DIGITAL OBJECT IDENTIFIER RESOLUTION PROTOCOL SPECIFICATION

VERSION 3.0
JUNE 30, 2022

RELEASED BY
DONA FOUNDATION

ACKNOWLEDGEMENTS

The DONA Foundation would like to thank the following individuals for their many contributions and insights in the preparation of this DO-IRP Specification:

Robert E. Kahn, Christophe Blanchi, Robert Tupelo-Schneck, Laurence Lannom, Patrice A. Lyons, Giridhar Manepalli, and Sam Sun.

COPYRIGHT LICENSE AGREEMENT

1. This License Agreement (**Agreement**) is between the DONA Foundation (**DONA**), a registered Swiss non-profit entity, and the Individual or Organization (**User**) that has accessed, downloaded or implemented this **Digital Object Identifier Resolution Protocol Specification (Version 3)** (hereinafter called **DO-IRP(v3)**) available to the public free of charge subject to the terms and conditions in the Agreement.

2. While there are no patent claims asserted by DONA in the Digital Object Architecture itself, or in DO-IRP(v3), there may be third party rights or interests in the external specifications referenced in DO-IRP(v3); and no licenses under such third party external specifications are granted to User in this Agreement.

3. DONA hereby grants User a non-exclusive, fully paid-up, world-wide license to reproduce, implement and further disseminate DO-IRP(v3) to the public, provided that the DONA Foundation copyright notice and this Agreement are both retained in DO-IRP(v3).

4. User hereby acknowledges that DONA is making DO-IRP(v3) available to the public on an "AS IS" basis and DONA MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, DONA MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF FITNESS FOR ANY PARTICULAR PURPOSE, OR THAT THE USE OF DO-IRP(v3) WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. The Agreement will automatically terminate upon a material breach of its terms and conditions. Neither the names of the persons acknowledged as contributing to the preparation of DO-IRP(v3) nor the mark DONA may be used in a trademark sense to endorse or promote products or services of User, or any third party.

6. The Agreement shall be construed and enforced in accordance with the laws of Switzerland.

DONA

CONTENTS

DIGITAL OBJECT IDENTIFIER RESOLUTION PROTOCOL SPECIFICATION VERSION 3.0

ABSTRACT

This document specifies the *Digital Object Identifier Resolution Protocol* (DO-IRP), that may be used to enable centralized or de-centralized, secure identifier resolution and administration in a computational environment such as the Internet. Resolution means a retrieval of a structured record associated with an identifier from a DO-IRP service. Administration means creation, modification or deletion of an identifier record as part of the overall management of identifier records.

This specification provides a detailed description of the DO-IRP identifier space; and the data, service, and associated operation models necessary for clients to interact with DO-IRP servers to perform such resolution and administration functions. The identifier space definition specifies the identifier syntax and its semantic structure. The data model defines the data structures used by the protocol and any pre-defined data types for carrying out the DO-IRP service. The service model provides definitions of various DO-IRP components and explains how they work together within a computer network or other computational environment. The DO-IRP operation model defines the messages transmitted between client and server both for authentication purposes as well as for carrying out the identifier resolution and administration.

1 INTRODUCTION

DO-IRP is a service protocol for securely creating, updating, maintaining, and accessing a distributed set of identifier records associated with identifiers allotted to digital objects. It is based on the Digital Object (DO) Architecture[1] and is designed to provide access to protected identifier information for enabling interoperability and for sharing resources over networks such as the Internet. A companion protocol, called the Digital Object Interface Protocol (DOIP)[2], enables digital objects to interact with each other based primarily on the exchange of identifiers. The DOIP

[1] For background information on the DO Architecture, including early work by CNRI on the technology and its implementation, see "A Framework for Distributed Digital Object Services" [14]. Available on the Internet at https://www.doi.org/topics/2006_05_02_Kahn_Framework.pdf

[2] The DOIP specification [15] is available on the Internet at https://www.dona.net/sites/default/files/2018-11/DOIPv2Spec_1.pdf

specification contains a brief description of the DO Architecture along with references to relevant work.

An early implementation of the identifier/resolution component of the DO Architecture was documented by Corporation for National Research Initiatives (CNRI) in IETF RFCs 3650 [1], 3651 [2] and 3652 [3]. An important element of this early work by CNRI was the description of the identifier /resolution protocol, known as *Handle System Protocol (ver 2.1) Specification* in RFC 3652. While the DO-IRP draws on the Handle System[3] Protocol for basic technical matters, and is largely backwards compatible with it, DO-IRP is a neutral specification that is not tied to any specific implementation. This DO-IRP specification replaces the three RFCs mentioned above.

1.1 DIGITAL OBJECT CONCEPT

The concept of a "digital object" is intrinsic to the DO Architecture.[4] A DO is a sequence of bits, or a set of sequences of bits, incorporating a work or portion of a work, or other information in which a party has rights or interests, or in which there is value, each of the sequences being structured in a way that is interpretable by one or more computational facilities. Each DO has, as an essential element, an associated unique persistent identifier, known as a digital object identifier, or informally as a handle or simply an identifier. Resolution of identifiers is provided in a distributed fashion by individual organizations that provide identifier services. Each identifier service has an "administrator" that is responsible for that particular service.

Applications of the DO-IRP could include the provision of metadata services for information represented in digital form, identity management services for individuals, organizations and Internet resources, and more generally applications that require resolution and/or administration of unique persistent identifiers.

1.2 HIGH GRANULARITY OF ADMINISTRATION

The service protocol has been designed for rapid resolution of identifiers and to support administration of their associated identifier records. Unlike most conventional systems (even distributed systems) that are designed to have a relatively small number of broadly empowered administrators, DO-IRP allows extremely fine granularity of administrative control. It makes use of a self-contained administrative framework, where each administrator is uniquely identified and has

[3] Since the publication of the three RFCs, the trademark "Handle System" was transferred by CNRI to the DONA Foundation, located in Geneva, Switzerland.

[4] For all practical purposes, the concept of a digital object is substantially similar to the notion of "digital entity" as defined in ITU-T Recommendation X.1255 that is based largely on the DO Architecture.

an associated identifier record containing a public or secret key to perform DO-IRP authentication, that de-couples each identifier from its administration and, instead, allows access control to be defined for and applied to each identifier record.

1.3 DO-IRP HIGH RESOLUTION PERFORMANCE

To assure high resolution performance, a service site may consist of a single server or a set of servers, usually residing at a given Internet location. These computers work together to distribute the data storage and processing load at the site. It is possible, although not recommended, to compose a site from servers at widely different locations.

Each DO-IRP service may be replicated into multiple service sites (see section 4.3.2 below). Each service site may consist of multiple computers. Service requests targeted at any DO-IRP service can be distributed into different service sites, and into different servers within any service site. Such architecture assures that each DO-IRP service could have the capacity to manage very large numbers of identifiers and requests. It also provides ways for each DO-IRP service to avoid any single point of failure.

Each DO-IRP service may provide the same set of functions for resolving and administering its collection of identifiers. DO-IRP services differ primarily in that each service is responsible for a distinct set of identifiers. They are also likely to differ in the selection, number, and configuration of their components such as the servers used to provide resolution and administration. Different services may be created and managed by different organizations. Each of them may have their own goals and policies.

2 IDENTIFIER

2.1 IDENTIFIER SYNTAX AND ENCODING

Every identifier, as used in DO-IRP, consists of two parts: its *prefix* and a unique *suffix* under the prefix:

The prefix and suffix are separated by the ASCII character "/". The collection of suffixes under a prefix defines the *local identifier space* for that prefix. Each suffix must be unique under its local identifier space. The combination of a unique prefix and suffix under that prefix ensures that any identifier is unique within the context of the DO-IRP.

Prefixes are defined in a hierarchical fashion resembling a tree structure. Each node and leaf of the tree is given a label that corresponds to a prefix segment. DO-IRP identifier prefixes are constructed left to right, concatenating the labels from the root of the tree to the node that represents the

prefix. Each label is separated by the octet used for ASCII character "." (0x2E). The dot "." is known as a delimiter.

The identifier syntax definition is described in ABNF notation below:

```
<Identifier>        = <Prefix> "/" <Suffix>
<Prefix>            = *(<Prefix> ".") <PrefixSegment>

<PrefixSegment>     = 1*(%x00-2D / %x30-FF)
                      ; any octets that map to UTF-8 encoded
                      ; Unicode characters except
                      ; octets '0x2E' and '0x2F' (which
                      ; correspond to the ASCII characters '.',
                      ; and '/').

<Suffix>            = *(%x00-FF)
                      ; any octets that map to UTF-8 encoded
                      ; Unicode characters
```

Three examples of identifiers are 35.1234/abc, 35.1234/HQ, and 0.NA/35.1234 – these example identifiers are hypothetical and are used in this documentation for illustration only.

Identifiers may consist of any printable characters from Unicode, which is the exact character set defined by the Universal Character Set (UCS) of ISO/IEC 10646 [4]. The Unicode character set encompasses most characters used in every major language written today. To allow compatibility with most of the existing systems and to prevent ambiguity among different encodings, the DO-IRP mandates UTF-8 [4] to be the only encoding used for identifiers. The UTF-8 encoding preserves any ASCII encoded names so as to allow maximum compatibility with existing systems without causing conflict.

Prefixes are case-insensitive (for ASCII letters), and, by default, identifiers are case sensitive. However, any individual DO-IRP service may define its identifier space such that ASCII characters within any identifier under that identifier space are case insensitive. Each identifier associated with a digital object will resolve to an identifier record. A user that requests resolution of an identifier will normally be provided with the identifier record. Most identifier records are intended to be made publicly available, but some may be restricted at the discretion of those creating the identifier records, in which case the resolution response should indicate explicitly that the identifier record is not available.

2.2 PREFIXES AND PREFIX REGISTRATION AND RESOLUTION SERVICE

A zero-delimiter prefix allotted to a party and registered in a Prefix Registration and Resolution Service (PRS), as indicated below, is known as a *credential* for that party. Such credentials shall be

numbers, unless specifically agreed for selected cases in the policies and procedures developed by a PRS. Prefixes are then derived from the credential in a tree structure. Client software may interrogate the PRS to determine the relevant prefix information. A PRS is used in conjunction with a set of DO-IRP services and may be integrated with or administered separately from them. A PRS may be administered by multiple organizations that agree to cooperate with each other on the allotment of unique prefixes Further details of the structure of the identifier record are discussed in section 4 For clarity, a credential as described in this section is different from a message credential that is used in communicating messages and which is described later in this document.

The use of a tree structure in essence enables one or more prefixes to be derived from a given prefix. There is no intrinsic administrative relationship between the identifier space represented by a given original prefix, and the identifier space represented by prefixes derived from it. The identifier space represented by a given prefix, and those corresponding to prefixes derived from the given prefix may be served by the same or by different DO-IRP services, and they may or may not share any administration privileges.

To be precise, this document will say that a prefix <X>.<Y> is "derived from" or is a "derived prefix of" or a "derived prefix under" the prefix <X>, where <X> and <Y> may themselves consist of multiple prefix segments.

The means by which unique prefixes are allotted to, or generated for use by potential administrators is not addressed in this document, but it is strongly recommended that they be numeric. Prefixes may be derived from numeric prefixes using a convention of "delimiters" known as "dots". Prefixes with one or more delimiters may include any alphanumeric string after each such delimiter. The PRS may be composed of all prefixes; or it may consist of a subset truncated at a delimiter, which might be all zero-delimiter prefixes, or all zero and one-delimiter prefixes, and so forth.

3 SERVICE MODEL

3.1 RESOLUTION OVERVIEW

DO-IRP provides an identifier-to-"identifier record" binding service, which can be used in private systems, but is otherwise assumed to be made available in the Internet.

From the client's point of view, the DO-IRP provides access to a distributed collection of identifier records that contain state information about the digital objects being identified. Different identifier records may be maintained by different servers at different network locations. In brief, the system uses a two-step resolution process as follows. Given an identifier, the user's client software first attempts to locate the Local Identifier Service (LIS) responsible for resolving that

identifier by querying the PRS for information regarding the prefix in the identifier. The PRS returns information about the prefix including the location of the LIS from which the identifier record may be obtained, if it exists.

Once the LIS is determined by the client software, the client requests the identifier record corresponding to the identifier, if it exists, from that LIS. It is assumed that most LISs designated by the PRS will make their records available. However, sensitive (or other) identifier records may be further cordoned off in restricted access LISs or using other LISs that are only accessible by indirection from another LIS. If an identifier record is not available, the reason for its unavailability shall be provided. The messages exchanged between the client and server are described in section 6.

The DO-IRP service model, described in this section, spells out the workflow a client must use to determine which server it should send its requests to concerning a given identifier record.

3.2 PREFIX IDENTIFIER

Each prefix has a corresponding prefix identifier, which is formed by prepending the system prefix 0.NA to the designated prefix along with a slash between the designated prefix and 0.NA; and this identifier resolves to a corresponding prefix identifier record. For example, the prefix identifier corresponding to the hypothetical prefix 35.1234 is 0.NA/35.1234. Like all other identifiers in DO-IRP services, a prefix identifier is resolvable to an identifier record using the DO-IRP protocol.

3.3 PREFIX IDENTIFIER RECORD

The prefix identifier record can, in principle, contain other information, but within the DO-IRP service model there are three main uses for the prefix identifier record as indicated below:

1. The prefix identifier record specifies the service information which DO-IRP clients will use to send further identifier resolution requests or to administer identifiers under the prefix. See the sections 4.3.2 and 4.3.4 on HS_SITE and HS_SERV.
2. The prefix identifier record specifies permissions which apply to the entire prefix, in particular the permission to create identifiers under the prefix, the permission to create derived prefixes under the prefix, the permission to list identifiers under the prefix, and the permission to list derived prefixes under the prefix. See section 4.3.1 on HS_ADMIN.
3. The prefix identifier record may specify service information which DO-IRP servers may use to create prefix referral responses, indicating to a DO-IRP client that requests for a derived prefix should be sent to that service. See the sections 4.3.3 and 4.3.5 on HS_SITE.PREFIX and HS_SERV.PREFIX.

Historical note: the prefix 0.NA comes from the obsolete term “naming authority”.

3.4 PREFIX REGISTRATION & RESOLUTION SERVICE

A DO-IRP client can obtain the service information (See HS_SITE in section 4.3.2) of the service responsible for an identifier by resolving the prefix identifier for the corresponding prefix. A PRS is a special DO-IRP service where clients can, in general, send requests about prefixes. It administers a registry of prefix identifier records that may be centralized or distributed. However, this document does not address its implementation. A PRS will accept requests about all prefixes, but may send a helpful “prefix referral” response for prefixes not otherwise resolvable by the PRS.

Since prefix identifiers are simply identifiers with the special prefix 0.NA, there is then a prefix identifier which contains information about the prefix 0.NA, including its service information; this identifier is 0.NA/0.NA, and is called the root identifier. DO-IRP clients will in general have a small amount of bootstrapping information about the PRS (since without such information, it is impossible to resolve 0.NA/0.NA), and the clients may periodically update their bootstrapping information by resolving the then current record for 0.NA/0.NA.

DO-IRP services may be implemented to provide identifier registration, resolution, or other related services for digital objects on a centralized or de-centralized basis, or some combination thereof.[5]

The management of the PRS is not otherwise addressed in this specification.

3.5 DETERMINING THE RESPONSIBLE IDENTIFIER SERVICE

The expected workflow for a DO-IRP client to send a resolution request is as follows.

1) After syntactically extracting the prefix from the identifier, the client sends the resolution request for the prefix identifier to the PRS. (Although the entire prefix identifier record may be requested, the resolution request can request only the element types HS_SITE and HS_SERV, which are how the prefix identifier record defines a service. See the sections 4.3.2 and 4.3.4 on HS_SITE and HS_SERV.)
    a) Follow any indicated referrals that result from a resolution request, possibly sending the same resolution request to a different service.

[5] Where such services are provided on a de-centralized basis, it is advisable that the collaborating parties, whether an organization, individual or other entity, agree on a lead party to be responsible for managing the process by which credentials are allotted to assure their uniqueness.

   b) If the eventual response is “identifier not found”, or the prefix identifier record does not define a service, return an error (no service found).

   c) Otherwise, the prefix identifier record has elements which define the service responsible for the identifier of interest.

2) Send the original identifier resolution request to the service found in step (1) and follow any indicated referrals, possibly sending the same request to a different service.

A DO-IRP client may, when appropriate, shortcut step (1) for finding the responsible service. Although the information may be found in a local cache or even be part of a local client configuration, the described workflow is intended to be the canonical way to determine the responsible service. The client may request and subsequently validate signatures of the responses in the above “back and forth” exchanges to ensure that only the expected servers are responding.

Consider as a typical example a client which wishes to resolve the record for the identifier 35.500.1234/ABC.

1) The client first sends a resolution request for 0.NA/35.500.1234 to the PRS. If it has the prefix identifier record, the PRS responds with the requested record.

2) If the PRS does not have the requested prefix identifier record, it may respond with a prefix referral response, if it has one to offer. In either case, the response defines the service sites for a service X.

3) If a referral response has been provided, the client now sends the same resolution request for 0.NA/35.500.1234 to the service X.

