Internet Engineering Task Force
INTERNET-DRAFT
draft-ietf-rps-dist-00

Curtis Villamizar
ANS
Cengiz Alaettinoglu
ISI
Ramesh Govindan
ISI
David M. Meyer
University of Oregon
September 29, 1998

Distributed Routing Policy System

Status of this Memo

Abstract

The RIPE database specifications [2] and RPSL language [1] 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 on importance to the operation of the Internet. The implementation and deployment of a routing policy system must maintain some degree of integrety to be of any use. The Routing Policy System Security internet-draft [3] addresses the need to assure integrety 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 multiple repositories without compromising the authorization model extablished in [3].

1 Overview

A routing registry must maintain some degree of integrety to be of any use. The IRR is increasingly used for purposes that have a stronger requirement for data integrety 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 integrety 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 itegrety 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 integrety 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 distinction is made between a repository and a mirror. A repository has responsibility for the initial authorization and authentication checks for transsactions related to its local objects which are then flooded to adjacent repositories (either by unicast flooding or by multicast in subsets of the topology of repository adjacencies). A mirror receives flooded transactions from remote repositories but contains no local objects. From a protocl 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 recieved thorugh flooding, which is a very low volume. Only the signature of the adjacent mirror or repocitory 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 an routing registry.

Some form of encapsulation must be used to exchange data. The defacto 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 seqence 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 5.1. An encapsulation suitable for exchange of transaction between repositories is addressed in Section 5. Query encapsulation and protocol is outside the scope of this document.

3 Athentication 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" or "mnt-routes" 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 integrety 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 Hierachy

With multiple repositories, "repository" objects are needed to propogate the existance 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 [3].

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 a "refresh" and an "expire" attribute. The refresh attribute may be used if transactions are exchanged in a polling mode. Flooding is preferred. The expire attribute is used to determine if a repository must be updated before a local transaction can 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 queries.

5 Interactions with a Repository or Mirror

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 5.1.

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. Transactions can also be recinded. This is discussed in Section 5.2.

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

Recinding Transactions Although it is hoped that the feature is never needed, it may be necessary to recind transactions (Section 5.3).

Transaction Commit and Confirmation: Repositories may optionally implement a commit protocol and a completion indication that gives the submitter of a transaction an 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 5.4.

5.1 Initial Transaction Submission

The simplest form of transaction submission is an object or set of objects submitted with RFC–822 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 6.

5.2 Redistribution of Transactions

Redistribution of transactions can be accomplished using unicast or optionally using multicast capabilities. There are three types of requests for redistribution of transactions.

1. A repository snapshots 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.
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 requests may be of two types. One type is a request for a direct unicast feed. The other type of request is for a multicast group on which a particular repository is flooded using reliable multicast.

This section describes the operations somewhat qualitatively. Data formats and state diagrams are provided in Section 6.

5.3 Recinding Transactions

Recinding a transaction is a manual intervention. The administrators of a repository may find it necessary to request that a specific set of transactions be removed. Database mirrors would have to roll back the entire database to the first transaction being recinded and then roll forward the transaction log from that point forward. Authorizations in other repositories may be affected.

There are many reasons for having to recind a transaction whose cause is outside the control of the operator of the repository. For example, a disgruntled employee at a client of the repository may remove all authorization from that clients database objects. There may be opportunities for malicious entries in objects for which there is no authorization hierarchy (See [3]). An example is the anonymous registration of falsified person objects or liablous or obescene person or role objects. In addition, mistakes or program bugs are inevitable.

5.4 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.

6 Data Format Summaries

@@ This draft is in an early stage. This actual formats are likely to change. @@

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 whitespace is equivalent to a single space. Line continuation is supported by a backslash at the end of a line or the following line beginning with whitespace. When transferred externally attributes are generally broken into shorter lines using line continuation though this is not a requirement. An object is externally represented as a series of attributes. Objects are separated by blank lines.

There are about 80 attribute types in the current RIPE schema and about 15 object types. Some of the attributes are manditory in certain objects. Some attributes may appear multiple times. One or more attributes may for a key. Some attributes or sets of attributes may be required to be unique. Some of the attributes may reference a key field in an object type and may be required to be a valid reference. Some attributes may be used in inverse lookups.

A review of the entire RIPE or RPSL schema would be too lengthy to include here. Only the differences in the schema are described.

Interactions with the registry either use a legacy format or are encapsulated using sets of name and value pairs that are formated like RPSL objects. These are not part of RPSL and are referred to as meta-objects. The meta-objects serve mostly as delimiters to the transactions and to carry information about the type of operation.

