when multiple message exchanges are expected to fulfill a request. For example, the server will automatically establish a session if it receives a CREATE_HANDLE request that requires client authentication.

Every session is identified by a non-zero Session ID that appears in the Message Header. Servers are responsible for generating a unique Session ID for each outstanding session. Each session may have a set of state information associated with it. The state information may

include the session key and the information obtained from client authentication, as well as any communication options. Servers and clients are responsible for keeping the state information in sync until the session is terminated.

A session may be terminated with an OC_SESSION_TERMINATE request from the client. Servers may also terminate a session that has been idle for a significant amount of time.

3.8.1. Session Setup Request

Clients establish a session with a handle server with a SESSION_SETUP request. A SESSION_SETUP request can also be used to update any state information associated to an existing session. The Message Header of the SESSION_SETUP request must have its <OpCode> set to OC_SESSION_SETUP and <ResponseCode> to 0.

The Message Body of any SESSION_SETUP request is defined as follows:

<SESSION_SETUP Request Message Body> ::= <SessionAttributes>

where

<SessionAttributes>
A 4-byte unsigned integer followed by a list of session attributes. The integer indicates the number of session attributes in the list. Possible session attributes include the <HS_SESSION_IDENTITY>, the <HS_SESSION_TIMEOUT>, and the <HS_SESSION_KEY_EXCHANGE>. Each of these attributes is defined as follows:

<HS_SESSION_IDENTITY> ::= <Key>
<Handle>
<ValueIndex>

where

<Key>
A UTF-8 string constant "HS_SESSION_IDENTITY".

<Handle>
<ValueIndex>
A UTF-8 string followed by a 4-byte unsigned integer that specifies the handle and the handle value used for client authentication. It must refer to a handle value that contains the public key of the client. The public key is used by the server to authenticate the client.

```
<HS_SESSION_KEY_EXCHANGE> ::= <Key>
                              <KeyExchangeData>
   where

      <Key>
      A UTF-8 string constant "HS_SESSION_KEY_EXCHANGE".

      <KeyExchangeData>
      One of the these tuples: <ClientCipher
      <ClientCipher KeyExchange>,
      <HdlCipher KeyExchange>, or
      <ServerCipher KeyExchange>.
      Each of these tuples is defined as follows:

      <ClientCipher KeyExchange> ::= <Key>
                                     <PubKey>
         where

            <Key>
            A UTF-8 string constant "CLIENT_CIPHER".

            <PubKey>
            A public key provided by the client and used
            by the server to encrypt the session key.

      <HdlCipher KeyExchange> ::= <Key>
                                  <ExchangeKeyHdl>
                                  <ExchangeKeyIndex>
         where

            <Key>
            A UTF-8 string constant "HDL_CIPHER".

            <ExchangeKeyHdl>
            <ExchangeKeyIndex>
            A UTF-8 string followed by a 4-byte unsigned
            integer.  The <ExchangeKeyHdl> and
            <ExchangeKeyIndex> refers to a handle value
            used for session key exchange.  The handle
            value must contain the public key of the
            client.  The public key will be used by the
            server to encrypt the session key before
            sending it to the client.

      <ServerCipher KeyExchange> ::= <Key>

         where
```

<Key>
A UTF-8 string constant "SERVER_CIPHER". This tells the server that the client will be responsible for generating the session key. The server will have to provide its public key in the response message and set the <ResponseCode> to RC_SESSION_EXCHANGEKEY. The client can use the server's public key to encrypt the session key and send it to the server via a subsequent SESSION_EXCHANGEKEY request.

<DiffieHellman KeyExchange> ::= <Key>
<DHParams>

where

<Key>
A UTF-8 string constant "DIFFIE_HELLMAN"

<DHParams>
The values used as input in the Diffie-Hellman algorithm. It consists of three big integers of variable length. Each big integer is encoded in terms of a 4-byte unsigned integer followed by an octet string. The octet string contains the big integer itself. The 4-byte unsigned integer specifies the number of octets of the octet string.

