www.sipspectrum.com

repro 1.8 Overview

Scott Godin

Version 1.5 – Sept 17, 2012

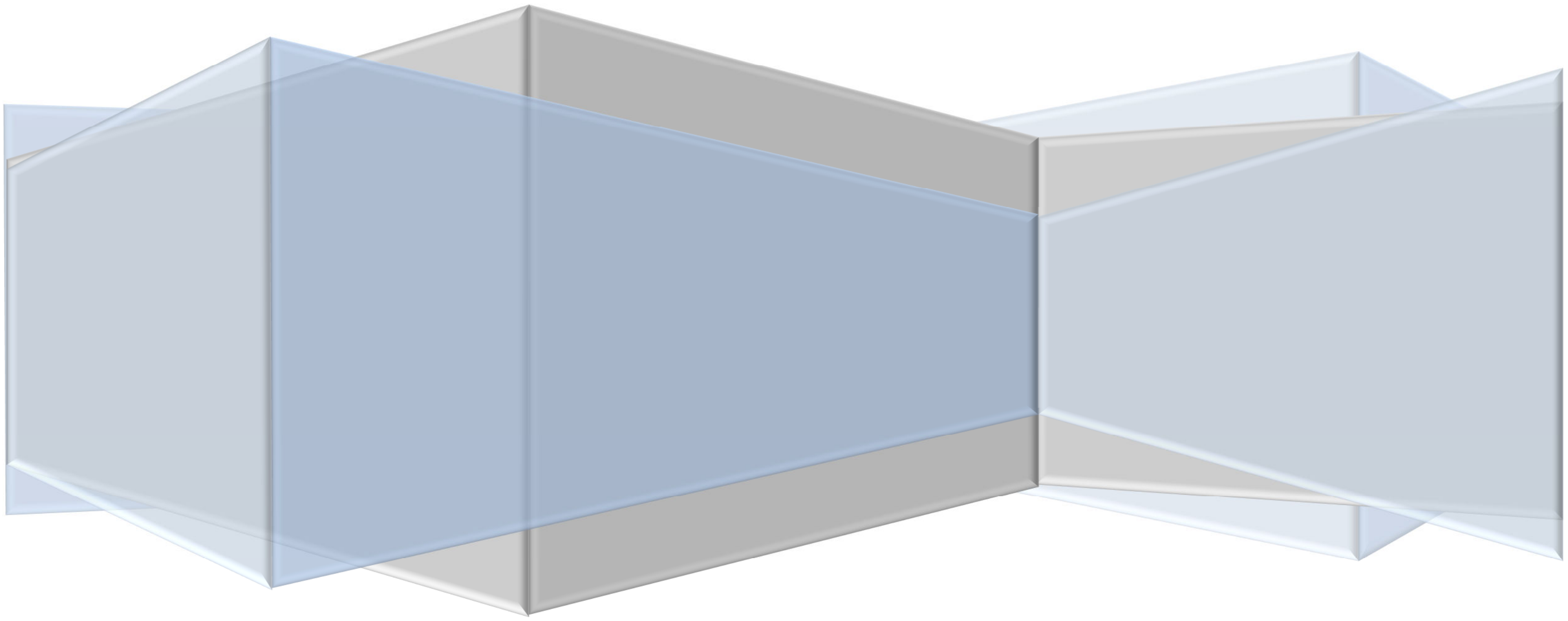


Table of Contents

Introduction

What is repro?

repro is an open-source, free SIP server. SIP is changing the way people communicate using the Internet. It is not only about making phone calls over the Net. The SIP protocol and its extensions defines the way of establishing, modifying and ending interactive sessions, no matter if they are voice, video, IM or a combination of them. At the heart of SIP architecture, there are certain services which need to be provided at some place in the network. repro provides SIP proxy, registrar, redirect, and identity services. These services are the foundation needed to run a SIP service.

Where can repro be used?

- As the central rendezvous service for peer-to-peer voice, IM, and presence services
- As the core of large scale internet telephony services
- As a tool to enforce policy at the boundary between networks or domains
- As an edge server, providing Outbound (RFC5626) or SIP protocol conversion services

What makes repro unique and valuable?

- closely tracks the evolving standardization efforts
- focuses on large-scale operation, including high availability
- pays careful attention to both provider and subscriber security issues

https://www.resiprocate.org/About_Repro

Repro Configuration Overview

Repro is currently entirely configured via a combination of a web page (HTTP) settings GUI and a configuration file with command line argument overrides. Some of the settings relate to SIP transports and other global SIP processing settings and some of the settings are specific to particular Processors in the various chains of responsibility (ie. Monkeys, Baboons and Lemurs). The settings that appear on the web interface are changeable at runtime and are stored in a BerkeleyDb or MySQL database , however some settings that are configuration file / command line only, require a restart to apply them.

Web Interface Settings – Global

Domain List

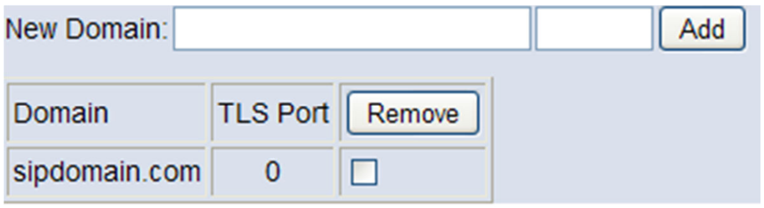
A list of domains for which this proxy authorative. You must add at least one "domain" before you can use the proxy server. The domains are names that the proxy recognizes after the at-sign (@) in the SIP URI. The list of domains is used by the proxy and the registrar to decide if repro is responsible for SIP requests it receives. The WebAdmin also uses the list of domains to make sure users you add are in one of the domains.

For example, if you want the proxy to answer requests for atlanta.com, you need to add atlanta.com to the list of domains. You still need to make sure that you configure DNS so that SIP requests for that domain resolve to repro.

If you don't have a fully qualified domain name, you can use your IP address as a "domain".

NOTE: TLS Port is currently not used.

WARNING: repro must be restarted after adding domains here.

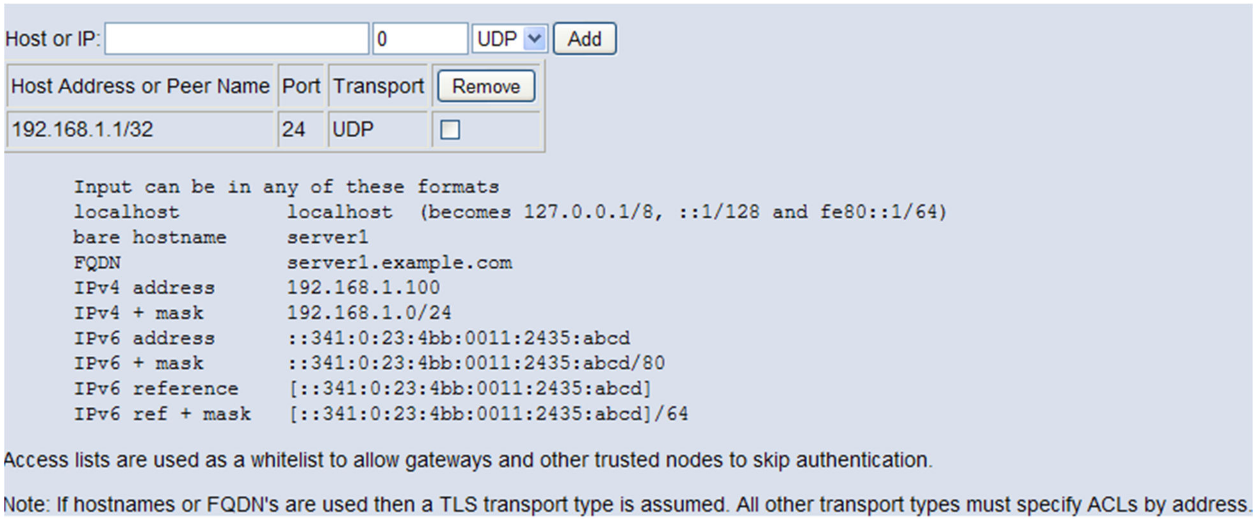

New Domain:

Domain	TLS Port	Remove
sipdomain.com	0	☐

Web Interface Settings – Processor Specific

ACLS / Is Trusted Node Monkey Settings

Access Control List – a list of Hostname/IP, and/or port, and transport type of sender that you would like to treat as trusted. Other Monkeys can use the IsTrusted flag to determine behaviour. For example the Digest Authenticator does not digest challenge trusted senders



Host Address or Peer Name	Port	Transport	Remove
192.168.1.1/32	24	UDP	☐

```
Input can be in any of these formats
localhost          localhost  (becomes 127.0.0.1/8, ::1/128 and fe80::1/64)
bare hostname      server1
FQDN               server1.example.com
IPv4 address       192.168.1.100
IPv4 + mask        192.168.1.0/24
IPv6 address       ::341:0:23:4bb:0011:2435:abcd
IPv6 + mask        ::341:0:23:4bb:0011:2435:abcd/80
IPv6 reference     [::341:0:23:4bb:0011:2435:abcd]
IPv6 ref + mask    [::341:0:23:4bb:0011:2435:abcd]/64
```

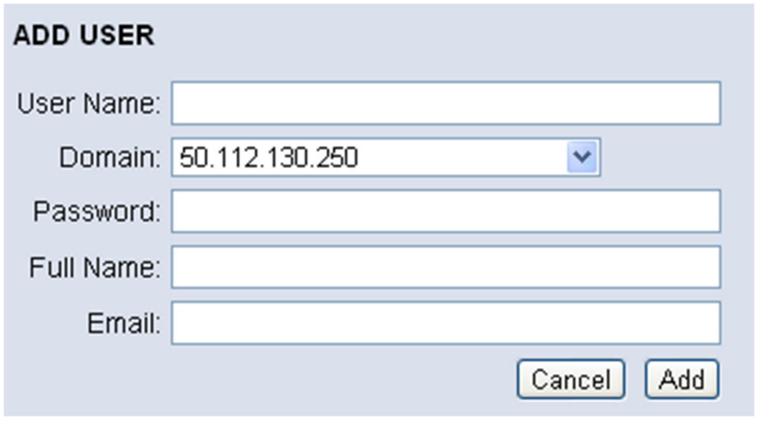
Access lists are used as a whitelist to allow gateways and other trusted nodes to skip authentication.

Note: If hostnames or FQDN's are used then a TLS transport type is assumed. All other transport types must specify ACLs by address.

Users / Digest Authenticator Monkey Settings

User List – A list of User Names with associated Domain, Password, Full Name and email address settings. This information is used by the Digest Authenticator to digest challenge requests.

Note: Full Name and Email are for display purposes only.



ADD USER

User Name:	
Domain:	50.112.130.250
Password:	
Full Name:	
Email:	

Request Filters Monkey Settings

Request Filters are a list of conditions that when met can be used to reject or accept a particular request. Utilization of a MySQL database routine to make this decision is also possible. Request filters allow two regular expression conditions that can be applied to any SIP message header (including the request-line, standard SIP headers and custom SIP headers). If a header that can appear multiple times is specified, then each instance of the header is checked.

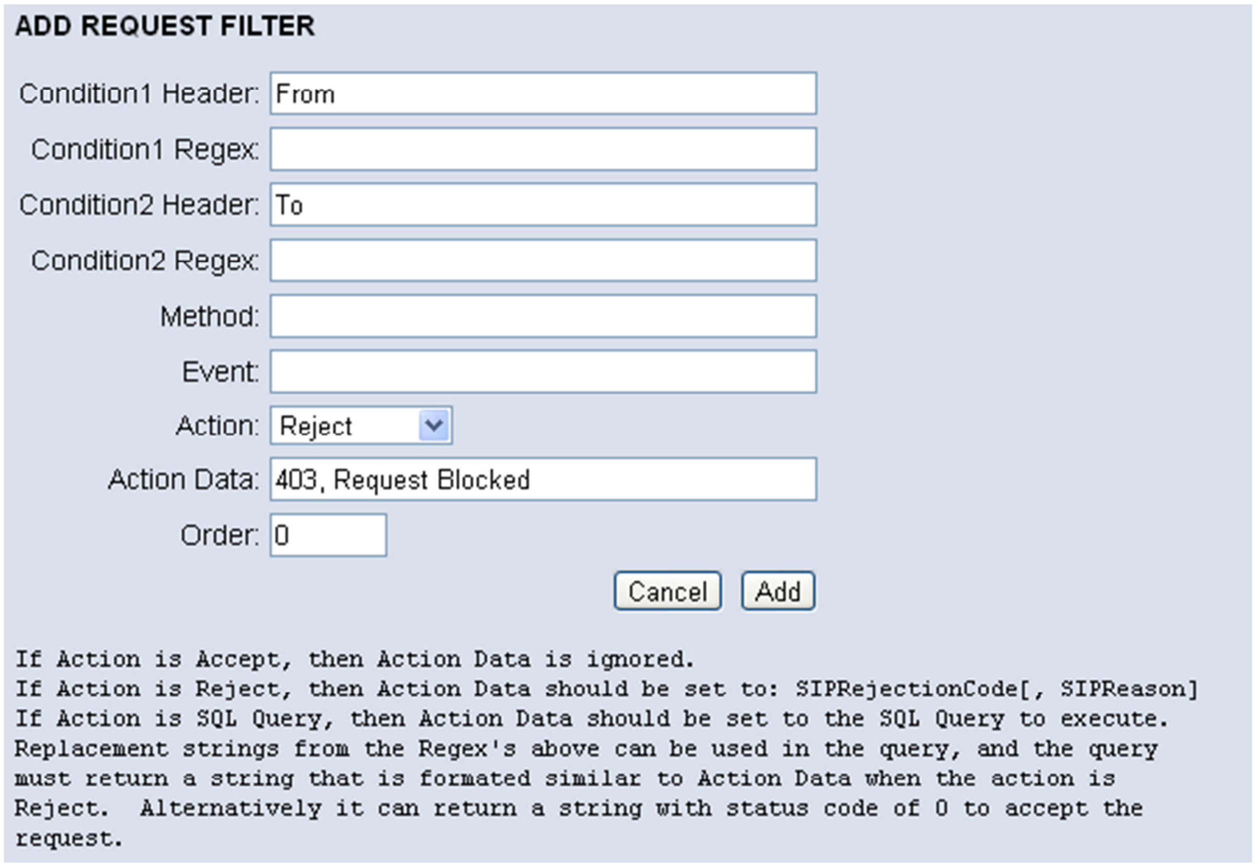
When conditions are met, the action carried out can be defined as one of:

- Accept - accepts this request and stops further processing in Request Filter monkey
- Reject - rejects this request with the provided SIP status code and reason text
- SQL query - only available when MySQL support is compiled in - runs an arbitrary stored procedure or query, using replacement strings from the two condition regular expressions
    - query must return an empty string or "0" to instruct repro to Accept the request, or a string containing "<SIP Reject Status Code>[, <SIP Reject Reason>]" to Reject the request
    - using the repro configuration file the SQL Query can be configured to operate on a completely different MySQL instance/server than the repro configuration

There is an ability to test the condition regular expressions from the Show Filters web page.

