Network Working Group
Request for Comments: 2136
Updates: 1035
Category: Standards Track

P. Vixie, Editor
ISC
S. Thomson
Bellcore
Y. Rekhter
Cisco
J. Bound
DEC

April 1997

Dynamic Updates in the Domain Name System (DNS UPDATE)

Status of this Memo

Abstract

The Domain Name System was originally designed to support queries of a statically configured database. While the data was expected to change, the frequency of those changes was expected to be fairly low, and all updates were made as external edits to a zone's Master File.

Using this specification of the UPDATE opcode, it is possible to add or delete RRs or RRsets from a specified zone. Prerequisites are specified separately from update operations, and can specify a dependency upon either the previous existence or nonexistence of an RRset, or the existence of a single RR.

UPDATE is atomic, i.e., all prerequisites must be satisfied or else no update operations will take place. There are no data dependent error conditions defined after the prerequisites have been met.

1 - Definitions

This document intentionally gives more definition to the roles of "Master," "Slave," and "Primary Master" servers, and their enumeration in NS RRs, and the SOA MNAME field. In that sense, the following server type definitions can be considered an addendum to [RFC1035], and are intended to be consistent with [RFC1996]:

Slave — an authoritative server that uses AXFR or IXFR to retrieve the zone and is named in the zone's NS RRset.

```
   Master          an authoritative server configured to be the
                   source of AXFR or IXFR data for one or more slave
                   servers.

   Primary Master  master server at the root of the AXFR/IXFR
                   dependency graph.  The primary master is named in
                   the zone's SOA MNAME field and optionally by an NS
                   RR.  There is by definition only one primary master
                   server per zone.
```

A domain name identifies a node within the domain name space tree structure. Each node has a set (possibly empty) of Resource Records (RRs). All RRs having the same NAME, CLASS and TYPE are called a Resource Record Set (RRset).

The pseudocode used in this document is for example purposes only. If it is found to disagree with the text, the text shall be considered authoritative. If the text is found to be ambiguous, the pseudocode can be used to help resolve the ambiguity.

1.1 - Comparison Rules

1.1.1. Two RRs are considered equal if their NAME, CLASS, TYPE, RDLENGTH and RDATA fields are equal. Note that the time-to-live (TTL) field is explicitly excluded from the comparison.

1.1.2. The rules for comparison of character strings in names are specified in [RFC1035 2.3.3].

1.1.3. Wildcarding is disabled. That is, a wildcard ("*") in an update only matches a wildcard ("*") in the zone, and vice versa.

1.1.4. Aliasing is disabled: A CNAME in the zone matches a CNAME in the update, and will not otherwise be followed. All UPDATE operations are done on the basis of canonical names.

1.1.5. The following RR types cannot be appended to an RRset. If the following comparison rules are met, then an attempt to add the new RR will result in the replacement of the previous RR:

```
   SOA    compare only NAME, CLASS and TYPE -- it is not possible to
          have more than one SOA per zone, even if any of the data
          fields differ.

   WKS    compare only NAME, CLASS, TYPE, ADDRESS, and PROTOCOL
          -- only one WKS RR is possible for this tuple, even if the
          services masks differ.
```

CNAME compare only NAME, CLASS, and TYPE -- it is not possible to have more than one CNAME RR, even if their data fields differ.

1.2 - Glue RRs

For the purpose of determining whether a domain name used in the UPDATE protocol is contained within a specified zone, a domain name is "in" a zone if it is owned by that zone's domain name. See section 7.18 for details.

1.3 - New Assigned Numbers

```
CLASS = NONE (254)
RCODE = YXDOMAIN (6)
RCODE = YXRRSET (7)
RCODE = NXRRSET (8)
RCODE = NOTAUTH (9)
RCODE = NOTZONE (10)
Opcode = UPDATE (5)
```

2 - Update Message Format

The DNS Message Format is defined by [RFC1035 4.1]. Some extensions are necessary (for example, more error codes are possible under UPDATE than under QUERY) and some fields must be overloaded (see description of CLASS fields below).

The overall format of an UPDATE message is, following [ibid]:

```
+---------------------+
|        Header       |
+---------------------+
|         Zone        | specifies the zone to be updated
+---------------------+
|     Prerequisite    | RRs or RRsets which must (not) preexist
+---------------------+
|        Update       | RRs or RRsets to be added or deleted
+---------------------+
|   Additional Data   | additional data
+---------------------+
```

The Header Section specifies that this message is an UPDATE, and describes the size of the other sections. The Zone Section names the zone that is to be updated by this message. The Prerequisite Section specifies the starting invariants (in terms of zone content) required for this update. The Update Section contains the edits to be made, and the Additional Data Section contains data which may be necessary to complete, but is not part of, this update.

2.1 - Transport Issues

An update transaction may be carried in a UDP datagram, if the request fits, or in a TCP connection (at the discretion of the requestor). When TCP is used, the message is in the format described in [RFC1035 4.2.2].

2.2 - Message Header

The header of the DNS Message Format is defined by [RFC 1035 4.1]. Not all opcodes define the same set of flag bits, though as a practical matter most of the bits defined for QUERY (in [ibid]) are identically defined by the other opcodes. UPDATE uses only one flag bit (QR).

