Internet Draft
Expires June 23, 2000
draft-ietf-rps-dist-06

Curtis Villamizar
Avici Systems
Cengiz Alaettinoglu
ISI
Ramesh Govindan
ISI
David M. Meyer
Cisco
December 23, 1999

Routing Policy System Replication

Status of this Memo

This document is an Internet-Draft and is in full conformance with all provisions of Section 10 of RFC2026.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet- Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

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 RFC 2119.

Abstract

The RIPE database specifications and RPSL define languages used as the basis for representing information in a routing policy system. A repository for routing policy system information is known as a routing registry. A routing registry provides a means of exchanging information needed to address many issues of importance to the operation of the Internet. The implementation and deployment of a routing policy system must maintain some degree of integrity to be of any use. The Routing Policy

System Security RFC [3] addresses the need to assure integrity of the data by proposing an authentication and authorization model. This document addresses the need to distribute data over multiple repositories and delegate authority for data subsets to other repositories without compromising the authorization model established in Routing Policy System Security RFC.

Contents

1 Overview

A routing registry must maintain some degree of integrity to be of any use. The IRR is increasingly used for purposes that have a stronger requirement for data integrity and security. There is also a desire to further decentralize the IRR. This document proposes a means of decentralizing the routing registry in a way that is consistent with the usage of the IRR and which avoids compromising data integrity and security even if the IRR is distributed among less trusted repositories.

Two methods of authenticating the routing registry information have been proposed.

authorization and authentication checks on transactions: The integrity of the routing registry data is insured by repeating authorization checks as transactions are processed. As transactions are flooded each remote registry has the option to repeat the authorization and authentication checks. This scales with the total number of changes to the registry regardless of how many registries exist. When querying, the integrity of the repository must be such that it can be trusted. If an organization is unwilling to trust any of the available repositories or mirrors they have the option to run their own mirror and repeat authorization checks at that mirror site. Queries can then be directed to a mirror under their own administration which presumably can be trusted.

signing routing registry objects: An alternate which appears on the surface to be attractive is signing the objects themselves. Closer examination reveals that the approach of signing objects by itself is flawed and when used in addition to signing transactions and rechecking authorizations as changes are made adds nothing. In order for an insertion of critical objects such as inetnums and routes to be valid, authorization checks must be made which allow the insertion. The objects on which those authorization checks are made may later change. In order to later repeat the authorization checks the state of other objects, possibly in other repositories would have to be known. If the repository were not trusted then the change history on the object would have to be traced back to the object's insertion. If the repository were not trusted, the change history of any object that was depended upon for authorization would also have to be rechecked. This trace back would have to go back to the epoch or at least to a point where only trusted objects were being relied upon for the authorizations. If the depth of the search is at all limited, authorization could be falsified simply by exceeding the search depth with a chain of authorization references back to falsified objects. This would be grossly inefficient. Simply verifying that an object is signed provides no assurance that addition of the object addition was properly authorized.

A minor distinction is made between a repository and a mirror. A repository has responsibility for the initial authorization and authentication checks for transactions related to its local objects which are then flooded to adjacent repositories. A mirror receives flooded transactions from remote repositories but is not the authoritative source for any objects. From a protocol standpoint, repositories and mirrors appear identical in the flooding topology.

Either a repository or a mirror may recheck all or a subset of transactions that are flooded to it. A repository or mirror may elect not to recheck authorization and authentication on transactions received from a trusted adjacency on the grounds that the adjacent repository is trusted and would not have flooded the information unless authorization and authentication checks had been made.

If it can be arranged that all adjacencies are trusted for a given mirror, then there is no need to implement the code to check authorization and authentication. There is only a need to be able to check the signatures on the flooded transactions of the adjacent repository. This is an important special case because it could allow a router to act as a mirror. Only changes to the registry database would be received through flooding, which is a very low volume. Only the signature of the adjacent mirror or repository would have to be checked.

2 Data Representation

RPSL provides a complete description of the contents of a routing repository [1]. Many RPSL data objects remain unchanged from the RIPE, and RPSL references the RIPE-181 specification as recorded in RFC-1786 [2]. RPSL provides external data representation. Data may be stored differently internal to a routing registry. The integrity of the distributed registry data requires the use of the authorization and authentication additions to RPSL described in [3].

Some additions to RPSL are needed to locate all of the repositories after having located one of them and to make certain parameters selectable on a per repository basis readily available. These additions are described in Section 5.

Some form of encapsulation must be used to exchange data. The de-facto encapsulation has been that which the RIPE tools accept, a plain text file or plain text in the body of an RFC-822 formatted mail message with information needed for authentication derived from the mail headers. Merit has slightly modified this using the PGP signed portion of a plain text file or PGP signed portion of the body of a mail message.

The exchange that occurs during flooding differs from the initial submission. In order to repeat the authorization checks the state of all repositories containing objects referenced by the authorization checks needs to be known. To accomplish this a sequence number is associated with each transaction in a repository and the flooded transactions must contain the sequence number of each repository on which authorization of the transaction depends.

In order to repeat authorization checks it must be possible to retrieve back revisions of objects. How this is accomplished is a matter local to the implementation. One method which is quite simple is to keep the traversal data structures to all current objects even if the state is deleted, keep the sequence number that the version of the object became effective and keep back links to prior versions of the objects. Finding a prior version of an object involves looking back through the references until the sequence number of the version of the object is less than or equal to the sequence number being searched for.

The existing very simple forms of encapsulation are adequate for the initial submission of a database transaction and should be retained as long as needed for backward compatibility. A more robust encapsulation and submission protocol, with optional confirmation is defined in Section 6.1. An encapsulation suitable for exchange of transaction between repositories is addressed in Section 6. Query encapsulation and protocol is outside the scope of this document.

3 Authentication and Authorization

Control must be exercised over who can make changes and what changes they can make. The distinction of who vs what separates authentication from authorization.

- Authentication is the means to determine who is attempting to make a change.
- Authorization is the determination of whether a transaction passing a specific authentication check is allowed to perform a given operation.

A submitted transaction contains a claimed identity. Depending on the type of transaction, the authorization will depend on related objects. The "mnt-by", "mnt-routes", or "mnt-lower" attributes in those related objects reference "maintainer" objects. Those maintainer objects contain "auth" attributes. The auth attributes contain an authorization method and data which generally contains the claimed identity and some form of public encryption key used to authenticate the claim.

Authentication is done on transactions. Authentication should also be done between repositories to insure the integrity of the information exchange. In order to comply with import, export, and use restrictions throughout the world no encryption capability is specified. Transactions must not be encrypted because it may be illegal to use decryption software in some parts of the world.

4 Repository Hierarchy

With multiple repositories, "repository" objects are needed to propagate the existence of new repositories and provide an automated means to determine the supported methods of access and other characteristics of the repository. The repository object is described in Section 5.

In each repository there should be a special repository object named ROOT. This should point to the root repository or to a higher level repository. This is to allow queries to be directed to the local repository but refer to the full set of registries for resolution of hierarchically allocated objects.

Each repository may have an "expire" attribute. The expire attribute is used to determine if a repository must be updated before a local transaction that depends on it can proceed.

The repository object also contains attributes describing the access methods and supported authentication methods of the repository. The "query-address" attribute provides a host name and a port number used to direct queries. The "response-auth-type" attribute provides the authentication types that may be used by the repository when responding to queries. The "submit-address" attribute provides a host name and a port number used to submit objects to the repository. The "submit-auth-type" attribute provides the authentication types that may be used by the repository when responding to submissions.

