INTERNET-DRAFT
draft-irtf-idrm-handle-system-protocol-01.txt

Sam X. Sun
Sean Reilly
Larry Lannom
Jing Shi
CNRI
May, 2001

Handle System Protocol (ver 2.0) Specification

<draft-irtf-idrm-handle-system-protocol-01.txt>

Status of this Memo

Abstract

The Handle System is a general-purpose global name service that allows secured name resolution and administration over the public Internet. The Handle System manages handles, which are unique names for digital objects and other Internet resources. This document specifies the protocol used between handle system clients and servers for handle resolution and administration. 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" document [2].

1. Introduction

The handle system employs 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 handle client to handle server communication and does not include a description of the

procedures and protocols used to manage handle servers or services. A discussion of these procedures and protocols is out of the scope of this document and may be made available in a future document.

The Handle System consists of a set of service components, as defined in the "Handle System Namespace and Service Definition" [2]. From the client's point of view, the Handle System manages a distributed database of handles and their values. Each handle under the Handle System is managed by its individual service component. The handle system protocol specifies the procedure for a client to locate the responsible handle server within the service component. It also defines the messages exchanged between the client and server for any handle operations.

The handle system protocol is an application level protocol, defined according to the data and service model described in the "Handle System Namespace and Service Definition" [2]. It is assumed that readers are familiar with the concepts and definitions, especially the data model and service model, specified in that document.

Some key aspects of the handle system protocol include:

- The handle system protocol supports both handle resolution and administration. The protocol follows the access control and administration model defined in the "Handle System Namespace and Service Definition" [2].

- A client may verify the integrity of any server response via the server's digital signature.

- A server may authenticate its client as handle administrator via the challenge response protocol that allows either public key or secret key based authentication to be used.

- A session can be established between the client and server to allow authentication information or network resource (e.g., TCP connection) to be shared among multiple operations. A session key can be established to encrypt the message and to provide integrity checks for messages exchanged within the session.

- 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.

- Service referral may occur among service components.

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

The handle system protocol (version 2.0) specified in this document has

changed significantly from its earlier version. These changes are necessary due to changes made in the handle system data model and the administration model, as well as the service model. Servers that implement the protocol specified in this document may continue to support the earlier version of the protocol by checking the protocol version specified in the Message Envelope (see section 2.2.1).

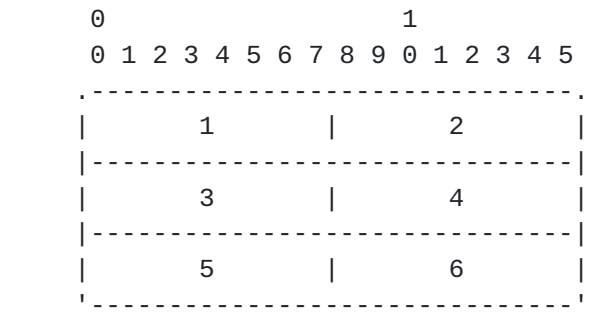
2. Protocol Elements

2.1. Conventions

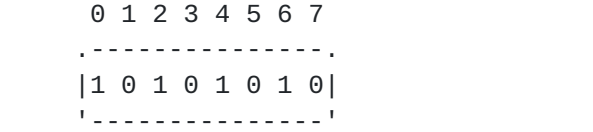
The following conventions are followed by the handle system protocol to ensure interoperability among different implementations.

2.1.1. Data Transmission Order

The order of transmission of data packets follows the network byte order (also called the Big-Endian). That is, when a datagram 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 system protocol is designed so that messages may be transmitted as separate datagrams over UDP, or as a continuous byte stream via a TCP connection. The recommended port number for both UDP

and TCP is 2641.

2.1.2.1. UDP usage

Messages carried by UDP are restricted to 512 bytes (not including the IP or UDP header). Longer messages must be truncated into UDP packets where each packet is assigned with a proper sequence number defined 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 slowdown of the community at large. The recommended retransmission interval is 2-5 seconds.

2.1.2.2. TCP Usage

Messages under the handle system protocol can be mapped directly into a TCP bytestream. However, the size of each message is limited by the range of a 4-byte unsigned integer. Longer messages may be truncated and transmitted as multiple messages and reassembled at the receiving end before further processing.

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 encoded according to UTF-8 encoding. By default, characters are treated as case-sensitive by the handle system protocol. A handle service, however, may be implemented such that ASCII characters are processed case-insensitively. For example, the Global Handle Registry is implemented so that 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 the Handle System, their original case

should be preserved. To minimize user confusion, handle servers should be prevented from creating any handles whose character string matches any existing handle when treated case insensitively. For example, if handle "X/Y" is already created in the database, the server should refuse any request to create the handle "x/y" or any of its case variations.

2.1.4. String type: UTF8-String

Throughout this document, the UTF8-String stands for the data type that consists of a 4-byte unsigned integer followed by an UTF-8 encoded character string. The leading integer specifies the number of octets of the character string. Character strings are exchanged as UTF8-Strings under the handle system protocol.

2.2. Common Elements

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

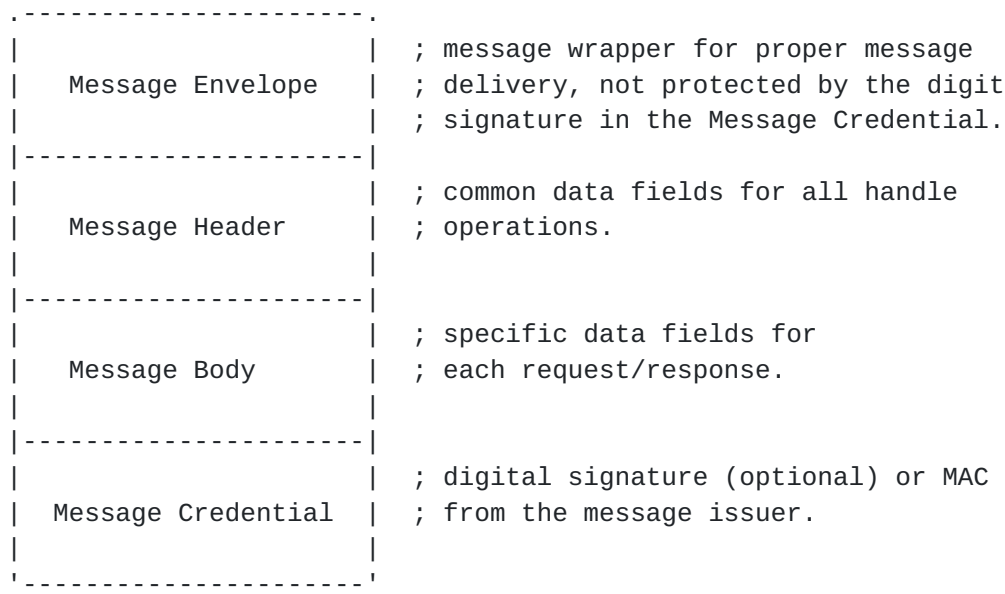


Fig 2.2.1 Top-level format of messages exchanged under the handle system protocol

