Network Working Group
Internet-Draft
Expires: 28 August 2002

Keith Moore
University of Tennessee
28 February 2002

The Resource Catalog

draft-ietf-rescap-rc-01.txt

Abstract

This memo describes version 3 of the Resource Catalog. The Resource Catalog (RC) is a service for storing and obtaining metadata that describes network-accessible resources. This service is primarily intended for use by clients prior to accessing, or attempting to access, a particular resource. The RC service may thus be used (for example) to inform a client as to which of a variety of features are supported by the resource, or which methods may be used to access the resource, or which of a small set of locations may be used to access the resource.

1. Introduction

This memo describes version 3 of the Resource Catalog. The Resource Catalog is a service for storing and obtaining metadata that describes

network-accessible resources. This service is primarily intended for use by clients prior to accessing, or attempting to access, a particular resource. The RC service may thus be used (for example) to inform a client as to which of a variety of features are supported by the resource, or which methods may be used to access the resource, or which of a small set of locations may be used to access the resource.

A 'network-accessible resource' is potentially any resource which is accessible on the network and which has a distinguished name (probably a URI) by which the metadata can be queried. Such resources could potentially include downloadable files, web pages, electronic mailboxes (including voice and fax mailboxes), instant messaging mailboxes, real-time text, voice, and/or video conferencing terminals, hosts, routers, etc. 'Metadata' are represented as name-value pairs, each of which describes some aspect of the resource.

NB: The above examples of resources are intended only as illustration of the flexibility of this approach. It is not expected that RC would, if adopted, be used in all of these contexts (or for that matter, in any particular context.) Each application or service using RC as a component would need to have an explicit definition for the metadata and specific use of RC within that application or service, prior to deployment of code.

Earlier versions of the Resource Catalog were designed for use in research projects dealing with replicating network-accessible content and distributed software repositories. This version of the Resource Catalog builds on experience with those earlier efforts, and is intended to be suitable for standardization and widespread deployment.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [1].

2. Design Goals and Implications

Given the overall intended purpose of the service as described above (and in the charter of the rescap working group) RC version 3 was designed with several specific goals in mind. These are reflected in the protocol design as follows:

2.1. Federation

Ideally, the owner of a resource being described should be able to control the metadata used to describe that resource within RC. Such control would be hampered if RC were a centralized service; this would also impact RC's scalability. It is therefore desirable for the owner to be able to control both the cost and the quality of the RC service

which describes his/her/its resources.

The RC service is therefore federated according to resource name; it is assumed that the network addresses of the servers that provide authoritative metadata describing a particular named resource are somehow obtainable from that name. RC uses NAPTR [2] and SRV [3] records to locate the RC servers for URIs which contain embedded DNS names. Similar mechanisms would need to be defined for other kinds of resource names (as for instance the URN working group is defining resolution mechanisms for URNs, and the ENUM working group is currently defining a means of finding metadata associated with telephone numbers).

This approach further allows the maintainer of a DNS zone to specify the locations of RC services for resource names for which the SRV records are located in that zone; thus, the RC servers corresponding to a particular resource name can be provided locally, outsourced, or some of each. This further implies that "RC service provider" and "resource owner" are different roles, and that the capability to manipulate resource attributes must therefore be delegated to the resource owner through a standard interface.

The need to be able to outsource RC services also implies that an RC query must self-contained, i.e. that no part of the question being asked is implicit in the choice of a server to which the question is being presented.

It is of course likely that the owner of a resource does not control all of the DNS zones which are traversed in the process of finding the RC servers for a resource name. And while it is hoped that DNS names will continue to be easily obtainable and inexpensive, some kinds of resource names (such as URNs) inevitably lessen owner control of the resource name in order to obtain increased stability of those names. For this reason, and because the RC service provider may be independent of the resource owner, RC provides the capability for resource owners to digitally sign any of the metadata associated with a resource.

2.2. Fast response to client queries

As RC is intended for use prior to accessing a resource, RC queries will tend to increase the overall time required to access that resource. It is therefore essential that RC provide fast response. This implies that server operations should be cheap in terms of CPU cycles. It further implies (in the case of low bandwidth links) that payloads and protocol overhead are small and (in the case of long delay links) that unnecessary round-trips are avoided.

To this end, RC queries are designed to run over UDP and to require a single UDP datagram in each direction. Servers MAY also support TCP as

a transport. RC is also optimized for small payloads, and it attempts to impose minimal protocol overhead. RC also provides a number of features designed to minimize the transmission of unnecessary response data, including the ability to query several named attributes in a single transaction, and the ability to specify whether signatures (and which types of signatures) should be returned with the response.

2.3. Scalability to large numbers of reading clients

If RC is used to provide essential metadata for heavily-used resources it potentially imposes a barrier to access of those resources. The scalability of access to those resources would thus be limited by the scalability of RC. It is therefore necessary that RC be scalable to handle large numbers of reading clients. RC attempts to achieve this by being implementable with minimal server overhead, and by allowing RC queries to be distributed over several replicated RC servers, with a well-defined model for consistency between servers.

2.4. Ease of implementation

In order to allow for quick deployment, the RC protocol is designed to be simple and easy to implement, and for simple implementations to perform acceptably on modern CPUs with moderate workloads. However it is naturally expected that high performance implementations will be more complex and more difficult to implement.

2.5. Security

RC is designed with consideration of the following security threats:

- Attempts by unauthorized parties to write metadata to servers.

  RC servers require writers (and optionally readers) to authenticate themselves via secure means; such requests will not be processed unless the authentication credentials can be verified.

- Attempts to obtain authentication credentials via the network, via active or passive attacks.

  The RC protocol allows for multiple authentication methods in order to accommodate the needs of different environments, and to accommodate additional authentication methods which might be defined in the future.

- Attempts to replay a previously successful write request for the purpose of modifying metadata in unauthorized ways.

Requests to update metadata include a serial number which must always be larger than the serial number of the previous request to update the metadata for that resource. This serial number is authenticated and checked for integrity along with the rest of the request. Attempts to replay write requests can therefore be detected, and unauthorized changes to the metadata avoided.