5 Additions to RPSL

There are very few additions to RPSL defined here. The additions to RPSL are referred to as RPSL "objects". They reside in the repository database and can be retrieved with ordinary queries. Objects consist of "attributes", which are name/value pairs. Attributes may be mandatory or optional. They may be single or multiple. One or more attributes may be part of a key field. Some attributes may have the requirement of being unique.

Most of the data formats described in this document are encapsulations used in transaction exchanges. These are referred to as "meta-objects". These "meta-objects", unlike RPSL "objects" do not reside in the database but some must be retained in a transaction log. A similar format is used to represent "meta-objects". They also consist of "attributes" which are name/value pairs.

This section contains all of the additions to RPSL described in this document. This section describes only RPSL objects. Other sections described only meta-objects.

5.1 repository object

A root repository must be agreed upon. Ideally such a repository would contain only top level delegations and pointers to other repositories used in these delegations. It would be wise to allow only cryptographically strong transactions in the root repository [3].

The root repository contains references to other repositories. An object of the following form identifies another repository.

```
repository:         RIPE
query-address:      whois://whois.ripe.net
response-auth-type: PGPKEY-23F5CE35 # pointer to key-cert object
response-auth-type: none
remarks:            you can request rsa signature on queries
remarks:            PGP required on submissions
submit-address:     mailto://auto-dbm@ripe.net
submit-address:     rps-query://whois.ripe.net:43
submit-auth-type:   pgp-key, crypt-pw, mail-from
remarks:            these are the authentication types supported
mnt-by:             maint-ripe-db
expire:             0000 04:00:00
heartbeat-interval: 0000 01:00:00
...
remarks:            admin and technical contact, etc
source:             IANA
```

The attributes of the repository object are listed below.

```
repository      key  mandatory  single
query-address        mandatory  multiple
```

```
  response-auth-type   mandatory  multiple
  submit-address       mandatory  multiple
  submit-auth-type     mandatory  multiple
  repository-cert      mandatory  multiple
  expire               mandatory  single
  heartbeat-interval   mandatory  single
  descr                optional   multiple
  remarks              optional   multiple
  admin-c              mandatory  multiple
  tech-c               mandatory  multiple
  notify               optional   multiple
  mnt-by               mandatory  multiple
  changed              mandatory  multiple
  source               mandatory  single
```

In the above object type only a small number of the attribute types are new. These are:

repository This attribute provides the name of the repository. This is the key field for the object and is single and must be globally unique. This is the same name used in the source attribute of all objects in that repository.

query-address This attribute provides a url for directing queries. "rps-query" or "whois" can be used as the protocol identifier.

response-auth-type This attribute provides an authentication type that may be used by the repository when responding to user queries. Its syntax and semantics is same as the auth attribute of the maintainer class.

submit-address This attribute provides a url for submitting objects to the repository.

submit-auth-type This attribute provides the authentication types that are allowed by the repository for users when submitting registrations.

repository-cert This attribute provides a reference to a public key certificate in the form of an RPSL key-cert object. This attribute can be multiple to allow the repository to use more than one method of signature.

heartbeat-interval Heartbeat meta-objects are sent by this repository at the rate of one heartbeat meta-object per the interval indicated. The value of this attribute shall be expressed in the form "dddd hh:mm:ss", where the "dddd" represents days, "hh" represents hours, "mm" minutes and "ss" seconds.

expire If no heartbeat or new registrations are received from a repository for expire period, objects from this repository should be considered non-authoritative, and cannot be used for authorization purposes. The value of this attribute shall be expressed in the form "dddd hh:mm:ss", where the "dddd" represents days, "hh" represents hours, "mm" minutes and "ss" seconds. This value should be bigger than heartbeat-interval.

Please note that the "heartbeat" meta-objects mentioned above, like other meta-objects described in this document are part of the protocol to exchange information but are not placed in the database itself. See Section 7.3.2 for a description of the heartbeat meta-object.

The remaining attributes in the repository object are defined in RPSL.

5.2 delegated attribute

For many RPSL object types a particular entry should appear only in one repository. These are the object types for which there is a natural hierarchy, "as-block", "aut-num", "inetnum", and "route". In order to facilitate putting an object in another repository, a "delegated" attribute is added.

delegated The delegated attribute is allowed in any object type with a hierarchy. This attribute indicates that further searches for object in the hierarchy must be made in one or more alternate repositories. The current repository may be listed. The ability to list more than one repository serves only to accommodate grandfathered objects (those created prior to using an authorization model). The value of a delegated attribute is a list of repository names.

If an object contains a "delegated" attribute, an exact key field match of the object may also be contained in each repository listed in the "delegated" attribute. For the purpose of authorizing changes only the "mnt-by" in the object in the repository being modified is considered.

The following is an example of the use of a "delegated" attribute.

```
inetnum:        193.0.0.0 - 193.0.0.255
delegated:      RIPE
...
source:         IANA
```

This inetnum simply delegates the storage of any more specific inetnum objects overlapping the stated range to the RIPE repository. An exact match of this inetnum may also exist in the RIPE repository to provide hooks for the attributes referencing maintainer objects. In this case, when adding objects to the RIPE repository, the "mnt-lower", "mnt-routes", and "mnt-by" fields in the IANA inetnum object will not be considered, instead the values in the RIPE copy will be used.

5.3 integrity attribute

The "integrity" attribute can be contained in any RPSL object. It is intended solely as a means to facilitate a transition period during which some data has been moved from repositories prior to the use of a strong authorization model and is therefore questionable, or when some repositories are not properly checking authorization.

The "integrity" attribute may have the values "legacy", "no-auth", "auth-failed", or "authorized". If absent, the integrity is considered to be "authorized". The integrity values have the following meanings:

legacy: This data existed prior to the use of an adequate authorization model. The data is highly suspect.

no-auth: This data was added to a repository during an initial transition use of an authorization model but authorization depended on other objects whose integrity was not "authorized". Such an addition is being allowed during the transition but would be disallowed later.

auth-failed: The authoritative repository is not checking authorization. Had it been doing so, authorization would have failed. This attribute may be added by a repository that is mirroring before placing the object in its local storage, or can add this attribute to an encapsulating meta-object used to further propagate the transaction. If the failure to enforce authorization is intentional and part of a transition (for example, issuing warnings only), then the authoritative repository may add this attribute to the encapsulating meta-object used to further propagate the transaction.

authorized: Authorization checks were passed. The maintainer contained a "referral-by" attribute, a form of authentication deemed adequate by the repository was used, and all objects that were needed for authorization were objects whose integrity was "authorized".

Normally once an object is added to a repository another object cannot overwrite it unless authorized to do so by the maintainers referenced by the "mnt-by" attributes in the object itself. If the integrity attribute is anything but "authorized", an object can be overwritten or deleted by any transaction that would have been a properly authorized addition had the object of lesser integrity not existed.

During such a transition grandfathered data and data added without proper authorization becomes advisory until a properly authorized addition occurs. After transition additions of this type would no longer be accepted. Those objects already added without proper authorization would remain but would be marked as candidates for replacement.

6 Interactions with a Repository or Mirror

This section presents an overview of the transaction distribution mechanisms. The detailed format of the meta-objects for encapsulating and distributing transactions, and the rules for processing meta-objects are described in Section 7. There are a few different types of interactions between routing repositories or mirrors.

Initial submission of transactions: Transactions may include additions, changes, and deletions. A transaction may operate on more than one object and must be treated as an atomic operation. By definition initial submission of transactions is not applicable to a mirror. Initial submission of transactions is described in Section 6.1.

Redistribution of Transactions: The primary purpose of the interactions between registries is the redistribution of transactions. There are a number of ways to redistribute transactions. This is discussed in Section 6.2.

Queries: Query interactions are outside the scope of this document.