4) If the service X responds with an identifier record for 0.NA/35.500.1234; the elements of the record define a service, Y.

5) The client sends a new resolution request for 35.500.1234/ABC to the service Y.

6) Y responds with the desired identifier record if it exists.

Any given request, which is processed by a client in the canonical way without use of caching or other special configuration, is expected to perform in the same predictable fashion. It may have fewer or more referrals at both stages, but it will always have at least two stages: first, the resolution of the service information obtained from the prefix identifier record; and second, the sending of the actual identifier request to the service defined in the prefix identifier record.

3.6 STRUCTURE OF A DO-IRP SERVICE.

A DO-IRP service is, in general, a collection of multiple service sites, each hosting a full replica of the identifiers of the service. Each service site, in general, consists of multiple servers, each with a particular subset of the identifiers of the service (typically determined by a hashing algorithm). The DO-IRP protocol specifies the procedure for a client to determine which site within the service and which server within the site should be contacted (see section 7.1).

4 DATA MODEL

Each identifier may have a set of elements assigned to it, which are collectively known as the identifier's "Identifier Record". There is no inherent limit to the number of elements in each identifier record. Records can optionally contain an element which is a signature certifying the contents of the record; additionally, if requested by a client, messages used to transmit records shall be signed for security. Each such element has a specified "type" which is represented by a unique identifier of its own. DO-IRP servers maintain these Identifier Records and typically return an instance of each such requested Identifier Record in response to a request for resolution of that designated identifier. More tailored requests can result in specific elements of an identifier record being shown in a given returned identifier record.

4.1 IDENTIFIER RECORD

An identifier record thus consists of one or more elements, each of which contains multiple fields including its type, a value associated with that type, and a unique index number that distinguishes that element from other elements in the record, especially those with the same type. The specific type shown in an element may define the syntax, structure, possible semantics, or any other necessary descriptive characteristics of the element's value field. Each element also contains a set of administrative information such as a timestamp, Time to Live ("TTL") and certain associated permissions. Pre-defined types are discussed in section 4.3.

Figure 4.1 below shows the identifier "35.1234/abc" with a sample record with three elements with indexes 1,2 and 3; The element 1 is shown in detail below.

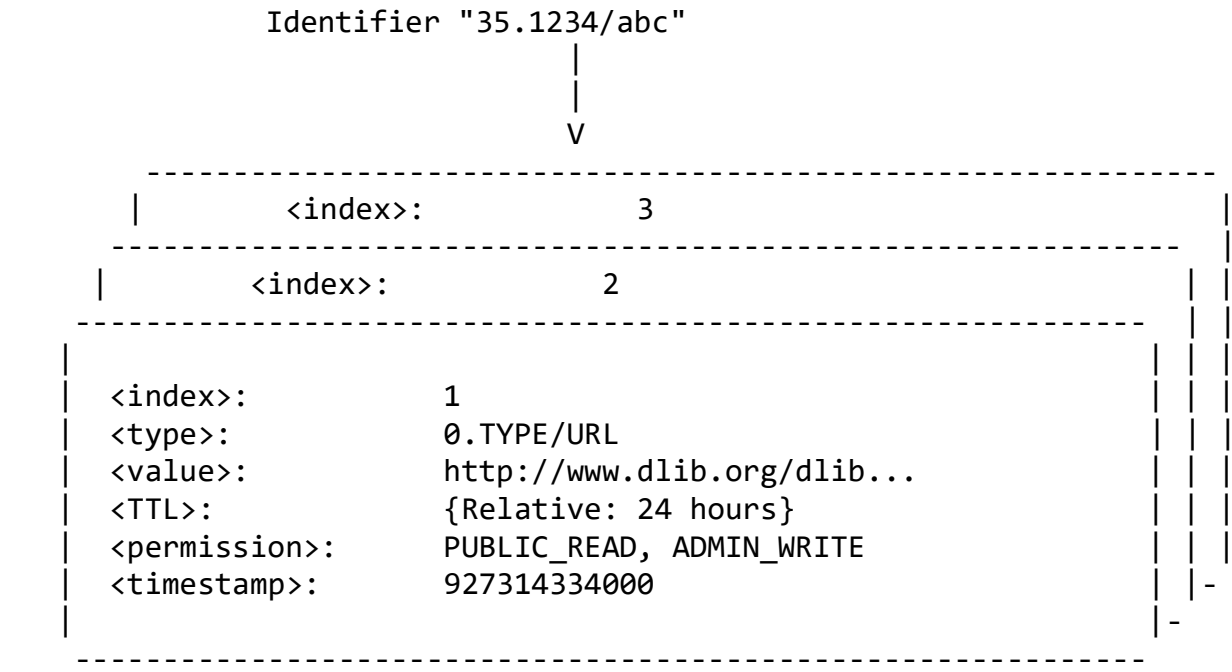

Figure 4.1: Identifier "35.1234/abc" and its set of three elements

Figure 4.1 shows a single element whose index is set to 1. The type for this element is URL [5], which is represented in the system by the resolvable identifier 0.TYPE/URL. In general, a type is represented as a unique identifier and will either be a system type or a user created type. The URL data as stated in the <value> field is "http://www.dlib.org/dlib... ". The TTL (time to live) entry suggests that the record should be cached no more than 24 hours before the source of the information should be consulted again. The <permission> field grants anyone permission to read; but only the administrator has permission to update that particular element.

Thus, an element may be thought of as group of fields which are defined below, along with the standard binary encoding of an element used by the DO-IRP protocol.

<index>
An unsigned 32-bit integer that uniquely differentiates an element from all the other elements in the identifier record. The index 0 is reserved. For maximum compatibility with existing implementations, indexes greater than or equal to 2^31 (which would represent negative values if interpreted as signed integers) should not be used.

<type>
A UTF8-string that specifies the identifier for the type of the value field in the element. Note that throughout this document, a UTF8-string consists of a 4-byte unsigned integer followed by an UTF-8 encoded character string. The integer specifies the number of octets in the character string.
The <type> field is used to specify the identifier of the type that defines the syntax, format, and possible semantics of the data in the element's <value> field. The identifier for a type

can be specified by the creator of the element as desired, but this standard recommends that it be an identifier resolvable using DO-IRP to make it globally resolvable and accessible. See section 9 on Type Identifiers Consideration for more details.
The UTF8-string in the <type> field shall not end with the "." character. However, for clarity, the "." character may appear in the end of the <type> prefix used in an identifier query. This prefix string is used to query for all elements under a common type hierarchy. For example, one may query for all elements under the type hierarchy "a.b" (e.g., elements of <type> "a.b.x", "a.b.y" and "a.b.z") by setting the <type> parameter of the query to "a.b. ". Note here that the <type> prefix in the query must specifically end with the "." character. Details of the query operation can be found in the protocol specification in section 7.2.

<value>
A sequence of octets (preceded by its length in a 4-byte unsigned integer). The syntax, format, and semantics of these octets are determined by the <type> field.

<permission>
An eight-bit bit-mask for access control of the element. Access control is defined in terms of read and write permissions, applicable to either the general public or administrator(s). DO-IRP servers must evaluate each element's permissions before fulfilling any client request to make sure that only properly authenticated and authorized clients receive the element. Each element can have its permission field specified as any combination of the following bits:

- PUBLIC_WRITE (0x01)
  permission that allows anyone to modify or delete the element.
- PUBLIC_READ (0x02)
  permission that allows anyone to read the element.
- ADMIN_WRITE (0x04)
  permission that allows any authenticated administrator with Modify_Element / Delete_Element privileges (see 4.3.1) to update or delete the element
- ADMIN_READ (0x08)
  permission that allows any authenticated administrator with "Authorized_Read" privilege (see section 4.3.1 below) to read the element.

It is normally intended that only administrators shall be granted write permissions (in addition to read permissions). An element with no PUBLIC_READ nor ADMIN_READ permission will not be accessible via DO-IRP by anyone except by the DO-IRP server administrator, once set. Such an element may be used, for example, to store secret keys for authentication purposes. An element with neither PUBLIC_WRITE nor ADMIN_WRITE permission makes the element immutable and it cannot be deleted via DO-IRP. The most common permissions are "1110" (admin read and write, public read) and "1100" (admin read and write, no public access).
The administrator for a given identifier must specify the permission for each element. It is recommended that implementations use PUBLIC_READ, ADMIN_READ and ADMIN_WRITE as

the default permission for each element.
The more significant bits of the permission field are reserved.

<TTL>
An octet followed by a 4-byte integer that specifies the Time-To-Live of the element. It is used to describe how long the element should be cached by clients before the source of the information (i.e., the corresponding DO-IRP server) should again be consulted. A zero value for a TTL indicates that the element should only be used for the transaction in progress and should not be cached. Any non-zero TTL is defined in terms of a TTL type (specified in the first octet), followed by the TTL value (the 32-bit integer that follows the TTL type). The TTL type indicates whether the TTL value is relative (octet 0x00) or absolute (octet 0x01). An absolute TTL value is an unsigned integer which defines the time to live as an expiration in terms of seconds since 00:00:00 UTC, January 1st 1970. A relative TTL specifies the time to live in terms of the number of seconds elapsed since the element was obtained by the client from any DO-IRP server. A relative TTL which would be negative if interpreted as a signed integer indicates that the element should not be cached.

<timestamp>
A 4-byte unsigned integer that records the last time the element was updated at the server. The field contains elapsed time since 00:00:00 UTC, January 1970 in seconds.

<references>
This field is deprecated and should be populated with four zero octets 0x00, 0x00, 0x00, 0x00. Since older implementations may have used this field, clients may need to process it in which case the structure of the field is a 4-byte integer followed by list of "references" each composed of a string. The initial integer specifies the number of references in the list. Each reference in the list is a UTF8-string followed by a 4-byte integer.

4.2 IDENTIFIER RECORD CONSIDERATIONS

By default, DO-IRP servers return all the elements with public-read permission in response to any resolution request. It is possible for a client to ask for a subset of those elements with a specific type (e.g., all URLs stored in the identifier record). The client may also ask for a specific element based on a specific index.

Each element can be uniquely referenced by the combination of the identifier and its element index. Care must be taken when changing the index as it may break an existing reference to the element. For example, suppose the record of identifier X/Y has an element whose index is 1. That element may be referred to as 1:X/Y. If the administrator changes the index from 1 to 2, the reference to 1:X/Y will become obsolete. To remain accurate, any reference to the element will have to change to 2:X/Y.

Elements within an identifier record may or may not have contiguous index numbers. Nor should it be assumed that the indexes will start with 1. An administrator may assign an element with any index as long as each index is a unique integer within the record.

An element may be "privatized" or "disabled" by clearing the "public-read" bit on its <permission> field. This limits read-access to the administrator only. The "privatized" element can then be used to keep any historical data (on behalf of the administrator) without exposing it to public. Such an approach may also be used to keep any obsolete identifier or prefix from being reused accidentally. This is also used for the HS_SECKEY type; see section 4.3.7.

4.3 PRE-DEFINED ELEMENT TYPES

Type is a required field in an element, although the exact type to be used is open-ended. Some types are defined in this document and have specific server and client processing semantics. DO-IRP implementations must use the defined types whenever applicable. New types can be defined if different processing rules were to apply; in such cases, those new types may be registered by an organization as identifiers for resolution using DO-IRP under a prefix controlled by the organization.

Types defined in this document are assumed to have an implicit prefix of "0.TYPE", although the prefix is generally omitted when writing about the use of these types.

The following types have defined DO-IRP server and client processing semantics: HS_ADMIN, HS_SITE, HS_SITE.PREFIX, HS_SERV, HS_SERV.PREFIX, HS_PUBKEY, HS_SECKEY, and HS_VLIST.

Additionally, the types HS_ALIAS, HS_CERT, and HS_SIGNATURE, while not necessary for DO-IRP service use, are considered standard system types that cut across multiple applications; and all are registered under the prefix "0.TYPE".[6]

The following sub-sections provide detailed descriptions of these system types.

4.3.1 IDENTIFIER ADMINISTRATOR: HS_ADMIN

An administrative operation on an identifier record or on the DO-IRP service (e.g., add, delete or modify an element) can only be performed by an administrator that is authenticated and that has adequate privileges.

[6] Historical note: the term HS in the name of these types comes from the legacy notation that identified these types as Handle System (HS) types.

We defined an “administrator” as an entity represented by an identifier and an element index within that identifier record. This identifier-index is a reference to an element that must contain a public key or secret key to be used when challenging the entity to authenticate itself.

An authenticated administrator is an entity that has demonstrated that it possesses either the private key matching the public key, or the secret key, pointed to by the administrator’s identifier-index element reference.

The privileges or authorizations that an administrator has over an identifier record is specified in one or more elements of type HS_ADMIN within that identifier record. Each HS_ADMIN element can be used to define a set of administrators sharing the same administration privilege. Identifiers with multiple administrators of different privileges may have multiple HS_ADMIN elements. HS_ADMIN elements are used by DO-IRP servers to authenticate administrators before fulfilling any DO-IRP administration request. However, as an implementation option, an identifier record without an HS_ADMIN element can be modified by a qualified DO-IRP server administrator.

An HS_ADMIN element is one whose <type> field is HS_ADMIN and whose <value> field consists of the following entries encoded as binary data:

> <AdminPermission>
> A 16-bit bit-mask that defines the administration privilege of the administrator or set of administrators identified by the <AdminRef> entry; see below for the permissions.
>
> <AdminRef>
> Specifies the record’s administrator. It consists of the administrator’s identifier and element index (as defined above), encoded as a 4-byte length followed by that many UTF-8 encoded bytes, followed by a 4-byte unsigned integer for the index of the element that is referenced in the administrator’s identifier record. The referred element in the administrator’s identifier record must either be an identity authentication element (such as an HS_PUBKEY or HS_SECKEY element) or an HS_VLIST element, which specifies a group of administrators consisting of a list of one or more administrator identification and authentication elements references. (See HS_VLIST paragraph below for more description as well as HS_VLIST in section 4.2.8 for its detailed specification). An index value of 0 implies that the reference is to all elements in the administrator’s identifier record; this means all HS_SECKEY or HS_PUBKEY elements in that record can be used.

The <AdminRef> entry may refer to an administrator with the same identifier the record of which has the HS_ADMIN element. In this case, authentication of the administrator does not rely on any other identifiers and does not require any additional identifier resolution. Alternatively, the <AdminRef> entry may refer to an administrator in a different identifier record requiring additional resolution operations. As a consequence, HS_ADMIN elements from different identifiers may share common administrator(s), even across different prefixes. This feature provides a flexible solution to the task of administrating large collections of identifier records which could range from a few

identifier records in a single prefix to very large numbers of identifier records spanning multiple prefixes.

The HS_VLIST element is used to specify a list of one or more administrators. It is typically used to define a list of administrators that can administer an element. This list is referred to as an administrative group. The use of HS_VLIST allows a single HS_ADMIN element to have as many individual administrators as specified in the HS_VLIST. Each administrator reference is defined in terms of an <index>:<identifier> pair. An administrator group may also contain other administrator groups as its members. This allows administrator groups to be defined in a hierarchical fashion. Care must be taken, however, to avoid cyclic definition of administrators or administrator groups. Multiple levels of administrator groups should be avoided due to their lack of efficiency, but will not be signaled as an error. Client software should be prepared to detect any potential cyclic definition of administrators or <AdminRef> entries that point to non-existent elements and treat them as an error.

An identifier record can have multiple HS_ADMIN elements, each of which references a different administrator and with possibly different permissions. Different administrators can play different roles or be granted different permissions. For example, the identifier record that corresponds to a prefix, such as the one associated with the identifier "0.NA/35.978", may have two administrators, one of which may only have permission to create new identifiers under the prefix, while the other may have permission to create new derived prefixes (e.g., "35.978.123").

Beyond its use in the HS_ADMIN, the HS_VLIST could be used in other situations where a grouping of identifiers is needed.

The set of possible permissions for an administrator is defined as follows:

Add_Identifier (0x0001)
This permission, when set in a prefix identifier record, allows an authenticated administrator to create new identifiers under that prefix. This permission is only meaningful when set in an identifier record pertaining to a prefix, as it is a prefix-only permission.

Delete_Identifier (0x0002)
This permission allows an authenticated administrator to delete the identifier record.

Add_Derived_Prefix (0x0004)
This permission, when set in a prefix identifier record, allows an authenticated administrator to create new prefixes derived from the stated prefix identifier. This permission is only meaningful when set in a prefix identifier record and it is a prefix-only permission.

Reserved (0x0008)
This bit is reserved for historical reasons.

Modify_Element (0x0010)
This permission allows an authenticated administrator to modify any elements other than HS_ADMIN elements. HS_ADMIN elements are used to define administrators and are managed by a different set of permissions (as described next).

Delete_Element (0x0020)
This permission allows an authenticated administrator to delete any element other than the HS_ADMIN elements.

Add_Element (0x0040)
This permission allows an authenticated administrator to add elements other than the HS_ADMIN elements.

Modify_Admin (0x0080)
This permission allows an authenticated administrator to modify HS_ADMIN elements.