The DNS Message Format specifies record counts for its four sections (Question, Answer, Authority, and Additional). UPDATE uses the same fields, and the same section formats, but the naming and use of these sections differs as shown in the following modified header, after [RFC1035 4.1.1]:

```
                                    1  1  1  1  1  1
      0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    |                      ID                       |
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    |QR|   Opcode  |          Z         |   RCODE   |
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    |                    ZOCOUNT                    |
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    |                    PRCOUNT                    |
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    |                    UPCOUNT                    |
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    |                    ADCOUNT                    |
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

These fields are used as follows:

ID — A 16-bit identifier assigned by the entity that generates any kind of request. This identifier is copied in the corresponding reply and can be used by the requestor to match replies to outstanding requests, or by the server to detect duplicated requests from some requestor.

QR — A one bit field that specifies whether this message is a request (0), or a response (1).

Opcode — A four bit field that specifies the kind of request in this message. This value is set by the originator of a request and copied into the response. The Opcode value that identifies an UPDATE message is five (5).

Z — Reserved for future use. Should be zero (0) in all requests and responses. A non-zero Z field should be ignored by implementations of this specification.

RCODE — Response code - this four bit field is undefined in requests and set in responses. The values and meanings of this field within responses are as follows:

Mneumonic	Value	Description
NOERROR	0	No error condition.
FORMERR	1	The name server was unable to interpret the request due to a format error.
SERVFAIL	2	The name server encountered an internal failure while processing this request, for example an operating system error or a forwarding timeout.
NXDOMAIN	3	Some name that ought to exist, does not exist.
NOTIMP	4	The name server does not support the specified Opcode.
REFUSED	5	The name server refuses to perform the specified operation for policy or security reasons.
YXDOMAIN	6	Some name that ought not to exist, does exist.
YXRRSET	7	Some RRset that ought not to exist, does exist.
NXRRSET	8	Some RRset that ought to exist, does not exist.

```
           NOTAUTH     9       The server is not authoritative for
                               the zone named in the Zone Section.
           NOTZONE     10      A name used in the Prerequisite or
                               Update Section is not within the
                               zone denoted by the Zone Section.
```

ZOCOUNT The number of RRs in the Zone Section.

PRCOUNT The number of RRs in the Prerequisite Section.

UPCOUNT The number of RRs in the Update Section.

ADCOUNT The number of RRs in the Additional Data Section.

2.3 - Zone Section

The Zone Section has the same format as that specified in [RFC1035 4.1.2], with the fields redefined as follows:

```
                                    1  1  1  1  1  1
      0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    |                                               |
    /                     ZNAME                     /
    /                                               /
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    |                     ZTYPE                     |
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    |                     ZCLASS                    |
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

UPDATE uses this section to denote the zone of the records being updated. All records to be updated must be in the same zone, and therefore the Zone Section is allowed to contain exactly one record. The ZNAME is the zone name, the ZTYPE must be SOA, and the ZCLASS is the zone's class.

2.4 - Prerequisite Section

This section contains a set of RRset prerequisites which must be satisfied at the time the UPDATE packet is received by the primary master server. The format of this section is as specified by [RFC1035 4.1.3]. There are five possible sets of semantics that can be expressed here, summarized as follows and then explained below.

(1) RRset exists (value independent). At least one RR with a specified NAME and TYPE (in the zone and class specified by the Zone Section) must exist.

(2) RRset exists (value dependent). A set of RRs with a specified NAME and TYPE exists and has the same members with the same RDATAs as the RRset specified here in this Section.

(3) RRset does not exist. No RRs with a specified NAME and TYPE (in the zone and class denoted by the Zone Section) can exist.

(4) Name is in use. At least one RR with a specified NAME (in the zone and class specified by the Zone Section) must exist. Note that this prerequisite is NOT satisfied by empty nonterminals.

(5) Name is not in use. No RR of any type is owned by a specified NAME. Note that this prerequisite IS satisfied by empty nonterminals.

The syntax of these is as follows:

2.4.1 - RRset Exists (Value Independent)

At least one RR with a specified NAME and TYPE (in the zone and class specified in the Zone Section) must exist.

For this prerequisite, a requestor adds to the section a single RR whose NAME and TYPE are equal to that of the zone RRset whose existence is required. RDLENGTH is zero and RDATA is therefore empty. CLASS must be specified as ANY to differentiate this condition from that of an actual RR whose RDLENGTH is naturally zero (0) (e.g., NULL). TTL is specified as zero (0).

2.4.2 - RRset Exists (Value Dependent)

A set of RRs with a specified NAME and TYPE exists and has the same members with the same RDATAs as the RRset specified here in this section. While RRset ordering is undefined and therefore not significant to this comparison, the sets be identical in their extent.

For this prerequisite, a requestor adds to the section an entire RRset whose preexistence is required. NAME and TYPE are that of the RRset being denoted. CLASS is that of the zone. TTL must be specified as zero (0) and is ignored when comparing RRsets for identity.

2.4.3 - RRset Does Not Exist

No RRs with a specified NAME and TYPE (in the zone and class denoted by the Zone Section) can exist.

