hekad Documentation

Release 0.4.1

Mozilla

November 26, 2013

Contents

The hekad daemon is the core component of the heka project, which handles routing messages, generating metrics, aggregating statsd-type messages, running plugins on the messages, and sending messages to the configured destinations.

See Also:

heka project

Contents:

CHAPTER 1

Installing

1.1 Binaries

hekad releases are available on the Github project releases page. Binaries are available for Linux and OSX, with packages for Debian and RPM based distributions.

1.2 From Source

hekad requires a Go work environment to be setup for the binary to be built; this task is automated by the build process. The build script will override the Go environment for the shell window it is executed in. This creates an isolated environment that is intended specifically for building and developing Heka. The build script should be be run every time a new shell is opened for Heka development to ensure the correct dependencies are found and being used. To create a working *hekad* binary for your platform you'll need to install some prerequisites. Many of these are standard on modern Unix distributions and all are available for installation on Windows systems.

Prerequisites (all systems):

- CMake 2.8.7 or greater http://www.cmake.org/cmake/resources/software.html
- Git http://git-scm.com/download
- Go 1.1 or greater (1.1.1 recommended) http://code.google.com/p/go/downloads/list
- Mercurial http://mercurial.selenic.com/downloads/
- Protobuf 2.3 or greater (optional - only needed if message.proto is modified) http://code.google.com/p/protobuf/downloads/list
- Sphinx (optional - used to generate the documentation) http://sphinx-doc.org/

Prerequisites (Unix):

- make
- gcc
- patch
- dpkg (optional)
- rpmbuild (optional)

- packagemaker (optional)

Prerequisites (Windows):

- MinGW http://sourceforge.net/projects/tdm-gcc/

1.2.1 Build Instructions

1. Check out the **'heka'_** repository:

```
git clone https://github.com/mozilla-services/heka
```

2. Run *build* in the heka directory

```
cd heka
source build.sh # Unix (this file must be sourced to properly setup the environment)
build.bat  # Windows
```

You will now have a *hekad* binary in the *build/heka/bin* directory.

3. (Optional) Run the tests to ensure a functioning *hekad*.

```
ctest             # All, see note
# Or use the makefile target
make test         # Unix
mingw32-make test # Windows
```

Note: In addition to the standard test build target, ctest can be called directly providing much greater control over the tests being run and the generated output (see ctest –help). i.e., 'ctest -R pi' will only run the pipeline unit test.

1.2.2 Clean Targets

- clean-heka - Use this target any time you change branches or pull from the Heka repository, it will ensure the Go workspace is in sync with the repository tree.
- clean - You will never want to use this target (it is autogenerated by cmake), it will cause all external dependencies to be re-fetched and re-built. The best way to 'clean-all' is to delete the build directory and re-run the build.(sh|bat) script.

1.2.3 Build Options

There are two build customization options that can be specified during the cmake generation process.

- INCLUDE_MOZSVC (bool) Include the Mozilla services plugins (default Unix: true, Windows: false).
- BENCHMARK (bool) Enable the benchmark tests (default false)

For example: to enable the benchmark tests in addition to the standard unit tests type 'cmake -DBENCHMARK=true ..' in the build directory.

1.3 Building *hekad* with External Plugins

It is possible to extend *hekad* by writing input, decoder, filter, or output plugins in Go (see *Extending Heka*). Because Go only supports static linking of Go code, your plugins must be included with and registered into

Heka at compile time. The build process supports this through the use of an optional cmake file *{heka root}/cmake/plugin_loader.cmake*. A cmake function has been provided *add_external_plugin* taking the repository type (git, hg, or svn), repository URL, the repository tag to fetch, and an optional list of sub-packages to be initialized.

```
add_external_plugin(git https://github.com/mozilla-services/heka-mozsvc-plugins dev)
add_external_plugin(git https://github.com/example/path dev util filepath)
```

The preceeding entry clones the *heka-mozsvc-plugins* git repository into the Go work environment, checks out the dev branch, and imports the package into *hekad* when *make* is run. By adding an init() function in your package you can make calls into *pipeline.RegisterPlugin* to register your plugins with Heka's configuration system.

1.4 Creating Packages

Installing packages on a system is generally the easiest way to deploy *hekad*. These packages can be easily created after following the above *From Source* directions:

1. Run *cpack* to build the appropriate package(s) for the current system:

```
cpack                # All
# Or use the makefile target
make package         # Unix
mingw32-make package # Windows
```

The packages will be created in the build directory.

Note: You will need *rpmbuild* installed to build the rpms.

See Also:

Setting up an rpm-build environment

CHAPTER 2

Configuring hekad

A hekad configuration file specifies what inputs, decoders, filters, and outputs will be loaded. The configuration file is in TOML format. TOML looks is very similar to INI configuration formats, but with slightly more rich data structures and nesting support.

The config file is broken into sections, with each section representing a single instance of a plugin. The section name specifies the name of the plugin, and the "type" parameter specifies the plugin type; this must match one of the types registered via the *pipeline.RegisterPlugin* function. For example, the following section describes a plugin named "tcp:5565", an instance of Heka's plugin type "TcpInput":

```
[tcp:5565]
type = "TcpInput"
parser_type = "message.proto"
decoder = "ProtobufDecoder"
address = ":5565"
```

If you choose a plugin name that also happens to be a plugin type name, then you can omit the "type" parameter from the section and the specified name will be used as the type. Thus, the following section describes a plugin named "TcpInput", also of type "TcpInput":

```
[TcpInput]
address = ":5566"
parser_type = "message.proto"
decoder = "ProtobufDecoder"
```

Note that it's fine to have more than one instance of the same plugin type, as long as their configurations don't interfere with each other.

Any values other than "type" in a section, such as "address" in the above examples, will be passed through to the plugin for internal configuration (see *Plugin Configuration*).

A ProtobufDecoder will be automatically setup if not specified explicitly in the configuration file.

If a plugin fails to load during startup, hekad will exit at startup. When hekad is running, if a plugin should fail (due to connection loss, inability to write a file, etc.) then hekad will either shut down or restart the plugin if the plugin supports restarting. When a plugin is restarting, hekad will likely stop accepting messages until the plugin resumes operation (this applies only to filters/output plugins).

Plugins specify that they support restarting by implementing the Restarting interface (see *restarting_plugins*). Plugins supporting Restarting can have *their restarting behavior configured*.

An internal diagnostic runner runs every 30 seconds to sweep the packs used for messages so that possible bugs in heka plugins can be reported and pinned down to a likely plugin(s) that failed to properly recycle the pack.

2.1 Global configuration options

You can optionally declare a *[hekad]* section in your configuration file to configure some global options for the heka daemon.

Parameters:

- **cpuprof (string *output_file*):** Turn on CPU profiling of hekad; output is logged to the *output_file*.
- **max_message_loops (uint):** The maximum number of times a message can be re-injected into the system. This is used to prevent infinite message loops from filter to filter; the default is 4.
- **max_process_inject (uint):** The maximum number of messages that a sandbox filter's ProcessMessage function can inject in a single call; the default is 1.
- **max_process_duration (uint64):** The maximum number of nanoseconds that a sandbox filter's ProcessMessage function can consume in a single call before being terminated; the default is 100000.
- **max_timer_inject (uint):** The maximum number of messages that a sandbox filter's TimerEvent function can inject in a single call; the default is 10.
- **max_pack_idle (string):** A time duration string (e.x. "2s", "2m", "2h") indicating how long a message pack can be 'idle' before its considered leaked by heka. If too many packs leak from a bug in a filter or output then heka will eventually halt. This setting indicates when that is considered to have occurred.
- **maxprocs (int):** Enable multi-core usage; the default is 1 core. More cores will generally increase message throughput. Best performance is usually attained by setting this to 2 x (number of cores). This assumes each core is hyper-threaded.
- **memprof (string *output_file*):** Enable memory profiling; output is logged to the *output_file*.
- **poolsize (int):** Specify the pool size of maximum messages that can exist; default is 100 which is usually sufficient and of optimal performance.
- **decoder_poolsize (int):** Specify the number of decoder sets to spin up for use converting input data to Heka's Message objects. Default is 4, optimal value is variable, depending on number of total running plugins, number of expected concurrent connections, amount of expected traffic, and number of available cores on the host.
- **plugin_chansize (int):** Specify the buffer size for the input channel for the various Heka plugins. Defaults to 50, which is usually sufficient and of optimal performance.
- **base_dir (string):** Base working directory Heka will use for persistent storage through process and server restarts. Defaults to */var/cache/hekad* (or *c:\varcache\hekad* on windows).

2.2 Example hekad.toml file

```
[hekad]
cpuprof = "/var/log/hekad/cpuprofile.log"
decoder_poolsize = 10
max_message_loops = 4
max_process_inject = 10
max_timer_inject  = 10
maxprocs = 10
```

```
memprof = "/var/log/hekad/memprof.log"
plugin_chansize = 10
poolsize = 100

# Listens for Heka messages on TCP port 5565.
[TcpInput]
address = ":5565"
parser_type = "message.proto"
decoder = "ProtobufDecoder"

# Writes output from `CounterFilter`, `lua_sandbox`, and Heka's internal
# reports to stdout.
[debug]
type = "LogOutput"
message_matcher = "Type == 'heka.counter-output' || Type == 'heka.all-report' || Type == 'heka.sandbo

# Counts throughput of messages sent from a Heka load testing tool.
[CounterFilter]
message_matcher = "Type == 'hekabench' && EnvVersion == '0.8'"
output_timer = 1

# Defines a sandboxed filter that will be written in Lua.
[lua_sandbox]
type = "SandboxFilter"
message_matcher = "Type == 'hekabench' && EnvVersion == '0.8'"
output_timer = 1
script_type = "lua"
preserve_data = true
filename = "lua/sandbox.lua"
memory_limit = 32767
instruction_limit = 1000
output_limit = 1024
```

2.3 Common Roles

- **Agent** - Single default filter that passes all messages directly to another *hekad* daemon on a separate machine configured as an Router.
- **Aggregator** - Runs filters that can roll-up statistics (similar to statsd), and handles aggregating similar messages before saving them to a back-end directly or possibly forwarding them to a *hekad* router.
- **Router** - Collects input messages from multiple sources (including other *hekad* daemons acting as Agents), rolls up stats, and routes messages to appropriate back-ends.

2.4 Command Line Options

-version Output the version number, then exit.

-config ***config_file*** Specify the configuration file to use; the default is /etc/hekad.toml. (See hekad.config(5).)

2.5 Configuring Restarting Behavior

Plugins that support being restarted have a set of options that govern how the restart is handled. If preferred, the plugin can be configured to not restart at which point hekad will exit, or it could be restarted only 100 times, or restart attempts can proceed forever.

Adding the restarting configuration is done by adding a config section to the plugins' config called *retries*. A small amount of jitter will be added to the delay between restart attempts.

Parameters:

- **max_jitter (string):** The longest jitter duration to add to the delay between restarts. Jitter up to 500ms by default is added to every delay to ensure more even restart attempts over time.
- **max_delay (string):** The longest delay between attempts to restart the plugin. Defaults to 30s (30 seconds).
- **delay (string):** The starting delay between restart attempts. This value will be the initial starting delay for the exponential back-off, and capped to be no larger than the *max_delay*. Defaults to 250ms.
- **max_retries (int):** Maximum amount of times to attempt restarting the plugin before giving up and shutting down hekad. Use 0 for no retry attempt, and -1 to continue trying forever (note that this will cause hekad to halt possibly forever if the plugin cannot be restarted).

Example (UdpInput does not actually support nor need restarting, illustrative purposes only):

```
[UdpInput]
address = "127.0.0.1:4880"
parser_type = "message.proto"
decoder = "ProtobufDecoder"

[UdpInput.retries]
max_delay = 30s
delay = 250ms
max_retries = 5
```

2.6 Inputs

2.6.1 AMQPInput

Connects to a remote AMQP broker (RabbitMQ) and retrieves messages from the specified queue. If the message is serialized by hekad's AMQPOutput then the message will be de-serialized, otherwise the message will be run through the specified PayloadRegexDecoder's. As AMQP is dynamically programmable, the broker topology needs to be specified.

Parameters:

- **URL (string):** An AMQP connection string formatted per the RabbitMQ URI Spec.
- **Exchange (string):** AMQP exchange name
- **ExchangeType (string):** AMQP exchange type (*fanout*, *direct*, *topic*, or *headers*).
- **ExchangeDurability (bool):** Whether the exchange should be configured as a durable exchange. Defaults to non-durable.
- **ExchangeAutoDelete (bool):** Whether the exchange is deleted when all queues have finished and there is no publishing. Defaults to auto-delete.