Transaction Commit and Confirmation: Repositories may optionally implement a commit protocol and a completion indication that gives the submitter of a transaction a response that indicates that a transaction has been successful and will not be lost by a crash of the local repository. A submitter may optionally request such a confirmation. This is discussed in Section 6.3.

6.1 Initial Transaction Submission

The simplest form of transaction submission is an object or set of objects submitted with RFC-822 email encapsulation. This form is still supported for backwards compatibility. A preferred form allows some meta-information to be included in the submission, such as a preferred form of confirmation. Where either encapsulation is used, the submitter will connect to a host and port specified in the repository object. This allows immediate confirmation. If an email interface similar to the interface provided by the existing RIPE code is desired, then an external program can provide the email interface.

The encapsulation of a transaction submission and response is described in detail in Section 7.

6.2 Redistribution of Transactions

Redistribution of transactions can be accomplished using one of:

1. A repository snapshot is a request for the complete contents of a given repository. This is usually done when starting up a new repository or mirror or when recovering from a disaster, such as a disk crash.
2. A transaction sequence exchange is a request for a specific set of transactions. Often the request is for the most recent sequence number known to a mirror to the last transactions. This is used in polling.
3. Transaction flooding is accomplished through a unicast adjacency.

This section describes the operations somewhat qualitatively. Data formats and state diagrams are provided in Section 7.

6.3 Transaction Commit and Confirmation

If a submission requires a strong confirmation of completion, or if a higher degree of protection against false positive confirmation is desired as a matter of repository policy, a commit may be performed.

A commit request is a request from the repository processing an initial transaction submission to another repository to confirm that they have been able to advance the transaction sequence up to the sequence number immediately below the transaction in the request and are willing to accept the transaction in the request as a further advance in the sequence. This indicates that either the authorization was rechecked by the responding repository and passed or that the responding repository trusts the requesting repository and has accepted the transaction.

A commit request can be sent to more than one alternate repository. One commit completion response is sufficient to respond to the submitter with a positive confirmation that the transaction has been completed. However, the repository or submitter may optionally require more than one.

7 Data Format Summaries, Transaction Encapsulation and Processing

RIPE-181 [2] and RPSL [1] data is represented externally as ASCII text. Objects consist of a set of attributes. Attributes are name/value pairs. A single attribute is represented as a single line with the name followed by a colon followed by whitespace characters (space, tab, or line continuation) and followed by the value. Within a value all consecutive whitespace characters is equivalent to a single space. Line continuation is supported by putting a white space or '+' character to the beginning of the continuation lines. An object is externally represented as a sequence of attributes. Objects are separated by blank lines.

Protocol interactions between registries are activated by passing "meta objects". Meta objects are not part of RPSL but conform to RPSL object representation. They serve mostly as delimiters to the protocol messages or to carry the request for an operation.

7.1 Transaction Submit and Confirm

The de-facto method for submitting database changes has been via email. This method should be supported by an external application. Merit has added the pgp-from authentication method to the RADB (replaced by "pgpkey" in [4]), where the mail headers are essentially ignored and the body of the mail message must be PGP signed.

This specification defines a different encapsulation for transaction submission. When submitting a group of objects to a repository, a user MUST append to that group of objects, exactly one "timestamp" and one or more "signature" meta-objects, **in that order**.

The "timestamp" meta-object contains a single attribute:

timestamp This attribute is mandatory and single-valued. This attribute specifies the time at which the user submits the transaction to the repository. The format of this attribute is "YYYYMMDD hh:mm:ss [+/-]xx:yy", where "YYYY" specifies the four digit year, "MM" represents the month, "DD" the date, "hh" the hour, "mm" the minutes, "ss" the seconds of the timestamp, and "xx" and "yy" represents the hours and minutes respectively that that timestamp is ahead or behind UTC.

A repository may reject a transaction which does not include the "timestamp" meta-object. The timestamp object is used to prevent replaying registrations. How this is actually used is a local matter. For example, a repository can accept a transaction only if the value of the timestamp attribute is greater than the timestamp attribute in the previous registration received from this user (maintainer), or the repository may only accept transactions with timestamps within its expire window.

Each "signature" meta-object contains a single attribute:

signature This attribute is mandatory and single-valued. This attribute, a block of free text, contains the signature corresponding to the authentication method used for the transaction.

When the authentication method is a cryptographic hash (as in PGP-based authentication), the signature must include all text upto (but not including) the last blank line before the first "signature" meta-object.

A repository must reject a transaction that does not include any "signature" meta-object.

The group of objects submitted by the user, together with the "timestamp" and "signature" meta-objects, constitute the "submitted text" of the transaction.

The protocol used for submitting a transaction, and for receiving confirmation of locally committed transactions, is not specified in this document. This protocol may define additional encapsulations around the submitted text. The rest of this section gives an example of one such protocol. Implementations are free to choose another encapsulation.

The meta-objects "transaction-submit-begin" and "transaction-submit-end" delimit a transaction. A transaction is handled as an atomic operation. If any part of the transaction fails none of the changes take effect. For this reason a transaction can only operate on a single database.

A socket connection is used to request queries or submit transactions. An email interface may be provided by an external program that connects to the socket. A socket connection must use the "transaction-submit-begin" and "transaction-submit-end" delimiters but can request a legacy style confirmation. Multiple transactions may be sent prior to the response for any single transaction. Transactions may not complete in the order sent.

The "transaction-submit-begin" meta-object may contain the following attributes.

transaction-submit-begin This attribute is mandatory and single. The value of the attribute contains name of the database and an identifier that must be unique over the course of the socket connection.

response-auth-type This attribute is optional and multiple. The remainder of the line specifies an authentication type that would be acceptable in the response. This is used to request a response cryptographically signed by the repository.

transaction-confirm-type This attribute is optional and single. A confirmation type keyword must be provided. Keywords are "none", "legacy", "normal", "commit". The confirmation type can be followed by the option "verbose".

The "transaction-submit-end" meta-object consists of a single attribute by the same name. It must contain the same database name and identifier as the corresponding "transaction-submit-begin" attribute.

Unless the confirmation type is "none" a confirmation is sent. If the confirmation type is "legacy", then an email message of the form currently sent by the RIPE database code will be returned on the socket (suitable for submission to the sendmail program).

A "normal" confirmation does not require completion of the commit protocol. A "commit" confirmation does. A "verbose" confirmation may contain additional detail.

A transaction confirmation is returned as a "transaction-confirm" meta-object. The "transaction-confirm" meta-object may have the following attributes.

transaction-confirm This attribute is mandatory and single. It contains the database name and identifier associated with the transaction.

confirmed-operation This attribute is optional and multiple. It contains one of the keywords "add", "delete" or "modify" followed by the object type and key fields of the object operated on.

commit-status This attribute is mandatory and single. It contains one of the keywords "succeeded", "error", or "held". The "error" keyword may be followed by an optional text string. The "held" keyword is returned when a repository containing a dependent object for authorization has expired.

7.2 Redistribution of Transactions

In order to redistribute transactions, each repository maintains a TCP connection with one or more other repositories. After locally committing a submitted transaction, a repository assigns a sequence number to the transaction, signs and encapsulates the transaction, and then sends one copy to each neighboring (or "peer") repository. In turn, each repository authenticates the transaction (as described in Section 7.6), may re-sign the transaction and redistributes the transaction to its neighbors. We use the term "originating repository" to distinguish the repository that redistributes a locally submitted transaction.

This document also specifies two other methods for redistributing transactions to other repositories: a database snapshot format used for initializing a new registry, and a polling technique used by mirrors.

In this section, we first describe how a repository may encapsulate the submitted text of a transaction. We then describe the protocol for flooding transactions or polling for transactions, and the database snapshot contents and format.