Remove_Admin (0x0100)
This permission allows an authenticated administrator to remove HS_ADMIN elements.

Add_Admin (0x0200)
This permission allows an authenticated administrator to add new HS_ADMIN elements.

Authorized_Read (0x0400)
This permission grants an authenticated administrator read-access to elements with the ADMIN_READ and without PUBLIC_READ permission. Administrators without this permission will not have access to elements that require authentication for read access.

List_Identifiers (0x0800)
This permission allows an authenticated administrator to list all identifiers under a designated prefix, even if such identifiers are managed in a distributed fashion on multiple servers. Identifiers that are based on prefixes derived from the specified prefix are not included in the listing. This is a prefix-wide setting and must be set on the respective prefix identifier record.

List_Derived_Prefixes (0x1000)
This permission allows the administrator to list all prefixes derived from a designated prefix. If such derived prefixes have in turn their own derived prefixes, those further derivatives are also included in the listing as long as such derived prefix records are on the same DO-IRP service on which this listing operation is performed. This is a prefix-wide setting and must be set on a prefix identifier record.

Administrator permissions are encoded in the <AdminPermission> entry in the <value> field of any HS_ADMIN element. Each permission is encoded as a bit flag. The permission is granted if the flag is set to 1.

HS_ADMIN elements are used by DO-IRP servers to authorize the administrator before fulfilling any administrative requests. The server authenticates a client by checking whether the client has possession of the secret key or the private key that matches the secret key or the public key

corresponding to the identity claimed by the client; the server considers an authenticated client authorized if that identity matches any of the administrator references (i.e., AdminRef). The authentication is carried out via the DO-IRP authentication process, see section 5.2.

HS_ADMIN elements may require authentication for read access (when no PUBLIC_READ permission is configured at element level) in order to prevent public exposure of the data. Additionally, if a secret key is used instead of a public key, the administrator reference that contains the administrator's secret key should not have PUBLIC_READ permission so that the key is not publicly visible.

Modify_Element, Delete_Element, Modify_Admin, and Remove_Admin permissions are not usable if the respective element is not configured with the ADMIN_WRITE or PUBLIC_WRITE permission (see 4.1).

4.3.2 SERVICE SITE INFORMATION: HS_SITE

HS_SITE is used to provide information about a DO-IRP service to inform a DO-IRP clients how to contact it. This information specifies the server's address, its protocol version, its public key. Each prefix identifier should have one or more HS_SITE elements in its prefix identifier record. This is called the service information for the prefix and denotes, among other things, the location of servers where any existing identifier that is based on that prefix can be created, updated, deleted, or resolved. A user that tries to resolve an identifier that does not exist will get a "identifier not found" response from the local service specified in the HS_SITE information. Attempts to resolve an invalid identifier, namely one that does not exist within the designated LIS, will get a "identifier not found" response from that local service.

The service information is managed by the system administrator. It must reflect the configuration of the DO-IRP service for each of the prefixes it manages. An additional layer of indirection is provided by the use of HS_SERV, which is called a service identifier; HS_SERV allows multiple prefixes to reference a single set of HS_SITE elements, as described in section 4.3.4. DO-IRP clients depend on the service information to locate a responsible DO-IRP server before they can send their service requests. The service information can also be used by clients to authenticate any service response from the server.

An HS_SITE element is one whose <type> field is HS_SITE and whose <value> field consists of the following entries encoded as binary data:

> <Version>
> A 2-byte value that identifies the version number of the data format used in the encoding of the <value> field of the HS_SITE element. It is defined to allow backward compatibility over time. This section of this document defines the HS_SITE encoding with version number 1.

Note that this version number is not the same as the DO-IRP protocol version number used in request and response messages.

<ProtocolVersion>
A 2-byte integer value that identifies the highest DO-IRP protocol version understood by servers of the site. The higher byte of the value identifies the major version and the lower byte the minor version. Details of the message format for DO-IRP protocol version 3.0 are specified in section 6.2. Although the terminology used is the same, this field does not denote the protocol version of this DO-IRP specification. Rather, it denotes the protocol version as specified by the prefix administrator for the implementation of the local service that supports the identifier record. Local services which support the DO-IRP protocol version 3.0 specified in this document should be described with HS_SITE elements specifying <ProtocolVersion> 3.0.

<SerialNumber>
A 2-byte integer value that should be increased by administrators each time the HS_SITE element gets changed. For uses currently envisioned, it is acceptable for the 16-bit serial number to wrap around. It is used in the DO-IRP protocol to synchronize the HS_SITE elements between client and server. In general, clients are expected to synchronize HS_SITE elements when they expire from cache, which they do by re-resolving the identifier.

<PrimaryMask>
An 8-bit mask that identifies the primary site(s) of the DO-IRP service. The first bit of the octet is the <PrimarySite> bit. It indicates whether the HS_SITE element is a primary site. A primary site is the one that supports administrative operations for its identifiers. The second bit of the octet is the <MultiPrimary> bit. It indicates whether the service has multiple primary sites. A <MultiPrimary> entry with zero value indicates that the service has a single primary site and all administration has to be done at that site. A non-zero <MultiPrimary> entry indicates that the service has multiple primary sites. Each primary site may be used to administer identifiers managed under the service.

<HashOption>
An octet that identifies the hash option used by the service site to distribute identifiers among its multiple servers. This is not meaningful for a typical service site with a single server. Valid options include HASH_BY_PREFIX (0x00), HASH_BY_SUFFIX (0x01), or HASH_BY_IDENTIFIER (0x02). These options indicate whether the hash operation should only be applied to the prefix portion of the identifier, or only the suffix portion of the identifier, or the entire identifier, respectively. See section 7.1 for the hashing algorithm used by each service site to distribute identifiers among its servers.

<HashFilter>
An UTF8-string entry reserved for future use.

<AttributeList>
Attributes used to add literal explanations of the service site. An attribute is a <type>:<value> pair. The entry consists of a 4-byte integer followed by a list of UTF8-string pairs. The integer

indicates the number of attributes (UTF8-string pairs) that follow. The most common <type> is "desc", for which the <value> should contain a description of the organization hosting the service site. Other attributes may be defined to help distinguish the service sites from each other. Resolvers may use the attribute type "alt_addr" as an indication that a site's server (or "alt_addr.<ServerID>" for one of a site's multiple servers) is reachable via an alternate IP address. This can be used to indicate a site with dual-stack reachability over IPv4 and IPv6. Resolvers may use the attribute types "domain" and "path" (or "domain.<ServerID>" and "path.<ServerID>" for use with HTTP(S) tunneling as described in section 6.1.2.3.

<NumOfServer>
A 4-byte integer that defines the number of servers in the service site. The entry is followed by a list of <ServerRecord>s. Each <ServerRecord> defines a server that is part of the service site.
Each <ServerRecord> consists of the following data fields:

```
<ServerRecord> ::= <ServerID>
                   <Address>
                   <PublicKeyRecord>
                   <ServiceInterface>
```

where each field is defined as follows:

<ServerID>
A 4-byte unsigned integer that uniquely identifies a server process under the service site. <ServerID>s do not have to begin with 1 and they don't have be consecutive numbers. They are used to distinguish servers under a service site from each other. Note that there can be multiple servers residing on any given computer, each with a different <ServerID>.

<Address>
The 16-byte IPv6 address of the server. Any IPv4 address should be presented as either ::xxxx:xxxx or ::FFFF:xxxx:xxxx (where xxxx:xxxx can be any 4-byte IPv4 address).

<PublicKeyRecord>
A 4-byte integer followed by a byte-array that contains the server's public key (server authentication key). The integer specifies the size of the byte-array. For key types described in this specification, the byte-array (for the public key) consists of a number of parts: a UTF8-string that describes the key type, a two-byte option field reserved for future use, and a key-type-dependent number of length-prefixed byte-arrays that describe the public key itself. The key types in current use are "DSA_PUB_KEY," where there are four byte-arrays after the two-byte option field for the four DSA parameters q, p, g, and y; and "RSA_PUB_KEY", where after the two-byte option field are two byte-arrays for the exponent and modulus, followed by an empty byte-array (four zero bytes). Other key types may be useful to consider in the future.

The public key in the <PublicKeyRecord> can be used to authenticate any service response from the DO-IRP server.

<ServiceInterface>
Consists of the following data fields:

```
<ServiceInterface> ::=  <InterfaceCounter>
                        * [ <ServiceType>
                            <TransportProtocol>
                            <PortNumber> ]
```

A 4-byte integer followed by an array of triplets consisting of <ServiceType, TransportProtocol, PortNumber>. The 4-byte integer specifies the number of triplets. Each triplet lists a service interface provided by the server. For each triplet, the <ServiceType> is an octet (as a bit mask) that specifies whether the interface is for administration (0x01), resolution (0x02), or both (0x03). The <TransportProtocol> is also an octet (as a bit mask) that specifies the protocol. Possible protocols include UDP (0x00), TCP (0x01), HTTP (0x02), and HTTPS (0x03). The <PortNumber> is a 4-byte unsigned integer that specifies the port number used by the interface. The conventional port number used by DO-IRP servers is 2641 for UDP and TCP, and 8000 for HTTP.

Each server within a service site is responsible for a subset of identifiers managed by the DO-IRP service. Clients can find the responsible server by performing a common hash-operation. See section 7.1.

4.3.3 SERVICE SITE INFORMATION: HS_SITE.PREFIX

The HS_SITE.PREFIX has the exact same format as the HS_SITE type described above. Like HS_SITE elements, HS_SITE.PREFIX elements are used to describe service sites of a DO-IRP service. HS_SITE.PREFIX elements may be assigned to prefix identifiers to designate that derived prefixes may be resolved at the specified service. A prefix identifier associated with a set of HS_SITE.PREFIX elements indicates that derived prefixes of the prefix, if any, may be managed by the service described by the HS_SITE.PREFIX elements.

HS_SITE.PREFIX and HS_SERV.PREFIX elements are not generally used directly by clients, but by a server which receives client requests about prefix identifiers, in order to determine whether to respond with a RC_PREFIX_REFERRAL response instead of a RC_ID_NOT_FOUND response; see section 7.4.

4.3.4 SERVICE IDENTIFIER: HS_SERV

Any DO-IRP service can be defined in terms of one or more HS_SITE elements. These HS_SITE elements may be assigned directly to the relevant prefix identifier, or an additional level of indirection may be introduced through the use of an HS_SERV element in the prefix identifier record. The value of the HS_SERV element contains the service identifier, meaning an identifier

whose record contains the HS_SITE elements defining the DO-IRP service. This way, the HS_SITE elements can be maintained in a separate record

Use of service identifiers allows sharing of service information among multiple prefixes. It also allows changes to service configuration (e.g., adding a new site) to be made in one place rather than in every prefix identifier involved.

Although not typical, a prefix identifier may have multiple HS_SITE and multiple HS_SERV elements. In such a case the service information for the prefix should be considered as the concatenation of the HS_SITE elements in the prefix identifier record, together with the service information from all of the HS_SERV elements.

The use of service identifiers raises several special considerations. Multiple levels of service identifier redirection should be avoided due to their lack of efficiency, but are not signaled as an error. Looped reference of service identifiers or HS_SERV elements that refer to non-existent service identifiers should be caught and error conditions passed back to the user.

4.3.5 SERVICE IDENTIFIER: HS_SERV.PREFIX

HS_SERV.PREFIX serves the same role with respect to HS_SITE.PREFIX as HS_SERV serves with respect to HS_SITE. If a prefix identifier has an HS_SERV.PREFIX element, the data of that element is an identifier, whose corresponding record describes, via one or more HS_SITE.PREFIX elements and/or recursively via HS_SERV.PREFIX elements, the service information at which clients may be able to resolve derived prefixes of the original prefix.

4.3.6 IDENTITY: HS_PUBKEY

An element of type HS_PUBKEY stores a public key. The element can be used as an administrative identity, for referring to in HS_ADMIN or HS_VLIST elements for authorization, or for identifying the administrative identity in DO-IRP authentication. A reference to the element for use as an administrative identity is as a pair of the identifier and the index of the element within the identifier record; in this document this is often written with a colon as <index>:<identifier>.

The <value> of the element is a binary encoding of the public key which for key types considered in this specification is as follows. First, there is a UTF8-string that describes the key type; then, a two-byte option field reserved for future use; and finally, a key-type-dependent number of length-prefixed byte-arrays that contains the public key itself. The key types in current use are "DSA_PUB_KEY", where there are four byte-arrays after the two-byte option field for the four DSA parameters q, p, g, and y; and "RSA_PUB_KEY", where after the two-byte option field are two byte-arrays for the exponent and modulus, followed by an empty byte-array (four zero bytes).

4.3.7 IDENTITY: HS_SECKEY

An element of type HS_SECKEY is used to store a secret key. The element can be used as an administrative identity, for referring to in HS_ADMIN or HS_VLIST elements for authorization, or for identifying the administrative identity in DO-IRP authentication.

The <value> of the element is the secret key. In order to protect the secret key, the <permission> of the element should be set to forbid PUBLIC_READ. This is the only common usage of any <permission> other than PUBLIC_READ, ADMIN_READ, and ADMIN_WRITE.

Use of public keys is recommended over use of secret keys for DO-IRP authentication.

4.3.8 VALUE LIST: HS_VLIST

HS_VLIST allows referencing a list of elements in other identifiers, and allows the referenced elements to be updated without requiring an update of the referring identifier record. An HS_VLIST element is one whose <type> is HS_VLIST and whose <value> consists of a 4-byte unsigned integer followed by a list of references to other elements. The integer specifies the number of references in the list. The references may refer to elements under the same identifier or elements from other identifiers. Each reference is encoded as an UTF8-string followed by a 4-byte unsigned integer that identifies the referenced identifier and its index.

HS_VLIST elements may be used to define administrator groups for identifiers. Each administrator (as defined in 4.3.1) in the HS_VLIST includes is a member of the administrator group. Each element reference is defined in terms of an <index>:<identifier> pair. An administrator group may also contain other administrator groups as its members. This allows administrator groups to be defined in a hierarchical fashion. Care must be taken, however, to avoid a cyclic definition of administrators or administrator groups. Multiple levels of administrator groups should be avoided due to their lack of efficiency, but will not be signaled as an error. Client software should be prepared to detect any potential cyclic definition of administrators that refer to non-existent elements and treat them as errors.

4.3.9 ALIASES: HS_ALIAS

It is possible that the set of elements (i.e., type-value pairs) of a digital object may be included in other digital objects and thus multiple identifiers could be associated with the same information represented in digital form, thus creating, in effect, multiple different digital objects all with the same information represented in digital form. DO-IRP does not specify a specific mechanism for identifier records to reference all such alternatives, but various DO-IRP implementations and their users should use the pre-defined HS_ALIAS type in such cases.

An HS_ALIAS element is one whose <type> field is HS_ALIAS and whose <value> field contains a reference to another identifier. An identifier whose record contains an HS_ALIAS element is an alias to the identifier referenced in the HS_ALIAS element. An alias record should not have any additional elements other than HS_ALIAS or HS_ADMIN (for administration) elements. This is necessary to prevent any inconsistency between an identifier and its aliases.

During an identifier resolution, a client may get back an HS_ALIAS element. This indicates that the identifier in question is an alias identifier. The client may then retry the query against the identifier specified in the HS_ALIAS element until final results are obtained.

The use of HS_ALIAS introduces a number of special considerations. For example, multiple levels of aliases should be avoided for the sake of efficiency, but are not signaled as an error. Alias loops and aliases that point to non-existent identifiers should be caught and error conditions passed back to the user.

4.3.10 CRYPTOGRAPHIC CLAIMS: HS_CERT

HS_CERT and HS_SIGNATURE elements are used for offline signatures of identifier records. However, if an implementer chooses to do so, he/she could develop interoperable client and server software without making use of HS_CERT and HS_SIGNATURE. The DO-IRP protocol (see sections 6.2.3 and 6.2.4) allows a client to authenticate responses received from a server by requesting that the server include a request digest and sign the response. Offline signatures complement this by including in an identifier record an HS_SIGNATURE element, which is a claim that certain elements are present, signed by an identified entity; and HS_CERT elements, which form a chain of trust which can be used to determine whether the entity signing an HS_SIGNATURE element has the authority to state which elements should be in the identifier record. By means of HS_SIGNATURE and HS_CERT elements, a client can validate the contents of identifier records, regardless of the trustworthiness of the communication channel by which it obtains them.

DO-IRP servers can use the HS_CERT chain of trust as a parallel authorization mechanism for the creation and update of identifier records. This is particularly useful when a DO-IRP service consists of multiple sites with separate administrative and organizational control; the sites need to mirror an identical collection of records, but in principle do not trust each other. The sites can choose to mirror only records that have a valid HS_SIGNATURE according to a specified chain of trust. Even for direct administration by clients, a DO-IRP server could require that not only the client have correct permissions for the operation according to HS_ADMIN elements, but also that the resulting records have a valid HS_SIGNATURE element.

Both HS_CERT and HS_SIGNATURE elements are serialized as JWT claims sets in a JWS structure (RFC 7515, RFC 7519). They differ in the expected claims, interpretation, and validation.