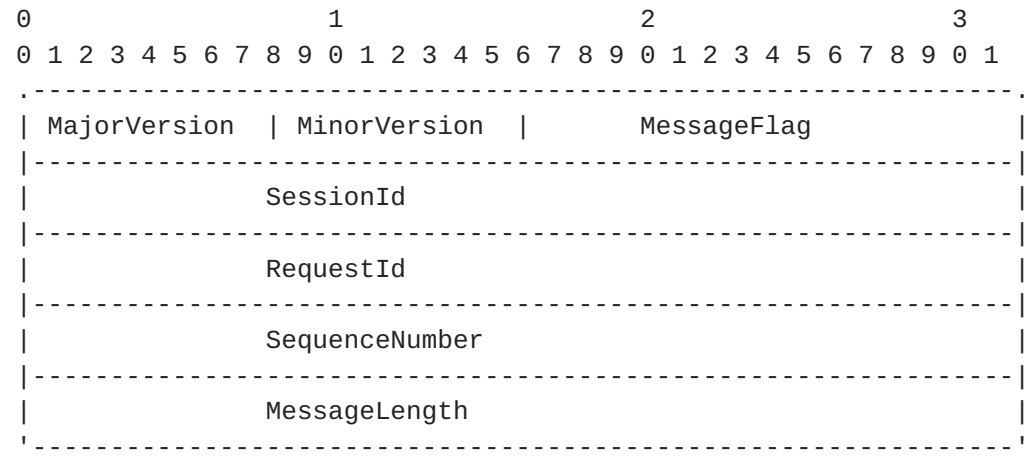
The Message Envelope must always be present. It has a fixed size of 20 octets. Contents in the Message Envelope do not carry any application level information and is primarily used to help deliver the message. Content in the Message Envelope is not protected by the digital signature in the Message Credential.

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 the protocol operations, including the operation code, the response code, and the control options for each operation. Contents of the Message Header are protected by the digital signature in the Message Credential.
The Message Body contains data specific to each operation, and its format varies according to the operation code and the response code in the Message Header. The Message Body may be empty and is protected by the digital signature in the Message Credential.

The Message Credential is used to provide transport security for messages exchanged between the client and server. A non-empty Message Credential may contain the digital signature from the message issuer, or the one-way Message Authentication Code (MAC) generated from a pre-established secret key. The Message Credential can be used to authenticate the message between the host computers and to check the integrity of the message over its transmission. It does not guarantee the trustworthiness of the data (i.e., the handle value) in the Message Body. As stated in the "Handle System Namespace and Service Definition" [2], the trustworthiness of any handle value should ultimately be determined by examining the owner of the handle, as well as any <Reference>'s associated with the handle value.

2.2.1. Message Envelope

The Message Envelope must appear at the top of each message or each truncated portion of the message during its transmission. It has a fixed size of 20 octets and is divided into seven fields as follows:



2.2.1.1. <MajorVersion> and <MinorVersion>

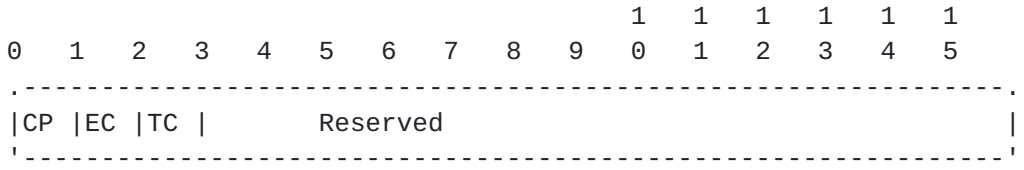
The <MajorVersion> and <MinorVersion> are used to identify the version of the handle system protocol. Each of them is defined as a one-byte unsigned integer. This specification defines the protocol version whose

<MajorVersion> is 2 and <MinorVersion> is 1.

<MajorVersion> and <MinorVersion> are designed to allow future backward compatibility. A difference in <MajorVersion> indicates major variation between 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 changing the general message parsing.

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 (minus the Message Envelope section) is compressed. If the CP bit is set, the message is compressed. Otherwise, the message is not compressed. The handle system protocol uses the same compression method as used by FTP, as specified in RFC468 [8].

Bit 1 is the EC (EnCrypted) flag which indicates whether the message (minus the Message Envelope section) is encrypted. The EC bit could only be set under an established session where a session key is in place. If the EC bit is set, the message is encrypted using the current session key. Otherwise the message is not encrypted.

Bit 2 is the TC (TrunCated) flag that indicates whether this is a truncated block of a certain message. Message truncation happens mostly when transmitting large messages over a UDP channel. See section 2.3 for details about its usage.

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 the communication session between the client and server.

Session and its <SessionId> are assigned by the server either upon a client's request, or when multiple message transactions are expected to fulfill a client's request. For example, the server will assign a unique <SessionId> in the response message if it has to authenticate the client. A client may also explicitly ask to establish a session with the server to setup 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 each new session a non-

zero unique <SessionId> among existing sessions. Servers are also responsible for closing sessions that are not in use for some period of time.

Both clients and servers must reply with the same <SessionId> in response to messages that have a non-zero <SessionId>. A message whose <SessionId> is zero indicates that no session is yet established.

The session and its state information may be shared among multiple handle operations and/or TCP connections between the client and server. 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, any session key that was established for message encryption or authentication, etc. See section 3.8 for details of sessions and their establishment.

2.2.1.4. <RequestId>

Requests from clients are identified by their <RequestId>, a 4-byte unsigned integer assigned by the client. Each request must have a unique <RequestId> among all outstanding requests from the same client. Any response from the server must maintain the same <RequestId> as the original request.

2.2.1.5. <SequenceNumber>

The <SequenceNumber> is a 4-byte unsigned integer used as a counter to keep track of truncated portions of any message transmitted over the UDP channel. The <SequenceNumber> must start with 0 for each message. Truncated messages are identified by the TC flag in the Message Envelope. Messages that are not truncated must set their <SequenceNumber> to zero. Recipients can reassemble a message based on the <RequestId> and the <SequenceNumber> from each truncated portion.

