Heka Documentation

Release 0.10.0b0

Mozilla

July 13, 2015

Contents

Heka is an open source stream processing software system developed by Mozilla. Heka is a "Swiss Army Knife" type tool for data processing, useful for a wide variety of different tasks, such as:

- Loading and parsing log files from a file system.
- Accepting statsd type metrics data for aggregation and forwarding to upstream time series data stores such as graphite or InfluxDB.
- Launching external processes to gather operational data from the local system.
- Performing real time analysis, graphing, and anomaly detection on any data flowing through the Heka pipeline.
- Shipping data from one location to another via the use of an external transport (such as AMQP) or directly (via TCP).
- Delivering processed data to one or more persistent data stores.

The following resources are available to those who would like to ask questions, report problems, or learn more:

- Mailing List: https://mail.mozilla.org/listinfo/heka
- Issue Tracker: https://github.com/mozilla-services/heka/issues
- Github Project: https://github.com/mozilla-services/heka/
- IRC: #heka channel on irc.mozilla.org

Heka is a heavily plugin based system. Common operations such as adding data to Heka, processing it, and writing it out are implemented as plugins. Heka ships with numerous plugins for performing common tasks.

There are six different types of Heka plugins:

Inputs

Input plugins acquire data from the outside world and inject it into the Heka pipeline. They can do this by reading files from a file system, actively making network connections to acquire data from remote servers, listening on a network socket for external actors to push data in, launching processes on the local system to gather arbitrary data, or any other mechanism.

Input plugins must be written in Go.

Splitters

Splitter plugins receive the data that is being acquired by an input plugin and slice it up into individual records. They must be written in Go.

Decoders

Decoder plugins convert data that comes in through the Input plugins to Heka's internal Message data structure. Typically decoders are responsible for any parsing, deserializing, or extracting of structure from unstructured data that needs to happen.

Decoder plugins can be written entirely in Go, or the core logic can be written in sandboxed Lua code.

Filters

Filter plugins are Heka's processing engines. They are configured to receive messages matching certain specific characteristics (using Heka's *Message Matcher Syntax*) and are able to perform arbitrary monitoring, aggregation, and/or processing of the data. Filters are also able to generate new messages that can be reinjected into the Heka pipeline, such as summary messages containing aggregate data, notification messages in cases where suspicious anomalies are detected, or circular buffer data messages that will show up as real time graphs in Heka's dashboard.

Filters can be written entirely in Go, or the core logic can be written in sandboxed Lua code. It is also possible to configure Heka to allow Lua filters to be dynamically injected into a running Heka instance

without needing to reconfigure or restart the Heka process, nor even to have shell access to the server on which Heka is running.

Encoders

Encoder plugins are the inverse of Decoders. They generate arbitrary byte streams using data extracted from Heka Message structs. Encoders are embedded within Output plugins; Encoders handle the serialization, Outputs handle the details of interacting with the outside world.

Encoder plugins can be written entirely in Go, or the core logic can be written in sandboxed Lua code.

Outputs

Output plugins send data that has been serialized by an Encoder to some external destination. They handle all of the details of interacting with the network, filesystem, or any other outside resource. They are, like Filters, configured using Heka's *Message Matcher Syntax* so they will only receive and deliver messages matching certain characteristics.

Output plugins must be written in Go.

Information about developing plugins in Go can be found in the *Extending Heka* section. Details about using Lua sandboxes for Decoder, Filter, and Encoder plugins can be found in the *Sandbox* section.

CHAPTER 1

hekad

The core of the Heka system is the *hekad* daemon. A single *hekad* process can be configured with any number of plugins, simultaneously performing a variety of data gathering, processing, and shipping tasks. Details on how to configure a *hekad* daemon are in the *Configuring hekad* section.

CHAPTER 2

hekad Command Line Options

-version Output the version number, then exit.

-config ***config_path*** Specify the configuration file or directory to use; the default is /etc/hekad.toml. If *config_path* resolves to a directory, all files in that directory must be valid TOML files. (See hekad.config(5).)

Contents:

2.1 Installing

2.1.1 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.

2.1.2 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 sourced 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 3.0.0 or greater http://www.cmake.org/cmake/resources/software.html
- Git http://git-scm.com/download
- Go 1.4 or greater http://golang.org/dl/
- Mercurial http://mercurial.selenic.com/wiki/Download
- 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/
- An internet connection to fetch sub modules

Prerequisites (Unix):

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

Prerequisites (Windows):

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

Build Instructions

1. Check out the *heka* repository:

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

2. Source (Unix-y) or run (Windows) the build script in the heka directory:

```
cd heka
source build.sh # Unix (or `. build.sh`; 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.

---

4. Run `make install` to install libs and modules into a usable location:

```
make install         # Unix
mingw32-make install # Windows
```

   This will install all of Heka's required support libraries, modules, and other files into a usable `share_dir`, at the following path:

```
/path/to/heka/repo/heka/share/heka
```

5. Specify Heka configuration:

   When setting up your Heka configuration, you'll want to make sure you set the global `share_dir` setting to point to the path above. The `[hekad]` section might look like this:

```
[hekad]
maxprocs = 4
share_dir = "/path/to/heka/repo/heka/share/heka"
```

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.

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.

2.1.3 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, svn, or hg), 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 6fe574dbd32a21f5
add_external_plugin(git https://github.com/example/path <tag> util filepath)
add_external_plugin(git https://github.com/bellycard/heka-sns-input :local)
# The ':local' tag is a special case, it copies {heka root}/externals/{plugin_name} into the Go
# work environment every time `make` is run. When local development is complete, and the source
# is checked in, the value can simply be changed to the correct tag to make it 'live'.
# i.e. {heka root}/externals/heka-sns-input -> {heka root}/build/heka/src/github.com/bellycard/h
```

The preceeding entry clones the *heka-mozsvc-plugins* git repository into the Go work environment, checks out SHA 6fe574dbd32a21f5d5583608a9d2339925edd2a7, 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.

2.1.4 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 (no deb, see below)
make deb             # Unix (if dpkg is available see below)
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

Note: For file name convention reasons, deb packages won't be created by running *cpack* or *make package*, even on a Unix machine w/ dpkg installed. Instead, running *source build.sh* on such a machine will generate a Makefile with a separate 'deb' target, so you can run *make deb* to generate the appropriate deb package.

2.2 Getting Started

A brand new Heka installation is something of a blank canvas, full of promise but not actually interesting on its own. One of the challenges with a highly flexible tool like Heka is that newcomers can easily become overwhelmed by the wide assortment of features and options, making it difficult to understand exactly how to begin. This document will try to address this issue by taking readers through the process of configuring a hekad installation that demonstrates a number of Heka's common use cases, hopefully providing enough context that users will be able to then adjust and extend the given examples to meet their own particular needs.

When we're done our configuration will have Heka performing the following tasks:

- Accepting data from a statsd client over UDP.
- Forwarding aggregated statsd data on to both a Graphite Carbon server and an InfluxDB server.
- Generating a real time graph of a specific set of statsd statistics.
- Loading and parsing a rotating stream of nginx access log files.
- Generating JSON structures representing each request loaded from the Nginx log files and sending them on to an ElasticSearch database cluster.
- Generating a real time graph of the HTTP response status codes of the requests that were recorded in the nginx access logs.
- Performing basic algorithmic anomaly detection on HTTP status code data, sending notification messages via email when such events occur.

But before we dig in to that, let's make sure everything is working by trying out a very simple setup.

2.2.1 Simplest Heka Config

One of the simplest Heka configurations possible is one that loads a single file from the local file system and then outputs the contents of that file to stdout. The following is an example of such a configuration:

```
[LogstreamerInput]
log_directory = "/var/log"
file_match = 'auth\.log'

[PayloadEncoder]
append_newlines = false

[LogOutput]
message_matcher = "TRUE"
encoder = "PayloadEncoder"
```

Heka is configured via one or more TOML format configuration files, each of which is comprised of one or more sections. The configuration above consists of three sections, the first of which specifies a LogstreamerInput, Heka's primary mechanism for loading files from the local file system. This one is loading */var/log/auth.log*, but you can change this to load any other file by editing the *log_directory* setting to point to the folder where the file lives and the *file_match* setting to a regular expression that uniquely matches the filename. Note the single quotes (*'auth\.log'*) around the regular expression; this is TOML's way of specifying a raw string, which means we don't need to escape the regular expression's backslashes like we would with a regular string enclosed by double quotes (*"auth\\.log"*).

In most real world cases a LogstreamerInput would include a *decoder* setting, which would parse the contents of the file to extract data from the text format and map them onto a Heka message schema. In this case, however, we stick with the default behavior, where Heka creates a new message for each line in the log file, storing the text of the log line as the payload of the Heka message.

The next two sections tell Heka what to do with the messages that the LogstreamerInput is generating. The LogOutput simply writes data out to the Heka process's stdout. We set *message_matcher = "TRUE"* to specify that this output should capture every single message that flows through the Heka pipeline. The *encoder* setting tells Heka to use the PayloadEncoder that we've configured, which extracts the payload from each captured message and uses that as the raw data that the output will send.

To see whether or not you have a functional Heka system, you can create a file called *sanity_check.toml* and paste in the above configuration, adjusting the LogstreamerInput's settings to point to another file if necessary. Then you can run Heka using *hekad -config=/path/to/sanity_check.toml*, and you should see the contents of the log file printed out to the console. If any new lines are written to the log file that you're loading, Heka will notice and will write them out to stdout in real time.

Note that the LogstreamerInput keeps track of how far it has gotten in a particular file, so if you stop Heka using *ctrl-c* and then restart it you will not see the same data. Heka stores the current location in a "seekjournal" file, at */var/cache/hekad/logstreamer/LogstreamerInput* by default. If you delete this file and then restart Heka you should see it load the entire file from the beginning again.

Congratulations! You've now successfully run Heka with a full, working configuration. But clearly there are much simpler tools to use if all you want to do is write the contents of a log file out to stdout. Now that we've got an initial success under our belt, let's take a deeper dive into a much more complex Heka configuration that actually handles multiple real world use cases.

2.2.2 Global Configuration

As mentioned above, Heka is configured using TOML configuration files. Most sections of the TOML configuration contain information relevant to one of Heka's plugins, but there is one section entitled *hekad* which allows you to tweak a number of Heka's *global configuration options*. In many cases the defaults for most of these options will suffice, and your configuration won't need a *hekad* section at all. A few of the options are worth looking at here, however:

- **maxprocs (int, default 1):** This setting corresponds to Go's GOMAXPROCS environment variable. It specifies how many CPU cores the hekad process will be allowed to use. The best choice for this setting depends on a number of factors such as the volume of data Heka will be processing, the number of cores on the machine on which Heka is running, and what other tasks the machine will be performing. For dedicated Heka aggregator machines, this should usually be equal to the number of cpu cores available, or perhaps number of cores minus one, while for Heka processes running on otherwise busy boxes one or two is probably a better choice.
- **base_dir (string, default '/var/cache/hekad' or 'c:\var\cache\hekad'):** In addition to the location of the configuration files, there are two directories that are important to a running *hekad* process. The first of these is called the *base_dir*, which is a working directory where Heka will be storing information crucial to its functioning, such as seekjournal files to track current location in a log stream, or sandbox filter aggregation data that is meant to survive between Heka restarts. It is of course important that the user under which the *hekad* process is running has write access to the base_dir.

- **share_dir (string, default '/usr/share/heka' or 'c:\usr\share\heka'):** The second directory important to Heka's functioning is called the *share_dir*. This is a place where Heka expects to find certain static resources that it needs, such as the HTML/javascript source code used by the dashboard output, or the source code to various Lua based plugins. The user owning the *hekad* process requires read access to this folder, but should not have write access.

It's worth noting that while Heka defaults to expecting to find certain resources in the *base_dir* and/or the *share_dir* folders, it is nearly always possible to override the location of a particular resource on a case by case basis in the plugin configuration. For instance, the *filename* option in a SandboxFilter specifies the filesystem path to the Lua source code for that filter. If it is specified as a relative path, the path will be computed relative to the *share_dir*. If it is specified as an absolute path, the absolute path will be honored.

For our example, we're going to keep the defaults for most global options, but we'll bump the *maxprocs* setting from 1 to 2 so we can get at least some parallel behavior:

```
[hekad]
maxprocs = 2
```

2.2.3 Accepting Statsd Data

Once we've got Heka's global settings configured, we're ready to start on the plugins. The first thing we'll tackle is getting Heka set up to accept data from statsd clients. This involves two different plugins, a *Statsd Input* that accepts network connections and parses the received stats data, and a *Stat Accumulator Input* that will accept the data gathered by the StatsdInput, perform the necessary aggregation, and periodically generate 'statmetric' messages containing the aggregated data.

The configuration for these plugins is quite simple:

```
[StatsdInput]

[StatAccumInput]
ticker_interval = 1
emit_in_fields = true
```

These two TOML sections tell Heka that it should include a StatsdInput and a StatAccumInput. The StatsdInput uses the default value for every configuration setting, while the StatAccumInput overrides the defaults for two of its settings. The *ticker_interval = 1* setting means that the statmetric messages will be generated once every second instead of the default of once every five seconds, while the *emit_in_fields = true* setting means that the aggregated stats data will be embedded in the dynamic fields of the generated statmetric messages, in addition to the default of embedding the graphite text format in the message payload.

This probably seems pretty straightforward, but there are actually some subtleties hidden in there that are important to point out. First, it's not immediately obvious, but there is an explicit connection between the two plugins. The StatsdInput has a *stat_accum_name* setting, which we didn't need to set because it defaults to 'StatAccumInput'. The following configuration is exactly equivalent:

```
[StatsdInput]
stat_accum_name = "StatAccumInput"

[StatAccumInput]
ticker_interval = 1
emit_in_fields = true
```

The next subtlety to note is that we've used a common piece of Heka config shorthand by embedding both the name *and* the type in the TOML section header. Heka lets you do this as a convenience if you don't need to use a name that is separate from the type. This doesn't have to be the case, it's possible to give a plugin a different name, expressing the type inside the TOML section instead of in its header:

```
[statsd_input]
type = "StatsdInput"
stat_accum_name = "stat_accumulator"

[stat_accumulator]
type = "StatAccumInput"
ticker_interval = 1
emit_in_fields = true
```

The config above is ever so slightly different from the original two, because our plugins now have different name identifiers, but functionally the behavior is identical to the prior versions. Being able to separate a plugin name from its type is important in cases where you want more than one instance of the same plugin type. For instance, you'd use the following configuration if you wanted to have a second StatsdInput listening on port 8126 in addition to the default on port 8125:

```
[statsd_input_8125]
type = "StatsdInput"
stat_accum_name = "stat_accumulator"

[statsd_input_8126]
type = "StatsdInput"
stat_accum_name = "stat_accumulator"
address = "127.0.0.1:8126"

[stat_accumulator]
type = "StatAccumInput"
ticker_interval = 1
emit_in_fields = true
```

We don't need two StatsdInputs for our example, however, so for simplicity we'll go with the most concise configuration.

2.2.4 Forwarding Aggregated Stats Data

Collecting stats alone doesn't actually provide much value, we want to be able to actually see the data that has been gathered. Statsd servers are typically used to aggregate incoming statistics and then periodically deliver the totals to an upstream time series database, usually Graphite, although InfluxDB is rapidly growing in popularity. For Heka to replace a standalone statsd server it needs to be able to do the same.

To understand how this will work, we need to step back a bit to look at how Heka handles message routing. First, data enters the Heka pipeline through an input plugin. Then it needs to be converted from its original raw format into a message object that Heka knows how to work with. Usually this is done with a decoder plugin, although in the statsd example above instead the StatAccumInput itself is periodically generating statmetric messages.

After the data has been marshaled into one (or more) message(s), the message is handed to Heka's internal message router. The message router will then iterate through all of the registered filter and output plugins to see which ones would like to process the message. Each filter and output provides a *message matcher* to specify which messages it would like to receive. The router hands each message to each message matcher, and if there's a match then the matcher in turn hands the message to the plugin.

To return to our example, we'll start by setting up a *Carbon Output* plugin that knows how to deliver messages to an upstream Graphite Carbon server. We'll configure it to receive the statmetric messages generated by the StatAccumInput:

```
[CarbonOutput]
message_matcher = "Type == 'heka.statmetric'"
```

```
address = "mycarbonserver.example.com:2003"
protocol = "udp"
```

Any messages that pass through the router with a Type field equal to *heka.statmetric* (which is what the StatAccumOutput emits by default) will be handed to this output, which will in turn deliver it over UDP to the specified carbon server address. This is simple, but it's a fundamental concept. Nearly all communication within Heka happens using Heka message objects being passed through the message router and being matched against the registered matchers.

Okay, so that gets us talking to Graphite. What about InfluxDB? InfluxDB has an extension that allows it to support the graphite format, so we could use that and just set up a second CarbonOutput:

```
[carbon]
type = "CarbonOutput"
message_matcher = "Type == 'heka.statmetric'"
address = "mycarbonserver.example.com:2003"
protocol = "udp"

[influx]
type = "CarbonOutput"
message_matcher = "Type == 'heka.statmetric'"
address = "myinfluxserver.example.com:2003"
protocol = "udp"
```

A couple of things to note here. First, don't get confused by the *type = "CarbonOutput"*, which is specifying the type of the **plugin** we are configuring, and the "Type" in *message_matcher = "Type == 'heka.statmetric'"*, which is referring to the Type field of the **messages** that are passing through the Heka router. They're both called "type", but other than that they are unrelated.

Second, you'll see that it's fine to have more than one output (and/or filter, for that matter) plugin with identical message_matcher settings. The router doesn't care, it will happily give the same message to both of them, and any others that happen to match.

This will work, but it'd be nice to just use the InfluxDB native HTTP API. For this, we can instead use our handy HttpOutput:

```
[CarbonOutput]
message_matcher = "Type == 'heka.statmetric'"
address = "mycarbonserver.example.com:2003"
protocol = "udp"

[statmetric_influx_encoder]
type = "SandboxEncoder"
filename = "lua_encoders/statmetric_influx.lua"

[influx]
type = "HttpOutput"
message_matcher = "Type == 'heka.statmetric'"
address = "http://myinfluxserver.example.com:8086/db/stats/series"
encoder = "statmetric_influx_encoder"
username = "influx_username"
password = "influx_password"
```

The HttpOutput configuration above will also capture statmetric messages, and will then deliver the data over HTTP to the specified address where InfluxDB is listening. But wait! what's all that *statmetric-influx-encoder* stuff? I'm glad you asked...

2.2.5 Encoder Plugins

We've already briefly mentioned how, on the way in, raw data needs to be converted into a standard message format that Heka's router, filters, and outputs are able to process. Similarly, on the way out, data must be extracted from the standard message format and serialized into whatever format is required by the destination. This is typically achieved through the use of encoder plugins, which take Heka messages as input and generate as output raw bytes that an output plugin can send over the wire. The CarbonOutput doesn't specify an encoder because it assumes that the Graphite data will be in the message payload, where the StatAccumInput puts it, but most outputs need an encoder to be specified so they know how to generate their data stream from the messages that are received.

In the InfluxDB example above, you can see that we've defined a *statmetric_influx_encoder*, of type SandboxEncoder. A "Sandbox" plugin is one where the core logic of the plugin is implemented in Lua and is run in a protected sandbox. Heka has support for *Sandbox Decoder*, *Sandbox Filter*, and *Sandbox Encoder* plugins. In this instance, we're using a SandboxEncoder implementation provided by Heka that knows how to extract data from the fields of a heka.statmetric message and use that data to generate JSON in a format that will be understood by InfluxDB (see *StatMetric InfluxDB Encoder*).

This separation of concerns between encoder and output plugins allows for a great deal of flexibility. It's easy to write your own SandboxEncoder plugins to generate any format needed, allowing the same HttpOutput implementation can be used for multiple HTTP-based back ends, rather than needing a separate output plugin for each service. Also, the same encoder can be used with different outputs. If, for instance, we wanted to write the InfluxDB formatted data to a file system file for later processing, we could use the statmetric_influx encoder with a FileOutput to do so.

2.2.6 Real Time Stats Graph

While both Graphite and InfluxDB provide mechanisms for displaying graphs of the stats data they receive, Heka is also able to provide graphs of this data directly. These graphs will be updated in real time, as the data is flowing through Heka, without the latency of the data store driven graphs. The following config snippet shows how this is done:

```
[stat_graph]
type = "SandboxFilter"
filename = "lua_filters/stat_graph.lua"
ticker_interval = 1
preserve_data = true
message_matcher = "Type == 'heka.statmetric'"

        [stat_graph.config]
        num_rows = 300
        secs_per_row = 1
        stats = "stats.counters.000000.count stats.counters.000001.count stats.counters.000002.count"
        stat_labels = "counter_0 counter_1 counter_2"
        preservation_version = 0

[DashboardOutput]
ticker_interval = 1
```

There's a lot going on in just a short bit of configuration here, so let's consider it one piece at a time to understand what's happening. First, we've got a *stat_graph* config section, which is telling Heka to start up a SandboxFilter plugin, a filter plugin with the processing code implemented in Lua. The *filename* option points to a filter implementation that ships with Heka. This *filter implementation* knows how to extract data from statmetric messages and store that data in a circular buffer data structure. The *preserve_data* option tells Heka that the all global data in this filter (the circular buffer data, in this case) should be flushed out to disk if Heka is shut down, so it can be reloaded again when Heka is restarted. And the *ticker_interval* option is specifying that our filter will be emitting an output message containing the cbuf data back into the router once every second. This message can then be consumed by other filters and/or outputs, such as our DashboardOutput which will use it to generate graphs (see next section).

After that we have a *stat_graph.config* section. This isn't specifying a new plugin, this is nested configuration, a subsection of the outer *stat_graph* section. (Note that the section nesting is specified by the use of the *stat_graph.* prefix in the section name; the indentation helps readability, but has no impact on the semantics of the configuration.) The *stat-graph* section configures the SandboxFilter and tells it what Lua source code to use, the *stat_graph.config* section is passed *in* to the Lua source code for further customization of the filter's behavior.