- **RoutingKey (string):** The message routing key used to bind the queue to the exchange. Defaults to empty string.
- **PrefetchCount (int):** How many messages to fetch at once before message acks are sent. See RabbitMQ performance measurements for help in tuning this number. Defaults to 2.
- **Queue (string):** Name of the queue to consume from, an empty string will have the broker generate a name for the queue. Defaults to empty string.
- **QueueDurability (bool):** Whether the queue is durable or not. Defaults to non-durable.
- **QueueExclusive (bool):** Whether the queue is exclusive (only one consumer allowed) or not. Defaults to non-exclusive.
- **QueueAutoDelete (bool):** Whether the queue is deleted when the last consumer un-subscribes. Defaults to auto-delete.
- **Decoder (string):** Decoder name used to transform a raw message body into a structured hekad message. Must be a decoder appropriate for the messages that come in from the exchange.

Since many of these parameters have sane defaults, a minimal configuration to consume serialized messages would look like:

```
[AMQPInput]
url = "amqp://guest:guest@rabbitmq/"
exchange = "testout"
exchangeType = "fanout"
```

Or if using a PayloadRegexDecoder to parse OSX syslog messages may look like:

```
[AMQPInput]
url = "amqp://guest:guest@rabbitmq/"
exchange = "testout"
exchangeType = "fanout"
decoder = "logparser"

[logparser]
type = "MultiDecoder"
order = ["logline", "leftovers"]

  [logparser.subs.logline]
  type = "PayloadRegexDecoder"
  MatchRegex = '\w+ \d+ \d+:\d+:\d+ \S+ (?P<Reporter>[^\[]+)\[(?P<Pid>\d+)](?P<Sandbox>[^:]+)?: (?P

    [logparser.subs.logline.MessageFields]
    Type = "amqplogline"
    Hostname = "myhost"
    Reporter = "%Reporter%"
    Remaining = "%Remaining%"
    Logger = "%Logger%"
    Payload = "%Remaining%"

  [leftovers]
  type = "PayloadRegexDecoder"
  MatchRegex = '.*'

    [leftovers.MessageFields]
    Type = "drop"
    Payload = ""
```

2.6.2 UdpInput

Listens on a specific UDP address and port for messages. If the message is signed it is verified against the signer name and specified key version. If the signature is not valid the message is discarded otherwise the signer name is added to the pipeline pack and can be use to accept messages using the message_signer configuration option.

Note: The UDP payload is not restricted to a single message; since the stream parser is being used multiple messages can be sent in a single payload.

Parameters:

- **address (string):** An IP address:port on which this plugin will listen.
- **signer:** Optional TOML subsection. Section name consists of a signer name, underscore, and numeric version of the key.
  - **hmac_key (string):** The hash key used to sign the message.

New in version 0.4.

- **decoder (string):** A decoder must be specified for the message.proto parser (i.e. ProtobufDecoder) but is optional for token and regexp parsers (if no decoder is specified the parsed data is available in the Heka message payload).
- **parser_type (string):**
  - token - splits the stream on a byte delimiter.
  - regexp - splits the stream on a regexp delimiter.
  - message.proto - splits the stream on protobuf message boundaries.
- **delimiter (string): Only used for token or regexp parsers.** Character or regexp delimiter used by the parser (default "\n"). For the regexp delimiter a single capture group can be specified to preserve the delimiter (or part of the delimiter). The capture will be added to the start or end of the message depending on the delimiter_location configuration.
- **delimiter_location (string): Only used for regexp parsers.**
  - start - the regexp delimiter occurs at the start of the message.
  - end - the regexp delimiter occurs at the end of the message (default).

Example:

```
[UdpInput]
address = "127.0.0.1:4880"
parser_type = "message.proto"
decoder = "ProtobufDecoder"

[UdpInput.signer.ops_0]
hmac_key = "4865ey9urgkidls xtb0[7lf9rzcivthkm"
[UdpInput.signer.ops_1]
hmac_key = "xdd908lfcgikauexdi8elogusridaxoalf"

[UdpInput.signer.dev_1]
hmac_key = "haeoufyaiofeugdsnzaogpi.ua,dp.804u"
```

2.6.3 TcpInput

Listens on a specific TCP address and port for messages. If the message is signed it is verified against the signer name and specified key version. If the signature is not valid the message is discarded otherwise the signer name is added to the pipeline pack and can be use to accept messages using the message_signer configuration option.

Parameters:

- **address (string):** An IP address:port on which this plugin will listen.
- **signer:** Optional TOML subsection. Section name consists of a signer name, underscore, and numeric version of the key.
  - **hmac_key (string):** The hash key used to sign the message.

New in version 0.4.

- **decoder (string):** A decoder must be specified for the message.proto parser (i.e. ProtobufDecoder) but is optional for token and regexp parsers (if no decoder is specified the parsed data is available in the Heka message payload).
- **parser_type (string):**
  - token - splits the stream on a byte delimiter.
  - regexp - splits the stream on a regexp delimiter.
  - message.proto - splits the stream on protobuf message boundaries.
- **delimiter (string): Only used for token or regexp parsers.** Character or regexp delimiter used by the parser (default "\n"). For the regexp delimiter a single capture group can be specified to preserve the delimiter (or part of the delimiter). The capture will be added to the start or end of the message depending on the delimiter_location configuration.
- **delimiter_location (string): Only used for regexp parsers.**
  - start - the regexp delimiter occurs at the start of the message.
  - end - the regexp delimiter occurs at the end of the message (default).

Example:

```
[TcpInput]
address = ":5565"
parser_type = "message.proto"
decoder = "ProtobufDecoder"

[TcpInput.signer.ops_0]
hmac_key = "4865ey9urgkidls xtb0[7lf9rzcivthkm"
[TcpInput.signer.ops_1]
hmac_key = "xdd908lfcgikauexdi8elogusridaxoalf"

[TcpInput.signer.dev_1]
hmac_key = "haeoufyaiofeugdsnzaogpi.ua,dp.804u"
```

2.6.4 LogfileInput

Tails a single log file, creating a message for each line in the file being monitored. Files are read in their entirety, and watched for changes. This input gracefully handles log rotation via the file moving but may lose a few log lines if using the "truncation" method of log rotation. It's recommended to use log rotation schemes that move the file to another location to avoid possible loss of log lines.

In the event the log file does not currently exist, it will be placed in an internal discover list, and checked for existence every *discover_interval* milliseconds (5000ms or 5s by default).

A single LogfileInput can only be used to read a single file. If you have multiple identical files spread across multiple directories (e.g. a */var/log/hosts/<HOSTNAME>/app.log* structure, where each <HOSTNAME> folder contains a log file originating from a separate host), you'll want to use the *config_logfile_directory_manager_input*.

Parameters:

- **logfile (string):** Each LogfileInput can have a single logfile to monitor.
- **hostname (string):** The hostname to use for the messages, by default this will be the machines qualified hostname. This can be set explicitly to ensure its the correct name in the event the machine has multiple interfaces/hostnames.
- **discover_interval (int):** During logfile rotation, or if the logfile is not originally present on the system, this interval is how often the existence of the logfile will be checked for. The default of 5 seconds is usually fine. This interval is in milliseconds.
- **stat_interval (int):** How often the file descriptors for each file should be checked to see if new log data has been written. Defaults to 500 milliseconds. This interval is in milliseconds.
- **logger (string):** Each LogfileInput may specify a logger name to use in the case an error occurs during processing of a particular line of logging text. By default, the logger name is set to the logfile name.
- **use_seek_journal (bool):** Specifies whether to use a seek journal to keep track of where we are in a file to be able to resume parsing from the same location upon restart. Defaults to true.
- **seek_journal_name (string):** Name to use for the seek journal file, if one is used. Only refers to the file name itself, not the full path; Heka will store all seek journals in a *seekjournal* folder relative to the Heka base directory. Defaults to a sanitized version of the *logger* value (which itself defaults to the filesystem path of the input file). This value is ignored if *use_seek_journal* is set to false.
- **resume_from_start (bool):** When heka restarts, if a logfile cannot safely resume reading from the last known position, this flag will determine whether hekad will force the seek position to be 0 or the end of file. By default, hekad will resume reading from the start of file.

New in version 0.4.

- **decoder (string):** A decoder must be specified for the message.proto parser (i.e. ProtobufDecoder) but is optional for token and regexp parsers (if no decoder is specified the parsed data is available in the Heka message payload).
- **parser_type (string):**
  - token - splits the log on a byte delimiter (default).
  - regexp - splits the log on a regexp delimiter.
  - message.proto - splits the log on protobuf message boundaries
- **delimiter (string): Only used for token or regexp parsers.** Character or regexp delimiter used by the parser (default "\n"). For the regexp delimiter a single capture group can be specified to preserve the delimiter (or part of the delimiter). The capture will be added to the start or end of the log line depending on the delimiter_location configuration. Note: when a start delimiter is used the last line in the file will not be processed (since the next record defines its end) until the log is rolled.
- **delimiter_location (string): Only used for regexp parsers.**
  - start - the regexp delimiter occurs at the start of a log line.
  - end - the regexp delimiter occurs at the end of the log line (default).

```
[LogfileInput]
logfile = "/var/log/opendirectoryd.log"
logger = "opendirectoryd"

[LogfileInput]
logfile = "/var/log/opendirectoryd.log"
```

2.6.5 LogfileDirectoryManagerInput

Scans for log files in a globbed directory path and when a new file matching the specified path is discovered it will start an instance of the LogfileInput plugin to process it. Each LogfileInput will inherit its configuration from the manager's settings with the logfile property properly adjusted.

Parameters: (identical to LogfileInput with the following exceptions)

- **logfile (string):** A path with a globbed directory component pointing to a common (statically named) log file. Note that only directories can be globbed; the file itself must have the same name in each directory.
- **seek_journal_name (string):** With a LogfileInput it is possible to specify a particular name for the seek journal file that will be used. This is not possible with the LogfileDirectoryManagerInput; the seek_journal_name will always be auto- generated, and any attempt to specify a hard coded seek_journal_name will be treated as a configuration error.
- **ticker_interval (uint):** Time interval (in seconds) between directory scans for new log files. Defaults to 0 (only scans once on startup).

```
[vhosts]
type = "LogfileDirectoryManagerInput"
logfile = "/var/log/vhost/*/apache.log"
```

Note: The spawned LogfileInput plugins are named *manager_name-logfile* i.e.,

- vhosts-/var/log/www/apache.log
- vhosts-/var/log/internal/apache.log

2.6.6 StatsdInput

Listens for statsd protocol *counter*, *timer*, or *gauge* messages on a UDP port, and generates *Stat* objects that are handed to a *StatAccumulator* for aggregation and processing.

Parameters:

- **address (string):** An IP address:port on which this plugin will expose a statsd server. Defaults to "127.0.0.1:8125".
- **stat_accum_name (string):** Name of a StatAccumInput instance that this StatsdInput will use as its StatAccumulator for submitting received stat values. Defaults to "StatAccumInput".

Example:

```
[StatsdInput]
address = ":8125"
stat_accum_input = "custom_stat_accumulator"
```

2.6.7 StatAccumInput

Provides an implementation of the *StatAccumulator* interface which other plugins can use to submit *Stat* objects for aggregation and roll-up. Accumulates these stats and then periodically emits a "stat metric" type message containing aggregated information about the stats received since the last generated message.

Parameters:

- **emit_in_payload (bool):** Specifies whether or not the aggregated stat information should be emitted in the payload of the generated messages, in the format accepted by the carbon portion of the graphite graphing software. Defaults to true.
- **emit_in_fields (bool):** Specifies whether or not the aggregated stat information should be emitted in the message fields of the generated messages. Defaults to false. *NOTE*: At least one of 'emit_in_payload' or 'emit_in_fields' *must* be true or it will be considered a configuration error and the input won't start.
- **percent_threshold (int):** Percent threshold to use for computing "upper_N%" type stat values. Defaults to 90.
- **ticker_interval (uint):** Time interval (in seconds) between generated output messages. Defaults to 10.
- **message_type (string):** String value to use for the *Type* value of the emitted stat messages. Defaults to "heka.statmetric".

2.6.8 ProcessInput

Executes one or more external programs on an interval, creating messages from the output. If a chain of commands is used, stdout is piped into the next command's stdin. In the event the program returns a non-zero exit code, ProcessInput will stop, logging the exit error.