For this prerequisite, a requestor adds to the section a single RR whose NAME and TYPE are equal to that of the RRset whose nonexistence is required. The RDLENGTH of this record is zero (0), and RDATA field is therefore empty. CLASS must be specified as NONE in order to distinguish this condition from a valid RR whose RDLENGTH is naturally zero (0) (for example, the NULL RR). TTL must be specified as zero (0).

2.4.4 - Name Is In Use

Name is in use. At least one RR with a specified NAME (in the zone and class specified by the Zone Section) must exist. Note that this prerequisite is NOT satisfied by empty nonterminals.

For this prerequisite, a requestor adds to the section a single RR whose NAME is equal to that of the name whose ownership of an RR is required. RDLENGTH is zero and RDATA is therefore empty. CLASS must be specified as ANY to differentiate this condition from that of an actual RR whose RDLENGTH is naturally zero (0) (e.g., NULL). TYPE must be specified as ANY to differentiate this case from that of an RRset existence test. TTL is specified as zero (0).

2.4.5 - Name Is Not In Use

Name is not in use. No RR of any type is owned by a specified NAME. Note that this prerequisite IS satisfied by empty nonterminals.

For this prerequisite, a requestor adds to the section a single RR whose NAME is equal to that of the name whose nonownership of any RRs is required. RDLENGTH is zero and RDATA is therefore empty. CLASS must be specified as NONE. TYPE must be specified as ANY. TTL must be specified as zero (0).

2.5 - Update Section

This section contains RRs to be added to or deleted from the zone. The format of this section is as specified by [RFC1035 4.1.3]. There are four possible sets of semantics, summarized below and with details to follow.

(1) Add RRs to an RRset.
(2) Delete an RRset.
(3) Delete all RRsets from a name.
(4) Delete an RR from an RRset.

The syntax of these is as follows:

2.5.1 - Add To An RRset

RRs are added to the Update Section whose NAME, TYPE, TTL, RDLENGTH and RDATA are those being added, and CLASS is the same as the zone class. Any duplicate RRs will be silently ignored by the primary master.

2.5.2 - Delete An RRset

One RR is added to the Update Section whose NAME and TYPE are those of the RRset to be deleted. TTL must be specified as zero (0) and is otherwise not used by the primary master. CLASS must be specified as ANY. RDLENGTH must be zero (0) and RDATA must therefore be empty. If no such RRset exists, then this Update RR will be silently ignored by the primary master.

2.5.3 - Delete All RRsets From A Name

One RR is added to the Update Section whose NAME is that of the name to be cleansed of RRsets. TYPE must be specified as ANY. TTL must be specified as zero (0) and is otherwise not used by the primary master. CLASS must be specified as ANY. RDLENGTH must be zero (0) and RDATA must therefore be empty. If no such RRsets exist, then this Update RR will be silently ignored by the primary master.

2.5.4 - Delete An RR From An RRset

RRs to be deleted are added to the Update Section. The NAME, TYPE, RDLENGTH and RDATA must match the RR being deleted. TTL must be specified as zero (0) and will otherwise be ignored by the primary master. CLASS must be specified as NONE to distinguish this from an RR addition. If no such RRs exist, then this Update RR will be silently ignored by the primary master.

2.6 - Additional Data Section

This section contains RRs which are related to the update itself, or to new RRs being added by the update. For example, out of zone glue (A RRs referred to by new NS RRs) should be presented here. The server can use or ignore out of zone glue, at the discretion of the server implementor. The format of this section is as specified by [RFC1035 4.1.3].

3 - Server Behavior

A server, upon receiving an UPDATE request, will signal NOTIMP to the requestor if the UPDATE opcode is not recognized or if it is recognized but has not been implemented. Otherwise, processing continues as follows.

3.1 - Process Zone Section

3.1.1. The Zone Section is checked to see that there is exactly one RR therein and that the RR's ZTYPE is SOA, else signal FORMERR to the requestor. Next, the ZNAME and ZCLASS are checked to see if the zone so named is one of this server's authority zones, else signal NOTAUTH to the requestor. If the server is a zone slave, the request will be forwarded toward the primary master.

3.1.2 - Pseudocode For Zone Section Processing

```
if (zcount != 1 || ztype != SOA)
     return (FORMERR)
if (zone_type(zname, zclass) == SLAVE)
     return forward()
if (zone_type(zname, zclass) == MASTER)
     return update()
return (NOTAUTH)
```

Sections 3.2 through 3.8 describe the primary master's behaviour, whereas Section 6 describes a forwarder's behaviour.

3.2 - Process Prerequisite Section

Next, the Prerequisite Section is checked to see that all prerequisites are satisfied by the current state of the zone. Using the definitions expressed in Section 1.2, if any RR's NAME is not within the zone specified in the Zone Section, signal NOTZONE to the requestor.

3.2.1. For RRs in this section whose CLASS is ANY, test to see that TTL and RDLENGTH are both zero (0), else signal FORMERR to the requestor. If TYPE is ANY, test to see that there is at least one RR in the zone whose NAME is the same as that of the Prerequisite RR, else signal NXDOMAIN to the requestor. If TYPE is not ANY, test to see that there is at least one RR in the zone whose NAME and TYPE are the same as that of the Prerequisite RR, else signal NXRRSET to the requestor.

