Corporation for National Research Initiatives

Digital Object Protocol Specification

Sean Reilly <sreilly@cnri.reston.va.us>

Version: 1.0

November 12, 2009

This work was supported in part by DARPA under Grant HR0011-05-1-0003.

Table of Contents

1. Overview

CNRI's Digital Object Architecture provides a mechanism for the creation of and access to digital objects as discrete data structures with unique, resolvable identifiers. These Digital Objects provide a foundation for representing and interacting with information on the Internet. The Repository Access Protocol (RAP) consists of three parts: 1) An identifier resolution system (Handle), 2) a low level communication protocol (Digital Object Protocol, or DOP), and 3) a set of Digital Object Operations (DOO) that can be applied to digital objects over DOP connections. DOP provides entity authentication, connection encryption and multi-channel communication over a single TCP/IP socket. The Handle System is specified at http://handle.net/ in three IETF RFCs, and this document provides a specification of the Digital Object Protocol and Digital Object Operations that combine to form RAP.

2. Digital Object Protocol

This section describes the Digital Object Protocol (DOP) - the protocol and authentication mechanism used to interact with digital objects through a connection to a digital object server. Each interaction consists of a caller invoking or applying an operation on a digital object. The server, caller, each operation, and the object itself are all uniquely identified using Handle Identifiers.

Operations are applied to objects by digital object servers in which the objects are said to reside. The DOP defines the method by which entities on the network communicate with DO servers for the purpose of invoking operations on the digital objects within them.

A digital object server is addressed as a digital object that contains other digital objects. Operations are applied to digital objects by the digital object servers in which the objects reside. The DOP defines the method by which entities on the network communicate with DO servers for the purpose of invoking operations on the digital objects within them.

An operation on a digital object consists of the following elements:

- CallerID: The identifier of the entity requesting invocation of the operation

- ObjectID: The identifier of the object to be operated upon

- OperationID: The identifier that specifies the operation to be performed

- Input: A stream of bytes that contains the input to the operation, including any parameters, or content

- Output: A stream of bytes that contains the output of the operation, including any content or messages.

Upon invocation of an operation, the client will write data to the invocation's input stream and read data from the output stream. The DOP API includes a mechanism for including operation parameters as a set of key-value pairs that are considered part of the input.

2.1. Locating an Object

The first step to interacting with a digital object is locating the server where it currently resides. This is done by using the Handle System to resolve the handle identifier of the digital object to a set of handle values (Figure 1). Every handle value has a type (UTF-8 string) and a data (byte array) field whose format depends upon the content of the type field. The data field of the handle value with type "CNRI.OBJECT_SERVER" is a UTF-8 string containing the identifier of the DO server that is responsible for the object.

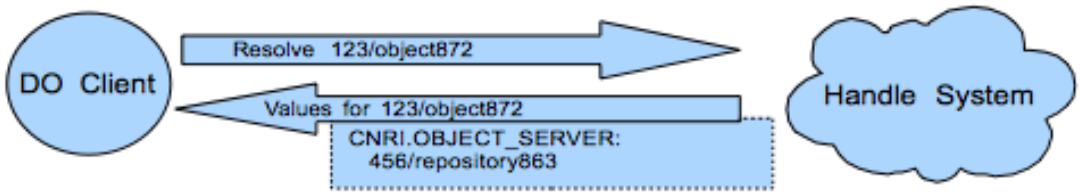


Figure 1: Resolving an Object ID to a Server

Resolving the DO server identifier yields a handle value with type CNRI.OBJECT_SERVER_INFO that contains information about the server such as IP address, public key, port number, protocol version, name and description. This information is encoded in an XML format, leaving room for additional information for future implementations. With this information the DO client can connect to, authenticate, and interact with digital objects on the same server. The XML format for the server information is as follows:

The top level tag is <serverinfo> and contains one or more <server> tags. Each <server> tag contains the following sub-tags:

<id> - An identifier that differentiates this server from others in the same service

<label> - A natural language label for the server

<publickey> - A hexadecimal encoded public key that can be used to authenticate the server

<hostaddress> - A text string indicating the IP address of the server

<port> - An integer encoded as a text string, indicating the port on which the server can be contacted

<protocol> - A text string indicating the basic protocol spoken by the server

The following is an example of a CNRI.OBJECT_SERVER_INFO value:

<serverinfo>

<server>

<id>1</id>

<label>sean's test server</label>

<publickey>0000000B4453415F5...</publickey>

<hostaddress>127.0.0.1</hostaddress>

<port>9901</port>

<https-port>443</https-port>

<protocol>DOP</protocol>

</server>

</serverinfo>

During handle resolution, the 'certify' flag is set for all Handle System messages in order to verify that the information returned by the Handle System was not modified by an unauthorized party.