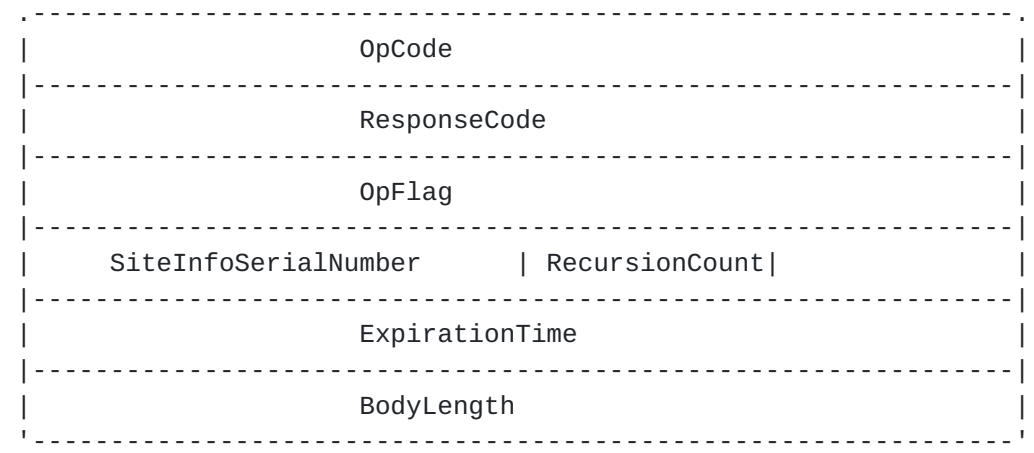
2.2.1.6. <MessageLen>

A 4-byte unsigned integer that specifies the total number of octets of the entire message, excluding the Message Envelope. The length of any single message exchanged over the handle system 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 elements of any handle system protocol operation. It has a fixed size of 24 octets and is divided into eight fields as follows.

```
  0                   1                   2                   3
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
```



Message Header must be present in any message exchanged between the client and server. If a message has to be truncated for its transmission, the Message Header must only appear in the first truncated portion of the message. This is different from the Message Envelope, which must appear 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.

Operation 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
301	OC_SESSION_SETUP	Session setup request
302	OC_SESSION_TERMINATE	Session termination request

A detailed description of each of these <OpCode>s can be found in section 3 of this document. In general, the client assigns the <OpCode> in its request message to the server. Response from the server must maintain the same <OpCode> as the original request and use the <ResponseCode> below to reveal the result.

2.2.2.2. <ResponseCode>

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

R. 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
400	RC_NOT_AUTHORIZED	Not authorized or permitted
401	RC_ACCESS_DENIED	No access to data
402	RC_AUTHEN_NEEDED	Authentication required
403	RC_AUTHEN_FAILED	Failed to authenticate the client
404	RC_INVALID_CREDENTIAL	Invalid credential
405	RC_AUTHEN_TIMEOUT	Authentication timed out
406	RC_UNABLE_TO_AUTHEN	Unable to perform authentication
500	RC_SESSION_TIMEOUT	Session expired
501	RC_SESSION_FAILED	Unable to establish a session
502	RC_NO_SESSION_KEY	No session yet available
503	RC_SESSION_NO_SUPPORT	Session not supported by the server

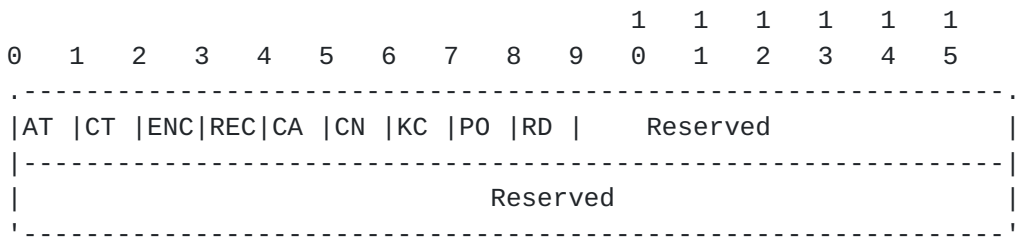
900	RC_TRYING	Request under processing
901	RC_FORWARDED	Request forwarded to another server
902	RC_QUEUED	Request queued for later processing

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, requests from clients must have their <ResponseCode>s set to 0. Servers look at the <OpCode> from the request to determine the kind of operation requested by the client, and must reply with a non-zero <ResponseCode> to indicate the result. For example, a response message 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 defined as a 32-bit bit-mask that defines various control options for the message. The following figure shows the location of each option bit in the <OpFlag> field.



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

CT - CerTified bit. A request with the CT bit set indicates that the server must return the message with the digital signature signed using the server's private key. The server's public key is stored in the service information (i.e., the HS_SITE value) that is used by the client to contact the server. The response message with the CT bit set indicates that the message is signed.

ENC - ENCryption bit. A request with the ENC bit set requires the server to encrypt any response using the pre-established session key before sending it back to the client.

REC - RECursive bit. A request with the REC bit set asks the server to perform the query on behalf of the client if the request is to be processed by another handle server. The server may honor the request by sending the request to the responsible server, obtain the result from that server, and send the result back to the client. The server also has the option to 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 asks the caching server (if any) to perform the authentication of any server response (e.g. verifying the server's signature) on behalf of the client. A response with the CA bit set indicates that the response has been authenticated by the caching server before reaching the client.

CN - ContiNuous bit. A message with the CN bit set instructs the message recipient to expect more messages regarding the current request/response (based on the <RequestId> in the Message Envelope). This happens when any request (or response) has data that is too large to fit within any single message and has to be transmitted as multiple messages.

KC - Keep Connection bit. A message with the KC bit set requires the message recipient to keep the TCP connection open (even 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 and must be kept the same between request and response. A query request with the PO bit set indicates that the client is only asking for handle values that have PUB_READ permissions. Otherwise, the server must return any values that have either PUB_READ or ADMIN_READ permission and will have to authenticate the client as the handle administrator.

RD - Request-Digest bit. A request with the RD bit set asks the server to return the message digest of the request along with the server response. A response message with the RD bit set indicates that the first field in the Message Body contains the message digest of the original request. Clients can check the integrity of the server response by matching the message digest against the original request.

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 the requested options 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 that specifies

the <SerialNumber> of the HS_SITE value used by the client (to access the server). Servers look at the <SiteInfoSerialNumber> in the request to check whether the client has the up-to-date service information.

When possible, the server should fulfill a client's request even if the client has the out-of-date service information. The response message from the server should specify the current 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, or send a query (with <OpCode> set to OC_GET_SITEINFO) directly to the server. If the server can not 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 before the message reaches the recipient. The client must always set the <RecursionCount> to 0 in the initial request. If the server has to send a recursive request on behalf of the client, it must increment the <RecursionCount> by 1. Otherwise, the server must send the response with the same <RecursionCount>. The server should be configured to refuse any service request with <RecursionCount> up to a certain value to prevent possible loop during the recursion.