3.2.2. For RRs in this section whose CLASS is NONE, test to see that the TTL and RDLENGTH are both zero (0), else signal FORMERR to the requestor. If the TYPE is ANY, test to see that there are no RRs in the zone whose NAME is the same as that of the Prerequisite RR, else signal YXDOMAIN to the requestor. If the TYPE is not ANY, test to see that there are no RRs in the zone whose NAME and TYPE are the same as that of the Prerequisite RR, else signal YXRRSET to the requestor.

3.2.3. For RRs in this section whose CLASS is the same as the ZCLASS, test to see that the TTL is zero (0), else signal FORMERR to the requestor. Then, build an RRset for each unique <NAME,TYPE> and compare each resulting RRset for set equality (same members, no more, no less) with RRsets in the zone. If any Prerequisite RRset is not entirely and exactly matched by a zone RRset, signal NXRRSET to the requestor. If any RR in this section has a CLASS other than ZCLASS or NONE or ANY, signal FORMERR to the requestor.

3.2.4 - Table Of Metavalues Used In Prerequisite Section

CLASS	TYPE	RDATA	Meaning
ANY	ANY	empty	Name is in use
ANY	rrset	empty	RRset exists (value independent)
NONE	ANY	empty	Name is not in use
NONE	rrset	empty	RRset does not exist
zone	rrset	rr	RRset exists (value dependent)

3.2.5 - Pseudocode for Prerequisite Section Processing

```
for rr in prerequisites
     if (rr.ttl != 0)
          return (FORMERR)
     if (zone_of(rr.name) != ZNAME)
          return (NOTZONE);
     if (rr.class == ANY)
          if (rr.rdlength != 0)
               return (FORMERR)
          if (rr.type == ANY)
               if (!zone_name<rr.name>)
                    return (NXDOMAIN)
          else
               if (!zone_rrset<rr.name, rr.type>)
                    return (NXRRSET)
     if (rr.class == NONE)
          if (rr.rdlength != 0)
               return (FORMERR)
          if (rr.type == ANY)
               if (zone_name<rr.name>)
                    return (YXDOMAIN)
          else
               if (zone_rrset<rr.name, rr.type>)
                    return (YXRRSET)
     if (rr.class == zclass)
          temp<rr.name, rr.type> += rr
     else
          return (FORMERR)

for rrset in temp
     if (zone_rrset<rrset.name, rrset.type> != rrset)
          return (NXRRSET)
```

3.3 - Check Requestor's Permissions

3.3.1. Next, the requestor's permission to update the RRs named in the Update Section may be tested in an implementation dependent fashion or using mechanisms specified in a subsequent Secure DNS Update protocol. If the requestor does not have permission to perform these updates, the server may write a warning message in its operations log, and may either signal REFUSED to the requestor, or ignore the permission problem and proceed with the update.

3.3.2. While the exact processing is implementation defined, if these verification activities are to be performed, this is the point in the server's processing where such performance should take place, since if a REFUSED condition is encountered after an update has been partially applied, it will be necessary to undo the partial update and restore the zone to its original state before answering the requestor.

3.3.3 - Pseudocode for Permission Checking

```
   if (security policy exists)
        if (this update is not permitted)
             if (local option)
                  log a message about permission problem
             if (local option)
                  return (REFUSED)
```

3.4 - Process Update Section

Next, the Update Section is processed as follows.

3.4.1 - Prescan

The Update Section is parsed into RRs and each RR's CLASS is checked to see if it is ANY, NONE, or the same as the Zone Class, else signal a FORMERR to the requestor. Using the definitions in Section 1.2, each RR's NAME must be in the zone specified by the Zone Section, else signal NOTZONE to the requestor.

3.4.1.2. For RRs whose CLASS is not ANY, check the TYPE and if it is ANY, AXFR, MAILA, MAILB, or any other QUERY metatype, or any unrecognized type, then signal FORMERR to the requestor. For RRs whose CLASS is ANY or NONE, check the TTL to see that it is zero (0), else signal a FORMERR to the requestor. For any RR whose CLASS is ANY, check the RDLENGTH to make sure that it is zero (0) (that is, the RDATA field is empty), and that the TYPE is not AXFR, MAILA, MAILB, or any other QUERY metatype besides ANY, or any unrecognized type, else signal FORMERR to the requestor.

3.4.1.3 - Pseudocode For Update Section Prescan

```
[rr] for rr in updates
     if (zone_of(rr.name) != ZNAME)
          return (NOTZONE);
     if (rr.class == zclass)
          if (rr.type & ANY|AXFR|MAILA|MAILB)
               return (FORMERR)
     elsif (rr.class == ANY)
          if (rr.ttl != 0 || rr.rdlength != 0
              || rr.type & AXFR|MAILA|MAILB)
               return (FORMERR)
     elsif (rr.class == NONE)
          if (rr.ttl != 0 || rr.type & ANY|AXFR|MAILA|MAILB)
               return (FORMERR)
     else
          return (FORMERR)
```

3.4.2 - Update

The Update Section is parsed into RRs and these RRs are processed in order.