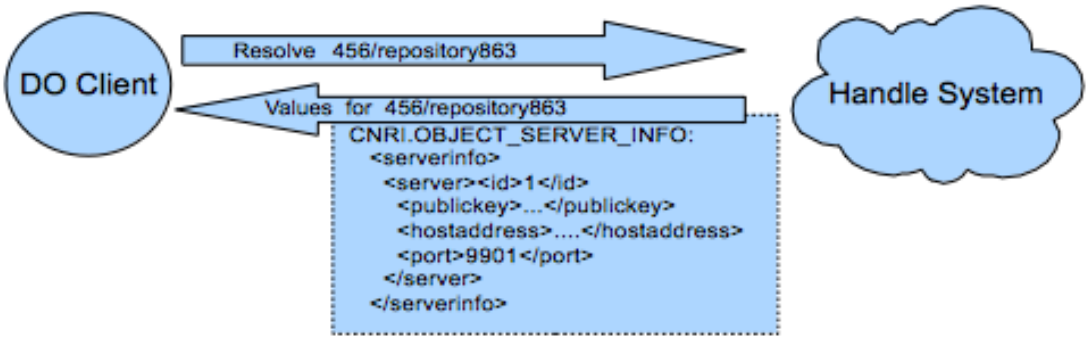


Figure 2: Resolving a Server ID to Connection Information

Note: A client can choose to invoke operations on digital objects via registries that are not referenced in the digital object's handle. This capability can be used to implement proxies for interacting with digital objects in order to provide additional services and operations on the objects. For more information on this feature see the Forwarding Operations section.

2.2. Communication Messages

Most messages in the DO protocol are represented as sets of key-value pairs with a single UTF-8 text label. Pairs are ASCII text strings delimited by ampersands (&). Each pair is split into a key and value by an equals sign(=). All non-ASCII characters, equals signs and ampersands in the keys or values are UTF-8 encoded and %-escaped. A message has the following structure:

```
<message> := <messagetype> ':' <segment><newline>
<segment> :=
<segment> := <kvpair> <segment> '&' <kvpair>
<segment> := <kvpair>
<kvpair> := <key>
<kvpair> := <key> '=' <value>
<messagetype> := <encodedtoken>
<key> := <encodedtoken>
<value> := <encodedtoken>
<encodedtoken> := <encodedtoken><alphanumeric>
<encodedtoken> :=
<alphanumeric> := 'A-Z'
```

```
<alphanumeric> := 'a-z'
<alphanumeric> := '0-9'
<alphanumeric> := '%'
<alphanumeric> := '_'
<alphanumeric> := '-'
<newline> := 10
```

In the DO API, keys are represented as UTF-8 text strings. Values are stored as UTF-8 strings, but many data types can be represented using a standard encoding:

byte arrays - bytes are encoded in a hexadecimal string representation

integers - encoded as the decimal string representation using ASCII 0-9

real - encoded as the decimal string representation using ASCII 0-9 and .

boolean - encoded as an ASCII 1 or 0

string array - strings have ASCII commas escaped with an ASCII backslash, then concatenated together separated by an ASCII comma.

sub-messages - key-value sets can be embedded in other messages by adding each key-value pair after pre-pending a certain string to each key from the merged set

These messages can be very easily encoded and decoded using a trivial amount of code and resources, yet the messages themselves are extremely flexible and can convey arbitrarily complex information.

In this document, messages will be displayed as a message type followed by key-value pairs on separate lines as in the following example:

messagetype:

key1 = value1

key2 = value2

key3 = value3

This is not to be confused with the actual encoding of those messages which looks more like the following:

messagetype:key1=value1&key2=value2&key3=value3

2.3. Connection Initialization

Communication with a digital object server is established using a standard TCP/IP socket connected to the address and port contained in the CNRI.OBJECT_SERVER_INFO handle value described above. The client initiates communication by sending a message with the following key-value pairs:

init:

protocol = dop

protocol_major_version = 1

protocol_minor_version = 0

The server responds with an acknowledgment that includes the highest protocol version that both the client and server understand:

response:

status = OK

protocol_major_version = 1

protocol_minor_version = 0

If there is an error with the first exchange, the server will response with a status other than "OK" and an optional "code" and "message" pairs:

response:

status = ERROR

code = 102

message = Unknown protocol: 'HTTP'

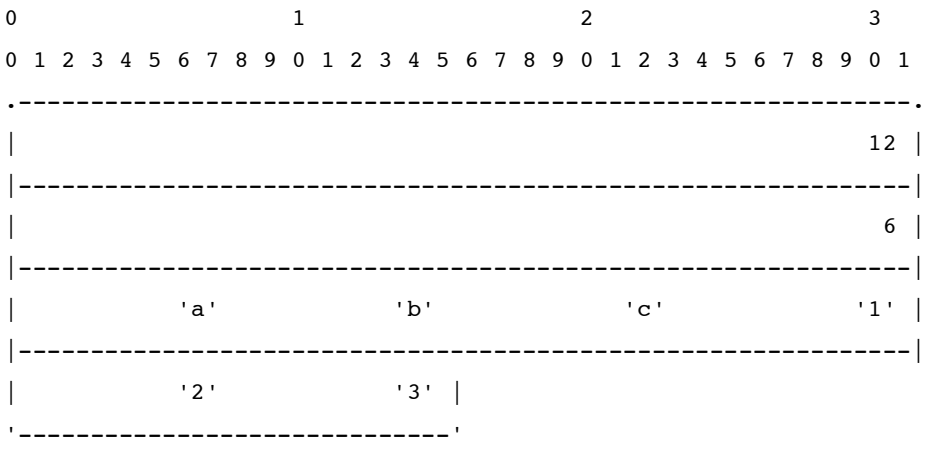
Additional key-value pairs can be included with all messages in order to provide additional information about the client, communication parameters, etc. Unexpected key-value pairs should simply be ignored.

Immediately after sending the initial response, the server will go into multi-channel mode. The client will go into multi-channel mode immediately upon receiving the initial response from the server. Multi-channel mode is described in the next section.