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 section always follows the Message Header section. The number of octets in the Message Body is specified by the <BodyLength> in the Message Header. The Message Body may be empty. The exact format of the Message Body is defined according to the <OpCode> and <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 <OpFlag> in the Message Header has the RD bit set, the Message Body must include the message digest of the original request. The message digest must be the first field in the Message Body and be encoded as follows:

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

where

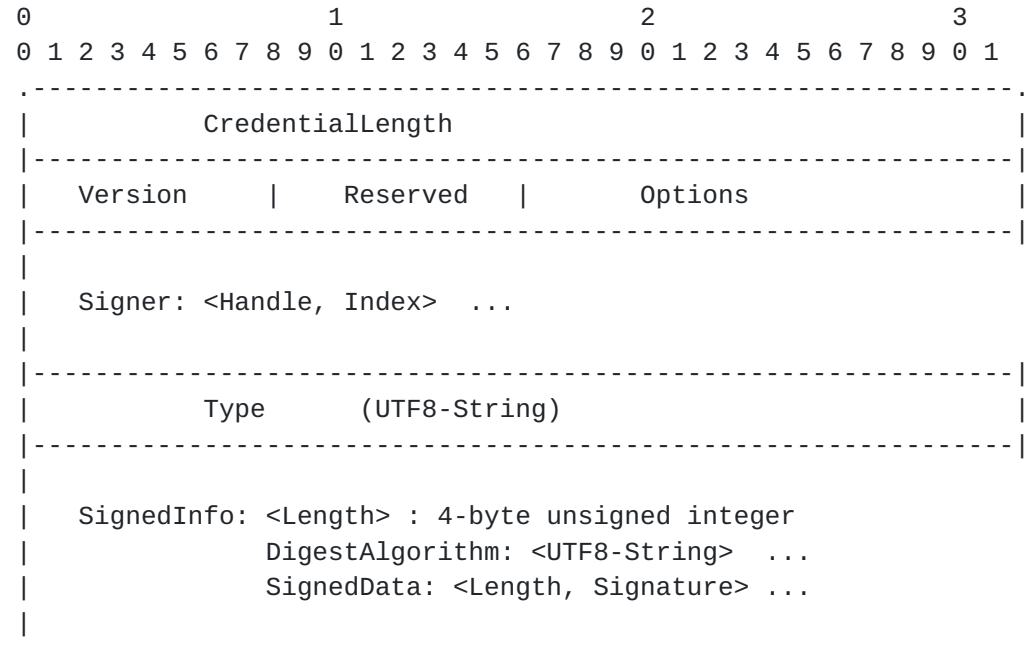
<DigestAlgorithmIdentifier>
An octet that identifies the algorithm used to generate the message digest. If the octet is set to 1, MD5 [9] algorithm is used to generate the message digest. If the octet is set to 2, SHA-1 [10] algorithm is used.

<MessageDigest>
Message digest 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.

If a message has to be truncated for its transmission, the Message Body may be truncated into several portions during the delivery. The message recipient will have to reassemble the Message Body from each portion before further processing.

2.2.4. Message Credential

The Message Credential is primarily used to carry any digital signature signed by the message issuer. The signature is used to protect contents in the Message Header and the Message Body from being tampered during transmission. The Message Credential is defined to be semantically compatible with PKCS#7 [5], and 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 server does not provide any credential.

<Version>
An octet that gives the syntax version number, for compatibility with future revisions. It shall be 0 for this version of the standard.

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

<Options>
Two octets reserved for various cryptography options.

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

where

<HANDLE>
A UTF8-String (i.e. a 4-byte unsigned integer followed by a UTF-8 encoded character string) that refers to the handle that holds the public or secret key used to generate the message credential.

<INDEX>
A 4-byte unsigned integer that identifies the handle value that holds the key.

<Type>
A UTF8-String that specifies content type in the <SignedInfo> field. This document defines six content types: HS_Signed for public key signed data, HS_SignedAndEnveloped for signed and enveloped data, HS_Digest for message digest, HS_Encrypted for encrypted data (using either public or secret key), and HS_MAC for message authentication code generated from a secret key.

```
<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 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 of the message. The syntax and semantics of the signature is determined by the public or secret key referenced in the <Signer> field. For example, if the key referred to by the <Signer> field is a DSA [6] public key, the signature will be the X.509 [7] (using ASN.1 encoding) representation of the parameter R and S used by DSA.

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

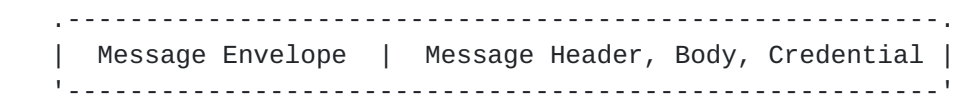
When the server's public key is used to generate the Message Credential, the <HANDLE> must be set to a zero-length UTF8-String and the <INDEX> must be set to 0. This tells the client to authenticate the server's message using the public key specified in the service information.

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

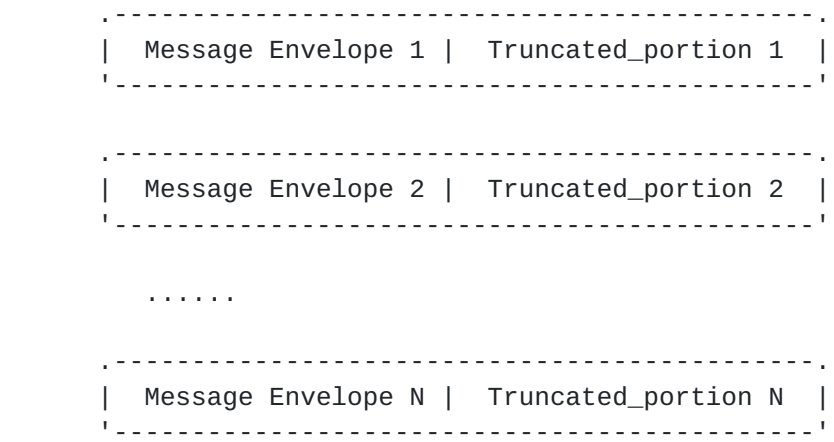
2.3. Message Transmission

A large message may have to be truncated into multiple portions during its transmission. For example, to fit the UDP packet size limit, the message issuer must truncate any large message into multiple UDP packets. Message truncation is performed upon the entire message except the Message Envelope. A Message Envelope has to be inserted in front of

each truncated portion before its transmission. A large message that consists of



can be truncated into:



where the "Truncated_portion 1", "Truncated_portion 2", ..., and "Truncated_portion N" are the result of truncating the Message Header, the Message Body and the Message Credential only. Each "Message Envelope i" (inserted before each truncated portion) must set its TC flag to 1 and must keep the proper sequence count (in the <SequenceNumber>). Its <MessageLength> must also reflect the size of the truncated_portion. The recipient of the truncated message can reassemble the message by concatenating each "Truncated Portion i" together according to the <SequenceNumber> in the "Message Envelope i".

3. Protocol Operations

This section describes handle system operations in terms of messages exchanged between the client and server. It also defines the format of the Message Body according to the <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 is used as the starting point for clients to locate the responsible handle service component and to resolve their handles. The service information for the Global Handle Registry is maintained as HS_SITE values assigned to the handle "0.NA/0.NA", also called the root handle. The service information may come with the client software, or be downloaded from the Handle

System web site at http://www.handle.net.

Changes to any service site of the Global Handle Registry, hence its service information, are identified by the <SerialNumber> within the HS_SITE value. Clients are informed of the service information change when the <ResponseCode> from the server is set to RC_EXPIRED_SITE_INFO. Clients can query for the current service information of the Global Handle Registry from the root handle. The Global Handle Registry must sign the service information using the public key associated with the out-of-date service information (identified in the client's request) so that the client may verify the signature.

3.1.2. Locating the handle system service component

Each handle under the Handle System is managed by a unique handle system service component. For any given handle, the responsible service component (and its service information) can be found from the handle's naming authority handle, managed by the Global Handle Registry.

For example, to find the responsible service component for handle "1000/abc", a client can send a query to the Global Handle Registry for any HS_SITE or HS_SERV values assigned to the naming authority handle "0.NA/1000". The set of HS_SITE values assigned to the naming authority handle is the service information of the service component that manages every handle under the naming authority "1000". If no HS_SITE values are returned, clients can check the existence of any HS_SERV value. The HS_SERV value holds the name of the designated handle, called the service handle (usually under the naming authority "0.SERV"), assigned to the service component. The service handle must hold all the HS_SITE values of the service component.

>From the service information, clients may select the proper service site and figure out the responsible handle server within the site. The procedure to locate the responsible handle server from any service information is described in the following section.

3.1.3. Selecting the responsible server

Each handle system service component is defined in terms of a set of HS_SITE values. Each HS_SITE value defines a service site that consists of a group of handle servers. For any given handle, the responsible handle server within the service component can be found according to the following procedure:

1. Select a service site from the service information.

   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> field) to help the selection. Clients must select the primary service site for any administrative operations.

2. Select the responsible server within the service site.

   Apply MD5 hash function to the handle (or a portion of the handle string as specified in the HS_SITE value) after converting all the ASCII characters with the handle string to the upper case. Take the last 4 bytes as an unsigned integer. Modulo the integer against the number of servers (i.e., the <NumOfServer> field) specified in the HS_SITE value. The result is a non-negative integer that identifies the server record by its location in the server list within the HS_SITE value (see "Handle System Namespace and Service Definition" [2] for the definition of HS_SITE value record). The server record provides all the necessary information for the client to communicate with the server.

3.2. Query Operation

A query operation consists of client sending a query to the responsible handle server and the server returning the query result back to the client. A client may query for the entire value set assigned to the handle, or a subset of it based on the value index or value type.

3.2.1. Query Request

The Message Header of any query request must have its <OpCode> set to OC_RESOLUTION (i.e., unsigned integer 1) and <ResponseCode> to 0.

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