3.4.2.1. If any system failure (such as an out of memory condition, or a hardware error in persistent storage) occurs during the processing of this section, signal SERVFAIL to the requestor and undo all updates applied to the zone during this transaction.

3.4.2.2. Any Update RR whose CLASS is the same as ZCLASS is added to the zone. In case of duplicate RDATAs (which for SOA RRs is always the case, and for WKS RRs is the case if the ADDRESS and PROTOCOL fields both match), the Zone RR is replaced by Update RR. If the TYPE is SOA and there is no Zone SOA RR, or the new SOA.SERIAL is lower (according to [RFC1982]) than or equal to the current Zone SOA RR's SOA.SERIAL, the Update RR is ignored. In the case of a CNAME Update RR and a non-CNAME Zone RRset or vice versa, ignore the CNAME Update RR, otherwise replace the CNAME Zone RR with the CNAME Update RR.

3.4.2.3. For any Update RR whose CLASS is ANY and whose TYPE is ANY, all Zone RRs with the same NAME are deleted, unless the NAME is the same as ZNAME in which case only those RRs whose TYPE is other than SOA or NS are deleted. For any Update RR whose CLASS is ANY and whose TYPE is not ANY all Zone RRs with the same NAME and TYPE are deleted, unless the NAME is the same as ZNAME in which case neither SOA or NS RRs will be deleted.

3.4.2.4. For any Update RR whose class is NONE, any Zone RR whose NAME, TYPE, RDATA and RDLENGTH are equal to the Update RR is deleted, unless the NAME is the same as ZNAME and either the TYPE is SOA or the TYPE is NS and the matching Zone RR is the only NS remaining in the RRset, in which case this Update RR is ignored.

3.4.2.5. Signal NOERROR to the requestor.

3.4.2.6 - Table Of Metavalues Used In Update Section

CLASS	TYPE	RDATA	Meaning
ANY	ANY	empty	Delete all RRsets from a name
ANY	rrset	empty	Delete an RRset
NONE	rrset	rr	Delete an RR from an RRset
zone	rrset	rr	Add to an RRset

3.4.2.7 - Pseudocode For Update Section Processing

```
[rr] for rr in updates
     if (rr.class == zclass)
          if (rr.type == CNAME)
               if (zone_rrset<rr.name, ~CNAME>)
                    next [rr]
          elsif (zone_rrset<rr.name, CNAME>)
               next [rr]
          if (rr.type == SOA)
               if (!zone_rrset<rr.name, SOA> ||
                   zone_rr<rr.name, SOA>.serial > rr.soa.serial)
                    next [rr]
          for zrr in zone_rrset<rr.name, rr.type>
               if (rr.type == CNAME || rr.type == SOA ||
                   (rr.type == WKS && rr.proto == zrr.proto &&
                    rr.address == zrr.address) ||
                   rr.rdata == zrr.rdata)
                    zrr = rr
                    next [rr]
          zone_rrset<rr.name, rr.type> += rr
     elsif (rr.class == ANY)
          if (rr.type == ANY)
               if (rr.name == zname)
                    zone_rrset<rr.name, ~(SOA|NS)> = Nil
               else
                    zone_rrset<rr.name, *> = Nil
          elsif (rr.name == zname &&
                 (rr.type == SOA || rr.type == NS))
               next [rr]
          else
```

```
                  zone_rrset<rr.name, rr.type> = Nil
        elsif (rr.class == NONE)
             if (rr.type == SOA)
                  next [rr]
             if (rr.type == NS && zone_rrset<rr.name, NS> == rr)
                  next [rr]
             zone_rr<rr.name, rr.type, rr.data> = Nil
   return (NOERROR)
```

3.5 - Stability

When a zone is modified by an UPDATE operation, the server must commit the change to nonvolatile storage before sending a response to the requestor or answering any queries or transfers for the modified zone. It is reasonable for a server to store only the update records as long as a system reboot or power failure will cause these update records to be incorporated into the zone the next time the server is started. It is also reasonable for the server to copy the entire modified zone to nonvolatile storage after each update operation, though this would have suboptimal performance for large zones.

3.6 - Zone Identity

If the zone's SOA SERIAL is changed by an update operation, that change must be in a positive direction (using modulo 2**32 arithmetic as specified by [RFC1982]). Attempts to replace an SOA with one whose SERIAL is less than the current one will be silently ignored by the primary master server.

If the zone's SOA's SERIAL is not changed as a result of an update operation, then the server shall increment it automatically before the SOA or any changed name or RR or RRset is included in any response or transfer. The primary master server's implementor might choose to autoincrement the SOA SERIAL if any of the following events occurs:

(1) Each update operation.

(2) A name, RR or RRset in the zone has changed and has subsequently been visible to a DNS client since the unincremented SOA was visible to a DNS client, and the SOA is about to become visible to a DNS client.

(3) A configurable period of time has elapsed since the last update operation. This period shall be less than or equal to one third of the zone refresh time, and the default shall be the lesser of that maximum and 300 seconds.

(4) A configurable number of updates has been applied since the last SOA change. The default value for this configuration parameter shall be one hundred (100).