The HS_CERT element claims set will have the following claims:

- "iss": the issuer of the certificate, as an element reference (<index>:<identifier> pair) or identifier
- "sub": the subject of the certificate, as an element reference (<index>:<identifier> pair) or identifier
- "exp", "nbf", "iat": expiration, not-before, and issued-at dates as numeric seconds since the epoch
- "publicKey": the public key of the subject, in JWK format (RFC 7517) [6]
- "perms": a list of permission objects, where each permission object has the form
  { "handle": "<identifier>", "perm": "<permission>" }
  with "<permission>" one of "everything", "thisHandle", "derivedPrefixes", or "handlesUnderThisPrefix"
- "chain": an optional list of element references (<index>:<identifier> pairs) or identifiers, used to build a chain of trust for validating the certificate issuer (in the absence of an explicit chain, the chain can still be built implicitly, as will be discussed)

An HS_CERT indicates an assertion by one entity (the issuer) that a second entity (the subject) is authorized to sign certain identifier records with its public key. It is typically the case that the issuer and subject reference HS_PUBKEY elements in identifier records; however, the "publicKey" contained in the HS_CERT itself is actually used in validating claims signed by the subject. (It is conventional for the HS_PUBKEY element to be identical to the "publicKey".)

The "perms" claim indicates for which identifier records the subject's signature should be considered authorized, as a subset of the identifier records for which the issuer's signature is authorized:

- { "perm": "everything" } indicates authorization over the same handle records as the issuer
- { "handle": "<identifier>", "perm": "thisHandle" } indicates authorization over <identifier>, and, if <identifier> is a prefix identifier, also identifiers with that prefix
- { "handle": "<prefixIdentifier>", "perm": "derivedPrefixes" } indicates authorization over prefix identifiers derived from <prefixIdentifier>, as well as identifiers with any prefix derived from <prefixIdentifier>
- { "handle": "<prefixIdentifier>", "perm": "handlesUnderThisPrefix" } indicates authorization over identifiers with the prefix whose prefix identifier is <prefixIdentifier>

In no case should the subject be considered to be given authorizations not available to the issuer.

The building of a chain of trust using HS_CERT elements is discussed below.

4.3.11 CRYPTOGRAPHIC SIGNATURES: HS_SIGNATURE

The HS_SIGNATURE element claims set will have the following claims:

- "iss": the issuer of the certificate, as an element reference (<index>:<identifier> pair) or identifier
- "sub": the subject of the certificate, an identifier (the identifier of the record signed)
- "exp", "nbf", "iat": expiration, not-before, and issued-at dates as numeric seconds since the epoch
- "digests": a structure indicating the hashed values of the elements of the signed identifier record. It is an object with two properties: "alg", a string indicating the digest algorithm (e.g. "SHA-256" [7]); and "digests", a list of objects. Each of those objects has two properties: "index", a number indicating the index of an identifier record element, and "digest", a string with a Base64-encoding of the digest (or hash) of the element, as described below.
- "chain": an optional list of element references (<index>:<identifier> pairs) or identifiers, used to build a chain of trust for validating the signature issuer (in the absence of an explicit chain, the chain can still be built implicitly, as will be discussed).

An HS_SIGNATURE indicates an assertion about the elements contained in an identifier record. The issuer is the entity making the claim; the subject is the identifier; the digests indicate the digests (according to a hash algorithm like SHA-256) of the elements. The digests are of the byte-arrays of the elements as encoded for the DO-IRP protocol, but omitting the first eight bytes, which are the index and the timestamp (this is necessary for the client to generate digests of new elements, as the timestamp is generally server-generated).

In order to validate HS_SIGNATURE or HS_CERT claims, a client must build a chain of trust: a sequence of elements where the issuer of one element is the subject of the next element. The first element may be either an HS_SIGNATURE or HS_CERT; all subsequent elements will be HS_CERT. The sequence ends with a “self-signed certificate” where the issuer and subject are the same. A client can use the following algorithm to build the chain. At each point, the client must maintain the result so far (a sequence of elements) and the coming chain (a sequence of element references or identifiers). Initially, the result is the singleton element to be validated, and the coming chain is empty.

1) If the coming chain is empty, replace the coming chain with the "chain" claim from the last element of the result so far; if that is empty or missing, replace the coming chain with a

singleton identifier corresponding to the identifier of the issuer of the last element of the result so far (with any index removed).

2) Take the first reference from the coming chain, now guaranteed to be non-empty. If it has an index, resolve the referenced element to get the next element of the result so far. If not, resolve the entire identifier record for all HS_CERT elements, and from all the HS_CERT elements where the subject of that element is the issuer of the last element of the result so far, choose one which has the latest "iat" (issued-at) time, and that is the next element of the result so far.

3) Remove the just-used first element of the coming chain, and repeat, until either reaching a self-signed certificate, or the result so far is too long according to some configuration (for example 50 elements) which is an error condition.

Once the chain of trust is built, it can be validated according to the following conditions:

1) Each element must be a validly-signed JWS with the public key given in the next element of the chain (or, in the case of the self-signed one, with its own key);
2) The current time must be between the "nbf" (not-before) and "exp" (expiration) times of the claims set;
3) The “root” self-signed certificate at the end of the chain of trust is trusted according to client configuration.

Then a particular HS_SIGNATURE can be further validated against an identifier record as follows:

1) The subject must match the identifier of the record.
2) That identifier must be included in the "perms" claims of each HS_CERT in the built chain.
3) The digests of the HS_SIGNATURE must exactly match some subset of the elements of the identifier record.

Finally, an identifier record is considered validly signed if HS_SIGNATURE elements are valid according to the above, and all elements except HS_SIGNATURE elements and elements (like HS_SECKEY elements) not publicly readable are included in the digests of at least one HS_SIGNATURE element.

The specification assumes the existence of the specific identifier 0.0/0.0, which resolves to a record which is used as a default root of trust for the system, in the absence of more specific client configuration. By default, clients can trust a self-signed certificate where the identifier of the issuer is 0.0/0.0 and the public key is in an HS_PUBKEY element of the record of 0.0/0.0. Clients which are validating must not in general trust a new resolution of the 0.0/0.0 record, but should maintain the

0.0/0.0 as part of bootstrapping information, and only trust a new public key on that record when it is signed by a previously-known root key.

5 OPERATION MODEL

DO-IRP operations can be categorized into those involving resolution and those involving administration. Clients use the DO-IRP resolution service to query for identifier elements. DO-IRP administration allows clients to manage identifiers and identifier records, including adding and deleting identifiers, and updating their recorded elements. It also deals with prefix administration via prefix identifiers. This section explains how various DO-IRP components work together to accomplish these service operations.

Both resolution and administration may require authentication of the client. The authentication can be done via the DO-IRP authentication protocol described later in this section. Whether authentication is required or not depends on the kind of operation involved and the permissions assigned to the relevant element, and policies deployed by the relevant service components.

The DO-IRP protocol specifies the syntax and semantics of each message exchanged between DO-IRP clients and DO-IRP server components. This section provides a high-level overview of the protocol used to accomplish any service operation. The exact programmatic detail of each message (i.e., its byte layout or syntax) is specified later in this document (see sections 6 and 7).

5.1 SERVICE REQUEST AND RESPONSE

DO-IRP servers provide their service in response to client requests. A client may send a request to any DO-IRP server to provoke a response. The response either provides an answer to the request, or a status code with associated information that either refers the request to another service component, asks for client authentication, or signals some error status.

Clients may require digital signatures from a DO-IRP server in order to authenticate any response from the server. The signature can be generated using the server's private key. Clients may verify the signature using the public key available from the service information (refer to the <PublicKeyRecord> entry discussed in 4.3.2). Nothing in this document is intended to specify how private keys should be maintained or managed, but care should be taken to protect their privacy, including ensuring that the private key cannot be compromised if it is temporarily stored and used with the operating environment that is host to the DO-IRP.

A communication session may also be established between any client and DO-IRP server. Each session is identified by a unique session ID managed by the server. A session may be used to manage requests that require multiple interactions. It may also be used to share any authentication information among multiple service transactions. Each session may establish a

session key and use it to authenticate any message exchanged within the session. It may also be used to encrypt any message between the client and the server to achieve data confidentiality.

5.2 AUTHENTICATION PROCESS

DO-IRP supports administration in a computational environment such as the Internet. Access controls can be defined on each identifier record. The DO-IRP authentication protocol is used by any DO-IRP server to authenticate an administrator upon any administration request. The authentication is also necessary when clients query for elements that are read-only by the administrator. DO-IRP administration includes adding, deleting or modifying elements, as well as creating or deleting identifiers and their corresponding identifier records. Prefix administrations are carried out as identifier administrations over the corresponding prefix identifiers.

The DO-IRP authentication protocol does not perform any server authentication. However, a client may authenticate any server response by asking the server to sign its response with a digital signature. The session mechanism uses message authentication codes to ensure that all messages in a session are appropriately authenticated as coming from the server without the expense of each response being signed.

DO-IRP servers authenticate clients via a challenge-response process. That is, after receiving a client's request, the server issues a challenge to the client if authentication is necessary. To be authenticated as the administrator, the client has to return a challenge-response, a message that demonstrates possession of the administrator's secret. The secret may be the private key or the secret key of the administrator. This challenge-response allows the server to authenticate the client as the administrator. Upon successful authentication, the server will fulfill the client's request if the administrator is given sufficient permission.

For example, suppose a client sends a request to the server to add a new element. The server will issue a challenge to the client in order to authenticate the client as one of the administrators. If the client possesses the private key of the administrator, it can use it to sign the server's challenge and return the signature as part of the challenge-response. The server will validate the signature in order to authenticate the client. The client will be notified if the validation fails. Otherwise, the server will further check if the administrator has the permission to add the element. If so, the server will add the element and report success to the client. Otherwise, a permission-denied message will be returned.

The following diagram shows a typical authentication process in terms of the messages exchanged between the client and the DO-IRP server.

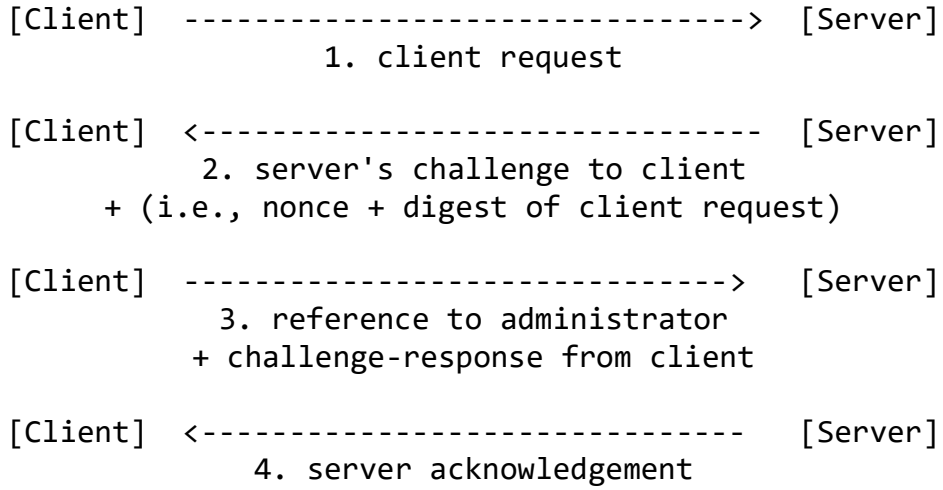

Figure 5.2-a: DO-IRP authentication process

In Figure 5.2-a, the client sends an administration request to the server. The server decides that client authentication is required and issues a challenge to the client. The client identifies itself as an administrator (with the administrator reference) and returns the challenge-response to the server. The server authenticates the client as the administrator based on the challenge-response. It also checks to see if the administrator is authorized for the administration request. If so, the server will fulfill the request and acknowledge the client.

DO-IRP servers must authenticate the client before fulfilling any request that requires administrator privilege. The exact authentication process varies depending on whether public key or secret key is used by the administrator. It also depends on whether the identifier used to store the administrator's key is managed by the same server or not.

When public key is used, the challenge-response from the client contains its digital signature over the server's challenge. The server can authenticate the client by verifying the digital signature based on the administrator's public key, whether that public key is available at the server itself or needs to be resolved at another DO-IRP service.

If secret key is used, the challenge-response from the client carries the Message Authentication Code (MAC) generated using the secret key. How the challenge-response is verified depends on the administrator identifier record where the identity information is managed. For purposes of describing this process, the reference to administrator in Figure 5.2-a will be called a key-reference. It refers to an identifier element that contains the key used by the administrator. If the key-reference is managed by the same server (e.g., an element assigned to the same identifier), the server may use the key directly to do the authentication. If the key-reference is managed by some

other server (whether or not within the same DO-IRP service), the server will have to send a verification-request to this other server, call it the key-server, in order to authenticate the client. The verification-request to the key-server carries both the server's challenge and the client's challenge-response. The key-server will verify the challenge-response on behalf of the requesting server and return the result in the verification-response along with a signature of it made using the key-server's private key. Figure 5.2-b shows the control flow of the authentication process where the key-reference refers to an element that contains the administrator's secret key and the key-server is some other server.

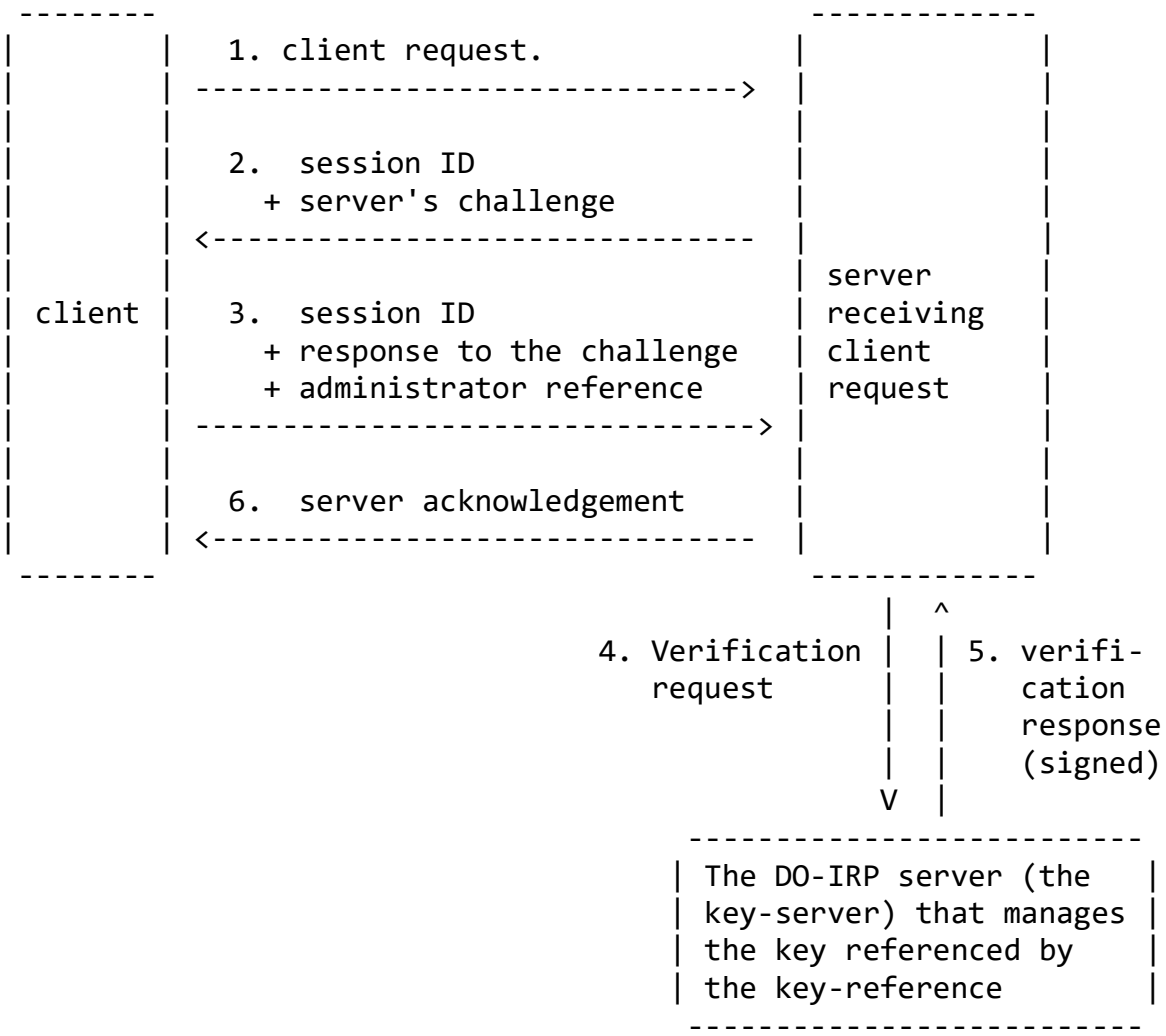

Figure 5.2-b: Authentication process requiring verification from a second DO-IRP server

Secret key based authentication via a second DO-IRP server, i.e., the key server, provides a convenient way to share a common secret key (e.g., pass phrase) among identifiers managed by different servers. However, it should not be used to manage highly sensitive identifiers or data. The authentication process itself is expensive and relies on a third party, i.e., the key-server, for

proper operation. Additionally, the secret key itself is subject to dictionary attack since the key-server cannot determine whether the verification-request comes from a legitimate DO-IRP server. A DO-IRP service may set its local policy so that secret-key-based authentication can only be carried out if the server (receiving the client request) is also the key-server.