<HS_SESSION_TIMEOUT> ::= <Key>
<TimeOut>

where

<Key>
A UTF-8 string constant "HS_SESSION_TIMEOUT".

<TimeOut>
A 4-byte unsigned integer that specifies the desired duration of the session in seconds.

Note that it should be treated as an error if the same session attribute is listed multiple times in the <SessionAttribute> field. When this happens, the server should return an error message with <ResponseCode> set to RC_PROTOCOL_ERROR.

A SESSION_SETUP_REQUEST can be used to change session attributes of any established session. This happens if the <SessionId> is non-zero

and matches one of the established sessions. Care must be taken by the server to prevent any unauthorized request from changing the session attributes. For example, an encrypted session may only be changed into an unencrypted session by a SESSION_SETUP_REQUEST with an appropriate MAC in its Message Credential.

3.8.2. Session Setup Response

The Message Header of the SESSION_SETUP response must set its <OpCode> to OC_SESSION_SETUP. The <ResponseCode> of the SESSION_SETUP response varies according to the SESSION_SETUP request. It must be set to RC_SUCCESS if the SESSION_SETUP request is successful and the server does not expect a session key to be returned by the client.

The Message Body of the SESSION_SETUP response is empty unless the request is asking for <HS_SESSION_KEY_EXCHANGE>. In this case, the Message Body of the SESSION_SETUP response may contain the encrypted session key from the server, or the server's public key, to be used for session key exchange. The exact format depends on the content of the <HS_SESSION_KEY_EXCHANGE> in the SESSION_SETUP request. If <ClientCipher KeyExchange> or <HdlCipher KeyExchange> is given in the SESSION_SETUP request, the Message Body of the SESSION_SETUP response will contain the encrypted session key from the server and is defined as follows:

```
<Message Body of SESSION_SETUP Response>
                                    ::= <RequestDigest>
                                        <EncryptedSessionKey>
                                      [ <EncryptionAlgorithm> ]
```

where

<RequestDigest>
Message digest of the SESSION_SETUP request is as specified in section 2.2.3.

<EncryptedSessionKey>
Session key is encrypted using the public key provided in the SESSION_SETUP request. The session key is a randomly generated octet string from the server. The server will only return the <EncryptedSessionKey> if the <KeyExchangeData> in the SESSION_SETUP request provides the public key from the client.

<EncryptionAlgorithm>
(optional) UTF-8 string that identifies the encryption algorithm used by the session key.

If <ServerCipher KeyExchange> is given in the SESSION_SETUP request, the server must provide its public key in the SESSION_SETUP response. The public key can be used by the client in a subsequent SESSION_EXCHANGEKEY request (defined below) for session key exchange. In this case, the Message Header of the SESSION_SETUP response must set its <ResponseCode> to RC_SESSION_EXCHANGEKEY. The Message Body of the SESSION_SETUP response must include the server's public key and is defined as follows:

```
<Message Body of SESSION_SETUP response>
                          ::= <RequestDigest>
                              <Public Key for Session Key Exchange>
```

where

<RequestDigest>
Message digest of the SESSION_SETUP request as specified in section 2.2.3.

<Public Key for Session Key Exchange>
Public key from the server to be used for session key exchange. It is encoded in the same format as the <PublicKey> record in the HS_SITE value (see section 3.2.2 in [2]).

3.8.3. Session Key Exchange

If the <ResponseCode> of a SESSION_SETUP response is RC_SESSION_EXCHANGEKEY, the client is responsible for generating the session key and sending it to the server. In this case, the client can generate a session key, encrypt it with the public key provided by the server in the SESSION_SETUP response, and send the encrypted session key to the server in a SESSION_EXCHANGEKEY request.