Other settings are configured in the repro configuration file or via the command line: DisableRequestFilterProcessor, RequestFilterDefaultNoMatchBehavior, RequestFilterDefaultDBErrorBehavior, RequestFilterMySQLServer (and other mySQL related settings)

Request Filters can be used to implement "User Blocking" functionality - ie. calls and instant messages from user X to user Y should always be blocked, because user X is in user Y's block list.

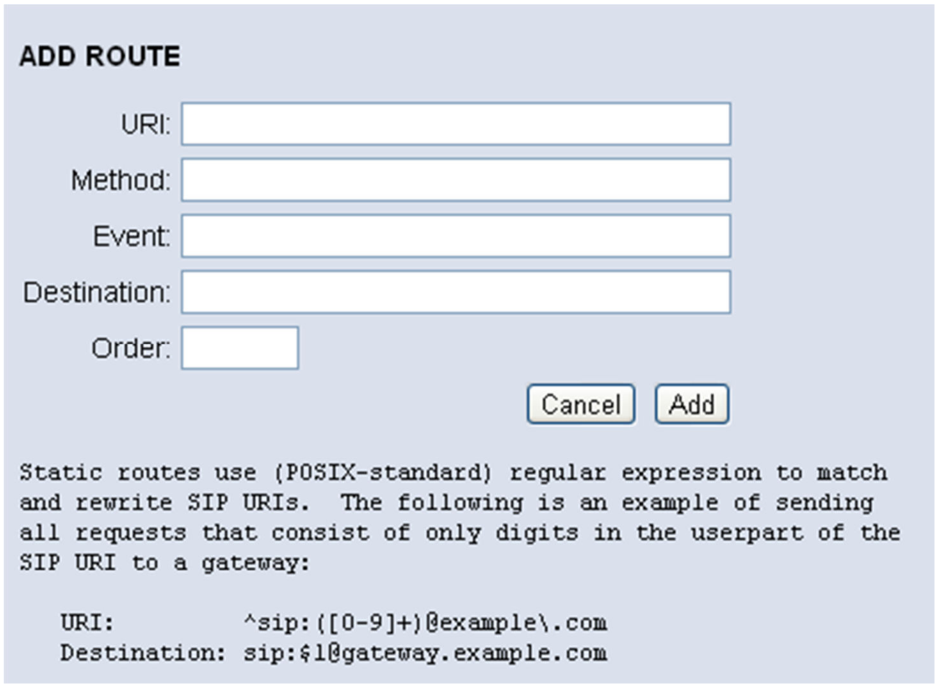

ADD REQUEST FILTER

Condition1 Header:	From
Condition1 Regex:	
Condition2 Header:	To
Condition2 Regex:	
Method:	
Event:	
Action:	Reject
Action Data:	403, Request Blocked
Order:	0

```
If Action is Accept, then Action Data is ignored.
If Action is Reject, then Action Data should be set to: SIPRejectionCode[, SIPReason]
If Action is SQL Query, then Action Data should be set to the SQL Query to execute.
Replacement strings from the Regex's above can be used in the query, and the query
must return a string that is formated similar to Action Data when the action is
Reject.  Alternatively it can return a string with status code of 0 to accept the
request.
```

Routes / Static Route Monkey Settings

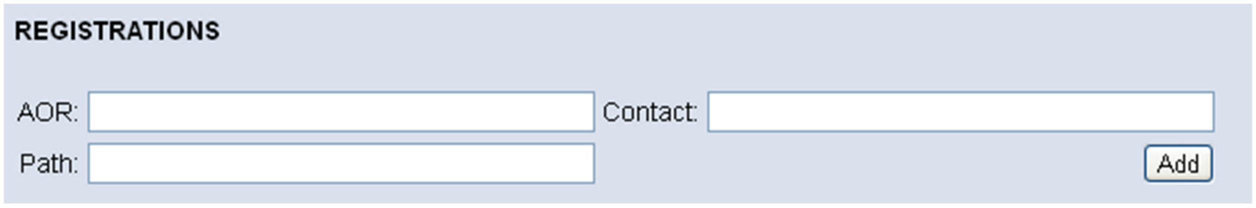
Routes – a list of routes that are selected based on a regular expression examination of the incoming request URI, and or SIP method, and or RFC3265 event type. Routes include a destination expression that is used to re-write the request URI to a new destination.



```
Static routes use (POSIX-standard) regular expression to match
and rewrite SIP URIs.  The following is an example of sending
all requests that consist of only digits in the userpart of the
SIP URI to a gateway:

   URI:         ^sip:([0-9]+)@example\.com
   Destination: sip:$1@gateway.example.com
```

Registrations

Offers and ability for a repro admin to add manual / permanent registrations. Such manually added registrations are persisted to the database, and loaded at startup. Manually added registrations never expire.



Settings

The settings page displays all of the currently active settings. It also shows some internal information about the SipStack and internal fifo sizes, Congestion Manager statistics (if enabled) and a snapshot of the DNS cache. The settings page also offers two buttons at the bottom:

- Clear DNS Cache – this flushes the cash and forces a new DNS lookups
- Restart Proxy – this will restart the proxy, applying any changes made to the configuration file, without losing the in memory registrations. Note: this button only appears if the CommandPort setting is non 0.

Configuration File / Command Line Settings – Global

The following settings are configurable via a configuration file and are optionally override able via the command line:

Command line format is:

```
repro [<ConfigFilename>] [--<ConfigValueName>=<ConfigValue>] [--<ConfigValueName>=<ConfigValue>]
```

Sample Command lines:

```
repro repro.config --RecordRouteUri=sip:proxy.sipdomain.com --ForceRecordRouting=true
repro repro.config /RecordRouteUri:sip:proxy.sipdomain.com /ForceRecordRouting:true
```