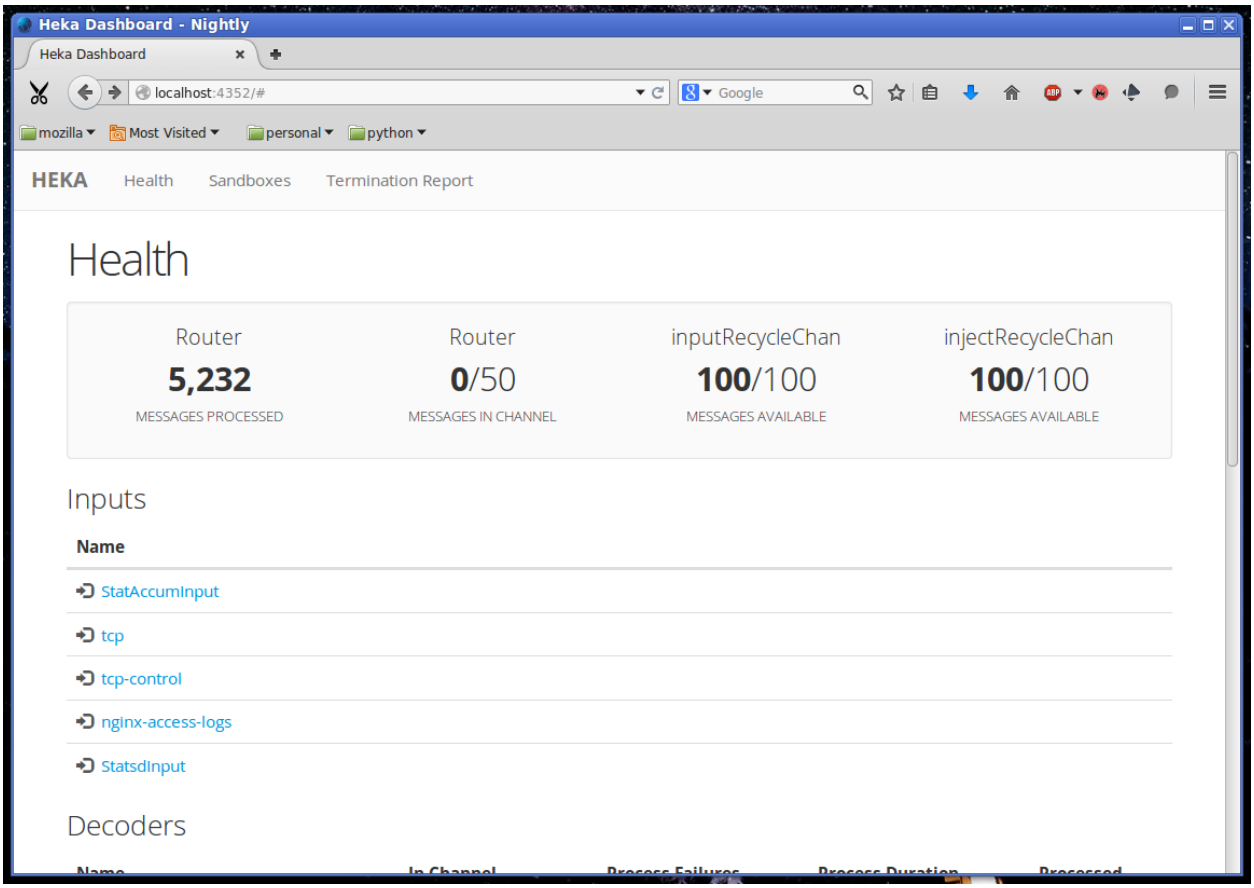
So what is contained in this nested configuration? The first two options, *num_rows* and *secs_per_row*, are configuring the circular buffer data structure that the filter will use to store the stats data. It can be helpful to think of circular buffer data structures as a spreadsheet. Our spreadsheet will have 300 rows, and each row will represent one second of accumulated data, so at any given time we will be holding five minutes worth of stats data in our filter. The next two options, *stats* and *stat_labels*, tell Heka which statistics we want to graph and provide shorter labels for use in the graph legend. Finally the *preservation_version* section allows us to version our data structures. This is needed because our data structures might change. If you let this filter run for a while, gathering data, and then shut down Heka, the 300 rows of circular buffer data will be written to disk. If you then change the *num_rows* setting and try to restart Heka the filter will fail to start, because the 300 row size of the preserved data won't match the new size that you've specified. In this case you would increment the *preservation_version* value from 0 to 1, which will tell Heka that the preserved data is no longer valid and the data structures should be created anew.

2.2.7 Heka Dashboard

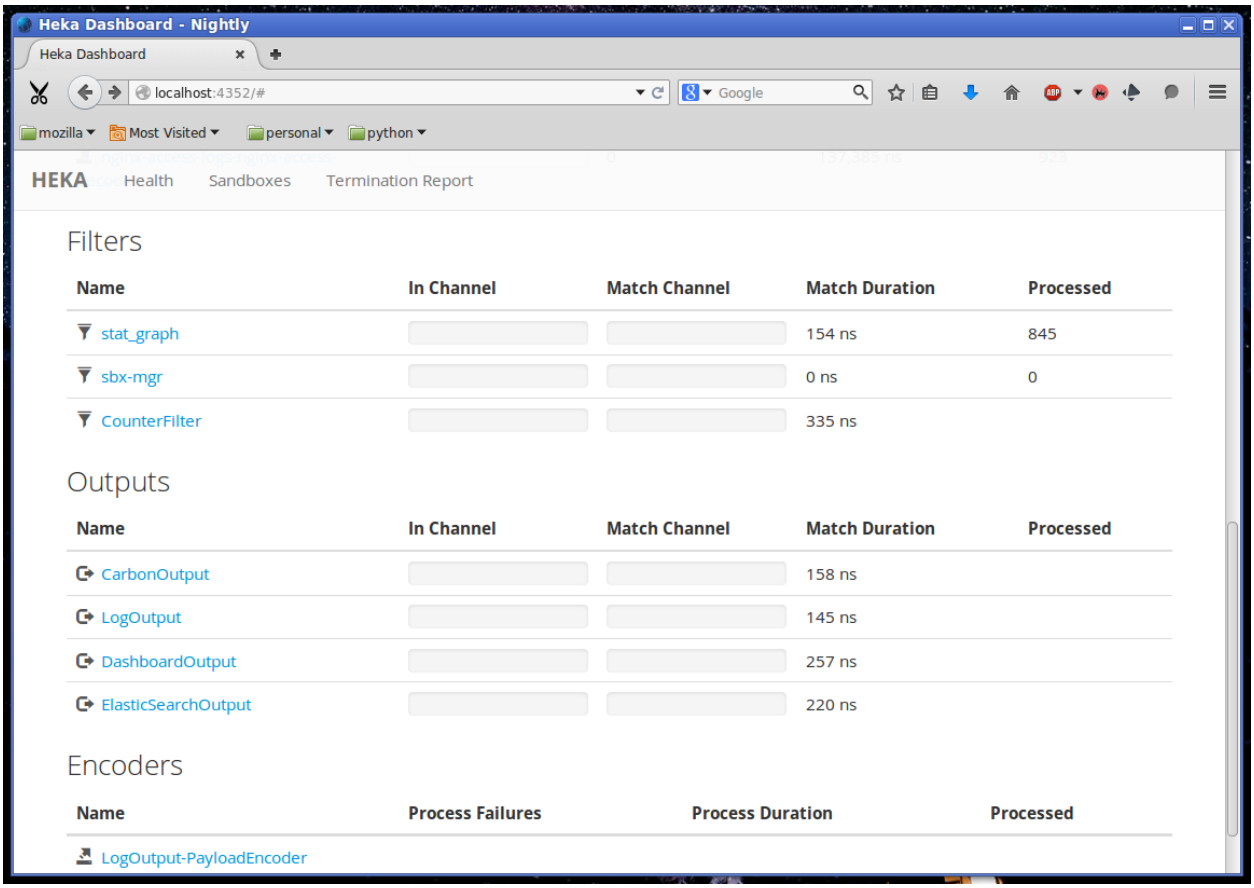
At this point it's useful to notice that, while the SandboxFilter gathers the data that we're interested in and packages it up an a format that's useful for graphing, it doesn't actually *do* any graphing. Instead, it periodically creates a message of type *heka.sandbox-output*, containing the current circular buffer data, and injects that message back into Heka's message router. This is where the *Dashboard Output* that we've configured comes in.

Heka's DashboardOutput is configured by default to listen for *heka.sandbox-output* messages (along with a few other message types, which we'll ignore for now). When it receives a sandbox output message, it will examine the contents of the message, and if the message contains circular buffer data it will automatically generate a real time graph of that data.

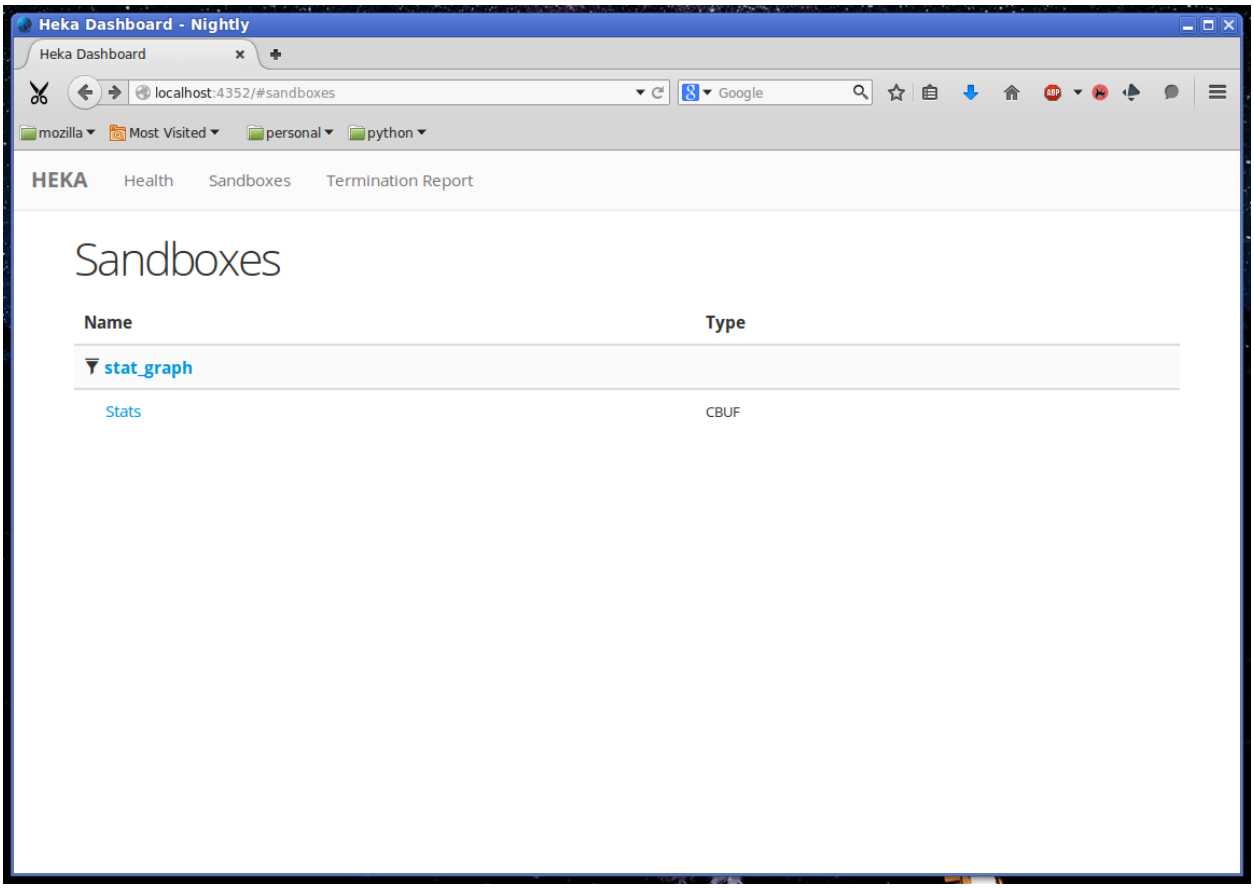
By default, the dashboard UI is available by pointing a web browser at port 4352 of the machine where Heka is running. The first page you'll see is the Health report, which provides an overview of the plugins that are configured, along with some information about how messages are flowing through the Heka pipeline:



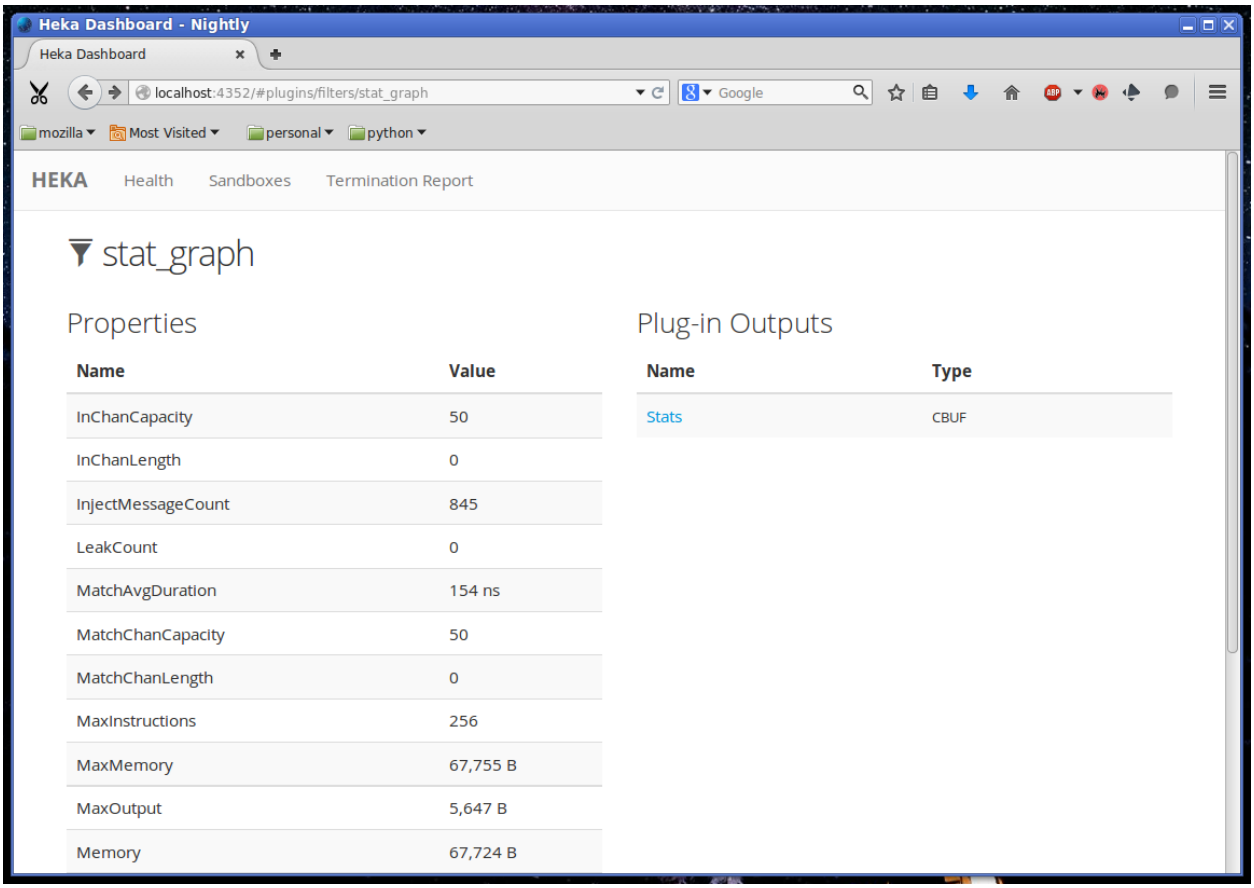
... and scrolling further down the page ...



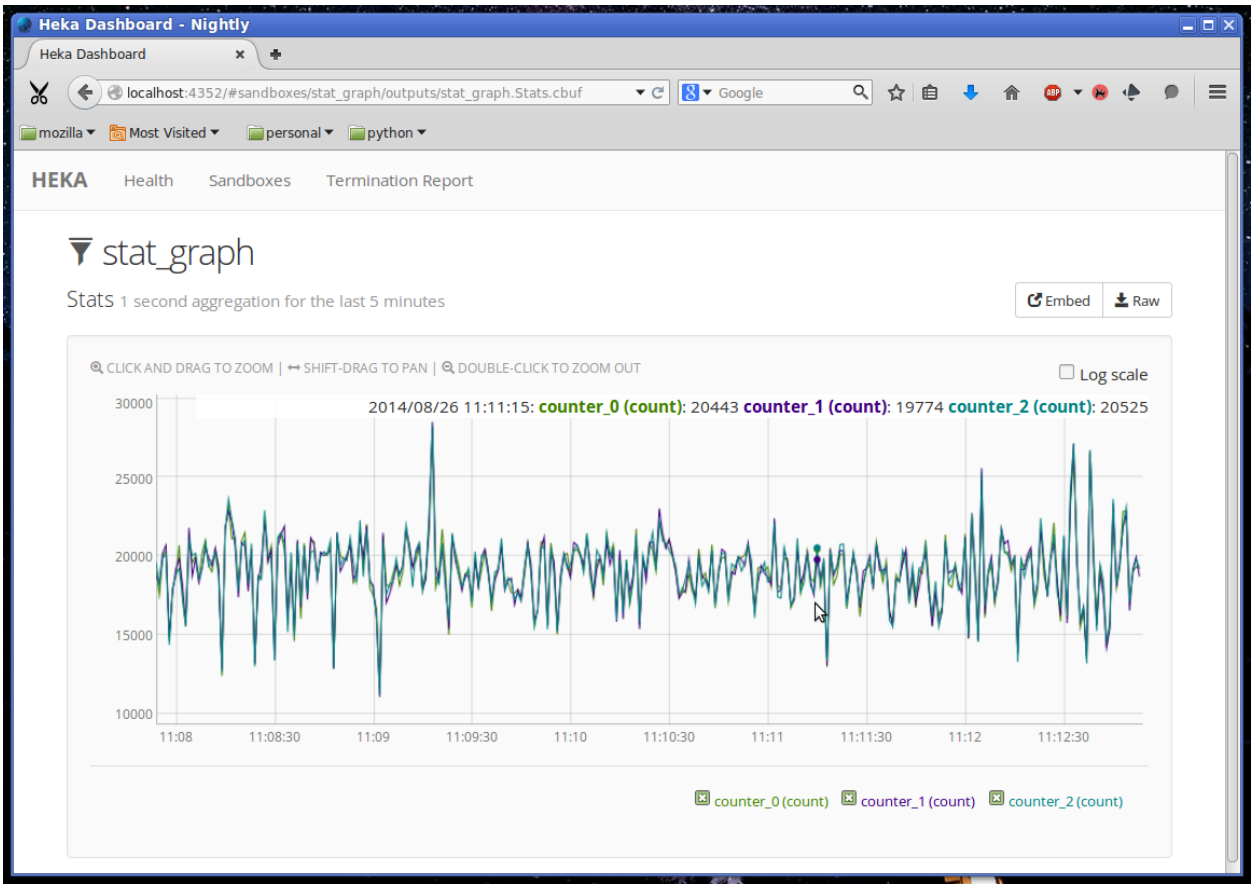
In the page header is a *Sandboxes* link, which will take you to a listing of all of the running SandboxFilter plugins, along with a list of the outputs they emit. Clicking on this we can see our *stat_graph* filter and the *Stats* circular buffer ("CBUF") output:



If you click on the filter name *stat_graph*, you'll see a page showing detailed information about the performance of that plugin, including how many messages have been processed, the average amount of time a message matcher takes to match a message, the average amount of time spent processing a message, and more:



Finally, clicking on the *Stats* link will take us to the actual rendered output, a line graph that updates in real time, showing the values of the specific counter stats that we have specified in our *stat_graph* SandboxFilter configuration:



Other stats can be added to this graph by adjusting the *stats* and *stat_labels* values for our existing *stat_graph* filter config, although if we do so we'll have to bump the *preservation_version* to tell Heka that the previous data structures are no longer valid. You can create multiple graphs by including additional SandboxFilter sections using the same *stat_graph.lua* source code.

It also should be mentioned that, while the *stat_graph.lua* filter we've been using only emits a single output graph, it is certainly possible for a single filter to generate multiple graphs. It's also possible for SandboxFilters to emit other types of output, such as raw JSON data, which the DashboardOutput will happily serve as raw text. This can be very useful for generating ad-hoc API endpoints based on the data that Heka is processing. Dig in to our *Sandbox* documentation to learn more about writing your own Lua filters using our Sandbox API.

2.2.8 Loading and Parsing Nginx Log Files

For our next trick, we'll be loading an Nginx HTTP server's access log files and extracting information about each HTTP request logged therein, storing it in a more structured manner in the fields of a Heka message. The first step is telling Heka where it can find the Nginx access log file. Except that the Nginx log typically isn't just a single file, it's a series of files subject to site specific rotation schemes. On the author's Ubuntu-ish system, for instance, the */var/log/nginx* directory looks like this, at the time of writing:

```
access.log
access.log.1
access.log.2.gz
access.log.3.gz
access.log.4.gz
access.log.5.gz
access.log.6.gz
```

```
access.log.7.gz
access.log.8.gz
access.log.9.gz
error.log
```

This is a common rotation scheme, but there are many others out there. And in cases where many domains are being hosted, there might be several sets of log files, one for each domain, each distinguished from the others by file and/or folder name. Luckily Heka's *Logstreamer Input* provides a mechanism for handling all of these cases and more. The LogstreamerInput already has *extensive documentation*, so we won't go into exhaustive detail here, instead we'll show an example config that correctly handles the above case:

```
[nginx_access_logs]
type = "LogstreamerInput"
splitter = "TokenSplitter"
decoder = "nginx_access_decoder"
log_directory = "/var/log/nginx"
file_match = 'access\.log\.?(?P<Index>\d+)?(.gz)?'
priority = ["^Index"]
```

The *splitter* option above tells Heka that each record will be delimited by a one character token, in this case the default token *\n*. If our records were delimited by a different character we could add a *Token Splitter* section specifying an alternate. If a single character isn't sufficient for finding our record boundaries, such as in cases where a record spans multiple lines, we can use a *Regex Splitter* to provide a regular expression that describes the record boundary. The *log_directory* option tells where the files we're interested in live. The *file_match* is a regular expression that matches all of the files comprising the log stream. In this case, they all must start with *access.log*, after which they can (optionally) be followed by a dot (.), then (optionally, again) one or two digits, then (optionally, one more time) a gzip extension (*.gz*). Any digits that are found are captured as the *Index* match group, and the *priority* option specifies that we use this Index value to determine the order of the files. The leading carat character (^) reverses the order of the priority, since in our case lower digits mean newer files.