```
<Query Request MessageBody>  ::=  <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 following array. Each number in the integer array specifies the index of a handle value requested by the client. The client sets the first integer to zero (followed by an empty array) to query all handle values regardless of their indexes.

<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. The list may be empty if the first integer is 0, in which case the server must return all handle values regardless of their data type.

If the query request does not specify any index or data type, the server should return all handle values that have the PUBLIC_READ permission if the PO flag (in the Message Header) is set. Clients can also send out queries without the PO flag set to ask for handle values with or without PUBLIC_READ or ADMIN_READ permission. If the query requests a specific handle value via its index and the value has no PUBLIC_READ permission, the server should honor 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 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 treat the query as a logical OR relationship between the set of handle values specified in the <IndexList> and the set of values specified in the <TypeList>, and should return both sets of handle values.

3.2.2. Successful Query Response

The Message Header of any query response must set its <OpCode> to OC_RESOLUTION. 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. It is the client's responsibility to check whether the <Handle> is the one from the original request.

<ValueList>
A 4-byte unsigned integer followed by a list of handle values concatenated one after another. The integer specifies the number of handle values in the list. The integer is set to zero if there

is no handle value that satisfies the query, in which case the <ResponseCode> (in the message header) must be set to RC_SUCCEED (instead of RC_VALUE_NOT_FOUND, which designates an error condition for handle administration). The encoding of each handle value follows the specification given in section 3.1 of the "Handle System Namespace and Service Definition" [2].

3.2.3. Unsuccessful Query Response

If the server can not fulfill the client's request, it must return an error message to explain the error. The common format for any error message returned from the server is defined in section 3.3 of this document.

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

Note that servers must NOT return an RC_HANDLE_NOT_FOUND message if they are not responsible for the requested handle. It is possible that the requested handle exists but is managed by some other handle service or handle server. When this happens, the server should either return a service referral (see section 3.4) that directs the client to the responsible handle service (e.g., the Global Handle Registry), or simply return an error message that has its <ResponseCode> set to RC_SERVER_NOT_RESP(onsible).

The server may send a response-message with <ResponseCode> set to RC_SERVER_BUSY if the server is too busy to fulfill client's request. Like RC_HANDLE_NOT_FOUND, a RC_SERVER_BUSY message also has an empty message body.

Servers should return an RC_ACCESS_DENIED message if the request asks for any specific handle value that has neither PUBLIC_READ nor ADMIN_READ permission.

Servers may also ask a client to authenticate himself as the administrator of the requested handle. This happens if any handle value in the query has the ADMIN_READ permission but no PUBLIC_READ permission. In this case, servers should return an RC_AUTHEN_NEEDED message, as described in section 3.5 of this document.

3.3. Error Response from Server

Any error message returned from the server must keep the same <OpCode> (in the message header) as the original query. The <ResponseCode> in the message header is defined in section 2.2.2.2 which unambiguously identifies every error condition.