Sample Configuration file:

```
########################################################
# repro configuration file
########################################################


########################################################
# Log settings
########################################################

# Logging Type: syslog|cerr|cout|file
# Note:  Logging to cout can negatively effect performance.
#        When repro is placed into production 'file' or
#        'syslog' should be used.
LoggingType = cout

# Logging level: NONE|CRIT|ERR|WARNING|INFO|DEBUG|STACK
LogLevel = INFO

# Log Filename
LogFilename = repro.log

# Log file Max Bytes
LogFileMaxBytes = 5242880


########################################################
# Transport settings
########################################################

# Local IP Address to bind SIP transports to. If left blank
# repro will bind to all adapters.
#IPAddress = 192.168.1.106
#IPAddress = 2001:5c0:1000:a::6d
IPAddress =

# Local port to listen on for SIP messages over UDP - 0 to disable
UDPPort = 5060

# Local port to listen on for SIP messages over TCP - 0 to disable
TCPPort = 5060

# Local port to listen on for SIP messages over TLS - 0 to disable
TLSPort = 5061

# Local port to listen on for SIP messages over DTLS - 0 to disable
DTLSPort = 0

# TLS domain name for this server (note: domain cert for this domain must be present)
```

```
TLSDomainName =

# Whether or not we ask for (Optional) or expect (Mandatory) TLS
# clients to present a client certificate
# Possible values:
#  None: client can connect without any cert, if a cert is sent, it is not checked
#  Optional: client can connect without any cert, if a cert is sent, it must be acceptable to us
#  Mandatory: client can not connect without any cert, cert must be acceptable to us
# How we decide if a cert is acceptable: it must meet two criteria:
# 1. it must be signed by a CA that we trust (see CADirectory)
# 2. the domain or full sip: URI in the cert must match the From: URI of all
#    SIP messages coming from the peer
TLSClientVerification = None

# Whether we accept the subjectAltName email address as if it was a SIP
# address (when checking the validity of a client certificate)
# Very few commercial CAs offer support for SIP addresses in subjectAltName
# For many purposes, an email address subjectAltName may be considered
# equivalent within a specific domain.
# Currently, this accepts such certs globally (for any incoming connection),
# not just for connections from the local users.
TLSUseEmailAsSIP = false

# Alternate and more flexible method to specify transports to bind to.  If specified here
# then IPAddress, and port settings above are ignored.
# Transports MUST be numbered in sequential order, starting from 1.  Possible settings are:
# Transport<Num>Interface = <IPAddress>:<Port>
# Transport<Num>Type = <'TCP'|'UDP'|'TLS'|'DTLS'> - default is UDP if missing
# Transport<Num>TlsDomain = <TLSDomain> - only required if transport is TLS or DTLS
# Transport<Num>TlsClientVerification = <'None'|'Optional'|'Mandatory'> - default is None
# Transport<Num>RecordRouteUri = <'auto'|URI> - if set to auto then record route URI
#                                               is automatically generated from the other
#                                               transport settings.  Otherwise explicity
#                                               enter the full URI you want repro to use.
#                                               Do not specify 'auto' if you specified
#                                               the IPAddress as INADDR_ANY (0.0.0.0).
#                                               If nothing is specified then repro will
#                                               use the global RecordRouteUri setting.
#
# Transport<Num>RcvBufLen = <SocketReceiveBufferSize> - currently only applies to UDP transports,
#                                                       leave empty to use OS default
# Example:
# Transport1Interface = 192.168.1.106:5060
# Transport1Type = TCP
# Transport1RecordRouteUri = auto
#
# Transport2Interface = 192.168.1.106:5060
# Transport2Type = UDP
# Transport2RecordRouteUri = auto
# Transport2RcvBufLen = 10000
#
# Transport3Interface = 192.168.1.106:5061
# Transport3Type = TLS
# Transport3TlsDomain = sipdomain.com
# Transport3TlsClientVerification = Mandatory
# Transport3RecordRouteUri = sip:h1.sipdomain.com;transport=TLS


# Comma separated list of DNS servers, overrides default OS detected list (leave blank
# for default)
DNSServers =

# Enable IPv6
EnableIPv6 = false

# Enable IPv4
DisableIPv4 = false

# Port on which to run the HTTP configuration interface and/or certificate server
# 0 to disable (default: 5080)
HttpPort = 5080
```

```
# disable HTTP challenges for web based configuration GUI
DisableHttpAuth = false

# Web administrator password
HttpAdminPassword = admin

# Port on which to listen for and send XML RPC messaging used in command processing
# 0 to disable (default: 5081)
CommandPort = 5081

# Port on which to listen for and send XML RPC messaging used in registration sync
# process - 0 to disable (default: 0)
RegSyncPort = 0

# Hostname/ip address of another instance of repro to synchronize registrations with
# (note xmlrpcport must also be specified)
RegSyncPeer =


########################################################
# Misc settings
########################################################

# Must be true or false, default = false, not supported on Windows
Daemonize = false

# On UNIX it is normal to create a PID file
# if unspecified, no attempt will be made to create a PID file
#PidFile = /var/run/repro/repro.pid

# Path to load certificates from (default:  "$(HOME)/.sipCerts on *nix, and c:\sipCerts
# on windows)
# Note that repro loads ALL root certificates found by the settings
# CertificatePath, CADirectory and CAFile.  Setting one option does
# not disable the other options.
# Certificates in this location have to match one of the filename
# patterns expected by the legacy reSIProcate SSL code:
#   domain_cert_NAME.pem, root_cert_NAME.pem, ...
CertificatePath =

# Path to load root certificates from
# Iff this directory is specified, all files in the directory
# will be loaded as root certificates, prefixes and suffixes are
# not considered
# Note that repro loads ALL root certificates found by the settings
# CertificatePath, CADirectory and CAFile.  Setting one option does
# not disable the other options.
# On Debian, the typical location is /etc/ssl/certs
#CADirectory = /etc/ssl/certs

# Specify a single file containing one or more root certificates
# and possible chain/intermediate certificates to be loaded
# Iff this filename is specified, the certificates in the file will
# be loaded as root certificates
#
# This does NOT currently support bundles of unrelated root certificates
# stored in the same PEM file, it ONLY supports related/chained root
# certificates.  If multiple roots must be supported, use the CADirectory
# option.
#
# In the future, this behavior may change to load a bundle,
# such as /etc/ssl/certs/ca-certificates.txt on Debian and
# /etc/pki/tls/cert.pem on Red Hat/CentOS
#
# Note that repro loads ALL root certificates found by the settings
# CertificatePath, CADirectory and CAFile.  Setting one option does
# not disable the other options.
#
# This example loads just the CACert.org chain, which typically
# includes the class 1 root and the class 3 root (signed by the class 1 root)
```

```
#CAFile = /etc/ssl/certs/cacert.org.pem

# The Path to read and write Berkely DB database files
DatabasePath = ./

# The hostname running MySQL server to connect to, leave blank to use BerkelyDB.
# The value of host may be either a host name or an IP address. If host is "localhost",
# a connection to the local host is assumed. For Windows, the client connects using a
# shared-memory connection, if the server has shared-memory connections enabled. Otherwise,
# TCP/IP is used. For Unix, the client connects using a Unix socket file. For a host value of
# "." on Windows, the client connects using a named pipe, if the server has named-pipe
# connections enabled. If named-pipe connections are not enabled, an error occurs.
# WARNING: repro must be compiled with the USE_MYSQL flag in order for this work.
MySQLServer =

# The MySQL login ID to use when connecting to the MySQL Server. If user is empty string "",
# the current user is assumed. Under Unix, this is the current login name. Under Windows,
# the current user name must be specified explicitly.
MySQLUser = root

# The password for the MySQL login ID specified.
MySQLPassword = root

# The database name on the MySQL server that contains the repro tables
MySQLDatabaseName = repro

# If port is not 0, the value is used as the port number for the TCP/IP connection. Note that
# the host parameter determines the type of the connection.
MySQLPort = 3306

# The Users and MessageSilo database tables are different from the other repro configuration
# database tables, in that they are accessed at runtime as SIP requests arrive.  It may be
# desirable to use BerkeleyDb for the other repro tables (which are read at starup time, then
# cached in memory), and MySQL for the runtime accessed tables; or two seperate MySQL instances
# for these different table sets.  Use the following settings in order to specify a seperate
# MySQL instance for use by the Users and MessageSilo tables.
#
# WARNING: repro must be compiled with the USE_MYSQL flag in order for this work.
#
# Note:  If this setting is left blank then repro will fallback all remaining my sql
# settings to use the global MySQLServer settings.  If the MySQLServer setting is also
# blank, then repro will use BerkelyDB for all configuration tables.  See the
# documentation on the global MySQLServer settings for more details on the following
# individual settings.
RuntimeMySQLServer =
RuntimeMySQLUser = root
RuntimeMySQLPassword = root
RuntimeMySQLDatabaseName = repro
RuntimeMySQLPort = 3306

# If you would like to be able to authenticate users from a MySQL source other than the repro
user
# database table itself, then specify the query here.  The following conditions apply:
# 1.  The database table must reside on the same MySQL server instance as the repro database
#     or Runtime tables database.
# 2.  The statement provided will be UNION'd with the hardcoded repro query, so that auth from
#     both sources is possible.  Note:  If the same user exists in both tables, then the repro
#     auth info will be used.
# 3.  The provided SELECT statement must return the SIP A1 password hash of the user in question.
# 4.  The provided SELECT statement must contain two tags embedded into the query: $user and
$domain
#     These tags should be used in the WHERE clause, and repro will replace these tags with the
#     actual user and domain being queried.
# Example:  SELECT sip_password_ha1 FROM directory.users WHERE sip_userid = '$user' AND
#           sip_domain = '$domain' AND account_status = 'active'
MySQLCustomUserAuthQuery =

# Session Accounting - When enabled resiprocate will push a JSON formatted
# events for sip session related messaging that the proxy receives,
# to a persistent message queue that uses berkeleydb backed storage.
# The following session events are logged:
```

```
#   Session Created - INVITE passing authentication was received
#   Session Routed - received INVITE was forward to a target
#   Session Redirected - session was 3xx redirected or REFERed
#   Session Established - there was 2xx answer to an INVITE (only generate for first 2xx)
#   Session Cancelled - CANCEL was received
#   Session Ended - BYE was received from either end
#   Session Error - a 4xx, 5xx, or 6xx response was sent to the inviter
# Consuming Accounting Events:
# Users must ensure that this message queue is consumed, or it will grow without
# bound.  A queuetostream consumer process is provided, that will consume the
# events from the message queue and stream them to stdout.  This output stream can
# be consumed by linux scripting tools and converted to database records or some
# other relevant representation of the data.
# For example: ./queuetostream ./sessioneventqueue > streamconsumer
# In the future a MySQL consumer may also be provided in order to update
# session accounting records in a MySQL database table.
SessionAccountingEnabled = false

# The following setting determines if repro will add routing header information
# (ie. Route, and Record-Route headers)to the Session Created, Session Routed
# and Session Established events.
SessionAccountingAddRoutingHeaders = false

# The following setting determines if we will add via header information to
# the Session Created event.
SessionAccountingAddViaHeaders = false

# Registration Accounting - When enabled resiprocate will push a JSON formatted
# events for every registration, re-registration, and unregistration message
# received to a persistent message queue that uses berkeleydb backed storage.
# The following registration events are logged:
#   Registration Added - initial registration received
#   Registration Refreshed - registration refresh received / re-register
#   Registration Removed - registration removed by client / unregister
#   Registration Removed All - all contacts registration remove / unregister
# Consuming Accounting Events:
# Users must ensure that this message queue is consumed, or it will grow without
# bound.  A queuetostream consumer process is provided, that will consume the
# events from the message queue and stream them to stdout.  This output stream can
# be consumed by linux scripting tools and converted to database records or some
# other relevant representation of the data.
# For example: ./queuetostream ./regeventqueue > streamconsumer
# In the future a MySQL consumer may also be provided in order to update
# login/registration accounting records in a MySQL database table.
RegistrationAccountingEnabled = false

# The following setting determines if repro will add routing header information
# (ie. Route and Path headers)to registration accounting events.
RegistrationAccountingAddRoutingHeaders = false

# The following setting determines if we will add via header information to
# the registration accounting events.
RegistrationAccountingAddViaHeaders = false

# The following setting determines if we log the RegistrationRefreshed events
RegistrationAccountingLogRefreshes = false

# Run a Certificate Server - Allows PUBLISH and SUBSCRIBE for certificates
EnableCertServer = false

# Value of server header for local UAS responses
ServerText =

# Enables Congestion Management
CongestionManagement = true

# Congestion Management Metric - can take one of the following values:
# SIZE : Based solely on the number of messages in each fifo
# TIME_DEPTH : Based on the age of the oldest (front-most) message
#              in each fifo.
# WAIT_TIME : Based on the expected wait time for each fifo; this is
#             calculated by multiplying the size by the average service time.
#             This is the recommended metric.
CongestionManagementMetric = WAIT_TIME

# Congestion Management Tolerance for the given metric.  This determines when the
RejectionBehavior
# changes.
# 0-80 percent of max tolerance -> NORMAL (Not rejecting any work.)
# 80-100 percent of max tolerance -> REJECTING_NEW_WORK (Refuses new work,
#        not continuation of old work.)
# >100 percent of max tolerance -> REJECTING_NON_ESSENTIAL (Rejecting all work
#      that is non-essential to the health of the system (ie, if dropping
#      something is liable to cause a leak, instability, or state-bloat, don't drop it.
#      Otherwise, reject it.)
# Units specified are dependent on Metric specified above:
#  If Metric is SIZE then units are number of messages
#  If Metric is TIME_DEPTH then units are the number seconds old the oldest message is
#  If Metric is WAIT_TIME then units are the expected wait time of each fifo in milliseconds
CongestionManagementTolerance = 200

# Specify the number of seconds between writes of the stack statistics block to the log files.
# Specifying 0 will disable the statistics collection entirely.  If disabled the statistics
# also cannot be retreived using the reprocmd interface.
StatisticsLogInterval = 3600

# Use MultipleThreads stack processing.
ThreadedStack = true

# The number of worker threads used to asynchronously retrieve user authentication information
# from the database store.
NumAuthGrabberWorkerThreads = 2

# The number of worker threads in Async Processor tread pool.  Used by all Async Processors
# (ie. RequestFilter)
NumAsyncProcessorWorkerThreads = 2

# Specify domains for which this proxy is authorative (in addition to those specified on web
# interface) - comma separate list
# Note:  Domains specified here cannot be used when creating users, domains used in user
#        AORs must be specified on the web interface.
Domains =

# Uri to use as Record-Route
RecordRouteUri =

# Force record-routing
# WARNING: Before enabling this, ensure you have a RecordRouteUri setup, or are using
# the alternate transport specification mechanism and defining a RecordRouteUri per
# transport: TransportXRecordRouteUri
ForceRecordRouting = false

# Assume path option
AssumePath = false

# Disable registrar
DisableRegistrar = false

# Specify a comma separate list of enum suffixes to search for enum dns resolution
EnumSuffixes =

# Specify length of timer C in sec (0 or negative will disable timer C) - default 180
TimerC = 180

# Override the default value of T1 in ms (you probably should not change this) - leave
# as 0 to use default of 500ms)
TimerT1 = 0

# Disable outbound support (RFC5626)
# WARNING: Before enabling this, ensure you have a RecordRouteUri setup, or are using
# the alternate transport specification mechanism and defining a RecordRouteUri per
# transport: TransportXRecordRouteUri
```

```
DisableOutbound = true

# Set the draft version of outbound to support (default: RFC5626)
# Other accepted values are the versions of the IETF drafts, before RFC5626 was issued
# (ie. 5, 8, etc.)
OutboundVersion = 5626

# There are cases where the first hop in a particular network supports the concept of outbound
# and ensures all messaging for a client is delivered over the same connection used for
# registration.  This could be a SBC or other NAT traversal aid router that uses the Path
# header.  However such endpoints may not be 100% compliant with outbound RFC and may not
# include a ;ob parameter in the path header.  This parameter is required in order for repro
# to have knowledge that the first hop does support outbound, and it will reject registrations
# that appear to be using outboud (ie. instanceId and regId) with a 439 (First Hop Lacks Outbound
# Support).  In this case it can be desirable when using repro as the registrar to not reject
# REGISTRATION requests that contain an instanceId and regId with a 439.
# If this setting is enabled, then repro will assume the first hop supports outbound
# and not return this error.
AssumeFirstHopSupportsOutbound = false

# Enable use of flow-tokens in non-outbound cases
# WARNING: Before enabling this, ensure you have a RecordRouteUri setup, or are using
# the alternate transport specification mechanism and defining a RecordRouteUri per
# transport: TransportXRecordRouteUri
EnableFlowTokens = false

# Enable use of flow-tokens in non-outbound cases for clients detected to be behind a NAT.
# This a more selective flow token hack mode for clients not supporting RFC5626.  The
# original flow token hack (EnableFlowTokens) will use flow tokens on all client requests.
# Possible values are:  DISABLED, ENABLED and PRIVATE_TO_PUBLIC.
# WARNING: Before enabling this, ensure you have a RecordRouteUri setup, or are using
# the alternate transport specification mechanism and defining a RecordRouteUri per
# transport: TransportXRecordRouteUri
ClientNatDetectionMode = DISABLED

# Set to greater than 0 to enable addition of Flow-Timer header to REGISTER responses if
# outbound is enabled (default: 0)
FlowTimer = 0


########################################################
# CertificateAuthenticator Monkey Settings
########################################################

# Enables certificate authenticator - note you MUST use a TlsTransport
# with TlsClientVerification set to Optional or Mandatory.
# There are two levels of checking:
# a) cert must be signed by a CA trusted by the stack
# b) the CN or one of the subjectAltName values must match the From:
#    header of each SIP message on the TlsConnection
# Examples:
# Cert 1:
#    common name = daniel@pocock.com.au
#    => From: <daniel@pocock.com.au> is the only value that will pass
# Cert 2:
#    subjectAltName = pocock.com.au
#    => From: <<anything>@pocock.com.au> will be accepted
# Typically, case 1 is for a real client connection (e.g. Jitsi), case 2
# (whole domain) is for federated SIP proxy-to-proxy communication (RFC 5922)
EnableCertificateAuthenticator = false


########################################################
# DigestAuthenticator Monkey Settings
########################################################

# Disable DIGEST challenges - disables this monkey
DisableAuth = false

# Http hostname for this server (used in Identity headers)
HttpHostname =
# Disable adding identity headers
DisableIdentity = false

# Enable addition and processing of P-Asserted-Identity headers
EnablePAssertedIdentityProcessing = false

# Disable auth-int DIGEST challenges
DisableAuthInt = false

# Send 403 if a client sends a bad nonce in their credentials (will send a new
# challenge otherwise)
RejectBadNonces = false

# allow To tag in registrations
AllowBadReg = false


########################################################
# RequestFilter Monkey Settings
########################################################

# Disable RequestFilter monkey processing
DisableRequestFilterProcessor = false

# Default behavior for when no matching filter is found.  Leave empty to allow
# request processing to continue.  Otherwise set to a SIP status error code
# (400-699) that should be used to reject the request (ie. 500, Server Internal
# Error).
# The status code can optionally be followed by a , and SIP reason text.
RequestFilterDefaultNoMatchBehavior =

# Default behavior for SQL Query db errors.  Leave empty to allow request processing
# to continue.  Otherwise set to a SIP status error code (400-699) that should be
# used to reject the request (ie. 500 - Server Internal Error).
# The status code can optionally be followed by a , and SIP reason text.
# Note: DB support for this action requires MySQL support.
RequestFilterDefaultDBErrorBehavior = 500, Server Internal DB Error

# The hostname running MySQL server to connect to for any blocked entries
# that are configured to used a SQL statement.
# WARNING: repro must be compiled with the USE_MYSQL flag in order for this work.
#
# Note:  If this setting is left blank then repro will fallback all remaining my sql
# settings to use the global RuntimeMySQLServer or MySQLServer settings.  See the
# documentation on the global MySQLServer settings for more details on the following
# individual settings.
RequestFilterMySQLServer =
RequestFilterMySQLUser = root
RequestFilterMySQLPassword = root
RequestFilterMySQLDatabaseName =
RequestFilterMySQLPort = 3306


########################################################
# StaticRoute Monkey Settings
########################################################

# Specify where to route requests that are in this proxy's domain - disables the
# routes in the web interface and uses a SimpleStaticRoute monkey instead.
# A comma seperated list of routes can be specified here and each route will
# be added to the outbound Requests with the RequestUri left in tact.
Routes =

# Parallel fork to all matching static routes
ParallelForkStaticRoutes = false

# By default (false) we will stop looking for more Targets if we have found
# matching routes.  Setting this value to true will allow the LocationServer Monkey
# to run after StaticRoutes have been found.  In this case the matching
# StaticRoutes become fallback targets, processed only after all location server
# targets fail.
ContinueProcessingAfterRoutesFound = false


########################################################
# Message Silo Monkey Settings
########################################################

# Specify where the Message Silo is enabled or not.  If enabled,
# then repro will store MESSAGE requests for users that are not online.
# When the user is back online (ie. registers with repro), the stored
# messages will be delivered.
MessageSiloEnabled = false

# A regular expression that can be used to filter which URI's not to
# do message storage (siloing) for.  Destination/To URI's matching
# this regular expression will not be silo'd.
MessageSiloDestFilterRegex =

# A regular expression that can be used to filter which body/content/mime
# types not to do message storage (siloing) for.  Content-Type's matching
# this regular expression will not be silo'd.
MessageSiloMimeTypeFilterRegex = application\/im\-iscomposing\+xml

# The number of seconds a message request will be stored in the message silo.
# Messages older than this time, are candidates for deletion.
# Default (259200 seconds = 30 days)
MessageSiloExpirationTime = 2592000

# Flag to indicate if a Date header should be added to replayed SIP
# MESSAGEs from the silo, when a user registers.
MessageSiloAddDateHeader = true

# Defines the maximum message content length (bytes) that will be stored in
# the message silo.  Messages with a Content-Length larger than this
# value will be discarded.
# WARNING:  Do not increasing this value beyond the capabilities of the
# database storage or internal buffers.
# Note: AbstractDb uses a read buffer size of 8192 - do not exceed this size.
MessageSiloMaxContentLength = 4096

# The status code returned to the sender when a messages is successfully
# silo'd.
MessageSiloSuccessStatusCode = 202

# The status code returned to the sender when a messages mime-type matches
# the MessageSiloMimeTypeFilterRegex.  Can be used to avoid sending errors
# to isComposing mime bodies that don't need to be silod.  Set to 0 to use
# repro standard response (ie. 480).
MessageSiloFilteredMimeTypeStatusCode = 200

# The status code returned to the sender when a messages is not silo'd due
# to the MaxContentLength being exceeded.
MessageSiloFailureStatusCode = 480


########################################################
# Recursive Redirect Lemur Settings
########################################################

# Handle 3xx responses in the proxy - enables the Recursive Redirect Lemur
RecursiveRedirect = false


########################################################
# Geo Proximity Target Sorter Baboon Settings
########################################################

# If enabled, then this baboon can post-process the target list.
# This includes targets from the StaticRoute monkey and/or targets
# from the LocationServer monkey.  Requests that meet the filter
# criteria will have their Target list, flatened (serialized) and
# ordered based on the proximity of the target to the client sending
# the request.  Proximity is determined by looking for a
# x-repro-geolocation="<latitude>,<longitude>" parameter on the Contact
# header of a received request, or the Contact headers of Registration
# requests.  If this parameter is not found, then this processor will
# attempt to determine the public IP address closest to the client or
# target and use the MaxMind Geo IP library to lookup the geo location.
GeoProximityTargetSorting = false

# Specify the full path to the IPv4 Geo City database file
# Note:  A free version of the database can be downloaded from here:
# http://geolite.maxmind.com/download/geoip/database/GeoLiteCity.dat.gz
# For a more accurate database, please see the details here:
# http://www.maxmind.com/app/city
GeoProximityIPv4CityDatabaseFile = GeoLiteCity.dat

# Specify the full path to the IPv6 Geo City database file
# Note:  A free version of the database can be downloaded from here:
# http://geolite.maxmind.com/download/geoip/database/GeoLiteCityv6-beta/
# For a more accurate database, please see the details here:
# http://www.maxmind.com/app/city
# Leave blank to disable V6 lookups.  Saves memory (if not required).
#GeoProximityIPv6CityDatabaseFile = GeoLiteCityv6.dat
GeoProximityIPv6CityDatabaseFile =

# This setting specifies a PCRE compliant regular expression to attempt
# to match against the request URI of inbound requests.  Any requests
# matching this expression, will have their targets sorted as described
# above.  Leave blank to match all requests.
GeoProximityRequestUriFilter = ^sip:mediaserver.*@mydomain.com$

# The distance (in Kilometers) to use for proximity sorting, when the
# Geo Location of a target cannot be determined.
GeoProximityDefaultDistance = 0

# If enabled, then targets that are determined to be of equal distance
# from the client, will be placed in a random order.
LoadBalanceEqualDistantTargets = true


########################################################
# Q-Value Target Handler Baboon Settings
########################################################

# Enable sequential q-value processing - enables the Baboon
QValue = true

# Specify forking behavior for q-value targets: FULL_SEQUENTIAL, EQUAL_Q_PARALLEL,
# or FULL_PARALLEL
QValueBehavior = EQUAL_Q_PARALLEL

# Whether to cancel groups of parallel forks after the period specified by the
# QValueMsBeforeCancel parameter.
QValueCancelBetweenForkGroups = true

# msec to wait before cancelling parallel fork groups when QValueCancelBetweenForkGroups
# is true
QValueMsBeforeCancel = 30000

# Whether to wait for parallel fork groups to terminate before starting new fork-groups.
QValueWaitForTerminateBetweenForkGroups = true

# msec to wait before starting new groups of parallel forks when
# QValueWaitForTerminateBetweenForkGroups is false
QValueMsBetweenForkGroups = 3000
```