7.3 Redistribution Protocol Description

The originating repository must first authenticate a submitted transaction using methods described in [3].

Before redistributing a transaction, the originating repository must encapsulate the submitted text of the transaction with several meta-objects, which are described below.

The originating repository must prepend the submitted text with exactly one "transaction-label" meta-object. This meta-object contains the following attributes:

transaction-label This attribute is mandatory and single. The value of this attribute conforms to the syntax of an RPSL word, and represents a globally unique identifier for the database to which this transaction is added.

sequence This attribute is mandatory and single. The value of this attribute is an RPSL integer specifying the sequence number assigned by the originating repository to the transaction. Successive transactions distributed by the same originating repository have successive sequence numbers. The first transaction originated by a registry is assigned a sequence number 1. Each repository must use sequence numbers drawn from a range at least as large as 64 bit unsigned integers.

timestamp This attribute is mandatory and single-valued. This attribute specifies the time at which the originating repository encapsulates the submitted text. The format of this attribute is "YYYYMMDD hh:mm:ss [+/-]xx:yy", where "YYYY" specifies the four digit year, "MM" represents the month, "DD" the date, "hh" the hour, "mm" the minutes, "ss" the seconds of the timestamp, and "xx" and "yy" represents the hours and minutes respectively that that timestamp is ahead or behind UTC.

integrity This attribute is optional and single-valued. It may have the values "legacy", "no-auth", "auth-failed", or "authorized". If absent, the integrity is considered to be "authorized".

The originating repository may append to the submitted text one or more "auth-dependency" meta-objects. These meta-objects are used to indicate which other repositories' objects were used by the originating registry to authenticate the submitted text. The "auth-dependency" meta-objects should be ordered from the most preferred repository to the least preferred repository. This order is used by a remote repository to tie break between the multiple registrations of an object with the same level of integrity. The "auth-dependency" meta-object contains the following attributes:

auth-dependency This attribute mandatory and single-valued. It equals a repository name from which an object is used to authorize/authenticate this transaction.

sequence This attribute mandatory and single-valued. It equals the transaction sequence number of the dependent repository known at the originating repository at the time of processing this transaction.

timestamp This attribute mandatory and single-valued. It equals the timestamp of the dependent repository known at the originating repository at the time of processing this transaction.

If the originating repository needs to modify submitted objects in a way that the remote repositories can not re-create, it can append an "override-objects" meta-object followed by the modified versions of these objects. An example modification can be auto assignment of NIC handles. The "override-objects" meta-object contains the following attributes:

override-objects A free text remark.

Other repositories may or may not honor override requests, or limit the kinds of overrides they allow.

Following this, the originating repository must append exactly one "repository-signature" meta-object. The "repository-signature" meta-object contains the following attributes:

repository-signature This attribute is mandatory and single-valued. It contains the name of the repository.

integrity This attribute is optional and single-valued. It may have the values "legacy", "no-auth", "auth-failed", or "authorized". If absent, the value is same as the value in the transaction-label. If a different value is used, the value here takes precedence.

signature This attribute is optional and single-valued. This attribute, a block of free text, contains the repository's signature using the key in the repository-cert attribute of the repository object. When the authentication method is a cryptographic hash (as in PGP-based authentication), the signature must include all text upto (but not including) this attribute. That is, the "repository-signature" and "integrity" attributes of this object are included. This attribute is optional since cryptographic authentication may not be available everywhere. However, its use where it is available is highly reccomended.

A repository must reject a redistributed transaction that does not include any "repository-signature" meta-object.

The transaction-label, the submitted text, the dependency objects, the override-objects, the overriden objects, and the repository's signature together constitute what we call the "redistributed text".

In preparation for redistributing the transaction to other repositories, the originating repository must perform the following protocol encapsulation. This protocol encapsulation may involve transforming the redistributed text according to one of the "transfer-method"s described below.

The transformed redistributed text is first prepended with exactly one "transaction-begin" meta-object. One newline character separates this meta-object from the redistributed text. This meta-object has the following attributes:

transaction-begin This attribute is mandatory and single. The value of this attribute is the length, in bytes, of the transformed redistributed text.

transfer-method This attribute is optional and single-valued. Its value is either "gzip", or "plain". The value of the attribute describes the kind of text encoding that the repository has performed on the redistributed text. If this attribute is not specified, its value is assumed to be "plain". An implementation must be capable of encoding and decoding both of these types.

The "transaction-begin" meta-object and the transformed redistributed text constitute what we call the "transmitted text". The originating repository may distribute the transmitted text to one or more peer repositories.

When a repository receives the transmitted text of a transaction, it must perform the following steps. After performing the following steps, a transaction may be marked successful or failed.

1. It must decapsulate the "transaction-begin" meta-object, then decode the original redistributed text according to the value of the transfer-method attribute specified in the "transaction-begin" meta-object.
2. It should then extract the "transaction-label" meta-object from the transmitted text. If this transaction has already been processed, or is currently being held, the repository must silently discard this incarnation of the same transaction.
3. It should verify that the signature of the originating repository matches the first "repository-signature" meta-object in the redistributed text following the "auth-dependency" meta-objects.
4. If not all previous (i.e., those with a lower sequence number) transactions from the same repository have been received or completely processed, the repository must "hold" this transaction.

5. It may check whether any subsequent "repository-signature" meta-objects were appended by a trusted repository. If so, this indicates that the trusted repository verified the transaction's integrity and marked its conclusion in the integrity attribute of this object. The repository may verify the trusted repositories signature and also mark the transaction with the same integrity, and skip the remaining steps.

6. It should verify the syntactic correctness of the transaction. An implementation may allow configurable levels of syntactic conformance with RPSL [1]. This enables RPSL extensions to be incrementally deployed in the distributed registry scheme.

7. The repository must authorize and authenticate this transaction. To do this, it may need to reference objects and transactions from other repositories. If these objects are not available, the repository must "hold" this transaction as described in Section 7.6, until it can be authorized and authenticated later. In order to verify authorization/authentication of this transaction, the repository must not use an object from a repository not mentioned in an "auth-dependency" meta-obect. The repository should also only use the latest objects (by rolling back to earlier versions if necessary) which are within the transaction sequence numbers of the "auth-dependency" meta-objects.

A non-originating repository must redistribute a failed transaction in order not to cause a gap in the sequence. (If the transaction was to fail at the originating registry, it would simply not be assigned a sequence number).

To the redistributed text of a transaction, a repository may append another "repository-signature" meta-object. This indicates that the repository has verified the transaction's integrity and marked it in the "integrity" attribute of this object. The signature covers the new redistributed text from (and including) the transaction-label object to this object's signature attribute (including the "repository-signature" and "integrity" attributes of this object, but excluding the "signature" attribute). The original redistributed text, together with the new "repository-signature" meta-object constitutes the modified redistributed text.

To redistribute a successful or failed transaction, the repository must encapsulate the (original or modified) redistributed text with a "transaction-begin" object. This step is essentially the same as that performed by the originating repository (except that the repository is free to use a different "transfer-method" from the one that was in the received transaction.

7.3.1 Explicitly Requesting Transactions

A repository may also explicitly request one or more transactions belonging to a specified originating repository. This is useful for catching up after a repository has been off-line for a period of time. It is also useful for mirrors which intermittently poll a repository for recently received transactions.

To request a range of transactions from a peer, a repository must send a "transaction-request" meta-object to the peer. A "transaction-request" meta-object may contain the following attributes:

transaction-request This attribute is mandatory and single. It contains the name of the database whose transactions are being requested.

sequence-begin This attribute is optional and single. It contains the sequence number of the first transaction being requested.

sequence-end This attribute is optional and single. It contains the sequence number of the last transaction being requested.