The Message Body of an error message may be empty or consist of the following default data fields, except in those cases otherwise specified in this document:

```
<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 following list is a 4-byte unsigned integer that refers to a handle value that contributed to the failure. Note that servers are not obliged to return a complete list of handle values that may cause the request to fail.

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 send an error response with <ResponseCode> set to RC_SERVER_NOT_RESP. Service referral typically 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.

A service referral must keep the same <OpCode> as the original request, and set the <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 that is used to manage the referral information. If the <ReferralHandle> is set to "0.NA/0.NA", the server is referring the client to the Global Handle Registry. Otherwise, the client is referred to a local handle service.

<ValueList>
Optional field. 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 values in the list.

Unlike normal query responses, which may consist of any kind of handle values, 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 from the service referral. A client should always select to use its own copy of the service information of the Global Handle Registry whenever 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 privilege, including adding, removing, or modifying handles or handle values.

Client authentication consists of multiple messages exchanged between the client and server. Messages exchanged during the authentication are correlated via a unique <SessionId> assigned by the server.

The authentication starts with a response message from the server that contains a challenge to the client. The client must respond with a challenge-response message. The server validates the challenge-response either by verifying the digital signature within the challenge-response message, or by sending a verification request to the handle server, herein referred to as the verification server, that manages the handle of the administrator (along with its secret key). The purpose of the challenge and the challenge-response is for the client to prove to the server that it possesses the private key (or the secret key) of the handle administrator.

Upon successful authentication of the client as a handle administrator, the server must make sure that the administrator has the proper privilege before fulfilling the client's request. If the administrator has sufficient privilege, the server must continue to fulfill the

client's request and send back the result. If the administrator does not have sufficient privilege, the server must return an error message with <ResponseCode> set to RC_NOT_AUTHORIZED. Requests that failed the authentication will receive an error response with <ResponseCode> set to RC_AUTHEN_FAILED.

The following sections provide the exact definitions of each message exchanged during client authentication.

3.5.1. Challenge from Server to Client

The Message Header of the server 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 process. It must also set the RD bit in the <OpFlags> in the Message Header, regardless whether the original request had the RD bit set or not.

The message body of the 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.

3.5.2. Challenge-Response from Client to Server

The message header of the challenge-response must set its <OpCode> to OC_CHALLENGE_RESPONSE. 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 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" when

the client selects to use some secret key, or is set to "HS_PUBKEY" if client selects to use the public key for authentication.

<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>. This is the handle value that holds the public or secret key of the administrator.

<ChallengeResponse>
An octet followed by either the MAC (Message Authentication Code) 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 message digest algorithm used to generate the MAC, as defined in section 2.2.3. The MAC is calculated by applying a one-way hash function over the server's challenge and the secret key of the handle administrator. In other words, the MAC is the hash result over the concatenation of the secret key (available from <KeyHandle> & <KeyIndex>), the server's challenge, and the secret key again. For example, if the first octet is set to 1, the MAC is generated by

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

where the <SecretKey> is the administrator's secret key referenced by <KeyHandle> and <KeyIndex>. The <ServerChallenge> consists of the Message Header and the Message Body portion of the server's challenge.

If the <AuthenticationType> is "HS_PUBKEY", the <ChallengeResponse> contains the digital signature over the message body of the server's challenge. The signature is encoded the same way as the <SignedInfo> field of the Message Credential, specified in section 2.2.4. It consists of a 4-byte unsigned integer, followed by the UTF8-String that specifies the digest algorithm used for the signature, followed by the signature over the server's challenge. The <KeyHandle> and <KeyIndex> must refer to the 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 specifies the set of privileges granted to the corresponding administrator, as well as the reference to the authentication key that is used to authenticate the administrator. An HS_ADMIN value may make direct reference to the authentication key

via the <AdminRef> field (see definition of HS_ADMIN record in the "Handle System Namespace and Service Definition" [2]). It may also define its authentication key indirectly via an administrator group, defined in terms of another handle value of type HS_VLIST. The HS_VLIST value makes reference to a list of authentication keys, each of which belongs to a member of the administrator group.

For handles with multiple HS_ADMIN values, the server will have to retrieve all the HS_ADMIN values that have sufficient privilege and check if the <KeyHandle> and <KeyIndex> match the <AdminRef> of any of the HS_ADMIN values. If no matches are found but there are HS_ADMIN values whose <AdminRef> refers to an administrator group, the server must check if the <KeyHandle> and <KeyIndex> belongs to any of the administrator groups. An administrator group may contain another administrator group as a member. Servers must be careful to avoid infinite loops when navigating the administrator groups.

If the <KeyHandle> and <KeyIndex> do not match any of the HS_ADMIN values, or members of the handle administrator group, the server must return an error message with <ResponseCode> set to RC_NOT_AUTHORIZED. If the <KeyHandle> and <KeyIndex> do match any of the HS_ADMIN values, or belong to any of the handle administrator group, the server must verify the challenge-response as follows:

If the <AuthenticationType> is "HS_PUBKEY", the server can obtain the administrator's public key according to the <KeyHandle> and <KeyIndex>. The public key is then used to verify the <ChallengeResponse> against the server's <Challenge>. If the <Challege> and the <ChallengeResponse> match, the server must fulfill the original request and send back the result. Otherwise, the server must 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 manages the handle referred to by the <KeyHandle>. The verification request and its response are defined in the following sections.

3.5.3. Challenge-Response Verification-Request

If the <AuthenticationType> field in the Challenge-Response contains "HS_SECKEY", secret key authentication is used to authenticate the client. In this case, the server will have to issue a Verification-Request to the verification server that manages the <KeyHandle> (in the Challenge-Response), hence the secret key. The verification server will verify the <ChallengeResponse> against the <Challenge> on behalf of the handle server that issued the <Challenge>.

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 identifies the handle that holds the secret key to verify the <ChallengeResponse> against the <Challenge>.

<KeyIndex>
A 4-byte unsigned integer that is the index of the handle value (of <KeyHandle>) that contains the secret key.

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

<ChallengeResponse>
The same <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, thus to ensure that the verification server will sign the Verification-Response defined 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 the <ResponseCode> to RC_SUCCESS whether the <ChallengeResponse> matches the <Challenge> or not. The <OpFlag> in the Message Header must set its RD bit and the Message Body is defined as follows:

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

where

<RequestDigest>
Contains the message digest of the Challenge-Response Verification-Request message and is encoded according to 2.2.3.

<VerificationResult>
An octet that is set to 1 if the client's challenge-response matches the server's challenge. Otherwise it must be set to 0.

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

3.6. Handle Administration

Clients send handle administration requests to the handle server to add, delete, or modify handles or handle values. Naming authority administration is performed as handle administration on the corresponding naming authority handles. For any handle administration request, clients should expect to receive a challenge from the server for client authentication, as described in section 3.5.

This section defines the message layout of each handle administration request and the possible response from the 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 any ADD_VALUE request must set its <OpCode> to OC_ADD_VALUE.

The message body of any ADD_VALUE request is encoded as follows:

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

where

<Handle>
A UTF8-String that specifies the handle for the new value(s).

<ValueList>
A 4-byte unsigned integer followed by a list of handle values. The integer indicates the number of handle values in the list. Each handle value may have a different size but must be encoded according to the "Handle System Namespace and Service Definition" [2].

The server that receives the ADD_VALUE request must first authenticate the client as the administrator with ADD_VALUE privilege. Upon successful authentication, the server will proceed to add the new handle value(s) to the <Handle> provided all the values in the <ValueList> can be added without causing any error. If successful, the server must notify the client with a RC_SUCCESS message (i.e., a response message with <ResponseCode> set to RC_SUCCESS and an empty Message Body).

The server must carry out the ADD_VALUE request as a transaction such

that, if adding any of the values 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 return a response to the client that explains the error. For example, if any value in an ADD_VALUE request has the same index as any of the existing values, the server must return an error message that has the <ResponseCode> set to RC_VALUE_ALREADY_EXISTS.

ADD_VALUE requests are also used to add handle administrators (defined in terms of HS_ADMIN values). This happens if the <ValueList> in the ADD_VALUE request contains any HS_ADMIN values. The server must authenticate the client as the administrator with 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) is to be removed.

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

The server that receives the REMOVE_VALUE request must first authenticate the client as the administrator with REMOVE _VALUE privilege. Upon successful authentication, the server will proceed to remove the handle values referenced in the <IndexList> from its database. If successful, the server must notify the client with a RC_SUCCESS message.

Servers must carry out each REMOVE_VALUE request as a transaction so that, if removing any values specified in the <IndexList> raises any

error, the entire operation must be rolled back. For any failed REMOVE_VALUE request, none of the values referenced in the <IndexList> should be removed. 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 message. Note that any REMOVE_VALUE requests asking to remove any non-existing handle values should not be treated as an error.

REMOVE_VALUE requests are also used to remove handle administrators (defined in terms of HS_ADMIN values). This happens if any of the indexes in the <IndexList> (in the REMOVE_VALUE request) refer to a HS_ADMIN value. Servers must authenticate the client as the administrator with REMOVE_ADMIN privilege before fulfilling such requests.

3.6.3. Modify Handle Value(s)

Clients can make modifications to any existing handle values 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) is to be modified.

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

Servers that receive any MODIFY_VALUE requests must first authenticate the client as the administrator with MODIFY_VALUE privilege. Upon successful authentication, the server will proceed to replace those handle values whose indexes coincide with values in the <ValueList>, provided all the handle values have PUB_WRITE or ADMIN_WRITE permission. If successful, the server must notify the client with a RC_SUCCESS message.

Servers must carry out each MODIFY_VALUE request as a transaction so that, if replacing any values specified in the <ValueList> raises an error, the entire operation must be rolled back. For any failed

MODIFY_VALUE request, none of values referenced 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 request asks to remove a handle value that has neither PUB_WRITE nor ADMIN_WRITE permission, the server must return an error message that has the <ResponseCode> set to RC_ACCESS_DENIED. Any MODIFY_VALUE request to replace non-existing handle values must also be acknowledged with an error with <ResponseCode> set to RC_VALUE_NOT_FOUND.

MODIFY_VALUE requests are also used to update handle administrators. This happens if any of the values in the <ValueList> (of the MODIFY_VALUE request) refer to a HS_ADMIN value. Servers must authenticate the client as the administrator with MODIFY_ADMIN privilege before fulfilling such request.

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 to be added to the new handle. The integer indicates the number of handle values in the list. The <ValueList> should at least include a HS_ADMIN value that defines the handle administrator.

Only naming authority administrators granted CREATE_HANDLE privilege can create new handles under the naming authority. The server that receives CREATE_HANDLE requests must first authenticate the client as the administrator of the corresponding naming authority handle. The administrator must be granted the CREATE_HANDLE privilege. This is different from an ADD_VALUE request where the server authenticates the client as the administrator of the handle. Upon successful authentication, the server must proceed to create the new handle and assign to it the values in the <ValueList>. If successful, the server must notify the client with a RC_SUCCESS message.

Servers must carry out each CREATE_HANDLE request as a transaction so that, 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 must be acknowledged with an error with <ResponceCode> set to RC_HANDLE_ALREADY_EXIST.

CREATE_HANDLE requests are also used to create new naming authorities. New naming authorities are created as naming authority handles (i.e., handles under the naming authority "0.NA") at the Global Handle Registry. Before creating a new naming authority handle, the server must authenticate the client as the administrator of the parent naming authority handle. Root level naming authorities can only be created by the administrator of the root handle "0.NA/0.NA". Only administrators with the CREATE_NA privilege are allowed to create naming authorities.

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 to be deleted.
```