The LogstreamerInput will use this configuration data to find all of the relevant files, then it will start working its way through the entire stream of files from oldest to newest, tracking its progress along the way. If Heka is stopped and restarted, it will pick up where it left off, even if that file was rotated during the time that Heka was down. When it gets to the end of the newest file, it will follow along, loading new lines as they're added, and noticing when the file is rotated so it can hop forward to start loading the newer one.

Which then brings us to the *decoder* option. This tells Heka which decoder plugin the LogstreamerInput will be using to parse the loaded log files. The *nginx_access_decoder* configuration is as follows:

```
[nginx_access_decoder]
type = "SandboxDecoder"
filename = "lua_decoders/nginx_access.lua"

        [nginx_access_decoder.config]
        log_format = '$remote_addr - $remote_user [$time_local] "$request" $status $body_bytes_sent "
        type = "nginx.access"
```

Some of this should be looking familiar by now. This is a SandboxDecoder, which means that it is a decoder plugin with the actual parsing logic implemented in Lua. The outer config section configures the SandboxDecoder itself, while the nested section provides additional config information that is passed in to the Lua code.

While it's certainly possible to write your own custom Lua parsing code, in this case we are again using a plugin *provided by Heka*, specifically designed for parsing Nginx access logs. But Nginx doesn't have a single access log format, the exact output is dynamically specified by a *log_format* directive in the Nginx configuration. Luckily Heka's decoder is quite sophisticated; all you have to do to parse your access log output is copy the appropriate *log_format* directive out of the Nginx configuration file and paste it into the *log_format* option in your Heka decoder config, as above, and Heka will use the magic of LPEG to dynamically create a grammar that will extract the data from the log

lines and store them in Heka message fields. Finally the *type* option above lets you specify what the Type field should be set to on the messages generated by this decoder.

2.2.9 Sending Nginx Data to ElasticSearch

One common use case people are interested in is taking the data extracted from their HTTP server logs and sending it on to ElasticSearch, often so they can peruse that data using dashboards generated by the excellent dashboard creation tool Kibana. We've handled loading and parsing the information with our input and decoder configuration above, now let's look at the other side with the following output and encoder settings:

```
[ESJsonEncoder]
es_index_from_timestamp = true
type_name = "%{Type}"

[ElasticSearchOutput]
server = "elasticsearch.example.com:9200"
message_matcher = "Type == 'nginx.access'"
encoder = "ESJsonEncoder"
flush_interval = 50
```

Working backwards, we'll first look at the *ElasticSearch Output* configuration. The *server* setting indicates where ElasticSearch is listening. The *message_matcher* tells us we'll be catching messages with a Type value of *nginx.access*, which you'll recall was set in the decoder configuration we discussed above. The *flush_interval* setting specifies that we'll be batching our records in the output and flushing them out to ElasticSearch every 50 milliseconds.

Which leaves us with the *encoder* setting, and the corresponding *ElasticSearch JSON Encoder* section. The ElasticSearchOutput uses ElasticSearch's Bulk API to tell ElasticSearch how the documents should be indexed, which means that each document insert consists of a small JSON object satisfying the Bulk API followed by another JSON object containing the document itself. At the time of writing, Heka provides three encoders that will extract data from a Heka message and generate an appropriate Bulk API header, the *ElasticSearch JSON Encoder* we use above, which generates a clean document schema based on the schema of the message that is being encoded; the *ElasticSearch Logstash V0 Encoder*, which uses the "v0" schema format defined by Logstash (specifically intended for HTTP request data, natively supported by Kibana), and the *ElasticSearch Payload Encoder*, which assumes that the message payload will already contain a fully formed JSON document ready for sending to ElasticSearch, and just prepends the necessary Bulk API segement.

In our *ESJsonEncoder* section, we're mostly adhering to the default settings. By default, this decoder inserts documents into an ElasticSearch index based on the current date: *heka-YYYY.MM.DD* (spelled as *heka-%{2006.01.02}* in the config). The *es_index_from_timestamp = true* option tells Heka to use the timestamp from the message when determining the date to use for the index name, as opposed to the default behavior which uses the system clock's current time as the basis. The *type* option tells Heka what ElasticSearch record type should be used for each record. This option supports interpolation of various values from the message object; in the example above the message's Type field will be used as the ElasticSearch record type name.

2.2.10 Generating HTTP Status Code Graphs

ElasticSearch and Kibana provide a number of nice tools for graphing and querying the HTTP request data that is being parsed from our Nginx logs but, as with the stats data above, it would be nice to get real time graphs of some of this data directly from Heka. As you might guess, Heka already provides plugins specifically for this purpose:

```
[http_status]
type = "SandboxFilter"
filename = "lua_filters/http_status.lua"
ticker_interval = 1
preserve_data = true
```

```
message_matcher = "Type == 'nginx.access'"

        [http_status.config]
        sec_per_row = 1
        rows = 1800
        perservation_version = 0
```

As mentioned earlier, graphing in Heka is accomplished through the cooperation of a filter which emits messages containing circular buffer data, and the DashboardOutput which consumes those messages and displays the data on a graph. We already configured a DashboardOutput earlier, so now we just need to add a filter that catches the *nginx.access* messages and aggregates the data into a circular buffer.

Heka has a standard message format that it uses for data that represents a single HTTP request, used by the Nginx access log decoder that is parsing our log files. In this format, the status code of the HTTP response is stored in a dynamic message field called, simply, *status*. The above filter will create a circular buffer data structure to store these response status codes in 6 columns: 100s, 200s, 300s, 400s, 500s, and unknown. Similar to before, the nested configuration tells the filter how many rows of data to keep in the circular buffer and how many seconds of data each row should represent. It also gives us a *preservation_version* so we can flag when the data structures have changed.

Once we add this section to our configuration and restart hekad, we should be able to browse to the dashboard UI and be able to find a graph of the various response status categories that are extracted from our HTTP server logs.

2.2.11 Anomaly Detection

We're getting close to the end of our journey. All of the data that we want to gather is now flowing through Heka, being delivered to external data stores for off line processing and analytics, and being displayed in real time graphs by Heka's dashboard. The only remaining behavior we're going to activate is anomaly detection, and the generation of notifiers based on anomalous events being detected. We'll start by looking at the anomaly detection piece.

We've already discussed how Heka uses a circular buffer library to track time series data and generate graphs in the dashboard. Well it turns out that the *anomaly detection* features that Heka provides make use of the same circular buffer library.

Under the hood, how it works is that you provide an "anomaly config", which is a string that looks something like a programming function call. The anomaly config specifies which anomaly detection algorithm should be used. Algorithms currently supported by Heka are a standard deviation rate of change test, and both parametric (i.e. Gaussian) and non-parametric Mann-Whitney-Wilcoxon tests. Included in the anomaly config is information about which column in a circular buffer data structure we want to monitor for anomalous behavior. Later, the parsed anomaly config is passed in to the detection module's *detect* function, along with a populated circular buffer data structure, and the circular buffer data will be analyzed using the specified algorithm.

Luckily, for our use cases, you don't have to worry too much about all of the details of using the anomaly detection library, because the SandboxFilters we've been using have already taken care of the hard parts. All we need to do is create an anomaly config string and add that to our config sections. For instance, here's an example of how we might monitor our HTTP response status codes:

```
[http_status]
type = "SandboxFilter"
filename = "lua_filters/http_status.lua"
ticker_interval = 1
preserve_data = true
message_matcher = "Type == 'nginx.access'"

        [http_status.config]
        sec_per_row = 1
        rows = 1800
```

```
        perservation_version = 0
        anomaly_config = 'roc("HTTP Status", 2, 15, 0, 1.5, true, false) mww_nonparametric("HTTP Stat
```

Everything is the same as our earlier configuration, except we've added an *anomaly_config* setting. There's a lot in there, so we'll examine it a piece at a time. The first thing to notice is that there are actually two anomaly configs specified. You can add as many as you'd like. They're space delimited here for readability, but that's not strictly necessary, the parentheses surrounding the config parameters are enough for Heka to identify them. Next we'll dive into the configurations, each in turn.

The first anomaly configuration by itself looks like this:

```
roc("HTTP Status", 2, 15, 0, 1.5, true, false)
```

The *roc* portion tells us that this config is using the rate of change algorithm. Each algorithm has its own set of parameters, so the values inside the parentheses are those that are required for a rate of change calculation. The first argument is *payload_name*, which needs to correspond to the *payload_name* value used when the message is injected back into Heka's message router, which is "HTTP Status" in the case of this filter.

The next argument is the circular buffer column that we should be watching. We're specifying column 2 here, which a quick peek at the *http_status.lua* source code will show you is the column where we're tracking 200 status codes. The next value specifies how many intervals (i.e. circular buffer rows) should we use in our analysis window. We've said 15, which means that we'll be examining the rate of change between the values in two 15 second intervals. Specifically, we'll be comparing the data in rows 2 through 16 to the data in rows 17 through 31 (we always throw out the current row because it might not yet be complete).

After that we specify the number of intervals to use in our historical analysis window. Our setting of 0 means we're using the entire history, rows 32 through 1800. This is followed by the standard deviation threshold parameter, which we've set to 1.5. So, put together, we're saying if the rate of change of the number of 200 status responses over the last two 15 second intervals is more than 1.5 standard deviations off from the rate of change over the 29 minutes before that, then an anomaly alert should be triggered.

The last two parameters here are boolean values. The first of these is whether or not an alert should be fired in the event that we stop receiving input data (we're saying yes), the second whether or not an alert should be fired if we start receiving data again after a gap (we're saying no).

That's the first one, now let's look at the second:

```
mww_nonparametric("HTTP Status", 5, 15, 10, 0.8)
```

The *mww_nonparametric* tells us, as you might guess, that this config will be using the Mann-Whitney-Wilcoxon non-parametric algorithm for these computations. This algorithm can be used to identify similarities (or differences) between multiple data sets, even when those data sets have a non- Gaussian distribution, such as cases where the set of data points is sparse.

The next argument tells us what column we'll be looking at. In this case we're using column 5, which is where we store the 500 range status responses, or server errors. After that is the number of intervals to use in a analysis window (15), followed by the number of analysis windows to compare (10). In this case, that means we'll be examining the last 15 seconds, and comparing what we find there with the 10 prior 15 second windows, or the 150 previous seconds.

The final argument is called *pstat*, which is a floating point value between 0 and 1. This tells us what type of data changes we're going to be looking for. Anything over a 0.5 means we're looking for an increasing trend, anything below 0.5 means we're looking for a decreasing trend. We've set this to 0.8, which is clearly in the increasing trend range.

So, taken together, this anomaly config means that we're going to be watching the last 15 seconds to see whether there is an anomalous spike in server errors, compared to the 10 intervals immediately prior. If we do detect a sizable spike in server errors, we consider it an anomaly and an alert will be generated.

In this example, we've only specified anomaly detection on our HTTP response status monitoring, but the *anomaly_config* option is also available to the stat graph filter, so we could apply similar monitoring to any of the

statsd data that is contained in our statmetric messages.

2.2.12 Notifications

But what do we mean, exactly, when we say that detecting an anomaly will generate an alert? As with nearly everything else in Heka, what we're really saying is that a message will be injected into the message router, which other filter and output plugins are then able to listen for and use as a trigger for action.

We won't go into detail here, but along with the anomaly detection module Heka's Lua environment provides an *alert module* that generates alert messages (with throttling, to make sure hundreds of alerts in rapid succession don't actually generate hundreds of separate notifications) and an *annotation module* that causes the dashboard to apply annotations to the graphs based on our circular buffer data. Both the http status and stat graph filters make use of both of these, so if you specify anomaly configs for either of those filters, output graphs will be annotated and alert messages will be generated when anomalies are detected.

Alert messages aren't of much use if they're just flowing through Heka's message router and nothing is listening for them, however. So let's set up an SmtpOutput that will listen for the alert messages, sending emails when they come through:

```
[alert_smtp_encoder]
type = "SandboxEncoder"
filename = "lua_encoders/alert.lua"

[SmtpOutput]
message_matcher = "Type == 'heka.sandbox-output' && Fields[payload_type] == 'alert'"
encoder = "alert_smtp_encoder"
send_from = "heka@example.com"
send_to = ["alert_recipient@example.com"]
auth = "Plain"
user = "smtpuser"
password = "smtpassword"
host = "127.0.0.1:25"
```

First we specify an encoder, using a very simple encoder implementation provided by Heka which extracts the timestamp, hostname, logger, and payload from the message and emits those values in a text format. Then we add the output itself, listening for any alert messages that are emitted by any of our SandboxFilter plugins, using the encoder to format the message body, and sending an outgoing mail message through the SMTP server as specified by the other configuration options.

And that's it! We're now generating email notifiers from our anomaly detection alerts.

2.2.13 Tying It All Together

Here's what our full config looks like if we put it all together into a single file:

```
[hekad]
maxprocs = 2

[StatsdInput]

[StatAccumInput]
ticker_interval = 1
emit_in_fields = true

[CarbonOutput]
message_matcher = "Type == 'heka.statmetric'"
address = "mycarbonserver.example.com:2003"
```

```
protocol = "udp"

[statmetric-influx-encoder]
type = "SandboxEncoder"
filename = "lua_encoders/statmetric_influx.lua"

[influx]
type = "HttpOutput"
message_matcher = "Type == 'heka.statmetric'"
address = "http://myinfluxserver.example.com:8086/db/stats/series"
encoder = "statmetric-influx-encoder"
username = "influx_username"
password = "influx_password"

[stat_graph]
type = "SandboxFilter"
filename = "lua_filters/stat_graph.lua"
ticker_interval = 1
preserve_data = true
message_matcher = "Type == 'heka.statmetric'"

        [stat_graph.config]
        num_rows = 300
        secs_per_row = 1
        stats = "stats.counters.000000.count stats.counters.000001.count stats.counters.000002.count"
        stat_labels = "counter_0 counter_1 counter_2"
        preservation_version = 0

[DashboardOutput]
ticker_interval = 1

[nginx_access_logs]
type = "LogstreamerInput"
splitter = "TokenSplitter"
decoder = "nginx_access_decoder"
log_directory = "/var/log/nginx"
file_match = 'access\.log\.?(?P<Index>\d+)?(.gz)?'
priority = ["^Index"]

[nginx_access_decoder]
type = "SandboxDecoder"
script_type = "lua"
filename = "lua_decoders/nginx_access.lua"

        [nginx_access_decoder.config]
        log_format = '$remote_addr - $remote_user [$time_local] "$request" $status $body_bytes_sent "
        type = "nginx.access"

[ESJsonEncoder]
es_index_from_timestamp = true
type_name = "%{Type}"

[ElasticSearchOutput]
message_matcher = "Type == 'nginx.access'"
encoder = "ESJsonEncoder"
flush_interval = 50

[http_status]
```

```
type = "SandboxFilter"
filename = "lua_filters/http_status.lua"
ticker_interval = 1
preserve_data = true
message_matcher = "Type == 'nginx.access'"

        [http_status.config]
        sec_per_row = 1
        rows = 1440
        perservation_version = 0
        anomaly_config = 'roc("HTTP Status", 2, 15, 0, 1.5, true, false) mww_nonparametric("HTTP Stat

[alert_smtp_encoder]
type = "SandboxEncoder"
filename = "lua_encoders/alert.lua"

[SmtpOutput]
message_matcher = "Type == 'heka.sandbox-output' && Fields[payload_type] == 'alert'"
encoder = "alert_smtp_encoder"
send_from = "heka@example.com"
send_to = ["alert_recipient@example.com"]
auth = "Plain"
user = "smtpuser"
password = "smtpassword"
host = "127.0.0.1:25"
```

This isn't too terribly long, but even so it might be nice to break it up into smaller pieces. Heka supports the use of a directory instead of a single file for configuration; if you specify a directory all files ending with *.toml* will be merged together and loaded as a single configuration, which is preferable for more complex deployments.

This example is not in any way meant to be an exhaustive list of Heka's features. Indeed, we've only just barely scratched the surface. Hopefully, though, it gives those of you who are new to Heka enough context to understand how the pieces fit together, and it can be used as a starting point for developing configurations that will meet your own needs. If you have questions or need assistance getting things going, please make use of the mailing list, or use an IRC client to come visit in the #heka channel on irc.mozilla.org.

2.3 Configuring hekad

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

If hekad's config file is specified to be a directory, all contained files with a filename ending in ".toml" will be loaded and merged into a single config. Files that don't end with ".toml" will be ignored. Merging will happen in alphabetical order, settings specified later in the merge sequence will win conflicts.

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"
splitter = "HekaFramingSplitter"
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"
splitter = "HekaFramingSplitter"
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*).

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.

2.3.1 Global configuration options

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

Config:

- **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.
- **plugin_chansize (int):** Specify the buffer size for the input channel for the various Heka plugins. Defaults to 30.

- **base_dir (string):** Base working directory Heka will use for persistent storage through process and server restarts. The hekad process must have read and write access to this directory. Defaults to */var/cache/hekad* (or *c:\var\cache\hekad* on Windows).
- **share_dir (string):** Root path of Heka's "share directory", where Heka will expect to find certain resources it needs to consume. The hekad process should have read- only access to this directory. Defaults to */usr/share/heka* (or *c:\usr\share\heka* on Windows).

New in version 0.6.

- **sample_denominator (int):** Specifies the denominator of the sample rate Heka will use when computing the time required to perform certain operations, such as for the ProtobufDecoder to decode a message, or the router to compare a message against a message matcher. Defaults to 1000, i.e. duration will be calculated for one message out of 1000.

New in version 0.6.

- **pid_file (string):** Optionally specify the location of a pidfile where the process id of the running hekad process will be written. The hekad process must have read and write access to the parent directory (which is not automatically created). On a successful exit the pidfile will be removed. If the path already exists the contained pid will be checked for a running process. If one is found, the current process will exit with an error.

New in version 0.9.

- **hostname (string):** Specifies the hostname to use whenever Heka is asked to provide the local host's hostname. Defaults to whatever is provided by Go's *os.Hostname()* call.
- **max_message_size (uint32):** The maximum size (in bytes) of message can be sent during processing. Defaults to 64KiB.

2.3.2 Example hekad.toml file

```
[hekad]
maxprocs = 4

# Heka dashboard for internal metrics and time series graphs
[Dashboard]
type = "DashboardOutput"
address = ":4352"
ticker_interval = 15

# Email alerting for anomaly detection
[Alert]
type = "SmtpOutput"
message_matcher = "Type == 'heka.sandbox-output' && Fields[payload_type] == 'alert'"
send_from = "acme-alert@example.com"
send_to = ["admin@example.com"]
auth = "Plain"
user = "smtp-user"
password = "smtp-pass"
host = "mail.example.com:25"
encoder = "AlertEncoder"

# User friendly formatting of alert messages
[AlertEncoder]
type = "SandboxEncoder"
filename = "lua_encoders/alert.lua"
```

```
# Nginx access log reader
[AcmeWebserver]
type = "LogstreamerInput"
log_directory = "/var/log/nginx"
file_match = 'access\.log'
decoder = "CombinedNginxDecoder"

# Nginx access 'combined' log parser
[CombinedNginxDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/nginx_access.lua"

    [CombinedNginxDecoder.config]
    user_agent_transform = true
    user_agent_conditional = true
    type = "combined"
    log_format = '$remote_addr - $remote_user [$time_local] "$request" $status $body_bytes_sent "$htt

# Collection and visualization of the HTTP status codes
[AcmeHTTPStatus]
type = "SandboxFilter"
filename = "lua_filters/http_status.lua"
ticker_interval = 60
preserve_data = true
message_matcher = "Logger == 'AcmeWebserver'"

    # rate of change anomaly detection on column 1 (HTTP 200)
    [AcmeHTTPStatus.config]
    anomaly_config = 'roc("HTTP Status", 1, 15, 0, 1.5, true, false)'
```

2.3.3 Using Environment Variables

If you wish to use environmental variables in your config files as a way to configure values, you can simply use `%ENV[VARIABLE_NAME]` and the text will be replaced with the value of the environmental variable `VARIABLE_NAME`.

Example:

```
[AMQPInput]
url = "amqp://%ENV[USER]:%ENV[PASSWORD]@rabbitmq/"
exchange = "testout"
exchangeType = "fanout"
```

2.3.4 Configuring Restarting Behavior

Plugins that support being restarted have a set of options that govern how a restart is handled if they exit with an error. If preferred, the plugin can be configured to not restart, or it could be restarted only 100 times, or restart attempts can proceed forever. Once the *max_retries* have been exceeded the plugin will be unregistered, potentially triggering hekad to shutdown (depending on the plugin's *can_exit* configuration).

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

Config:

- **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 exiting the plugin. 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). Defaults to -1.

Example:

```
[AMQPOutput]
url = "amqp://guest:guest@rabbitmq/"
exchange = "testout"
exchange_type = "fanout"
message_matcher = 'Logger == "TestWebserver"'

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

2.4 Inputs

2.4.1 Common Input Parameters

New in version 0.9.

There are some configuration options that are universally available to all Heka input 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.

- **decoder (string, optional):** Decoder to be used by the input. This should refer to the name of a registered decoder plugin configuration. If supplied, messages will be decoded before being passed on to the router when the InputRunner's *Deliver* method is called.
- **synchronous_decode (bool, optional):** If *synchronous_decode* is false, then any specified decoder plugin will be run by a DecoderRunner in its own goroutine and messages will be passed in to the decoder over a channel, freeing the input to start processing the next chunk of incoming or available data. If true, then any decoding will happen synchronously and message delivery will not return control to the input until after decoding has completed. Defaults to false.
- **send_decode_failures (bool, optional):** If false, then if an attempt to decode a message fails then Heka will log an error message and then drop the message. If true, then in addition to logging an error message, decode failure will cause the original, undecoded message to be tagged with a *decode_failure* field (set to true) and delivered to the router for possible further processing.
- **can_exit (bool, optional):** If false, the input plugin exiting will trigger a Heka shutdown. If set to true, Heka will continue processing other plugins. Defaults to false on most inputs.
- **splitter (string, optional)** Splitter to be used by the input. This should refer to the name of a registered splitter plugin configuration. It specifies how the input should split the incoming data stream into individual records prior to decoding and/or injection to the router. Typically defaults to "NullSplitter", although certain inputs override this with a different default value.

2.4.2 Available Input Plugins

AMQP Input

Plugin Name: **AMQPInput**

Connects to a remote AMQP broker (RabbitMQ) and retrieves messages from the specified queue. As AMQP is dynamically programmable, the broker topology needs to be specified in the plugin configuration.

Config:

- **url (string):** An AMQP connection string formatted per the RabbitMQ URI Spec.
- **exchange (string):** AMQP exchange name
- **exchange_type (string):** AMQP exchange type (*fanout*, *direct*, *topic*, or *headers*).
- **exchange_durability (bool):** Whether the exchange should be configured as a durable exchange. Defaults to non-durable.
- **exchange_auto_delete (bool):** Whether the exchange is deleted when all queues have finished and there is no publishing. Defaults to auto-delete.
- **routing_key (string):** The message routing key used to bind the queue to the exchange. Defaults to empty string.
- **prefetch_count (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.
- **queue_durability (bool):** Whether the queue is durable or not. Defaults to non-durable.
- **queue_exclusive (bool):** Whether the queue is exclusive (only one consumer allowed) or not. Defaults to non-exclusive.
- **queue_auto_delete (bool):** Whether the queue is deleted when the last consumer un-subscribes. Defaults to auto-delete.
- **queue_ttl (int):** Allows ability to specify TTL in milliseconds on Queue declaration for expiring messages. Defaults to undefined/infinite.
- **retries (RetryOptions, optional):** A sub-section that specifies the settings to be used for restart behavior. See *Configuring Restarting Behavior*

New in version 0.6.

- **tls (TlsConfig):** An optional sub-section that specifies the settings to be used for any SSL/TLS encryption. This will only have any impact if *URL* uses the *AMQPS* URI scheme. See *Configuring TLS*.

New in version 0.9.

- **read_only (bool):** Whether the AMQP user is read-only. If this is true the exchange, queue and binding must be declared before starting Heka. Defaults to false.

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"
exchange_type = "fanout"
```

Or you might use a PayloadRegexDecoder to parse OSX syslog messages with the following:

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

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

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

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

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

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

Docker Event Input

New in version 0.10.0.