- Attempts to add, delete, or alter data in a response

RC allows (but does not require) writers to cryptographically sign metadata that are stored in a server. Reading clients may (but are not required to) request such signatures, attempt to verify them, and determine whether the party signing the metadata is authorized to make such assertions about the resource. However such signatures are opaque to RC servers; the RC protocol does not attempt to verify signatures that are stored with a resource.

When TLS [4] is used, the integrity protection provided by TLS will allow a client to detect responses altered in transit. However there is no requirement that TLS be used; furthermore, the protocol setup and server CPU overhead associated with TLS serve as disincentives to use of TLS.

- Removal of signatures from responses

When TLS is used, the integrity protection provided by TLS will allow a client to detect responses altered in transit. However there is no requirement that TLS be used and there are disincentives to use TLS.

NOTE: The protocol described in this document, could, with a very small modification, allow clients to request server-provided signatures on responses. Such signatures would not authenticate the response as coming from an authoritative source, but would serve to thwart this and other attacks that alter responses. If used frequently, such a facility would greatly increase server load, but probably no worse than if TLS were used frequently. On the other hand, to be effective this would also require that the client have a secure means of knowing that a server signature on the response should be expected, and a means of verifying that signature.

RC does not attempt to ameliorate the following threats:

- Attempts to direct clients to send queries to unauthorized servers

  Even if unauthorized servers contain data which are signed by authoritative sources, the data provided by unauthorized servers may be stale or obsolete.

  When DNS SRV and address records are used to direct queries to RC servers, DNSSEC may be useful in authenticating such records to clients.

  NOTE: TLS server certificates could be used by the client to determine whether the server can authentically claim to operate on behalf of a particular DNS name. However, authority to operate on behalf of a DNS name does not necessarily imply authority to supply information about a resource name, particularly when the server operator may not be authoritative to make assertions about a particular resource, even if the server is the correct one to consult for information about that resource. (e.g. the RC service might be outsourced)

- Repudiation of unsigned assertions

- Alteration of unsigned responses

- Threats which require physical access to server hardware

- Attempts to interfere with service by interfering with the transmission of requests or responses over the network

- Attempts to interfere with service by flooding the server with requests

2.6. Provision for proxies

Due to the large number of environments which separate local users from the public Internet with firewalls, and the anticipated use of this protocol to describe public Internet resources, it is essential that RC be usable through intermediaries.

2.7. Flexibility

RC is intended as a framework which can be used by a wide variety of applications and services, to store information about many different kinds of resources. The protocol therefore needs to be flexible enough to accommodate different applications' needs regarding (for instance) metadata size, rates of change, and security.

The feature-set included in RCv3 was selected based on observations from other information query protocols which have been in use for many years, as well as experience with earlier versions of RC.

DNS Though the resemblance may not be obvious, in many ways RC is heavily influenced by DNS [5]. Like DNS, RC is federated, handles multiple query types, and provides the ability to return "additional information" in response to a query. In addition, both RC and DNS provide signatures and the capability to support multiple signature types. RC and DNS have similar models for replication, with the RC replication granularity being that of a resource name and the DNS replication granularity being that of a zone.

However, RC attempts to generalize a number of DNS features. In particular, RC servers have no knowledge of the characteristics of attribute names (corresponding to DNS resource record types and query types). RC servers determine whether to perform "additional information" lookups based on explicit instructions in the query, rather than based on knowledge of the attribute name. Neither do RC servers have knowledge of signature types; signatures are opaque to the servers. Server processing of signatures in RC is limited to storing signatures that are supplied by writers and returning signatures to readers who ask for them. Finally, RC decouples the notion of "server ownership" from "resource name ownership" - the owner of the server is not inherently considered an authoritative source of information about any particular resource.

SNMP In some ways the service provided by RC is similar to that of an SNMP [6] agent. In particular, RC borrows from SNMP the capability to query aggregates of individually addressable resource attributes, where the aggregates are based on the structure of the names. However whereas SNMP uses ASN.1 object-identifiers (OIDs) for attribute names, RC uses human-readable ASCII strings. RC also separates resource names from attribute names rather than combining the two into a single name. Finally, RC lacks an analog to SNMP's trap mechanism, RC's information model is simpler than SNMP's SMI, and RC uses a simple presentation layer and RPC mechanism in contrast to SNMP's ASN.1 PDUs.

LDAP RC is heavily influenced by LDAP [7] in that RC avoids many of the design choices in LDAP which impose a barrier to the use of LDAP in applications for which RC is designed. In particular,

- RC allows the use of UDP to avoid the delay and bandwidth consumption associated with TCP connection setup

- RC uses a simpler data model than ASN.1, and a simpler presentation encoding than BER, for ease of implementation and improved processing efficiency

- RC provides lookup by resource name, but not searching of contents, for server efficiency and scalability

- RC provides the ability to query for and return "additional information" based on results of the initial query, thus avoiding multiple round-trips

- RC has support for caching of data returned from queries

- RC defines a model for consistency between replicas which is usable by applications

3. Data Model

3.1 Format of resource descriptions

RC models the description of a resource (its "metadata") as a set of (name, value) pairs called assertions. Each assertion consists of:

- an attribute name, which is a human-readable character string;

- a value, which is often human-readable, but can potentially be any sequence of octets;

- an optional time-to-live; and

- an optional expiration-date.

The time-to-live is some number of seconds from the present time during which the assertion is expected to remain valid. The expiration-date is an absolute time at which the assertion will no longer be valid. Both time-to-live and expiration-date may be supplied, in which case both limitations to the validity of the assertion apply. This allows an assertion to expire "gracefully" at a particular time without the server having to explicitly decrease times-to-live as the expiration time approaches; it also allows the expiration time to meaningful in a signed assertion.

3.1 Caching

Intermediaries (e.g. proxies) MAY cache assertions and return cached assertions in response to queries for specific resource_names during the expected lifetime of the assertions. However, all cached assertions about a resource_name which are returned in a query response MUST have

been obtained from the same version of the description of that resource. Intermediaries MUST NOT return cached responses to wildcard queries unless those responses were returned from an earlier wildcard query of greater or equal scope (i.e. a shorter prefix of the resource name) than the current query.