Parameters: Each command is defined with the following parameters:

- Name (string): Each ProcessInput *must* have a name defined for logging purposes. The messages will be tagged with *name*.stdout or *name*.stderr in the *ProcessInputName* field of the heka message.
- **Command (map[uint]cmd_config):** The command is a structure that contains the full path to the binary, command line arguments, optional enviroment variables and an optional working directory. See the *cmd_config* definition below. ProcessInput expects the commands to be indexed by integers starting with 0.
- **ticker_interval (uint):** The number of seconds to wait between runnning *command*. Defaults to 15. A ticker_interval of 0 indicates that the command is run once.
- **stdout (bool):** Capture stdout from *command*. Defaults to true.
- **stderr (bool):** Capture stderr from *command*. Defaults to false.
- **decoder (string):** Name of the decoder instance to send messages to. Default is to inject messages back into the main heka router.
- **parser_type (string):**
  - token - splits the log on a byte delimiter (default).
  - regexp - splits the log on a regexp delimiter.
- **delimiter (string): Only used for token or regexp parsers.** Character or regexp delimiter used by the parser (default "\n"). For the regexp delimiter a single capture group can be specified to preserve the delimiter (or part of the delimiter). The capture will be added to the start or end of the log line depending on the delimiter_location configuration. Note: when a start delimiter is used the last line in the file will not be processed (since the next record defines its end) until the log is rolled.
- **delimiter_location (string): Only used for regexp parsers.**

  - start - the regexp delimiter occurs at the start of a log line.
  - end - the regexp delimiter occurs at the end of the log line (default).

- timeout (uint): Timeout in seconds before any one of the commands in the chain is terminated.
- trim (bool) : Trim a single trailing newline character if one exists. Default is true.

cmd_config structure

```
- bin (string):
```

  The full path to the binary that will be executed.

- **args ([]string):** Command line arguments to pass into the executable.
- **environment ([]string):** Used to set environment variables before *command* is run. Default is nil, which uses the heka process's environment.
- **directory (string):** Used to set the working directory of *Bin* Default is "", which uses the heka process's working directory.

```
[ProcessInput]
name = "DemoProcessInput"
ticker_interval = 2
parser_type = "token"
delimiter = " "
stdout = true
stderr = false
trim = true

[ProcessInput.Command.0]
bin = "/bin/cat"
Args = ["../testsupport/process_input_pipes_test.txt"]

[ProcessInput.Command.1]
bin = "/usr/bin/grep"
Args = ["ignore"]
```

2.6.9 HttpInput

Starts a HTTP client which intermittently polls a URL for data. The entire response body is parsed by a decoder into a pipeline pack. Data is always fetched using HTTP GET and any errors are logged and are not fatal for the plugin.

Parameters:

- **url (string):** A HTTP URL which this plugin will regularly poll for data. No default URL is specified.
- **ticker_interval (uint):** Time interval (in seconds) between attempts to poll for new data. Defaults to 10.
- **decoder (string):** The name of the decoder used to transform the response body text into a structured hekad message. No default decoder is specified.

Example:

```
[HttpInput]
url = "http://localhost:9876/"
ticker_interval = 5
decoder = "ProtobufDecoder"
```

2.7 Decoders

A decoder may be specified for each encoding type defined in message.pb.go. Unless you are using a custom decoder you probably won't need to specify these by hand, by default the ProtobufDecoder will be configured as if you had included the following configuration.

Example:

```
[ProtobufDecoder]
encoding_name = "PROTOCOL_BUFFER"
```

The ProtobufDecoder converts protocol buffers serialized messages to *Message* struct objects. The hekad protocol buffers message schema in defined in the *message.proto* file in the *message* package.

Note: These sections remain configurable explicitly in the configuration file for possible future use where a different Decoder may want to handle one of these encodings.

See Also:

Protocol Buffers - Google's data interchange format

2.7.1 PayloadRegexDecoder

Decoder plugin that accepts messages of a specified form and generates new outgoing messages from extracted data, effectively transforming one message format into another. Can be combined w/ *message_matcher* capture groups (see *matcher_capture_groups*) to extract unstructured information from message payloads and use it to populate *Message* struct attributes and fields in a more structured manner.

Parameters:

- **match_regex:** Regular expression that must match for the decoder to process the message.
- **severity_map:** Subsection defining severity strings and the numerical value they should be translated to. hekad uses numerical severity codes, so a severity of *WARNING* can be translated to *3* by settings in this section.
- **message_fields:** Subsection defining message fields to populate and the interpolated values that should be used. Valid interpolated values are any captured in a regex in the message_matcher, and any other field that exists in the message. In the event that a captured name overlaps with a message field, the captured name's value will be used. Optional representation metadata can be added at the end of the field name using a pipe delimiter i.e. ResponseSize|B = "%ResponseSize%" will create Fields[ResponseSize] representing the number of bytes. Adding a representation string to a standard message header name will cause it to be added as a user defined field i.e., Payload|json will create Fields[Payload] with a json representation.

  Interpolated values should be surrounded with % signs, for example:

  ```
  [my_decoder.message_fields]
  Type = "%Type%Decoded"
  ```

  This will result in the new message's Type being set to the old messages Type with *Decoded* appended.
- **timestamp_layout (string):** A formatting string instructing hekad how to turn a time string into the actual time representation used internally. Example timestamp layouts can be seen in Go's time documetation.
- **timestamp_location (string):** Time zone in which the timestamps in the text are presumed to be in. Should be a location name corresponding to a file in the IANA Time Zone database (e.g. "America/Los_Angeles"), as parsed by Go's *time.LoadLocation()* function (see http://golang.org/pkg/time/#LoadLocation). Defaults to "UTC". Not required if valid time zone info is embedded in every parsed timestamp, since those can be parsed as specified in the *timestamp_layout*.

Example (Parsing Apache Combined Log Format):

```
[apache_transform_decoder]
type = "PayloadRegexDecoder"
match_regex = '/^(?P<RemoteIP>\S+) \S+ \S+ \[(?P<Timestamp>[^\]]+)\] "(?P<Method>[A-Z]+) (?P<Url>[^\s
timestamplayout = "02/Jan/2006:15:04:05 -0700"

[apache_transform_decoder.severity_map]
DEBUG = 1
WARNING = 2
INFO = 3

[apache_transform_decoder.message_fields]
Type = "ApacheLogfile"
Logger = "apache"
Url|uri = "%Url%"
Method = "%Method%"
Status = "%Status%"
RequestSize|B = "%RequestSize%"
Referer = "%Referer%"
Browser = "%Browser%"
```

2.7.2 PayloadJsonDecoder

This decoder plugin accepts JSON blobs and allows you to map parts of the JSON into Field attributes of the pipelinepack message using JSONPath syntax.

Parameters:

- **json_map:** A subsection defining a capture name that maps to a JSONPath expression. Each expression can fetch a single value, if the expression does not resolve to a valid node in the JSON message, the capture group will be assigned an empty string value.
- **severity_map:** Subsection defining severity strings and the numerical value they should be translated to. hekad uses numerical severity codes, so a severity of *WARNING* can be translated to *3* by settings in this section.
- **message_fields:** Subsection defining message fields to populate and the interpolated values that should be used. Valid interpolated values are any captured in a JSONPath in the message_matcher, and any other field that exists in the message. In the event that a captured name overlaps with a message field, the captured name's value will be used. Optional representation metadata can be added at the end of the field name using a pipe delimiter i.e. ResponseSize|B = "%ResponseSize%" will create Fields[ResponseSize] representing the number of bytes. Adding a representation string to a standard message header name will cause it to be added as a user defined field i.e., Payload|json will create Fields[Payload] with a json representation.

  Interpolated values should be surrounded with % signs, for example:

  ```
  [my_decoder.message_fields]
  Type = "%Type%Decoded"
  ```

  This will result in the new message's Type being set to the old messages Type with *Decoded* appended.
- **timestamp_layout (string):** A formatting string instructing hekad how to turn a time string into the actual time representation used internally. Example timestamp layouts can be seen in Go's time documetation. The default layout is ISO8601 - the same as Javascript.
- **timestamp_location (string):** Time zone in which the timestamps in the text are presumed to be in. Should be a location name corresponding to a file in the IANA Time Zone database (e.g. "America/Los_Angeles"), as parsed by Go's *time.LoadLocation()* function (see http://golang.org/pkg/time/#LoadLocation). Defaults

to "UTC". Not required if valid time zone info is embedded in every parsed timestamp, since those can be parsed as specified in the *timestamp_layout*.

Example:

```
[myjson_decoder]
type = "PayloadJsonDecoder"

[myjson_decoder.json_map]
Count = "$.statsd.count"
Name = "$.statsd.name"
Pid = "$.pid"
Timestamp = "$.timestamp"

[myjson_decoder.severity_map]
DEBUG = 1
WARNING = 2
INFO = 3

[myjson_decoder.message_fields]
Pid = "%Pid%"
StatCount = "%Count%"
StatName =  "%Name%"
Timestamp = "%Timestamp%"
```

PayloadJsonDecoder's json_map config subsection only supports a small subset of valid JSONPath expressions.

JSONPath	Description
$	the root object/element
.	child operator
[]	subscript operator to iterate over arrays

2.7.3 Examples:

```
var s = {
    "foo": {
        "bar": [
            {
                "baz": "",
                "noo": "aaa"
            },
            {
                "maz": "123",
                "moo": 256
            }
        ],
        "boo": {
            "bag": true,
            "bug": false
        }
    }
}

# Valid paths
$.foo.bar[0].baz
$.foo.bar
```

2.7.4 PayloadXmlDecoder

This decoder plugin accepts XML blobs in the message payload and allows you to map parts of the XML into Field attributes of the pipelinepack message using XPath syntax using the xmlpath library.

Parameters:

- **xpath_map:** A subsection defining a capture name that maps to an XPath expression. Each expression can fetch a single value, if the expression does not resolve to a valid node in the XML blob, the capture group will be assigned an empty string value.
- **severity_map:** Subsection defining severity strings and the numerical value they should be translated to. hekad uses numerical severity codes, so a severity of *WARNING* can be translated to *3* by settings in this section.
- **message_fields:** Subsection defining message fields to populate and the interpolated values that should be used. Valid interpolated values are any captured in an XPath in the message_matcher, and any other field that exists in the message. In the event that a captured name overlaps with a message field, the captured name's value will be used. Optional representation metadata can be added at the end of the field name using a pipe delimiter i.e. ResponseSize|B = "%ResponseSize%" will create Fields[ResponseSize] representing the number of bytes. Adding a representation string to a standard message header name will cause it to be added as a user defined field i.e., Payload|json will create Fields[Payload] with a json representation.

  Interpolated values should be surrounded with % signs, for example:

  ```
  [my_decoder.message_fields]
  Type = "%Type%Decoded"
  ```

  This will result in the new message's Type being set to the old messages Type with *Decoded* appended.
- **timestamp_layout (string):** A formatting string instructing hekad how to turn a time string into the actual time representation used internally. Example timestamp layouts can be seen in Go's time documetation. The default layout is ISO8601 - the same as Javascript.
- **timestamp_location (string):** Time zone in which the timestamps in the text are presumed to be in. Should be a location name corresponding to a file in the IANA Time Zone database (e.g. "America/Los_Angeles"), as parsed by Go's *time.LoadLocation()* function (see http://golang.org/pkg/time/#LoadLocation). Defaults to "UTC". Not required if valid time zone info is embedded in every parsed timestamp, since those can be parsed as specified in the *timestamp_layout*.

Example:

```
[myxml_decoder]
type = "PayloadXmlDecoder"

[myxml_decoder.xpath_map]
Count = "/some/path/count"
Name = "/some/path/name"
Pid = "//pid"
Timestamp = "//timestamp"

[myxml_decoder.severity_map]
DEBUG = 1
WARNING = 2
INFO = 3

[myxml_decoder.message_fields]
Pid = "%Pid%"
StatCount = "%Count%"
StatName =  "%Name%"
Timestamp = "%Timestamp%"
```

PayloadXmlDecoder's xpath_map config subsection supports XPath as implemented by the xmlpath library.

- All axes are supported ("child", "following-sibling", etc)
- All abbreviated forms are supported (".", "//", etc)
- All node types except for namespace are supported
- Predicates are restricted to [N], [path], and [path=literal] forms
- Only a single predicate is supported per path step
- Richer expressions and namespaces are not supported

New in version 0.4.

2.7.5 StatsToFieldsDecoder

The StatsToFieldsDecoder will parse statsd data in the graphite message format and encode the data into the message fields, in the same format produced by a *config_stat_accum_input* plugin with the *emit_in_fields* value set to true. This is useful if you have externally generated statsd string data flowing through Heka that you'd like to process without having to roll your own string parsing code.

This decoder has no configuration options, it simply expects to be passed a message with statsd string data in the payload. Incorrect or malformed content will cause a decoding error, dropping the message.

The fields format only contains a single "timestamp" field, so any payloads containing multiple timestamps will end up generating a separate message for each timestamp. Extra messages will be a copy of the original message except a) the payload will be empty and b) the unique timestamp and related stats will be the only message fields.