Plugin Name: **DockerEventInput**

The DockerEventInput plugin connects to the Docker daemon and watches the Docker events API, sending all events to the Heka pipeline. See: Docker Events API Messages will be populated as follows:

- Uuid: Type 4 (random) UUID generated by Heka.
- Timestamp: Time when the event was received by the plugin.
- Type: *DockerEvent*.
- Hostname: Hostname of the machine on which Heka is running.
- Payload: The event id, status, from and time. Example: *id:47e08ded0abb57ca263136291f14ed7689de8b6ec519f01ea76958fe512abff9 status:create from:gliderlabs/alpine:3.1 time:1429555298*
- Logger: The id provided in the event
- Fields["ID"] (string): The ID provided in the docker event.
- Fields["Status"] (string): The Status in the docker event.
- Fields["From"] (string): The From in the docker event.
- Fields["Time"] (string): The timestamp in the docker event.

Config:

- **endpoint (string):** A Docker endpoint. Defaults to "unix:///var/run/docker.sock".
- **cert_path (string, optional):** Path to directory containing client certificate and keys. This value works in the same way as DOCKER_CERT_PATH.

Example:

```
[DockerEventInput]
type = "DockerEventInput"

[PayloadEncoder]
append_newlines = false

[LogOutput]
type = "LogOutput"
message_matcher = "Type == 'DockerEvent'"
encoder = "PayloadEncoder"
```

Docker Log Input

New in version 0.8.

Plugin Name: **DockerLogInput**

The DockerLogInput plugin attaches to all containers running on a host and sends their logs messages into the Heka pipeline. The plugin is based on Logspout by Jeff Lindsay. Messages will be populated as follows:

- Uuid: Type 4 (random) UUID generated by Heka.
- Timestamp: Time when the log line was received by the plugin.
- Type: *DockerLog*.
- Hostname: Hostname of the machine on which Heka is running.
- Payload: The log line received from a Docker container.
- Logger: *stdout* or *stderr*, depending on source.
- Fields["ContainerID"] (string): The container ID.
- Fields["ContainerName"] (string): The container name.

Note: Logspout expects to be dealing exclusively with textual log file data, and always assumes that the file data is newline delimited, i.e. one line in the log file equals one logical unit of data. For this reason, the DockerLogInput currently does *not* support the use of alternate splitter plugins. Any splitter setting specified in a DockerLogInput's configuration will be ignored.

Config:

- **endpoint (string):** A Docker endpoint. Defaults to "unix:///var/run/docker.sock".
- **decoder (string):** The name of the decoder used to further transform the message into a structured hekad message. No default decoder is specified.

New in version 0.9.

- **cert_path (string, optional):** Path to directory containing client certificate and keys. This value works in the same way as DOCKER_CERT_PATH.

New in version 0.10.

- **name_from_env_var (string, optional):** Overwrite the ContainerName with this environment variable on the Container if exists. If left empty the container name will still be used.
- **fields_from_env (array[string], optional):** A list of environment variables to extract from the container and add as fields.

Example:

```
[nginx_log_decoder]
type = "SandboxDecoder"
filename = "lua_decoders/nginx_access.lua"

[nginx_log_decoder.config]
type = "nginx.access"
user_agent_transform = true
log_format = '$remote_addr - $remote_user [$time_local] "$request" $status $body_bytes_sent "$http_re

[DockerLogInput]
decoder = "nginx_log_decoder"
fields_from_env = [ "MESOS_TASK_ID" ]
```

File Polling Input

New in version 0.7.

Plugin Name: **FilePollingInput**

FilePollingInputs periodically read (unbuffered) the contents of a file specified, and creates a Heka message with the contents of the file as the payload.

Config:

- **file_path(string):** The absolute path to the file which the input should read.
- **ticker_interval (unit):** How often, in seconds to input should read the contents of the file.

Example:

```
[MemStats]
type = "FilePollingInput"
ticker_interval = 1
file_path = "/proc/meminfo"
decoder = "MemStatsDecoder"
```

HTTP Input

Plugin Name: **HttpInput**

HttpInput plugins intermittently poll remote HTTP URLs for data and populate message objects based on the results of the HTTP interactions. Messages will be populated as follows:

- Uuid: Type 4 (random) UUID generated by Heka.
- Timestamp: Time HTTP request is completed.
- **Type:** ***heka.httpinput.data*** **or** ***heka.httpinput.error*** **depending on whether or** not the request completed. (Note that a response returned with an HTTP error code is still considered complete and will generate type *heka.httpinput.data*.)
- Hostname: Hostname of the machine on which Heka is running.

- Payload: Entire contents of the HTTP response body.
- **Severity: HTTP response 200 uses *success_severity* config value, all other** results use *error_severity* config value.
- Logger: Fetched URL.
- Fields["Status"] (string): HTTP status string value (e.g. "200 OK").
- Fields["StatusCode"] (int): HTTP status code integer value.
- Fields["ResponseSize"] (int): Value of HTTP Content-Length header.
- **Fields["ResponseTime"] (float64): Clock time elapsed for HTTP request, in** seconds.
- **Fields["Protocol"] (string): HTTP protocol used for the request (e.g.** "HTTP/1.0")

The *Fields* values above will only be populated in the event of a completed HTTP request. Also, it is possible to specify a decoder to further process the results of the HTTP response before injecting the message into the router.

Config:

- **url (string):** A HTTP URL which this plugin will regularly poll for data. This option cannot be used with the urls option. No default URL is specified.
- **urls (array):** New in version 0.5.

  An array of HTTP URLs which this plugin will regularly poll for data. This option cannot be used with the url option. No default URLs are specified.
- **method (string):** New in version 0.5.

  The HTTP method to use for the request. Defaults to "GET".
- **headers (subsection):** New in version 0.5.

  Subsection defining headers for the request. By default the User-Agent header is set to "Heka"
- **body (string):** New in version 0.5.

  The request body (e.g. for an HTTP POST request). No default body is specified.
- **username (string):** New in version 0.5.

  The username for HTTP Basic Authentication. No default username is specified.
- **password (string):** New in version 0.5.

  The password for HTTP Basic Authentication. No default password is specified.
- **ticker_interval (uint):** Time interval (in seconds) between attempts to poll for new data. Defaults to 10.
- **success_severity (uint):** New in version 0.5.

  Severity level of successful HTTP request. Defaults to 6 (information).
- **error_severity (uint):** New in version 0.5.

  Severity level of errors, unreachable connections, and non-200 responses of successful HTTP requests. Defaults to 1 (alert).

Example:

```
[HttpInput]
url = "http://localhost:9876/"
ticker_interval = 5
success_severity = 6
error_severity = 1
decoder = "MyCustomJsonDecoder"
```

```
    [HttpInput.headers]
    user-agent = "MyCustomUserAgent"
```

HTTP Listen Input

New in version 0.5.

Plugin Name: **HttpListenInput**

HttpListenInput plugins start a webserver listening on the specified address and port. If no decoder is specified data in the request body will be populated as the message payload. Messages will be populated as follows:

- Uuid: Type 4 (random) UUID generated by Heka.
- Timestamp: Time HTTP request is handled.
- Type: *heka.httpdata.request*
- Hostname: The remote network address of requester.
- Payload: Entire contents of the HTTP request body.
- Severity: 6
- Logger: HttpListenInput
- Fields["UserAgent"] (string): Request User-Agent header (e.g. "GitHub Hookshot dd0772a").
- Fields["ContentType"] (string): Request Content-Type header (e.g. "application/x-www-form-urlencoded").
- **Fields["Protocol"] (string): HTTP protocol used for the request (e.g.** "HTTP/1.0")

New in version 0.6.

All query parameters are added as fields. For example, a request to "127.0.0.1:8325?user=bob" will create a field "user" with the value "bob".

Config:

- **address (string):** An IP address:port on which this plugin will expose a HTTP server. Defaults to "127.0.0.1:8325".

New in version 0.7.

- **headers (subsection, optional):** It is possible to inject arbitrary HTTP headers into each outgoing response by adding a TOML subsection entitled "headers" to you HttpOutput config section. All entries in the subsection must be a list of string values.

New in version 0.9.

- **request_headers ([]string):** Add additional request headers as message fields. Defaults to empty list.

New in version 0.10.

- **auth_type (string, optional):** If requiring Authentication specify "Basic" or "API" To use "API" you must set a header called "X-API-KEY" with the value of the "api_key" config.
- **username (string, optional):** Username to check against if auth_type = "Basic".
- **password (string, optional):** Password to check against if auth_type = "Basic".
- **api_key (string, optional):** String to validate the "X-API-KEY" header against when using auth_type = "API"
- **use_tls (bool):** Specifies whether or not SSL/TLS encryption should be used for the TCP connections. Defaults to false.

- **tls (TlsConfig):** A sub-section that specifies the settings to be used for any SSL/TLS encryption. This will only have any impact if *use_tls* is set to true. See *Configuring TLS*.

Example:

```
[HttpListenInput]
address = "0.0.0.0:8325"
```

With Basic Auth:

```
[HttpListenInput]
address = "0.0.0.0:8325"
auth_type = "Basic"
username = "foo"
password = "bar"
```

With API Key Auth:

```
[HttpListenInput]
address = "0.0.0.0:8325"
auth_type = "API"
api_key = "1234567"
```

Kafka Input

Plugin Name: **KafkaInput**

Connects to a Kafka broker and subscribes to messages from the specified topic and partition.

Config:

- **id (string)** Client ID string. Default is the hostname.
- **addrs ([]string)** List of brokers addresses.
- **metadata_retries (int)** How many times to retry a metadata request when a partition is in the middle of leader election. Default is 3.
- **wait_for_election (uint32)** How long to wait for leader election to finish between retries (in milliseconds). Default is 250.
- **background_refresh_frequency (uint32)** How frequently the client will refresh the cluster metadata in the background (in milliseconds). Default is 600000 (10 minutes). Set to 0 to disable.
- **max_open_reqests (int)** How many outstanding requests the broker is allowed to have before blocking attempts to send. Default is 4.
- **dial_timeout (uint32)** How long to wait for the initial connection to succeed before timing out and returning an error (in milliseconds). Default is 60000 (1 minute).
- **read_timeout (uint32)** How long to wait for a response before timing out and returning an error (in milliseconds). Default is 60000 (1 minute).
- **write_timeout (uint32)** How long to wait for a transmit to succeed before timing out and returning an error (in milliseconds). Default is 60000 (1 minute).
- **topic (string)** Kafka topic (must be set).
- **partition (int32)** Kafka topic partition. Default is 0.
- **group (string)** A string that uniquely identifies the group of consumer processes to which this consumer belongs. By setting the same group id multiple processes indicate that they are all part of the same consumer group. Default is the *id*.

- **default_fetch_size (int32)** The default (maximum) amount of data to fetch from the broker in each request. The default is 32768 bytes.
- **min_fetch_size (int32)** The minimum amount of data to fetch in a request - the broker will wait until at least this many bytes are available. The default is 1, as 0 causes the consumer to spin when no messages are available.
- **max_message_size (int32)** The maximum permittable message size - messages larger than this will return MessageTooLarge. The default of 0 is treated as no limit.
- **max_wait_time (uint32)** The maximum amount of time the broker will wait for min_fetch_size bytes to become available before it returns fewer than that anyways. The default is 250ms, since 0 causes the consumer to spin when no events are available. 100-500ms is a reasonable range for most cases.
- **offset_method (string)** The method used to determine at which offset to begin consuming messages. The valid values are:
  - *Manual* Heka will track the offset and resume from where it last left off (default).
  - *Newest* Heka will start reading from the most recent available offset.
  - *Oldest* Heka will start reading from the oldest available offset.
- **event_buffer_size (int)** The number of events to buffer in the Events channel. Having this non-zero permits the consumer to continue fetching messages in the background while client code consumes events, greatly improving throughput. The default is 16.

Example 1: Read Fxa messages from partition 0.

```
[FxaKafkaInputTest]
type = "KafkaInput"
topic = "Fxa"
addrs = ["localhost:9092"]
```

Example 2: Send messages between two Heka instances via a Kafka broker.

```
# On the producing instance
[KafkaOutputExample]
type = "KafkaOutput"
message_matcher = "TRUE"
topic = "heka"
addrs = ["kafka-broker:9092"]
encoder = "ProtobufEncoder"
```

```
# On the consuming instance
[KafkaInputExample]
type = "KafkaInput"
topic = "heka"
addrs = ["kafka-broker:9092"]
splitter = "KafkaSplitter"
decoder = "ProtobufDecoder"

[KafkaSplitter]
type = "NullSplitter"
use_message_bytes = true
```

Logstreamer Input

New in version 0.5.

Plugin Name:: **LogstreamerInput**

Tails a single log file, a sequential single log source, or multiple log sources of either a single logstream or multiple logstreams.

See also:

Complete documentation with examples

Config:

- **hostname (string):** The hostname to use for the messages, by default this will be the machine's qualified hostname. This can be set explicitly to ensure it's the correct name in the event the machine has multiple interfaces/hostnames.
- **oldest_duration (string):** A time duration string (e.x. "2s", "2m", "2h"). Logfiles with a last modified time older than `oldest_duration` ago will not be included for parsing.
- **journal_directory (string):** The directory to store the journal files in for tracking the location that has been read to thus far. By default this is stored under heka's base directory.
- **log_directory (string):** The root directory to scan files from. This scan is recursive so it should be suitably restricted to the most specific directory this selection of logfiles will be matched under. The log_directory path will be prepended to the file_match.
- **rescan_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.
- **file_match (string):** Regular expression used to match files located under the `log_directory`. This regular expression has `$` added to the end automatically if not already present, and `log_directory` as the prefix. WARNING: file_match should typically be delimited with single quotes, indicating use of a raw string, rather than double quotes, which require all backslashes to be escaped. For example, *'access\.log'* will work as expected, but *"access\.log"* will not, you would need *"access\\.log"* to achieve the same result.
- **priority (list of strings):** When using sequential logstreams, the priority is how to sort the logfiles in order from oldest to newest.
- **differentiator (list of strings):** When using multiple logstreams, the differentiator is a set of strings that will be used in the naming of the logger, and portions that match a captured group from the `file_match` will have their matched value substituted in.
- **translation (hash map of hash maps of ints):** A set of translation mappings for matched groupings to the ints to use for sorting purposes.
- **splitter (string, optional):** Defaults to "TokenSplitter", which will split the log stream into one Heka message per line.

Process Input

Plugin Name: **ProcessInput**

Executes one or more external programs on an interval, creating messages from the output. Supports a chain of commands, where stdout from each process will be piped into the stdin for the next process in the chain. ProcessInput creates Fields[ExitStatus] and Fields[SubcmdErrors]. Fields[ExitStatus] represents the platform dependent exit status of the last command in the command chain. Fields[SubcmdErrors] represents errors from each sub command, in the format of "Subcommand[<subcommand ID>] returned an error: <error message>".

Config:

- **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 below). ProcessInput expects the commands to be indexed by integers starting with 0, where 0 is the first process in the chain.
- **ticker_interval (uint):** The number of seconds to wait between each run of *command*. Defaults to 15. A ticker_interval of 0 indicates that the command is run only once, and should only be used for long running processes that do not exit. If ticker_interval is set to 0 and the process exits, then the ProcessInput will exit, invoking the restart behavior (see *Configuring Restarting Behavior*). Ignored when used in conjunction with *Process Directory Input*, where *ticker_interval* value is instead parsed from the directory path.
- **immediate_start (bool):** If true, heka starts process immediately instead of waiting for first interval defined by ticker_interval to pass. Defaults to false.
- **stdout (bool):** If true, for each run of the process chain a message will be generated with the last command in the chain's stdout as the payload. Defaults to true.
- **stderr (bool):** If true, for each run of the process chain a message will be generated with the last command in the chain's stderr as the payload. Defaults to false.
- **timeout (uint):** Timeout in seconds before any one of the commands in the chain is terminated.
- **retries (RetryOptions, optional):** A sub-section that specifies the settings to be used for restart behavior. See *Configuring Restarting Behavior*

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.
- **env ([]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.

Example:

```
[on_space]
type = "TokenSplitter"
delimiter = " "

[DemoProcessInput]
type = "ProcessInput"
ticker_interval = 2
splitter = "on_space"
stdout = true
stderr = false

    [DemoProcessInput.command.0]
    bin = "/bin/cat"
    args = ["../testsupport/process_input_pipes_test.txt"]

    [DemoProcessInput.command.1]
    bin = "/usr/bin/grep"
    args = ["ignore"]
```

Process Directory Input

Plugin Name: **ProcessDirectoryInput**

New in version 0.5.

The ProcessDirectoryInput periodically scans a filesystem directory looking for ProcessInput configuration files. The ProcessDirectoryInput will maintain a pool of running ProcessInputs based on the contents of this directory, refreshing the set of running inputs as needed with every rescan. This allows Heka administrators to manage a set of data collection processes for a running hekad server without restarting the server.

Each ProcessDirectoryInput has a *process_dir* configuration setting, which is the root folder of the tree where scheduled jobs are defined. It should contain exactly one nested level of subfolders, named with ASCII numeric characters indicating the interval, in seconds, between each process run. These numeric folders must contain TOML files which specify the details regarding which processes to run.

For example, a process_dir might look like this:

```
-/usr/share/heka/processes/
 |-5
   |- check_myserver_running.toml
 |-61
   |- cat_proc_mounts.toml
   |- get_running_processes.toml
 |-302
   |- some_custom_query.toml
```

This indicates one process to be run every five seconds, two processes to be run every 61 seconds, and one process to be run every 302 seconds.

Note that ProcessDirectoryInput will ignore any files that are not nested one level deep, are not in a folder named for an integer 0 or greater, and do not end with '.toml'. Each file which meets these criteria, such as those shown in the example above, should contain the TOML configuration for exactly one *Process Input*, matching that of a standalone ProcessInput with the following restrictions:

- The section name *must* be *ProcessInput*. Any TOML sections named anything other than ProcessInput will be ignored.
- Any specified *ticker_interval* value will be *ignored*. The ticker interval value to use will be parsed from the directory path.

By default, if the specified process fails to run or the ProcessInput config fails for any other reason, ProcessDirectoryInput will log an error message and continue, as if the ProcessInput's *can_exit* flag has been set to true. If the managed ProcessInput's *can_exit* flag is manually set to *false*, it will trigger a Heka shutdown.

Config:

- **ticker_interval (int, optional):** Amount of time, in seconds, between scans of the process_dir. Defaults to 300 (i.e. 5 minutes).
- **process_dir (string, optional):** This is the root folder of the tree where the scheduled jobs are defined. Absolute paths will be honored, relative paths will be computed relative to Heka's globally specified share_dir. Defaults to "processes" (i.e. "$share_dir/processes").
- **retries (RetryOptions, optional):** A sub-section that specifies the settings to be used for restart behavior of the ProcessDirectoryInput (not the individual ProcessInputs, which are configured independently). See *Configuring Restarting Behavior*

Example:

```
[ProcessDirectoryInput]
process_dir = "/etc/hekad/processes.d"
ticker_interval = 120
```

Sandbox Input

New in version 0.9.

Plugin Name: **SandboxInput**

The SandboxInput provides a flexible execution environment for data ingestion and transformation without the need to recompile Heka. Like all other sandboxes it needs to implement a process_message function. However, it doesn't have to return until shutdown. If you would like to implement a polling interface process_message can return zero when complete and it will be called again the next time TickerInterval fires (if ticker_interval was set to zero it would simply exit after running once). See *Sandbox*. Config:

- All of the common input configuration parameters are ignored since the data processing (splitting and decoding) should happen in the plugin.
- ***Common Sandbox Parameters***
  - `instruction_limit` is always set to zero for SandboxInputs

Example

```
[MemInfo]
type = "SandboxInput"
filename = "meminfo.lua"

[MemInfo.config]
path = "/proc/meminfo"
```

Stat Accumulator Input

Plugin Name: **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.

Config:

- **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".
- **legacy_namespaces (bool):** If set to true, then use the older format for namespacing counter stats, with rates recorded under *stats.<counter_name>* and absolute count recorded under *stats_counts.<counter_name>*. See statsd metric namespacing. Defaults to false.
- **global_prefix (string):** Global prefix to use for sending stats to graphite. Defaults to "stats".
- **counter_prefix (string):** Secondary prefix to use for namespacing counter metrics. Has no impact unless *legacy_namespaces* is set to false. Defaults to "counters".

- **timer_prefix (string):** Secondary prefix to use for namespacing timer metrics. Defaults to "timers".
- **gauge_prefix (string):** Secondary prefix to use for namespacing gauge metrics. Defaults to "gauges".
- **statsd_prefix (string):** Prefix to use for the statsd *numStats* metric. Defaults to "statsd".
- **delete_idle_stats (bool):** Don't emit values for inactive stats instead of sending 0 or in the case of gauges, sending the previous value. Defaults to false.