3.2 Consistency between replicas

RC services for a particular portion of URI-space MAY be replicated so that queries, updates, or other operations within that portion of URI-space can be issued to any of the replicas. A standard mechanism for replication between servers is not yet defined, but individual implementations MAY define their own mechanisms for doing so.

When the description of a resource is replicated between RC servers, the servers MUST maintain the appearance of a linear change history for the entire resource description across all of the servers. In other words, it is not acceptable for separate changes to be made to different replicas of the resource description. Servers MAY impose requirements on writers in order to facilitate this (e.g. they may require that all updates be sent to a single master server).

Replicated RC servers SHOULD make reasonable efforts to be synchronized with one another.

3.3 Permission model

RC is intended for the dissemination of public information. It is possible for servers to refuse to answer queries sent by non-authenticated parties or to specific parties who authenticate themselves, and it is possible for servers to omit information from the responses of queries sent by unauthorized parties. Until the effect of such restrictions on caching is understood, servers SHOULD NOT restrict the ability of parties to query information.

RC servers which support the Update operation MUST be able to prevent unauthorized parties from updating information about a particular resource; however, the granularity with which this is done is not currently defined. Servers MAY restrict the ability to update particular resource names, as well as particular attribute names. However, perhaps unfortunately, the currently defined status reporting mechanism provides little information about why an update attempt failed.

4. Protocol

The protocol is defined in terms of RPC operations. Each operation is defined in terms of an request structure, result structure, and side-

effects that take place when a valid request is received. The structures are defined using the example abstract syntax from the Binary Low-Overhead Block (BLOB) protocol (see [8], Appendix B), and BLOB is used as a presentation format.

The first scalar integer of each request is used as a request_type. This allows different types of requests to be distinguished from one another. The first string of each request is a request_id which is used to associate responses with requests.

The first scalar string of each response is the request_id from the request. The first scalar integer of each response is the status of the request.

4.1 Protocol Evolution and Version Negotiation

For most purposes, RC is designed to be used in a stateless manner, without the need for an explicit "connection setup". Explicit negotiation of the RC protocol version or features between client and server would increase the time required to perform RC operations. To the extent feasible, new versions of RC should continue to support request_types from previous versions of the protocol. If it is important to make clients aware of new server features, it might be preferable to advertise such information to clients via SRV or NAPTR DNS records rather than to force the client to negotiate a protocol version on each interaction with an RC server.

However, the request_type field of each request can be considered to contain an implicit version number. If it becomes necessary for future versions of RC to deprecate existing request_types, servers can return a RC_PROTO_MISMATCH response to any unsupported request_type. This response contains both protocol version information and a list of supported request_types.

Request_type 0 is reserved and can be used by a client to force an Unsupported Request response to be returned.

4.1.1 Unsupported request response structure

The Unsupported Request response structure is as follows:

```
BEGIN unsupported_request_response
string request_id
int status
int protocol_version
int<> supported_requests
END
```

This structure is returned in response to any request with an unsupported request_type.

request_id
: This is the request_id from the request.

status
: This will normally be RC_UNSUPP_REQUEST.

protocol_version
: This is the highest version of the RC protocol supported by the server. This document describes version 3.

supported_requests
: This is an array of integers indicating the range of request_types supported. The integers are in pairs indicating higher and lower bounds of a range of supported request_types. For instance, an integer array [1 4 6 6 10 12] would indicate that request_types 1-4 inclusive, 6, and 10-12 inclusive were the only ones supported.

  Note on caching of supported request_types: In principle servers can be up- or downgraded at any time and with no notice. It is therefore not safe for a client to assume that the set of request_types supported by a server will remain constant. Clients MUST therefore be prepared to handle a RC_UNSUPP_REQUEST in response to any request which is not required by this specification. For a client which can utilize additional request_types if they are supported by the server, a cached list of request_types may prove to be a useful performance optimization. In order to allow clients to discover new server request_types in a timely fashion, clients which can utilize additional request_types (beyond those which are required of servers) SHOULD discard cached request_types after a suitable interval, say a few hours.

4.2 Operations

The following operations are defined:

4.2.1 Query operation

The Query operation returns information describing the named resource.

4.2.1.1 Query request structure

The Query request structure is defined as follows:

```
BEGIN query_request
int request_type              # QUERY=1
string request_id
string resource_name
string<> attribute_names
int<> attribute_flags
# QUERY_RECURSE = 01
# WANT_SIGNATURES = 02
int<> signature_types
END
```

The meanings of the individual fields are outlined below.

request_type

A request_type of 1 indicates a Query request.

request_id

The request_id is used to associate the response with the request. This is a string which is chosen to be unique for the client's IP address and port pair, for any amount of time during which any request from that host and port pair is likely to be pending or during which a response may be traveling through the network. It is RECOMMENDED that request_ids not be reused within at least an hour.

The contents of the request_id are otherwise undefined; client implementations MAY generate them however they wish. (however see the Security Considerations section).

resource_name

The name of the resource for which information is being requested; for instance, a URI. This field MUST be present.

attribute_names, attribute_flags

These fields are parallel arrays. attribute_names specifies the names of the attributes which are being requested for the resource, while attribute_flags specifies optional query features desired for each attribute_name. attribute_flags[i] thus specifies the optional query features for attribute_names[i]. These two arrays MUST have the same number of elements, and servers MUST return an RC_DATA_FMT error in response to a Query operation for which the number of elements in these two arrays is different.

Currently defined attribute_flags are as follows:

- The QUERY_RECURSE bit controls searching for additional information; see section 4.2.1.6 below.

- The WANT_SIGNATURES bit requests signatures for this attribute; see section 4.2.1.7 for a discussion of signatures.

These flag bits can be ORed together to request both features.

Note: as hinted in section 2.5, a WANT_SERVER_SIGNATURE bit could be added here to request that the server sign the response. This would provide some protection against attacks that modify the response but would not by itself guarantee that the response came from an authoritative party.

An attribute_name that ends in "*" is a wildcard; see section 4.2.1.8.

signature_types
: The list of signature_types which are understood by the caller. Values for individual signature_types are to be defined.