2.7.6 MultiDecoder

This decoder plugin allows you to specify an ordered list of delegate decoders. The MultiDecoder will pass the PipelinePack to be decoded to each of the delegate decoders in turn until decode succeeds. In the case of failure to decode, MultiDecoder will return an error and recycle the message.

Parameters:

- **subs:** A subsection is used to declare the TOML configuration for any delegate decoders. The default is that no delegate decoders are defined.
- **order (list of strings):** PipelinePack objects will be passed in order to each decoder in this list. Default is an empty list.
- **name (string):** Defaults to MultiDecoder-<address of multidecoder>.
- **log_sub_errors (bool):** If true, the DecoderRunner will log the errors returned whenever a delegate decoder fails to decode a message. Defaults to false.
- **cascade_strategy (string):** Specifies behavior the MultiDecoder should exhibit with regard to cascading through the listed decoders. Supports only two valid values: "first-wins" and "all". With "first-wins", each decoder will be tried in turn until there is a successful decoding, after which decoding will be stopped. With "all", all listed decoders will be applied whether or not they succeed. In each case, decoding will only be considered to have failed if *none* of the sub-decoders succeed.

Example (Two PayloadRegexDecoder delegates):

```
[syncdecoder]
type = "MultiDecoder"
order = ['syncformat', 'syncraw']
```

```
[syncdecoder.subs.syncformat]
type = "PayloadRegexDecoder"
match_regex = '^(?P<RemoteIP>\S+) \S+ (?P<User>\S+) \[(?P<Timestamp>[^\]]+)\] "(?P<Method>[A-Z]+) (?P
timestamp_layout = "02/Jan/2006:15:04:05 -0700"

[syncdecoder.subs.syncformat.message_fields]
RemoteIP|ipv4 = "%RemoteIP%"
User = "%User%"
Method = "%Method%"
Url|uri = "%Url%"
StatusCode = "%StatusCode%"
RequestSize|B= "%RequestSize%"
Referer = "%Referer%"
Browser = "%Browser%"
ResponseTime|s = "%ResponseTime%"
ResponseSize|B = "%ResponseSize%"
Payload = ""

[syncdecoder.subs.syncraw]
type = "PayloadRegexDecoder"
match_regex = '^(?P<TheData>.*)'

[syncdecoder.subs.syncraw.message_fields]
Somedata = "%TheData%"
```

2.7.7 Sandbox Decoder

The sandbox decoder provides an isolated execution environment for data parsing and complex transformations without the need to recompile Heka.

SandboxDecoder Settings

2.8 Common Filter / Output Parameters

There are some configuration options that are universally available to all Heka filter and output plugins. These will be consumed by Heka itself when Heka initializes the plugin and do not need to be handled by the plugin- specific initialization code.

- **message_matcher (string, optional):** Boolean expression, when evaluated to true passes the message to the filter for processing. Defaults to matching nothing. See: *Message Matcher Syntax*
- **message_signer (string, optional):** The name of the message signer. If specified only messages with this signer are passed to the filter for processing.
- **ticker_interval (uint, optional):** Frequency (in seconds) that a timer event will be sent to the filter. Defaults to not sending timer events.

2.9 Filters

2.9.1 CounterFilter

Once a second a *CounterFilter* will generate a message of type *heka.counter- output*. The payload will contain text indicating the number of messages that matched the filter's *message_matcher* value during that second (i.e. it counts

the messages the plugin received). Every ten seconds an extra message (also of type *heka.counter-output*) goes out, containing an aggregate count and average per second throughput of messages received.

Parameters: **None**

Example:

```
[CounterFilter]
message_matcher = "Type != 'heka.counter-output'"
```

2.9.2 StatFilter

Filter plugin that accepts messages of a specfied form and uses extracted message data to generate statsd-style numerical metrics in the form of *Stat* objects that can be consumed by a *StatAccumulator*.

Parameters:

- **Metric:** Subsection defining a single metric to be generated
  - **type (string):** Metric type, supports "Counter", "Timer", "Gauge".
  - **name (string):** Metric name, must be unique.
  - **value (string):** Expression representing the (possibly dynamic) value that the *StatFilter* should emit for each received message.
- **stat_accum_name (string):** Name of a StatAccumInput instance that this StatFilter will use as its StatAccumulator for submitting generate stat values. Defaults to "StatAccumInput".

Example (Assuming you had TransformFilter inserting messages as above):

```
[StatsdInput]
address = "127.0.0.1:29301"
stat_accum_name = "my_stat_accum"

[my_stat_accum]
flushInterval = 5

[Hits]
type = "StatFilter"
stat_accum_name = "my_stat_accum"
message_matcher = 'Type == "ApacheLogfile"'

[Hits.Metric.bandwidth]
type = "Counter"
name = "httpd.bytes.%Hostname%"
value = "%Bytes%"

[Hits.Metric.method_counts]
type = "Counter"
name = "httpd.hits.%Method%.%Hostname%"
value = "1"
```

Note: StatFilter requires an available StatAccumulator to be running.

2.9.3 SandboxFilter

The sandbox filter provides an isolated execution environment for data analysis.

SandboxFilter Settings

2.9.4 SandboxManagerFilter

The sandbox manager provides dynamic control (start/stop) of sandbox filters in a secure manner without stopping the Heka daemon.

sandboxmanagerfilter_settings

2.10 Outputs

2.10.1 AMQPOutput

Connects to a remote AMQP broker (RabbitMQ) and sends messages to the specified queue. The message is serialized if specified, otherwise only the raw payload of the message will be sent. As AMQP is dynamically programmable, the broker topology needs to be specified.

Parameters:

- **URL (string):** An AMQP connection string formatted per the RabbitMQ URI Spec.
- **Exchange (string):** AMQP exchange name
- **ExchangeType (string):** AMQP exchange type (*fanout*, *direct*, *topic*, or *headers*).
- **ExchangeDurability (bool):** Whether the exchange should be configured as a durable exchange. Defaults to non-durable.
- **ExchangeAutoDelete (bool):** Whether the exchange is deleted when all queues have finished and there is no publishing. Defaults to auto-delete.
- **RoutingKey (string):** The message routing key used to bind the queue to the exchange. Defaults to empty string.
- **Persistent (bool):** Whether published messages should be marked as persistent or transient. Defaults to non-persistent.
- **Serialize (bool):** Whether published messages should be fully serialized. If set to true then messages will be encoded to Protocol Buffers and have the AMQP message Content-Type set to *application/hekad*. Defaults to true.

Example (that sends log lines from the logger):

```
[AMQPOutput]
url = "amqp://guest:guest@rabbitmq/"
exchange = "testout"
exchangeType = "fanout"
message_matcher = 'Logger == "/var/log/system.log"'
```

2.10.2 LogOutput

Logs messages to stdout using Go's *log* package.

Parameters:

- **payload_only (bool, optional):** If set to true, then only the message payload string will be output, otherwise the entire *Message* struct will be output in JSON format.

Example:

```
[counter_output]
type = "LogOutput"
message_matcher = "Type == 'heka.counter-output'"
payload_only = true
```

2.10.3 FileOutput

Writes message data out to a file system.

Parameters:

- **path (string):** Full path to the output file.
- **format (string, optional):** Output format for the message to be written. Supports *json* or *protobufstream*, both of which will serialize the entire *Message* struct, or *text*, which will output just the payload string. Defaults to `text`.
- **prefix_ts (bool, optional):** Whether a timestamp should be prefixed to each message line in the file. Defaults to `false`.
- **perm (string, optional):** File permission for writing. A string of the octal digit representation. Defaults to "644".

Example:

```
[counter_file]
type = "FileOutput"
message_matcher = "Type == 'heka.counter-output'"
path = "/var/log/heka/counter-output.log"
prefix_ts = true
perm = "666"
```

2.10.4 TcpOutput

Output plugin that serializes messages into the Heka protocol format and delivers them to a listening TCP connection. Can be used to deliver messages from a local running Heka agent to a remote Heka instance set up as an aggregator and/or router.

Parameters:

- **address (string):** An IP address:port to which we will send our output data.

Example:

```
[aggregator_output]
type = "TcpOutput"
address = "heka-aggregator.mydomain.com:55"
message_matcher = "Type != 'logfile' && Type != 'heka.counter-output' && Type != 'heka.all-report'"
```

2.10.5 DashboardOutput

Specialized output plugin that listens for certain Heka reporting message types and generates JSON data which is made available via HTTP for use in web based dashboards and health reports.

Parameters:

- **ticker_interval (uint):** Specifies how often, in seconds, the dashboard files should be updated. Defaults to 5.
- **message_matcher (string):** Defaults to *"Type == 'heka.all-report' || Type == 'heka.sandbox-output' || Type == 'heka.sandbox-terminated"'*. Not recommended to change this unless you know what you're doing.
- **address (string):** An IP address:port on which we will serve output via HTTP. Defaults to "0.0.0.0:4352".
- **working_directory (string):** File system directory into which the plugin will write data files and from which it will serve HTTP. The Heka process must have read / write access to this directory. Relative paths will be evaluated relative to the Heka base directory. Defaults to "dashboard" (i.e. "$(BASE_DIR)/dashboard").
- **static_directory (string):** File system directory where the Heka dashboard source code can be found. The Heka process must have read access to this directory. Relative paths will be evaluated relative to the Heka base directory. Defaults to "/usr/share/heka/dasher".

Example:

```
[DashboardOutput]
ticker_interval = 30
```

2.10.6 ElasticSearchOutput

Output plugin that serializes messages into JSON structures and uses HTTP requests to insert them into an ElasticSearch database.

Parameters:

- **cluster (string):** ElasticSearch cluster name. Defaults to "elasticsearch"
- **index (string):** Name of the ES index into which the messages will be inserted. If Field Name|Type|Hostname|Pid|UUID|Logger|EnvVersion|Severity are placed between within a %{}, it will be interpolated to their message value. Defaults to "heka-%{2006.01.02}".
- **type_name (string):** Name of ES record type to create. Defaults to "message". If Field Name|Type|Hostname|Pid|UUID|Logger|EnvVersion|Severity are placed between within a %{}, it will be interpolated to their message value.
- **flush_interval (int):** Interval at which accumulated messages should be bulk indexed into ElasticSearch, in milliseconds. Defaults to 1000 (i.e. one second).
- **flush_count (int):** Number of messages that, if processed, will trigger them to be bulk indexed into ElasticSearch. Defaults to 10.
- **format (string):** Message serialization format, either "clean", "logstash_v0", "payload" or "raw". "clean" is a more concise JSON representation of the message, "logstash_v0" outputs in a format similar to Logstash's original (i.e. "version 0") ElasticSearch schema, "payload" passes the message payload directly into ElasticSearch, and "raw" is a full JSON representation of the message. Defaults to "clean".
- **fields ([]string):** If the format is "clean", then the 'fields' parameter can be used to specify that only specific message data should be indexed into ElasticSearch. Available fields to choose are "Uuid", "Timestamp", "Type", "Logger", "Severity", "Payload", "EnvVersion", "Pid", "Hostname", and "Fields" (where "Fields" causes the inclusion of any and all dynamically specified message fields. Defaults to all.
- **timestamp (string):** Format to use for timestamps in generated ES documents. Defaults to "2006-01-02T15:04:05.000Z".
- **server (string):** ElasticSearch server URL. Supports http://, https:// and udp:// urls. Defaults to "http://localhost:9200".
- **ESIndexFromTimestamp (bool):** When generating the index name use the timestamp from the message instead of the current time. Defaults to false.

Example:

```
[ElasticSearchOutput]
message_matcher = "Type == 'sync.log'"
cluster = "elasticsearch-cluster"
index = "synclog-%{2006.01.02.15.04.05}"
type_name = "sync.log.line"
server = "http://es-server:9200"
format = "clean"
flush_interval = 5000
flush_count = 10
```

2.10.7 WhisperOutput

WhisperOutput plugins parse the "statmetric" messages generated by a StatAccumulator and write the extracted counter, timer, and gauge data out to a graphite compatible whisper database file tree structure.

Parameters:

- **base_path (string):** Path to the base directory where the whisper file tree will be written. Absolute paths will be honored, relative paths will be calculated relative to the Heka base directory. Defaults to "whisper" (i.e. "$(BASE_DIR)/whisper").
- **default_agg_method (int):** Default aggregation method to use for each whisper output file. Supports the following values:
  0. Unknown aggregation method.
  1. Aggregate using averaging. (default)
  2. Aggregate using summation.
  3. Aggregate using last received value.
  4. Aggregate using maximum value.
  5. Aggregate using minimum value.
- **default_archive_info ([][]int):** Default specification for new whisper db archives. Should be a sequence of 3-tuples, where each tuple describes a time interval's storage policy: [<offset> <# of secs per datapoint> <# of datapoints>] (see whisper docs for more info). Defaults to:

  ```
  [ [0, 60, 1440], [0, 900, 8], [0, 3600, 168], [0, 43200, 1456]]
  ```

  The above defines four archive sections. The first uses 60 seconds for each of 1440 data points, which equals one day of retention. The second uses 15 minutes for each of 8 data points, for two hours of retention. The third uses one hour for each of 168 data points, or 7 days of retention. Finally, the fourth uses 12 hours for each of 1456 data points, representing two years of data.
- **folder_perm (string):** Permission mask to be applied to folders created in the whisper database file tree. Must be a string representation of an octal integer. Defaults to "700".