The DO-IRP authentication process is designed so that secret keys are never sent over a network during authentication. Instead, the client sends a MAC of some server-provided data using the client's secret key; the authenticating server, if necessary, forwards the same MAC and the server-provided data to a key-server which can calculate the same MAC to verify the authentication. Although never sending the secret key over a network is advantageous, this approach has the disadvantage that the client's secret key must be available in the clear in storage of the key-server. This is clearly undesirable and a significant reason to prefer the use of public/private keypairs for DO-IRP authentication.

6 CLIENT-SERVER PROCESS

DO-IRP defines a client-server process in which client software submits requests via a network to DO-IRP servers. Each request describes the operation to be performed on the server and the target of the operation. The server will process the request and return a message indicating the result of the operation. The next part of this document specifies the process to be used by client software in accessing a server for identifier resolution and/or administration.

Some key aspects of the client-server process include:

- Support for both identifier resolution and administration.
- Authentication of any server response based on the server's digital signature.
- A server authenticates clients for requests that only an administrator can perform (see section 4.3.1 for details) via the DO-IRP authentication process. The DO-IRP authentication process is a challenge-response process 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 enables data integrity and confidentiality.
- The process can be extended to support new operations. Controls can be used to extend the existing operations. The process is defined to allow future backward compatibility with the help of versioning.
- Distributed service architecture. Supports service referral among different service components, thus enabling one server to suggest a client ask another server.

- Identifiers and their data types are based on the Unicode (ISO-10646) character set. UTF-8 is the mandated encoding under the DO-IRP.

While there are certain aspects of DO-IRP that have changed significantly from earlier versions from which it was derived, most of the basic aspects of the protocol remain basically unchanged. 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 6.2.1.1).

6.1 PROTOCOL ELEMENTS & CONVENTIONS

The following conventions are followed by the DO-IRP protocol to ensure interoperability among different implementations.

6.1.1 DATA TRANSMISSION ORDER

The order of transmission of data packets follows the network byte order (also called Big-Endian). 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.

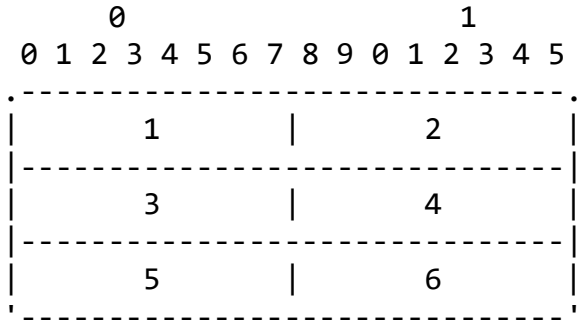

Figure 6.1.1-a: Octet transmission order

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).

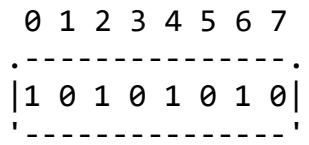

Figure 6.1.1-b: Representation of decimal 170

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.

6.1.2 COMMUNICATING MESSAGES

The DO-IRP protocol is designed so that messages may be communicated either as separate data-grams over UDP or as a continuous byte stream via a TCP connection. It is noted here that a message can be construed as a sequence of bits to be communicated from a source to a destination, and, among other things, may constitute a digital object in its own right.

Although use of UDP or TCP is included at present, other procedures may be included in the future. The recommended port number for both UDP and TCP is 2641.

6.1.2.1 UDP USAGE

This transport method is recommended as an option that, if implemented, should normally be turned off by default.

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 6.2.1.6).

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 should be no longer than 3 seconds and can be as low as 0.5 seconds depending on location and the quality of service.

6.1.2.2 TCP USAGE

Messages sent using DO-IRP 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. The use of TCP for transport of DO-IRP messages is recommended along with the follow connection management policies:

- 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.

6.1.2.3 HTTP AND HTTPS TUNNEL USAGE

DO-IRP requests and responses can be embedded into HTTP requests and responses. This allows encapsulating a DO-IRP exchange into an HTTP "tunnel". There is no difference in the message sent as octets; this section describes how those messages are embedded into HTTP messages. For details about HTTP, see RFC 7230 [8] and RFC 7231 [9].

For HTTPS, the public key of the server's TLS certificate should match the server's public key as given in the HS_SITE element. This is in addition to the standard DO-IRP mechanisms used for authenticating the server, which are still used in the same way.

The HTTP request should always use method POST, and should include the headers

```
Accept: application/x-hdl-message
Content-Type: application/x-hdl-message
```

The bytes of the DO-IRP request form the message body of the HTTP request. The request target URL of the HTTP request and other headers do not have special meaning for the DO-IRP server, although they may affect the behavior of HTTP caches or proxies as standard for HTTP.

The response should include the header

```
Content-Type: application/x-hdl-message
```

The bytes of the DO-IRP response form the message body of the HTTP response. As long as the response contains a DO-IRP message as indicated by the Content-Type header, the status code and other response headers should be ignored by a DO-IRP client. These may have meaning to intervening HTTP caches and proxies in the usual way, however. It is acceptable for a DO-IRP server to use HTTP status code 200 (OK) for all responses.

The HTTP(S) URL used for the connection, which manifests in the Host header and request target of the HTTP request, is not relevant for the DO-IRP message flow. However, it may be significant for the transport of the HTTP request. To this end, a DO-IRP client may use attributes of an HS_SITE to determine what to use. If an HS_SITE has an attribute "domain", the value of that attribute can be used in the Host header of the HTTP request, along with the port actually used; if an HS_SITE has an attribute "path", the value of that attribute can be used as the initial part of the request target path. (Existing client implementations append the encoded identifier of the request to the path; this does not affect existing server implementations.) For an HS_SITE with multiple servers, the attributes can specify different values "domain.<ServerID>", "path.<ServerID>" using the numeric <ServerID> of each of the servers.

6.1.3 CHARACTER CASE

Identifiers are character strings based on the ISO-10646 character set and must be encoded in UTF-8. By default, characters are treated as case-sensitive under the DO-IRP protocol. A DO-IRP service, however, may be implemented in such a way that ASCII characters are processed case-insensitively.

6.1.4 STANDARD STRING TYPE: UTF8-STRING

Identifiers are transmitted as UTF8-Strings under DO-IRP. 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.

6.2 COMMON ELEMENTS

Each message exchanged under the system protocol consists of four sections (see Figure 6.2). 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 is primarily used to help deliver the message. Only part of the Message Envelope is protected by the digital signature in the Message Credential, namely version information (as it may indicate how the message is to be interpreted) and request and session ids (to prevent replay attacks); see section 6.2.4 below.

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. The Message Header is protected by the digital signature in the Message Credential; see section 6.2.4.

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 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.

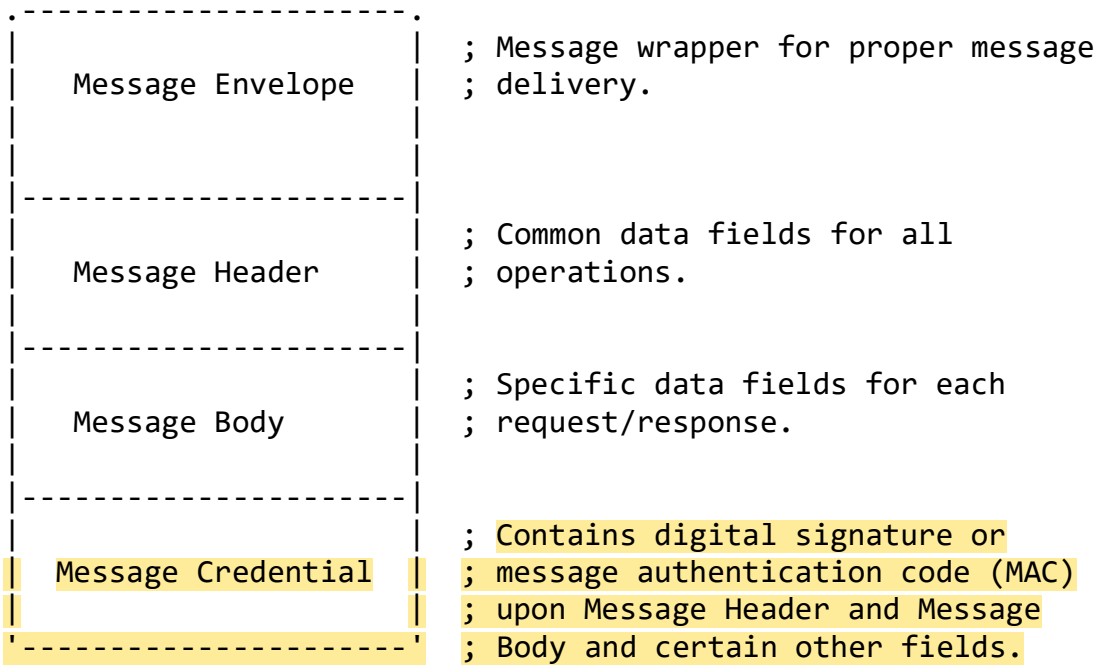

Figure 6.2: DO-IRP Message Sections

6.2.1 MESSAGE ENVELOPE

Each message begins with a Message Envelope under the DO-IRP 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 nine fields:

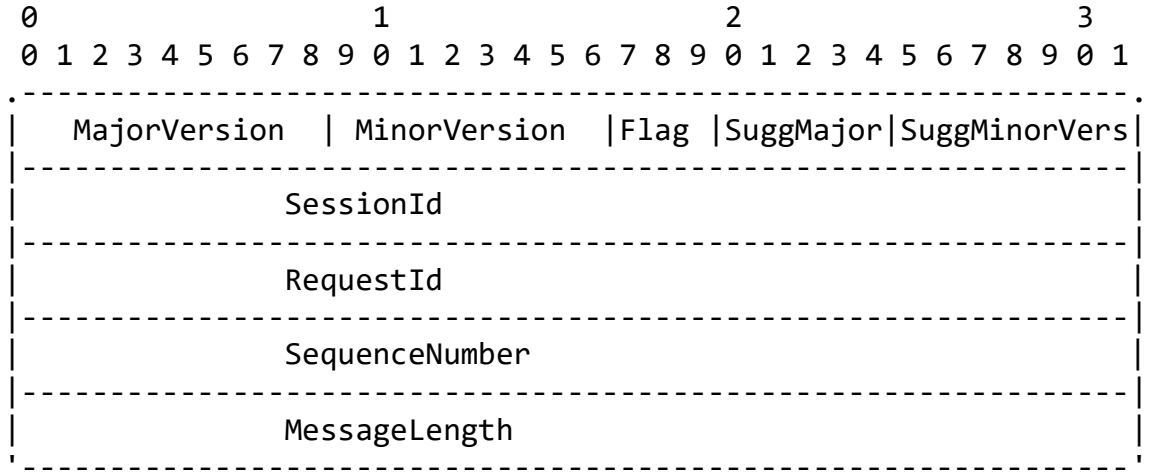

Figure 6.2.1: Message Envelope Fields

6.2.1.1 <MajorVersion> and <MinorVersion>

The <MajorVersion> and <MinorVersion> are used to identify the version of the DO-IRP protocol. Each of them is defined as a one- byte unsigned integer. This specification defines the protocol version whose <MajorVersion> is 3 and <MinorVersion> is 0.

<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.

6.2.1.2 <SuggMajor> and <SuggMinorVers>

The <SuggMajor> and <SuggMinorVers> represent a major version and minor version suggested by the client. This allows the client and server to negotiate a protocol version. Potentially the client obtained site information for the server indicating a lower version number than the highest known to the client. In such a case, the client can send the lower version in <MajorVersion> and <MinorVersion>, and the higher version in <SuggMajor> and <SuggMinorVersion>. If the server actually is able to respond in a higher version, it can.

6.2.1.3 <Flag>

The <Flag> consists of three bits defined as follows:

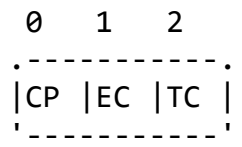

Figure 6.2.1.3: <Flag> Bits

Bit 0 is the CP (ComPressed) flag and is currently not used. It should be set to 0.

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 6.3.

6.2.1.4 <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 an authenticated and/or encrypted communication channel. 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 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 7.10.

6.2.1.5 <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>.

6.2.1.6 <SequenceNumber>

Messages under the DO-IRP 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.

6.2.1.7 <MessageLength>

A 4-byte unsigned integer that specifies the total number of octets of any message, excluding those in the Message Envelope, but including the Message Header, Message Body, and (when present) Message Credential. For a truncated message, each individual truncated message sets <MessageLength> to the full length of the original untruncated message. The length of any single message exchanged under the DO-IRP protocol is limited by the range of a 4-byte unsigned integer.

6.2.2 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.

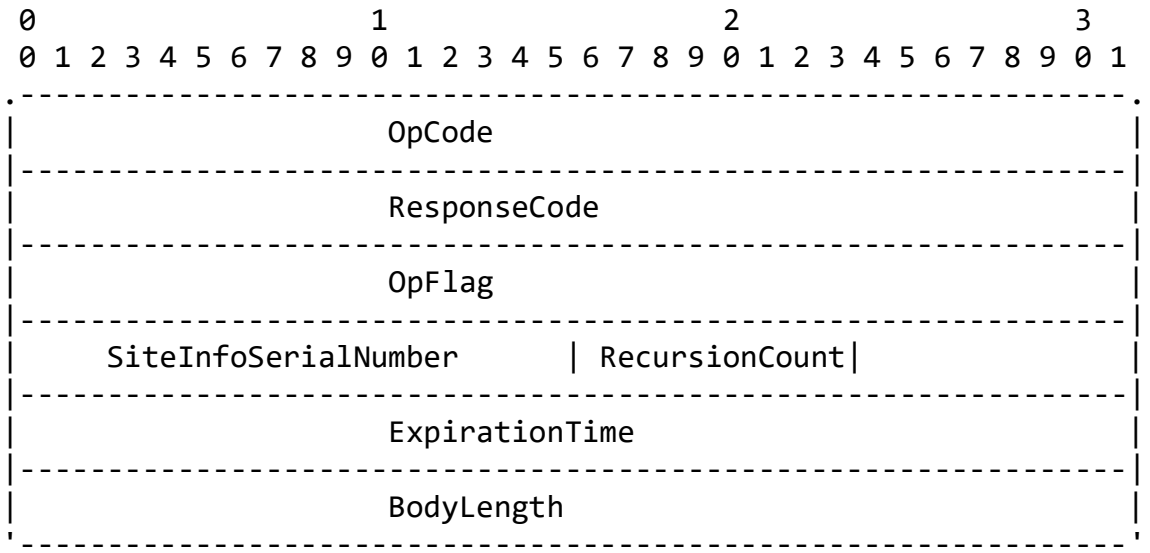

Figure 6.2.2: Message Header Fields

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.

6.2.2.1 <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.

Table 6.2.2.1: Required Operation Codes

Op_Code	Symbolic Name	Remark
0	OC_RESERVED	Reserved
1	OC_RESOLUTION	Identifier query
2	OC_GET_SITEINFO	Get HS_SITE element
100	OC_CREATE_ID	Create new identifier
101	OC_DELETE_ID	Delete existing identifier
102	OC_ADD_ELEMENT	Add element(s)
103	OC_REMOVE_ELEMENT	Remove element(s)
104	OC_MODIFY_ELEMENT	Modify element(s)
105	OC_LIST_IDS	List identifiers
106	OC_LIST_DERIVED_PREFIXES	List derived prefixes
200	OC_CHALLENGE_RESPONSE	Response to challenge
201	OC_VERIFY_RESPONSE	Verify challenge response
300	OC_HOME_PREFIX	Home prefix (instruct server to consider itself responsible for the prefix)
301	OC_UNHOME_PREFIX	Unhome prefix
302	OC_LIST_HOMED_PREFIXES	List homed prefixes
303 : 399	{ Reserved for server administration}	
400	OC_SESSION_SETUP	Session setup request
401	OC_SESSION_TERMINATE	Session termination request

A detailed description of each of these <OpCode>s can be found in section 7 of this document. In general, clients use the <OpCode> to tell the server what kind of 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.

6.2.2.2 <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 DO-IRP protocol is defined in the following table. Each response code is given a symbolic name that is used throughout this document for easy reference.