2.4. Multi-Channel Mode

In multi-channel mode, multiple communication channels are multiplexed over a single socket connection. A communication channel consists of a pair of byte streams, one from which incoming bytes can be read, and one to which outgoing bytes can be written. Bytes written to the outgoing stream of a channel can be read from the incoming stream of the corresponding channel on the other side of the connection, and vice versa.

Each channel is uniquely identified within a connection by a 32 bit integer. When bytes are written to a channel, they are prepended by the channel identifier and the number of bytes being written. For example, writing the bytes 'abc123' to channel 12 of a connection will cause the following to be written to the socket connection:



The channel ID, number of bytes, and content bytes are referred to as a 'chunk'. The process that manages the connection must ensure that only one chunk is written to the connection at a time. It is recommended that channel streams be buffered and flushed only when appropriate to ensure that fewer chunks are sent.

On the receiving end, it is recommended that the process that manages a connection maintain a thread that reads chunks from the connection and adds them to a buffer associated with each channel where they can be read by separate threads.

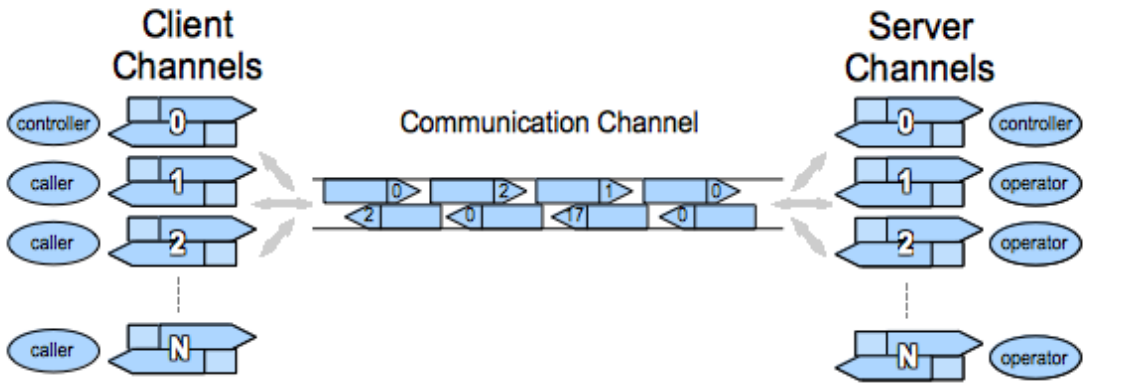


Figure 3: Multiplexing Chunks into Channels

2.5. Connection Control

Upon entering multi-channel mode, both the client and server understand that a single channel with identifier 0 already exists. Channel 0 is referred to as the 'control channel' and is used by the process that manages the connection to exchange messages with the other side of the connection. Communication on the control channel consists of a series of messages as defined above. Because these messages are encoded in ASCII and terminated with a line feed, control messages are easily debugged with simple tools.

Some of these messages, called 'synchronous' requests, require a response from the other side of the connection in order to fulfill their purpose. Because messages may be interleaved on the control channel, synchronous requests have an additional value with a key of '_requestid' added to them by the connection manager. The recipient of a synchronous request will include the same '_requestid' value in the response to the request. After a synchronous request is sent, the message sender waits for a response message that contains the same value for the '_requestid' key. When that response is received, or an appropriate amount of time has elapsed (currently 10 minutes) control is returned to the process that sent the synchronous request with either reference to the response or an error code.

Messages sent over the control channel are used to open new channels, authenticate the client and server to each other, establish an encrypted link for the connection, and any other connection-level operations.

Both sides of a connection will continually listen for messages on the control channel. When a control message is received, the message type field determines how it is processed. The following message types are understood in version 1.0 of the protocol:

openchannel: Requests that a new channel be opened. The sender must provide a positive integer value associated with the key 'channelid' that indicates the identifier for the channel that is to be created. If the client provides no identifier, an invalid identifier (ie non-integer or negative value), or an identifier for a channel that already exists then the server will return an response with an error message to the client. In version 1.0 of the protocol, messages with type 'openchannel' are only sent by the client. Messages with type 'openchannel' require a response and can only be sent by clients. In the Digital Object API, server connections register a callback/listener that is notified when new channels are opened. Every new channel is treated as an operation on a digital object.

closestream: Indicates that the other side of the connection has closed a byte stream for one of the open channels. Messages of this type should include both a 'channelid' and 'streamid' key whose values indicate the channel ID and the byte stream that is now closed. The channel ID is an integer value, and stream ID is one of 'input' or 'output'. If the stream ID is 'input' then the byte stream from which the message recipient reads is closed, If the stream ID is 'output' then the byte stream to which the message recipient writes is closed. Attempts to read from a closed input stream or write to a closed output stream should result in an error. Messages of type 'closestream' can be sent by the client or server and do not require a response.

authenticate: Requests that the message recipient authenticate themselves, proving their identity to the message sender. Messages with type 'authenticate' can be sent by either the client or server and require a response. Authentication messages are outlined in detail in the 'Authentication and Connection Encryption' section.

response: Indicates that this is a response to a message that was sent by the this side of the connection. The connection manager should extract the value associated with the '_requestid' key and use it to locate the pending request with the same value for the '_requestid' key. If no request with the same request ID exists, then the response is ignored. If a request with the same request ID exists, then any processes that are waiting for a response to the request are notified of the response, and the request is removed from the list of requests awaiting responses.

2.6. Authentication and Connection Encryption

The purpose of authentication with the Digital Object system is to verify the identity of the entity on the other side of the connection for logging purposes as well as to establish their access privileges. This is done using a modular authentication system that is currently based on DSA or RSA public/private keys, but can easily be fitted to an alternate system.