Example:

```
[WhisperOutput]
message_matcher = "Type == 'heka.statmetric'"
default_agg_method = 3
default_archive_info = [ [0, 30, 1440], [0, 900, 192], [0, 3600, 168], [0, 43200, 1456] ]
folder_perm = "755"
```

2.10.8 NagiosOutput

Specialized output plugin that listens for Nagios external command message types and generates an HTTP request against the Nagios cmd.cgi API. Currently the output will only send passive service check results. The message payload must consist of a state followed by a colon and then the message i.e., "OK:Service is functioning properly". The valid states are: OK|WARNING|CRITICAL|UNKNOWN. Nagios must be configured with a service name that matches the Heka plugin instance name and the hostname where the plugin is running.

Parameters:

- **url (string, optional):** An HTTP URL to the Nagios cmd.cgi. Defaults to "http://localhost/nagios/cgi-bin/cmd.cgi".
- **username (string, optional):** Username used to authenticate with the Nagios web interface. Defaults to "".
- **password (string, optional):** Password used to authenticate with the Nagios web interface. Defaults to "".
- **responseheadertimeout (uint, optional):** Specifies the amount of time, in seconds, to wait for a server's response headers after fully writing the request. Defaults to 2.

Example configuration to output alerts from SandboxFilter plugins:

```
[NagiosOutput]
url = "http://localhost/nagios/cgi-bin/cmd.cgi"
username = "nagiosadmin"
password = "nagiospw"
message_matcher = "Type == 'heka.sandbox-output' && Fields[payload_type] == 'nagios-external-command'
```

Example Lua code to generate a Nagios alert:

```
output("OK:Alerts are working!")
inject_message("nagios-external-command", "PROCESS_SERVICE_CHECK_RESULT")
```

2.10.9 CarbonOutput

CarbonOutput plugins parse the "stat metric" messages generated by a StatAccumulator and write the extracted counter, timer, and gauge data out to a graphite compatible carbon daemon. Output is written over a TCP socket using the plaintext protocol.

Parameters:

- **address (string):** An IP address:port on which this plugin will write to. Defaults to: localhost:2003

Example:

```
[CarbonOutput]
message_matcher = "Type == 'heka.statmetric'"
address = "localhost:2003"
```

CHAPTER 3

Monitoring Internal State

Heka can emit metrics about it's internal state to either an outgoing Heka message (and, through the DashboardOutput, to a web dashboard) or to stdout. Sending SIGUSR1 to hekad on a UNIX will send a plain text report tostdout. On Windows, you will need to send signal 10 to the hekad process using Powershell.

Sample text output

```
========[heka.all-report]========
inputRecycleChan:
    InChanCapacity: 100
    InChanLength: 99
injectRecycleChan:
    InChanCapacity: 100
    InChanLength: 98
Router:
    InChanCapacity: 50
    InChanLength: 0
    ProcessMessageCount: 26
ProtobufDecoder-0:
    InChanCapacity: 50
    InChanLength: 0
ProtobufDecoder-1:
    InChanCapacity: 50
    InChanLength: 0
ProtobufDecoder-2:
    InChanCapacity: 50
    InChanLength: 0
ProtobufDecoder-3:
    InChanCapacity: 50
    InChanLength: 0
DecoderPool-ProtobufDecoder:
    InChanCapacity: 4
    InChanLength: 4
OpsSandboxManager:
    InChanCapacity: 50
    InChanLength: 0
    MatchChanCapacity: 50
    MatchChanLength: 0
    MatchAvgDuration: 0
    ProcessMessageCount: 0
hekabench_counter:
```

```
    InChanCapacity: 50
    InChanLength: 0
    MatchChanCapacity: 50
    MatchChanLength: 0
    MatchAvgDuration: 445
    ProcessMessageCount: 0
    InjectMessageCount: 0
    Memory: 20644
    MaxMemory: 20644
    MaxInstructions: 18
    MaxOutput: 0
    ProcessMessageAvgDuration: 0
    TimerEventAvgDuration: 78532
LogOutput:
    InChanCapacity: 50
    InChanLength: 0
    MatchChanCapacity: 50
    MatchChanLength: 0
    MatchAvgDuration: 406
DashboardOutput:
    InChanCapacity: 50
    InChanLength: 0
    MatchChanCapacity: 50
    MatchChanLength: 0
    MatchAvgDuration: 336
========
```

To enable the HTTP interface, you will need to enable the dashboard output plugin, see *config_dashboard_output*.

CHAPTER 4

Extending Heka

The core of the Heka engine is written in the Go programming language. Heka supports four different types of plugins (inputs, decoders, filters, and outputs), which are also written in Go. This document will try to provide enough information for developers to extend Heka by implementing their own custom plugins. It assumes a small amount of familiarity with Go, although any reasonably experienced programmer will probably be able to follow along with no trouble.

NOTE: Heka also supports the use of Lua for dynamically loaded, security sandboxed filter plugins. This document only covers the use of Go plugins. You can learn more about sandboxed plugins in the *Sandbox* section.

4.1 Definitions

You should be familiar with the *glossary* terminology before proceeding.

4.2 Overview

Each Heka plugin type performs a specific task: inputs receive input from the outside world and inject the data into the Heka pipeline, decoders turn binary data into Message objects that Heka can process, filters perform arbitrary processing of Heka message data, and outputs send data from Heka back to the outside world. Each specific plugin has some custom behaviour, but it also shares behaviour w/ every other plugin of that type. A UDPInput and a TCPInput listen on the network differently, and a LogFileInput (reading logs off the file system) doesn't listen on the network at all, but all of these inputs need to interact w/ the Heka system to access data structures, gain access to decoders to which we pass our incoming data, respond to shutdown and other system events, etc.

To support this, each Heka plugin actually consists of two parts: the plugin itself, and an accompanying "plugin runner". Inputs have an InputRunner, decoders have a DecoderRunner, filters have a FilterRunner, and Outputs have an OutputRunner. The plugin itself contains the plugin-specific behaviour, and is provided by the plugin developer. The plugin runner contains the shared (by type) behaviour, and is provided by Heka. When Heka starts a plugin, it a) creates and configures a plugin instance of the appropriate type, b) creates a plugin runner instance of the appropriate type (passing in the plugin), and c) calls the Start method of the plugin runner. Most plugin runners (excepting decoders) then call the plugin's Run method, passing themselves and an additional PluginHelper object in as arguments so the plugin code can use their exposed APIs to interact w/ the Heka system.

For inputs, filters, and outputs, there's a 1:1 correspondence between sections specified in the config file and running plugin instances. This is not the case for decoders, however; a pool of decoder instances are created so that messages

from different sources can be decoded in parallel. Plugins can gain access to a set of running decoders using the DecoderSet method of the provided PluginHelper.

4.3 Plugin Configuration

Heka uses TOML as its configuration file format (see: *Configuring hekad*), and provides a simple mechanism through which plugins can integrate with the configuration loading system to initialize themselves from settings in hekad's config file.

The minimal shared interface that a Heka plugin must implement in order to use the config system is (unsurprisingly) *Plugin*, defined in pipeline_runner.go:

```
type Plugin interface {
        Init(config interface{}) error
}
```

During Heka initialization an instance of every input, filter, and output plugin (and many instances of every decoder) listed in the configuration file will be created. The TOML configuration for each plugin will be parsed and the resulting configuration object will be passed in to the above specified *Init* method. The argument is of type *interface{}*; by default the underlying type will be **pipeline.PluginConfig*, a map object that provides config data as key/value pairs. There is also a way for plugins to specify a custom struct to be used instead of the generic *PluginConfig* type (see *Custom Plugin Config Structs*). In either case, the config object will be already loaded with values read in from the TOML file, which your plugin code can then use to initialize itself.

As an example, imagine we're writing a filter that will deliver messages to a specific output plugin, but only if they come from a list of approved hosts. Both 'hosts' and 'output' would be required in the plugin's config section. Here's one version of what the plugin definition and *Init* method might look like:

```
type HostFilter struct {
    hosts  map[string]bool
    output string
}

// Extract hosts value from config and store it on the plugin instance.
func (f *HostFilter) Init(config interface{}) error {
    var (
        hostsConf  interface{}
        hosts      []interface{}
        host       string
        outputConf interface{}
        ok         bool
    )
    conf := config.(pipeline.PluginConfig)
    if hostsConf, ok = conf["hosts"]; !ok {
        return errors.New("No 'hosts' setting specified.")
    }
    if hosts, ok = hostsConf.([]interface{}); !ok {
        return errors.New("'hosts' setting not a sequence.")
    }
    if outputConf, ok = conf["output"]; !ok {
        return errors.New("No 'output' setting specified.")
    }
    if f.output, ok = outputConf.(string); !ok {
        return errors.New("'output' setting not a string value.")
    }
    f.hosts = make(map[string]bool)
    for _, h := range hosts {
```

```
        if host, ok = h.(string); !ok {
            return errors.New("Non-string host value.")
        }
        f.hosts[host] = true
    }
    return nil
}
```

(Note that this is a bit of a contrived example. In practice, you would generally route messages to specific outputs using the *Message Matcher Syntax*.)

4.4 Restarting Plugins

In the event that your plugin fails to initialize properly at startup, hekad will exit. However, once hekad is running, if a plugin should fail (perhaps because a network connection dropped, a file became unavailable, etc), then hekad will shutdown. This shutdown can be avoided if your plugin supports being restarted.

To add restart support to your plugin, the *Restarting* interface defined in the config.go file:

```
type Restarting interface {
    CleanupForRestart()
}
```

A plugin that implements this interface will not trigger shutdown should it fail while hekad is running. The *CleanupForRestart* method will be called when the plugins' main run method exits, a single time. Then the runner will repeatedly call the plugins Init method until it initializes successfully. It will then resume running it unless it exits again at which point the restart process will begin anew.

4.5 Custom Plugin Config Structs

In simple cases it might be fine to get plugin configuration data as a generic map of keys and values, but if there are more than a couple of config settings then checking for, extracting, and validating the values quickly becomes a lot of work. Heka plugins can instead specify a schema struct for their configuration data, into which the TOML configuration will be decoded.

Plugins that wish to provide a custom configuration struct should implement the *HasConfigStruct* interface defined in the config.go file:

```
type HasConfigStruct interface {
        ConfigStruct() interface{}
}
```

Any plugin that implements this method should return a struct that can act as the schema for the plugin configuration. Heka's config loader will then try to decode the plugin's TOML config into this struct. Note that this also gives you a way to specify default config values; you just populate your config struct as desired before returning it from the *ConfigStruct* method.

Let's say we wanted to write a *UdpOutput* that delivered messages to a UDP listener somewhere, defaulting to using my.example.com:44444 as the destination. The initialization code might look as follows:

```
// This is our plugin struct.
type UdpOutput struct {
    conn net.Conn
}
```

```
// This is our plugin's config struct
type UdpOutputConfig struct {
    Address string
}

// Provides pipeline.HasConfigStruct interface.
func (o *UdpOutput) ConfigStruct() interface{} {
    return &UdpOutputConfig{"my.example.com:44444"}
}

// Initialize UDP connection
func (o *UdpOutput) Init(config interface{}) (err error) {
    conf := config.(*UdpOutputConfig) // assert we have the right config type
    var udpAddr *net.UDPAddr
    if udpAddr, err = net.ResolveUDPAddr("udp", conf.Address); err != nil {
        return fmt.Errorf("can't resolve %s: %s", conf.Address,
            err.Error())
    }
    if o.conn, err = net.DialUDP("udp", nil, udpAddr); err != nil {
        return fmt.Errorf("error dialing %s: %s", conf.Address,
            err.Error())
    }
    return
}
```

In addition to specifying configuration options that are specific to your plugin, it is also possible to use the config struct to specify default values for the *ticker_interval* and *message_matcher* values that are available to all Filter and Output plugins. If a config struct contains a uint attribute called *TickerInterval*, that will be used as a default ticker interval value (in seconds) if none is supplied in the TOML. Similarly, if a config struct contains a string attribute called *MessageMatcher*, that will be used as the default message routing rule if none is specified in the configuration file.