The Message Header of the SESSION_EXCHANGEKEY request must set its <OpCode> to OC_SESSION_EXCHANGEKEY and its <ResponseCode> to 0. The Message Body of the SESSION_EXCHANGEKEY request is defined as follows:

```
<Message Body of OC_SESSION_EXCHANGEKEY>
                 ::=   <Encrypted Session Key>
                     [ <EncryptionAlgorithm> ]
```

where

<EncryptedSessionKey>
Session key encrypted using the public key provided in the SESSION_SETUP response. The session key is a randomly generated octet string by the client.

<EncryptionAlgorithm>
(optional) UTF-8 string that identifies the encryption algorithm used by the session key.

During the session key exchange, the server receiving the exchange key or session key has the responsibility of ensuring that the key meets the security requirements defined in its local policy. If the server considers the key being volunable, it must return an error message to the client with <ResponseCode> set to RC_SESSION_KEY_INVALID.

3.8.4. Session Termination

Clients can terminate a session with a SESSION_TERMINATE request. The Message Header of a SESSION_TERMINATE request must have its <OpCode> set to OC_SESSION_TERMINATE and its <ResponseCode> to 0. The message body of any SESSION_TERMINATE request must be empty.

The server must send a SESSION_TERMINATE response to the client after the session is terminated. The server should only terminate the session after it has finished processing all the requests (under the session) that were submitted before the Session Termination request.

The message header of the SESSION_TERMINATE response must set its <OpCode> to OC_SESSION_TERMINATE. A successful SESSION_TERMINATE response must have its <ResponseCode> set to RC_SUCCESS, and an empty message body.

4. Implementation Guidelines

4.1. Server Implementation

The optimal structure for any handle server will depend on the host operating system. This section only addresses those implementation considerations that are common to most handle servers.

A good server implementation should allow easy configuration or fine-tuning. A suggested list of configurable items includes the server's network interface(s) (e.g., IP address, port number, etc.), the number of concurrent processes/threads allowed, time-out intervals for any TCP connection and/or authentication process, re-try policy under UDP connection, policies on whether to support recursive service, case-sensitivity for ASCII characters, and different levels of transaction logging, etc.

All handle server implementations must support all the handle data types as defined in the "Handle System Namespace and Service Definition" [2]. They should also be able to store handle values of any application defined data type.

A handle server must support multiple concurrent activities, whether they are implemented as separate processes or threads in the host's operating system, or multiplexed inside a single name server program. A handle server should not block the service of UDP requests while it waits for TCP data or other query activities. Similarly, a handle server should not attempt to provide recursive service without processing such requests in parallel, though it may choose to serialize requests from a single client, or to regard identical requests from the same client as duplicates.

4.2. Client Implementation

Clients should be prepared to receive handle values of any data type. Clients may choose to implement a callback interface to allow new modules or plug-ins to be added to support any application-defined data types.

Clients that follow service referrals or handle aliases must avoid falling into an infinite loop. They should not repeatedly contact the same server for the same request with the same target entry. A client may choose to use a counter that is incremented each time it follows a service referral or handle alias. There should be a configurable upper limit to the counter to control the levels of service referrals or handle aliases followed by the client.

Clients that provide some caching can expect much better performance than those that do not. Client implementations should always consider caching the service information associated with a naming authority. This will reduce the number of roundtrips for subsequent handle requests under the same naming authority.

5. Security Considerations

The overall Handle System security considerations are discussed in "Handle System Overview" [1]; that discussion applies equally to this document. Security considerations regarding the Handle System data model and service model are discussed in "Handle System Namespace and Service Definition" [2].

For efficiency, the Handle protocol includes a simple challenge-response authentication protocol for basic client authentication. Handle servers are free to provide additional authentication mechanisms (e.g., SASL) as needed. Details of this will be discussed in a separate document.

Data integrity under the Handle protocol is achieved via the server's digital signature. Care must be taken to protect the server's private key from any impersonation attack. Any change to the server's public key pair must be registered (in terms of service information) with the GHR.