SIP Stack Statistics Overview

The following statistics are collected by the resiprocate stack and written to the resiprocate log files periodically when the stack is up and running:

- tuFifoSize – a snapshot of the number of messages waiting in the TU's fifo to be processed
- transportFifoSizeSum – a snapshot of the number of messages waiting in the transport fifo to be processed
- transactionFifoSize – a snapshot of the number of messages waiting in the transaction fifo to be processed
- activeTimers - a snapshot of the number of currently active timers in the stack
- activeClientTransactions – a snapshot of the number of currently active client transactions being managed by the stack
- activeServerTransactions – a snapshot of the number of currently active server transactions being managed by the stack
- requestsSent – the total number of requests sent since start up, including retransmissions
- responsesSent – the total number of responses sent since start up, including retransmissions
- requestsRetransmitted – the total number of request retransmissions sent since start up
- responsesRetransmitted – the total number of request retransmissions sent since start up
- requestsReceived – the total number of requests received since start up
- responsesReceived – the total number of responses received since start up
- responsesByCode[] – the total number of responses received since start up for each response code seen
- requestsSentByMethod[] – the total number of requests sent since start up, including retransmissions, for each SIP method type
- requestsRetransmittedByMethod[] – the total number of request retransmissions sent since start up, for each SIP method type
- responsesSentByMethod[] – the total number of responses sent since start up, including retransmissions, for each SIP method type
- responsesRetransmittedByMethod[] – the total number of response retransmissions sent since start up, for each SIP method type
- responsesReceivedByMethod[] – the total number of responses received since start up for each SIP method type
- responsesSentByMethodByCode[][] - the total number of responses sent since start up, including retransmissions, for each SIP method type and response code sent
- responsesRetransmittedByMethodByCode[][] - the total number of response retransmissions sent since start up, for each SIP method type and response code sent
- responsesReceivedByMethodByCode[][] - the total number of responses received since start up for each sip method type and response code seen
- sum2xxIn – the total number of success responses received by sip method type

- sumErrIn – the total number of failure responses received by sip method type
- sum2xxOut – the total number of success responses sent by sip method type
- sumErrOut – the total number of failure responses sent by sip method type

Sample Log file output string is:

```
TU summary: 0 TRANSPORT  0 TRANSACTION 0 CLIENTTX 0 SERVERTX 0 TIMERS 0
Transaction summary: reqi 0 reqo 0 rspi 0 rspo 0
Details: INVi 0/S0/F0 INVo 0/S0/F0 ACKi 0 ACKo 0 BYEi 0/S0/F0 BYEo 0/S0/F0 CANi 0/S0/F0 CANo
0/S0/F0 MSGi 0/S0/F0 MSGo 0/S0/F0 OPTi 0/S0/F0 OPTo 0/S0/F0 REGi 0/S0/F0 REGo 0/S0/F0 PUBi
0/S0/F0 PUBo 0/S0/F0 SUBi 0/S0/F0 SUBo 0/S0/F0 NOTi 0/S0/F0 NOTo 0/S0/F0 REFi 0/S0/F0 REFo
0/S0/F0 INFi 0/S0/F0 INFo 0/S0/F0 PRAi 0/S0/F0 PRAo 0/S0/F0 SERi 0/S0/F0 SERo 0/S0/F0 UDPi
0/S0/F0 UDPo 0/S0/F0
Retransmissions: INVx 0 finx 0 nonx 0 BYEx 0 CANx 0 MSGx 0 OPTx 0 REGx 0 PUBx 0 SUBx 0 NOTx 0
REFx 0 INFx 0 PRAx 0 SERx 0 UDPx 0
```

Congestion Manager Overview

Repro has a built in and optional congestion manager that can detect when repro is congested (over worked), by inspecting the state of the internal FIFOs. If it the stack determines that is congested then it takes necessary action to shed its load, ie: rejecting new inbound requests.

Shedding Load

- Efficient wait-time estimation in AbstractFifo; keeps track of how rapidly messages are consumed, allowing good estimates of how long a new message will take to be serviced. More efficient than the time-depth logic in TimeLimitFifo, and a better predictor too.
- The ability to shed load at the transport level when the TransactionController is congested.
- The ability to shed load coming from the TU when the TransactionController is congested. This is crucial when congestion is being caused by a TU trying to do too much.
- The TransactionController will defer retransmissions of requests if sufficiently congested (ie; the response is probably stuck in mStateMacFifo)

Configurable Settings

Congestion Management Metric

Can take one of the following values:

- SIZE : Based solely on the number of messages in each fifo
- TIME_DEPTH : Based on the age of the oldest (front-most) message in each fifo.
- WAIT_TIME : Based on the expected wait time for each fifo; this is calculated by multiplying the size by the average service time.   This is the recommended metric.

Congestion Management Tolerance

The tolerance setting for the given metric. This determines when the RejectionBehavior changes.

- 0-80 percent of max tolerance -> NORMAL (Not rejecting any work.)
- 80-100 percent of max tolerance -> REJECTING_NEW_WORK (Refuses new work, not continuation of old work.)
- >100 percent of max tolerance -> REJECTING_NON_ESSENTIAL (Rejecting all work that is non-essential to the health of the system (ie, if dropping something is liable to cause a leak, instability, or state-bloat, don't drop it. Otherwise, reject it.)

Units specified are dependent on Metric specified:

- If Metric is SIZE then units are number of messages
- If Metric is TIME_DEPTH then units are the number seconds old the oldest message is
- If Metric is WAIT_TIME then units are the expected wait time of each fifo in milliseconds

Sample Congestion Stats Output

```
UdpTransport::mTxFifo: Size=0 TimeDepth(sec)=0 ExpWait(msec)=0 AvgSvcTime(usec)=0
Metric=WAIT_TIME MaxTolerance=200 CurBehavior=NORMAL

TcpTransport::mTxFifo: Size=0 TimeDepth(sec)=0 ExpWait(msec)=0 AvgSvcTime(usec)=0
Metric=WAIT_TIME MaxTolerance=200 CurBehavior=NORMAL

TlsTransport::mTxFifo: Size=0 TimeDepth(sec)=0 ExpWait(msec)=0 AvgSvcTime(usec)=0
Metric=WAIT_TIME MaxTolerance=200 CurBehavior=NORMAL

TransactionController::mStateMacFifo: Size=0 TimeDepth(sec)=0 ExpWait(msec)=0 AvgSvcTime(usec)=8
Metric=WAIT_TIME MaxTolerance=200 CurBehavior=NORMAL

SipStack::mTUFifo: Size=0 TimeDepth(sec)=0 ExpWait(msec)=0 AvgSvcTime(usec)=0 Metric=WAIT_TIME
MaxTolerance=200 CurBehavior=NORMAL
```

Repro Command Interface

Repro will listen on a configured TCP port for inbound XML formatted request messages to perform certain operations or retrieve internal state information.

Available Commands

GetStackInfo - retrieves low level information about the stack state

GetStackStats - retrieves a dump of the stack statistics

ResetStackStats - resets all cumulative stack statistics to zero

LogDnsCache - causes the DNS cache contents to be written to the resip logs

ClearDnsCache - empties the stacks DNS cache

GetDnsCache - retrieves the DNS cache contents

GetCongestionStats - retrieves the stacks congestion manager stats and state

SetCongestionTolerance metric - sets congestion tolerances

Shutdown - signal the proxy to shut down.