There is an optional configuration interface called WantsName. It provides a a plug-in access to its configured name before the runner has started. The Sandbox filter plug-in uses the name to locate/load any preserved state before being run.

type WantsName interface { SetName(name string)

}

4.6 Inputs

Input plugins are responsible for acquiring data from the outside world and injecting this data into the Heka pipeline. An input might be passively listening for incoming network data or actively scanning external sources (either on the local machine or over a network). The input plugin interface is:

```
type Input interface {
        Run(ir InputRunner, h PluginHelper) (err error)
        Stop()
}
```

The *Run* method is called when Heka starts and, if all is functioning as intended, should not return until Heka is shut down. If a condition arises such that the input can not perform its intended activity it should return with an appropriate error, otherwise it should continue to run until a shutdown event is triggered by Heka calling the input's *Stop* method, at which time any clean-up should be done and a clean shutdown should be indicated by returning a *nil* error.

Inside the *Run* method, an input has three primary responsibilities:

1. Acquire information from the outside world
2. Use acquired information to populate *PipelinePack* objects that can be processed by Heka.
3. Pass the populated *PipelinePack* objects on to the appropriate next stage in the Heka pipeline (either to a decoder plugin so raw input data can be converted to a *Message* object, or by injecting them directly into the Heka message router if the *Message* object is already populated.)

The details of the first step are clearly entirely defined by the plugin's intended input mechanism(s). Plugins can (and should!) spin up goroutines as needed to perform tasks such as listening on a network connection, making requests to external data sources, scanning machine resources and operational characteristics, reading files from a file system, etc.

For the second step, before you can populate a *PipelinePack* object you have to actually *have* one. You can get empty packs from a channel provided to you by the *InputRunner*. You get the channel itself by calling *ir.InChan()* and then pull a pack from the channel whenever you need one.

Often, populating a *PipelinePack* is as simple as storing the raw data that was retrieved from the outside world in the pack's *MsgBytes* attribute. For efficiency's sake, it's best to write directly into the already allocated memory rather than overwriting the attribute with a *[]byte* slice pointing to a new array. Overwriting the array is likely to lead to a lot of garbage collector churn.

The third step involves the input plugin deciding where next to pass the *PipelinePack* and then doing so. Once the *MsgBytes* attribute has been set the pack will typically be passed on to a decoder plugin, which will convert the raw bytes into a *Message* object, also an attribute of the *PipelinePack*. An input can gain access to the decoders that are available by calling *PluginHelper.DecoderSet()*, which can be used to access decoders either by the name they have been registered as in the config, or by the Heka protocol's encoding header they have been specified as decoding.

It is up to the input to decide which decoder should be used. Once the decoder has been determined and fetched from the *DecoderSet* the input should call *decoder.InChan()* to fetch the input channel upon which the *PipelinePack* can be placed.

Sometimes the input itself might wish to decode the data, rather than delegating that job to a separate decoder. In this case the input can directly populate the *pack.Message* and set the *pack.Decoded* value as *true*, as a decoder would do. Decoded messages are then injected into Heka's routing system by calling *InputRunner.Inject(pack)*. The message will then be delivered to the appropriate filter and output plugins.

One final important detail: if for any reason your input plugin should pull a *PipelinePack* off of the input channel and *not* end up passing it on to another step in the pipeline (i.e. to a decoder or to the router), you *must* call *PipelinePack.Recycle()* to free the pack up to be used again. Failure to do so will cause the *PipelinePack* pool to be depleted and will cause Heka to freeze.

4.7 Decoders

Decoder plugins are responsible for converting raw bytes containing message data into actual *Message* struct objects that the Heka pipeline can process. As with inputs, the *Decoder* interface is quite simple:

```
type Decoder interface {
        Decode(pack *PipelinePack) error
}
```

There are two optional Decoder interfaces. The first provides the Decoder access to its DecoderRunner object when it is started.

type WantsDecoderRunner interface { SetDecoderRunner(dr DecoderRunner)

}

The second provides a notification to the Decoder when the DecoderRunner is exiting.

type WantsDecoderRunnerShutdown interface { Shutdown()

}

A decoder's *Decode* method should extract the raw message data from *pack.MsgBytes* and attempt to deserialize this and use the contained information to populate the Message struct pointed to by the *pack.Message* attribute. Again, to minimize GC churn, take care to reuse the already allocated memory rather than creating new objects and overwriting the existing ones.

If the message bytes are decoded successfully then *Decode* should return *nil*. If not, then an appropriate error should be returned, in which case the error message will be logged and the message will be dropped, no further pipeline processing will occur.

4.8 Filters

Filter plugins are the message processing engine of the Heka system. They are used to examine and process message contents, and trigger events based on those contents in real time as messages are flowing through the Heka system.

The filter plugin interface is just a single method:

```
type Filter interface {
        Run(r FilterRunner, h PluginHelper) (err error)
}
```

Like input plugins, filters have a *Run* method which accepts a runner and a helper, and which should not return until shutdown unless there's an error condition. And like input plugins, filters should call *runner.InChan()* to gain access to the plugin's input channel.

The similarities end there, however. A filter's input channel provides pointers to *PipelinePack* objects, defined in pipeline_runner.go

The *Pack* contains a fully decoded *Message* object from which the filter can extract any desired information.

Upon processing a message, a filter plugin can perform any of three tasks:

1. Pass the original message through unchanged to one or more specific alternative Heka filter or output plugins.
2. Generate one or more *new* messages, which can be passed to either a specific set of Heka plugins, or which can be handed back to the router to be checked against all registered plugins' *message_matcher* rules.
3. Nothing (e.g. when performing counting / aggregation / roll-ups).

To pass a message through unchanged, a filter can call *PluginHelper.Filter()* or *PluginHelper.Output()* to access a filter or output plugin, and then call that plugin's *Deliver()* method, passing in the *PipelinePack*.

To generate new messages, your filter must call *PluginHelper.PipelinePack(msgLoopCount int)*. The *msgloopCount* value to be passed in should be obtained from the *MsgLoopCount* value on the *PipelinePack* that you're already holding, or possibly zero if the new message is being triggered by a timed ticker instead of an incoming message. The *PipelinePack* method will either return a pack ready for you to populate or *nil* if the loop count is greater than the configured maximum value, as a safeguard against inadvertently creating infinite message loops.

Once a *PipelinePack* has been obtained, a filter plugin can populate its *Message* object. The pack can then be passed along to a specific plugin (or plugins) as above. Alternatively, the pack can be injected into the Heka message router queue, where it will be checked against all plugin message matchers, by passing it to the *FilterRunner.Inject(pack *PipelinePack)* method. Note that, again as a precaution against message looping, a plugin will not be allowed to inject a message which would get a positive response from that plugin's own matcher.

Sometimes a filter will take a specific action triggered by a single incoming message. There are many cases, however, when a filter is merely collecting or aggregating data from the incoming messages, and instead will be sending out reports on the data that has been collected at specific intervals. Heka has built-in support for this use case. Any filter

(or output) plugin can include a *ticker_interval* config setting (in seconds, integers only), which will automatically be extracted by Heka when the configuration is loaded. Then from within your plugin code you can call *FilterRunner.Ticker()* and you will get a channel (type *<-chan time.Time*) that will send a tick at the specified interval. Your plugin code can listen on the ticker channel and take action as needed.

Observant readers might have noticed that, unlike the *Input* interface, filters don't need to implement a *Stop* method. Instead, Heka will communicate a shutdown event to filter plugins by closing the input channel from which the filter is receiving the *PipelinePack* objects. When this channel is closed, a filter should perform any necessary clean-up and then return from the *Run* method with a *nil* value to indicate a clean exit.

Finally, there is one very important point that all authors of filter plugins should keep in mind: if you are *not* passing your received *PipelinePack* object on to another filter or output plugin for further processing, then you *must* call *PipelinePack.Recycle()* to tell Heka that you are through with the pack. Failure to do so will cause Heka to not free up the packs for reuse, exhausting the supply and eventually causing the entire pipeline to freeze.

4.9 Outputs

Finally we come to the output plugins, which are responsible for receiving Heka messages and using them to generate interactions with the outside world. The *Output* interface is nearly identical to the *Filter* interface:

```
type Output interface {
        Run(or OutputRunner, h PluginHelper) (err error)
}
```

In fact, there is very little difference between filter and output plugins, other than tasks that they will be performing. Like filters, outputs should call the *InChan* method on the provided runner to get an input channel, which will feed *PipelinePack* objects. Like filters, outputs should listen on this channel until it is closed, at which time they should perform any necessary clean-up and thenreturn. And, like filters, any output plugin with a *ticker_interval* value in the configuration will use that value to create a ticker channel that can be accessed using the runner's *Ticker* method. And, finally, outputs should also be sure to call *PipelinePack.Recycle()* when they finish w/ a pack so that Heka knows the pack is freed up for reuse.

4.10 Registering Your Plugin

The last step you have to take after implementing your plugin is to register it with *hekad* so it can actually be configured and used. You do this by calling the *pipeline* package's *RegisterPlugin* function:

```
func RegisterPlugin(name string, factory func() interface{})
```

The *name* value should be a unique identifier for your plugin, and it should end in one of "Input", "Decoder", "Filter", or "Output", depending on the plugin type.

The *factory* value should be a function that returns an instance of your plugin, usually a pointer to a struct, where the pointer type implements the *Plugin* interface and the interface appropriate to its type (i.e. *Input*, *Decoder*, *Filter*, or *Output*).

This sounds more complicated than it is. Here are some examples from Heka itself:

```
RegisterPlugin("UdpInput", func() interface{} {return new(UdpInput)})
RegisterPlugin("TcpInput", func() interface{} {return new(TcpInput)})
RegisterPlugin("ProtobufDecoder", func() interface{} {return new(ProtobufDecoder)})
RegisterPlugin("CounterFilter", func() interface{} {return new(CounterFilter)})
RegisterPlugin("StatFilter", func() interface{} {return new(StatFilter)})
RegisterPlugin("LogOutput", func() interface{} {return new(LogOutput)})
RegisterPlugin("FileOutput", func() interface{} {return new(FileOutput)})
```

It is recommended that *RegisterPlugin* calls be put in your Go package's init() function so that you can simply import your package when building *hekad* and the package's plugins will be registered and available for use in your Heka config file. This is made a bit easier if you use **'plugin_loader.cmake'_**, see *Building hekad with External Plugins*.

CHAPTER 5

Message Matcher Syntax

Message matching is done by the *hekad* router to choose an appropriate filter(s) to run. Every filter that matches will get a copy of the message.

5.1 Examples

- Type == “test” && Severity == 6
- (Severity == 7 || Payload == “Test Payload”) && Type == “test”
- Fields[foo] != “bar”
- Fields[foo][1][0] == ‘alternate’
- Fields[MyBool] == TRUE
- TRUE
- Fields[created] =~ /%TIMESTAMP%/

5.2 Relational Operators

- **==** equals
- **!=** not equals
- **>** greater than
- **>=** greater than equals
- **<** less than
- **<=** less than equals
- **=~** regular expression match
- **!~** regular expression negated match

5.3 Logical Operators

- Parentheses are used for grouping expressions
- **&&** and (higher precedence)
- || or

5.4 Boolean

- **TRUE**
- **FALSE**

5.5 Message Variables

- All message variables must be on the left hand side of the relational comparison
- **String**
  - **Uuid**
  - **Type**
  - **Logger**
  - **Payload**
  - **EnvVersion**
  - **Hostname**
- **Numeric**
  - **Timestamp**
  - **Severity**
  - **Pid**
- **Fields**
  - **Fields[_field_name_]** (shorthand for Field[_field_name_][0][0])
  - **Fields[_field_name_][_field_index_]** (shorthand for Field[_field_name_][_field_index_][0])
  - **Fields[_field_name_][_field_index_][_array_index_]**
  - If a field type is mis-match for the relational comparison, false will be returned i.e. Fields[foo] == 6 where 'foo' is a string

5.6 Quoted String

- single or double quoted strings are allowed
- must be placed on the right side of a relational comparison i.e. Type == 'test'

5.7 Regular Expression String

- enclosed by forward slashes
- must be placed on the right side of the relational comparison i.e. Type =~ /test/
- capture groups will be ignored

5.7.1 Regular Expression Helpers

Commonly used complex regular expressions are provide as template variables in the form of %TEMPLATE%.

i.e., Fields[created] =~ /%TIMESTAMP%/

Available templates - TIMESTAMP - matches most common date/time string formats

See Also:

Regular Expression re2 syntax

CHAPTER 6

Sandbox

Sandboxes are Heka plugins that are implemented in a sandboxed scripting language. They provide a dynamic and isolated execution environment for data parsing, transformation, and analysis. They allow real time access to data in production without jeopardizing the integrity or performance of the monitoring infrastructure and do not require Heka to be recompiled. This broadens the audience that the data can be exposed to and facilitates new uses of the data (i.e. debugging, monitoring, dynamic provisioning, SLA analysis, intrusion detection, ad-hoc reporting, etc.)