6.1 Transaction Submit and Confirm

The defacto method for submitting database changes has been via email. This method should be supported by an external application for backwards compatibility only. Merit has added the pgp-from authentication method to the RADB (replaced by pgp-key in [3]), where the mail headers are essentially ignored and the body of the mail message must be PGP signed. For backwards compatibility objects submitted in an email message, even if signed as a group, should be treated as separate transactions as they are today. RFC–822 encapsulated messages should default to a confirmation of type “LEGACY”.

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. Use of the email interface is discouraged and the email interface will eventually be depricated. 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 manditory 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 See Section 6.6.

date-time-stamp See Section 6.6.

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 contains 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 manditory 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 See Section 6.6.

date-time-stamp See Section 6.6.

6.2 Transaction Commit

The commit protocol consists of two steps.

1. commit request
2. commit completion

The "commit request" consists of a set of delimiters around a single transaction that has yet to be committed. The delimiters are the "mirror-request-begin" meta-object and "mirror-request-end" meta-object. The "mirror-request-begin" meta-object may contain the following attributes.

mirror-request-begin This attribute is manditory and single. It contains the database name and sequence number of the transaction about to be committed.

date-time-stamp See Section 6.6.

The "mirror-request-end" meta-object consists of a single attribute of the same name containing the same database name and sequence number provided by the corresponding "mirror-request-begin".

The "commit-completion" meta-object is sent in response to a "commit request". Prior to attempting completion the remote database may have some catching up to do to reach the requested sequence number. If so, the remote database will send a "transaction-request" (Section 6.4) to bring its database copy to the sequence number below the transaction being commited.

The "commit-completion" meta-object may contain the following attributes.

commit-completion This attribute is manditory and single. It contains the same name sequence number provided by the corresponding "mirror-request-begin".

date-time-stamp See Section 6.6.

commit-status See Section 6.6.

6.3 Database Snapshot

A database snapshot provides a complete copy of a database. It is intended only for repository initialization and and disaster recovery.

A database snapshot request is represented by a "snapshot-request" meta-object. The "snapshot-request" meta-object may contain the the following attributes.

snapshot-request This attribute is manditory and single. It contains the database name of the database being requested.

response-auth-type See Section 6.6.

A database snapshot is returned. The database snapshot is delimited by a "shapshot-begin" and "shapshot-end" meta-object. The "shapshot-begin" meta-object may contain the following attributes.

shapshot-begin This attribute is manditory and single. It contains the database name and sequence number of the database snapshot being returned.

date-time-stamp See Section 6.6.

The "shapshot-end" meta-object contains a single attribute by the same name containing the same database name and sequence number provided in the corresponding "shapshot-begin".

6.4 Redistribution of Transactions

There are three ways to track database changes. One method is to join a multicast group where a repository multicasts changes for a specific database (see Section 6.4.3). The multicasting repository is not necessarily the authoritative repository for the database). Another method is to make a unicast connection and request unicast mirroring for a specific repository (see Section 6.4.2). A third method is to poll by requesting a transaction sequence (see Section 6.4.1). To get updated to the current state of the database a request can be made with the end sequence number set to the special value "last".

6.4.1 Polling for Specific Transaction Sequences

A transaction sequence can be requested by sending a "transaction-request" meta-object. A "transaction-request" meta-object may contain the following attributes.

transaction-request This attribute is manditory and single. It contains the database name and a sequence list. The sequence list is two sequence numbers separated by a dash. The keyword "last" may be used in place of a number to indicate the last sequence number available. The sequence list "last-last" can be requested to simply get the last sequence number in an empty transaction sequence.

response-auth-type See Section 6.6.

6.4.2 Unicast Flooding Redistribution

A unicast mirror request is represented by a "unicast-mirror-request" meta-object which may contain the following attributes.

unicast-mirror-request This attribute is manditory and single. It contains the database and next sequence number needed. This may optionally followed by an maximum update frequency in seconds, a heartbeat rate in minutes, and an idle timeout in minutes. If there are no new transactions during the heartbeat period an empty transaction sequence is sent. If there are no new transactions and no other activity on the socket before the idle period the connection is dropped.

response-auth-type See Section 6.6.

A unicast mirror request is answered by a unicast mirror response. This is represented in a "unicast-mirror-response" meta-object, which may contain one of the following attributes.

unicast-mirror-response This attribute is manditory and single. It contains the name of the database.

unicast-mirror-status This attribute is optional and single. It may contain the word "rejected".

unicast-referal This attribute is optional and multiple. It contains the name of a that are known or likely to provide a unicast feed of the requested database.

A repository may reject a request for a unicast feed for a variety of reasons. Offering an alternative place to look may be helpful to the requestor. The alternative could be adjacent repositories providing a feed.

A unicast feed may be cancelled without disrupting other use of the socket. See Section 6.4.6.