It is imperative that the zone's contents and the SOA's SERIAL be tightly synchronized. If the zone appears to change, the SOA must appear to change as well.

3.7 - Atomicity

During the processing of an UPDATE transaction, the server must ensure atomicity with respect to other (concurrent) UPDATE or QUERY transactions. No two transactions can be processed concurrently if either depends on the final results of the other; in particular, a QUERY should not be able to retrieve RRsets which have been partially modified by a concurrent UPDATE, and an UPDATE should not be able to start from prerequisites that might not still hold at the completion of some other concurrent UPDATE. Finally, if two UPDATE transactions would modify the same names, RRs or RRsets, then such UPDATE transactions must be serialized.

3.8 - Response

At the end of UPDATE processing, a response code will be known. A response message is generated by copying the ID and Opcode fields from the request, and either copying the ZOCOUNT, PRCOUNT, UPCOUNT, and ADCOUNT fields and associated sections, or placing zeros (0) in the these "count" fields and not including any part of the original update. The QR bit is set to one (1), and the response is sent back to the requestor. If the requestor used UDP, then the response will be sent to the requestor's source UDP port. If the requestor used TCP, then the response will be sent back on the requestor's open TCP connection.

4 - Requestor Behaviour

4.1. From a requestor's point of view, any authoritative server for the zone can appear to be able to process update requests, even though only the primary master server is actually able to modify the zone's master file. Requestors are expected to know the name of the zone they intend to update and to know or be able to determine the name servers for that zone.

4.2. If update ordering is desired, the requestor will need to know the value of the existing SOA RR. Requestors who update the SOA RR must update the SOA SERIAL field in a positive direction (as defined by [RFC1982]) and also preserve the other SOA fields unless the requestor's explicit intent is to change them. The SOA SERIAL field must never be set to zero (0).

4.3. If the requestor has reasonable cause to believe that all of a zone's servers will be equally reachable, then it should arrange to try the primary master server (as given by the SOA MNAME field if matched by some NS NSDNAME) first to avoid unnecessary forwarding inside the slave servers. (Note that the primary master will in some cases not be reachable by all requestors, due to firewalls or network partitioning.)

4.4. Once the zone's name servers been found and possibly sorted so that the ones more likely to be reachable and/or support the UPDATE opcode are listed first, the requestor composes an UPDATE message of the following form and sends it to the first name server on its list:

```
ID:                         (new)
Opcode:                     UPDATE
Zone zcount:                1
Zone zname:                 (zone name)
Zone zclass:                (zone class)
Zone ztype:                 T_SOA
Prerequisite Section:       (see previous text)
Update Section:             (see previous text)
Additional Data Section:    (empty)
```

4.5. If the requestor receives a response, and the response has an RCODE other than SERVFAIL or NOTIMP, then the requestor returns an appropriate response to its caller.

4.6. If a response is received whose RCODE is SERVFAIL or NOTIMP, or if no response is received within an implementation dependent timeout period, or if an ICMP error is received indicating that the server's port is unreachable, then the requestor will delete the unusable server from its internal name server list and try the next one, repeating until the name server list is empty. If the requestor runs out of servers to try, an appropriate error will be returned to the requestor's caller.

5 - Duplicate Detection, Ordering and Mutual Exclusion

5.1. For correct operation, mechanisms may be needed to ensure idempotence, order UPDATE requests and provide mutual exclusion. An UPDATE message or response might be delivered zero times, one time, or multiple times. Datagram duplication is of particular interest since it covers the case of the so-called "replay attack" where a correct request is duplicated maliciously by an intruder.

5.2. Multiple UPDATE requests or responses in transit might be delivered in any order, due to network topology changes or load balancing, or to multipath forwarding graphs wherein several slave servers all forward to the primary master. In some cases, it might be required that the earlier update not be applied after the later update, where "earlier" and "later" are defined by an external time base visible to some set of requestors, rather than by the order of request receipt at the primary master.

5.3. A requestor can ensure transaction idempotence by explicitly deleting some "marker RR" (rather than deleting the RRset of which it is a part) and then adding a new "marker RR" with a different RDATA field. The Prerequisite Section should specify that the original "marker RR" must be present in order for this UPDATE message to be accepted by the server.

5.4. If the request is duplicated by a network error, all duplicate requests will fail since only the first will find the original "marker RR" present and having its known previous value. The decisions of whether to use such a "marker RR" and what RR to use are left up to the application programmer, though one obvious choice is the zone's SOA RR as described below.

5.5. Requestors can ensure update ordering by externally synchronizing their use of successive values of the "marker RR." Mutual exclusion can be addressed as a degenerate case, in that a single succession of the "marker RR" is all that is needed.

5.6. A special case where update ordering and datagram duplication intersect is when an RR validly changes to some new value and then back to its previous value. Without a "marker RR" as described above, this sequence of updates can leave the zone in an undefined state if datagrams are duplicated.

5.7. To achieve an atomic multitransaction "read-modify-write" cycle, a requestor could first retrieve the SOA RR, and build an UPDATE message one of whose prerequisites was the old SOA RR. It would then specify updates that would delete this SOA RR and add a new one with an incremented SOA SERIAL, along with whatever actual prerequisites and updates were the object of the transaction. If the transaction succeeds, the requestor knows that the RRs being changed were not otherwise altered by any other requestor.