Restart - signal the proxy to restart - leaving active registrations in place.

GetProxyConfig - retrieves the all of configuration file settings currently being used by the proxy

reprocmd

reprocmd is a command line utility that can be used to send properly formatted commands to the repro command server, and display the results.

Command line format:

```
reprocmd <server> <port> <command> [<parm>=<value>]
  Valid Commands are:
  GetStackInfo - retrieves low level information about the stack state
  GetStackStats - retrieves a dump of the stack statistics
  ResetStackStats - resets all cumulative stack statistics to zero
  LogDnsCache - causes the DNS cache contents to be written to the resip logs
  ClearDnsCache - emptys the stacks DNS cache
  GetDnsCache - retrieves the DNS cache contents
  GetCongestionStats - retrieves the stacks congestion manager stats and state
  SetCongestionTolerance metric=<SIZE|WAIT_TIME|TIME_DEPTH> maxTolerance=<value>
                         [fifoDescription=<desc>] - sets congestion tolerances
  Shutdown - signal the proxy to shut down.
  Restart - signal the proxy to restart - leaving active registrations in place.
  GetProxyConfig - retrieves the all of configuration file settings currently
                   being used by the proxy
```

Sample Output:

```
>reprocmd 127.0.0.1 5081 GetDnsCache
DNS cache retrieved.
DNSCACHE: Type=A(Host): proxy.sipdomain.com -> 66.66.111.111 secsToExpirey=824 status=0
DNSCACHE: Type=SRV: _sip._tcp.sipdomain.com -> proxy.sipdomain.com:5060 priority=0 weight=0
secsToExpirey=43124 status=0
DNSCACHE: Type=SRV: _sip._udp.sipdomain.com -> proxy.sipdomain.com:5060 priority=0 weight=0
secsToExpirey=824 status=0
```

Sample XML Messaging

The following is an example of how these commands are formatted on the wire:

Sending:

```
<SetCongestionTolerance>
  <Request>
    <metric>TIME_DEPTH</metric>
    <maxTolerance>300</maxTolerance>
  </Request>
</SetCongestionTolerance>
```

Received response:

```
<SetCongestionTolerance>
  <Request>
    <metric>TIME_DEPTH</metric>
    <maxTolerance>300</maxTolerance>
  </Request>

  <Response>
    <Result Code="200">Congestion Tolerance set.</Result>
  </Response>
</SetCongestionTolerance>
```

Sending:

```
<GetDnsCache>
  <Request>
  </Request>
</GetDnsCache>
```

Received response:

```
<GetDnsCache>
  <Request>
  </Request>

  <Response>
    <Result Code="200">DNS cache retrieved.</Result>
    <Data>
DNSCACHE: Type=A(Host): proxy.sipdomain.com -> 66.66.111.111 secsToExpirey=824 status=0
DNSCACHE: Type=SRV: _sip._tcp.sipdomain.com -> proxy.sipdomain.com:5060 priority=0 weight=0
secsToExpirey=43124 status=0
DNSCACHE: Type=SRV: _sip._udp.sipdomain.com -> proxy.sipdomain.com:5060 priority=0 weight=0
secsToExpirey=824 status=0
    </Data>
  </Response>
</GetDnsCache>
```

Accounting Overview

**Note: The accounting feature just missed the deadline for inclusion in the 1.8 release. It will be included in the next version of repro – however it is still described here and is available directly via SVN checkout.

The repro accounting collector engine gathers events and data and places them in an event queue that is persisted to disk using a BerkeleyDb backing store. This queuing mechanism ensures that events are not lost when blocking occurs on the consumer side.   The event data is formatted as a JSON (http://www.json.org/) text blob to facilitate easy parsing using scripting engines. Parties that are interested in using this data for accounting or billing purposes are required to implement an accounting event consumer.   There are currently two types of accounting events generated by repro: session accounting events and registration accounting events.

Accounting Consumers

Users must ensure that the accounting message queues are consumed, or they will grow without bound and take up space on the harddisk. The BerkeleyDb backing store is configured to create two separate directories, one for each accounting event type, that contains the backing store requiring consumption. A queuetostream consumer process is provided, that will consume the events from a particular message queue and stream them to stdout. This output stream can be consumed by linux scripting tools and converted to database records or some other relevant representation of the data.

For example:

```
./queuetostream ./sessioneventqueue > streamconsumer
```

```
./queuetostream ./regeventqueue > streamconsumer
```

In the future a MySQL consumer may also be provided in order to update accounting records in a MySQL database table.

Use the queuetostream source code as a guideline for how to create a custom consumers using C++ code. This code shows how to handle BerkeleyDb database recovery logic properly. At a basic level this code utilizes the PersistentMessageDequeue class that is part of reprolib (PersistentMessageQueue.hxx). This class provides three basic functions for consuming the queue:

1. bool pop(size_t numRecords, std::vector<resip::Data>& records, bool autoCommit);
   This method returns true for success, false for failure. It can return true and 0 records if none are available. If the autoCommit flag is used then the consumer is not required to call commit or abort. When using the autoCommit flag it is safe to allow multiple consumers, otherwise you must ensure there is only one consumer running per queue.
2. bool commit()
   If autoCommit is not used on the pop call, then the consumer is expected to consume the event (ie. write a record out to a database or called a stored procedure), and once everything is

successful, call commit to remove the event from queue. This mechanism is used to ensure we only remove events once successfully processed.

3. void abort();
   If autoCommit is not used on the pop call and the consumer has some form of failure when trying to consume the event (ie. can't connect to database), then this call can be used to leave the event on the queue.

Note: the queuetostream sample consumer uses the autoCommit flag.

Session Accounting

In SIP an INVITE request is used to establish a session. Session accounting generates JSON formatted events for SIP session related messaging that the proxy receives.

The repro configuration file contains the following settings for controlling session accounting:

- SessionAccountingEnabled – used to enable/disable this feature
- SessionAccountingAddRoutingHeaders - determines if repro will add routing header information (ie. Route, and Record-Route headers) to the Session Created, Session Routed, and Session Established events.
- SessionAccountingAddViaHeaders - determines if repro will add Via header information to the Session Created event.

All session events log the following data:

- EventId - a numeric value representing the event
- EventName – a friendly name for the event
- Datetime – the date and time that the event occurred in SIP Date header format
- CallId – the globally unique call id for the session (from SIP Call-ID header)

Data that is specific to an event are described below. In general if the request doesn't contain the SIP header then it will not be included in the JSON output.

The following session events are logged:

- Session Created (ID:1) – an INVITE request passing proxy authentication was received. The following additional properties/headers are logged:
    - RequestUri
    - To - DisplayName and Uri in subfields
    - From - DisplayName and Uri in subfields
    - Contact – sip Contact header
    - Vias[] – list of Via headers, if enabled
    - ClientPublicAddress – Transport, IP and Port in subfields – parsed from Via headers, this is the first public address (non RFC1918) found closest to the client
    - Routes[] – list of Route headers, if enabled

    - RecordRoutes[] – list of Record-Route headers, if enabled
    - UserAgent - SIP User-Agent header
- Session Routed (ID:2) – a received INVITE was forward to a target. It is possible get this event multiple times if the request is forked. The following additional properties/headers are logged:
    - TargetUri – if the INVITE was routed to a registered user, then this will contain the Contact header of the registration. If the request triggered a repro Route, then this will contain the Route target. If the request is simply forwarded, then this will contain the RequestUri of the original INVITE.
    - Routes[] – list of routes use on the forwarded/routed INVITE
- Session Redirected (ID:3) - session was redirected with a 3xx response or was REFERed. The following additional properties/headers are logged:
    - ReferredBy – DisplayName and Uri in subfields – from From header of REFER request
    - TargetUri – if a Refer this is the Refer-To header Uri
    - TargetUris[] – a list of redirect targets from the Contact headers of a 3xx response
    - UserAgent – SIP User-Agent header , only present for 3xx redirections
- Session Established (ID:4) - there was 2xx answer to an INVITE (only generate for first 2xx). The following additional properties/headers are logged:
    - Contact – SIP Contact header
    - RecordRoutes[] – list of Record-Route headers, if enabled
    - UserAgent – SIP User-Agent header
- Session Cancelled (ID:5) – a CANCEL request was received. The following additional properties/headers are logged:
    - Reason – Value, Cause and Text in subfields
- Session Ended (ID:6) – a BYE request was received from either end. The following additional properties/headers are logged:
    - From - DisplayName and Uri in subfields – identifies the party that disconnected
    - Reason – Value, Cause and Text in subfields
- Session Error (ID:7) - a 4xx, 5xx, or 6xx response was sent to the inviter. The following additional properties/headers are logged:
    - Status – Code and Text in subfields from status line of error reponse
    - Warning – Code and Text in subfields – from SIP Warning header
    - Reason – Value, Cause and Text in subfields – from SIP Reason header. Note: a reason header is not usually present on a response.
    - UserAgent – SIP User-Agent header

Sample Events:

```
{
        "EventId" : 1,
        "EventName" : "Session Created",
        "Datetime" : "Wed, 20 Jun 2012 17:28:41 GMT",
        "CallId" : "YzU5MzA3ZGI3YTU3ZDExNTBlYzFkMDNkMjRiNTZjZGE.",
        "RequestUri" : "sip:testuser@sip.testdomain.com",
        "To" : {
                "DisplayName" : "Test User",
                "Uri" : "sip:testuser@sip.testdomain.com"
        },
        "From" : {
                "DisplayName" : "Test User 2",
                "Uri" : "sip:testuser2@sip.testdomain.com"
        },
        "Contact" :
"<sip:testuser2@192.168.1.1:38520;ob;transport=tcp>;+sip.instance=\"<urn:uuid:FEEFEFEE-E3B0-4B51-
A2C2-2206E99F459C>\"",
        "Vias" : [
                "SIP/2.0/TCP 192.168.1.1:5060;branch=z9hG4bK-d5dede36x4a73d9b8",
                "SIP/2.0/TCP 127.0.0.1:38520;branch=z9hG4bK-d8754z-491cdc2a2e87785a-1---d8754z-
;rport=55073"
        ],
        "ClientPublicAddress" : {
                "Transport" : "TLS",
                "IP" : "99.250.1.1",
                "Port" : 55070
        },
        "RecordRoutes" : [
                "<sip:192.168.1.106:17439;lr>",
                "<sip:192.168.1.106:17439;lr;transport=TCP>"
        ],
        "UserAgent" : "Test UA 1.1.2"
}
{
        "EventId" : 2,
        "EventName" : "Session Routed",
        "Datetime" : "Wed, 20 Jun 2012 17:28:41 GMT",
        "CallId" : "YzU5MzA3ZGI3YTU3ZDExNTBlYzFkMDNkMjRiNTZjZGE.",
        "TargetUri" : "sip:testuser@192.168.1.2:42365;transport=tcp",
        "Routes" : [
                "<sip:192.168.1.117:16220;lr>"
        ]
}
{
        "EventId" : 4,
        "EventName" : "Session Established",
        "Datetime" : "Wed, 20 Jun 2012 17:28:47 GMT",
        "CallId" : "YzU5MzA3ZGI3YTU3ZDExNTBlYzFkMDNkMjRiNTZjZGE.",
        "Contact" :
"<sip:testuser@192.168.1.2:42365;ob;transport=tcp>;+sip.instance=\"<urn:uuid:AB34234-C2E7-4805-
A4AA-70194D3959DF>\"",
        "RecordRoutes" : [
                "<sip:192.168.1.117:16220;lr;transport=tcp>",
                "<sip:192.168.1.117:16220;lr;transport=TLS>",
                "<sip:192.168.1.106:17439;lr>",
                "<sip:192.168.1.106:17439;lr;transport=TCP>"
        ],
        "UserAgent" : " Test UA 1.1.2"
}
{
        "EventId" : 6,
        "EventName" : "Session Ended",
        "Datetime" : "Wed, 20 Jun 2012 17:28:54 GMT",
        "CallId" : "YzU5MzA3ZGI3YTU3ZDExNTBlYzFkMDNkMjRiNTZjZGE.",
        "From" : {
                "DisplayName" : "Test User",
                "Uri" : "sip:testuser@sip.testdomain.com"
        },
        "Reason" : {
                "Value" : "Ended",
                "Cause" : 0,
                "Text" : "User Hangup"
        }
}
```

Registration Accounting

Registration accounting generates JSON formatted events for SIP REGISTER messages that the proxy receives.