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

Servers must carry out each DELETE_HANDLE request as a transaction so that if any part of the DELETE_HANDLE process failed, 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 without deleting the handle. This may happen if the handle contains a value that has neither PUB_WRITE nor ADMIN_WRITE permission. DELETE_HANDLE requests that ask to delete a non-existing handle must be acknowledged with an error with <ResponseCode> set to RC_HANDLE_NOT_EXIST.

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

authorities.

3.6.6. List Handle(s) under a Naming Authority

Client may send a LIST_HANDLE request to a handle server to get all its handles that are under client specified naming authority. To obtain a full list of handles under the naming authority, client must send the LIST_HANDLE requests to each handle server under the responsible handle Service, and concatenate the results from each handle server. 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.

The message body of successful LIST_HANDLE response (from each handle server) is defined as follows:

```
<Successful LIST_HANDLE Response>  ::=  <Num_Handles>
                                              <HandleList>
```

where

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

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

LIST_HANDLE may potentially slow down the overall server performance. A handle service has the freedom to choose whether to support LIST_HANDLE operation or not. The server may return a RC_OPERATION_DENIED message when LIST_HANDLE is not supported. Servers that choose to provide LIST_HANDLE service may select to authenticate the client as the naming authority administrator with LIST_HANDLE permission before carrying out the operation.

3.6.7. List Sub-Naming Authorities under a Naming Authority

Client may send a LIST_NA request to a handle server (under the Global Handle Registry, or GHR) to obtain a list of all the sub-naming authorities handlers that are managed by the handle server. A complete

listing of sub-naming authorities can be obtained by sending a LIST_NA request to every handle server under the GHS and concatenate their results.

The message header of the LIST_NA request must set its <OpCode> to OC_LIST_NA. Its message body is defined as follows:

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

where

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

The message body of successful LIST_NA response (from each handle server) is defined as follows:

```
<Successful LIST_HANDLE Response> ::=  <Num_Handles>
                                       <HandleList>
```

where

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

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

LIST_NA are performed by the Global Handle Registry (GHS) which manages all the naming authority handles. It may potentially slow down the overall system performance, especially the GHS. It is recommended that only naming authority administrators are granted with such privilege. Handle servers under the GHS should authenticate the client as the naming authority administrator with LIST_HANDLE permission before carrying out the operation.

3.7. Naming Authority (NA) Administration

The Handle System manages naming authorities as handles under the naming authority "0.NA". The naming authority administration follows the same procedures as those used by handle administration, which are defined in section 3.6.

Naming authority handles are managed by the Global Handle Registry. 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. A new (non-root level) naming authority can only be created by the administrator of its parent naming authority, i.e., the administrator of the parent naming authority handle with NA_CREATE privilege. Root level naming authority handles can only be created by the administrator of the root handle "0.NA/0.NA".

For example, one can create the naming authority "10.1000" by sending a CREATE_HANDLE request to the Global Handle Registry to create the naming authority handle "0.NA/10.1000". The server at the Global Handle Registry will authenticate the client as the administrator of the parent naming authority, that is, the administrator of the parent naming authority handle "0.NA/10". The server also has to make sure that the administrator has CREATE_NA privilege before fulfilling the request.

3.8. Session and Session Management

Sessions are used to allow sharing of authentication information among multiple message-exchanges between client and server. For example, a naming authority administrator may authenticate itself once through the session setup and register multiple handles under the same session.

A client may also ask the server to establish a session key and use it for secured message exchange. A session key is a secret key that is shared by the client and server and can be used to establish an encrypted and/or authenticated session. A session is encrypted if every message exchanged within the session is encrypted using the session key. A session is authenticated when every message is authenticated by the Message Authentication Code (MAC) using the session key.

Session may be established automatically by the server when multiple message exchanges are expected to fulfill the client's request (as in the case of client authentication discussed in section 3.5). A client may also request a session explicitly with an OC_SESSION_SETUP request.

Every session between the client and server is identified by a non-zero Session ID in the Message Header. Servers are responsible for generating a unique Session ID for each outstanding session. Session IDs under 1024 are reserved for system use.

For each established session, the server must maintain the list of session attributes provided by the client, as well as any options specified in the Session Setup Request. The list of possible session attributes and options are specified in the next section.