6 - Forwarding

When a zone slave forwards an UPDATE message upward toward the zone's primary master server, it must allocate a new ID and prepare to enter the role of "forwarding server," which is a requestor with respect to the forward server.

6.1. The set of forward servers will be same as the set of servers this zone slave would use as the source of AXFR or IXFR data. So, while the original requestor might have used the zone's NS RRset to locate its update server, a forwarder always forwards toward its designated zone master servers.

6.2. If the original requestor used TCP, then the TCP connection from the requestor is still open and the forwarder must use TCP to forward the message. If the original requestor used UDP, the forwarder may use either UDP or TCP to forward the message, at the whim of the implementor.

6.3. It is reasonable for forward servers to be forwarders themselves, if the AXFR dependency graph being followed is a deep one involving firewalls and multiple connectivity realms. In most cases the AXFR dependency graph will be shallow and the forward server will be the primary master server.

6.4. The forwarder will not respond to its requestor until it receives a response from its forward server. UPDATE transactions involving forwarders are therefore time synchronized with respect to the original requestor and the primary master server.

6.5. When there are multiple possible sources of AXFR data and therefore multiple possible forward servers, a forwarder will use the same fallback strategy with respect to connectivity or timeout errors that it would use when performing an AXFR. This is implementation dependent.

6.6. When a forwarder receives a response from a forward server, it copies this response into a new response message, assigns its requestor's ID to that message, and sends the response back to the requestor.

7 - Design, Implementation, Operation, and Protocol Notes

Some of the principles which guided the design of this UPDATE specification are as follows. Note that these are not part of the formal specification and any disagreement between this section and any other section of this document should be resolved in favour of the other section.

7.1. Using metavalues for CLASS is possible only because all RRs in the packet are assumed to be in the same zone, and CLASS is an attribute of a zone rather than of an RRset. (It is for this reason that the Zone Section is not optional.)

7.2. Since there are no data-present or data-absent errors possible from processing the Update Section, any necessary data-present and data- absent dependencies should be specified in the Prerequisite Section.

7.3. The Additional Data Section can be used to supply a server with out of zone glue that will be needed in referrals. For example, if adding a new NS RR to HOME.VIX.COM specifying a nameserver called NS.AU.OZ, the A RR for NS.AU.OZ can be included in the Additional Data Section. Servers can use this information or ignore it, at the discretion of the implementor. We discourage caching this information for use in subsequent DNS responses.

7.4. The Additional Data Section might be used if some of the RRs later needed for Secure DNS Update are not actually zone updates, but rather ancillary keys or signatures not intended to be stored in the zone (as an update would be), yet necessary for validating the update operation.

7.5. It is expected that in the absence of Secure DNS Update, a server will only accept updates if they come from a source address that has been statically configured in the server's description of a primary master zone. DHCP servers would be likely candidates for inclusion in this statically configured list.

7.6. It is not possible to create a zone using this protocol, since there is no provision for a slave server to be told who its master servers are. It is expected that this protocol will be extended in the future to cover this case. Therefore, at this time, the addition of SOA RRs is unsupported. For similar reasons, deletion of SOA RRs is also unsupported.

7.7. The prerequisite for specifying that a name own at least one RR differs semantically from QUERY, in that QUERY would return <NOERROR,ANCOUNT=0> rather than NXDOMAIN if queried for an RRset at this name, while UPDATE's prerequisite condition [Section 2.4.4] would NOT be satisfied.

7.8. It is possible for a UDP response to be lost in transit and for a request to be retried due to a timeout condition. In this case an UPDATE that was successful the first time it was received by the primary master might ultimately appear to have failed when the response to a duplicate request is finally received by the requestor. (This is because the original prerequisites may no longer be satisfied after the update has been applied.) For this reason, requestors who require an accurate response code must use TCP.

7.9. Because a requestor who requires an accurate response code will initiate their UPDATE transaction using TCP, a forwarder who receives a request via TCP must forward it using TCP.

7.10. Deferral of SOA SERIAL autoincrements is made possible so that serial numbers can be conserved and wraparound at 2**32 can be made an infrequent occurance. Visible (to DNS clients) SOA SERIALs need to differ if the zone differs. Note that the Authority Section SOA in a QUERY response is a form of visibility, for the purposes of this prerequisite.

7.11. A zone's SOA SERIAL should never be set to zero (0) due to interoperability problems with some older but widely installed implementations of DNS. When incrementing an SOA SERIAL, if the result of the increment is zero (0) (as will be true when wrapping around 2**32), it is necessary to increment it again or set it to one (1). See [RFC1982] for more detail on this subject.

7.12. Due to the TTL minimalization necessary when caching an RRset, it is recommended that all TTLs in an RRset be set to the same value. While the DNS Message Format permits variant TTLs to exist in the same RRset, and this variance can exist inside a zone, such variance will have counterintuitive results and its use is discouraged.