6.1 Features

- **dynamic loading**
  - SandboxFilters can be started/stopped on a self-service basis while Heka is running
  - SandboxDecoder can only be started/stopped on a Heka restart but no recompilation is required to add new functionality.
- small - memory requirements are about 16 KiB for a basic sandbox
- fast - microsecond execution times
- stateful - ability to resume where it left off after a restart/reboot
- isolated - failures are contained and malfunctioning sandboxes are terminated

6.2 Sandbox Decoder

The sandbox decoder provides an isolated execution environment for data parsing and complex transformations without the need to recompile Heka.

6.2.1 SandboxDecoder Settings

- **script_type (string):** The language the sandbox is written in. Currently the only valid option is 'lua'.
- **filename (string):** The path to the sandbox code; if specified as a relative path it will be appended to Heka's global base_dir.

- **memory_limit (uint):** The number of bytes the sandbox is allowed to consume before being terminated (max 8MiB, default max).
- **instruction_limit (uint):** The number of instructions the sandbox is allowed the execute during the process_message function before being terminated (max 1M, default max).
- **output_limit (uint):** The number of bytes the sandbox output buffer can hold before before being terminated (max 63KiB, default max). Anything less than 1KiB will default to 1KiB.
- **config (object):** A map of configuration variables available to the sandbox via read_config. The map consists of a string key with: string, bool, int64, or float64 values.

Example

```
[sql_decoder]
type = "SandboxDecoder"
script_type = "lua"
filename = "sql_decoder.lua"
```

6.3 Sandbox Filter

The sandbox filter provides an isolated execution environment for data analysis. The output generated by the sandbox is injected into the payload of a new message for further processing or to be output.

6.3.1 SandboxFilter Settings

- *Common Filter / Output Parameters*
- **script_type (string):** The language the sandbox is written in. Currently the only valid option is 'lua'.
- **filename (string):** For a static configuration this is the path to the sandbox code; if specified as a relative path it will be appended to Heka's global base_dir. The filename must be unique between static plugins, since the global data is preserved using this name. For a dynamic configuration the filename is ignored and the the physical location on disk is controlled by the SandboxManagerFilter.
- **preserve_data (bool):** True if the sandbox global data should be preserved/restored on Heka shutdown/startup. The preserved data is stored along side the sandbox code i.e. counter.lua.data so Heka must have read/write permissions to that directory.
- **memory_limit (uint):** The number of bytes the sandbox is allowed to consume before being terminated (max 8MiB, default 32767).
- **instruction_limit (uint):** The number of instructions the sandbox is allowed the execute during the process_message/timer_event functions before being terminated (max 1M, default 1000).
- **output_limit (uint):** The number of bytes the sandbox output buffer can hold before before being terminated (max 63KiB, default 1024). Anything less than 1KiB will default to 1KiB.
- **profile (bool):** When true a statistically significant number of ProcessMessage timings are immediately captured before reverting back to the regular sampling interval. The main purpose is for more accurate sandbox comparison/tuning/optimization.
- **config (object):** A map of configuration variables available to the sandbox via read_config. The map consists of a string key with: string, bool, int64, or float64 values.

Example

```
[hekabench_counter]
type = "SandboxFilter"
message_matcher = "Type == 'hekabench'"
ticker_interval = 1
script_type  = "lua"
filename = "counter.lua"
preserve_data = true
memory_limit = 32767
instruction_limit = 1000
output_limit = 1024
profile = false

[hekabench_counter.config]
rows = 1440
sec_per_row = 60
```

6.4 Sandbox Manager

The SandboxManagerFilter allows SandboxFilters to be dynamically started and stopped using a signed Heka message. The intent is to have one manager per access control group each with their own message signing key. Users in each group can submit a signed control message to manage any filters running under the associated manager. A signed message is not an enforced requirement but it is highly recommended in order to restrict access to this functionality.

6.4.1 SandboxManagerFilter Settings

- *Common Filter / Output Parameters*
- **working_directory (string):** The directory where the filter configurations, code, and states are preserved. The directory can be unique or shared between sandbox managers since the filter names are unique per manager. Defaults to a directory in ${BASE_DIR}/sbxmgrs with a name generated from the plugin name.
- **max_filters (uint):** The maximum number of filters this manager can run.

Example

```
[OpsSandboxManager]
type = "SandboxManagerFilter"
message_signer = "ops"
message_matcher = "Type == 'heka.control.sandbox'"
max_filters = 100
```

6.4.2 Control Message

The sandbox manager control message is a regular Heka message with the following variables set to the specified values.

Starting a SandboxFilter

- Type: "heka.control.sandbox"
- Payload: *sandbox code*
- Fields[action]: "load"
- Fields[config]: the TOML configuration for the SandboxFilter *SandboxFilter Settings*

Stopping a SandboxFilter

- Type: "heka.control.sandbox"
- Fields[action]: "unload"
- Fields[name]: The SandboxFilter name specified in the configuration

6.4.3 sbmgr

Sbmgr is a tool for managing (starting/stopping) sandbox filters by generating the control messages defined above.

Command Line Options

sbmgr [`-config` *config_file*] [`-action` *load|unload*] [`-filtername` *specified on unload*] [`-script` *sandbox script filename*] [`-scriptconfig` *sandbox script configuration filename*]

6.4.4 sbmgrload

Sbmgrload is a test tool for starting/stopping a large number of sandboxes. The script and configuration are built into the tool and the filters will be named: CounterSandbox**N** where **N** is the instance number.

Command Line Options

sbmgrload [`-config` *config_file*] [`-action` *load|unload*] [`-num` *number of sandbox instances*]

Configuration Variables

- ip_address (string): IP address of the Heka server.
- **signer (object): Signer information for the encoder.**
  - name (string): The name of the signer.
  - hmac_hash (string): md5 or sha1
  - hmac_key (string): The key the message will be signed with.
  - version (int): The version number of the hmac_key.

Example

```
ip_address          = "127.0.0.1:5565"
[signer]
    name         = "test"
    hmac_hash    = "md5"
    hmac_key     = "4865ey9urgkidls xtb0[7lf9rzcivthkm"
    version      = 0
```

6.5 Tutorial - How to use the dynamic sandboxes

6.5.1 SandboxManager/Heka Setup

1. The SandboxManagerFilters are defined in the hekad configuration file and are created when hekad starts. The manager provides a location/namespace for SandboxFilters to run and controls access to this space via a signed Heka message. By associating a message_signer with the manager we can restrict who can load and unload the associated filters. Lets start by configuring a SandboxManager for a specific set of users; platform developers. Choose a unique filter name [PlatformDevs] and a signer name "PlatformDevs", in this case we will use the same name for each.

```
[PlatformDevs]
type = "SandboxManagerFilter"
message_signer = "PlatformDevs"
message_matcher = "Type == 'heka.control.sandbox'"
working_directory = "/var/heka/sandbox"
max_filters = 100
```

2. Configure the input that will receive the SandboxManager control messages. For this setup we will extend the current TCP input to handle our signed messages. The signer section consists of the signer name followed by an underscore and the key version number (the reason for this notation is to simply flatten the signer configuration structure into a single map). Multiple key versions are allowed to be active at the same time facilitating the rollout of new keys.

```
[TCP:5565]
type = "TcpInput"
address = ":5565"
    [TCP:5565.signer.PlatformDevs_0]
    hmac_key = "Old Platform devs signing key"
    [TCP:5565.signer.PlatformDevs_1]
    hmac_key = "Platform devs signing key"
```

3. Configure the sandbox manager utility (sbmgr). The signer information must exactly match the values in the input configuration above otherwise the messages will be discarded. Save the file as PlatformDevs.toml.

```
ip_address       = ":5565"
[signer]
    name         = "PlatformDevs"
    hmac_hash    = "md5"
    hmac_key     = "Platform devs signing key"
    version      = 1
```

6.5.2 SandboxFilter Setup

1. Create a SandboxFilter script and save it as "example.lua". See *lua_tutorials* for more detail.

```
data = circular_buffer.new(1440, 1, 60) -- message count per minute
local COUNT = data:set_header(1, "Messages", "count")
function process_message ()
    local ts = read_message("Timestamp")
    data:add(ts, COUNT, 1)
    return 0
end

function timer_event(ns)
    output(data)
    inject_message("cbuf")
end
```

2. Create the SandboxFilter configuration and save it as "example.toml".

The only difference between a static and dynamic SandboxFilter configuration is the filename. In the dynamic configuration it can be left blank or left out entirely. The manager will assign the filter a unique system wide name, in this case, "PlatformDevs-Example".

```
[Example]
type = "SandboxFilter"
message_matcher = "Type == 'Widget'"
ticker_interval = 60
```

```
script_type = "lua"
filename = ""
preserve_data = false
memory_limit = 64000
instruction_limit = 100
output_limit = 64000
```

3. Load the filter using sbmgr.

```
sbmgr -action=load -config=PlatformDevs.toml -script=example.lua -scriptconfig=example.toml
```

If you are running the *config_dashboard_output* the following links are available:

- Information about the running filters: http://localhost:4352/heka_report.html.
- Graphical Output (after 1 minute in this case): http://localhost:4352/PlatformDevs-Example.html

Otherwise

- Information about the terminated filters: http://localhost:4352/heka_sandbox_termination.html.

Note: A running filter cannot be 'reloaded' it must be unloaded and loaded again. The state is not preserved in this case for two reasons (in the future we hope to remedy this):

1. During the unload/load process some data can be missed creating a small gap in the analysis causing anomalies and confusion.
2. The internal data representation may have changed and restoration may be problematic.

---

4. Unload the filter using sbmgr

```
sbmgr -action=unload -config=PlatformDevs.toml -filtername=Example
```

6.6 Lua Sandbox

The *Lua* sandbox provides full access to the Lua language in a sandboxed environment under *hekad* that enforces configurable restrictions.

See Also:

Lua Reference Manual

6.6.1 API

6.6.2 Functions that must be exposed from the Lua sandbox

int process_message() Called by Heka when a message is available to the sandbox. The instruction_limit configuration parameter is applied to this function call.

Arguments none

Return

- < 0 for non-fatal failure (increments ProcessMessageFailures)
- 0 for success
- > 0 for fatal error (terminates the sandbox)

timer_event(ns) Called by Heka when the ticker_interval expires. The instruction_limit configuration parameter is applied to this function call. This function is only required in SandboxFilters (SandboxDecoders do not support timer events).

Arguments

- ns (int64) current time in nanoseconds since the UNIX epoch

Return none

6.6.3 Heka functions that are exposed to the Lua sandbox

read_config(variableName) Provides access to the sandbox configuration variables.

Arguments

- variableName (string)

Return number, string, bool, nil depending on the type of variable requested

read_message(variableName, fieldIndex, arrayIndex) Provides access to the Heka message data.

Arguments

- **variableName (string)**
  - Uuid
  - Type
  - Logger
  - Payload
  - EnvVersion
  - Hostname
  - Timestamp
  - Severity
  - Pid
  - Fields[_name_]
- **fieldIndex (unsigned) only used in combination with the Fields variableName**
  - use to retrieve a specific instance of a repeated field _name_
- **arrayIndex (unsigned) only used in combination with the Fields variableName**
  - use to retrieve a specific element out of a field containing an array

Return number, string, bool, nil depending on the type of variable requested

output(arg0, arg1, ...argN) Appends data to the payload buffer, which cannot exceed the output_limit configuration parameter.

Arguments

- arg (number, string, bool, nil, table, circular_buffer) Lua variable or literal to be appended the output buffer

Return none

Notes

Outputting a Lua table will serialize it to JSON according to the following guidelines/restrictions:

- Tables cannot contain internal of circular references.
- **Keys starting with an underscore are considered private and will not be serialized.**
  - '_name' is a special private key that can be used to specify the the name of the top level JSON object, if not provided the default is 'table'.
- Arrays only use contiguous numeric keys starting with an index of 1. Private keys are the exception i.e. local a = {1,2,3,_name="my_name"} will be serialized as: `{"my_name":[1,2,3]}\n`
- Hashes only use string keys (numeric keys will not be quoted and the JSON output will be invalid). Note: the hash keys are output in an arbitrary order i.e. local a = {x = 1, y = 2} will be serialized as: `{"table":{"y":2,"x":1}}\n`.