4.2.1.2 Query result structure

The Query result structure is as follows:

```
BEGIN query_response
string request_id
struct<> answers
END
```

request_id
: This is the request_id from the Query structure, used to associate the response with the request.

answers
: The list of answers. Each answer contains the results of the query for a particular resource_name. The first answer is the one for the resource_name which was explicitly part of the Query structure. Any additional answers are the result of additional queries requested via the QUERY_RECURSE flag.

4.2.1.3 Answer structure

Each Answer is formatted as a blob using the following structure:

```
BEGIN answer
string resource_name
int query_status
int version_hi
int version_lo
struct<> assertions
struct<> signatures
END
```

Within an answer are the following fields:

resource_name
: The resource_name for which the following assertions are made.

query_status
: A code describing the result of the attempt to query this resource_name. See section 3.5.

version_hi, version_lo
: An 8-byte version number (treated as a single 8-byte integer, but represented for the purpose of presentation as two 4-byte integers) for the resource information. The version number increases (not necessarily by 1) each time the resource is modified.

assertions
: This is an array of blobs, each of which contains information about the named resource. The structure of an assertion is defined below.

signatures
: This is an array of blobs, each of which contains a signature over some set of assertions. The structure of signatures is defined below.

4.2.1.4 Assertion structure

An assertion has the following structure:

```
BEGIN assertion
string name
string value
int ttl
int expire_date_lo
int expire_date_hi
int flags
END
```

name This is the name of the attribute.

value
: This is the value associated with the attribute name.

ttl, expire_date_*
: The ttl and expire_date describe the lifetime during which the information is expected to be valid (and during which it may be cached by intermediaries).

  The ttl field is the number of seconds during which the assertion is expected to be valid; if the ttl field is 0x7fffffff (2**31 - 1) then no TTL is assumed.

  The expire_date_* fields contain an absolute expiration date. expire_date_hi contains the day of expiration, expressed as the number of whole days (60*60*24 second periods) since Jan 1, 1970 UTC. expire_date_lo contains the number of seconds within the day following the number of days in expire_date_hi. Any leap days which have occurred or will occur prior to the expiration date are included in expire_date_hi. Any leap seconds which have occurred or will occur prior to the expiration date are included in expire_date_lo. Note that expire_date_lo can therefore be greater than 86400 seconds. However most applications will not need that degree of precision, and most applications cannot count on having their clocks synchronized to that degree of precision. An application supplying expiration dates to RC MAY fail to consider leap seconds in its calculations if the loss of precision is not considered significant for that application. If expire_date_hi and expire_date_lo are both zero, no expiration date is assumed.

flags
: These are single-bit characteristics of the attribute. They are for future use.

4.2.1.5 Signature structure

Signatures which can be used to verify authenticity and integrity of the assertions. Each signature is a blob formatted according to the following structure:

```
BEGIN signature
int<> component_list
int algorithm
struct bits
END
```

component_list
: This is an ordered list of the indices, relative to the start of the assertions array from the answer, of the assertions that are signed by this signature. The first assertion of the answer is always index 0, even even if the implementation language uses one-based arrays.

  Note that the contents of the assertions are signed, and the indices are NOT signed. The indices have no significance outside of a particular Answer to a Query.

algorithm
: This is an integer which specifies both the method for translating a resource_name, version_hi, version_lo, and an ordered list of assertions into a single linear octet string, and the digital signature algorithm used to create and verify the signature.

bits
: This is the actual signature, formatted according to the algorithm defined by the signature algorithm.

A signature is verified by concatenating the names and values of the assertions in a manner specified by the signature algorithm, and then verifying the result of that concatenation against the bits field, again according to the manner specified by the signature algorithm. NOTE: ttl fields of assertions are NOT considered in signature calculations.

4.2.1.6 Recursive Processing

If the QUERY_RECURSE bit of attribute_flags[i] is set, it signifies that if any attribute_values matching attribute_names[i] are found, they are to be interpreted as resource_names, and additional queries (using the same set of attribute_names and attribute_flags as the original query) are performed for those resource names. The server may decline to service such "additional information" requests for any reason, including for example that the server does not have authoritative information for

the resource name, or the query is being made by UDP and the result does not fit within a reasonable UDP packet size, or processing of the additional query would likely cause the caller's retransmission timer to expire.

It is possible that an attribute_value of some answer will contain a resource_name for which results have already been obtained while processing this request (either because the resource_name was explicitly listed in the query or because it appeared in the result of a field for which QUERY_RECURSE was set). Servers MUST NOT return multiple sets of answers for the same resource_name in the response to a single Query operation.

4.2.1.7 Signature Processing

When any of the attributes in a Query operation has an attribute_flags field with the WANT_SIGNATURES bit set, the server will return, along with the attribute information, any signatures which cover the attribute and which are included in the list of signature_types that the client understands. If those signatures include other attributes which are not requested, those attributes are also included in the response. Thus a response may include more attributes than requested.

If the list of signature_types is empty but WANT_SIGNATURES is set for any attribute, the server will return all available signatures for that attribute.

A server MAY omit signatures (and the attributes needed to verify those signatures) if including them would make the result too large for transmission over UDP; however, inclusion of signatures takes precedence over QUERY_RECURSE processing.

4.2.1.8 Wildcard attribute names

For the purposes of a query, an attribute_name ending in the character '*' (ASCII 2A hex) causes information to be returned for each of the attribute_names in the resource which begin with the portion of the attribute_name prior to the '*'. The character '*' MUST NOT appear in an attribute_name parameter of a Query operation except as the last character. Issuing a query for an attribute_name of "*" causes all attributes to be returned.

4.2.2 Update operation

Note: the rescap charter states that the update protocol is part of the "second task" of the group's activity which has not yet begun. A sample update protocol is included here only for the purpose of illustrating how Query and Update might work together. This part of the

specification is therefore, for the time being, incomplete.

The Update operation may be used to add attributes, change the values associated with attributes, or delete attributes from the set of attributes associated with a URI. The Update protocol may also be used to add or delete signatures to sets of attributes. Servers SHOULD support the Update protocol, but MAY provide alternate means to perform these operations than via the Update protocol.