Table 6.2.2.2: Response Codes

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
7	RC_SERVER_BACKUP	Server storage is temporarily read-only as for backup
100	RC_ID_NOT_FOUND	Identifier not found
101	RC_ID_ALREADY_EXIST	Identifier already exists
102	RC_INVALID_ID	Encoding (or syntax) error
200	RC_ELEMENT_NOT_FOUND	Element not found
201	RC_ELEMENT ALREADY_EXIST	Element already exists
202	RC_ELEMENT_INVALID	Invalid Element
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_PREFIX_REFERRAL	Prefix referral
400	RC_INVALID_ADMIN	Not an admin for operation
401	RC_ACCESS_DENIED	Insufficient permissions
402	RC_AUTHEN_NEEDED	Authentication required
403	RC_AUTHEN_FAILED	Failed to authenticate
404	RC_INVALID_CREDENTIAL	Invalid message credential
405	RC_AUTHEN_TIMEOUT	Authentication timed out
406	RC_UNABLE_TO_AUTHEN	Unexpected error authenticating
500	RC_SESSION_TIMEOUT	Session expired
501	RC_SESSION_FAILED	Unable to establish session
502	RC_SESSION_KEY_INVALID	Invalid session key or session Authentication failure
505	RC_SESSION_MSG_REJECTED	Potential session replay attack

Detailed descriptions of these <ResponseCode>s can be found in section 7 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.

6.2.2.3 <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.

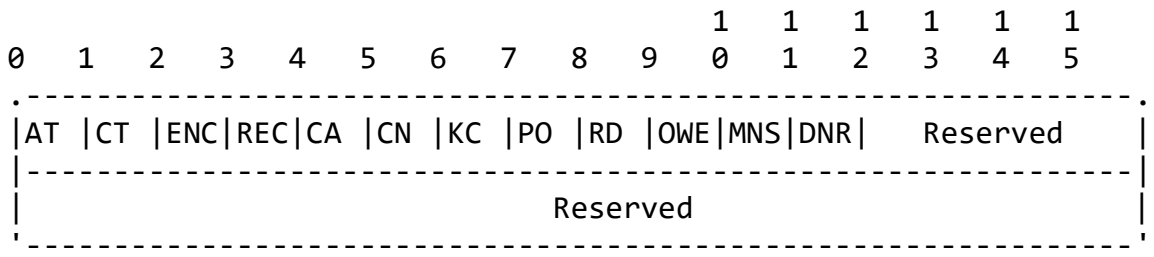

Figure 6.2.2.3: Option flags in the <OpFlag> field

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 by including a Message Credential (see section 6.2.4). 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 DO-IRP server. The server may honor the request by forwarding the request to the appropriate 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. This bit is reserved.

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 sent as multiple messages. Unlike truncation at the level of the Message Envelope, which breaks down a single message into multiple for reasons of transport such as UDP message size, this indicates a request or response that is deliberately sent as multiple messages. This is used for example for the responses to requests to list identifiers.

KC
Keep Connection bit. A message with the KC bit set requests that the message recipient keep the TCP connection open (after the response is sent back). This allows the same TCP connection to be used for multiple 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 elements that have the PUB_READ permission. A request with PO bit set to zero asks for all the elements regardless of their read permission. If any of the elements require ADMIN_READ permission, the server must authenticate the client as an 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 in section 6.2.3.

OWE
Overwrite when exists. When this bit is set on a request with OpCode OC_CREATE_ID or OC_ADD_ELEMENT, elements which already exist will be overwritten by the elements presented in the request. Otherwise the request will result in an error.

MNS
Mint new suffix. When this bit is set on a request to create an identifier, the "identifier" field of the request will be considered the initial portion of the eventual identifier, which will be extended by the server to a new complete identifier. The initial portion should contain the slash.

DNR
Do not refer. When this bit is set, a request which would normally result in a service referral or prefix referral response, will instead be applied on the server where the request is

received. This would generally be used on an administrative request to override an existing configured referral.

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.

6.2.2.4 <SiteInfoSerialNumber>

The <SiteInfoSerialNumber> is a two-byte unsigned integer. The <SiteInfoSerialNumber> in a request refers to the <SerialNumber> of the HS_SITE element 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. 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.

6.2.2.5 <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.

6.2.2.6 <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.

6.2.2.7 <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.

6.2.3 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 7 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 algorithm. If the octet is set to 2, SHA-1 algorithm is used. If the octet is set to 3, the SHA-256 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. For SHA-256, the length is 32 octets.

The Message Body (along with Message Header and Message Credential) 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.

6.2.4 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 server must sign its response by including a Message Credential if the request has its CT bit (in <OpFlag>) set (to 1).

Each Message Credential consists of the following fields:

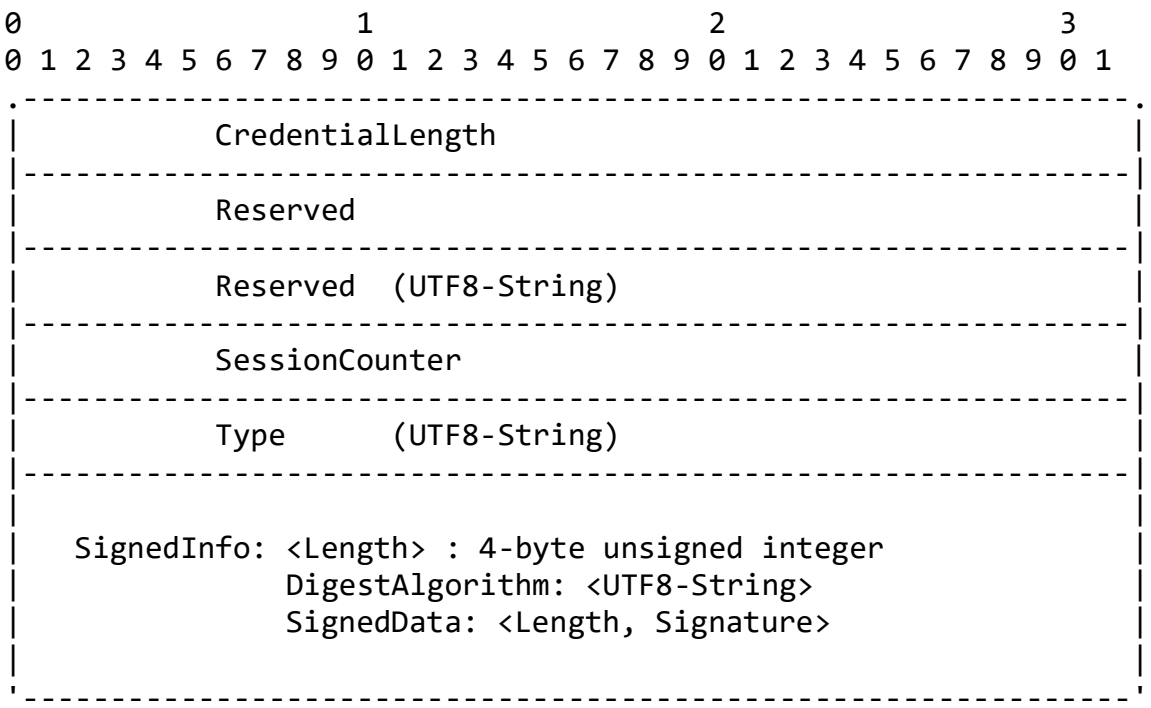

Figure 6.2.4: Message Credential Fields

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.

<Reserved>
Eight octets that must be set to zero. The second group of four octets may be used to represent the length of a UTF8-String in a future version.

<SessionCounter>
4-byte unsigned integer. For messages in a session, client and server may keep track of session counters sent by the other party, and are expected to reject messages which reuse session counters in order to prevent replay attacks. Using an incrementing counter is an appropriate choice.

<Type>
A UTF8-String that indicates the type of payload in the <SignedInfo> field (described below). It may contain HS_MAC if <SignedInfo> contains the Message Authentication Code (MAC), or

HS_SIGNED if <SignedInfo> contains the digital signature generated from a public/private key pair.

<SignedInfo>
Consists of the following fields:

```
<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 MAC or digital signature. For example, the value "SHA-1" indicates that the SHA-1 algorithm is used to generate the message digest for the signature. Implementations are expected to support SHA-1 and SHA-256 for digital signatures and MACs, and HMAC-SHA-1 and HMAC-SHA-256 for MACs.

<SignedData>
Consists of the following fields:

```
<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 part of the Message Envelope and all of the Message Header and Message Body. The syntax and semantics of the signature depend on the <Type> field and the <DigestAlgorithm> field.

The data to be either hashed to a MAC or signed is as follows. The first twelve octets are from the Message Envelope:

```
<MajorVersion><MinorVersion><SuggMajor><SuggMinorVers> (one octet each)
<SessionId> (four octets)
<RequestId> (four octets)
```

The next four octets are the <SessionCounter> from the Message Credential. These sixteen octets are followed by the complete Message Header and Message Body.

If the <Type> field is "HS_SIGNED", the signature will be bytes of a signature of this data by the public key of 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 the case of a DSA public key, the signature will be the ASN.1 octet string representation of the parameters R and S as described in RFC 3370. In the case of an RSA public key, the signature will be an RSA PKCS#1 v1.5 signature [10]. In either case the signature uses the hash algorithm specified in <DigestAlgorithm>.

If the <Type> field is "HS_MAC", the Message Credential contains the message authentication code (MAC) generated using a pre-established session key. The <SIGNATURE > field must contain the MAC. The MAC is the result of the one-way hash of the data using the algorithm specified in <DigestAlgorithm>. In the case of SHA-1 or SHA-256, this is the hash of the concatenation of the session key, the data, and the session key again. HMAC-SHA1 and HMAC-SHA256 are preferred.

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 Envelope, 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.

6.3 MESSAGE TRANSPORT

A large message may be fragmented into multiple successive packets for communication. For example, to fit the size limit of a UDP packet, the message issuer must first break a large message into multiple successive UDP packets. The message recipient must reassemble the message from these successive packets before further processing. Message fragmentation must be carried out over the entire message except the Message Envelope. A new Message Envelope has to be inserted in front of each successive packet before its transmission.

For example, a large message that consists of

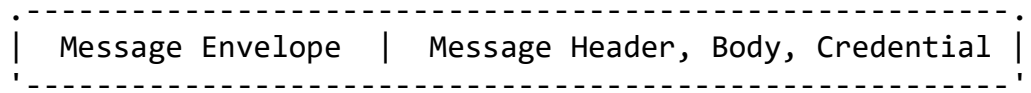

Figure 6.3-a: Large message

may be fragmented into:

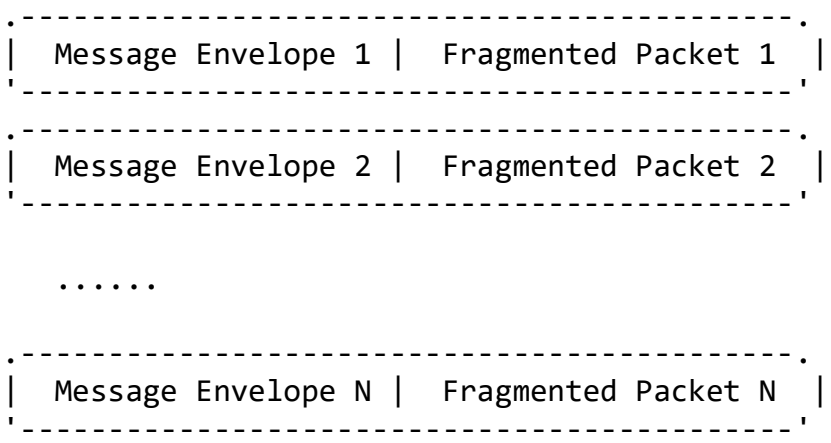

Figure 6.3-b: Fragmented message

where the "Fragmented Packet 1", "Fragmented Packet 2", ..., and "Fragmented Packet N" result from the original Message Header, the Message Body and the Message Credential. Each "Message Envelope <#>" (inserted before each fragmented packet) must set its TC flag to 1 and maintain the proper sequence count (in the <SequenceNumber>). Each "Message Envelope <#>" retains its <MessageLength> to reflect the size of the full unfragmented original message. The recipient of these fragmented packets can reassemble the message by concatenating these packets based on their <SequenceNumber>.

7 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.

7.1 SELECTING THE RESPONSIBLE SERVER

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

a. Select a preferred service site within the service's set of sites.
   Each service site is defined in terms of an HS_SITE element. Clients must select a primary service site for any administrative operations. The HS_SITE element may contain a description or other attributes (under the <AttributeList>) to help the selection. In most cases clients can

choose a service site randomly, restricting only to primary sites for administration, as all identifier records are replicated across all service sites in the service component.

b. Locate the responsible server within the service site.
   Most commonly service sites have a single server; when there are multiple servers, the responsible server can be located as follows: Convert every ASCII character in the identifier to its upper case. Calculate the MD5 hash of the converted string according to the <HashOption> given in the HS_SITE element. 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 element. The result is the sequence number (starting with zero) of the <ServerRecord> listed in the HS_SITE element. 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 server for the given identifier.

Implementers may also wish to explore other techniques for such distribution of identifier records.

7.2 QUERY OPERATION

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

7.2.1 QUERY REQUEST

The Message Header of any query request must set its <OpCode> to OC_RESOLUTION (defined in section 6.2.2.1) and <ResponseCode> to 0.

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

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

where

<Identifier>
A UTF8-String (as defined in section 6.1.4) that specifies the identifier 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 an index and refers to an element to be retrieved. The client sets the first integer to zero (followed by an empty array) to ask for all the elements 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 type. This tells the server to return all elements whose type is listed in the list. If a UTF8-String ends with the "." (0x2E) character, the server must return all elements whose type is under the type hierarchy specified in the UTF8-String. This means that the type is either exactly the supplied UTF8-String omitting the terminal "." character, or starts with the supplied UTF8-String including the "." character. The <TypeList> may contain no UTF8-String if the first integer is 0. In this case, the server must return all elements regardless of their 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 elements 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 elements with PUBLIC_READ permission and all the elements with ADMIN_READ permission. If the PO flag is set, the server should never authenticate the client or return elements without PUBLIC_READ permission; if the query requests a specific element via the index and the element does not have PUBLIC_READ permission, and the request has its PO flag set, then the server must return RC_ELEMENT_NOT_FOUND.

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

When both <IndexList> and <TypeList> fields are non-empty, the server should return the union of all elements whose indexes are listed in the <IndexList>, together with all elements whose data types are listed in the <TypeList>.

7.2.2 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> ]
                                                <Identifier>
                                                <ElementList>
```

where

<RequestDigest>
Optional field as defined in section 6.2.3.

<Identifier>
A UTF8-String that specifies the identifier queried by the client.

<ElementList>
A 4-byte unsigned integer followed by a list of elements. The integer specifies the number of elements in the list. The encoding of each element follows the specification given earlier (see section 4.1). The integer is set to zero if there is no element that satisfies the query.

7.2.3 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 7.3 of this document.

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

Note that a server should NOT return an RC_ID_NOT_FOUND message if the server is not responsible for the identifier being queried. It is possible that the queried identifier exists but is managed by another DO-IRP server (under some other DO-IRP service). When this happens, the server should either send a service referral (see section 7.4) or simply return an error message with <ResponseCode> set to RC_SERVER_NOT_RESP. See the requests for OC_HOME_PREFIX, OC_UNHOME_PREFIX, and OC_LIST_HOMED_PREFIXES for administering which prefixes for which a server considers itself responsible.

If the identifier exists, but there are no elements which match the client's query, then the server must return an RC_ELEMENT_NOT_FOUND message.

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_ID_NOT_FOUND, an RC_SERVER_BUSY message also has an empty message body.

Servers should return an RC_ACCESS_DENIED message if the request does not have its PO flag set and asks for a specific element (via the index) that has neither PUBLIC_READ nor ADMIN_READ permission.

A server may ask its client to authenticate itself as the administrator during the resolution. This happens if the request does not have its PO flag set and any element in the query has ADMIN_READ permission, but no PUBLIC_READ permission. Details of client authentication are described in section 7.5

7.3 ERROR RESPONSE FROM SERVER

A 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 6.2.2.2 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 6.2.3.

<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 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 an element that contributed to the error. An example would be a server that is asked to add three elements, with indexes 1, 2, and 3, and elements with indexes of 1 and 2 already in existence. In this case, the server could return an error message with <ResponseCode> set to RC_ELEMENT_ALREADY_EXIST and add index 1 and 2 to the <IndexList>. Note that the server is not obligated to return the complete list of indexes that may have caused the error.

7.4 SERVICE REFERRAL AND PREFIX REFERRAL

A server may receive requests for identifiers that are managed by some other server or service. When this happens, the server has the option to either return a referral message that directs the client to the proper DO-IRP service, or simply return an error message with <ResponseCode> set to RC_SERVER_NOT_RESP. Service referral also happens when ownership of identifiers moves from one service to another. It may also be used by any DO-IRP service to indicate that some or all of its service may be performed by another DO-IRP service. Prefix referral in particular is used to indicate that a given prefix or prefixes may be resolvable and administered at a different service.

DO-IRP servers use the HS_SITE.PREFIX and HS_SERV.PREFIX element types in order to determine whether to send an RC_PREFIX_REFERRAL request. When a DO-IRP server is asked to resolve a prefix identifier 0.NA/<X>.<Y>, and the server does not have any information about 0.NA/<X>.<Y>

but does have 0.NA/<X> and the record for 0.NA/<X> includes HS_SITE.PREFIX and/or HS_SERV.PREFIX elements, those elements are returned to the client as a prefix referral response.

Since this behavior is purely in the server, server configuration can override this. This specification does not define under what conditions servers will send other prefix referral or service referral responses.

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 or RC_PREFIX_REFERRAL.

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

```
<Message Body of Service Referral> ::= [ <RequestDigest> ]
                                         <ReferralIdentifier>
                                       [ <ElementList> ]
```

where

<RequestDigest>
Optional field as defined in section 6.2.3.

<ReferralIdentifier>
A UTF8-String that identifies the identifier (e.g., a service identifier) that maintains the referral information (i.e., the service information of the DO-IRP service to which it corresponds).

<ElementList>
An optional field that must be empty if the <ReferralIdentifier> is provided. When not empty, it consists of a 4-byte unsigned integer, followed by a list of HS_SITE or HS_SERV elements, or, in the case of a prefix referral, HS_SITE.PREFIX or HS_SERV.PREFIX elements. The integer specifies the number of elements in the list.

The <ReferralIdentifier> may contain an empty UTF8-String if the elements in the <ElementList> are not maintained by any identifier.

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.

If the client's request is sent with the DNR (do not refer) flag set, then the server should not respond with a referral response. This can be used with administrative requests, for example to create an identifier directly on a server which normally would be referred to a different server. This can be used to override a referral for one or more specific identifiers.

7.5 CLIENT AUTHENTICATION

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

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 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 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_INVALID_ADMIN.

Authentication is not about encryption (although encryption may be used in the process) and is accomplished by use of a Message Authentication Code or digital signature verification.

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

7.5.1 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. If the message is not already part of an established session, 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 6.2.2.3) 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 6.2.3.

<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 16 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.

7.5.2 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>
                                          <KeyIdentifier>
                                          <KeyIndex>
                                          <ChallengeResponse>
```

where

<AuthenticationType>
A UTF8-String that identifies the type of authentication key used by the client. 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.