In most cases circular buffers should be directly output using inject_message. However, in order to create graph annotations the annotation table has to be written to the output buffer followed by the circular buffer. The output function is the only way to combine this data before injection (use a unique payload_type when injecting a message with a non-standard circular buffer mashups).

inject_message(payload_type, payload_name) Creates a new Heka message using the contents of the output payload buffer and then clears the buffer. Two pieces of optional metadata are allowed and included as fields in the injected message i.e., Fields[payload_type] == 'csv' Fields[payload_name] == 'Android Usage Statistics'. The number of messages that may be injected by the process_message or timer_event functions are globally controlled by the hekad *Command Line Options*; if these values are exceeded the sandbox will be terminated.

Arguments

- payload_type (**optional, default "txt"** string) Describes the content type of the injected payload data.
- payload_name (**optional, default ""** string) Names the content to aid in downstream filtering.

Return none

inject_message(circular_buffer, payload_name) Creates a new Heka message placing the circular buffer output in the message payload (overwriting whatever is in the output buffer). The payload_type is set to the circular buffer output format string. i.e., Fields[payload_type] == 'cbuf'. The Fields[payload_name] is set to the provided payload_name.

Arguments

- circular_buffer (circular_buffer)
- payload_name (**optional, default ""** string) Names the content to aid in downstream filtering.

Return none

Notes

- injection limits are enforced as described above

inject_message(message_table) Creates a new Heka protocol buffer message using the contents of the specified Lua table (overwriting whatever is in the output buffer). Notes about message fields:

- Timestamp is automatically generated if one is not provided. Nanosecond since the UNIX epoch is the only valid format.
- UUID is automatically generated, anything provided by the user is ignored.
- Hostname and Logger are automatically set by the SandboxFilter and cannot be overridden.
- Type is prepended with "heka.sandbox." by the SandboxFilter to avoid data confusion/mis-representation.

- **Fields can be represented in multiple forms and support the following primitive types: string, double, bool. These co**
  - name=value i.e., foo="bar"; foo=1; foo=true
  - name={array} i.e., foo={"b", "a", "r"}
  - **name={object} i.e. foo={value=1, representation="s"}; foo={value={1010, 2200, 1567}, representation="ms"}**
    - value (required) may be a single value or an array of values
    - representation (optional) metadata for display and unit management

Arguments

- message_table A table with the proper message structure.

Return none

Notes

- injection limits are enforced as described above

require(libraryName) Loads optional sandbox libraries

Arguments

- **libraryName (string)**
  - **lpeg** loads the Lua Parsing Expression Grammar Library http://www.inf.puc-rio.br/~roberto/lpeg/lpeg.html
  - **cjson** loaded the cjson.safe module in a global cjson table, exposing the decoding functions only. http://www.kyne.com.au/~mark/software/lua-cjson-manual.html.

Return a table (which is also globally registered with the library name).

6.6.4 Sample Lua Message Structure

```
{
Uuid        = "data",               -- always ignored
Logger      = "nginx",              -- ignored in the SandboxFilter
Hostname    = "bogus.mozilla.com",  -- ignored in the SandboxFilter

Timestamp   = 1e9,
Type        = "TEST",               -- will become "heka.sandbox.TEST" in the SandboxFilter
Papload     = "Test Payload",
EnvVersion  = "0.8",
Pid         = 1234,
Severity    = 6,
Fields      = {
            http_status     = 200,
            request_size    = {value=1413, representation="B"}
            }
}
```

6.7 Lua Circular Buffer Library

The library is a sliding window time series data store and is implemented in the `circular_buffer` table.

6.7.1 Constructor

circular_buffer.**new**(rows, columns, seconds_per_row, enable_delta)

Arguments

- rows (unsigned) The number of rows in the buffer (must be > 1)
- columns (unsigned)The number of columns in the buffer (must be > 0)
- seconds_per_row (unsigned) The number of seconds each row represents (must be > 0).
- enable_delta (**optional, default false** bool) When true the changes made to the circular buffer between delta outputs are tracked.

Return A circular buffer object.

6.7.2 Methods

Note: All column arguments are 1 based. If the column is out of range for the configured circular buffer a fatal error is generated.

---

double **add**(nanoseconds, column, value)

Arguments

- nanosecond (unsigned) The number of nanosecond since the UNIX epoch. The value is used to determine which row is being operated on.
- column (unsigned) The column within the specified row to perform an add operation on.
- value (double) The value to be added to the specified row/column.

Return The value of the updated row/column or nil if the time was outside the range of the buffer.

double **set**(nanoseconds, column, value)

Arguments

- nanosecond (unsigned) The number of nanosecond since the UNIX epoch. The value is used to determine which row is being operated on.
- column (unsigned) The column within the specified row to perform a set operation on.
- value (double) The value to be overwritten at the specified row/column.

Return The value passed in or nil if the time was outside the range of the buffer.

double **get**(nanoseconds, column)

Arguments

- nanosecond (unsigned) The number of nanosecond since the UNIX epoch. The value is used to determine which row is being operated on.
- column (unsigned) The column within the specified row to retrieve the data from.

Return The value at the specifed row/column or nil if the time was outside the range of the buffer.

int **set_header**(column, name, unit, aggregation_method)

Arguments

- column (unsigned) The column number where the header information is applied.

- name (string) Descriptive name of the column (maximum 15 characters). Any non alpha numeric characters will be converted to underscores. (default: Column_N)
- unit (string - optional) The unit of measure (maximum 7 characters). Alpha numeric, '/', and '*' characters are allowed everything else will be converted to underscores. i.e. KiB, Hz, m/s (default: count)
- **aggregation_method (string - optional) Controls how the column data is aggregated when combining multiple ci**
  - **sum** The total is computed for the time/column (default).
  - **min** The smallest value is retained for the time/column.
  - **max** The largest value is retained for the time/column.
  - **avg** The average is computed for the time/column.
  - **none** No aggregation will be performed the column.

Return The column number passed into the function.

double **compute**(function, column, start, end)

Arguments

- function (string) The name of the compute function (sum|avg|sd|min|max).
- column (unsigned) The column that the computation is performed against.
- start (optional - unsigned) The number of nanosecond since the UNIX epoch. Sets the start time of the computation range; if nil the buffer's start time is used.
- end (optional- unsigned) The number of nanosecond since the UNIX epoch. Sets the end time of the computation range (inclusive); if nil the buffer's end time is used. The end time must be greater than or equal to the start time.

Return The result of the computation for the specifed column over the given range or nil if the range fell outside of the buffer.

cbuf format(format) Sets an internal flag to control the output format of the circular buffer data structure; if deltas are not enabled or there haven't been any modifications, nothing is output.

Arguments

- **format (string)**
  - **cbuf** The circular buffer full data set format.
  - **cbufd** The circular buffer delta data set format.

Return The circular buffer object.

6.7.3 Output

The circular buffer can be passed to the output() function. The output format can be selected using the format() function.

The cbuf (full data set) output format consists of newline delimited rows starting with a json header row followed by the data rows with tab delimited columns. The time in the header corresponds to the time of the first data row, the time for the other rows is calculated using the seconds_per_row header value.

```
{json header}
row1_col1\trow1_col2\n
.
.
.
rowN_col1\trowN_col2\n
```

The cbufd (delta) output format consists of newline delimited rows starting with a json header row followed by the data rows with tab delimited columns. The first column is the timestamp for the row (time_t). The cbufd output will only contain the rows that have changed and the corresponding delta values for each column.

```
{json header}
row14_timestamp\trow14_col1\trow14_col2\n
row10_timestamp\trow10_col1\trow10_col2\n
```

6.7.4 Sample Cbuf Output

```
{"time":2,"rows":3,"columns":3,"seconds_per_row":60,"column_info":[{"name":"HTTP_200","unit":"count",
10002   0   0
11323   0   0
10685   0   0
```

6.7.5 Example

```
-- This Source Code Form is subject to the terms of the Mozilla Public
-- License, v. 2.0. If a copy of the MPL was not distributed with this
-- file, You can obtain one at http://mozilla.org/MPL/2.0/.

data = circular_buffer.new(1440, 5, 60) -- 1 day at 1 minute resolution
local HTTP_200      = data:set_header(1, "HTTP_200"     , "count")
local HTTP_300      = data:set_header(2, "HTTP_300"     , "count")
local HTTP_400      = data:set_header(3, "HTTP_400"     , "count")
local HTTP_500      = data:set_header(4, "HTTP_500"     , "count")
local HTTP_UNKNOWN  = data:set_header(5, "HTTP_UNKNOWN" , "count")

function process_message()
    local ts = read_message("Timestamp")
    local sc = read_message("Fields[http_status_code]")
    if sc == nil then return 0 end

    if sc >= 200 and sc < 300 then
        data:add(ts, HTTP_200, 1)
    elseif sc >= 300 and sc < 400 then
        data:add(ts, HTTP_300, 1)
    elseif sc >= 400 and sc < 500 then
        data:add(ts, HTTP_400, 1)
    elseif sc >= 500 and sc < 600 then
        data:add(ts, HTTP_500, 1)
    else
        data:add(ts, HTTP_UNKNOWN, 1)
    end
    return 0
end

function timer_event()
```

```
    output(data)
    inject_message("cbuf", "HTTP Status Code Statistics")
end
```

Setting the inject_message payload_type to "cbuf" will cause the *config_dashboard_output* to automatically generate an HTML page containing a graphical view of the data.

6.8 Lua Sandbox Tutorial

6.8.1 How to create a simple sandbox filter

1. Implement the required Heka interface in Lua

```
function process_message ()
    return 0
end

function timer_event(ns)
end
```

2. Add the business logic (count the number of 'demo' events per minute)

```
total = 0 -- preserved between restarts since it is in global scope
local count = 0 -- local scope so this will not be preserved

function process_message()
    total= total + 1
    count = count + 1
    return 0
end

function timer_event(ns)
    output(string.format("%d messages in the last minute; total=%d", count, total))
    count = 0
    inject_message()
end
```

3. Setup the configuration

```
[demo_counter]
type = "SandboxFilter"
message_matcher = "Type == 'demo'"
ticker_interval = 60
script_type = "lua"
filename = "counter.lua"
preserve_data = true
memory_limit = 32767
instruction_limit = 100
output_limit = 256
```

4. Extending the business logic (count the number of 'demo' events per minute per device)

```
device_counters = {}

function process_message()
    local device_name = read_message("Fields[DeviceName]")
    if device_name == nil then
```

```
        device_name = "_unknown_"
    end

    local dc = device_counters[device_name]
    if dc == nil then
        dc = {count = 1, total = 1}
        device_counters[device_name] = dc
    else
        dc.count = dc.count + 1
        dc.total = dc.total + 1
    end
    return 0
end

function timer_event(ns)
    output("#device_name\tcount\ttotal\n")
    for k, v in pairs(device_counters) do
        output(string.format("%s\t%d\t%d\n", k, v.count, v.total))
        v.count = 0
    end
    inject_message()
end
```

5. Depending on the number of devices being counted you will most likely want to update the configuration to account for the additional resource requirements.

```
memory_limit = 65536
instruction_limit = 20000
output_limit = 64512
```

CHAPTER 7

Testing Heka

7.1 Flood

Flood is a Heka load test tool; it is capable of generating a large number of messages to exercise Heka using different protocols, message types, and error conditions.

7.1.1 Command Line Options

flood [`-config` *config_file*] [`-test` *config_section_name*]

7.1.2 Configuration Variables

- test (object): Name of the test section (toml key) in the configuration file.
- ip_address (string): IP address of the Heka server.
- sender (string): tcp or udp
- pprof_file (string): The name of the file to save the profiling data to.
- encoder (string): protobuf or json
- num_messages (int): The number of messages to be sent, 0 for infinite.
- corrupt_percentage (float): The percentage of messages that will be randomly corrupted.
- signed_percentage (float): The percentage of message that will signed.
- variable_size_messages (bool): True, if a random selection of variable size messages are to be sent. False, if a single fixed message will be sent.
- **signer (object): Signer information for the encoder.**
  - name (string): The name of the signer.
  - hmac_hash (string): md5 or sha1
  - hmac_key (string): The key the message will be signed with.
  - version (int): The version number of the hmac_key.

- ascii_only (bool): True, if generated message payloads should only contain ASCII characters. False, if message payloads should contain arbitrary binary data. Defaults to false.

Example

```
[default]
ip_address          = "127.0.0.1:5565"
sender              = "tcp"
pprof_file          = ""
encoder             = "protobuf"
num_messages        = 0
corrupt_percentage  = 0.0001
signed_percentage   = 0.00011
variable_size_messages = true
[default.signer]
    name            = "test"
    hmac_hash       = "md5"
    hmac_key        = "4865ey9urgkidls xtb0[7lf9rzcivthkm"
    version          = 0
```

CHAPTER 8

Indices and tables

- *search*
- *glossary*