4.2.2.1 Update request structure

The Update request structure is as follows:

```
BEGIN update_request
int request_type         # UPDATE = 2
string request_id
int version_hi
int version_lo
string resource_name
int flags
# UPDATE_CREATE_NEW = 01
# UPDATE_VERSION_MATCH = 02
# UPDATE_CLOBBER_SIGS = 04
struct<> assertions
struct<> signatures
END
```

request_type
: For an Update operation, the request_type is 2.

request_id
: This is used to associate the response with the request. See the definition of request_id for the Query operation.

version_hi, version_lo
: These is the version number which the client expects corresponds to the server's current version number for that resource. If the UPDATE_VERSION_MATCH flag is set, this version number is compared with the server's version number. Otherwise, version_hi and version_lo are ignored.

  The server will increase the version number (as represented by version_hi and version_lo), with each successful Update operation, by a small positive integer, using addition modulo 2**64. The amount added to the version number need not be a constant, but MUST be sufficiently small that no version number will be reused (due to wrap-around) within ten years.

Clients MUST NOT use UPDATE_VERSION_MATCH to update portions of a resource's metadata if the version of the record they have previously obtained is more than than a year old or is of uncertain age. In such cases it is necessary to first perform a Query operation and compare the actual data (not merely the version fields) to determine whether the data has been changed.

resource_name

The resource name for which attributes will be updated.

update_flags

If the UPDATE_CREATE_NEW flag is set, and no record for resource_name exists, this requests that a new record be created.

If the UPDATE_VERSION_MATCH flag is set, the update fails with status VERSION_MISMATCH unless the version_hi and version_lo field of the Update request match the version_hi and version_lo fields of the existing record. A nonexistent record (before it is created) is treated as if version_hi and version_lo were set to zero.

If the UPDATE_CLOBBER_SIGS flag is not set, and the update would modify some attributes covered by a signature without modifying all of those attributes, the operation fails with status WOULD_CLOBBER_SIGS. Setting the UPDATE_CLOBBER_SIGS flag causes the signature to be deleted if the update would otherwise be successful.

assertions

This is the list of assertions about resource_name to be added to, changed, or deleted, formatted according to the "assertions" structure defined above. The assertions are updated as follows:

- If there is no assertion matching the attribute_name, the assertion from the Update request is added

- If there is already an assertion matching the attribute_name, the assertion from the Update request replaces the previous one from that principal. however, if the ttl field in the Update request is 0, the previous assertion is deleted.

- If the attribute_name ends in '*', the Update operation applies to all attributes beginning with the prefix of the attribute_name. However, this can only be used to delete old assertions (by setting ttl=0) or to update their ttls (by setting ttl to something other than 0) and expiration dates (by setting these to something other than 0). Thus it is possible to update the ttl and/or expiration date of all assertions associated with a resource_name in a single Update operation.

signatures
: The signatures included in the update. Each of the components of the signature (in component_list) is an index into the assertions array. Signatures can only be provide for assertions included in the Update operation; if it is needed to sign assertions that are already included in the record, they must first be obtained using a Query option and then the signatures supplied using an Update operation (probably with the MATCH_VERSION option set).

4.2.2.2 Update response structure

The Update response structure is as follows:

```
BEGIN update_response
string request_id
int status
END
```

The update response is a single status code. The entire Update operation either succeeds or fails atomically. Keeping this structure simple makes it easier to handle retransmitted Update requests without performing multiple Updates - the server only has to remember the status of the most recently transmitted request, for that resource_name, from any party authorized to Update that resource.

4.2.3 One-Shot Authenticate operation

The One-Shot Authenticate operation is used to authenticate another request via a lightweight mechanism. It is intended for use with requests (such as Update) which require authentication, when the enclosing transport does not provide sufficient authentication. However, it MAY be used over any transport, including bare TCP or UDP.

The One-Shot Authenticate operation only authenticates a single operation; it DOES NOT establish authentication for subsequent operations between those parties.

4.2.3.1 One-Shot Authenticate request structure

The Authenticate request structure is as follows:

```
BEGIN one_shot_authenticate_request
int request_type    # ONE_SHOT_AUTHENTICATE = 3
string request_id
string authentication_type
struct authentication_credentials
int serial_number_hi
int serial_number_lo
struct inner_request
END
```

request_type
: For One-Shot Authenticate, the request number is 3.

request_id
: See request_id as described for the Query operation.

authentication_type
: The type of one-shot authentication used. These are to be determined. Possible authentication types would include digital signature formats based on public key cryptography and keyed-hash functions using shared secrets.

authentication_credentials
: This is an opaque data structure, specific to authentication_type, which is intended to assure the server of the client's identity and to provide integrity protection for the serial_number and the inner_request.

serial_number_hi, serial_number_lo
: This is a serial number, represented as two 4-byte integers. It is used to thwart replay attacks. Each request from a particular principal must have a greater serial number than the previous one. If a serial number is repeated, and the request is otherwise valid, the request will be processed in such a manner as to neither alter information on the server nor to return potentially sensitive information in the result. However, for an Update request sent over UDP, it is appropriate to return the status of the previous operation, in case the duplicate request was caused by a legitimate retransmission.

4.2.3.2 One-Shot Authenticate response structure

The One-Shot Authenticate response structure is as follows:

```
BEGIN one_shot_authenticate_response
string request_id
int status
struct inner_response
END
```

request_id
: See request_id as described for the Query operation.

status
: A code giving the result of the One-Shot Authenticate operation, not that of the inner request. If the status code for the One-Shot Authenticate operation indicates failure, the inner_response field will be zero length. If the status code for the One-Shot Authenticate operation indicates success, the inner_response field will have a status code indicating the status of the authenticated request.

inner_response
: This is a BLOB-encoded field containing the server's response to the authenticated request, if the authentication was successful.

4.2.4 SASL Authenticate operation

The SASL Authenticate operation is used to authenticate the connection using SASL [9] mechanisms. Unlike the One-Shot Authenticate operation, the SASL Authenticate operation establishes authentication credentials for subsequent operations over the same TCP or TLS connection; and unlike the One-Shot Authenticate operation, the SASL Authenticate operation does NOT provide integrity protection for subsequent requests or responses. The SASL Authenticate operation MUST NOT be supported over UDP.