6.4.3 Multicast Redistribution

A multicast mirror request is represented by a "multicast-mirror-request" meta-object which may contain the following attributes.

multicast-mirror-request This attribute is manditory and single. It contains the database name to be mirrored.

response-auth-type See Section 6.6.

A multicast mirror request is answered with a multicast mirror response. This is represented in a "multicast-mirror-response" meta-object, which may contain one of the following attributes.

multicast-mirror-response This attribute is manditory and single. It contains the name of the database to be mirrored.

multicast-mirror This attribute is optional and multiple. It cotains a multicast group and the authentication type arguments of the authentication methods used when multicasting.

multicast-referal This attribute is optional and multiple. It contains the name of a repository that is known or likely to multicast the requested database.

A "multicast-mirror-response" will be returned even if the database is not being multicast or the requested authentication type is not being used. The response may contain a zero or more "multicast-mirror" attributes and zero or more "multicast-referal" attributes. An empty response means the repository is not multicasting the database requested and does not know of any other repository which is doing so.

If an object intended to be multicast is too big to fit inside a single packet, it may be necessary to send the object as a multipart-compressed object (Section 6.4.5).

A multicast feed can be cancelled simply by leaving the multicast group.

If listenning on a multicast group, loss is detected by the receipt of sequences beyond the current sequence number. When loss is detected, a unicast connection can be used to request the missing transaction sequence. [@@ alt method to be provided by ISI?]

6.4.4 Transaction Sequence Format

A transaction sequence may contain one or more transactions. Whether obtaining transactions through unicast or multicast, transactions are encapsulated as transaction sequences.

A transaction sequence is delimited by a "sequence-begin" and "sequence-end" meta-object. If the sequence is sent via multicast and requires multiple packets, reliable, in-order delivery is not assured as it is for TCP. To overcome this, each subsequent packets in a transaction sequence not sent using TCP must begin with a "sequence-continue". The complete transaction sequence should be treated as an atomic operation.

The following attributes may be contained in a "sequence-begin" meta-object.

sequence-begin This attribute is manditory and single. It contains the database name and the next available sequence number. This sequence number will be used for the first transaction in the sequence if the sequence in not empty.

database-sequence This attribute is optional and multiple. It contains the database name and sequence number of a database that is needed for authorization of one or more transaction in the sequence.

date-time-stamp See Section 6.6.

The "sequence-continue" and "sequence-end" meta-objects contain an attribute by the same name. A "sequence-continue" attribute contains the the database name and the next available sequence number followed by a fragment number. The first fragment is numbered one (the sequence-begin can be considered fragment zero). The fragment number can be followed by the word "append" which means that the current fragment should be appended to the last object in the prior fragment. A "sequence-end" attribute contains the the database name and the next available sequence number. The next available sequence number must be followed by the number of fragments if any "sequence-continue" fragments were sent. These meta-objects may also contain a "date-time-stamp" attribute.

Transactions are encapsulated by embedding the initial transaction submission intact including any authentication.

Transactions can also be recinded. The operation of recinding a transaction is represented by a "transaction-recind" meta-object which itself consumes one sequence number. The "transaction-recind" meta-object contains one attribute by the same name. The value of the attribute is the sequence number of the first transaction being recinded and the sequence number following the last transaction recinded.

6.4.5 Compressed and Multipart Objects

It may be necessary or advantageous to compress, ascii encode, and sometimes split a message into multiple parts. To accomplish this, a "transfer encoding" is used. The transfer encoding consists of one or more "transfer-encoding" meta-objects which may contain the following attribues:

transfer-encoding This attribute is manditory and single. It contains only an identifier.

transfer-part This attribute is optional and single. It contains a part number starting at one and the total number of parts.

transfer-method This attribute is manditory and single. It contains one or more of the following keywords "gzip", "uuencode", "base50", "radix64".

transfer-contents This attribute is manditory and single. The contents being transferred is contained in this attribute. A single leading space can be used for line continuation.

6.4.6 Cancelling Operations

A request can be made to cancel most operations. The most common would be to cancel a “query” which is returning too much information or cancel a long running operation like a “unicast-mirror-request”.

A “cancel-operation” meta object contains only an attribute by the same name. The attribute contains the operation type represented by the key attribute name in the request without the trailing “-begin”. The remainder of the “cancel-operation” attribute contains the key field of the request.

When an operation is cancelled a “cancel-confirm” meta-object is returned. Any response in progress is ended by the “cancel-confirm” and a “-end” meta-object should not be expected. The “cancel-confirm” attribute contains the same operation type and key field as the corresponding “cancel-operation”.