Upon receiving a “transaction-request” object, a repository performs the following actions. If the “sequence-begin” attribute is not specified, the repository assumes the request first sequence number to be 1. The last sequence number is the lesser of the value of the “sequence-end” attributed and the highest completed transaction in the corresponding database. The repository then, in order, transmits the requested range of transactions. Each transaction is prepared exactly according to the rules for redistribution specified in Section 7.3.

After transmitting all the transactions, the peer repository must send a “transaction-response” meta-object. This meta-object has the following attributes:

transaction-response This attribute is mandatory and single. It contains the name of the database whose transactions are were requested.

sequence-begin This attribute is optional and mandatory. It contains the value of the “sequence-begin” attribute in the original request. It is omitted if the corresponding attribute was not specified in the original request.

sequence-end This attribute is optional and mandatory. It contains the value of the “sequence-end attribute in the original request. It is omitted if the corresponding attribute was not specified in the original request.

After receiving a “transaction-response” meta-object, a repository may tear down the TCP connection to its peer. This is useful for mirrors that intermittently resynchronize transactions with a repository. If the TCP connection stays open, repositories exchange subsequent transactions according to the redistribution mechanism specified in Section 7.3. While a repository is responding to a transaction-request, it MAY forward heartbeats and other transactions from the requested repository towards the requestor.

7.3.2 Heartbeat Processing

Each repository that has originated at least one transaction must periodically send a “heartbeat” meta-object. The interval between two successive transmissions of this meta-object is configurable but must be less than 1 day. This meta-object serves to indicate the liveness of a particular repository. The repository liveness determines how long transactions are held (See Section 7.6).

The “heartbeat” meta-object contains the following attributes:

heartbeat This attribute is mandatory and single. It contains the name of the repository which originates this meta-object.

sequence This attribute is mandatory and single. It contains the highest transaction sequence number that has been assigned by the repository.

timestamp This attribute is mandatory and single. It contains the time at which this meta-object was generated. The format of this attribute is “YYYYMMDD hh:mm:ss [+/-]xx:yy”, where

"YYYY" specifies the four digit year, "MM" represents the month, "DD" the date, "hh" the hour, "mm" the minutes, "ss" the seconds of the timestamp, and "xx" and "yy" represents the hours and minutes respectively that that timestamp is ahead or behind UTC.

Upon receiving a heartbeat meta-object, a repository must first check the timestamp of the latest previously received heartbeat message. If that timestamp exceeds the timestamp in the received heartbeat message, the repository must silently discard the heartbeat message. Otherwise, it must record the timestamp and sequence number in the heartbeat message, and redistribute the heartbeat message, without modification, to each of its peer repositories.

If the heartbeat message is from a repository previously unknown to the recipient, the recipient may send a "transaction-request" to one or more of its peers to obtain all transactions belonging to the corresponding database. If the heartbeat message contains a sequence number higher than the highest sequence number processed by the recipient, the recipient may send a "transaction-request" to one or more of its peers to obtain all transactions belonging to the corresponding database.

7.4 Transaction Commit

Submitters may require stronger confirmation of commit for their transactions (Section 6.3). This section describes a simple request-response protocol by which a repository may provide this stronger confirmation, by verifying if one or more other repositories have committed the transaction. Implementation of this request-response protocol is optional.

After it has redistributed a transaction, the originating repository may request a commit confirmation from one or more peer repositories by sending to them a "commit-request" meta-object. The "commit-request" contains two attributes:

commit-request This attribute is mandatory and single. It contains the name of the database for whom a commit confirmation is being requested.

sequence This attribute is mandatory and single. It contains the transaction sequence number for which a commit confirmation is being requested.

A repository that receives a "commit-request" must not redistribute the request. It must delay the response until the corresponding transaction has been processed. For this reason, the repository must keep state about pending commit requests. It should discard this state if the connection to the requester is lost before the response is sent. In that event, it is the responsibility of the requester to resend the request.

Once a transaction has been processed (Section 7.3), a repository must check to see if there exists any pending commit request for the transaction. If so, it must send a "commit-response" meta-object to the requester. This meta-object has three attributes:

commit-response This attribute is mandatory and single. It contains the name of the database for whom a commit response is being sent.

sequence This attribute is mandatory and single. It contains the transaction sequence number for which a commit response is being sent.

commit-status This attribute is mandatory and single. It contains one of the keywords "held", "error", or "succeeded". The "error" keyword may be followed by an optional text string. The "held" keyword is returned when a repository containing a dependent object for authorization has expired.

7.5 Database Snapshot

A database snapshot provides a complete copy of a database. It is intended only for repository initialization or disaster recovery. A database snapshot is an out of band mechanism. A set of files are created periodically at the source repository. These files are then transferred to the requestor out of band (e.g. ftp transfer). The objects in these files are then registered locally.

A snapshot of repository `X` contains the following set of files:

`X.db` This file contains the RPSL objects of repository `X`, separated by blank lines. In addition to the RPSL objects and blank lines, comment lines can be present. Comment lines start with the character '#'. The comment lines are ignored. The file `X.db` ends in a special comment line `''# eof''`.

`X.<class>.db` This optional file if present contains the RPSL objects in `X.db` that are of class `<class>`. The format of the file is same as that of `X.db`.

`X.transaction-label` This file contains a `transaction-label` object that records the timestamp and the latest sequence number of the repository at the time of the snapshot.

Each of these files can be optionally compressed uzing gzip. This is signified by appending the suffix `.gz` to the file name. Each of these files can optionally be PGP signed. In this case, the detached signature with ASCII armoring and platform-independent text mode is stored in a file whose name is constructed by appending `.sig` to the file name of the file being signed.

In order to construct a repository's contents from a snapshot, a repository downloads these files. After uncompressing and checking signatures, the repository records these objects in its database. No RPS authorization/authentication is done on these objects. The `transaction-label` object provides the seed for the replication protocol to receive the follow on transactions from this repository. Hence, it is not crucial to download an up to the minute snapshot.

After successfully playing a snapshot, it is possible that a repository may receive a transaction from a third repository that has a dependency on an earlier version of one of the objects in the snapshot. This can only happen within the expire period of the repository being downloaded, plus any possible network partition period. This dependency is only important if the repository wants to re-verify RPS authorization/authentication. There are three allowed alternatives in this case. The simplest alternative is for the repository to accept the transaction and mark it with integrity "no-auth". The second choice is to only peer with trusted repositories during this time period, and accept the transaction with the same integrity as the trusted repository (possibly as "authorized"). The most preferred alternative is not to download an up to the minute snapshot, but to download an older snapshot, at minimum twice the repositories expire time, in practice few days older. Upon replaying an older snapshot, the replication protocol will fetch the more current transactions from this repository. Together they provide the necessary versions of objects to re-verify rps authorization/authentication.

7.6 Authenticating Operations

The "signature" and "repository-signature" meta-objects represent signatures. Where multiple of these objects are present, the signatures should be over the original contents, not over other signatures. This allows signatures to be checked in any order.

A maintainer can also sign a transaction using several authentication methods (some of which may be available in some repositories only).

In the case of PGP, implementations should allow the signatures of the "signature" and "repository-signature" meta-objects to be either the detached signatures produced by PGP or regular signatures produced by PGP. In either case, ASCII armoring and platform-independent text mode should be used.

Note that the RPSL objects themselves are not signed but the entire transaction body is signed. When exchanging transactions among registries, the meta-objects (e.g. "auth-dependency") prior to the first "repository-signature" meta object in the redistributed text are also signed over.