The security built into the Handle System plays a large role in the authentication mechanism of the digital object protocol. In order to establish the identity of an entity in the digital object world, it must be possible to securely bind an identifier (handle) with a public key. This is done using certified resolution with the Handle System Protocol (see RFC 3652).

2.6.1. Authentication Request

Each side authenticates the other using an authentication request (a message with type 'authenticate') on the control channel. The authentication request must contain the *entity_id, nonce* and *setup_encryption* keys. If the boolean *setup_encryption* key has a true value, the *public_key* and *public_key_alg* keys are also required. The purpose of these keys are described below.

entity_id: String value indicating the identifier for the client that is making the connection. This is currently not used, but in a future version of the protocol the server will have the option of returning the session encryption information encrypted by a public key that is associated with the client.

nonce: Byte array containing a set of bytes to be signed by the message recipient. The byte array should contain unpredictably random bytes.

setup_encryption: Boolean value indicating whether connection level encryption should be established as part of the authentication process. The setup_encryption flag should always be set to true in the first authentication message on a connection. If a connection is not encrypted, the authentication is subject to man-in-the-middle attacks and will have no meaning. The only connections for which encryption can be omitted are connections from anonymous clients that do not require server authentication. The setup_encryption flag should only be set in the first authentica-

tion exchange, when the client authenticates the server.

public_key: If the *setup_encryption* value is true, then the public key in the request should be a Diffie-Hellman public key encoded into a byte array according to the X.509 standard.

public_key_alg: If the setup_encryption value is true, this will be the text string "DH", indicating that the *public_key* parameter is a Diffie-Hellman key. Future versions of the protocol will support more methods of exchanging secret keys.

digest_alg: This specifies the algorithm that the recipient should use as a digest if the client chooses to authenticate using the hsseckey authentication method (see below). If this key is missing then the recipient can choose their own digest algorithm when authenticating using the hsseckey method.

2.6.2. Authentication Response

The recipient of the authentication request will sign the bytes from the nonce parameter and return a response with the *auth_type* key indicating the method of authentication. If the *setup_encryption* flag in the request was set to true then the *cryptmacalg, cryptalg, cryptmode, cryptpadding, cryptsecretkey, public_key,* and *public_key_alg* keys should also be present in the response. The definitions of these keys are as follows:

auth_type: A text string that indicates the method used to generate the signature that will authenticate the recipient. Possible values are "hspubkey" and "hsseckey". The method of authentication determines which other keys must be present and the data that they may contain.

auth_alg: A text string specifying the algorithm that was used to generate the signature. The current version of the protocol supports either "SHA1withDSA", "MD5withRSA" or "SHA1withRSA". This key applies only to the "hspubkey" authentication method.

digest_alg: A text string specifying the digest algorithm that was used to generate the signature. The current version of the protocol supports either "sha1" or "md5". This key

applies only to the "hsseckey" authentication method.

client_nonce: A byte array consisting of the nonce that was generated by the recipient of the authentication request. The purpose of this nonce is to introduce an unpredictable component to the digest that ensures the resulting signature can not be used to successfully respond to an authentication challenge from the handle server. This key applies only to the "hsseckey" authentication type.

auth_response: A byte array consisting of the encoded signature. The format of this byte array depends upon the authentication type and are described below.

For "hspubkey" authentication, the hashing algorithm indicated by the *auth_alg* key (either SHA1 or MD5) is used to hash the nonce byte array from the authentication request. The result of that hash is then signed using the signature algorithm that is also indicated by the *auth_alg* key (either DSA or RSA). If a DSA key is used then the signature is encoded as an ASN.1 sequence of two INTEGER values: r and s, in that order according to the FIPS PUB 186 standard. If an RSA key was used then the signature is encoded according to the PKCS #1 standard.

For "hsseckey" authentication, the auth_response contains the digest of the following items in the order given: 1) secret key bytes, 2) nonce, 3) client nonce (specified by the *client_nonce* key), and 4) secret key bytes (a second time). The digest algorithm is specified by the *digest_alg* key and can be either "sha1" or "md5". The sender of the authentication request can specify the digest algorithm using the *digest_alg* key or it can let the recipient of the authentication request choose the algorithm. If the recipient of the authentication request uses a digest algorithm other than the one, if any, specified by the sender then the authentication fails.

cryptmacalg: A text string specifying the hashing (MAC) algorithm that is used to verify the contents of each chunk of data. The value of this key can currently be either "SHA1" or "MD5". If this key is not present, the "SHA1" algorithm is used.

cryptalg: A text string specifying the encryption algorithm that will be used. The currently supported encryption algorithm is "DESede", often referred to as triple-DES. If this key is not present, the "DESede" algorithm is used.

cryptmode: A text string specifying the mode component of the cipher. This can be one of the following:

"NONE" - No mode

"CBC" - Cipher Block Chaining Mode, as defined in FIPS PUB 81

"CFB" - Cipher Feedback Mode, as defined in FIPS PUB 81

"ECB" - Electronic Codebook Mode, as defined in FIPS PUB 81

"OFB" - Output Feedback Mode, as defined in FIPS PUB 81

"PCBC" - Propagating Cipher Block Chaining, as defined by Kerberos V4

The current implementation defaults to using ECB.

cryptpadding: A text string specifying the padding component of the cipher. This can be one of the following:

"NoPadding" - No padding

"PKCS5Padding" - The PKCS #5 scheme described in the RSA standard

The current implementation uses PKCS5Padding.

cryptsecretkey: A byte array containing the encrypted session key. This value is decrypted using the Diffie-Hellman key exchange algorithm. When using the "DESede" encryption algorithm, the decrypted value will be the DES encryption key. If the key value consists of more than 24 bytes, then only the first 24 bytes are used.

public_key: A byte array containing the Diffie-Hellman public key for the recipient of the authenticate request. The byte array is an ASN.1 encoding of a public key, encoded according to the ASN.1 type SubjectPublicKeyInfo. This public key is combined with the Diffie-Hellman keys from the authentication request to decrypt the session key in the *cryptsecretkey* value.

public_key_alg: A text string indicating the algorithm to be used with the *public_key* key. If the *public_key_alg* key is missing then it is assumed to be "DH".

Once the response is received by the sender of the original *authenticate* request, the original sender can verify the signature and thus be confident of the identity of the entity with which they are communicating. If the authentication method is "hspubkey" then the signature can be verified by retrieving the public key of the recipient using a secure mode handle resolution.

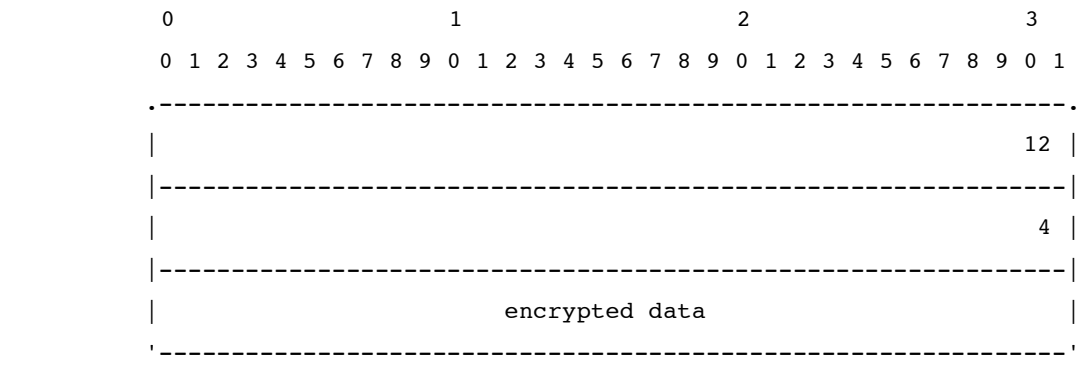
To verify an "hsseckey" authentication response, the server must build a "verify chal-

lenge" handle request (handle system op code 201) based on the nonce, client_nonce and auth_response values and submit the request (after setting the 'certify' flag) to the handle server or servers that contain the user's secret key for verification.

2.7. Connection Encryption

After receiving a response to an authenticate request, the two sides of the connection now have a shared session key which is used to encrypt further messages on the connection. It is important that the new session key be applied to the connection before any further messages are exchanged. Both sides of the connection should begin encrypting and decrypting all messages immediately after the authentication response is sent/received.

When a connection enters encrypted mode, the data portion of each 'chunk' as defined in the **Multi-Channel Mode** section are encrypted before being written to the socket. This results in chunks with the following format:



The data portion of each incoming chunk is decrypted immediately after having being read from the socket.

2.8. Additional Client Authentication (Certificates)

In addition to the public-key based authentication, clients and servers may also pro-

vide certificates establishing their credentials as a member of a group or organization. In any authentication response, clients or servers can include a set of certificates that prove their membership in an identified group.

These certificates are included with the authentication response in the following keys:

numcreds: Integer indicating the number of credentials being included

cred[0-N]: Multiple key-value pairs labeled *cred0, cred1, cred2,* etc. Each *credN* value is a byte array that contains an X.509-encoded certificate.

Both the subject and the issuer principals are encoded in the certificate as X.500 distinguished names. The handle identifier of each entity is specified in the organization ("O") attribute of the distinguished names. The organization attribute for the subject must be equal to the entity being authenticated in a case sensitive comparison of the identifiers. The entity identified as the issuer is understood to have delegated all of their access rights to the entity specified as the subject of the certificate for the period of time for which the certificate is valid. Any access that has been granted to the issuer should also be considered granted to the subject. For this reason, the entity doing the authentication is responsible for verifying that the certificate was signed by the issuer of the certificate. This can be done by resolving the issuer's identifier to a public key using a certificated handle resolution, and then using that public key to verify the signature on the certificate.

2.9. Invoking Operations

With digital object servers, the control channel (with ID zero) is used for authentication and connection administration. Each additional channel is used to invoke an operation on the digital object server. New channels are created when the client sends a *openchannel* message to the server over the control channel. The *openchannel* request includes the ID of the channel to be opened, meaning the client is responsible for preventing the use of duplicate channel identifiers.

For each new channel, the server will allocate a thread to process the new channel and perform the operation being requested. The client will send an operation request over the new channel, consisting of a message with the following keys:

callerid: A text string containing the identifier for the entity making the request

objectid: A text string containing the identifier for the digital object upon which the operation should be performed.

operationid: A text string containing the identifier for the operation that should be performed upon the object.

params: A sub-message containing application-level parameters for the operation.

After reading the initial operation request, the server will:

1) Log the operation request and begin recording everything sent over the input and output streams of the operation channel.

2) If the caller is not anonymous and has not previously been authenticated on this connection, send an *authenticate* message to the caller to establish their identity as indicated by the 'callerid' value. If the identity can not be established, return an error and close the channel.