Example:

```
[StatAccumInput]
emit_in_fields = true
delete_idle_stats = true
ticker_interval = 5
```

Statsd Input

Plugin Name: **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.

Config:

- **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".
- **max_msg_size (uint):** Size of a buffer used for message read from statsd. In some cases, when statsd sends a lots in single message of stats it's required to boost this value. All over-length data will be truncated without raising an error. Defaults to 512.

Example:

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

TCP Input

Plugin Name: **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.

Config:

- **address (string):** An IP address:port on which this plugin will listen.

New in version 0.4.

- **decoder (string):** Defaults to "ProtobufDecoder".

New in version 0.5.

- **use_tls (bool):** Specifies whether or not SSL/TLS encryption should be used for the TCP connections. Defaults to false.

- **tls (TlsConfig):** A sub-section that specifies the settings to be used for any SSL/TLS encryption. This will only have any impact if *use_tls* is set to true. See *Configuring TLS*.
- **net (string, optional, default: "tcp")** Network value must be one of: "tcp", "tcp4", "tcp6", "unix" or "unixpacket".

New in version 0.6.

- **keep_alive (bool):** Specifies whether or not TCP keepalive should be used for established TCP connections. Defaults to false.
- **keep_alive_period (int):** Time duration in seconds that a TCP connection will be maintained before keepalive probes start being sent. Defaults to 7200 (i.e. 2 hours).

New in version 0.9.

- **splitter (string):** Defaults to "HekaFramingSplitter".

Example:

```
[TcpInput]
address = ":5565"
```

UDP Input

Plugin Name: **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.

Config:

- **address (string):** An IP address:port or Unix datagram socket file path 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 0.5.

- **net (string, optional, default: "udp")** Network value must be one of: "udp", "udp4", "udp6", or "unixgram".

Example:

```
[UdpInput]
address = "127.0.0.1:4880"
splitter = "HekaFramingSplitter"
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"
```

2.5 Splitters

New in version 0.9.

2.5.1 Common Splitter Parameters

There are some configuration options that are universally available to all Heka splitter 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.

- **keep_truncated (bool, optional):** If true, then any records that exceed the capacity of the input buffer will still be delivered in their truncated form. If false, then these records will be dropped. Defaults to false.
- **use_message_bytes (bool, optional):** Most decoders expect to find the raw, undecoded input data stored as the payload of the received Heka Message struct. Some decoders, however, such as the ProtobufDecoder, expect to receive a blob of bytes representing an entire Message struct, not just the payload. In this case, the data is expected to be found on the MsgBytes attribute of the Message's PipelinePack. If *use_message_bytes* is true, then the data will be written as a byte slice to the MsgBytes attribute, otherwise it will be written as a string to the Message payload. Defaults to false in most cases, but defaults to true for the HekaFramingSplitter, which is almost always used with the ProtobufDecoder.
- **min_buffer_size (uint, optional):** The initial size, in bytes, of the internal buffer that the SplitterRunner will use for buffering data streams. Must not be greater than the globally configured max_message_size. Defaults to 8KiB, although certain splitters may specify a different default.
- **deliver_incomplete_final (bool, optional):** When a splitter is used to split a stream, that stream can end part way through a record. It's sometimes appropriate to drop that data, but in other cases the incomplete data can still be useful. If 'deliver_incomplete_final' is set to true, then when the SplitterRunner's SplitStream method is used a delivery attempt will be made with any partial record data that may come through immediately before an EOF. Defaults to false.

2.5.2 Available Splitter Plugins

. _config_heka_framing_splitter

Heka Framing Splitter

Plugin Name: **HekaFramingSplitter**

A HekaFramingSplitter is used to split streams of data that use Heka's built- in *Stream Framing*, with a protocol buffers encoded message header supporting HMAC key authentication.

A default configuration of the HekaFramingSplitter is automatically registered as an available splitter plugin as "HekaFramingSplitter", so it is only necessary to add an additional TOML section if you want to use an instance of the splitter with settings other than the default.

Config:

- **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.
- **use_message_bytes (bool, optional):** The HekaFramingSplitter is almost always used in concert with an instance of ProtobufDecoder, which expects the protocol buffer message data to be available in the PipelinePack's MsgBytes attribute, so *use_message_bytes* defaults to true.

- **skip_authentication (bool, optional):** Usually if a HekaFramingSplitter identifies an incorrectly signed message, that message will be silently dropped. In some cases, however, such as when loading a stream of protobuf encoded Heka messages from a file system file, it may be desirable to skip authentication altogether. Setting this to true will do so. Defaults to false.

Example:

```
[acl_splitter]
type = "HekaFramingSplitter"

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

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

[tcp_control]
type = "TcpInput"
address = ":5566"
splitter = "acl_splitter"
```

Null Splitter

Plugin Name: **NullSplitter**

The NullSplitter is used in cases where the incoming data is already naturally divided into logical messages, such that Heka doesn't need to do any further splitting. For instance, when used in conjunction with a UdpInput, the contents of each UDP packet will be made into a separate message.

Note that this means generally the NullSplitter should not be used with a stream oriented input transport, such as with TcpInput or LogstreamerInput. If this is done then the splitting will be arbitrary, each message will contain whatever happens to be the contents of a particular read operation.

The NullSplitter has no configuration options, and is automatically registered as an available splitter plugin of the name "NullSplitter", so it doesn't require a separate TOML configuration section.

Regex Splitter

Plugin Name: **RegexSplitter**

A RegexSplitter considers any text that matches a specified regular expression to represent a boundary on which records should be split. The regular expression may consist of exactly one capture group. If a capture group is specified, then the captured text will be included in the returned record. If not, then the returned record will not include the text that caused the regular expression match.

Config:

- **delimiter (string)** Regular expression to be used as the record boundary. May contain zero or one specified capture groups.
- **delimiter_eol (bool, optional):** Specifies whether the contents of a delimiter capture group should be appended to the end of a record (true) or prepended to the beginning (false). Defaults to true. If the delimiter expression does not specify a capture group, this will have no effect.

Example:

```
[mysql_slow_query_splitter]
type = "RegexSplitter"
delimiter = '\n(# User@Host:)'
delimiter_eol = false
```

Token Splitter

Plugin Name: **TokenSplitter**

A TokenSplitter is used to split an incoming data stream on every occurrence (or every Nth occurrence) of a single, one byte token character. The token will be included as the final character in the returned record.

A default configuration of the TokenSplitter (i.e. splitting on every newline) is automatically registered as an available splitter plugin as "TokenSplitter", so additional TOML sections don't need to be added unless you want to use different settings.

Config:

- **delimiter (string, optional):** String representation of the byte token to be used as message delimiter. Defaults to *"\n"*.
- **count (uint, optional):** Number of instances of the delimiter that should be encountered before returning a record. Defaults to 1. Setting to 0 has no effect, 0 and 1 will be treated identically. Often used in conjunction with the *deliver_incomplete_final* option set to true, to ensure trailing partial records are still delivered.

Example:

```
[split_on_space]
type = "TokenSplitter"
delimiter = " "

[split_every_50th_newline_keep_partial]
type = "TokenSplitter"
count = 50
deliver_incomplete_final = true
```

2.6 Decoders

2.6.1 Available Decoder Plugins

Apache Access Log Decoder

New in version 0.6.

Plugin Name: **SandboxDecoder**
File Name: **lua_decoders/apache_access.lua**

Parses the Apache access logs based on the Apache 'LogFormat' configuration directive. The Apache format specifiers are mapped onto the Nginx variable names where applicable e.g. %a -> remote_addr. This allows generic web filters and outputs to work with any HTTP server input.

Config:

- **log_format (string)** The 'LogFormat' configuration directive from the apache2.conf. %t variables are converted to the number of nanosecond since the Unix epoch and used to set the Timestamp on the message. http://httpd.apache.org/docs/2.4/mod/mod_log_config.html
- **type (string, optional, default nil):** Sets the message 'Type' header to the specified value
- **user_agent_transform (bool, optional, default false)** Transform the http_user_agent into user_agent_browser, user_agent_version, user_agent_os.
- **user_agent_keep (bool, optional, default false)** Always preserve the http_user_agent value if transform is enabled.
- **user_agent_conditional (bool, optional, default false)** Only preserve the http_user_agent value if transform is enabled and fails.
- **payload_keep (bool, optional, default false)** Always preserve the original log line in the message payload.

Example Heka Configuration

```
[TestWebserver]
type = "LogstreamerInput"
log_directory = "/var/log/apache"
file_match = 'access\.log'
decoder = "CombinedLogDecoder"

[CombinedLogDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/apache_access.lua"

[CombinedLogDecoder.config]
type = "combined"
user_agent_transform = true
# combined log format
log_format = '%h %l %u %t \"%r\" %>s %O \"%{Referer}i\" \"%{User-Agent}i\"'

# common log format
# log_format = '%h %l %u %t \"%r\" %>s %O'

# vhost_combined log format
# log_format = '%v:%p %h %l %u %t \"%r\" %>s %O \"%{Referer}i\" \"%{User-Agent}i\"'

# referer log format
# log_format = '%{Referer}i -> %U'
```

Example Heka Message

Timestamp 2014-01-10 07:04:56 -0800 PST

Type combined

Hostname test.example.com

Pid 0

UUID 8e414f01-9d7f-4a48-a5e1-ae92e5954df5

Logger TestWebserver

Payload

EnvVersion

Severity 7

Fields

name:"remote_user" value_string:"-"
name:"http_x_forwarded_for" value_string:"-"
name:"http_referer" value_string:"-"
name:"body_bytes_sent" value_type:DOUBLE representation:"B" value_double:82
name:"remote_addr" value_string:"62.195.113.219" representation:"ipv4"
name:"status" value_type:DOUBLE value_double:200
name:"request" value_string:"GET /v1/recovery_email/status HTTP/1.1"
name:"user_agent_os" value_string:"FirefoxOS"
name:"user_agent_browser" value_string:"Firefox"
name:"user_agent_version" value_type:DOUBLE value_double:29

Geo IP Decoder

New in version 0.6.

Plugin Name: **GeoIpDecoder**

Decoder plugin that generates GeoIP data based on the IP address of a specified field. It uses the GeoIP Go project as a wrapper around MaxMind's geoip-api-c library, and thus assumes you have the library downloaded and installed. Currently, only the GeoLiteCity database is supported, which you must also download and install yourself into a location to be referenced by the db_file config option. By default the database file is opened using "GEOIP_MEMORY_CACHE" mode. This setting is hard- coded into the wrapper's geoip.go file. You will need to manually override that code if you want to specify one of the other modes listed here.

Note: Due to external dependencies, this plugin is not compiled in to the released Heka binaries. It will automatically be included in a *source build* if GeoIP.h is available in the include path during build time. The generated binary will then only work on machines with the appropriate GeoIP shared library (e.g. *libGeoIP.so.1*) installed.

Note: If you are using this with the ES output you will likely need to specify the raw_bytes_field option for the target_field specified. This is required to preserve the formatting of the JSON object.

Config:

- **db_file:** The location of the GeoLiteCity.dat database. Defaults to "/var/cache/hekad/GeoLiteCity.dat"
- **source_ip_field:** The name of the field containing the IP address you want to derive the location for.
- **target_field:** The name of the new field created by the decoder. The decoder will output a JSON object with the following elements:
  - latitute: string,
  - longitude: string,
  - **location: [float64, float64],**
    - GeoJSON format intended for use as a geo_point for ES output. Useful when using Kibana's Bettermap panel
  - coordinates: [string, string],
  - countrycode: string,
  - countrycode3: string,
  - region: string,

  - city: string,
  - postalcode: string,
  - areacode: int,
  - charset: int,
  - continentalcode: string

```
[apache_geoip_decoder]
type = "GeoIpDecoder"
db_file="/etc/geoip/GeoLiteCity.dat"
source_ip_field="remote_host"
target_field="geoip"
```

Graylog Extended Log Format Decoder

New in version 0.8.

Plugin Name: **SandboxDecoder**
File Name: **lua_decoders/graylog_extended.lua**

Parses a payload containing JSON in the Graylog2 Extended Format specficiation. http://graylog2.org/resources/gelf/specification

Config:

- **type (string, optional, default nil):** Sets the message 'Type' header to the specified value
- **payload_keep (bool, optional, default false)** Always preserve the original log line in the message payload.

Example of Graylog2 Exteded Format Log

```
{
  "version": "1.1",
  "host": "rogueethic.com",
  "short_message": "This is a short message to identify what is going on.",
  "full_message": "An entire backtrace\ncould\ngo\nhere",
  "timestamp": 1385053862.3072,
  "level": 1,
  "_user_id": 9001,
  "_some_info": "foo",
  "_some_env_var": "bar"
}
```

Example Heka Configuration

```
[GELFLogInput]
type = "LogstreamerInput"
log_directory = "/var/log"
file_match = 'application\.gelf'
decoder = "GraylogDecoder"

[GraylogDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/graylog_decoder.lua"
```

```
    [GraylogDecoder.config]
    type = "gelf"
    payload_keep = true
```

Linux CPU Stats Decoder

New in version 0.10.

Plugin Name: **SandboxDecoder**
File Name: **lua_decoders/linux_procstat.lua**

Parses a payload containing the contents of file */proc/stat*.

Config:

- **payload_keep (bool, optional, default false)** Always preserve the original log line in the message payload.

Example Heka Configuration

```
[ProcStats]
type = "FilePollingInput"
ticker_interval = 1
file_path = "/proc/stat"
decoder = "ProcStatDecoder"

[ProcStatDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/linux_procstat.lua"
```

Example Heka Message

Timestamp 2014-12-10 22:38:24 +0000 UTC

Type stats.proc

Hostname yourhost.net

Pid 0

Uuid d2546942-7c36-4042-ad2e-f6bfdac11cdb

Logger

Payload

EnvVersion

Severity 7

Fields

name:"cpu" type:double value:[14384,125,3330,946000,333,0,356,0,0,0]
name:"cpu[1-#]" type:double value:[14384,125,3330,946000,333,0,356,0,0,0]
name:"ctxt" type:double value:2808304
name:"btime" type:double value:1423004780
name:"intr" type:double value:[14384,125,3330,0,0,0,0,0,0,0...0]
name:"processes" type:double value:3811
name:"procs_running" type:double value:1

name:"procs_blocked" type:double value:0
name:"softirq" type:double value:[288977,23,101952,19,13046,19217,7,...]

Cpu fields: 1 2 3 4 5 6 7 8 9 10 user nice system idle [iowait] [irq] [softirq] [steal] [guest] [guestnice]
Note: systems provide user, nice, system, idle. Other fields depend on kernel.

intr This line shows counts of interrupts serviced since boot time, for each of the possible system interrupts. The first column is the total of all interrupts serviced including unnumbered architecture specific interrupts; each subsequent column is the total for that particular numbered interrupt. Unnumbered interrupts are not shown, only summed into the total.

Linux Disk Stats Decoder

New in version 0.7.

Plugin Name: **SandboxDecoder**
File Name: **lua_decoders/linux_diskstats.lua**

Parses a payload containing the contents of a */sys/block/$DISK/stat* file (where *$DISK* is a disk identifier such as *sda*) into a Heka message struct. This also tries to obtain the TickerInterval of the input it recieved the data from, by extracting it from a message field named *TickerInterval*.

Config:

- **payload_keep (bool, optional, default false)** Always preserve the original log line in the message payload.

Example Heka Configuration

```
[DiskStats]
type = "FilePollingInput"
ticker_interval = 1
file_path = "/sys/block/sda1/stat"
decoder = "DiskStatsDecoder"

[DiskStatsDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/linux_diskstats.lua"
```

Example Heka Message

Timestamp 2014-01-10 07:04:56 -0800 PST

Type stats.diskstats

Hostname test.example.com

Pid 0

UUID 8e414f01-9d7f-4a48-a5e1-ae92e5954df5

Payload

EnvVersion

Severity 7

Fields

name:"ReadsCompleted" value_type:DOUBLE value_double:"20123"
name:"ReadsMerged" value_type:DOUBLE value_double:"11267"

name:"SectorsRead" value_type:DOUBLE value_double:"1.094968e+06"
name:"TimeReading" value_type:DOUBLE value_double:"45148"
name:"WritesCompleted" value_type:DOUBLE value_double:"1278"
name:"WritesMerged" value_type:DOUBLE value_double:"1278"
name:"SectorsWritten" value_type:DOUBLE value_double:"206504"
name:"TimeWriting" value_type:DOUBLE value_double:"3348"
name:"TimeDoingIO" value_type:DOUBLE value_double:"4876"
name:"WeightedTimeDoingIO" value_type:DOUBLE value_double:"48356"
name:"NumIOInProgress" value_type:DOUBLE value_double:"3"
name:"TickerInterval" value_type:DOUBLE value_double:"2"
name:"FilePath" value_string:"/sys/block/sda/stat"

Linux Load Average Decoder

New in version 0.7.

Plugin Name: **SandboxDecoder**
File Name: **lua_decoders/linux_loadavg.lua**

Parses a payload containing the contents of a */proc/loadavg* file into a Heka message.

Config:

- **payload_keep (bool, optional, default false)** Always preserve the original log line in the message payload.

Example Heka Configuration

```
[LoadAvg]
type = "FilePollingInput"
ticker_interval = 1
file_path = "/proc/loadavg"
decoder = "LoadAvgDecoder"

[LoadAvgDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/linux_loadavg.lua"
```

Example Heka Message

Timestamp 2014-01-10 07:04:56 -0800 PST

Type stats.loadavg

Hostname test.example.com

Pid 0

UUID 8e414f01-9d7f-4a48-a5e1-ae92e5954df5

Payload

EnvVersion

Severity 7

Fields

name:”1MinAvg” value_type:DOUBLE value_double:”3.05”
name:”5MinAvg” value_type:DOUBLE value_double:”1.21”
name:”15MinAvg” value_type:DOUBLE value_double:”0.44”
name:”NumProcesses” value_type:DOUBLE value_double:”11”
name:”FilePath” value_string:”/proc/loadavg”

Linux Memory Stats Decoder

New in version 0.7.

Plugin Name: **SandboxDecoder** File Name: **lua_decoders/linux_memstats.lua**

Parses a payload containing the contents of a */proc/meminfo* file into a Heka message.

Config:

- **payload_keep (bool, optional, default false)** Always preserve the original log line in the message payload.

Example Heka Configuration

```
[MemStats]
type = "FilePollingInput"
ticker_interval = 1
file_path = "/proc/meminfo"
decoder = "MemStatsDecoder"

[MemStatsDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/linux_memstats.lua"
```

Example Heka Message

Timestamp 2014-01-10 07:04:56 -0800 PST

Type stats.memstats

Hostname test.example.com

Pid 0

UUID 8e414f01-9d7f-4a48-a5e1-ae92e5954df5

Payload

EnvVersion

Severity 7

Fields

name:”MemTotal” value_type:DOUBLE representation:”kB” value_double:”4047616”
name:”MemFree” value_type:DOUBLE representation:”kB” value_double:”3432216”
name:”Buffers” value_type:DOUBLE representation:”kB” value_double:”82028”
name:”Cached” value_type:DOUBLE representation:”kB” value_double:”368636”
name:”FilePath” value_string:”/proc/meminfo”

The total available fields can be found in *man procfs*. All fields are of type double, and the representation is in kB (except for the HugePages fields). Here is a full list of fields available:

MemTotal, MemFree, Buffers, Cached, SwapCached, Active, Inactive, Active(anon), Inactive(anon), Active(file), Inactive(file), Unevictable, Mlocked, SwapTotal, SwapFree, Dirty, Writeback, AnonPages, Mapped, Shmem, Slab,

SReclaimable, SUnreclaim, KernelStack, PageTables, NFS_Unstable, Bounce, WritebackTmp, CommitLimit, Committed_AS, VmallocTotal, VmallocUsed, VmallocChunk, HardwareCorrupted, AnonHugePages, HugePages_Total, HugePages_Free, HugePages_Rsvd, HugePages_Surp, Hugepagesize, DirectMap4k, DirectMap2M, DirectMap1G.

Note that your available fields may have a slight variance depending on the system's kernel version.

MultiDecoder

Plugin Name: **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.

Config:

- **subs ([]string):** An ordered list of subdecoders to which the MultiDecoder will delegate. Each item in the list should specify another decoder configuration section by section name. Must contain at least one entry.
- **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.

Here is a slightly contrived example where we have protocol buffer encoded messages coming in over a TCP connection, with each message containin a single nginx log line. Our MultiDecoder will run each message through two decoders, the first to deserialize the protocol buffer and the second to parse the log text:

```
[TcpInput]
address = ":5565"
parser_type = "message.proto"
decoder = "shipped-nginx-decoder"

[shipped-nginx-decoder]
type = "MultiDecoder"
subs = ['ProtobufDecoder', 'nginx-access-decoder']
cascade_strategy = "all"
log_sub_errors = true

[ProtobufDecoder]

[nginx-access-decoder]
type = "SandboxDecoder"
filename = "lua_decoders/nginx_access.lua"

    [nginx-access-decoder.config]
    type = "combined"
    user_agent_transform = true
    log_format = '$remote_addr - $remote_user [$time_local] "$request" $status $body_bytes_sent "$htt
```

MySQL Slow Query Log Decoder

New in version 0.6.

Plugin Name: **SandboxDecoder**
File Name: **lua_decoders/mysql_slow_query.lua**

Parses and transforms the MySQL slow query logs. Use mariadb_slow_query.lua to parse the MariaDB variant of the MySQL slow query logs.

Config:

- **truncate_sql (int, optional, default nil)** Truncates the SQL payload to the specified number of bytes (not UTF-8 aware) and appends "...". If the value is nil no truncation is performed. A negative value will truncate the specified number of bytes from the end.

Example Heka Configuration

```
[Sync-1_5-SlowQuery]
type = "LogstreamerInput"
log_directory = "/var/log/mysql"
file_match = 'mysql-slow\.log'
parser_type = "regexp"
delimiter = "\n(# User@Host:)"
delimiter_location = "start"
decoder = "MySqlSlowQueryDecoder"

[MySqlSlowQueryDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/mysql_slow_query.lua"

    [MySqlSlowQueryDecoder.config]
    truncate_sql = 64
```

Example Heka Message

Timestamp 2014-05-07 15:51:28 -0700 PDT

Type mysql.slow-query

Hostname 127.0.0.1

Pid 0

UUID 5324dd93-47df-485b-a88e-429f0fcd57d6

Logger Sync-1_5-SlowQuery

Payload /* [queryName=FIND_ITEMS] */ SELECT bso.userid, bso.collection, ...