Transactions must remain intact, including the signatures, even if an authentication method provided by the submitter is not used by a repository handling the message. An originating repository may chose to remove clear text passwords signatures from a transaction, and replace it with the keyword "clear-text-passwd" followed by the maintainer's id.

```
signature: clear-text-passwd <maintainer-name>
```

Note that this does not make the system less secure since clear text password is an indication of total trust to the originating repository by the maintainer.

A repository may sign a transaction that it verified. If at any point the signature of a trusted repository is encountered, no further authorization or authentication is needed.

A Examples

RPSL provides an external representation of RPSL objects and attributes. An attribute is a name/value pair. RPSL is line oriented. Line continuation is supported, however most attributes fit on a single line. The attribute name is followed by a colon, then any amount of whitespace, then the attribute value. An example of the ASCII representation of an RPSL attribute is the following:

```
route:     140.222.0.0/16
```

An RPSL object is a set of attributes. Objects are separated from each other by one or more blank lines. An example of a complete RPSL object follows:

```
route:         140.222.0.0/16
```

```
descr:        ANS Communications
origin:       AS1673
member-of:    RS-ANSOSPFAGGREGATE
mnt-by:       ANS
changed:      tck@ans.net 19980115
source:       ANS
```

A.1 Initial Object Submission and Redistribution

Figure 1 outlines the steps involved in submitting an object and the initial redistribution from the authoritative registry to its flooding peers.

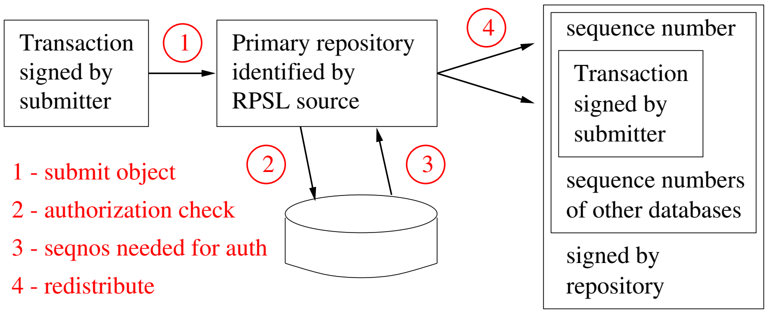

Figure 1: Initial Object Submission and Redistribution

If the authorization check requires objects from other repositories, then the sequence numbers of the local copies of those databases is required for mirrors to recheck the authorization.

To simply resubmit the object from the prior example, the submitter or a client application program acting on the submitter's behalf must submit a transaction. The legacy method was to send PGP signed email. The preferred method is for an interactive program to encapsulate a request between "transaction-submit-begin" and "transaction-submit-end" meta-objects and encapsulate that as a signed block as in the following example:

```
transaction-submit-begin:  ANS 1
response-auth-type:        PGP
transaction-confirm-type:  normal

route:        140.222.0.0/16
descr:        ANS Communications
origin:       AS1673
member-of:    RS-ANSOSPFAGGREGATE
mnt-by:       ANS
changed:      curtis@ans.net 19990401
source:       ANS

timestamp: 19990401 10:30:00 +08:00

signature:
```

```
+ -----BEGIN PGP SIGNATURE-----
+ Version: PGP for Personal Privacy 5.0
+ MessageID: UZi4b7kjlzP7rb72pATPywPxYfQj4gXI
+
+ iQCVAwUANsrwkP/OhQ1cphB9AQFOvwP/Ts8qn3FRRLQQHKmQGzy2IxOTiFOQXB4U
+ Xzb3gEvfeg8NWhAI32zBw/D6FjkEw7P6wDFDeok52A1SA/xdP5wYE8heWQmMJQLX
+ Avf8W49d3CF3qzh59UC0ALtA5BjI3r37ubzTf3mgtw+ONqVJ5+lB5upWbqKN9zqv
+ PGBIEN3/NlM=
+ =c93c
+ -----END PGP SIGNATURE-----

transaction-submit-end:    ANS 1
```

The signature covers the everything after the first blank line after the "transaction-submit-begin" object to the last blank line before the "signature" meta-object. If multiple signatures are needed, it would be quite easy to email this block and ask the other party to add a signature-block and return or submit the transaction. Because of delay in obtaining multiple signatures the accuracy of the "timestamp" cannot be strictly enforced. Enforcing accuracy to within the "expire" time of the database might be a reasonable compromise. The tradeoff is between convenience, allowing a longer time to obtain multiple signatures, and increased time of exposure to replay attack.

The ANS repository would look at its local database and make authorization checks. If the authorization passes, then the sequence number of any other database needed for the authorization is obtained.

If this operation was successful, then a confirmation would be returned. The confirmation would be of the form:

```
transaction-confirm:  ANS 1
confirmed-operation:  change route 140.222.0.0/16 AS1673
commit-status:        commit
timestamp:            19990401 10:30:10 +05:00
```

A.2 Transaction Redistribution Encoding

Having passed the authorization check the transaction is given a sequence number and stored in the local transaction log and is then flooded. The meta-object flooded to another database would be signed by the repository and would be of the following form:

```
transaction-label: ANS
sequence: 6666
timestamp: 19990401 13:30:10 +05:00
integrity: authorized

route:          140.222.0.0/16
descr:          ANS Communications
origin:         AS1673
```

```
member-of:     RS-ANSOSPFAGGREGATE
mnt-by:        ANS
changed:       curtis@ans.net 19990401
source:        ANS

timestamp: 19990401 10:30:00 +08:00

signature:
+ -----BEGIN PGP SIGNATURE-----
+ Version: PGP for Personal Privacy 5.0
+ MessageID: UZi4b7kjlzP7rb72pATPywPxYfQj4gXI
+
+ iQCVAwUANsrwkP/OhQ1cphB9AQFOvwP/Ts8qn3FRRLQQHKmQGzy2IxOTiF0QXB4U
+ Xzb3gEvfeg8NWhAI32zBw/D6FjkEw7P6wDFDeok52A1SA/xdP5wYE8heWQmMJQLX
+ Avf8W49d3CF3qzh59UC0ALtA5BjI3r37ubzTf3mgtw+ONqVJ5+lB5upWbqKN9zqv
+ PGBIEN3/NlM=
+ =c93c
+ -----END PGP SIGNATURE-----

auth-dependency: ARIN
sequence: 555
timestamp: 19990401 13:30:08 +05:00

auth-dependency: RADB
sequence: 4567
timestamp: 19990401 13:27:54 +05:00

repository-signature: ANS
signature:
+ -----BEGIN PGP SIGNATURE-----
+ Version: PGP for Personal Privacy 5.0
+ MessageID: UZi4b7kjlzP7rb72pATPywPxYfQj4gXI
+
+ iQCVAwUANsrwkP/OhQ1cphB9AQFOvwP/Ts8qn3FRRLQQHKmQGzy2IxOTiF0QXB4U
+ Xzb3gEvfeg8NWhAI32zBw/D6FjkEw7P6wDFDeok52A1SA/xdP5wYE8heWQmMJQLX
+ Avf8W49d3CF3qzh59UC0ALtA5BjI3r37ubzTf3mgtw+ONqVJ5+lB5upWbqKN9zqv
+ PGBIEN3/NlM=
+ =c93c
+ -----END PGP SIGNATURE-----
```

Note that the repository-signature above is a detached signature for another file and is illustrative only. The repository-signature covers from the “transaction-label” meta-object (including) to the last blank line before the first “repository-signature” meta-object (excluding the last blank line and the “repository-signature” object).