3) Establish whether the authenticated client has permission to perform the specified operation on the specified object. If the client does not have the right to perform the operation, return an error and close the channel.

4) Perform the operation, using the input and output streams for the operation channel as input to and output from the operation.

2.10. Operation Forwarding

While many operations will be performed on objects within the repository with which a client is connected, the protocol does not require that the repository actually contain objects for which it processes operations. A repository is simply a service through which operations can be performed on objects. A repository could be configured to perform some operations locally while forwarding other operations to a remote repository. For example, a repository could perform operations on local objects itself, but forward operations on objects that were not contained in the local repository to the remote repository that was responsible for the target object. The responsible repositories are determined by resolving the object identifiers. By forwarding operations a repository can act as a proxy for local clients that wish to access outside DO repositories.

A repository has several choices to make when forwarding an operation request to a remote repository. First, the repository needs to determine whether or not it will allow the client to perform the operation. Most repositories will not forward operations on remote objects unless the client is a member of a trusted group, or is connecting from a local IP address. A repository can verify that a client is a member of a group by requesting a group certificate as described in the Additional Client Authentication section.

A repository must also decide whether the client's authentication will be forwarded to the remote server or whether the repository will substitute its own set of credentials when invoking the forwarded operation. If the repository uses its own credentials then it will be responsible for both authenticating and authorizing the client before forwarding the requested operation. When the repository uses its own credentials for forwarded operations it is hiding the identity of the client and substituting its own authentication and group membership. A repository may choose to declare only a certain subset of group memberships when forwarding operations on behalf of certain clients, thus enabling multiple levels of groups of clients even while keeping the identities of clients private.

3. Digital Object Operations

This portion of the specification describes a set of operations that can be applied to digital objects in CNRI's Digital Object Architecture. These operations can be invoked over connections with digital object servers using the Digital Object Protocol (DOP). DOP allows the invocation of arbitrary named operations on objects that reside within, or are accessible through, the digital object server.

3.1. Operation Notation

Before continuing, some notation is necessary in order to describe the digital object operations. The digital object protocol, specified in the **Digital Object Protocol** section of this document, provides a means to invoke operations on a digital object and, by extension, a DO server. In this document each operation will be described with the following notation:

[<input> |] <object>.<operation>[(<parameters>)] [| <output>]

where:

<input> is a sequence of zero or more bytes that is sent as part of the operation. For

example, when performing an "add" operation, the input may be a sequence of numeric values. The format of the bytes in the <input> sequence is determined by the specification of the operation being performed.

<object> is the identifier of the digital object upon which this operation is applied.

<operation> is the identifier of the operation to be performed.

<parameters> is a set of optional parameters passed to the operation. One example of parameters might be a set of X, Y coordinates sent to a mapping operation.

<output> is a sequence of zero or more bytes that is returned as a result of the operation invocation.

Note: Because the <input> and <output> components are byte streams, operations can read from the input stream while simultaneously writing to the output stream. Unlike HTTP and many other protocols there is no need to wait until all of the input is received before output is sent.

Because the operations are identified with handles in order to maintain uniqueness and a strong identity, the operation identifiers are generally non-semantic and require a bit of description. The operation identifiers involved in registering and accessing basic digital objects are specified in the Operation Specifications section.

3.2. Data Model

By defining operations to interact with a specific data model we can construct and use digital objects that represent any type of structure. The standard Digital Object data model is reflected in the DO API and illustrated in Figure 4. Storage details are omitted from the interface specification so that implementations can store data in the best way for a given environment.