6.5 Authenticating Operations

PGP normally encapsulates text by starting with a line containing “—–BEGIN PGP SIGNED MESSAGE—–” and a blank line and then ending with the signature block. The signature block consists of a blank line, then a line with “—–BEGIN PGP SIGNATURE—–”, then a block containing the ASCII radix-64 signature and ending with a line containing “—–END PGP SIGNATURE—–”. This encapsulation can be recognized as a meta-objects allowing pgp to be used in normal pipe plumbing using the PGPPASSFD feature to provide a pass phrase.

Alternately, the the PGP delimiters can be replaced with meta-objects and then restored to the format compatible with the PGP code. This is preferable. The meta-objects “signed-object-begin” and “signed-object-end” can be used. The attributes “signed-object-begin” and “signed-object-end” contain only the authentication method name. The interpretation of any additional attributes depend on the authentication method.

Any objects can be signed, including large sequences of meta-objects and objects such as transaction sequences. Objects may be signed by more than one method. If more than one method is used to sign an object, then either method can be used to authenticate the object, ignore one of them.

Note that the RPSL objects themselves are not signed. What is signed by the submitter is the transaction. When exchanging transactions among registries, the objects that make up requests are signed by one registry and the transaction sequences returned are signed by the other registry. Within the transaction sequences there may be signed transactions. There is additional meta-information within the transaction sequences that falls outside of the submitter’s signature.

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. It is also useful to retain the transaction sequence signatures and add an addition signature when encapsulating a received transaction sequence.

Normally repositories will sign transactions between repositories. When unwrapping the authentication encapsulations, the identities of the signatures must be retained to establish authorization. If at any point the signature of a trusted repository is encountered, no further authorization or authentication is needed and any further nested “signed-object-begin” and “signed-object-end” can be ignored.

6.6 Attributes Common to Meta-Objects

A number of attributes are used by numerous meta-objects. They are described here rather than repeating their descriptions elsewhere.

date-time-stamp This attribute is manditory and single except were it is noted as being optional. The date and time are given in the form “YYYYMMDD HHMMSS” with an optional numeric timezone represented as “[+-]H”. The upper case letters are digits corresponding to the year, month, day of month, hour, minute, second, and hours before or after UTC.

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.

commit-status This attribute is manditory and single. It contains one of the keywords “timeout”, “error”, or “commit”. The “error” keyword must be followed by a numeric code and an optional text string.

@@ list of error codes ... yech

A Technical Discussion

A.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.

A.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 depricated but at least needs to be supported during transition. The second method is preferred, connect directly to a TCP socket. Multicast is a third method, but this is limited to use in data replication between repositories.

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.

A.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.

A.1.3 commiting 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.

A.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.

A.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 commiting to the transaction. This may avoid the need to recind transactions in the face of a newly discovered bug.

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 integrety 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 or the offending transaction is recinded.

A.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 integrety 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 preceeding repository or repeated the authorization checks.

This reduction in processing made possible by redistributing the transaction sequences at the application level favors flooding or query mechanisms of distribution rather than IP multicast redictribution. It might make sense to incorporate both application leve flooding and multicast. A repository can redistribute to a limited multicast group taking into account both network topology and the trust relationships.

A.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-volitile 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-volitile 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.

A.5 Dealing with Errors

If verification of an authorization check fails, the entire sequence 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 recinding transactions and rolling forward only those that were valid.

B 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.

Whether to use unicast or multicast as a means of distributing transactions is somewhat orthogonal to this deployment. Currently transactions are distributed on a query basis or a unicast connection basis. Where many repositories receive the same transaction information it may make sense to distribute transactions via multicast. A query method will need to be supported for the purpose of obtaining transaction sequences lost when multicasting and in the short term to accomodate discontinuity in the multicast topology or inadequate performance of deployed multicast service.

Acknowledgements

@@ Will fill in later. @@

References

[1] C. Alaettinoglu, T. Bates, E. Gerich, D. Karrenberg, D. Meyer, M. Terpstra, and C. Villamizar. Routing policy specification language (rpsl). Technical Report RFC 2280, Internet Engineering Task Force, 1998. ftp://ds.internic.net/rfc/rfc2280.txt.

[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. ftp://ds.internic.net/rfc/rfc1786.txt.

[3] David Meyer, C. Villamizar, Cengiz Alaettinoglu, S. Murphy, and Carol Orange. Routing policy system security. Internet Draft (Work in Progress) draft-ietf-rps-auth-01, Internet Engineering Task Force, 5 1998. ftp://ds.internic.net/internet-drafts/draft-ietf-rps-auth-01.txt.

Security Considerations

@@ later for this too.

Author's Addresses


```
Curtis Villamizar       Cengiz Alaettinoglu
ANS Communications      ISI
<curtis@ans.net>        <cengiz@ISI.EDU>


Ramesh Govindan         David M. Meyer
ISI                     University of Oregon
<govindan@ISI.EDU>      <meyer@uoregon.edu>
```