A.3 Transaction Protocol Encoding

```
transaction-begin: 1276
transfer-method: plain
```

```
transaction-label: ANS
sequence: 6666
timestamp: 19990401 13:30:10 +05:00
integrity: authorized

route:         140.222.0.0/16
descr:         ANS Communications
origin:        AS1673
member-of:     RS-ANSOSPFAGGREGATE
mnt-by:        ANS
changed:       curtis@ans.net 19990401
source:        ANS

timestamp: 19990401 10:30:00 +08:00

signature:
+ -----BEGIN PGP SIGNATURE-----
+ Version: PGP for Personal Privacy 5.0
+ MessageID: UZi4b7kjlzP7rb72pATPywPxYfQj4gXI
+
+ iQCVAwUANsrwkP/OhQ1cphB9AQFOvwP/Ts8qn3FRRLQQHKmQGzy2IxOTiF0QXB4U
+ Xzb3gEvfeg8NWhAI32zBw/D6FjkEw7P6wDFDeok52A1SA/xdP5wYE8heWQmMJQLX
+ Avf8W49d3CF3qzh59UC0ALtA5BjI3r37ubzTf3mgtw+ONqVJ5+lB5upWbqKN9zqv
+ PGBIEN3/NlM=
+ =c93c
+ -----END PGP SIGNATURE-----

auth-dependency: ARIN
sequence: 555
timestamp: 19990401 13:30:08 +05:00

auth-dependency: RADB
sequence: 4567
timestamp: 19990401 13:27:54 +05:00

repository-signature: ANS
signature:
+ -----BEGIN PGP SIGNATURE-----
+ Version: PGP for Personal Privacy 5.0
+ MessageID: UZi4b7kjlzP7rb72pATPywPxYfQj4gXI
+
+ iQCVAwUANsrwkP/OhQ1cphB9AQFOvwP/Ts8qn3FRRLQQHKmQGzy2IxOTiF0QXB4U
+ Xzb3gEvfeg8NWhAI32zBw/D6FjkEw7P6wDFDeok52A1SA/xdP5wYE8heWQmMJQLX
+ Avf8W49d3CF3qzh59UC0ALtA5BjI3r37ubzTf3mgtw+ONqVJ5+lB5upWbqKN9zqv
+ PGBIEN3/NlM=
+ =c93c
+ -----END PGP SIGNATURE-----
```

Before the transaction is sent to a peer, the repository prepends a "transaction-begin" meta-object. The value of the "transaction-begin" attribute is the number of octets in the transaction, not counting the "transaction-begin" meta-object and the first blank line after it.

Separating transaction-begin and transaction-label objects enables different encodings at different flooding peerings.

A.4 Transaction Redistribution

The last step in Figure 1 was redistributing the submitter's transaction through flooding (or later through polling). Figure 2 illustrates the further redistribution of the transaction.

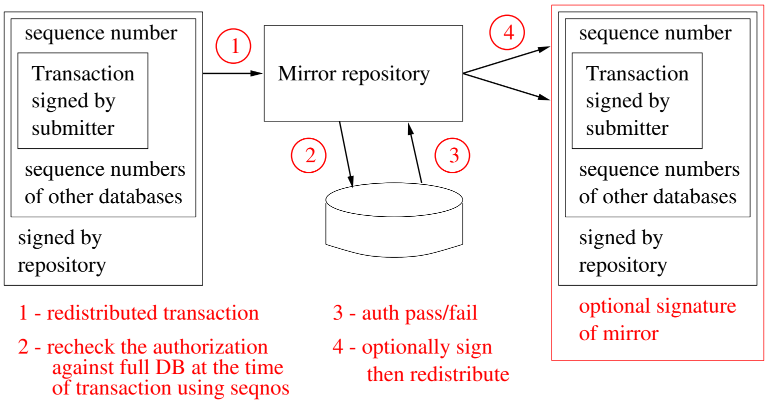

Figure 2: Further Transaction Redistribution

If the authorization check was repeated, the mirror may optionally add a repository-signature before passing the transaction any further. A "signature" can be added within that block. The previous signatures should not be signed.

Figure 3 illustrates the special case referred to as a "lightweight mirror". This is specifically intended for routers.

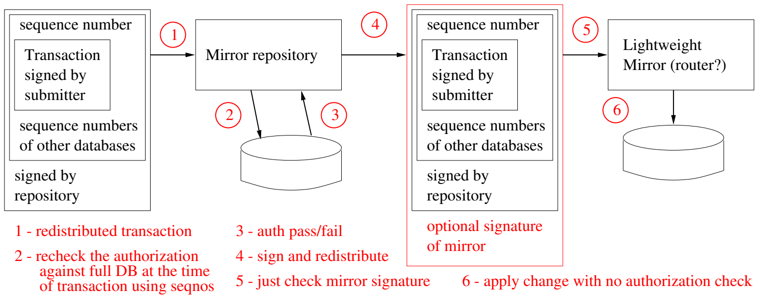

Figure 3: Redistribution to Lightweight Mirrors

The lightweight mirror must trust the mirror from which it gets a feed. This is a safe assumption if the two are under the same administration (the mirror providing the feed is a host owned by the same ISP who owns the routers). The lightweight mirror simply checks the signature of the adjacent repository to insure data integrity.

B Technical Discussion

B.1 Server Processing

This document does not mandate any particular software design, programming language choice, or underlying database or underlying operating system. Examples are given solely for illustrative purposes.

B.1.1 getting connected

There are two primary methods of communicating with a repository server. E-mail can be sent to the server. This method may be deprecated but at least needs to be supported during transition. The second method is preferred, connect directly to a TCP socket.

Traditionally the whois service is supported for simple queries. It might be wise to retain the whois port connection solely for simple queries and use a second port not in the reserved number space for all other operations including queries except those queries using the whois unstructured single line query format.

There are two styles of handling connection initiation is the dedicated daemon, in the style of BSD sendmail, or launching through a general purpose daemon such as BSD inetd. E-mail is normally handled sequentially and can be handled by a front end program which will make the connection to a socket in the process as acting as a mail delivery agent.

B.1.2 rolling transaction logs forward and back

There is a need to be able to easily look back at previous states of any database in order to repeat authorization checks at the time of a transaction. This is difficult to do with the RIPE database implementation, which uses a sequentially written ASCII file and a set of Berkeley DB maintained index files for traversal. At the very minimum, the way in which deletes or replacements are implemented would need to be altered.

In order to easily support a view back at prior versions of objects, the sequence number of the transaction at which each object was entered would need to be kept with the object. A pointer would be needed back to the previous state of the object. A deletion would need to be implemented as a new object with a deleted attribute, replacing the previous version of the object but retaining a pointer back to it.

A separate transaction log needs to be maintained. Beyond some age, the older versions of objects and the the older transaction log entries can be removed although it is probably wise to archive them.

B.1.3 committing or disposing of transactions

The ability to commit large transaction, or reject them as a whole poses problems for simplistic database designs. This form of commit operation can be supported quite easily using memory mapped files. The changes can be made in virtual memory only and then either committed or disposed of.

B.1.4 dealing with concurrency

Multiple connections may be active. In addition, a single connection may have multiple outstanding operations. It makes sense to have a single process or thread coordinate the responses for a given connection and have multiple processes or threads each tending to a single operation. The operations may complete in random order.

Locking on reads is not essential. Locking before write access is essential. The simplest approach to locking is to lock at the database granularity or at the database and object type granularity. Finer locking granularity can also be implemented. Because there are multiple databases, deadlock avoidance must be considered. The usual deadlock avoidance mechanism is to acquire all necessary locks in a single operation or acquire locks in a prescribed order.