4.2.4.1 SASL Authenticate request structure

The SASL Authenticate request structure is as follows:

```
BEGIN sasl_authenticate_request
int request_type    # SASL_AUTHENTICATE = 5
string request_id
string mechanism
int client_state
string response
END
```

request_type
: For SASL Authenticate, the request type is 5.

request_id
: See request_id as described for the Query operation. Each SASL Authenticate request gets a new request_id, even if the request is to respond to a challenge resulting from a previous SASL Authenticate request.

mechanism
: An ASCII string giving the registered name of the SASL mechanism.

client_state
: An integer giving the type of SASL request:

  0. Initial SASL Request. This state code is used to indicate an initial SASL request. In this case the client_response field MAY be used to indicate the first SASL authentication message for SASL methods where the client is the first party to send authentication data.

  1. Response to SASL Challenge. This state code is used when the client wishes to respond to a challenge that was issued by the server in the immediately previous SASL Authenticate Response message.

  2. Abort SASL Request. This state code is used when the client wishes to abandon an attempt to authenticate using SASL, or when the client wishes to revoke SASL authentication that is currently in effect. (note that the server MAY be unable to accept a subsequent SASL authentication on the same TCP connection).

response
: An octet string giving the client's response to a challenge (if

state == 1), or the client's initial response (if state == 0) for SASL methods which are defined to have the client be the first party to send authentication data.

4.2.4.2 SASL Authenticate response structure

The SASL Authenticate response structure is as follows:

```
BEGIN sasl_authenticate_response
string request_id
int status
int server_state
string challenge
END
```

request_id
: The request_id of the request.

status
: The status code resulting from the attempted operation:

RC_SUCCESS
: Authentication completed successfully. server_state will be set to 2.

RC_AUTH_UNSUPP
: Authentication failed because the SASL mechanism requested by the client is not supported by the server. server_state will be set to 0.

RC_AUTH_REQUIRES_ENCRYPTION
: Authentication failed because the client attempted to use a SASL mechanism that requires strong encryption, without having strong encryption in place. server_state will be set to 0.

RC_AUTH_CHALLENGE
: Authentication requires a response to a challenge issued by the server. server_state will be set to 1.

RC_TEMPORARY_FAILURE
: Authentication failed for temporary reasons. server_state will be set to 0.

other code
: Authentication failed. server_state will be set to either 0 or 3.

server_state
One of the following:

0. No SASL authentication is in effect or in progress.

1. SASL authentication is in progress; a response to a challenge is expected.

2. SASL authentication is in effect.

3. No SASL authentication is in effect or in progress, but the server is unable to accept further SASL authentication requests during this session. The client must close the current TCP or TLS connection and reopen a new connection to the server before a SASL authentication can be successful.

4.2.4.3 SASL Profile

The following profile is specified per [9]:

1. The SASL service name is {TBA IANA, presumably "rc"}.

2. The command to initiate the protocol exchange is the SASL Authenticate request with a client_state value of 0.

3. The authentication protocol exchange is carried out by having the client issue a series of SASL Authenticate requests. The first request (the one used to initiate the exchange) has a client_state value of 0. The server issues a response to each request. Depending on the value of the server_state field of the response, the client may need to issue additional SASL Authenticate requests in order to supply responses to server challenges. This repeats until the server either signals a success status code (with server_state set to 2) or a failure status code (with server_state set to 0). The challenges and responses are supplied in appropriate fields of the SASL Authenticate response and request structures, respectively, and are NOT encoded except for the BLOB encapsulation specified by the RC protocol.

4. The negotiated security layer takes effect immediately following the client's transmission of a SASL Authenticate Request structure, and the server's transmission of a SASL Authenticate Response structure for which (status = RC_SUCCESS) AND (server_state = 2). Once the client has sent a SASL Authenticate Request, the client MUST NOT transmit additional requests other than SASL Authenticate over that connection until the server has returned a SASL Authenticate response for which

server_state is not equal to 1. A SASL Authenticate request with a client_state of 2 may be used to abort a SASL exchange and cause the server to reset its server_state to 0.

5. The authorization identity that is passed to the server is used by the server, along with other configuration and permissions information, to determine whether the client is authorized to perform a particular operation. For now, the method by which the server makes such a determination is undefined by the RC protocol.

4.2.4.4 Notes on use of SASL

Due to implementation constraints, some servers may be unable to accept more than one SASL authentication per TCP or TLS connection, even if the original authentication is revoked.

Attempts to use SASL mechanisms which transmit shared secrets (e.g. passwords) MUST be rejected by the server unless there is an established TLS connection using a ciphersuite which provides encryption with an effective key strength (for the symmetric encryption key) of at least 112 bits.

4.3 Transport

4.3.1 Use over TCP and UDP

An RC server listens for requests on UDP port {TBD-IANA}. When used over UDP, each request or response is entirely contained within a single UDP datagram. The response is sent to the same address and port number that appeared in the IP header of the request. The request or response datagrams MAY be fragmented into multiple IP packets; so support for reassembly is required. The maximum possible request or response is constrained by the maximum UDP datagram size. Clients SHOULD NOT use UDP for a request when the response is expected to contain a large amount of data. Servers SHOULD have code to limit the amount of additional information returned in a UDP response to some reasonable figure. Servers MUST detect when a response would be too large (either larger than the maximum UDP datagram size or larger than some smaller administrative limit) and return an appropriate error indication.

UDP clients MUST implement a retransmission timer to keep from swamping the network - details are to be determined.

An RC server that supports TCP listens for requests on port {TBD-IANA}. The responses are returned over the same connection. Multiple requests and responses may be sent over a single TCP connection. The requests and responses are self-delimiting, each beginning with a 4-octet integer

length field (in network byte order). Multiple requests MAY be issued before reading a response, but this introduces the possibility of deadlock. A client that issues multiple requests MUST be able to read responses to earlier requests concurrently while issuing requests. When multiple outstanding requests are issued over a single TCP connection, servers MAY service them out of order, so long as no request is significantly delayed. Servers MAY place a limit on the number of outstanding requests from a single source, in which case they MUST respond to requests in excess of that limit with a {TBD} error.