A session may be terminated by the client via an OC_SESSION_TERMINATE request. When this happens, all pending data for the session has to be discarded. Servers are also responsible for terminating any session that has been idle for any significant amount of time.

3.8.1. Session Setup Request

Clients may setup a session with any handle server with a Session Setup Request. The Message Header of the Session Setup Request must have its <OpCode> set to OC_SESSION_SETUP and <ResponseCode> to 0. If the <SessionId> in the Message Envelope is zero, a new session will be established according to the request. If the <SessionId> in the Message Envelope is non-zero, the request will be applied to the existing session identified by the <SessionId>.

The Message Body of any Session Setup Request is defined as follows:

```
  <Session Setup Request MessageBody> ::= <SessionAttribute>
                                          <SessionOption>
```

where

<SessionAttribute>
A 4-byte unsigned integer followed by a list of session attributes. The leading integer indicates the number of attributes in the Session Setup Request. It is followed by any combination of <HS_SESSION_IDENTITY>, <HS_SESSION_KEY>, <HS_SESSION_TIMEOUT>, each of which is defined as follows:

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

where

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

<Handle>
<ValueIndex>
An UTF-8 String followed by a 4-byte unsigned integer that specify 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> ::= <Key>
                     <Handle>
                     <ValueIndex>
```

where

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

<Handle>
<ValueIndex>
An UTF-8 String and a 4-byte unsigned integer that specify the handle and the handle value for session key exchange. It must refer to a handle value that contains the public key of the client. The public key

is used by the server to encrypt the session key (generated by the server) before sending it to the client.

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

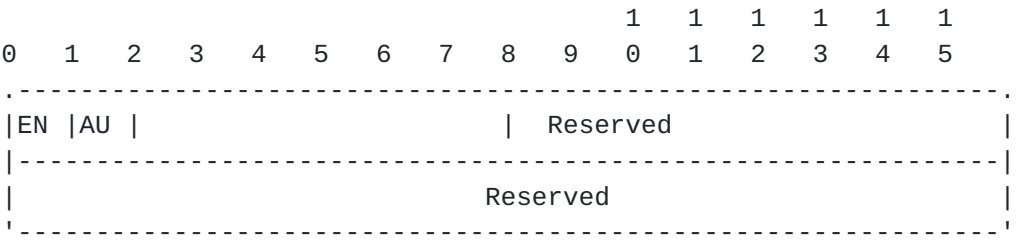
where

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

<TimeOut>
A 4-byte unsigned integer that specifies the desired duration of the session (since its establishment) in terms of number of seconds.

Note that it should be treated as an error if the same 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.

<SessionOption>
A 32-bit bit-mask that defines various control options for messages exchanged within the session. The following figure shows the location of each option bit in the <SessionOption> field.



where

EN - ENcryption bit, indicating that every message exchanged within the session must be encrypted using the session key.

AU - AUthentication bit, indicating that every message exchanged within the session must include the Message Authentication Code (MAC) in its Message Credential.

A Session Setup Request can be used to change session attributes or options of any established session. Care must be taken by the server to prevent any unauthorized request. For example, an encrypted session may only be changed into an unencrypted session by a Session Setup Request that contains the MAC (generated from the session key) in the Message Credential.

3.8.2. Session Setup Response

The message header of the Session Setup Response must set its <OpCode> to OC_SESSION_SETUP. Successful Session Setup Response must set its <ResponseCode> to RC_SUCCESS.

The message body of the successful Session Setup Response may be empty, or contain the encrypted session key as requested by the client:

```
<Message Body of Session Setup Response> ::= [ <RequestDigest> ]
                                             [ <EncryptedSessionKey> ]
```

where

<RequestDigest>
Optional field as defined in section 2.2.3.

<EncryptedSessionKey>
Session key encrypted using the public key specified in the <HS_SESSION_KEY> attribute from the Session Setup Request. The session key is a randomly generated octet string from the server. The server will only return the <EncryptedSessionKey> if <HS_SESSION_KEY> is specified in the Session Setup Request.

3.8.3. Session Termination

Clients can terminate a session with a Session Termination Request? The Message Header of a Session Termination Request must have its <OpCode> set to OC_SESSION_TERMINATE and its <ResponseCode> to 0. The Message Envelope of the Session Termination Request must have its <SessionId> set to the Session ID of the session to be terminated.

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 include the server's network interface(s) (e.g., IP address, port number, etc.), the number of concurrent processes/threads allowed, time-out interval for any TCP connection or authentication process, re-try policy under UDP connection, whether to support recursive service, case-sensitivity for ASCII characters, different levels of transaction tracking, etc.

All handle server implementations must be capable of recognizing all handle data types as defined in the "Handle System Namespace and Service Definition" [2] and be able to retrieve/store any kind of handle values.

A handle server must support multiple concurrent activities, whether they are implemented as separate processes/threads in the host's OS, 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 for refreshing or 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 new data types.

Clients that follow service referrals or handle aliases must avoid falling into an infinite loop. They must not repeatedly contact the same server for the same request with the same target entry name. A client may choose to apply 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 honored by the client.

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

5. Security Considerations

The overall handle system security considerations are discussed in "Handle System Overview" [1] and that discussion applies equally to this document.

6. Author's Address

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

Phone: 703-262-5331
Email: sreilly@cnri.reston.va.us

Larry Lannom
Corporation for National Research Initiatives (CNRI)
1895 Preston White Dr. Suite 100
Reston, VA 20191 USA
Phone: 703-620-8990
Email: llannom@cnri.reston.va.us

Jing Shi
Corporation for National Research Initiatives (CNRI)
1895 Preston White Dr. Suite 100
Reston, VA 20191 USA
Phone: 703-620-8990
Email: jshi@cnri.reston.va.us

7. References

[1] S. Sun, L. Lannom, S. Reilly, "Handle System Overview", IETF draft, http://www.ietf.org/internet-drafts/draft-sun-handle-system-06.txt, work in progress.
[2] S. Sun, S. Reilly, L. Lannom, "Handle System Namespace and Service Definition", IETF draft, http://www.ietf.org/internet-drafts/draft-sun-handle-system-def-04.txt, work in progress.
[3] F. Yergeau, "UTF-8, A Transform Format for Unicode and ISO10646", RFC2044, October 1996. http://www.ietf.org/rfc/rfc2044.txt
[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] ITU/ISO Recommendation X.509 - Information technology - Open System Interconnection - The directory: Autherntication framework. Nov. 1993
[8] R. Braden, "FTP DATA COMPRESSION", RFC468, March 8, 1973, ftp://ftp.isi.edu/in-notes/rfc468.txt
[9] R. Rivest, "The MD5 Message-Digest Algorithm", RFC1321, April 1992, ftp://ftp.isi.edu/in-notes/rfc1321.txt
[10] NIST, FIPS PUB 180-1: Secure Hash Standard, April 1995. http://csrc.nist.gov/fips/fip180-1.txt (ascii)