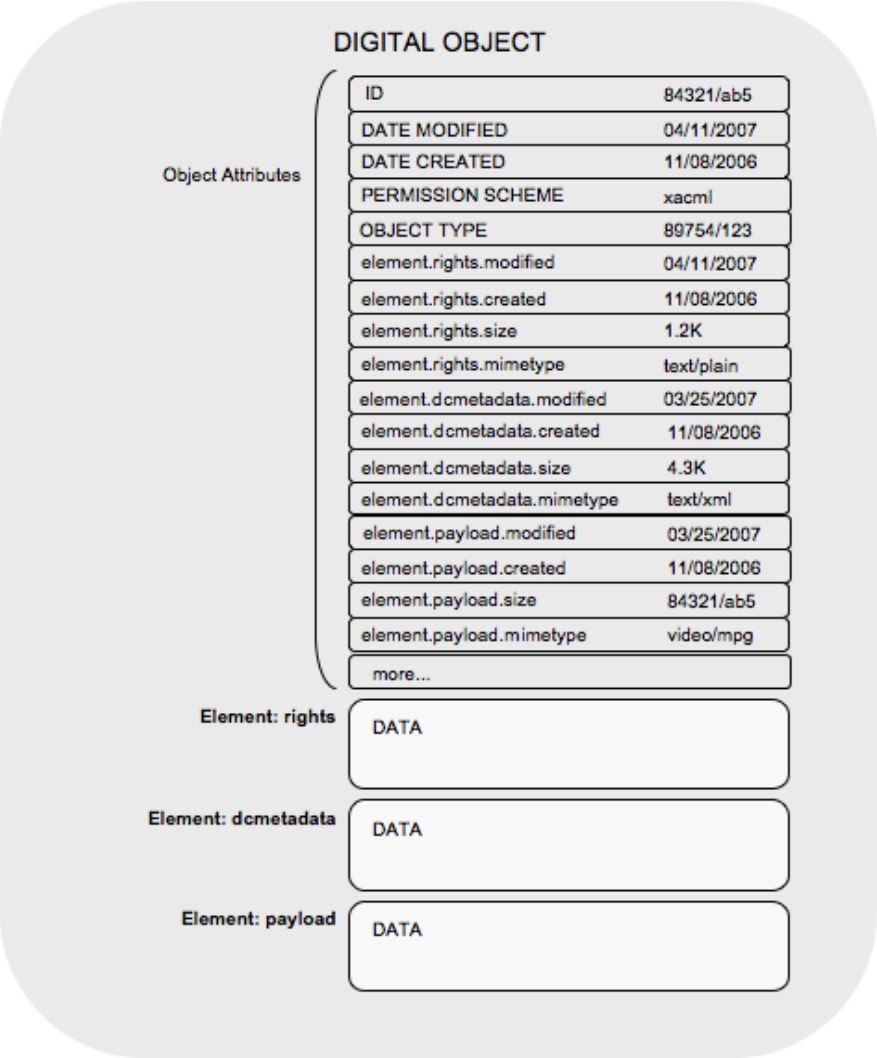


Figure 4: Example Instance of the DO Data Model

Each DO has a fixed, or intrinsic, set of attributes, a user defined set of attributes, and zero or more elements containing the DO content, e.g., text, video, images, etc. Each element, in turn, has a fixed set of attributes, a user defined set of attributes, and the data pay load. All of these pieces are exposed and made available through the API. Intrinsic attributes are those controlled by the Repository and will be part of every DO and every Element.

The DO intrinsic attributes are Identifier, in the form of a Handle, Date Last Modified, and Date Created. Future implementations could include additional DO level attributes, including user extensible attributes.

The intrinsic Element attributes are Identifier, in the form of a Handle, Date Last Modified, Date Created, and Size. The user extensible attributes may be set by users with appropriate permissions. Likely applications include data type and subject specific metadata.

Key attributes which are not specifically addressed by the basic model are ownership and permissions. These attributes will be an important part of most DO implementations, but we don't believe that there will be a single solution. Ownership and rights data will likely be contained in user extensible DO level attributes or in separate data elements.

3.3. Operation Specifications

The digital object operations referenced in this document must be fully specified and any changes to the operation specification must be fully backwards compatible. The protocol by which the operations are invoked is defined in the Repository Protocol Specification, version 1.0. The identifier of each operation as well as that operation's acceptable input, parameters, and expected output are fully specified below.

Note: The <message> structure from the protocol specification is used for representing structured information in the input and output for many of the operations.

Operation:: 1037/0
Description: List available operations. This operation is invoked to obtain a list of operations that can be performed on the target object by the caller.
Parameters: None
Input: None
Output: A UTF8 encoded newline delimited list of the operations that can be invoked on the target object

Operation: 1037/1
Description: Create digital object. This operation is only available on the repository/server object itself. Successfully invoking this object results in the creation of a new object on the target repository.
Parameters: objectid: (optional) the identifier of the object to be created.

initencoding: (optional) if the new object is to be initialized from a serialized object then this parameter must be present to specify the encoding in which the serialized object contents are provided. The currently supported encoding is "basiczip" in which the input is a zip-encoded byte stream with each entry in the zip file being one data element

Input: If the initencoding parameter is present then the input stream contains the serialized version of the object contents. If no initencoding parameter is present then no input is expected.

Output: A single <message> structure having a key *objectid* and corresponding value containing the ID of the newly created object.

Operation: 1037/5
Description: Get data element. This operation returns the bytes contained in a specified data element in the target object
Parameters: elementid: The name of the data element to be returned
Input: None
Output: The content of the requested data element

Operation: 1037/6
Description: Update data element
Parameters: elementid: the name of the data element to be updated
Input: The bytes to store in the named data element
Output: None

Operation: 1037/7
Description: Delete data element
Parameters: elementid: the name of the data element to be deleted from the object
Input: None
Output: None

Operation: 1037/8
Description: List data elements
Parameters: None
Input: None
Output: A series of <message> structures, delimited by newline characters. Each structure contains a key *elementid* that specifies the name of the data element. Additional keys and values, such as attributes, may be added to future versions.

Operation: 1037/10

Description: List objects. This operation is used to acquire a list of object in the repository and is therefore only available on the repository object.
Parameters: None
Input: None
Output: A series of <message> structures, delimited by newline characters. Each structure contains a key *objectid* that specifies an identifier for an object. Additional keys and values, such as attributes, may be added to future versions.

Operation: 1037/42
Description: Get user credentials. This operation returns a list of valid credentials (currently X509 certificates) that are stored in a digital object.
Parameters: None
Input: None
Output: An XML structure with a top level tag named <credentials> containing a series of sub-tags named <credential>. Each <credential> tag has a hex-encoded X509 certificate for its value. Additional tags and tag attributes may be added to future versions.