EnvVersion

Severity 7

Fields

name:"Rows_examined" value_type:DOUBLE value_double:16458
name:"Query_time" value_type:DOUBLE representation:"s" value_double:7.24966
name:"Rows_sent" value_type:DOUBLE value_double:5001
name:"Lock_time" value_type:DOUBLE representation:"s" value_double:0.047038

Nginx Access Log Decoder

New in version 0.5.

Plugin Name: **SandboxDecoder**
File Name: **lua_decoders/nginx_access.lua**

Parses the Nginx access logs based on the Nginx 'log_format' configuration directive.

Config:

- **log_format (string)** The 'log_format' configuration directive from the nginx.conf. $time_local or $time_iso8601 variable is converted to the number of nanosecond since the Unix epoch and used to set the Timestamp on the message. http://nginx.org/en/docs/http/ngx_http_log_module.html
- **type (string, optional, default nil):** Sets the message 'Type' header to the specified value
- **user_agent_transform (bool, optional, default false)** Transform the http_user_agent into user_agent_browser, user_agent_version, user_agent_os.
- **user_agent_keep (bool, optional, default false)** Always preserve the http_user_agent value if transform is enabled.
- **user_agent_conditional (bool, optional, default false)** Only preserve the http_user_agent value if transform is enabled and fails.
- **payload_keep (bool, optional, default false)** Always preserve the original log line in the message payload.

Example Heka Configuration

```
[TestWebserver]
type = "LogstreamerInput"
log_directory = "/var/log/nginx"
file_match = 'access\.log'
decoder = "CombinedLogDecoder"

[CombinedLogDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/nginx_access.lua"

[CombinedLogDecoder.config]
type = "combined"
user_agent_transform = true
# combined log format
log_format = '$remote_addr - $remote_user [$time_local] "$request" $status $body_bytes_sent "$http_re
```

Example Heka Message

Timestamp 2014-01-10 07:04:56 -0800 PST

Type combined

Hostname test.example.com

Pid 0

UUID 8e414f01-9d7f-4a48-a5e1-ae92e5954df5

Logger TestWebserver

Payload

EnvVersion

Severity 7

Fields

name:"remote_user" value_string:"-"
name:"http_x_forwarded_for" value_string:"-"
name:"http_referer" value_string:"-"
name:"body_bytes_sent" value_type:DOUBLE representation:"B" value_double:82
name:"remote_addr" value_string:"62.195.113.219" representation:"ipv4"
name:"status" value_type:DOUBLE value_double:200
name:"request" value_string:"GET /v1/recovery_email/status HTTP/1.1"
name:"user_agent_os" value_string:"FirefoxOS"
name:"user_agent_browser" value_string:"Firefox"
name:"user_agent_version" value_type:DOUBLE value_double:29

Nginx Error Log Decoder

New in version 0.6.

Plugin Name: **SandboxDecoder**
File Name: **lua_decoders/nginx_error.lua**

Parses the Nginx error logs based on the Nginx hard coded internal format.

Config:

- **tz (string, optional, defaults to UTC)** The conversion actually happens on the Go side since there isn't good TZ support here.

Example Heka Configuration

```
[TestWebserverError]
type = "LogstreamerInput"
log_directory = "/var/log/nginx"
file_match = 'error\.log'
decoder = "NginxErrorDecoder"

[NginxErrorDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/nginx_error.lua"

[NginxErrorDecoder.config]
tz = "America/Los_Angeles"
```

Example Heka Message

Timestamp 2014-01-10 07:04:56 -0800 PST

Type nginx.error

Hostname trink-x230

Pid 16842

UUID 8e414f01-9d7f-4a48-a5e1-ae92e5954df5

Logger TestWebserverError

Payload using inherited sockets from "6;"

EnvVersion

Severity 5

Fields

name:"tid" value_type:DOUBLE value_double:0
name:"connection" value_type:DOUBLE value_double:8878

Payload Regex Decoder

Plugin Name: **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.

Note: The Go regular expression tester is an invaluable tool for constructing and debugging regular expressions to be used for parsing your input data.

Config:

- **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. See *Heka Message*.
- **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 (see *Field Variables*).

  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 documentation. In addition to the Go time formatting, special *timestamp_layout* values of "Epoch", "EpochMilli", "EpochMicro", and "EpochNano" are supported for Unix style timestamps represented in seconds, milliseconds, microseconds, and nanoseconds since the Epoch, respectively.
- **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*. This setting will have no impact if one of the supported "Epoch*" values is used as the *timestamp_layout* setting.
- **log_errors (bool):** New in version 0.5.

  If set to false, payloads that can not be matched against the regex will not be logged as errors. Defaults to true.

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]
timestamp_layout = "02/Jan/2006:15:04:05 -0700"

# severities in this case would work only if a (?P<Severity>...) matching
# group was present in the regex, and the log file contained this information.
[apache_transform_decoder.severity_map]
DEBUG = 7
INFO = 6
WARNING = 4

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

Payload XML Decoder

Plugin Name: **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 pipeline pack message using XPath syntax using the xmlpath library.

Config:

- **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. See *Heka Message*.
- **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 (see *Field Variables*).

  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 documentation.

The default layout is ISO8601 - the same as Javascript. In addition to the Go time formatting, special *timestamp_layout* values of "Epoch", "EpochMilli", "EpochMicro", and "EpochNano" are supported for Unix style timestamps represented in seconds, milliseconds, microseconds, and nanoseconds since the Epoch, respectively.

- **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*. This setting will have no impact if one of the supported "Epoch*" values is used as the *timestamp_layout* setting.

Example:

```
[myxml_decoder]
type = "PayloadXmlDecoder"

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

[myxml_decoder.severity_map]
DEBUG = 7
INFO = 6
WARNING = 4

[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

Protobuf Decoder

Plugin Name: **ProtobufDecoder**

The ProtobufDecoder is used for Heka message objects that have been serialized into protocol buffers format. This is the format that Heka uses to communicate with other Heka instances, so one will always be included in your Heka configuration under the name "ProtobufDecoder", whether specified or not. The ProtobufDecoder has no configuration options.

The hekad protocol buffers message schema in defined in the *message.proto* file in the *message* package.

Example:

```
[ProtobufDecoder]
```

See also:

Protocol Buffers - Google's data interchange format

Rsyslog Decoder

New in version 0.5.

Plugin Name: **SandboxDecoder**
File Name: **lua_decoders/rsyslog.lua**

Parses the rsyslog output using the string based configuration template.

Config:

- **hostname_keep (boolean, defaults to false)** Always preserve the original 'Hostname' field set by Logstreamer's 'hostname' configuration setting.
- **template (string)** The 'template' configuration string from rsyslog.conf. http://rsyslog-5-8-6-doc.neocities.org/rsyslog_conf_templates.html
- **tz (string, optional, defaults to UTC)** If your rsyslog timestamp field in the template does not carry zone offset information, you may set an offset to be applied to your events here. Typically this would be used with the "Traditional" rsyslog formats.

  Parsing is done by Go, supports values of "UTC", "Local", or a location name corresponding to a file in the IANA Time Zone database, e.g. "America/New_York".

Example Heka Configuration

```
[RsyslogDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/rsyslog.lua"

[RsyslogDecoder.config]
type = "RSYSLOG_TraditionalFileFormat"
template = '%TIMESTAMP% %HOSTNAME% %syslogtag%%msg:::sp-if-no-1st-sp%%msg:::drop-last-lf%\n'
tz = "America/Los_Angeles"
```

Example Heka Message

Timestamp 2014-02-10 12:58:58 -0800 PST

Type RSYSLOG_TraditionalFileFormat

Hostname trink-x230

Pid 0

UUID e0eef205-0b64-41e8-a307-5772b05e16c1

Logger RsyslogInput

Payload "imklog 5.8.6, log source = /proc/kmsg started."

EnvVersion

Severity 7

Fields

name:"programname" value_string:"kernel"

Sandbox Decoder

Plugin Name: **SandboxDecoder**

The SandboxDecoder provides an isolated execution environment for data parsing and complex transformations without the need to recompile Heka. See *Sandbox*. Config:

- *Common Sandbox Parameters*

Example

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

Scribble Decoder

New in version 0.5.

Plugin Name: **ScribbleDecoder**

The ScribbleDecoder is a trivial decoder that makes it possible to set one or more static field values on every decoded message. It is often used in conjunction with another decoder (i.e. in a MultiDecoder w/ cascade_strategy set to "all") to, for example, set the message type of every message to a specific custom value after the messages have been decoded from Protocol Buffers format. Note that this only supports setting the exact same value on every message, if any dynamic computation is required to determine what the value should be, or whether it should be applied to a specific message, a *Sandbox Decoder* using the provided *write_message* API call should be used instead.

Config:

- **message_fields:** Subsection defining message fields to populate. Optional representation metadata can be added at the end of the field name using a pipe delimiter i.e. *host|ipv4 = "192.168.55.55"* will create Fields[Host] containing an IPv4 address. 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 (see *Field Variables*). Does not support Timestamp or Uuid.

Example (in MultiDecoder context)

```
[mytypedecoder]
type = "MultiDecoder"
subs = ["ProtobufDecoder", "mytype"]
cascade_strategy = "all"
log_sub_errors = true

[ProtobufDecoder]

[mytype]
type = "ScribbleDecoder"

    [mytype.message_fields]
    Type = "MyType"
```

Stats To Fields Decoder

New in version 0.4.

Plugin Name: **StatsToFieldsDecoder**

The StatsToFieldsDecoder will parse time series statistics data in the graphite message format and encode the data into the message fields, in the same format produced by a *Stat Accumulator Input* plugin with the *emit_in_fields* value set to true. This is useful if you have externally generated graphite 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 messages 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.

Example:

```
[StatsToFieldsDecoder]
```

2.7 Filters

2.7.1 Common Filter Parameters

There are some configuration options that are universally available to all Heka filter 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.

New in version 0.7.

- **can_exit (bool, optional)** Whether or not this plugin can exit without causing Heka to shutdown. Defaults to false for non-sandbox filters, and true for sandbox filters.

New in version 0.10.

- **use_buffering (bool, optional)** If true, all messages delivered to this filter will be buffered to disk before delivery, preventing back pressure and allowing retries in cases of message processing failure. Defaults to false, unless otherwise specified by the individual filter's documentation.
- **buffering (QueueBufferConfig, optional)** A sub-section that specifies the settings to be used for the buffering behavior. This will only have any impact if *use_buffering* is set to true. See *Configuring Buffering*.

2.7.2 Available Filter Plugins

Circular Buffer Delta Aggregator

New in version 0.5.

Plugin Name: **SandboxFilter**
File Name: **lua_filters/cbufd_aggregator.lua**

Collects the circular buffer delta output from multiple instances of an upstream sandbox filter (the filters should all be the same version at least with respect to their cbuf output). The purpose is to recreate the view at a larger scope in each level of the aggregation i.e., host view -> datacenter view -> service level view.

Config:

- **enable_delta (bool, optional, default false)** Specifies whether or not this aggregator should generate cbuf deltas.
- **anomaly_config(string) - (see *Anomaly Detection Module*)** A list of anomaly detection specifications. If not specified no anomaly detection/alerting will be performed.
- **preservation_version (uint, optional, default 0)** If *preserve_data = true* is set in the SandboxFilter configuration, then this value should be incremented every time the *enable_delta* configuration is changed to prevent the plugin from failing to start during data restoration.

Example Heka Configuration

```
[TelemetryServerMetricsAggregator]
type = "SandboxFilter"
message_matcher = "Logger == 'TelemetryServerMetrics' && Fields[payload_type] == 'cbufd'"
ticker_interval = 60
filename = "lua_filters/cbufd_aggregator.lua"
preserve_data = true

[TelemetryServerMetricsAggregator.config]
enable_delta = false
anomaly_config = 'roc("Request Statistics", 1, 15, 0, 1.5, true, false)'
preservation_version = 0
```

CBuf Delta Aggregator By Hostname

New in version 0.5.

Plugin Name: **SandboxFilter**
File Name: **lua_filters/cbufd_host_aggregatory.lua**

Collects the circular buffer delta output from multiple instances of an upstream sandbox filter (the filters should all be the same version at least with respect to their cbuf output). Each column from the source circular buffer will become its own graph. i.e., 'Error Count' will become a graph with each host being represented in a column.

Config:

- **max_hosts (uint)** Pre-allocates the number of host columns in the graph(s). If the number of active hosts exceed this value, the plugin will terminate.
- **rows (uint)** The number of rows to keep from the original circular buffer. Storing all the data from all the hosts is not practical since you will most likely run into memory and output size restrictions (adjust the view down as necessary).
- **host_expiration (uint, optional, default 120 seconds)** The amount of time a host has to be inactive before it can be replaced by a new host.

- **preservation_version (uint, optional, default 0)** If *preserve_data = true* is set in the SandboxFilter configuration, then this value should be incremented every time the *max_hosts* or *rows* configuration is changed to prevent the plugin from failing to start during data restoration.

Example Heka Configuration

```
[TelemetryServerMetricsHostAggregator]
type = "SandboxFilter"
message_matcher = "Logger == 'TelemetryServerMetrics' && Fields[payload_type] == 'cbufd'"
ticker_interval = 60
filename = "lua_filters/cbufd_host_aggregator.lua"
preserve_data = true

[TelemetryServerMetricsHostAggregator.config]
max_hosts = 5
rows = 60
host_expiration = 120
preservation_version = 0
```

Counter Filter

Plugin Name: **CounterFilter**

Once per ticker interval 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 interval (i.e. it counts the messages the plugin received). Every ten intervals an extra message (also of type *heka.counter-output*) goes out, containing an aggregate count and average per second throughput of messages received.

Config:

- **ticker_interval (int, optional):** Interval between generated counter messages, in seconds. Defaults to 5.

Example:

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

CPU Stats Filter

New in version 0.10.

Plugin Name: **SandboxFilter**
File Name: **lua_filters/procstat.lua**

Calculates deltas in /proc/stat data. Also emits CPU percentage utilization information.

Config:

- **whitelist (string, optional, default "")** Only process fields that fit the pattern, defaults to match all.
- **extras (boolean, optional, default false)** Process extra fields like ctxt, softirq, cpu fields.
- **percent_integer (boolean, optional, default true)** Process percentage as whole number.

Example Heka Configuration

```
[ProcStats]
type = "FilePollingInput"
ticker_interval = 1
file_path = "/proc/stat"
decoder = "ProcStatDecoder"

[ProcStatDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/linux_procstat.lua"

[ProcStatFilter]
type = "SandboxFilter"
filename = "lua_filters/procstat.lua"
preserve_data = true
message_matcher = "Type == 'stats.procstat'"
[ProcStatFilter.config]
    whitelist = "cpu$"
    extras = false
    percent_integer = true
```

Cpu fields: 1 2 3 4 5 6 7 8 9 10 user nice system idle [iowait] [irq] [softirq] [steal] [guest] [guestnice] Note: systems provide user, nice, system, idle. Other fields depend on kernel.

user: Time spent executing user applications (user mode). nice: Time spent executing user applications with low priority (nice). system: Time spent executing system calls (system mode). idle: Idle time. iowait: Time waiting for I/O operations to complete. irq: Time spent servicing interrupts. softirq: Time spent servicing soft-interrupts. steal: ticks spent executing other virtual hosts [virtualization setups] guest: Used in virtualization setups. guestnice: running a niced guest

intr This line shows counts of interrupts serviced since boot time, for each of the possible system interrupts. The first column is the total of all interrupts serviced including unnumbered architecture specific interrupts; each subsequent column is the total for that particular numbered interrupt. Unnumbered interrupts are not shown, only summed into the total.

ctxt 115315 The number of context switches that the system underwent.

btime 769041601 Boot time, in seconds since the Epoch, 1970-01-01 00:00:00 +0000 (UTC).

processes 86031 Number of forks since boot.

procs_running 6 Number of process in runnable state. (Linux 2.5.45 onward.)

procs_blocked 2 Number of process blocked waiting for I/O to complete. (Linux 2.5.45 onward.)

softirq 288977 23 101952 19 13046 19217 7 19125 92077 389 43122 Time spent servicing soft-interrupts.

Disk Stats Filter

New in version 0.7.

Plugin Name: **SandboxFilter**
File Name: **lua_filters/diskstats.lua**

Graphs disk IO stats. It automatically converts the running totals of Writes and Reads into rates of the values. The time based fields are left as running totals of the amount of time doing IO. Expects to receive messages with disk

IO data embedded in a particular set of message fields which matches what is generated by *Linux Disk Stats Decoder*: WritesCompleted, ReadsCompleted, SectorsWritten, SectorsRead, WritesMerged, ReadsMerged, TimeWriting, TimeReading, TimeDoingIO, WeightedTimeDoingIO, TickerInterval.

Config:

- **rows (uint, optional, default 1440)** Sets the size of the sliding window i.e., 1440 rows representing 60 seconds per row is a 24 sliding hour window with 1 minute resolution.
- anomaly_config(string) - (see *Anomaly Detection Module*)

Example Heka Configuration

```
[DiskStatsFilter]
type = "SandboxFilter"
filename = "lua_filters/diskstats.lua"
preserve_data = true
message_matcher = "Type == 'stats.diskstats'"
ticker_interval = 10
```

Frequent Items

New in version 0.5.

Plugin Name: **SandboxFilter**
File Name: **lua_filters/frequent_items.lua**

Calculates the most frequent items in a data stream.

Config:

- **message_variable (string)** The message variable name containing the items to be counted.
- **max_items (uint, optional, default 1000)** The maximum size of the sample set (higher will produce a more accurate list).
- **min_output_weight (uint, optional, default 100)** Used to reduce the long tail output by only outputting the higher frequency items.
- **reset_days (uint, optional, default 1)** Resets the list after the specified number of days (on the UTC day boundary). A value of 0 will never reset the list.

Example Heka Configuration

```
[FxaAuthServerFrequentIP]
type = "SandboxFilter"
filename = "lua_filters/frequent_items.lua"
ticker_interval = 60
preserve_data = true
message_matcher = "Logger == 'nginx.access' && Type == 'fxa-auth-server'"

[FxaAuthServerFrequentIP.config]
message_variable = "Fields[remote_addr]"
max_items = 10000
min_output_weight = 100
reset_days = 1
```

Heka Memory Statistics

New in version 0.6.

Plugin Name: **SandboxFilter**
File Name: **lua_filters/heka_memstat.lua**

Graphs the Heka memory statistics using the heka.memstat message generated by pipeline/report.go.

Config:

- **rows (uint, optional, default 1440)** Sets the size of the sliding window i.e., 1440 rows representing 60 seconds per row is a 24 sliding hour window with 1 minute resolution.
- **sec_per_row (uint, optional, default 60)** Sets the size of each bucket (resolution in seconds) in the sliding window.
- **anomaly_config (string, optional)** See *Anomaly Detection Module*.
- **preservation_version (uint, optional, default 0)** If *preserve_data = true* is set in the SandboxFilter configuration, then this value should be incremented every time the *rows* or *sec_per_row* configuration is changed to prevent the plugin from failing to start during data restoration.

Example Heka Configuration

```
[HekaMemstat]
type = "SandboxFilter"
filename = "lua_filters/heka_memstat.lua"
ticker_interval = 60
preserve_data = true
message_matcher = "Type == 'heka.memstat'"
```

HTTP Status Graph

New in version 0.5.

Plugin Name: **SandboxFilter**
File Name: **lua_filters/http_status.lua**

Graphs HTTP status codes using the numeric Fields[status] variable collected from web server access logs.

Config:

- **sec_per_row (uint, optional, default 60)** Sets the size of each bucket (resolution in seconds) in the sliding window.
- **rows (uint, optional, default 1440)** Sets the size of the sliding window i.e., 1440 rows representing 60 seconds per row is a 24 sliding hour window with 1 minute resolution.
- **anomaly_config (string, optional)** See *Anomaly Detection Module*.
- **alert_throttle (uint, optional, default 3600)** Sets the throttle for the anomaly alert, in seconds.

- **preservation_version (uint, optional, default 0)** If *preserve_data = true* is set in the SandboxFilter configuration, then this value should be incremented every time the *sec_per_row* or *rows* configuration is changed to prevent the plugin from failing to start during data restoration.

Example Heka Configuration

```
[FxaAuthServerHTTPStatus]
type = "SandboxFilter"
filename = "lua_filters/http_status.lua"
ticker_interval = 60
preserve_data = true
message_matcher = "Logger == 'nginx.access' && Type == 'fxa-auth-server'"

[FxaAuthServerHTTPStatus.config]
sec_per_row = 60
rows = 1440
anomaly_config = 'roc("HTTP Status", 2, 15, 0, 1.5, true, false) roc("HTTP Status", 4, 15, 0, 1.5, t
alert_throttle = 300
preservation_version = 0
```

Load Average Filter

New in version 0.7.

Plugin Name: **SandboxFilter**
File Name: **lua_filters/loadavg.lua**

Graphs the load average and process count data. Expects to receive messages containing fields entitled *1MinAvg*, *5MinAvg*, *15MinAvg*, and *NumProcesses*, such as those generated by the *Linux Load Average Decoder*.

Config:

- **sec_per_row (uint, optional, default 60)** Sets the size of each bucket (resolution in seconds) in the sliding window.
- **rows (uint, optional, default 1440)** Sets the size of the sliding window i.e., 1440 rows representing 60 seconds per row is a 24 sliding hour window with 1 minute resolution.
- **anomaly_config (string, optional)** See *Anomaly Detection Module*.
- **preservation_version (uint, optional, default 0)** If *preserve_data = true* is set in the SandboxFilter configuration, then this value should be incremented every time the *sec_per_row* or *rows* configuration is changed to prevent the plugin from failing to start during data restoration.

Example Heka Configuration

```
[LoadAvgFilter]
type = "SandboxFilter"
filename = "lua_filters/loadavg.lua"
ticker_interval = 60
preserve_data = true
message_matcher = "Type == 'stats.loadavg'"
```

Memory Stats Filter

New in version 0.7.

Plugin Name: **SandboxFilter**
File Name: **lua_filters/memstats.lua**

Graphs memory usage statistics. Expects to receive messages with memory usage data embedded in a specific set of message fields, which matches the messages generated by *Linux Memory Stats Decoder*: MemFree, Cached, Active, Inactive, VmallocUsed, Shmem, SwapCached.

Config:

- **sec_per_row (uint, optional, default 60)** Sets the size of each bucket (resolution in seconds) in the sliding window.
- **rows (uint, optional, default 1440)** Sets the size of the sliding window i.e., 1440 rows representing 60 seconds per row is a 24 sliding hour window with 1 minute resolution.
- **anomaly_config (string, optional)** See *Anomaly Detection Module*.
- **preservation_version (uint, optional, default 0)** If *preserve_data = true* is set in the SandboxFilter configuration, then this value should be incremented every time the *sec_per_row* or *rows* configuration is changed to prevent the plugin from failing to start during data restoration.

Example Heka Configuration