The repro configuration file contains the following settings for controlling registration accounting:

- RegistrationAccountingEnabled – used to enable/disable this feature
- RegistrationAccountingAddRoutingHeaders - determines if repro will add routing header information (ie. Route and Path headers) to the registration accounting events.
- RegistrationAccountingAddViaHeaders - determines if repro will add Via header information to the registration accounting events.
- RegistrationAccountingLogRefreshes – determines if repro will log the Registration Refresh event or not

All registration events log the following data:

- EventId - a numeric value representing the event
- EventName – a friendly name for the event
- Datetime – the date and time that the event occurred in SIP Date header format
- CallId – the globally unique call id for the registration (from SIP Call-ID header)
- User – DisplayName and Aor in subfields – taken from SIP To header
- From – DisplayName and Uri in subfields – taken from SIP From header, but only present if From header is different than To header (ie. third party registration)
- Contacts[] – a list of Contacts from the SIP Contact header
- Expires – the value of the SIP Expires header
- Vias[] – list of Via headers, if enabled
- ClientPublicAddress – Transport, IP and Port in subfields – parsed from Via headers, this is the first public address (non RFC1918) found closest to the client
- Routes[] – list of Route headers, if enabled
- Paths[] – list of Path headers, if enabled
- UserAgent – SIP User-Agent header

The following registration events are logged:

- Registration Added - initial REGISTER request received
- Registration Refreshed - registration refresh received / re-register
- Registration Removed - registration removed by client / unregister
- Registration Removed All - all contacts registration remove / unregister

Sample Event:

```
{
        "EventId" : 1,
        "EventName" : "Registration Added",
        "Datetime" : "Thu, 21 Jun 2012 15:07:08 GMT",
        "CallId" : "OWJlNDk2YTAxODJjMGVjMDU5M2JmN2U5YzE2ZWNmYWY.",
        "User" : {
                "Aor" : "sip:testuser@sip.testdomain.com"
        },
        "Contacts" : [
               "<sip:testuser@192.168.1.2:42365;transport=tcp>;+sip.instance=\"<urn:uuid:AB34234-
E3B0-4B51-A2C2-2206E99F459C>\";reg-
id=1;methods=\"INVITE,ACK,CANCEL,OPTIONS,BYE,UPDATE,OPTIONS,MESSAGE,NOTIFY,INFO,REFER\""
        ],
        "Expires" : 1800,
        "Vias" : [
                "SIP/2.0/TLS 192.168.1.106:17439;branch=z9hG4bK-d8754z-
cd4034f6d0585f7c4aa1e5efb15eaae2-1---d8754z-;rport=9343;received=99.250.1.2",
        ],
        "ClientPublicAddress" : {
                "Transport" : "TLS",
                "IP" : "99.250.1.2",
                "Port" : 9343
        },
        "Paths" : [
                "<sip:192.168.1.106:17439;lr>"
        ],
        "UserAgent" : "Test UA 1.1.2"
}
```

Request Filter Processor

Request Filters are a list of conditions that when met can be used to reject or accept a particular request. Utilization of a MySQL database routine to make this decision is also possible. Request filters allow two regular expression conditions that can be applied to any SIP message header (including the request-line, standard SIP headers and custom SIP headers). If a header that can appear multiple times is specified, then each instance of the header is checked.

When conditions are met, the action carried out can be defined as one of:

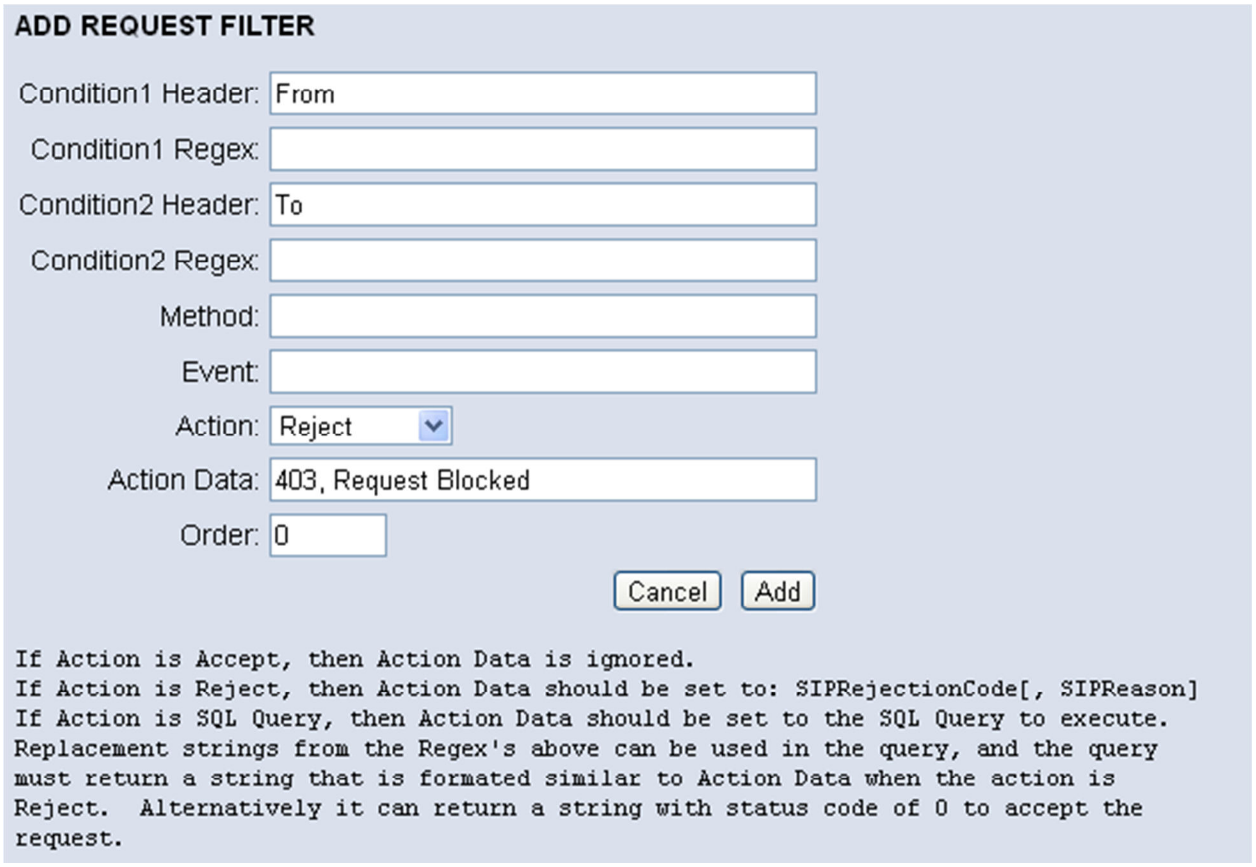
- Accept - accepts this request and stops further processing in Request Filter monkey
- Reject - rejects this request with the provided SIP status code and reason text
- SQL query - only available when MySQL support is compiled in - runs an arbitrary stored procedure or query, using replacement strings from the two condition regular expressions
    - query must return an empty string or "0" to instruct repro to Accept the request, or a string containing "<SIP Reject Status Code>[, <SIP Reject Reason>]" to Reject the request
    - using the repro configuration file the SQL Query can be configured to operate on a completely different MySQL instance/server than the repro configuration

There is an ability to test the condition regular expressions from the Show Filters web page.

Other settings are configured in the repro configuration file or via the command line:

```
########################################################
# RequestFilter Monkey Settings
########################################################

# Disable RequestFilter monkey processing
DisableRequestFilterProcessor = false

# Default behavior for when no matching filter is found.  Leave empty to allow
# request processing to continue.  Otherwise set to a SIP status error code
# (400-699) that should be used to reject the request (ie. 500, Server Internal
# Error).
# The status code can optionally be followed by a , and SIP reason text.
RequestFilterDefaultNoMatchBehavior =

# Default behavior for SQL Query db errors.  Leave empty to allow request processing
# to continue.  Otherwise set to a SIP status error code (400-699) that should be
# used to reject the request (ie. 500 - Server Internal Error).
# The status code can optionally be followed by a , and SIP reason text.
# Note: DB support for this action requires MySQL support.
RequestFilterDefaultDBErrorBehavior = 500, Server Internal DB Error

# The hostname running MySQL server to connect to for any blocked entries
# that are configured to used a SQL statement.
# WARNING: repro must be compiled with the USE_MYSQL flag in order for this work.
#
# Note:  If this setting is left blank then repro will fallback all remaining my sql
# settings to use the global RuntimeMySQLServer or MySQLServer settings.  See the
# documentation on the global MySQLServer settings for more details on the following
# individual settings.
RequestFilterMySQLServer =
RequestFilterMySQLUser = root
RequestFilterMySQLPassword = root
RequestFilterMySQLDatabaseName =
RequestFilterMySQLPort = 3306
```

Request Filters can be used to implement “User Blocking” functionality - ie. calls and instant messages from user X to user Y should always be blocked, because user X is in user Y's block list.



ADD REQUEST FILTER

Condition1 Header:	From
Condition1 Regex:	
Condition2 Header:	To
Condition2 Regex:	
Method:	
Event:	
Action:	Reject
Action Data:	403, Request Blocked
Order:	0

```
If Action is Accept, then Action Data is ignored.
If Action is Reject, then Action Data should be set to: SIPRejectionCode[, SIPReason]
If Action is SQL Query, then Action Data should be set to the SQL Query to execute.
Replacement strings from the Regex's above can be used in the query, and the query
must return a string that is formated similar to Action Data when the action is
Reject.  Alternatively it can return a string with status code of 0 to accept the
request.
```

Message Silo'ing

Message Silo'ing is used to store MESSAGE requests for users that are offline. Messages are delivered when an offline user registers with the proxy. The Message Silo is disabled by default, and is enabled via the configuration file. Stored messages are persisted to a database table, so they survive shutdowns. Configurable filters exist for destination URI, mime type, and message body size.

```
########################################################
# Message Silo Monkey Settings
########################################################

# Specify where the Message Silo is enabled or not.  If enabled,
# then repro will store MESSAGE requests for users that are not online.
# When the user is back online (ie. registers with repro), the stored
# messages will be delivered.
MessageSiloEnabled = false

# A regular expression that can be used to filter which URI's not to
# do message storage (siloing) for.  Destination/To URI's matching
# this regular expression will not be silo'd.
MessageSiloDestFilterRegex =

# A regular expression that can be used to filter which body/content/mime
# types not to do message storage (siloing) for.  Content-Type's matching
# this regular expression will not be silo'd.
MessageSiloMimeTypeFilterRegex = application\/im\-iscomposing\+xml

# The number of seconds a message request will be stored in the message silo.
# Messages older than this time, are candidates for deletion.
# Default (259200 seconds = 30 days)
MessageSiloExpirationTime = 2592000

# Flag to indicate if a Date header should be added to replayed SIP
# MESSAGEs from the silo, when a user registers.
MessageSiloAddDateHeader = true

# Defines the maximum message content length (bytes) that will be stored in
# the message silo.  Messages with a Content-Length larger than this
# value will be discarded.
# WARNING:  Do not increasing this value beyond the capabilities of the
# database storage or internal buffers.
# Note: AbstractDb uses a read buffer size of 8192 - do not exceed this size.
MessageSiloMaxContentLength = 4096

# The status code returned to the sender when a messages is successfully
# silo'd.
MessageSiloSuccessStatusCode = 202

# The status code returned to the sender when a messages mime-type matches
# the MessageSiloMimeTypeFilterRegex.  Can be used to avoid sending errors
# to isComposing mime bodies that don't need to be silod.  Set to 0 to use
# repro standard response (ie. 480).
MessageSiloFilteredMimeTypeStatusCode = 200

# The status code returned to the sender when a messages is not silo'd due
# to the MaxContentLength being exceeded.
MessageSiloFailureStatusCode = 480
```

Geo Proximity Target Sorter Processor

When enabled, then this baboon (target processor) can post-process the target list. This includes targets from the StaticRoute monkey (request processor) and/or targets from the LocationServer monkey. Requests that meet the filter criteria will have their target list flattened (serialized) and ordered based on the proximity of the target to the client sending the request. Proximity is determined by looking for a

x-repro-geolocation="<latitude>,<longitude>"

parameter on the Contact header of a received request, or the Contact headers of Registration requests. If this parameter is not found, then this processor will attempt to determine the public IP address closest to the client or target and use the MaxMind Geo IP library to lookup the geo location.

If multiple targets have the same proximity then they can be randomly load balanced by enabling the following configuration setting: LoadBalanceEqualDistantTargets.

```
########################################################
# Geo Proximity Target Sorter Baboon Settings
########################################################

# If enabled, then this baboon can post-process the target list.
# This includes targets from the StaticRoute monkey and/or targets
# from the LocationServer monkey.  Requests that meet the filter
# criteria will have their Target list, flatened (serialized) and
# ordered based on the proximity of the target to the client sending
# the request.  Proximity is determined by looking for a
# x-repro-geolocation="<latitude>,<longitude>" parameter on the Contact
# header of a received request, or the Contact headers of Registration
# requests.  If this parameter is not found, then this processor will
# attempt to determine the public IP address closest to the client or
# target and use the MaxMind Geo IP library to lookup the geo location.
GeoProximityTargetSorting = false

# Specify the full path to the IPv4 Geo City database file
# Note:  A free version of the database can be downloaded from here:
# http://geolite.maxmind.com/download/geoip/database/GeoLiteCity.dat.gz
# For a more accurate database, please see the details here:
# http://www.maxmind.com/app/city
GeoProximityIPv4CityDatabaseFile = GeoLiteCity.dat

# Specify the full path to the IPv6 Geo City database file
# Note:  A free version of the database can be downloaded from here:
# http://geolite.maxmind.com/download/geoip/database/GeoLiteCityv6-beta/
# For a more accurate database, please see the details here:
# http://www.maxmind.com/app/city
# Leave blank to disable V6 lookups.  Saves memory (if not required).
#GeoProximityIPv6CityDatabaseFile = GeoLiteCityv6.dat
GeoProximityIPv6CityDatabaseFile =

# This setting specifies a PCRE compliant regular expression to attempt
# to match against the request URI of inbound requests.  Any requests
# matching this expression, will have their targets sorted as described
# above.  Leave blank to match all requests.
GeoProximityRequestUriFilter = ^sip:mediaserver.*@mydomain.com$

# The distance (in Kilometers) to use for proximity sorting, when the
# Geo Location of a target cannot be determined.
GeoProximityDefaultDistance = 0

# If enabled, then targets that are determined to be of equal distance
# from the client, will be placed in a random order.
LoadBalanceEqualDistantTargets = true
```

Repro High Availability

Active Registration Replication

Repro keeps an in-memory database of all active registrations. This database is updated when a new SIP Registration message arrives that modifies a particular registration (ie. new registration, registration update, or registration removal). When repro is started it attempts to get a complete copy of the in-memory registration database from the configured reg-sync peer. Once this is obtained it receives updates in real-time to ensure the two instances are synchronized. This design supports a setup where there are two proxy servers live at any one time. Clients could then use DNS SRV records to load balance registrations and routing between the two proxies.