Operation: 1037/43
Description: Add user credentials. This operation is used to add an X509 certificate to an object's list of credentials. The X509 certificated provided as input must specify the target object as a principal in the certificate, not be expired and contain a valid signature.
Parameters: None
Input: An X509 Certificate.
Output: None

Operation: 1037/44
Description: Retrieve object transactions. This operation is currently only available for the repository object. It is used to retrieve a list of changes to objects since a given point in time. This operation is used for incrementally mirroring objects across multiple locations as well as (re)indexing any objects that have recently changed.
Parameters: txn_id: (optional) The identifier of the last transaction that the caller received. The response should include only transactions that have occurred after the given transaction ID. If no *txn_id* is provided then all transactions are returned.
Input: None
Output: A series of <message> structures, delimited by newline characters. Each structure contains the following values:

txn_type: the action that is represented by this transaction. This can be one of *add* (create object), *del* (delete object), *update_element*, *del_element*, *comment*, update_attribute or *delete_attribute*.

object_id: the identifier of the object that was updated in the transaction

tstamp: The transaction ID and transaction timestamp

elementid: (optional) the data element, if any that was affected by the action

atstamp: (optional) the actual time, in milliseconds since 1/1/1970 0:00.000 UTC that the action occurred. The *ts* value is used if this is not present. This value is used to resolve conflicts from concurrent updates to attributes, elements and objects. Transactions with earlier *atstamp* values than the data in the local storage are ignored.

att: (optional) the key value pairs of the attributes that were changed, if any

Operation: 1037/45
Description: Delete digital object.
Parameters: None
Input: None
Output: None

Operation: 1037/46
Description: Verify object existence
Parameters: None
Input: None
Output: A single <message> structure having a key *result* with a boolean value indicating whether or not the object exists within the server. Additional keys and values may be added to future versions

Operation: 1037/47
Description: Push object transaction. This operation applies only to the repository object. This is the "push" version of the 1037/44 (retrieve object transactions) operation. Note: This should be considered experimental as we may want to use a more application level approach for pushing object changes.
Parameters: object_id: the identifier of the object being updated

elementid: (optional) the name of the data element that is affected by the trasnsaction

txn_type: The type of modification. This can be any of the values from the *txn_type* record in the 1037/44 (retrieve object transactions) operation results.
Input: Whether there is input or not depends on the value of the *txn_type* parameter. When the *txn_type* is set to *update_element* then the input will contain the bytes that should be stored in the data element.
Output: None

Operation: 1037/48
Description: Retrieve serialized object.
Parameters: *encoding*: the name of the encoding used to serialize the object. The only currently supported value for this is "basiczip" which encodes the data elements as entries in the zip file format.
Input: None
Output: The serialized form of the object

Operation: 1037/49
Description: Set attributes in object or element
Parameters: *elementid*: (optional) the name of the data element to which the attributes apply. If no *elementid* is given, the attributes will apply to the target object itself.

Att: The attributes to be applied to the object or data element. These attributes are presented as a set of key value pairs in a <message> structure.
Input: None
Output: None

Operation: 1037/50
Description: Get attributes for object and/or elements
Parameters: *elementid*: (optional) the name of the data element for which the attributes are being requested. If no *elementid* is given, the object-level attributes will be returned.
Input: None
Output: If the *elementid* parameter was provided, the output will consist of a single <message> structure with message type *elementatts* and keys *elementid* and *att*. The value of the *elementid* key will be the name of the element to which the attributes apply. The value of the *att* key will be a <sub-message> containing the key-value pairs that make up the attributes of the element. If no *elementid* parameter was provided, the output will consist of a series of <message> structures, the first of which having message type *objatts* and one key *att* that has a value of a <sub-message> containing the key-value pairs that make up the attributes of the object. Subsequent <message> structures will contain the attributes for each element ID as defined above.

Operation: 1037/51
Description: Delete attributes for object or element
Parameters: *elementid:* (optional) the name of the data element from which the attributes should be deleted. If no *elementid* is given, the named attributes will be deleted

from the object itself.

att: A string array containing the attribute keys that are to be deleted

Input: None

Output: None

Operation: 1037/52

Description: Grant encryption key. This method is invoked on an object in order to "add" an encrypted bit of information to the target object. The encrypted bit of information is often an encryption key that has itself been encrypted with the public key associated with the target object. This operation will cause the given information to be added to an object in a new, uniquely named (within the target object) data element.

Parameters: None

Input: The encrypted information

Output: None

3.4. Depositing a Digital Object

Depositing a digital object is a matter of creating the digital object and then adding data to it. In most cases the creator of the object should also set object and data-level attributes to make it easier for entities that wish to interact with the object and the data elements within.

An object can be created by invoking the 'create object' operation on the repository. In this case we will register the object in the repository having identifier '1039/repository'. The create-object operation can therefore be described as:

1039/repository.1037/1() | <objectid>

Which indicates an invocation of the Create Digital Object operation (1037/1) on the given server object (1039/repository), with the result being stored in the variable <objectid>. The server takes care of calculating a unique object ID and registering that ID in the Handle System. The new <objectid> handle will contain a reference to the 1039/repository as the location of the digital object. The client also has the option of specifying the identifier for the new object by passing the objectid parameter in the operation invocation. Clients that supply the object ID are responsible for registering that identifier in the handle system so that it refers to the digital object server.

Now that the empty object exists, we can add data to it:

To store data in the data element named "content":

<data> | <objectid>.1037/6 (elementid=content)

To set the "mime-type" attribute for the new data element:

<objectid>.1037/49 (elementid=content&att.mime-type=text/plain)

To set the "creator" attribute for the object:

<objectid>.1037/49 (att.creator=myid)

3.5. Accessing the Document as a Digital Object

Accessing the digital object is similar to the deposit process. Accessing the information within the digital object is done by invoking the following operations:

To get the document content:

<objectid>.1037/5 (elementid=content) | <data>

To list the attributes for the object (which include the element attributes):

<objectid>.1037/50 | <documentattributes>