```
[MemoryStatsFilter]
type = "SandboxFilter"
filename = "lua_filters/memstats.lua"
ticker_interval = 60
preserve_data = true
message_matcher = "Type == 'stats.memstats'"
```

Heka Process Message Failures

New in version 0.7.

Plugin Name: **SandboxFilter**
File Name: **lua_filters/heka_process_message_failures.lua**

Monitors Heka's process message failures by plugin.

Config:

- **anomaly_config(string) - (see *Anomaly Detection Module*)** A list of anomaly detection specifications. If not specified a default of 'mww_nonparametric("DEFAULT", 1, 5, 10, 0.7)' is used. The "DEFAULT" settings are applied to any plugin without an explict specification.

Example Heka Configuration

```
[HekaProcessMessageFailures]
type = "SandboxFilter"
filename = "lua_filters/heka_process_message_failures.lua"
ticker_interval = 60
preserve_data = false # the counts are reset on Heka restarts and the monitoring should be too.
message_matcher = "Type == 'heka.all-report'"
```

Heka Message Schema

New in version 0.5.

Plugin Name: **SandboxFilter**
File Name: **lua_filters/heka_message_schema.lua**

Generates documentation for each unique message in a data stream. The output is a hierarchy of Logger, Type, EnvVersion, and a list of associated message field attributes including their counts (number in the brackets). This plugin is meant for data discovery/exploration and should not be left running on a production system.

Config:

<none>

Example Heka Configuration

```
[SyncMessageSchema]
type = "SandboxFilter"
filename = "lua_filters/heka_message_schema.lua"
ticker_interval = 60
preserve_data = false
message_matcher = "Logger != 'SyncMessageSchema' && Logger =~ /^Sync/"
```

Example Output

Sync-1_5-Webserver [54600]
    slf [54600]
        -no version- [54600]
            upstream_response_time (mismatch)
            http_user_agent (string)
            body_bytes_sent (number)
            remote_addr (string)
            request (string)
            upstream_status (mismatch)
            status (number)
            request_time (number)
            request_length (number)
Sync-1_5-SlowQuery [37]
    mysql.slow-query [37]
        -no version- [37]
            Query_time (number)
            Rows_examined (number)
            Rows_sent (number)
            Lock_time (number)

MySQL Slow Query

New in version 0.6.

Plugin Name: **SandboxFilter**
File Name: **lua_filters/mysql_slow_query.lua**

Graphs MySQL slow query data produced by the *MySQL Slow Query Log Decoder*.

Config:

- **sec_per_row (uint, optional, default 60)** Sets the size of each bucket (resolution in seconds) in the sliding window.
- **rows (uint, optional, default 1440)** Sets the size of the sliding window i.e., 1440 rows representing 60 seconds per row is a 24 sliding hour window with 1 minute resolution.
- **anomaly_config (string, optional)** See *Anomaly Detection Module*.
- **preservation_version (uint, optional, default 0)** If *preserve_data = true* is set in the SandboxFilter configuration, then this value should be incremented every time the *sec_per_row* or *rows* configuration is changed to prevent the plugin from failing to start during data restoration.

Example Heka Configuration

```
[Sync-1_5-SlowQueries]
type = "SandboxFilter"
message_matcher = "Logger == 'Sync-1_5-SlowQuery'"
ticker_interval = 60
filename = "lua_filters/mysql_slow_query.lua"

    [Sync-1_5-SlowQueries.config]
    anomaly_config = 'mww_nonparametric("Statistics", 5, 15, 10, 0.8)'
    preservation_version = 0
```

Sandbox Filter

Plugin Name: **SandboxFilter**

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

Config:

- *Common Filter Parameters*
- *Common Sandbox Parameters*
- **timer_event_on_shutdown (bool):** True if the sandbox should have its timer_event function called on shutdown.

Example:

```
[hekabench_counter]
type = "SandboxFilter"
message_matcher = "Type == 'hekabench'"
ticker_interval = 1
filename = "counter.lua"
preserve_data = true
profile = false

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

Sandbox Manager Filter

Plugin Name: **SandboxManagerFilter**

The SandboxManagerFilter provides dynamic control (start/stop) of sandbox filters in a secure manner without stopping the Heka daemon. Commands are sent to a SandboxManagerFilter 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.

SandboxManagerFilter Settings

- *Common Filter 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.
- **module_directory (string):** The directory where 'require' will attempt to load the external Lua modules from. Defaults to ${SHARE_DIR}/lua_modules.
- **max_filters (uint):** The maximum number of filters this manager can run.

New in version 0.5.

- **memory_limit (uint):** The number of bytes managed sandboxes are allowed to consume before being terminated (default 8MiB).
- **instruction_limit (uint):** The number of instructions managed sandboxes are allowed to execute during the process_message/timer_event functions before being terminated (default 1M).
- **output_limit (uint):** The number of bytes managed sandbox output buffers can hold before being terminated (default 63KiB). Warning: messages exceeding 64KiB will generate an error and be discarded by the standard output plugins (File, TCP, UDP) since they exceed the maximum message size.

Example

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

Stat Filter

Plugin Name: **StatFilter**

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

Config:

- Metric:

  Subsection defining a single metric to be generated. Both the *name* and *value* fields for each metric support interpolation of message field values (from 'Type', 'Hostname', 'Logger', 'Payload', or any dynamic field name) with the use of %% delimiters, so *%Hostname%* would be replaced by the message's Hostname field, and %Foo% would be replaced by the first value of a dynamic field called "Foo":

  - **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.
  - **replace_dot (boolean):** Replace all dots . per an underscore _ during the string interpolation. It's useful if you output this result in a graphite instance.

- **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:

```
[StatAccumInput]
ticker_interval = 5

[StatsdInput]
address = "127.0.0.1:29301"

[Hits]
type = "StatFilter"
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 StatAccumInput to be running.

Stats Graph

New in version 0.7.

Plugin Name: **SandboxFilter**
File Name: **lua_filters/stat_graph.lua**

Converts stat values extracted from statmetric messages (see *Stat Accumulator Input*) to circular buffer data and periodically emits messages containing this data to be graphed by a DashboardOutput. Note that this filter expects the stats data to be available in the message fields, so the StatAccumInput *must* be configured with *emit_in_fields* set to true for this filter to work correctly.

Config:

- **title (string, optional, default "Stats"):** Title for the graph output generated by this filter.
- **rows (uint, optional, default 300):** The number of rows to store in our circular buffer. Each row represents one time interval.
- **sec_per_row (uint, optional, default 1):** The number of seconds in each circular buffer time interval.

- **stats (string):** Space separated list of stat names. Each specified stat will be expected to be found in the fields of the received statmetric messages, and will be extracted and inserted into its own column in the accumulated circular buffer.
- **stat_labels (string):** Space separated list of header label names to use for the extracted stats. Must be in the same order as the specified stats. Any label longer than 15 characters will be truncated.
- **anomaly_config (string, optional):** Anomaly detection configuration, see *Anomaly Detection Module*.
- **preservation_version (uint, optional, default 0):** If *preserve_data = true* is set in the SandboxFilter configuration, then this value should be incremented every time any edits are made to your *rows*, *sec_per_row*, *stats*, or *stat_labels* values, or else Heka will fail to start because the preserved data will no longer match the filter's data structure.
- **stat_aggregation (string, optional, default "sum"):**

  Controls how the column data is aggregated when combining multiple circular buffers. "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. "none" - No aggregation will be performed the column.
- **stat_unit (string, optional, default "count"):** 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).

Example Heka Configuration

```
[stat-graph]
type = "SandboxFilter"
filename = "lua_filters/stat_graph.lua"
ticker_interval = 10
preserve_data = true
message_matcher = "Type == 'heka.statmetric'"

  [stat-graph.config]
  title = "Hits and Misses"
  rows = 1440
  stat_aggregation = "none"
  stat_unit = "count"
  sec_per_row = 10
  stats = "stats.counters.hits.count stats.counters.misses.count"
  stat_labels = "hits misses"
  anomaly_config = 'roc("Hits and Misses", 1, 15, 0, 1.5, true, false) roc("Hits and Misses", 2, 15,
  preservation_version = 0
```

Unique Items

New in version 0.6.

Plugin Name: **SandboxFilter**
File Name: **lua_filters/unique_items.lua**

Counts the number of unique items per day e.g. active daily users by uid.

Config:

- **message_variable (string, required)** The Heka message variable containing the item to be counted.

- **title (string, optional, default "Estimated Unique Daily *message_variable*")** The graph title for the cbuf output.
- **enable_delta (bool, optional, default false)** Specifies whether or not this plugin should generate cbuf deltas. Deltas should be enabled when sharding is used; see: *Circular Buffer Delta Aggregator*.
- **preservation_version (uint, optional, default 0)** If *preserve_data = true* is set in the SandboxFilter configuration, then this value should be incremented every time the *enable_delta* configuration is changed to prevent the plugin from failing to start during data restoration.

Example Heka Configuration

```
[FxaActiveDailyUsers]
type = "SandboxFilter"
filename = "lua_filters/unique_items.lua"
ticker_interval = 60
preserve_data = true
message_matcher = "Logger == 'FxaAuth' && Type == 'request.summary' && Fields[path] == '/v1/certifica

    [FxaActiveDailyUsers.config]
    message_variable = "Fields[uid]"
    title = "Estimated Active Daily Users"
    preservation_version = 0
```

2.8 Encoders

New in version 0.6.

2.8.1 Available Encoder Plugins

Alert Encoder

Plugin Name: **SandboxEncoder**
File Name: **lua_encoders/alert.lua**

Produces more human readable alert messages.

Config:

<none>

Example Heka Configuration

```
[FxaAlert]
type = "SmtpOutput"
message_matcher = "Type == 'heka.sandbox-output' && Fields[payload_type] == 'alert' && Logger =~ /^Fx
send_from = "heka@example.com"
send_to = ["alert@example.com"]
auth = "Plain"
user = "test"
password = "testpw"
host = "localhost:25"
encoder = "AlertEncoder"

[AlertEncoder]
```

```
type = "SandboxEncoder"
filename = "lua_encoders/alert.lua"
```

Example Output

Timestamp 2014-05-14T14:20:18Z

Hostname ip-10-226-204-51

Plugin FxaBrowserIdHTTPStatus

Alert HTTP Status - algorithm: roc col: 1 msg: detected anomaly, standard deviation exceeds 1.5

CBUF Librato Encoder

New in version 0.8.

Plugin Name: **SandboxEncoder**
File Name: **lua_encoders/cbuf_librato.lua**

Extracts data from SandboxFilter circular buffer output messages and uses it to generate time series JSON structures that will be accepted by Librato's POST API. It will keep track of the last time it's seen a particular message, keyed by filter name and output name. The first time it sees a new message, it will send data from all of the rows except the last one, which is possibly incomplete. For subsequent messages, the encoder will automatically extract data from all of the rows that have elapsed since the last message was received.

The SandboxEncoder *preserve_data* setting should be set to true when using this encoder, or else the list of received messages will be lost whenever Heka is restarted, possibly causing the same data rows to be sent to Librato multiple times.

Config:

- **message_key (string, optional, default "%{Logger}:%{payload_name}")** String to use as the key to differentiate separate cbuf messages from each other. Supports *message field interpolation*.

Example Heka Configuration

```
[cbuf_librato_encoder]
type = "SandboxEncoder"
filename = "lua_encoders/cbuf_librato.lua"
preserve_data = true
  [cbuf_librato_encoder.config]
  message_key = "%{Logger}:%{Hostname}:%{payload_name}"

[librato]
type = "HttpOutput"
message_matcher = "Type == 'heka.sandbox-output && Fields[payload_type] == 'cbuf'"
encoder = "cbuf_librato_encoder"
address = "https://metrics-api.librato.com/v1/metrics"
username = "username@example.com"
password = "SECRET"
    [librato.headers]
    Content-Type = ["application/json"]
```

Example Output

```
{"gauges":[{"value":12,"measure_time":1410824950,"name":"HTTP_200","source":"thor"},{"value":1,"measu
```

ElasticSearch JSON Encoder

Plugin Name: **ESJsonEncoder**

This encoder serializes a Heka message into a clean JSON format, preceded by a separate JSON structure containing information required for ElasticSearch BulkAPI indexing. The JSON serialization is done by hand, without the use of Go's stdlib JSON marshalling. This is so serialization can succeed even if the message contains invalid UTF-8 characters, which will be encoded as U+FFFD. Config:

- **index (string):** Name of the ES index into which the messages will be inserted. Supports interpolation of message field values (from 'Type', 'Hostname', 'Pid', 'UUID', 'Logger', 'EnvVersion', 'Severity', a field name, or a timestamp format) with the use of '%{}' chars, so '%{Hostname}-%{Logger}-data' would add the records to an ES index called 'some.example.com-processname-data'. Allows to use strftime format codes. Defaults to 'heka-%{%Y.%m.%d}'.
- **type_name (string):** Name of ES record type to create. Supports interpolation of message field values (from 'Type', 'Hostname', 'Pid', 'UUID', 'Logger', 'EnvVersion', 'Severity', field name, or a timestamp format) with the use of '%{}' chars, so '%{Hostname}-stat' would create an ES record with a type of 'some.example.com-stat'. Defaults to 'message'.
- **fields ([]string):** The 'fields' parameter specifies 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 "DynamicFields" (where "DynamicFields" causes the inclusion of dynamically specified message fields, see `dynamic_fields`). Defaults to including all of the supported message fields.
- **timestamp (string):** Format to use for timestamps in generated ES documents. Allows to use strftime format codes. Defaults to "%Y-%m-%dT%H:%M:%S".
- **es_index_from_timestamp (bool):** When generating the index name use the timestamp from the message instead of the current time. Defaults to false.
- **id (string):** Allows you to optionally specify the document id for ES to use. Useful for overwriting existing ES documents. If the value specified is placed within %{}, it will be interpolated to its Field value. Default is allow ES to auto-generate the id.
- **raw_bytes_fields ([]string):** This specifies a set of fields which will be passed through to the encoded JSON output without any processing or escaping. This is useful for fields which contain embedded JSON objects to prevent the embedded JSON from being escaped as normal strings. Only supports dynamically specified message fields.
- **field_mappings (map[string]string):** Maps Heka message fields to custom ES keys. Can be used to implement a custom format in ES or implement Logstash V1. The available fields are "Timestamp", "Uuid", "Type", "Logger", "Severity", "Payload", "EnvVersion", "Pid" and "Hostname".
- **dynamic_fields ([]string):** This specifies which of the message's dynamic fields should be included in the JSON output. Defaults to including all of the messages dynamic fields. If `dynamic_fields` is non-empty, then the `fields` list *must* contain "DynamicFields" or an error will be raised.

Example

```
[ESJsonEncoder]
index = "%{Type}-%{2006.01.02}"
es_index_from_timestamp = true
type_name = "%{Type}"
    [ESJsonEncoder.field_mappings]
    Timestamp = "@timestamp"
```

```
    Severity = "level"

[ElasticSearchOutput]
message_matcher = "Type == 'nginx.access'"
encoder = "ESJsonEncoder"
flush_interval = 50
```

ElasticSearch Logstash V0 Encoder

Plugin Name: **ESLogstashV0Encoder**

This encoder serializes a Heka message into a JSON format, preceded by a separate JSON structure containing information required for ElasticSearch BulkAPI indexing. The message JSON structure uses the original (i.e. "v0") schema popularized by Logstash. Using this schema can aid integration with existing Logstash deployments. This schema also plays nicely with the default Logstash dashboard provided by Kibana.

The JSON serialization is done by hand, without using Go's stdlib JSON marshalling. This is so serialization can succeed even if the message contains invalid UTF-8 characters, which will be encoded as U+FFFD. Config:

- **index (string):** Name of the ES index into which the messages will be inserted. Supports interpolation of message field values (from 'Type', 'Hostname', 'Pid', 'UUID', 'Logger', 'EnvVersion', 'Severity', a field name, or a timestamp format) with the use of '%{}' chars, so '%{Hostname}-%{Logger}-data' would add the records to an ES index called 'some.example.com-processname-data'. Defaults to 'logstash-%{2006.01.02}'.
- **type_name (string):** Name of ES record type to create. Supports interpolation of message field values (from 'Type', 'Hostname', 'Pid', 'UUID', 'Logger', 'EnvVersion', 'Severity', field name, or a timestamp format) with the use of '%{}' chars, so '%{Hostname}-stat' would create an ES record with a type of 'some.example.com-stat'. Defaults to 'message'.
- **use_message_type (bool):** If false, the generated JSON's @type value will match the ES record type specified in the type_name setting. If true, the message's Type value will be used as the @type value instead. Defaults to false.
- **fields ([]string):** The 'fields' parameter specifies 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 "DynamicFields" (where "DynamicFields" causes the inclusion of dynamically specified message fields, see `dynamic_fields`). Defaults to including all of the supported message fields. The "Payload" field is sent to ElasticSearch as "@message".
- **timestamp (string):** Format to use for timestamps in generated ES documents. Allows to use strftime format codes. Defaults to "%Y-%m-%dT%H:%M:%S".
- **es_index_from_timestamp (bool):** When generating the index name use the timestamp from the message instead of the current time. Defaults to false.
- **id (string):** Allows you to optionally specify the document id for ES to use. Useful for overwriting existing ES documents. If the value specified is placed within %{}, it will be interpolated to its Field value. Default is allow ES to auto-generate the id.
- **raw_bytes_fields ([]string):** This specifies a set of fields which will be passed through to the encoded JSON output without any processing or escaping. This is useful for fields which contain embedded JSON objects to prevent the embedded JSON from being escaped as normal strings. Only supports dynamically specified message fields.
- **dynamic_fields ([]string):** This specifies which of the message's dynamic fields should be included in the JSON output. Defaults to including all of the messages dynamic fields. If `dynamic_fields` is non-empty, then the `fields` list *must* contain "DynamicFields" or an error will be raised.

Example

```
[ESLogstashV0Encoder]
es_index_from_timestamp = true
type_name = "%{Type}"

[ElasticSearchOutput]
message_matcher = "Type == 'nginx.access'"
encoder = "ESLogstashV0Encoder"
flush_interval = 50
```

ElasticSearch Payload Encoder

Plugin Name: **SandboxEncoder**
File Name: **lua_encoders/es_payload.lua**

Prepends ElasticSearch BulkAPI index JSON to a message payload.

Config:

- **index (string, optional, default "heka-%{%Y.%m.%d}")** String to use as the *_index* key's value in the generated JSON. Supports field interpolation as described below.
- **type_name (string, optional, default "message")** String to use as the *_type* key's value in the generated JSON. Supports field interpolation as described below.
- **id (string, optional)** String to use as the *_id* key's value in the generated JSON. Supports field interpolation as described below.
- **es_index_from_timestamp (boolean, optional)** If true, then any time interpolation (often used to generate the ElasticSeach index) will use the timestamp from the processed message rather than the system time.

Field interpolation:

All of the string config settings listed above support *message field interpolation*.

Example Heka Configuration

```
[es_payload]
type = "SandboxEncoder"
filename = "lua_encoders/es_payload.lua"
    [es_payload.config]
    es_index_from_timestamp = true
    index = "%{Logger}-%{%Y.%m.%d}"
    type_name = "%{Type}-%{Hostname}"

[ElasticSearchOutput]
message_matcher = "Type == 'mytype'"
encoder = "es_payload"
```

Example Output

```
{"index":{"_index":"mylogger-2014.06.05","_type":"mytype-host.domain.com"}}
{"json":"data","extracted":"from","message":"payload"}
```

Payload Encoder

Plugin Name: **PayloadEncoder**

The PayloadEncoder simply extracts the payload from the provided Heka message and converts it into a byte stream for delivery to an external resource. Config:

- **append_newlines (bool, optional):** Specifies whether or not a newline character (i.e. *n*) will be appended to the captured message payload before serialization. Defaults to true.
- **prefix_ts (bool, optional):** Specifies whether a timestamp will be prepended to the captured message payload before serialization. Defaults to false.
- **ts_from_message (bool, optional):** If true, the prepended timestamp will be extracted from the message that is being processed. If false, the prepended timestamp will be generated by the system clock at the time of message processing. Defaults to true. This setting has no impact if *prefix_ts* is set to false.
- **ts_format (string, optional):** Specifies the format that should be used for prepended timestamps, using Go's standard time format specification strings. Defaults to *[2006/Jan/02:15:04:05 -0700]*. If the specified format string does not end with a space character, then a space will be inserted between the formatted timestamp and the payload.

Example

```
[PayloadEncoder]
append_newlines = false
prefix_ts = true
ts_format = "2006/01/02 3:04:05PM MST"
```

Protobuf Encoder

Plugin Name: **ProtobufEncoder**

The ProtobufEncoder is used to serialize Heka message objects back into Heka's standard protocol buffers format. This is the format that Heka uses to communicate with other Heka instances, so one will always be included in your Heka configuration using the default "ProtobufEncoder" name whether specified or not.

The hekad protocol buffers message schema is defined in the *message.proto* file in the *message* package.

Config:

<none>

Example:

```
[ProtobufEncoder]
```

See also:

Protocol Buffers - Google's data interchange format

Restructured Text Encoder

Plugin Name: **RstEncoder**

The RstEncoder generates a reStructuredText rendering of a Heka message, including all fields and attributes. It is useful for debugging, especially when coupled with a *Log Output*.

Config:

<none>

Example:

```
[RstEncoder]

[LogOutput]
message_matcher = "TRUE"
encoder = "RstEncoder"
```

Sandbox Encoder

Plugin Name: **SandboxEncoder**

The SandboxEncoder provides an isolated execution environment for converting messages into binary data without the need to recompile Heka. See *Sandbox*. Config:

- *Common Sandbox Parameters*

Example

```
[custom_json_encoder]
type = "SandboxEncoder"
filename = "path/to/custom_json_encoder.lua"