7.13. Zone cut management presents some obscure corner cases to the add and delete operations in the Update Section. It is possible to delete an NS RR as long as it is not the last NS RR at the root of a zone. If deleting all RRs from a name, SOA and NS RRs at the root of a zone are unaffected. If deleting RRsets, it is not possible to delete either SOA or NS RRsets at the top of a zone. An attempt to add an SOA will be treated as a replace operation if an SOA already exists, or as a no-op if the SOA would be new.

7.14. No semantic checking is required in the primary master server when adding new RRs. Therefore a requestor can cause CNAME or NS or any other kind of RR to be added even if their target name does not exist or does not have the proper RRsets to make the original RR useful. Primary master servers that DO implement this kind of checking should take great care to avoid out-of-zone dependencies (whose veracity cannot be authoritatively checked) and should implement all such checking during the prescan phase.

7.15. Nonterminal or wildcard CNAMEs are not well specified by [RFC1035] and their use will probably lead to unpredictable results. Their use is discouraged.

7.16. Empty nonterminals (nodes with children but no RRs of their own) will cause <NOERROR,ANCOUNT=0> responses to be sent in response to a query of any type for that name. There is no provision for empty terminal nodes -- so if all RRs of a terminal node are deleted, the name is no longer in use, and queries of any type for that name will result in an NXDOMAIN response.

7.17. In a deep AXFR dependency graph, it has not historically been an error for slaves to depend mutually upon each other. This configuration has been used to enable a zone to flow from the primary master to all slaves even though not all slaves have continuous connectivity to the primary master. UPDATE's use of the AXFR dependency graph for forwarding prohibits this kind of dependency loop, since UPDATE forwarding has no loop detection analagous to the SOA SERIAL pretest used by AXFR.

7.18. Previously existing names which are occluded by a new zone cut are still considered part of the parent zone, for the purposes of zone transfers, even though queries for such names will be referred to the new subzone's servers. If a zone cut is removed, all parent zone names that were occluded by it will again become visible to queries. (This is a clarification of [RFC1034].)

7.19. If a server is authoritative for both a zone and its child, then queries for names at the zone cut between them will be answered authoritatively using only data from the child zone. (This is a clarification of [RFC1034].)

7.20. Update ordering using the SOA RR is problematic since there is no way to know which of a zone's NS RRs represents the primary master, and the zone slaves can be out of date if their SOA.REFRESH timers have not elapsed since the last time the zone was changed on the primary master. We recommend that a zone needing ordered updates use only servers which implement NOTIFY (see [RFC1996]) and IXFR (see [RFC1995]), and that a client receiving a prerequisite error while attempting an ordered update simply retry after a random delay period to allow the zone to settle.

8 - Security Considerations

8.1. In the absence of [RFC2137] or equivilent technology, the protocol described by this document makes it possible for anyone who can reach an authoritative name server to alter the contents of any zones on that server. This is a serious increase in vulnerability from the current technology. Therefore it is very strongly recommended that the protocols described in this document not be used without [RFC2137] or other equivalently strong security measures, e.g. IPsec.

8.2. A denial of service attack can be launched by flooding an update forwarder with TCP sessions containing updates that the primary master server will ultimately refuse due to permission problems. This arises due to the requirement that an update forwarder receiving a request via TCP use a synchronous TCP session for its forwarding operation. The connection management mechanisms of [RFC1035 4.2.2] are sufficient to prevent large scale damage from such an attack, but not to prevent some queries from going unanswered during the attack.

Acknowledgements

We would like to thank the IETF DNSIND working group for their input and assistance, in particular, Rob Austein, Randy Bush, Donald Eastlake, Masataka Ohta, Mark Andrews, and Robert Elz. Special thanks to Bill Simpson, Ken Wallich and Bob Halley for reviewing this document.

References

[RFC1035]
Mockapetris, P., "Domain Names - Implementation and Specification", STD 13, RFC 1035, USC/Information Sciences Institute, November 1987.

[RFC1982]
Elz, R., "Serial Number Arithmetic", RFC 1982, University of Melbourne, August 1996.

[RFC1995]
Ohta, M., "Incremental Zone Transfer", RFC 1995, Tokyo Institute of Technology, August 1996.

[RFC1996]
Vixie, P., "A Mechanism for Prompt Notification of Zone Changes", RFC 1996, Internet Software Consortium, August 1996.

[RFC2065]
Eastlake, D., and C. Kaufman, "Domain Name System Protocol Security Extensions", RFC 2065, January 1997.

[RFC2137]
Eastlake, D., "Secure Domain Name System Dynamic Update", RFC 2137, April 1997.

Authors' Addresses

Yakov Rekhter
Cisco Systems
170 West Tasman Drive
San Jose, CA 95134-1706

Phone: +1 914 528 0090
EMail: yakov@cisco.com

Susan Thomson
Bellcore
445 South Street
Morristown, NJ 07960

Phone: +1 201 829 4514
EMail: set@thumper.bellcore.com

Jim Bound
Digital Equipment Corp.
110 Spitbrook Rd ZK3-3/U14
Nashua, NH 03062-2698

Phone: +1 603 881 0400
EMail: bound@zk3.dec.com

Paul Vixie
Internet Software Consortium
Star Route Box 159A
Woodside, CA 94062

Phone: +1 415 747 0204
EMail: paul@vix.com