B.2 Repository Mirroring for Redundancy

There are numerous reasons why the operator of a repository might mirror their own repository. Possibly the most obvious are redundancy and the relative ease of disaster recovery. Another reason might be the widespread use of a small number of implementations (but more than one) and the desire to insure that the major repository software releases will accept a transaction before fully committing to the transaction.

The operation of a repository mirror used for redundancy is quite straightforward. The transactions of the primary repository host can be immediately fed to the redundant repository host. For tighter assurances that false positive confirmations will be sent, as a matter of policy the primary repository host can require commit confirmation before making a transaction sequence publicly available.

There are many ways in which the integrity of local data can be assured regardless of a local crash in the midst of transaction disk writes. For example, transactions can be implemented as memory mapped file operations, with disk synchronization used as the local commit mechanism, and disposal of memory copies of pages used to handle commit failures. The old pages can be written to a separate file, the new pages written into the database. The transaction can be logged and old pages file can then be removed. In the event of a crash, the existence of a old pages file and the lack of a record of the transaction completing would trigger a transaction roll back by writing the old pages back to the database file.

The primary repository host can still sustain severe damage such as a disk crash. If the primary repository host becomes corrupted, the use of a mirror repository host provides a backup and can provide a rapid recovery from disaster by simply reversing roles.

If a mirror is set up using a different software implementation with commit mirror confirmation required, any transaction which fails due a software bug will be deferred indefinitely allowing other transactions to proceed rather than halting the remote processing of all transactions until the bug is fixed everywhere.

B.3 Trust Relationships

If all repositories trust each other then there is never a need to repeat authorization checks. This enables a convenient interim step for deployment prior to the completion of software supporting that capability. The opposite case is where no repository trusts any other repository. In this case, all repositories must roll forward transactions gradually, checking the authorization of each remote transaction.

It is likely that repositories will trust a subset of other repositories. This trust can reduce the amount of processing a repository required to maintain mirror images of the full set of data. For example, a subset of repositories might be trustworthy in that they take reasonable security measures, the organizations themselves have the integrity not to alter data, and these repositories trust only a limited set of similar repositories. If any one of these repositories receives a transaction sequence and repeats the authorization checks, other major repositories which trusts that repository need not repeat the checks. In addition, trust need not be mutual to reap some benefit in reduced processing.

As a transaction sequence is passed from repository to repository each repository signs the transaction sequence before forwarding it. If a receiving repository finds that any trusted repository has signed the transaction sequence it can be considered authorized since the trusted repository either trusted a preceding repository or repeated the authorization checks.

B.4 A Router as a Minimal Mirror

A router could serve as a minimal repository mirror. The following simplifications can be made.

1. No support for repeating authorization checks or transaction authentication checks need be coded in the router.
2. The router must be adjacent only to trusted mirrors, generally operated by the same organization.
3. The router would only check the authentication of the adjacent repository mirrors.
4. No support for transaction submission or query need be coded in the router. No commit support is needed.
5. The router can dispose of any object types or attributes not needed for configuration of route filters.

The need to update router configurations could be significantly reduced if the router were capable of acting as a limited repository mirror.

A significant amount of non-volatile storage would be needed. There are currently an estimated 100 transactions per day. If storage were flash memory with a limited number of writes, or if there were some other reason to avoid writing to flash, the router could only update the non-volatile copy every few days. A transaction sequence request can be made to get an update in the event of a crash, returning only a few hundred updates after losing a few days of deferred writes. The routers can still take a frequent or continuous feed of transactions.

Alternately, router filters can be reconfigured periodically as they are today.

B.5 Dealing with Errors

If verification of an authorization check fails, the entire transaction must be rejected and no further advancement of the repository can occur until the originating repository corrects the problem. If the problem is due to a software bug, the offending transaction can be removed manually once the problem is corrected. If a software bug exists in the receiving software, then the transaction sequence is stalled until the bug is corrected. It is better for software to error on the side of denying a transaction than acceptance, since an error on the side of acceptance will require later removal of the effects of the transaction.

C Deployment Considerations

This section described deployment considerations. The intention is to raise issues rather than to provide a deployment plan.

This document calls for a transaction exchange mechanism similar to but not identical to the existing “near real time mirroring” supported by the code base widely used by the routing registries. As an initial step, the transaction exchange can be implemented without the commit protocol or the ability to recheck transaction authorization. This is a fairly minimal step from the existing capabilities.

The transition can be staged as follows:

1. Modify the format of “near real time mirroring” transaction exchange to conform to the specifications of this document.
2. Implement commit protocol and confirmation support.
3. Implement remote recheck of authorization. Prior to this step all repositories must be trusted.
4. Allow further decentralization of the repositories.

D Privacy of Contact Information

The routing registries have contained contact information. The redistribution of this contact information has been a delicate issue and in some countries has legal implications.

The person and role objects contain contact information. These objects are referenced by NIC-handles. There are some attributes such as the "changed" and "notify" attributes that require an email address. All of the fields that currently require an email address must also accept a NIC-handle.

The person and role objects should not be redistributed by default. If a submission contains an email address in a field such as a changed field rather than a NIC-handle the submitter should be aware that they are allowing that email address to be redistributed and forfeiting any privacy. Repositories which do not feel that prior warnings of this forfeiture are sufficient legal protection should reject the submission requesting that a NIC-handle be used.

Queries to role and person objects arriving at a mirror must be referred to the authoritative repository where whatever authentication, restrictions, or limitations deemed appropriate by that repository can be enforced directly.

Software should make it possible to restrict the redistribution of other entire object types as long as those object types are not required for the authorization of additions of other object types. It is not possible to redistribute objects with attributes removed or altered since this would invalidate the submitter's signature and make subsequent authentication checks impossible. Repositories should not redistribute a subset of the objects of a given type.

Software should also not let a transaction contain both redistributable (e.g. policy objects) and non-redustributable objects (e.g. person) since there is no way to verify the signature of these transactions without the non-redustributable objects.

When redistributing legacy data, contact information in attributes such as "changed" and "notify" should be stripped to maintain privacy. The "integrity" attribute on these objects should already be set to "legacy" indicating that their origin is questionable, so the issue of not being able to recheck signatures is not as significant.

References

[1] C. Alaettinoglu, C. Villamizar, E. Gerich, D. Kessens, D. Meyer, T. Bates, D. Karrenberg, and M. Terpstra. Routing Policy Specification Language (RPSL. Technical Report RFC 2622, Internet Engineering Task Force, 1999.

[2] T. Bates, E. Gerich, L. Joncheray, J-M. Jouanigot, D. Karrenberg, M. Terpstra, and J. Yu. Representation of IP Routing Policies in a Routing Registry (ripe-81++). Technical Report RFC 1786, Internet Engineering Task Force, 1995.

[3] C. Villamizar, C. Alaettinoglu, D. Meyer, and S. Murphy. Routing Policy System Security. Technical Report RFC 2725, Internet Engineering Task Force, 1999.

[4] J. Zsako. PGP Authentication for RIPE Database Updates. Technical Report RFC 2726, Internet Engineering Task Force, 1999.

Security Considerations

An authentication and authorization model for routing policy object submission is provided by [3]. Cryptographic authentication is addressed by [4]. This document provides a protocol for the exchange of information among distributed routing registries such that the authorization model provided by [3] can be adhered to by all registries and any deviation (hopefully accidental) from those rules on the part of a registry can be identified by other registries or mirrors.

Author's Addresses


```
Curtis Villamizar                  Cengiz Alaettinoglu
Avici Systems                      ISI
<curtis@avici.com>                 <cengiz@ISI.EDU>


Ramesh Govindan                    David M. Meyer
ISI                                Cisco
<govindan@ISI.EDU>                 <dmm@cisco.com>
```


Full Copyright Statement

Notices