Note: resip/repro supports RFC5766 which has a mode that allows clients to request that a proxy only routes messaging down an existing client initiated TCP connection. This type of registration cannot be meaningfully replicated to another proxy, since it will not contain the socket connection. Clients that support this RFC, are required to register with multiple proxies for fault tolerance. For these reasons – repro will not attempt to replicate any registrations that are using RFC5766 (ie. tagged as "use existing connection only").

Initial Synchronization

If configured for reg-sync, when repro first starts, it will attempt to connect to the RPC port on the paired node to obtain the initial list of currently active registrations. If it is unable to connect to the paired node, then it will assume the node is offline and that no registration information is available. The registration sync process with move to a disconnected state, where it will periodically attempt to connect to the paired node. Once a connection succeeds it will retrieve the entire list of active registrations on the pair node, and then move to the Synchronized state where push synchronization will be used. If we are doing an initial synchronization with active registrations in our local store, then conflicts are resolved by looking at the LastUpdated time of each contact in order to know which record has the latest information.

Real-time Synchronization

Once a complete copy of the registration database is obtained, we move to the Synchronized state. In this state the RPC server will send updates to the client whenever they occur in its local store. This way we can be assured that the instances are synchronized in real-time.

Transport Mechanism

Repro will form a TCP connection to its peer node (server). Once connected the client will send an InitialSync xml rpc request in order to retrieve a snapshot of all existing registrations, following this all communication will be one way from the server to the client. The initial and real-time registration update messaging will be sent from the server to the client as XML formatted stream data.

Data to Synchronize

The following data is stored in the InMemoryRegistrationDatabase class and is transported over the wire to the other instance of repro:

- AOR – Address of Record for a particular registration
- Operation – Add/Update or Remove
- Contact List – List of contacts for the AOR
- Contact – URI used to contact this registration instance for the given AOR
- Expires – The time that this registration will expire. In order to avoid requiring that machines have a synchronized clock, this field will be encoded as the number of seconds until the registration will expire.
- Last Update – The time that this registration was last updated. In order to avoid requiring that machines have a synchronized clock, this field will be encoded as the number of seconds since the last update.
- ReceivedFrom – Source IP address and port of the last endpoint to register if "outbound" is used.
- PublicAddress – The address repro calculated as being the the public address of the client
- SipPath – Path header from registration
- Instance – Instance parameter from the contact header
- RegId – Registration Id parameter from the contact header

Sample XML Format for Initial Sync Request

```
<InitialSync>
   <Request>
      <Version>2</Version>
   </Request>
</InitialSync>
```

Sample XML Format for Initial Sync Response

```
<InitialSync>
   <Request>
      <Version>2</Version>
   </Request>
   <Response>
      <Result Code="200">Initial Sync Completed.</Result>
   </Response>
</InitialSync>
```

Sample XML Format for Registration Information

```
<reginfo>
        <aor>sip:user@sipdomain.com</aor>
        <contactinfo>
                <contacturi>sip:user@192.168.1.3:11480</contacturi>
                <expires>1320</expires>
                <lastupate>343</lastupdate>
                <receivedfrom>2562B2C223EF</receivedfrom>
                <publicaddress>1258882223AB</publicaddress>
                <sippath></sippath>
                <instance></instance>
                <regid></regid>
        </contactinfo>
        <contactinfo>
        …
        </contactinfo>
</reginfo>
```

Note: empty XML tags, such as “instance”, can be removed and are only shown here for illustrative purposes.

Configuration Synchronization

The above mechanism synchronized live registration information only. Implementors must provide a means by which to synchronize the configuration data, in order to ensure that both instances of the proxy are operating with the same configuration.

Configuration File

Many settings are contained in the repro text based configuration file. This file is read in a startup and the settings are cached in memory. In order to apply changes to this file, you must either completely restart repro, or you can perform a soft reset that leaves the in memory registration database in tact via the web interface settings page, or the reprocmd restart command. Any appropriate file synchronization tool can be used to ensure the latest configuration settings exist on both repro instances. In cases where IP address appear the file that are required to be different per instance, appropriate tools need to be put into place to make these instance specific adjustments to the configuration files.

Configuration Database – BerkeleyDb

BerkeleyDb uses a simple file based backing store for its database tables. These files can be copied from one machine to another to ensure they are always in sync. Most of the tables are read in when repro starts and are cached in memory. If configuration changes are made at runtime, then they must either be manually duplicated on each instance or a tool must be written to use the HTTP interface from a scripted interface that can apply a change to multiple instances. See the Scripting Database Updates section below for more details. Note: there is no way to properly use a BerkeleyDb database for the Message Silo feature inconjunction with load balancing across instances. The following files comprise the BerekelyDb storage on the filesystem:

- repro_user.db – Users table
- repro_config.db – Repro config tables (stores domains configuration)
- repro_acl.db – Access Control List table
- repro_filter.db – Request filter table
- repro_silo.db – message silo table
- repro_silo_idx1.db – secondary index on message silo table
- repro_route.db – Static route table
- repro_staticreg.db – Static registration table

Note: The BerkeleyDb storage for accounting message queues do not need to be replicated between nodes. Each node should have its own consumer acting on its own event queues.

Configuration Database – MySQL

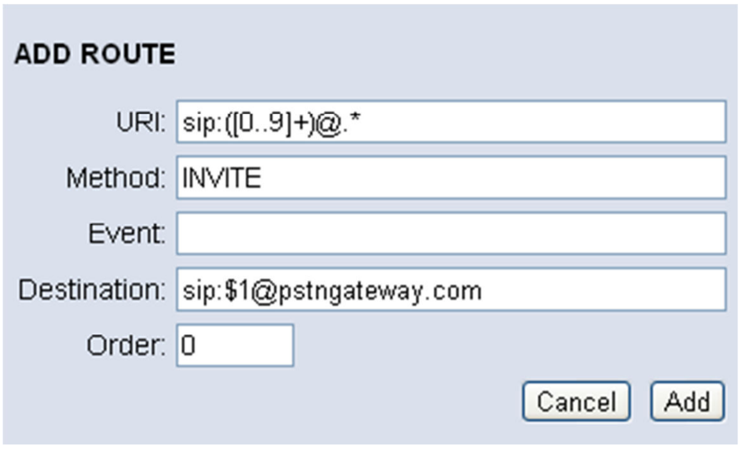
Any relevant MySQL table sycnronization mechanism can be used to ensure that the databases for each instance are kept synchronized. Most of the tables are read in when repro starts and are cached in memory. If configuration changes are made at runtime, then they must either be manually duplicated on each instance or a tool must be written to use the HTTP interface from a scripted interface that can apply a change to multiple instances. See the Scripting Database Updates section below for more details.

WARNING: repro instances cannot share MySQL database instances. Each instance requires its own copy of the tables. However the tables accessed at runtime: Users and MessageSilo should be shared; this is to ensure that the Message Silo for a particular user can be found not matter what instance the registration request arrives on.

Scripting Database Updates

Most of the database tables are read in when repro starts and are cached in memory. If configuration changes are made at runtime, then they must either be manually duplicated on each instance or a tool must be written to use the HTTP interface from a scripted interface that can apply a change to multiple instances. For example to add a new route you just need to examine that URL line when this is done manually on the web interface and duplicate this call from a script.

To add the following route from a script:



ADD ROUTE

URI:	sip:([0..9]+)@.*
Method:	INVITE
Event:	
Destination:	sip:$1@pstngateway.com
Order:	0

Use the following URL.

http://127.0.0.1:5090/addRoute.html?routeUri= sip%3A%28%5B0-9%5D%2B%29%40%5C.*&routeMethod=INVITE&routeEvent=&routeDestination=sip%3A%241%40pstngateway.com&routeOrder=0&routeAdd=Add

Repro Extensibility Overview

Chain Processors

Repro is built on the *chain of responsibility* design pattern. Repro passes SIP messages and events through chains of processors. Each processor has a process() method, which examines the SIP message or event, performs some actions and then returns one of the following to control the chain flow:

- Continue – move to the next Processor in the chain
- WaitingForEvent – chain processing is paused, waiting for an asynchronous action to finish (ie. a database lookup to complete)
- SkipThisChain/SkillAllChains – used when further chain processing is not needed (ie. this processor may be rejecting the request)

Asynchronous Chain Processors

In order to aid the development of processors that need to use blocking API's asynchronously (ie. perform a database read or write), an AsyncProcessor class is provided. The request/event is delivered to this Processor, like any other via the virtual method:

virtual processor_action_t process(RequestContext &)

When a blocking task is required, it can be queued up for the thread pool of workers, that are implemented in the constructor provided in asyncDispatcher, by constructing a new AsyncProcessorMessage and calling mAsyncDispatcher->post.

AsyncProcessorMessage implementations should contain any data needed by the threads and contain any members required to hold the blocking functions return data or results.

When a worker thread picks up the message it will call the AsyncProcessor virtual method:

virtual void asyncProcess(AsyncProcessorMessage* msg)

The implementation of this method should perform the actual blocking function calls - ie. database access call. If there is any data to return it should be set in the provided AsyncProcessorMessage before returning from asyncProcess.

The dispatcher will then queue this message up to be sent back to the AsyncProcessor, via the virtual method:

virtual processor_action_t process(RequestContext &)

This virtual method will need to examine the event type to see if the event is a new request or an asynchronous result message (AsyncProcessorMessage).

Example:

```
class MyAsyncProcessorAsyncMessage : public AsyncProcessorMessage
{
public:
   MyAsyncProcessorAsyncMessage(AsyncProcessor& proc,
                        const resip::Data& tid,
                        resip::TransactionUser* passedtu):
      AsyncProcessorMessage(proc,tid,passedtu)  { }

   virtual EncodeStream& encode(EncodeStream& strm) const
   {
      strm << "MyAsyncProcessorAsyncMessage(tid="<<mTid<<")"; return strm;
   }

   Data mDataRequiredToCallBlockingFunction;
   Data mDataReturnedFromBlockingFunction;
};

class MyAsyncProcessor : public AsyncProcessor
{
   public:
      MyAsyncProcessor(ProxyConfig& config, Dispatcher* asyncDispatcher) :
         AsyncProcessor("MyAsyncProcessor", asyncDispatcher) {}
      ~MyAsyncProcessor() {}

      // Processor virtual method
      virtual processor_action_t process(RequestContext &rc)
      {
         Message *message = rc.getCurrentEvent();

         MyAsyncProcessorAsyncMessage *async =
               dynamic_cast<MyAsyncProcessorAsyncMessage*>(message);
         if (async)
         {
            // Async Function is complete - do something with results and continue
            InfoLog(<< "Async function is complete, results are: "
                    << async->mDataReturnedFromBlockingFunction);
            return Continue;
         }
         else
         {
            // Control enters here when request arrives and is passed through process chain
            // Dispatch async request to worker thread pool
            MyAsyncProcessorAsyncMessage* async = new MyAsyncProcessorAsyncMessage(
                             *this, rc.getTransactionId(), &rc.getProxy());
            async->mDataRequiredToCallBlockingFunction = "foo";
            mAsyncDispatcher->post(std::auto_ptr<ApplicationMessage>(async));
            return WaitingForEvent;
         }
      }

      // Virtual method called from WorkerThreads - return true to queue to stack when complete,
      // false when no response is required
      virtual bool asyncProcess(AsyncProcessorMessage* msg)
      {
         MyAsyncProcessorAsyncMessage* async = dynamic_cast<MyAsyncProcessorAsyncMessage*>(msg);
         if(async)
         {
            // Running inside a worker thread here, do blocking work here
            // set any results in MyAsyncProcessorAsyncMessage and return control to Dispatcher
            // that will queue this message back to this processor.
            async->mDataReturnedFromBlockingFunction = "bar";
         }
      }

   private:
};
```

State Storage

A RequestContext C++ object is created for each new SIP request received. It is responsible for maintaining proxy state for the transaction. It also passes any inbound SIP requests through the request Processor chain.

A ResponseContext C++ object is a member of RequestContext and is used to track targets for the request as it traverses the request processor chain. It also collects responses from the various targets and forwards the best response back to the requestor (UAC). The ResponseContext handles cancelling of forked legs when one leg returns a 200 Success to a forked request.

There is generic storage available for custom Processors at three different scope levels via a KeyValueStore class:

- Global Proxy Scope - Proxy::getKeyValueStore
- Request Scope - RequestContext::getKeyValueStore
- Target Scope - Target::getKeyValueStore

Before this storage can be used you must statically allocate a storage key. See mFromTrustedNodeKey use in the IsTrustedNode processor class for an example.