NB: the above needs work. Since a response may be of arbitrary size, a server might be unable to return "too many outstanding requests" because a large response is blocking the return channel.

4.3.2 Use over TLS

Clients and servers MAY support TLS. TLS is layered on top of TCP, and an RC server which supports TLS uses the same port for RC-over-TLS-over-TCP as for RC-over-TCP. A client wishing to use TLS to communicate with a server first opens up a TCP connection to that server, and then issues a StartTLS request.

4.3.2.1 StartTLS request structure

A StartTLS request structure is defined as follows:

```
BEGIN starttls_request
int request_type    # STARTTLS = 4
string request_id
END
```

4.3.2.2 StartTLS response structure

The StartTLS response structure is:

```
BEGIN starttls_response
string request_id
int status
END
```

Servers which support TLS MUST support ciphersuite {TBD}. The appropriate use of TLS server certificates by clients, and the use of TLS client certificates by servers, is yet to be defined.

4.4 Mapping from URIs to RC server locations

The specifics of this are currently undefined, but are intended to be similar to the procedures defined in [11], [2], and [12], and consistent with any standards developed in this area.

4.5 Operation Through Proxies

RC clients MAY make queries via a proxy. Such proxies MAY cache successful results (up until their ttls and or expiration dates expire) and return them in response to subsequent Queries. Unsuccessful results MUST NOT be cached. Proxies MAY supply their own request_id fields of requests passed to servers, in order to better facilitate demultiplexing and routing of the responses to their clients, so long as they also modify the request_id fields of responses returned to the clients to match the request_id fields used by those clients.

4.6 Status Codes

RC_SUCCESS (0)
: successful operation

RC_NO_SUCH_NAME (1)
: operation failed; no record for this resource_name was found on this server

RC_NOT_AUTHORITATIVE (2)
: operation succeeded, but the answer returned is not authoritative

RC_RESULT_MISSING_SIGS (3)
: operation succeeded, but signatures could not be included in the response due to space limitations

RC_VERSION_MISMATCH (4)
: operation failed because VERSION_MATCH was set in the request but the version numbers did not match. (Note: This MUST NOT be used to signal a mismatch between versions of the RC protocol.)

RC_TEMPORARY_FAILURE (5)
: operation failed due to unspecified temporary conditions

RC_WOULD_CLOBBER_SIGS (6)
: operation failed because the Update would result in invalidation of existing signatures, and the DONT_CLOBBER_SIGS flag was set

RC_KEY_SYNTAX (7)
: operation failed; the syntax of the resource name was not valid

RC_CRED_VRFY (8)
operation failed, could not verify credentials

RC_CRED_REVOKED (9)
operation failed, credentials have been revoked or have expired

RC_NOPERM (10)
client does not have permission to perform the operation

RC_DATA_FMT (11)
the request was not in the proper format or could not be parsed

RC_REFUSED (12)
query refused

RC_AUTH_INSUFF (13)
The authentication method chosen by the client is not sufficiently secure for this operation; the actual operation was not attempted.

RC_AUTH_UNSUPP (14)
The authentication method chosen by the client is not supported by the server.

RC_UNSUPP_REQUEST(15)
The request_type is not supported by this server, either because it is deprecated or because it is not yet defined.

RC_AUTH_CHALLENGE (16)
This is a response to a SASL Authenticate request, indicating that a response to a challenge is necessary before authentication can be completed.

RC_AUTH_REQUIRES_ENCRYPTION (17)
The authentication failed because the client attempted to use an authentication method that requires strong encryption, without having strong encryption in place.

5. Conventions for attribute names

Attribute names are human-meaningful ASCII strings consisting of printable characters. They are chosen in such a way that wildcard queries are useful. Related attributes are grouped together by having a common prefix. Thus for example a set of attributes related to electronic mail might be grouped together with the prefix "rc:email.", and queries for "rc:email.*" might be issued to obtain all such attributes.

RC is designed to be mostly agnostic about the format of attribute names. Some existing data models use Uniform Resource Identifiers as attribute names. This seems unwise - not only do URIs tend to be more verbose than necessary, but their use as attribute names seems likely to encourage inappropriate semantic association between the URI components (prefix/protocol, domain name, etc) and the attribute named by the URI.

However, for the sake of consistency with existing practice, URIs MAY be used as attribute names in RC - subject only to the requirement that RC performs exact (octet-by-octet) comparison of attribute names. URIs used as attribute names in RC MUST be canonicalized so that the URI always has a consistent representation. The algorithm by which this canonicalization is done is not defined by the RC protocol; it MUST be specified by the higher-level protocol.

It is intended that a registry of attribute names be maintained by IANA in order to discourage conflicting uses of the same names. Assignment of registry names is via the IETF Consensus process or with IESG approval. Prefixes MAY be delegated to other parties via the IETF Consensus process.

The prefix "rc:" is reserved for attributes used by RC itself. The prefix ":x:" is reserved for ad hoc user extensions.

5.1. Inheritance of attributes

The rescap charter specifies a requirement that it must be possible to inherit attributes. In RC this is accomplished as follows: the attribute_name "rc:defaults" is defined. The value associated with this name should be a URI. If an attribute named "rc:defaults" exists for a particular resource_name, the default attributes for that resource_name may be obtained by issuing a query for the same attribute_names to the URI associated with "rc:defaults".

The defaults may thus easily be obtained by including the attribute_name "rc:defaults" in the set of attributes requested, and by setting the QUERY_RECURSE flag for that attribute_name. Multiple levels of defaults are possible, and each resource_name can have its own set of defaults. (However, clients MUST take pains to detect circular references in rc:defaults; any rc:defaults value which matches to either the request_uri or a previous rc:defaults value obtained during the same query, MUST be ignored.)

6. IANA Considerations

- A new TCP port and a new UDP port are needed

- A registry of attribute_names, and procedures for assigning and delegating attribute naming prefixes, are needed.

- A registry of signature algorithms, and a procedure for registering new signature algorithms identifiers, are needed.

- A service name (presumably "rc") for use with SASL (and therefore GSSAPI) needs to be defined.

7. Security Considerations

Many security considerations are discussed in section 2.5. Here are some others.