6. Acknowledgements

This work is derived from the earlier versions of the Handle System implementation. The overall digital object architecture, including the Handle System, was described in a paper by Robert Kahn and Robert Wilensky [22] in 1995. Development continued at CNRI as part of the Computer Science Technical Reports (CSTR) project, funded by the Defense Advanced Projects Agency (DARPA) under Grant Number MDA-972-92-J-1029 and MDA-972-99-1-0018. Design ideas are based on those discussed within the Handle System development team, including David Ely, Charles Orth, Allison Yu, Sean Reilly, Jane Euler, Catherine Rey, Stephanie Nguyen, Jason Petrone, and Helen She. Their contributions to this work are gratefully acknowledged.

The authors also thank Russ Housley (housley@vigilsec.com), Ted Hardie (hardie@qualcomm.com), and Mark Baugher (mbaugher@cisco.com) for their extensive review and comments, as well as recommendations received from other members of the IETF/IRTF community.

7. Informative References

[1] Sun, S. and L. Lannom, "Handle System Overview", RFC 3650, November 2003.

[2] Sun, S., Reilly, S. and L. Lannom, "Handle System Namespace and Service Definition", RFC 3651, November 2003.

[3] Yergeau, F., "UTF-8, a transformation format of ISO 10646", RFC 2279, January 1998.

[4] A. Freier, P. Karlton, P. Kocher "The SSL Protocol Version 3.0"

[5] RSA Laboratories, "Public-Key Cryptography Standard PKCS#7" http://www.rsasecurity.com/rsalabs/pkcs/

[6] U.S. Federal Information Processing Standard: Digital Signature Standard.

[7] Housley, R., "Cryptographic Message Syntax (CMS) Algorithms", RFC 3370, August 2002.

[8] Braden, R., "FTP Data Compression", RFC 468, March 1973.

[9] Rivest, R., "The MD5 Message-Digest Algorithm", RFC 1321, April 1992.

[10] NIST, FIPS PUB 180-1: Secure Hash Standard, April 1995.

[11] D. Cohen, "On Holy Wars and a Plea for Peace", Internet Experiment, Note IEN 137, 1 April 1980.

[12] Balakrishnan, H. and S. Seshan, "The Congestion Manager", RFC 3124, June 2001.

[13] R. Kahn, R. Wilensky, "A Framework for Distributed Digital Object Services, May 1995, http://www.cnri.reston.va.us/k-w.html

[14] Polk, W., Housley, R. and L. Bassham, "Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 3279, April 2002.

[15] Housley, R., Polk, W., Ford, W. and D. Solo, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 3280, April 2002.

[16] M. Bellare and P. Rogaway. The Exact Security of Digital Signatures - How to Sign with RSA and Rabin. In Advances in Cryptology-Eurocrypt ’96, pp.399-416, Springer-Verlag, 1996.

[17] Krawczyk, H., Bellare, M. and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.

[18] R. Kahn, R. Wilensky, "A Framework for Distributed Digital Object Services, May 1995, http://www.cnri.reston.va.us/k-w.html

8. Authors' Addresses

Sam X. Sun
Corporation for National Research Initiatives (CNRI)
1895 Preston White Dr., Suite 100
Reston, VA 20191

Phone: 703-262-5316
EMail: ssun@cnri.reston.va.us

Sean Reilly
Corporation for National Research Initiatives (CNRI)
1895 Preston White Dr., Suite 100
Reston, VA 20191

Phone: 703-620-8990
EMail: sreilly@cnri.reston.va.us

Larry Lannom
Corporation for National Research Initiatives (CNRI)
1895 Preston White Dr., Suite 100
Reston, VA 20191

Phone: 703-262-5307
EMail: llannom@cnri.reston.va.us

Jason Petrone
Corporation for National Research Initiatives (CNRI)
1895 Preston White Dr., Suite 100
Reston, VA 20191

Phone: 703-262-5340
EMail: jpetrone@cnri.reston.va.us

9. Full Copyright Statement

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.