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

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

<ChallengeResponse>
A 4-byte unsigned integer indicating the length of the following byte array, which contains either the Message Authentication Code (MAC) or the digital signature over the challenge from the server. The <ServerChallenge> used to produce the MAC or signature is the concatenation of the bytes of the <Nonce> from the CHALLENGE message, excluding the initial 4 length bytes, followed by the <RequestDigest> from the CHALLENGE message, excluding the initial <RequestDigestIdentifier> byte.

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 in the <ChallengeResponse> following the 4-byte length is set to 0x02, the MAC is generated using

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

where

<SecretKey> is the administrator's secret key referenced by the <KeyIdentifier> and <KeyIndex>.

A <ChallengeResponse> with its first octet (following the 4-byte length) set to 0x03 uses SHA-256 in a similar way.

A more secure approach is to use a Hash-based Message Authentication Code (HMAC) for the <ChallengeResponse>. The HMAC can be generated using the <SecretKey> and <ServerChallenge>. A <ChallengeResponse> with its first octet (following the 4-byte length) set to 0x12 indicates that the HMAC is generated using the SHA-1 algorithm, and 0x13 indicates HMAC using the SHA-256 algorithm.

If the first octet (following the 4-byte length) of the <ChallengeResponse> is set to 0x22, PBKDF2-HMAC-SHA1 is used. A derived key is produced from the secret key using the PBKDF2-HMAC-SHA1 algorithm with a specified salt (e.g., 16 random octets), number of iterations (e.g. 10000), and desired derived key length (e.g., 160 bits). A MAC is then produced using the derived key and the <ServerChallenge> via the HMAC-SHA1 algorithm. The <ChallengeResponse> is then encoded as follows:

```
<ChallengeResponse using PBKDF2-HMAC-SHA1>::= 4-byte length
                                                0x22
                                               <Salt>
                                               <Iterations>
                                               <DerivedKeyLength><MAC>
```

where

<Salt>
A 4-byte unsigned integer indicating the salt length follows by the octets of the salt.

<Iterations>
A 4-byte unsigned integer.

<DerivedKeyLength>
A 4-byte unsigned integer, the number of bits desired for the derived key.

<MAC>
The MAC produced using HMAC-SHA1 on the derived key and the <ServerChallenge>.

If the <AuthenticationType> is "HS_PUBKEY", the <ChallengeResponse> contains a 4-byte length followed by the digital signature over <ServerChallenge>. 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 4-byte length of the signature itself, followed by the signature over the server's challenge. In the case of a DSA [11] public key, the signature will be the ASN.1 octet string representation of the parameters R and S as described in RFC 3370 [12]. In the case of an RSA public key, the signature will be a RSA PKCS#1 v1.5 signature. The <KeyIdentifier> and <KeyIndex> refers to the administrator's public key that can be used to verify the signature.

Administrators are defined in terms of HS_ADMIN elements assigned to the identifier. Each HS_ADMIN element 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 element refers to the one 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 an element of type HS_VLIST. An HS_VLIST element defines an administrator group via a list of element references, each of which refers to the authentication key of an administrator.

For identifiers with multiple HS_ADMIN elements, the server will have to check each of those with sufficient privileges to see if its <AdminRef> field matches the <KeyIdentifier> and <KeyIndex>. If no match is found, but there are administrator groups defined, the server must check if the <KeyIdentifier> 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 an HS_ADMIN element has an <AdminRef> with an index of 0, it is considered to match an authentication using any <KeyIndex>. Note that such an <AdminRef> is always a direct reference to

some element of type HS_PUBKEY or HS_SECKEY; elements of type HS_VLIST cannot be referenced using an index of 0.

If an authentication uses a <KeyIndex> of 0, it might refer to any element of the record of the <KeyIdentifier>. In the case of an HS_SECKEY authentication, the authentication can only match an HS_ADMIN where the <AdminRef> has an index of 0 (because the server can not in general know which index has the correct HS_SECKEY). In the case of an HS_PUBKEY authentication, the authentication will match an HS_ADMIN where the <AdminRef> has an index of 0, but will also match an HS_ADMIN where the <AdminRef> has the index of the HS_PUBKEY element on the <KeyIdentifier> which actually validates the signature (in this case, the server can check each public key and so it knows which one is the matching index).

If the <KeyIdentifier> and <KeyIndex> are not referenced by any of the HS_ADMIN elements, or the administrator group that has sufficient privileges, the server will return an error message with <ResponseCode> set to RC_INVALID_ADMIN. 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 <KeyIdentifier> and <KeyIndex>. The public key can be used to verify the <ChallengeResponse> against the server's <ServerChallenge>. If the <ChallengeResponse> matches the <ServerChallenge>, 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 may in general have to send a verification request to the verification server; that is, the server that manages the referenced <KeyIdentifier>. 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 server.

7.5.3 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> ::= <KeyIdentifier>
                                           <KeyIndex>
                                           <ServerChallengeNonce>
                                           <ServerChallengeReqDigest>
                                           <ChallengeResponse>
```

where

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

<KeyIndex>
A 4-byte unsigned integer that is the index of the element that holds the secret key of the administrator (or 0 to indicate that which element is unspecified).

<ServerChallengeNonce>
The <Nonce> from the server's CHALLENGE, including an initial 4-byte length, as described in section 7.5.1.

<ServerChallengeReqDigest>
The <RequestDigest> from the server's CHALLENGE, excluding the <RequestDigestIdentifier> octet, but including an initial 4-byte length.

<ChallengeResponse>
The <ChallengeResponse> from the client in response to the server's <Challenge>, as defined in section 7.5.2.

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.

7.5.4 CHALLENGE-RESPONSE VERIFICATION-RESPONSE

The Verification-Response tells the requesting 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 <RequestDigest> 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, as defined in section 6.2.3.

<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 <KeyIdentifier> does not exist, or the <KeyIdentifier> and <KeyIndex> refer to an invalid element). When this happens, the server that performs the client authentication should relay the same error message back to the client.

If the verification server supports a lower version number than the server performing the client authentication, it may occur to the server software that the client uses a secret key MAC algorithm not understood by the verification server. In this circumstance the server performing the client authentication may return a second CHALLENGE response with a suggested protocol version (<SuggMajor> and <SuggMinorVers>) matching the verification server. The client can then, at its option, send a second CHALLENGE_RESPONSE request using a MAC algorithm (such as SHA-1 [7]) appropriate to the lower protocol version.

7.6 GET SITE INFO

A DO-IRP client can request that a DO-IRP server return its site information by sending a GET_SITEINFO request. This allows a client to learn about the site if the client only knows how to connect to the server (perhaps only one of multiple servers in the site) over the network. The Message Header of the GET_SITEINFO request must set its <OpCode> to OC_GET_SITEINFO.

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

```
<Message Body of GET_SITEINFO Request> ::= <Ignored_Identifier>
```

where

<Ignored_Identifier>
A UTF8-String, which may be empty; ignored by the server, but included to maintain the pattern that all requests begin with a UTF8-String identifier.

The Message Body of a successful GET_SITEINFO response is defined as follows:

```
<Message Body of GET_SITEINFO Response> ::= <SiteInfo>
```

where

<SiteInfo>
The site info of the site the server is part of, encoded as for an HS_SITE record element.

7.7 ADMINISTRATION

The DO-IRP protocol supports a set of administration functions that include adding, deleting, and modifying identifiers, or elements thereof, via their associated records. Before fulfilling any administration request, the server must authenticate the client as the administrator that is authorized for the administrative operation. Administration can only be carried out by a primary server.

7.7.1 ADD ELEMENT(S)

Clients add elements to existing identifier records by sending ADD_ELEMENT requests to the responsible server. The Message Header of the ADD_ELEMENT request must set its <OpCode> to OC_ADD_ELEMENT.

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

```
<Message Body of ADD_ELEMENT Request> ::=  <Identifier>
                                           <ElementList>
```

where

<Identifier>
A UTF8-String that specifies the identifier.

<ElementList>
A 4-byte unsigned integer followed by a list of elements. The integer indicates the number of elements in the list.

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

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

If the request has the OWE (overwrite-when-exists) flag set, then the server will modify existing elements to match the request, rather than returning RC_ELEMENT_ALREADY_EXISTS.

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

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

7.7.2 REMOVE ELEMENT(S)

Clients remove existing elements by sending REMOVE_ELEMENT requests to the responsible server. The Message Header of the REMOVE_ELEMENT request must set its <OpCode> to OC_REMOVE_ELEMENT.

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

```
<Message Body of REMOVE_ELEMENT Request> ::=  <Identifier>
                                              <IndexList>
```

where

<Identifier>
A UTF8-String that specifies the identifier whose element(s) needs to be removed.

<IndexList>
A 4-byte unsigned integer followed by a list of indexes. Each index refers to an element to be removed from the <Identifier>. 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_ELEMENT request must first authenticate the client as the administrator with the REMOVE ELEMENT privilege. Upon successful authentication, the server will proceed to remove the elements specified in the <IndexList> from the <Identifier>. If successful, the server will return an RC_SUCCESS message to the client.

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

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

7.7.3 MODIFY ELEMENT(S)

Clients can make modifications to an existing element by sending MODIFY_ELEMENT requests to the responsible server. The Message Header of the MODIFY_ELEMENT request must set its <OpCode> to OC_MODIFY_ELEMENT.

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

```
<Message Body of MODIFY_ELEMENT Response> ::= <Identifier>
                                              <ElementList>
```

where

<Identifier>
A UTF8-String that specifies the identifier whose element(s) needs to be modified.

<ElementList>
A 4-byte unsigned integer followed by a list of elements. The integer specifies the number of elements in the list. Each element in the <ElementList> specifies an element that will replace the existing element with the same index.

The server that receives the MODIFY_ELEMENT request must first authenticate the client as an administrator with the MODIFY_ELEMENT privilege. Upon successful authentication, the server will proceed to replace those elements listed in the <ElementList>, provided each element has PUBLIC_WRITE or ADMIN_WRITE permission. If successful, the server must notify the client with an RC_SUCCESS message.

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

MODIFY_ELEMENT requests can also be used to update administrators. This happens if both the element in the <ElementList> and the element to be replaced are HS_ADMIN elements. Servers

must authenticate the client as an administrator with the MODIFY_ADMIN privilege before fulfilling such a request. If the request replaces a non-HS_ADMIN element with an HS_ADMIN element, the required privileges are MODIFY_ELEMENT and ADD_ADMIN; if the request replaces an HS_ADMIN element with a non-HS_ADMIN element, the required privileges are MODIFY_ELEMENT and REMOVE_ADMIN.

7.7.4 CREATE IDENTIFIER

Clients can create new identifiers by sending CREATE_ID requests to the responsible server. The Message Header of any CREATE_ID request must set its <OpCode> to OC_CREATE_ID. The Message Body of any CREATE_ID request is defined as follows:

```
<Message Body of CREATE_ID Request> ::= <Identifier>
                                        <ElementList>
```

where

<Identifier>
A UTF8-String that specifies the identifier.

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

Only prefix administrators with the CREATE_ID privilege are allowed to create new identifiers under the prefix. The server that receives a CREATE_ID request must authenticate the client as the administrator of the corresponding prefix identifier and make certain that the administrator is authorized to create identifiers under the prefix. This is different from the ADD_ELEMENT request where the server authenticates the client as an administrator of the identifier. Upon successful authentication, the server will proceed to create the new identifier and add each element in the <ElementList> to the new <Identifier>. If successful, the server will return an RC_SUCCESS message to the client.

Each CREATE_ID request must be carried out as a transaction. If any part of the CREATE_ID process fails, the entire operation can be rolled back. For example, if the server fails to add elements in the <ElementList> to the new identifier, it must return an error message without creating the new identifier. A CREATE_ID request without the overwrite-when-exists (OWE) flag set that asks to create an identifier 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_ID_ALREADY_EXIST.

If the request has the OWE flag set, then the server will modify existing elements to match the request, rather than returning RC_ID_ALREADY_EXISTS. In this case the server can authenticate the

client as an administrator of the identifier (needing various privileges depending on the aggregate changes made) rather than as a prefix administrator.

If the request has the MNS (mint-new-suffix) flag set, the <Identifier> in the request is treated as the initial portion of the eventual identifier to be created. The initial portion should include the slash. The server will create a new suffix to be appended to the initial portion, thus creating a new identifier.

The Message Body of a successful Create Identifier Response is defined as follows:

```
<Message Body of CREATE_ID Response> ::= [<RequestDigest>]
                                         <Identifier>
```

The <RequestDigest> is the optional request digest of the Create Identifier Request, as defined in section 6.2.3. The <Identifier> is the UTF8-String of the identifier that was actually created, allowing the client to discover the created identifier when the MNS flag was set.

CREATE_ID requests can also be used to create derived prefixes. Before creating a new prefix identifier, the server must authenticate the client as the administrator of the parent prefix. Only administrators with the CREATE_DERIVED_PREFIX privilege are allowed to create any sub-prefix.

7.7.5 DELETE IDENTIFIER

Clients delete existing identifiers by sending DELETE_ID requests to the responsible DO-IRP server. The Message Header of the DELETE_ID request must set its <OpCode> to OC_DELETE_ID.

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

```
<Message Body of DELETE_ID Request> ::= <Identifier>
```

where

<Identifier>
A UTF8-String that specifies the identifier.

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

Each DELETE_ID request must be carried out as a transaction. If any part of the DELETE_ID process fails, the entire operation must be rolled back. For example, if the server fails to remove any elements assigned to the identifier (before deleting the identifier), it must return an error message without deleting the identifier. A DELETE_ID request that asks to delete a non-existing identifier will

also be treated as an error. The server will return an error message with the <ResponseCode> set to RC_ID_NOT_EXIST.

DELETE_ID requests can also be used to delete prefix identifiers.

7.8 PREFIX ADMINISTRATION

DO-IRP expects configuration pertaining to a prefix to be managed in the identifier record for that prefix. For example, administrators can change the service information of any prefix by changing the HS_SITE elements in the corresponding prefix identifier record. Creating or deleting prefixes is done by creating or deleting the corresponding prefix records. Derived prefixes may be created by the administrator of the parent prefix.

Clients can request a DO-IRP server to list identifiers or derived prefixes under a prefix. The client should be prepared to authenticate the administrator by using the client software. Details of these operations are described in the following sections.

7.8.1 LIST IDENTIFIER(S) FOR A SPECIFIED PREFIX

This command is available only to administrators. Clients send LIST_IDS requests to DO-IRP servers to get a list of identifiers for a given prefix. The Message Header of the LIST_IDS request must set its <OpCode> to OC_LIST_IDS.

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

```
<Message Body of LIST_IDS Request> ::= <Prefix_Identifier>
```

where

<Prefix_Identifier>
A UTF8-String that specifies the prefix identifier.

To obtain a complete list of the identifiers for a given prefix that are registered in a local identifier service site that consists of a set of servers, the client must query every DO-IRP server denoted in the HS_SITE information and get the list of identifiers from each of the servers separately. Upon request from an authorized party, each server within the designated identifier service site will return its list of identifiers for the specified prefix. Identifiers for derived prefixes must be separately requested.

The Message Body of a successful LIST_IDS response (from each server) is defined as follows:

```
<Message Body of LIST_IDS Response> ::=  <Num_Identifiers>
                                         <IdentifierList>
```

where

<Num_Identifiers>
Number of identifiers listed in this message.

<IdentifierList>
A list of UTF8-Strings, each of which identify a identifier for the prefix.

The LIST_IDS request may potentially slow down the overall system performance. A DO-IRP service (or its service site) has the option of whether or not to support such a request; and if it does support it, whether to make it available only to administrators or not. The server will return an RC_OPERATION_DENIED message if LIST_IDS is not supported. The server that receives a LIST_IDS request should authenticate the client as a prefix administrator with the LIST_IDS privilege before fulfilling the request.

Responses to list requests are the only messages described in this document which typically use the CN (continuous) bit, indicating that multiple messages are sent as a single response. When there are a large number of identifiers to be listed, the server can send multiple response messages, each containing a chunk of the response list (for example, 50 identifiers); each response message except the last will have the CN (continuous) bit set.

7.8.2 LIST DERIVED PREFIXES FOR A SPECIFIED PREFIX

Clients send LIST_DERIVED_PREFIXES requests to servers to get a list of derived prefixes for a specified prefix. The Message Header of the LIST_DERIVED_PREFIXES request must set its <OpCode> to OC_LIST_DERIVED_PREFIXES. This command is available only to administrators as described by the permission in HS_ADMIN.

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

```
<Message Body of LIST_DERIVED_PREFIXES Request> ::= <Prefix_Identifier>
```

where

<Prefix_Identifier>
A UTF8-String that specifies the prefix identifier.

To obtain a complete list of the derived prefixes, the request must be sent to every DO-IRP server listed in one of the service sites of the DO-IRP service. Each server within the service site will return its own list of derived prefixes for the given prefix.

The Message Body of a successful LIST_DERIVED_PREFIXES response (from each server) is defined as follows:

<Message Body of LIST_DERIVED_PREFIXES Response> ::= <Num_Identifiers>
<IdentifierList>

where

<Num_Identifiers>
Number of derived prefixes listed in this message.

<IdentifierList>
A list of UTF8-Strings, each of which identifies a derived prefix for the user-specified prefix.

LIST_DERIVED_PREFIXES requests must be sent to servers that manage prefix identifiers. The LIST_DERIVED_PREFIXES request may potentially slow down the overall system performance. A server (or service site) has the option of whether or not to support such requests. The server will return an RC_OPERATION_DENIED message if LIST_DERIVED_PREFIXES is not supported. The server that receives a LIST_DERIVED_PREFIXES request should authenticate the client as a prefix administrator with the LIST_DERIVED_PREFIXES privilege before fulfilling the request.

Like responses to LIST_IDS, these responses may make use of the CN (continuous) bit, in order to break up the list into multiple separate response messages.

7.9 MANAGEMENT OF HOMED PREFIXES

Servers need some way to determine whether to send RC_ID_NOT_FOUND or RC_SERVER_NOT_RESP in response to a query request for an identifier which has no record in the server's data store. Not responsible means that the server has not been authorized to administer that identifier, but it is important to note that this is purely internal to the single server in question.

One option, described below, is for the server to maintain a list of prefixes for which it is authorized to administer identifiers beginning with those prefixes. These are called "homed prefixes" and it is said that the prefixes are "homed to the server", but again this is purely internal to that server's configuration.

Certain DO-IRP messages are set aside for the maintenance of this list of homed prefixes. For servers using this option, clients send HOME_PREFIX requests to the servers to ensure that a prefix is included in the list, and UNHOME_PREFIX requests to ensure that a prefix is excluded. The Message Header of a HOME_PREFIX request must set its <OpCode> to OC_HOME_PREFIX, and an UNHOME_PREFIX request must set its <OpCode> to OC_UNHOME_PREFIX. These commands are available only to server administrators specified in the server configuration.

The Message Body of any HOME_PREFIX or UNHOME_PREFIX request is defined as follows:

<Message Body of HOME_PREFIX or UNHOME_PREFIX Request> ::=
<Prefix_Identifier>

If successful, the server will return an RC_SUCCESS message to the client.

Clients send LIST_HOMED_PREFIXES requests to servers to get a list of homed prefixes. The Message Header of the LIST_HOMED_PREFIXES request must set its <OpCode> to OC_LIST_HOMED_PREFIXES. This command is available only to administrators as described by the permission in HS_ADMIN.

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

<Message Body of LIST_HOMED_PREFIXES Request> ::= <Ignored_Identifier>

where

<Ignored_Identifier>
A UTF8-String, which may be empty; ignored by the server, but included to maintain the pattern that all requests begin with a UTF8-String identifier.

The Message Body of a successful LIST_HOMED_PREFIXES response is defined as follows:

<Message Body of LIST_HOMED_PREFIXES Response> ::= <Num_Identifiers>
<IdentifierList>

where

<Num_Identifiers>
Number of homed prefixes listed in this message.

<IdentifierList>
A list of UTF8-Strings, each of which identifies a homed prefix.

Like responses to LIST_IDS, these responses may make use of the CN (continuous) bit, in order to break up the list into multiple separate response messages.

DO-IRP servers which are authorized to administer prefix identifiers may include 0.NA in the list of homed prefixes by using the prefix identifier 0.NA/0.NA in a HOME_PREFIX request. However, in such cases, it is generally more useful for users if such a server indicates that it is authorized to administer only a subset of such prefix identifiers. To this end, the special notation 0.NA/0.NA/<X> may be used in a HOME_PREFIX request to indicate that the server is authorized to administer identifiers of prefixes derived from the prefix X.

A PRS server should never respond RC_SERVER_NOT_RESP to a query for a prefix. It should, however, include 0.NA in its list of homed prefixes, even if in a strict sense, the party managing the server is only formally authorized to administer certain prefixes. In other words, PRS servers should

be configured as "authorized to administer" exactly those identifiers for which it should return RC_ID_NOT_FOUND instead of RC_SERVER_NOT_RESP.

7.10 SESSIONS AND SESSION MANAGEMENT

Sessions are used to allow sharing of authentication information or network resources during multiple sequential protocol operations. For example, a prefix administrator may authenticate itself once through the session setup, and then register multiple identifiers 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 (via MAC) 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 with session keys are 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_ID request that requires client authentication. Such a session does not have a session key and cannot be reused past the challenge.

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.

7.10.1 SESSION SETUP REQUEST

Clients establish a session with a DO-IRP server using 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.

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

```
<SESSION_SETUP Request Message Body> ::= <Mode>
                                         <Timeout>
                                         <IdentityIdentifier>
                                         <IdentityIndex>
                                         <PublicKey>
```

where

<Mode>
A 2-byte integer which indicates how the client and server exchange the session key. This value must be 4, indicating Diffie-Hellman key exchange as described below.

<Timeout>
A 4-byte integer. The client is requesting that the server timeout the session after this many seconds elapse without requests.

<IdentityIdentifier>
UTF8-String that designates the administrative identifier intended to be used in this session. The administrator is the individual who holds the private key that is used to authenticate using the standard Challenge mechanism later in the session. An anonymous session can be created by setting <IdentityIdentifier> to the empty UTF8-String.

<IdentityIndex>
A 4-byte unsigned integer that specifies the index of the element (of the <IdentityIdentifier>) that holds the public or secret key of the administrator (or 0 to indicate authentication using any element).

<PublicKey>
A 4-byte integer length followed by that many octets encoding a Diffie-Hellman public key. This is encoded as the UTF8-String "DH_PUB_KEY", followed by two zero octets, followed by three byte arrays, each prepended with a 4-byte integer length. The three byte arrays are the parameters y, p, and g to be used in the Diffie-Hellman exchange.

7.10.2 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.

The Message Body of the SESSION_SETUP response is defined as follows:

```
<Message Body of SESSION_SETUP Response> ::= <RequestDigest>
                                             <Mode>
                                             <Data>
```

where

<RequestDigest>
Message digest of the SESSION_SETUP request is as specified in section 6.2.3.

<Mode>
A 2-byte integer which indicates how the client and server exchange the session key. This value must be 4, indicating Diffie-Hellman key exchange as described below.

<Data>
A four-byte integer, followed by a sequence of octets. The initial integer indicates the number of octets. The octets are a 4-byte integer, indicating the encryption algorithm used for the shared secret key, discussed below; and a Diffie-Hellman public key, encoded as the UTF8-String "DH_PUB_KEY", followed by two zero octets, followed by three byte arrays, each prepended with a 4-byte integer length. The three byte arrays are the parameters y, p, and g to be used in the Diffie-Hellman exchange. The p and g parameters will match those used by the client in the SESSION_SETUP request.

The encryption algorithm 4-byte integer sent by the server indicates the key size of the secret key. Even in the absence of encryption, this parameter is used to indicate the size of the key which is used for MAC codes to authenticate session messages. 1 indicates a 8-byte key (suitable for DES encryption); 2 indicates a 24-byte key (suitable for Triple DES encryption); 3 indicates a 32-byte key (suitable for AES encryption). Servers should use 3 and clients should generally refuse further communication with servers that suggest using keys sized for DES. The shared secret from the Diffie-Hellman key exchange has any initial zero octets removed, and is then truncated to the size indicated by the encryption algorithm; in the case of DES or Triple DES, parity bits are then adjusted.

The server will include a new non-zero session id in the <MessageEnvelope> of the response. The session setup response will be signed using the HS_SIGNED method and the server's public key; future messages in the session, both client and server, will be signed using the HS_MAC method using the shared secret key obtained via the Diffie-Hellman exchange. Messages can also be encrypted at the choice of either client or server. If the client encrypts its request, the server should encrypt the resulting response. The client may have specified an administrator identifier and index in its session setup request, however that identity will need to be authenticated using a standard Challenge, when the client sends another request which requires authentication. Once that is done, further requests in the same session will assume that administrator identity without challenge.

7.10.3 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.

8 IMPLEMENTATION GUIDELINES

8.1 SERVER IMPLEMENTATION

The optimal structure for any DO-IRP server will depend on the host operating system. This section only addresses those implementation considerations that are common to most DO-IRP 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 DO-IRP server implementations must support all the types as defined in this document. They should also be able to store elements for application defined types.

A DO-IRP 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 server should not block the service of UDP requests while it waits for TCP data or other query activities. Similarly, a 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.

While an option to shard identifier records across multiple servers is provided using the HashOption, implementations should consider other possibilities to enable load balancing. If data storage scalability is important, a scalable storage system can be used at a single server based service site. If distribution of service is important, multiple servers can be used behind a load balancer with each server talking to the same storage system. Yet another possibility is to do both, namely use a scalable storage system fronted by a load-balanced set of DO-IRP servers.

8.2 CLIENT IMPLEMENTATION

Clients should be prepared to receive elements of any type. Only those types, server features, operations, and interfaces defined in this document are expected to be commonly supported by implementations.

Clients that follow service referrals or 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 alias. Although no specific recommendation is made here, there should be a configurable upper limit to the counter to control the levels of service referrals or aliases followed by the client.

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

9 TYPE IDENTIFIER CONSIDERATIONS

The identifier for a type can be specified by the creator of the element as they so desire but this standard recommends that it be an identifier resolvable using DO-IRP to make it globally resolvable and accessible. The prefix 0.TYPE is reserved to identify all types defined in this document. Many system-defined types are identified using a simple string without a prefix as they were defined before this recommendation was made. Each of those system type has a corresponding identifier consisting of the prefix 0.TYPE followed by their name as a suffix. For example, the defined system type HS_ADMIN will have a corresponding identifier 0.TYPE/HS_ADMIN that when resolved using DO-IRP, will provides a full description of the HS_ADMIN type. This mechanism allows for new system types to be added in the future.

All other types should be uniquely identified and registered as identifiers resolvable using DO-IRP to avoid potential conflicts. For example, the string “URL” (as shown in Figure 4.1) is a registered type with specific characteristics and function and has the identifier “0.TYPE/URL”. Other prefixes, including, especially, those prefixes allotted to organizations, can be used to create identifiers for types. In such cases, it is assumed that one or more type registries, possibly federated registries, will be made available to foster type discovery and reuse. This DO-IRP specification does not itself specify any requirements about the DO-IRP identifier records used for types, or other information that might be made accessible concerning types. It is recommended that the identifier record have at a minimum human-readable information about how the type is used, and information about who to contact for more information.

Identifiers used to specify a given type may use their prefix/suffix syntax to reflect the hierarchical nature of the type. Each node of the hierarchy shall make use of a UTF8-String with no “.” (0x2E) characters. The “.” character is used to mark the boundary between nodes of the hierarchy. For example, the type with a prefix “a.b” may be considered as derived from type identified as “a”. Similarly, elements of <type> “a.b.x”, “a.b.y” and “a.b.z” may be considered as elements under the common type hierarchy “a.b”. For types which are identifiers with a slash, this applies to suffixes. The type “a/b.c” may be considered as a derived from the type identified a “a/b” and similarly, the

elements of <type> “a/b.c.d”, “a/b.c.e” and “a/b.c.f” may be considered as elements under the common type hierarchy “a/b.c”.

10 SECURITY CONSIDERATIONS

DO-IRP delegates administration to each administrator who may or may not be the server administrator. DO-IRP administrators are allowed to choose their own public/secret keys used for authentication. The security of DO-IRP authentication depends on the proper key selection and its maintenance by the administrator. Administrators must choose and protect their authentication keys carefully in order to protect the data. DO-IRP server implementations may deploy policies that regulate the selection of public/secret keys used for authentication. For example, a server may require that any authentication key must be no less than a certain number of bits. It may also prohibit the use of secret keys because of the potential dictionary attack.

The use of DO-IRP with UDP is potentially vulnerable to amplification attacks, especially in cases where the size of the identifier records requested is considerably larger than the size of the resolution request. When an organization needs the extra performance that UDP resolution allows, it should ensure that network configuration properly mitigates the risks from such attacks by taking measures such as using UDP in a closed network or limiting responses to the same requests by the same client. Consideration should also be given to turning off UDP access by default.

To protect against any irresponsible use of system resource, DO-IRP servers may implement a quota control strategy. The quota control strategy can be used to put limits on the number of identifiers using a given prefix, the number of elements allowed for any given identifier record, the maximum size of any element, and the number of derived prefixes under a given prefix. Servers must report errors if the result of an administration action violates any of these limits.

Secret key authentication will not send a secret key over the network. Instead, secret key authentication involves a key-server (potentially identical to the server being queried) which knows the shared secret of an administrator and can use it to reproduce the same Message Authentication Code. This means that the secret key is vulnerable to any attacker which can gain access to the storage of the key-server. It is recommended to use public/private key authentication whenever possible. Moreover, dictionary attacks can be performed to extract the secret key from the key-server; and it is recommended that DO-IRP server implementations limit the number of allowed incorrect attempts to authenticate a given administrator during any short window of time.

For efficiency, DO-IRP includes a simple challenge-response authentication process for basic client authentication.

Data integrity under DO-IRP 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 to make it known to clients.

Session management in the protocol has been designed to prevent session replay attacks, as might typically occur in a simple request/response message exchange. This involves a unique session counter in each client authentication exchange, as described in section 6.2.4. Each session is protected by its session key, and the Session ID ceases to exist after the session terminates.

11 INFORMATIVE REFERENCES

[1] S. Sun, L. Lannom and B. Brian, "Handle System Overview (RFC3650)," CNRI, November 2003. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc3650.

[2] S. R. L. L. S. Sun, "Handle System Namespace and Service Definition (RFC3651)," CNRI, November 2003. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc3651.

[3] S. Sun, J. Petrone, S. Reilly and L. Lannom, "Handle System Protocol (ver 2.1) Specification (RFC3652)," CNRI, November 2003. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc3652.

[4] F. Yergeau, "UTF-8, a transformation format of unicode and ISO 10646 (RFC2044)," October 1996. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc2044.

[5] B.-L. Tim, L. Masinter and M. McCahill, "Uniform Resoruce Locators (URL) (RFC1738)," December 1994. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc1738.

[6] Jones, M; Microsoft, "JSON Web Key (RFC7517)," May 2015. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7517.

[7] Information Technology Laboratory, NIST, "Secure Hash Standard (SHS)," August 2015. [Online]. Available: https://doi.org/10.6028/NIST.FIPS.180-4.

[8] R. Fielding and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing (RFC7230)," June 2014. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7230.

[9] R. Fielding and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content (RFC 7231)," June 2014. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7231.

[10] J. Jonsson and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1 (RFC3447)," February 2003. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc3447.

[11] Information Technology Laboratory, NIST, "Digital Signature Standard," July 2013. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.186-4.pdf.

[12] R. Housley and RSA Laboratories, "Cryptographical Message Syntax (CMS) Algorithms (RFC 3370)," August 2002. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc3370.

[13] R. Rivest, "The MD5 Message-Digest Algorithm (RFC1321)," MIT Laboratory for Computer Science and RSA Data Security, Inc., April 1992. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc1321.

[14] R. Kahn and R. Wilensky, "A framework for distributed digital object services," 13 March 2006. [Online]. Available: https://www.doi.org/topics/2006_05_02_Kahn_Framework.pdf.

[15] R. Kahn, C. Blanchi, L. Lannom, P. A. Lyons, G. Manepalli, R. Tupelo-Schneck and S. Sun, "Digital Object Interface Protocol Specification Version2.0," 12 November 2018. [Online]. Available: https://www.dona.net/sites/default/files/2018-11/DOIPv2Spec_1.pdf.