7.1 Privacy issues regarding choice of request_ids

Since request_ids need to be unique, it is tempting to define them in terms of some other convenient globally-unique number such as a GUID, network interface hardware address, (non-private) IP address, etc. This may result in inadvertent disclosure of information about usage patterns to unauthorized parties. Request-ids SHOULD be chosen in such a way as to minimize the potential for disclosure of information which could be associated with a particular person or host. The fact that similar information could be disclosed by other means does not lessen the burden for RC client implementations to minimize their own potential for disclosure through the request_id field.

Similarly, RC proxies that multiplex several clients' requests over the same TCP stream may need to map between clients' request_ids and their own request_ids for proxied requests, in order to ensure that each request_id transmitted over a TCP stream is unique. Such proxies SHOULD avoid disclosing identifying information about clients via the request_id field.

8. Differences from RC version 2

Just to provide some background, this section details major differences between RCv2 and RCv3:

- RCv2 had typed attributes; RCv3 makes all attributes octet-strings.

- RCv2 used ONC XDR and RPC [13]; RCv3 uses the BLOB encoding and defines its own RPC mechanism. For a variety of reasons, most having little to do with protocol quality, RCv2's use of ONC RPC was the single most frequently cited objection to that protocol. However the mechanism intended for use with RCv3 is easier to implement and more efficient to process than XDR.

- RCv2 maintained separate records for attributes of resources (like RCv3's assertions) and locations of resources, allowing the caller to choose the closest of several locations for that resource. The latter also included cached DNS A records in order to speed up accesses to the resource. RCv3 does not have this separation.

- RCv2 supported the principle of multiple 'asserters' for information about a resource. it kept track of who made each assertion, and if different asserters made conflicting assertions, it would return all of the assertions. RCv3 assumes that a small number of trusted parties will be allowed to make assertions about any particular resource, and that these parties will cooperate to the degree necessary to avoid making conflicting assertions about a resource. The QUERY_RECURSE feature of RCv3 is intended to provide similar functionality in a more general fashion.

- RCv2 supported the notion of redirects - the ability to redirect queries for portions of URI-space - at the protocol level; this is omitted in RCv3 because they were too much trouble to use. Redirects could be implemented in RCv3 by making them ordinary attributes and by using the QUERY_RECURSE bit in queries for that attribute name.

9. Acknowledgements

The author wishes to thank Graham Klyne for extensive comments on an earlier version of this document.

10. Author's Address

Keith Moore
University of Tennessee, Knoxville
1122 Volunteer Blvd, Suite 203
Knoxville TN, 37996-3450
USA
email: moore@cs.utk.edu

11. References

[1] Bradner, S. Key words for use in RFCs to Indicate Requirement Levels. RFC 2119, March 1997.

[2] Mealling, M., Daniel, R. The Naming Authority Pointer (NAPTR) DNS Resource Record. RFC 2915, September 2000.

[3] Gulbrandsen, A., Vixie, P., Esibow, L. A DNS RR for specifying the location of services (DNS SRV). RFC 2782, February 2000.

[4] Dierks, T., Allen, C. The TLS Protocol Version 1.0. RFC 2246, January 1999.

[5] Mockapetris, P.V. Domain names - implementation and specification. RFC 1035, November 1987.

[6] Case, J. Mundy, R., Partain, D., Stewart, B. Introduction to Version 3 of the Internet-standard Network Management Framework. RFC 2570, April 1999.

[7] Wahl, M., Howes, T., Kille, S. Lightweight Directory Access Protocol (v3). RFC 2251, December 1997.

[8] Moore, K. The Binary Low-Overhead Block Presentation Protocol. Internet-Draft draft-ietf-rescap-blob-01.txt, March 1, 2002, work in progress.

[9] Myers, J. Simple Authentication and Security Layer (SASL). RFC 2222, October 1997.

[10] Linn, J. Generic Security Service Application Program Interface Version 2, Update 1. RFC 2743, January 2000.

[11] Daniel, R., Mealling, M. Resolution of Uniform Resource Identifiers using the Domain Name System. RFC 2168, June 1997.

[12] Mealling, M. A DDDS Database Using the Domain Name System. Internet-Draft draft-ietf-urn-dns-ddds-database-08.txt, February 2002. work in progress.

[13] Srinivasan, R. RPC: Remote Procedure Call Protocol Specification Version 2. RFC 1831, August 1995.

Appendix A. Substantive changes since version -00

- section 2.5: removed what was essentially a duplicate description of the 'alteration of data' threat; also added some text explaining limitations of server-provided signatures.

- section 3.1: explain why it's useful to have both a TTL and an expiration date.

- section 4.1 et seq: renumber request types so that the "reserved" request is 0.

- section 4.1.1: change example of supported_requests array to illustrate a range consisting of a single request number. also added a note about caching of supported request information.

- various: updated section references to reflect current reality.

- section 4.2.1.5: clarify that a signature may be computed over resource_name and version_* in addition to the list of assertions.

- section 4.2.2.1: eliminate duplicate version_* fields in update_request. also state explicitly how the server updates version_* fields on completion of successful requests, and limitations on using UPDATE_VERSION_MATCH due to wrap-around of version_ fields.

- section 4.2.3: rename Authenticate operation to One-Shot Authenticate; add new SASL Authenticate operation. The former is designed for use with UDP; the latter, with TCP or TLS.

- section 4.3: remove text stating requirements for support of TCP and UDP; this needs re-thinking.

- section 4.4: forbid caching of negative responses by proxies.

- section 4.5: added new status codes

- section 5.1: clarify handling of circular references to "rc:defaults".

Appendix B. Open issues

- There is no model for access control. This can probably be deferred.

- A mandatory-to-implement TLS ciphersuite for servers supporting StartTLS operation needs to be chosen.

- Authentication methods for One-Shot authentication need to be defined.

- At least one signature algorithm needs to be defined.

- Characteristics of UDP request retransmission timer need to be defined. (borrow from SIP?)

- Conformance requrements for clients and servers regarding TCP and UDP, TLS, and authentication mechanisms, need to be defined.

- Rules for caches/proxies regarding use of authentication need to be defined.