Specific Chains of Responsibility

Monkeys

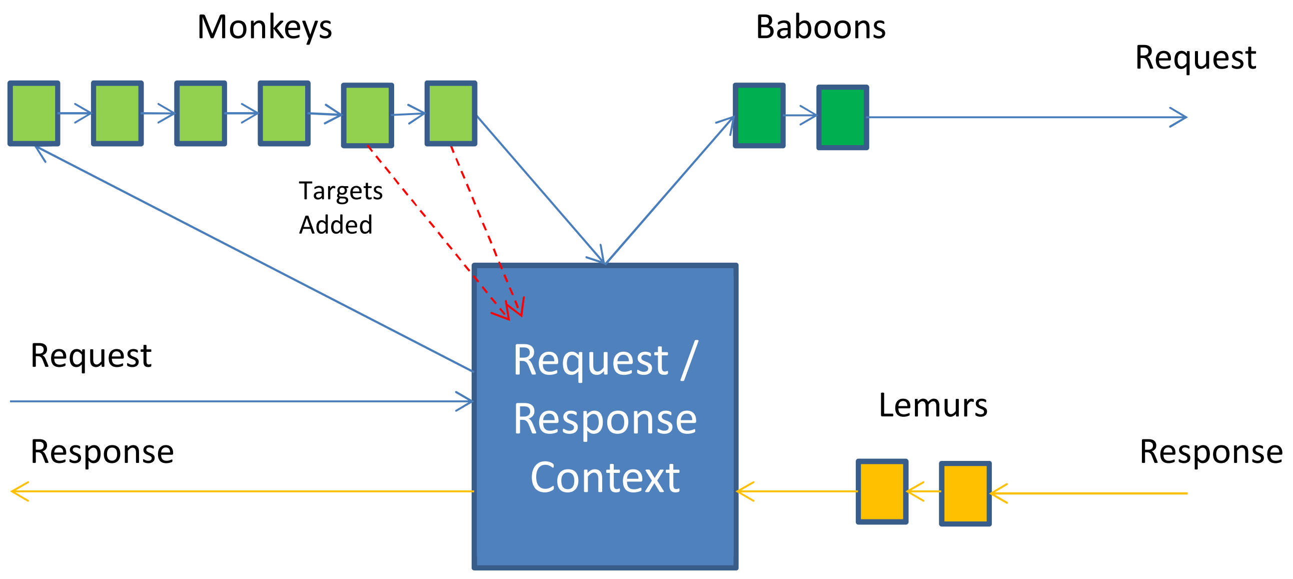
Chain of processors that handle inbound SIP requests. Monkeys may send SIP responses or add proxy targets to the Request Context, to route the request on to the next node.

Baboons

Chain of processors that can modify SIP requests that are generated post target selection. These are the requests that will be sent on outbound sockets to the proxied targets.

Lemurs

Chain of processors that handle inbound SIP responses and the final response which is sent back to the original requestor (UAC).



Existing Monkeys

Default monkey chain:

StrictRouteFixup ➡ IsTrustedNode ➡ DigestAuthenticator ➡ AmIResponsible ➡ RequestFilter ➡ StaticRoute ➡ LocationServer ➡ MessageSilo

StrictRouteFixup - Follows route headers(skipping rest of chain) after route header preprocessing has occurred

IsTrustedNode - Checks senders IP address, port, transport (or TLS domain name) for a match in the ACL list, and tags the request as trusted or not. Trusted requests are no digest challenged.

DigestAuthenticator - Challenges requests as necessary, uses the same database as the registrar to locate secrets/passwords.

AmIResponsible - Skips the rest of the current chain if Request is not in one of this repro instances domains.

RequestFilter – (disabled by default) Used to analyze the incoming request and reject it if particular conditions are met.

StaticRoute - Applies configured routes from the RouteStore, which uses the abstract database

LocationServer - Fetches locations from the RegistrationPersistenceManager instance (shared with the registrar) to proxy requests to.

MessageSilo – (disabled by default) Used to store MESSAGE requests for users that are offline. Messages are delivered when an offline user registers with the proxy.

Existing Baboons

Default baboon chain: GeoProximityTargetSorter ➡ QValueTargetHandler ➡ SimpleTargetHandler

GeoProximityTargetSorter – (disabled by default) Able to order multiple targets collected from the monkey chain based on geo proximity to the requestor. Uses the MaxMind GeoIP libraries to lookup geo location based on IP addresses.

QValueTargetHandler – Examines all of the targets for a request, and forks them according to the forking configuration settings.

SimpleTargetHandler – Used when Qvalue processing is not enabled – parallel forks to all targets.

Existing Lemurs

Default Lemur Chain: OutboundTargetHandler ➡ RecursiveRedirectHandler

OutboundTargetHandler – Used to try another flow to a target, if we get flow based error responses when sending a request.

RecursiveRedirectHandler – Handles 302 responses by internally re-issuing the request to the redirect target(s), instead of proxying them back to the requestor.

Adding Custom Processors

Extending Repro by adding custom processors to the chain is as easy as overriding one of the ReproRunner class virtual methods:

```
virtual void addProcessor(repro::ProcessorChain& chain, std::auto_ptr<repro::Processor> processor);
virtual void makeRequestProcessorChain(repro::ProcessorChain& chain);
virtual void makeResponseProcessorChain(repro::ProcessorChain& chain);
virtual void makeTargetProcessorChain(repro::ProcessorChain& chain);
```

Override the makeXXXProcessorChain methods to add processors to the beginning or end of any chain, or override the addProcessor method, and you can examine the name of the processor being added and add your own process either before or after the correct processor.

WARNING: Be careful when checking for names of optional processors. Depending on the configuration some processors may not be enabled.

Create an instance of your overridden ReproRunner class and call run to start everything up.

Example:

```
class MyCustomProcessor : public Processor
 {
    public:
       MyCustomProcessor(ProxyConfig& config) : Processor("MyCustomProcessor") {}
       virtual ~MyCustomProcessor() {}

       virtual processor_action_t process(RequestContext &context)
       {
          DebugLog(<< "Monkey handling request: " << *this << "; reqcontext = " << context);

          ...Do something interesting here....

          return Processor::Continue;
       }
    }
 };

 class MyReproRunner : public ReproRunner
 {
 public:
    MyReproRunner() {}
    virtual ~MyReproRunner() {}

 protected:
    virtual void addProcessor(repro::ProcessorChain& chain,
                              std::auto_ptr<repro::Processor> processor)
    {
       if(processor->getName() == "LocationServer")
       {
          // Add MyCustomProcessor before LocationServer
          addProcessor(chain, std::auto_ptr<Processor>(new MyCustomProcessor(*mProxyConfig)));
       }
       ReproRunner::addProcessor(chain, processor);  // call base class implementation
    }
 };
```

repro Target Routing Detailed Overview

Target Types

As a request traverses the Request Processors (Monkey's) it collects targets to route the request to. There are three types of targets used in repro for routing:

Target (Simple)

- States: Candidate, Started, Cancelled, Terminated, NonExistent
- Stores: PriorityMetric(0), ShouldAutoProcess, State, Via, ContactInstanceRecord
- Processors that add Targets of this type: RecursiveRedirect, SimpleStaticRoute, StaticRoute, AmIResponsible (when top most route has flow token, or request is for another domain), StrictRouteFixup (when there are Route headers)

QValueTarget

- Is a Target (overrides the Target class)
- Copies Qvalue from contact to PriorityMetric
- Processors that add Targets of this type: LocationServer (for registrations not using outbound), RecursiveRedirect

OutboundTarget

- Is a QValueTarget (overrides the QValueTarget class)
- Additionally Stores: Aor and ContactList (ie. contacts with same instance value)
- Processors that add Targets of this type: LocationServer (for registrations using outbound), OutboundTargetHandler (when flow fails to try next reg-id)

Response Context Target Handling

Target Related Members

Transaction Maps

The following maps store targets according to their state. They are indexed by transaction id (TID).

```
typedef std::map<resip::Data,repro::Target*> TransactionMap;

//Targets with status Candidate.
TransactionMap mCandidateTransactionMap;

//Targets with status Trying, Proceeding, or WaitingToCancel.
TransactionMap mActiveTransactionMap;

//Targets with status Terminated.
TransactionMap mTerminatedTransactionMap;
```

TransactionQueueCollection

The transaction queue collection is an ordered list of target batches to try to route to. This collection only stores the TID, however the Target can be found by using the TransactionMaps. This collection is a list of lists, with each list containing a list of TIDs in the batch. Ie:

- Batch1: TID1
- Batch2: TID2, TID3, TID4
- Batch3: TID5

```
std::list<std::list<resip::Data> > mTransactionQueueCollection;
resip::ContactList mTargetList;
```

addTarget

- If URI only is passed in adds a single simple Target as a new batch to the Target List (may add to first batch if bool is set)
- If Target* is passed in adds a target of the passed in type as a new batch to the Target List (may add to first batch if bool is set)
- optional flag to begin immediately that sends the client transaction immediate - no one uses this
- optional flag to add to first batch
- adds Target to CandidateTransactionMap
- adds TID to TransactionQueueCollection

addTargetBatch

- Adds a list of Targets grouped in a batch (may add to first batch if bool is set)
- optional highPriority parameter that adds batch to first in batch list instead of last - no one uses this
- adds Targets to CandidateTransactionMap
- adds TID batch to TransactionQueueCollection as a group of TIDs

beginClientTransactions()

- Iterates through CandidateTransactionMap, and begins them all at once
- checks for duplicates
- adds Target URI to mTargetList for future duplicate detection
- moves from CandidateTransactionMap to ActiveTransactionMap
- sends request

beginClientTransaction(tid)

- Checks for duplicates
- adds Target URI to mTargetList for future duplicate detection
- moves from CandidateTransactionMap to ActiveTransactionMap
- sends request

cancelActiveClientTransactions

- Only called if for INVITES
- Iterates through ActiveTransactionMap and issues a cancel (if INVITE) for each member
- Note: does not move Target to TerminatedTransactionMap

cancelAllClientTransactions

- Only called if for INVITES
- Iterates through ActiveTransactionMap and issues a cancel (if INVITE) for each member
- Note: does not move active Targets to TerminatedTransactionMap
- Iterates through CandidateTransactionMap and moves each Target to TerminatedTransactionMap

clearCandidateTransactions

- Iterates through CandidateTransactionMap and moves each Target to TerminatedTransactionMap

cancelClientTransaction(tid)

- Sends a cancel if active and INVITE
- Note: does not move active Targets to TerminatedTransactionMap
- If not active, then moves to TerminatedTransactionMap

terminateClientTransaction(tid)

- Moves all active and candidate Targets to the TerminateTransactionMap

removeClientTransaction(tid)

- Removes the Target from whatever map it is in

isDuplicate

- Checks a TargetList of contacts that have been activated before to ensure this target hasn't already been tried.

beginClientTransaction

- Forms the new message to route to the target and passes to the stack for sending
- Target status is changed from Candidate to Started

Processor Target Control

StaticRoute Monkey

- Get's list of matching routes from RouteStore
- a simple Target is created for each matching route
- if ParallelForkStaticRoutes is disabled, then each target is added as its own single entry batch and serial forking will occur
- if ParallelForkStaticRoutes is enabled, then each target is added to the same batch of targets

Location Server Monkey

- registration information is retrieved
- for each registered contact
    - if the contact is an outbound contact (ie. contains and instance and regId), then add to an outbound batch (OutboundTarget) for each instance GUID, ordered by last update time
    - if the contact is not outbound, then a QValue target is created and added to a QValue batch. The outbound and QValue target batches are merged, then ordered by PriorityMetric

GeoProximityTargetSorter Baboon

- If there is less than 2 targets, then there is nothing to do, so just continue to next processor
- Checks if request URI passes regex filter from configuration
- Flattens the Target list to a single batch – all targets are treated equally regardless of batch location
- Calculates the geo distance between the sender and each of the Targets
- That list order is first shuffled, in order to randomize targets that are at equivalent geographic distance
- Target list is then sorted based on the geographic distance – closer targets are ordered first
- Previous Target collection is then cleared and a newly built Target list is added in sorted order, with each Target in its own batch to facilitate serial forking to Targets.

QValue Target Handler Baboon

- looks through ResponseContext::mTransactionQueueCollection for a batch of QValue targets
    - note: OutboundTargets are also QValue Targets, so they are processed here too
- if any are active then bail
- calls fillNextTargetGroup and passed batch of QValue TIDS (queue)
    - finds the first Candidate (unstarted) Target in the queue and records the priority
    - if FULL_SEQUENTIAL then just return this Target
    - if EQUAL_Q_PARALLEL then return this Target and all subsequent targets in batch with equal q value
    - if FULL_PARALLEL then return this Target and all remaining Targets in batch
- bail if no targets were found
- call beginClientTransaction on each TID returned from fillNextTargetGroup
- if CancelBetweenForkGroups then schedule cancellations for each TID
- if we started some targets in this batch, and WaitForTerminate if false, then schedule next batch to dispatch
- remove any TIDs that have terminated (ie. Duplicates found during dispatch) from current batch
    - if entire batch has terminated, then remove from collection, and move on to next batch
- if we are processing a ForkControlMessage
    - If Cancel TIDs and there is more Candidates, then issue cancels (if there are no more Candidates to try, don't cancel)

    - If there are begin TIDs, then start next batch, and scheudle their cancels (if required)
- only move on to next processor (ie. SimpleTargetHandler) if there are no more active Q-Value targets

Simple Target Handler Baboon

- Runs only after all QValue and OutboundTargets have been processed
- if no targets are active
    - iterates through ResponseContext::mTransactionQueueCollection and starts the first TID via beginClientTransaction
    - continues until we find one that sticks (ie. is not a duplicate), then skip all chains

OutboundTargetHandler Lemur

- If we are receiving a response to an OutboundTarget and it is a locally generated 430/410 (Flow Dead), 408 or 503
    - then remove bad contact from registration table, and try routing to next reg-id

RecursiveRedirect Lemur

- If response is a 3xx response with Contact headers, then create a QValueTarget for each contact, order the list by q-value and add the batch to the ResponseContext