    [custom_json_encoder.config]
    msg_fields = ["field1", "field2"]
```

Schema InfluxDB Encoder

New in version 0.8.

Plugin Name: **SandboxEncoder**
File Name: **lua_encoders/schema_influx.lua**

Converts full Heka message contents to JSON for InfluxDB HTTP API. Includes all standard message fields and iterates through all of the dynamically specified fields, skipping any bytes fields or any fields explicitly omitted using the *skip_fields* config option.

Note: This encoder is intended for use with InfluxDB versions *prior* to 0.9. If you're working with InfluxDB v0.9 or greater, you'll want to use the *Schema InfluxDB Write Encoder* instead.

Config:

- **series (string, optional, default "series")** String to use as the *series* key's value in the generated JSON. Supports interpolation of field values from the processed message, using *%{fieldname}*. Any *fieldname* values of "Type", "Payload", "Hostname", "Pid", "Logger", "Severity", or "EnvVersion" will be extracted from the the base message schema, any other values will be assumed to refer to a dynamic message field. Only the first value of the first instance of a dynamic message field can be used for series name interpolation. If the dynamic field doesn't exist, the uninterpolated value will be left in the series name. Note that it is not possible to interpolate either the "Timestamp" or the "Uuid" message fields into the series name, those values will be interpreted as referring to dynamic message fields.
- **skip_fields (string, optional, default "")** Space delimited set of fields that should *not* be included in the InfluxDB records being generated. Any fieldname values of "Type", "Payload", "Hostname", "Pid", "Logger", "Severity", or "EnvVersion" will be assumed to refer to the corresponding field from the base message schema. Any other values will be assumed to refer to a dynamic message field.

- **multi_series (boolean, optional, default false)** Instead of submitting all fields to InfluxDB as attributes of a single series, submit a series for each field that sets a "value" attribute to the value of the field. This also sets the name attribute to the series value with the field name appended to it by a ".". This is the recommended by InfluxDB for v0.9 onwards as it is found to provide better performance when querying and aggregating across multiple series.
- **exclude_base_fields (boolean, optional, default false)** Don't send the base fields to InfluxDB. This saves storage space by not including base fields that are mostly redundant and unused data. If skip_fields includes base fields, this overrides it and will only be relevant for skipping dynamic fields.

Example Heka Configuration

```
[influxdb]
type = "SandboxEncoder"
filename = "lua_encoders/schema_influx.lua"
    [influxdb.config]
    series = "heka.%{Logger}"
    skip_fields = "Pid EnvVersion"

[InfluxOutput]
message_matcher = "Type == 'influxdb'"
encoder = "influxdb"
type = "HttpOutput"
address = "http://influxdbserver.example.com:8086/db/databasename/series"
username = "influx_username"
password = "influx_password"
```

Example Output

```
[{"points":[[1.409378221e+21,"log","test","systemName","TcpInput",5,1,"test"]],"name":"heka.MyLogger"
```

Schema InfluxDB Write Encoder

New in version 0.10.

Plugin Name: **SandboxEncoder**
File Name: **lua_encoders/schema_influx_write.lua**

StatMetric InfluxDB Encoder

New in version 0.7.

Plugin Name: **SandboxEncoder**
File Name: **lua_encoders/statmetric_influx.lua**

Extracts data from message fields in *heka.statmetric* messages generated by a *Stat Accumulator Input* and generates JSON suitable for use with InfluxDB's HTTP API. StatAccumInput must be configured with *emit_in_fields* = *true* for this encoder to work correctly.

Config:

<none>

Example Heka Configuration

```
[statmetric-influx-encoder]
type = "SandboxEncoder"
filename = "lua_encoders/statmetric_influx.lua"

[influx]
type = "HttpOutput"
message_matcher = "Type == 'heka.statmetric'"
address = "http://myinfluxserver.example.com:8086/db/stats/series"
encoder = "statmetric-influx-encoder"
username = "influx_username"
password = "influx_password"
```

Example Output

```
[{"points":[[1408404848,78271]],"name":"stats.counters.000000.rate","columns":["time","value"]},{"poi
```

2.9 Outputs

2.9.1 Common Output Parameters

There are some configuration options that are universally available to all Heka 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.
- **encoder (string, optional):** New in version 0.6.

  Encoder to be used by the output. This should refer to the name of an encoder plugin section that is specified elsewhere in the TOML configuration. Messages can be encoded using the specified encoder by calling the OutputRunner's *Encode()* method.
- **use_framing (bool, optional):** New in version 0.6.

  Specifies whether or not Heka's *Stream Framing* should be applied to the binary data returned from the OutputRunner's *Encode()* method.
- **can_exit (bool, optional)** New in version 0.7.

  Whether or not this plugin can exit without causing Heka to shutdown. Defaults to false.

2.9.2 Available Output Plugins

AMQP Output

Plugin Name: **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.

Config:

- **url (string):** An AMQP connection string formatted per the RabbitMQ URI Spec.
- **exchange (string):** AMQP exchange name
- **exchange_type (string):** AMQP exchange type (*fanout*, *direct*, *topic*, or *headers*).
- **exchange_durability (bool):** Whether the exchange should be configured as a durable exchange. Defaults to non-durable.
- **exchange_auto_delete (bool):** Whether the exchange is deleted when all queues have finished and there is no publishing. Defaults to auto-delete.
- **routing_key (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.
- **retries (RetryOptions, optional):** A sub-section that specifies the settings to be used for restart behavior. See *Configuring Restarting Behavior*

New in version 0.6.

- **content_type (string):** MIME content type of the payload used in the AMQP header. Defaults to "application/hekad".
- **encoder (string, optional)** Specifies which of the registered encoders should be used for converting Heka messages to binary data that is sent out over the AMQP connection. Defaults to the always available "ProtobufEncoder".
- **use_framing (bool, optional):** Specifies whether or not the encoded data sent out over the TCP connection should be delimited by Heka's *Stream Framing*. Defaults to true.

New in version 0.6.

- **tls (TlsConfig):** An optional sub-section that specifies the settings to be used for any SSL/TLS encryption. This will only have any impact if *URL* uses the *AMQPS* URI scheme. See *Configuring TLS*.

Example (that sends log lines from the logger):

```
[AMQPOutput]
url = "amqp://guest:guest@rabbitmq/"
exchange = "testout"
exchange_type = "fanout"
message_matcher = 'Logger == "TestWebserver"'
```

Carbon Output

Plugin Name: **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 or UDP socket using the plaintext protocol.

Config:

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

New in version 0.5.

- **protocol (string):** "tcp" or "udp" (default: "tcp")

- **tcp_keep_alive (bool)** if set, keep the TCP connection open and reuse it until a failure; then retry (default: false)

Example:

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

Dashboard Output

Plugin Name: **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.

Config:

- **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 *$(BASE_DIR)/dashboard*.
- **static_directory (string):** File system directory where the Heka dashboard source code can be found. The Heka process must have read access to this directory. Relative paths will be evaluated relative to the Heka base directory. Defaults to *${SHARE_DIR}/dasher*.

New in version 0.7.

- **headers (subsection, optional):** It is possible to inject arbitrary HTTP headers into each outgoing response by adding a TOML subsection entitled "headers" to you HttpOutput config section. All entries in the subsection must be a list of string values.

Example:

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

ElasticSearch Output

Plugin Name: **ElasticSearchOutput**

Output plugin that uses HTTP or UDP to insert records into an ElasticSearch database. Note that it is up to the specified encoder to both serialize the message into a JSON structure *and* to prepend that with the appropriate ElasticSearch BulkAPI indexing JSON. Usually this output is used in conjunction with an ElasticSearch-specific encoder plugin, such as *ElasticSearch JSON Encoder*, *ElasticSearch Logstash V0 Encoder*, or *ElasticSearch Payload Encoder*.

Config:

- **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.

- **server (string):** ElasticSearch server URL. Supports http://, https:// and udp:// urls. Defaults to "http://localhost:9200".
- **connect_timeout (int):** Time in milliseconds to wait for a server name resolving and connection to ES. It's included in an overall time (see 'http_timeout' option), if they both are set. Default is 0 (no timeout).
- **http_timeout (int):** Time in milliseconds to wait for a response for each http post to ES. This may drop data as there is currently no retry. Default is 0 (no timeout).
- **http_disable_keepalives (bool):** Specifies whether or not re-using of established TCP connections to ElasticSearch should be disabled. Defaults to false, that means using both HTTP keep-alive mode and TCP keep-alives. Set it to true to close each TCP connection after 'flushing' messages to ElasticSearch.
- **username (string):** The username to use for HTTP authentication against the ElasticSearch host. Defaults to "" (i. e. no authentication).
- **password (string):** The password to use for HTTP authentication against the ElasticSearch host. Defaults to "" (i. e. no authentication).

New in version 0.9.

- **tls (TlsConfig):** An optional sub-section that specifies the settings to be used for any SSL/TLS encryption. This will only have any impact if *URL* uses the *HTTPS* URI scheme. See *Configuring TLS*.
- **use_buffering (bool, optional):** Buffer records to a disk-backed buffer on the Heka server before writing them to ElasticSearch. Defaults to true.
- **buffering (QueueBufferConfig, optional):** All of the *buffering* config options are set to the standard default options.

Example:

```
[ElasticSearchOutput]
message_matcher = "Type == 'sync.log'"
server = "http://es-server:9200"
flush_interval = 5000
flush_count = 10
encoder = "ESJsonEncoder"
```

File Output

Plugin Name: **FileOutput**

Writes message data out to a file system.

Config:

- **path (string):** Full path to the output file. If date rotation is in use, then the output file path can support strftime syntax to embed timestamps in the file path: http://strftime.org
- **perm (string, optional):** File permission for writing. A string of the octal digit representation. Defaults to "644".
- **folder_perm (string, optional):** Permissions to apply to directories created for FileOutput's parent directory if it doesn't exist. Must be a string representation of an octal integer. Defaults to "700".
- **flush_interval (uint32, optional):** Interval at which accumulated file data should be written to disk, in milliseconds (default 1000, i.e. 1 second). Set to 0 to disable.
- **flush_count (uint32, optional):** Number of messages to accumulate until file data should be written to disk (default 1, minimum 1).

- **flush_operator (string, optional):** Operator describing how the two parameters "flush_interval" and "flush_count" are combined. Allowed values are "AND" or "OR" (default is "AND").

New in version 0.6.

- **use_framing (bool, optional):** Specifies whether or not the encoded data sent out over the TCP connection should be delimited by Heka's *Stream Framing*. Defaults to true if a ProtobufEncoder is used, false otherwise.

New in version 0.9.

- **rotation_interval (uint32, optional):** Interval at which the output file should be rotated, in hours. Only the following values are allowed: 0, 1, 4, 12, 24 (set to 0 to disable). The files will be named relative to midnight of the day. Defaults to 0, i.e. disabled.

Example:

```
[counter_file]
type = "FileOutput"
message_matcher = "Type == 'heka.counter-output'"
path = "/var/log/heka/counter-output.log"
perm = "666"
flush_count = 100
flush_operator = "OR"
encoder = "PayloadEncoder"
```

New in version 0.6.

HTTP Output

Plugin Name: **HttpOutput**

A very simple output plugin that uses HTTP GET, POST, or PUT requests to deliver data to an HTTP endpoint. When using POST or PUT request methods the encoded output will be uploaded as the request body. When using GET the encoded output will be ignored.

This output doesn't support any request batching; each received message will generate an HTTP request. Batching can be achieved by use of a filter plugin that accumulates message data, periodically emitting a single message containing the batched, encoded HTTP request data in the payload. An HttpOutput can then be configured to capture these batch messages, using a *Payload Encoder* to extract the message payload.

For now the HttpOutput only supports statically defined request parameters (URL, headers, auth, etc.). Future iterations will provide a mechanism for dynamically specifying these values on a per-message basis.

Config:

- **address (string):** URL address of HTTP server to which requests should be sent. Must begin with "http://" or "https://".
- **method (string, optional):** HTTP request method to use, must be one of GET, POST, or PUT. Defaults to POST.
- **username (string, optional):** If specified, HTTP Basic Auth will be used with the provided user name.
- **password (string, optional):** If specified, HTTP Basic Auth will be used with the provided password.
- **headers (subsection, optional):** It is possible to inject arbitrary HTTP headers into each outgoing request by adding a TOML subsection entitled "headers" to you HttpOutput config section. All entries in the subsection must be a list of string values.
- **http_timeout(uint, optional):** Time in milliseconds to wait for a response for each http request. This may drop data as there is currently no retry. Default is 0 (no timeout)

- **tls (subsection, optional):** A sub-section that specifies the settings to be used for any SSL/TLS encryption. This will only have any impact if an "https://" address is used. See *Configuring TLS*.

Example:

```
[PayloadEncoder]

[influxdb]
type = "HttpOutput"
message_matcher = "Type == 'influx.formatted'"
address = "http://influxdb.example.com:8086/db/stats/series"
encoder = "PayloadEncoder"
username = "MyUserName"
password = "MyPassword"
```

IRC Output

Plugin Name: **IrcOutput**

Connects to an Irc Server and sends messages to the specified Irc channels. Output is encoded using the specified encoder, and expects output to be properly truncated to fit within the bounds of an Irc message before being receiving the output.

Config:

- **server (string):** A host:port of the irc server that Heka will connect to for sending output.
- **nick (string):** Irc nick used by Heka.
- **ident (string):** The Irc identity used to login with by Heka.
- **password (string, optional):** The password used to connect to the Irc server.
- **channels (list of strings):** A list of Irc channels which every matching Heka message is sent to. If there is a space in the channel string, then the part after the space is expected to be a password for a protected irc channel.
- **timeout (uint, optional):** The maximum amount of time (in seconds) to wait before timing out when connect, reading, or writing to the Irc server. Defaults to 10.
- **tls (TlsConfig, optional):** A sub-section that specifies the settings to be used for any SSL/TLS encryption. This will only have any impact if *use_tls* is set to true. See *Configuring TLS*.
- **queue_size (uint, optional):** This is the maximum amount of messages Heka will queue per Irc channel before discarding messages. There is also a queue of the same size used if all per-irc channel queues are full. This is used when Heka is unable to send a message to an Irc channel, such as when it hasn't joined or has been disconnected. Defaults to 100.
- **rejoin_on_kick (bool, optional):** Set this if you want Heka to automatically re-join an Irc channel after being kicked. If not set, and Heka is kicked, it will not attempt to rejoin ever. Defaults to false.
- **ticker_interval (uint, optional):** How often (in seconds) heka should send a message to the server. This is on a per message basis, not per channel. Defaults to 2.
- **time_before_reconnect (uint, optional):** How long to wait (in seconds) before reconnecting to the Irc server after being disconnected. Defaults to 3.
- **time_before_rejoin (uint, optional):** How long to wait (in seconds) before attempting to rejoin an Irc channel which is full. Defaults to 3.

- **max_join_retries (uint, optional):** The maximum amount of attempts Heka will attempt to join an Irc channel before giving up. After attempts are exhausted, Heka will no longer attempt to join the channel. Defaults to 3.
- **verbose_irc_logging (bool, optional):** Enable to see raw internal message events Heka is receiving from the server. Defaults to false.
- **encoder (string):** Specifies which of the registered encoders should be used for converting Heka messages into what is sent to the irc channels.
- **retries (RetryOptions, optional):** A sub-section that specifies the settings to be used for restart behavior. See *Configuring Restarting Behavior*

Example:

```
[IrcOutput]
message_matcher = 'Type == "alert"'
encoder = "PayloadEncoder"
server = "irc.mozilla.org:6667"
nick = "heka_bot"
ident = "heka_ident"
channels = [ "#heka_bot_irc testkeypassword" ]
rejoin_on_kick = true
queue_size = 200
ticker_interval = 1
```

Kafka Output

Plugin Name: **KafkaOutput**

Connects to a Kafka broker and sends messages to the specified topic.

Config:

- **id (string)** Client ID string. Default is the hostname.
- **addrs ([]string)** List of brokers addresses.
- **metadata_retries (int)** How many times to retry a metadata request when a partition is in the middle of leader election. Default is 3.
- **wait_for_election (uint32)** How long to wait for leader election to finish between retries (in milliseconds). Default is 250.
- **background_refresh_frequency (uint32)** How frequently the client will refresh the cluster metadata in the background (in milliseconds). Default is 600000 (10 minutes). Set to 0 to disable.
- **max_open_reqests (int)** How many outstanding requests the broker is allowed to have before blocking attempts to send. Default is 4.
- **dial_timeout (uint32)** How long to wait for the initial connection to succeed before timing out and returning an error (in milliseconds). Default is 60000 (1 minute).
- **read_timeout (uint32)** How long to wait for a response before timing out and returning an error (in milliseconds). Default is 60000 (1 minute).
- **write_timeout (uint32)** How long to wait for a transmit to succeed before timing out and returning an error (in milliseconds). Default is 60000 (1 minute).
- **partitioner (string)** Chooses the partition to send messages to. The valid values are *Random*, *RoundRobin*, *Hash*. Default is Random.

- **hash_variable (string)** The message variable used for the Hash partitioner only. The variables are restricted to *Type*, *Logger*, *Hostname*, *Payload* or any of the message's dynamic field values. All dynamic field values will be converted to a string representation. Field specifications are the same as with the *Message Matcher Syntax* e.g. Fields[foo][0][0].
- **topic_variable (string)** The message variable used as the Kafka topic (cannot be used in conjunction with the 'topic' configuration). The variable restrictions are the same as the hash_variable.
- **topic (string)** A static Kafka topic (cannot be used in conjunction with the 'topic_variable' configuration).
- **required_acks (string)** The level of acknowledgement reliability needed from the broker. The valid values are *NoResponse*, *WaitForLocal*, *WaitForAll*. Default is WaitForLocal.
- **timeout (uint32)** The maximum duration the broker will wait for the receipt of the number of RequiredAcks (in milliseconds). This is only relevant when RequiredAcks is set to WaitForAll. Default is no timeout.
- **compression_codec (string)** The type of compression to use on messages. The valid values are *None*, *GZIP*, *Snappy*. Default is None.
- **max_buffer_time (uint32)** The maximum duration to buffer messages before triggering a flush to the broker (in milliseconds). Default is 1.
- **max_buffered_bytes (uint32)** The threshold number of bytes buffered before triggering a flush to the broker. Default is 1.
- **back_pressure_threshold_bytes (uint32)** The maximum number of bytes allowed to accumulate in the buffer before back-pressure is applied to QueueMessage. Without this, queueing messages too fast will cause the producer to construct requests larger than the MaxRequestSize (100 MiB). Default is 50 * 1024 * 1024 (50 MiB), cannot be more than (MaxRequestSize - 10 KiB).

Example (send various Fxa messages to a static Fxa topic):

```
[FxaKafkaOutput]
type = "KafkaOutput"
message_matcher = "Logger == 'FxaAuthWebserver' || Logger == 'FxaAuthServer'"
topic = "Fxa"
addrs = ["localhost:9092"]
encoder = "ProtobufEncoder"
```

Log Output

Plugin Name: **LogOutput**

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

Config:

<none>

Example:

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

Nagios Output

Plugin Name: **NagiosOutput**

Specialized output plugin that listens for Nagios external command message types and delivers passive service check results to Nagios using either HTTP requests made to the Nagios cmd.cgi API or the use of the *send_ncsa* binary. The message payload must consist of a state followed by a colon and then the message e.g., "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.

Config:

- **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 empty string.
- **password (string, optional):** Password used to authenticate with the Nagios web interface. Defaults to empty string.
- **response_header_timeout (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.
- **nagios_service_description (string, optional):** Must match Nagios service's service_description attribute. Defaults to the name of the output.
- **nagios_host (string, optional):** Must match the hostname of the server in nagios. Defaults to the Hostname attribute of the message.
- **send_nsca_bin (string, optional):** New in version 0.5.

  Use send_nsca program, as provided, rather than sending HTTP requests. Not supplying this value means HTTP will be used, and any other send_nsca_* settings will be ignored.
- **send_nsca_args ([]string, optional):** New in version 0.5.

  Arguments to use with send_nsca, usually at least the nagios hostname, e.g. *["-H", "nagios.somehost.com"]*. Defaults to an empty list.
- **send_nsca_timeout (int, optional):** New in version 0.5.

  Timeout for the send_nsca command, in seconds. Defaults to 5.
- **use_tls (bool, optional):** New in version 0.5.

  Specifies whether or not SSL/TLS encryption should be used for the TCP connections. Defaults to false.
- **tls (TlsConfig, optional):** New in version 0.5.

  A sub-section that specifies the settings to be used for any SSL/TLS encryption. This will only have any impact if *use_tls* is set to true. See *Configuring TLS*.

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:

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

Sandbox Output

New in version 0.9.

Plugin Name: **SandboxOutput**

The SandboxOutput provides a flexible execution environment for data encoding and transmission without the need to recompile Heka. See *Sandbox*. Config:

- The common output configuration parameter 'encoder' is ignored since all data transformation should happen in the plugin.
- *Common Sandbox Parameters*
- **timer_event_on_shutdown (bool):** True if the sandbox should have its timer_event function called on shutdown.

Example

```
[SandboxFileOutput]
type = "SandboxOutput"
filename = "fileoutput.lua"

[SandboxFileOutput.config]
path = "mylog.txt"
```

SMTP Output

Plugin Name: **SmtpOutput**

New in version 0.5.

Outputs a Heka message in an email. The message subject is the plugin name and the message content is controlled by the payload_only setting. The primary purpose is for email alert notifications e.g., PagerDuty.

Config:

- **send_from (string)** The email address of the sender. (default: "heka@localhost.localdomain")
- **send_to (array of strings)** An array of email addresses where the output will be sent to.
- **subject (string)** Custom subject line of email. (default: "Heka [SmtpOutput]")
- **host (string)** SMTP host to send the email to (default: "127.0.0.1:25")
- **auth (string)** SMTP authentication type: "none", "Plain", "CRAMMD5" (default: "none")
- **user (string, optional)** SMTP user name
- **password (string, optional)** SMTP user password

New in version 0.9.

- **send_interval (uint, optional)** Minimum time interval between each email, in seconds. First email in an interval goes out immediately, subsequent messages in the same interval are concatenated and all sent when the interval expires. Defaults to 0, meaning all emails are sent immediately.

Example:

```
[FxaAlert]
type = "SmtpOutput"
message_matcher = "((Type == 'heka.sandbox-output' && Fields[payload_type] == 'alert') || Type == 'he
send_from = "heka@example.com"
send_to = ["alert@example.com"]
```

```
auth = "Plain"
user = "test"
password = "testpw"
host = "localhost:25"
encoder = "AlertEncoder"
```

TCP Output

Plugin Name: **TcpOutput**

Output plugin that delivers Heka message data 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, or to any other arbitrary listening TCP server that knows how to process the encoded data.

Config:

- **address (string):** An IP address:port to which we will send our output data.
- **use_tls (bool, optional):** Specifies whether or not SSL/TLS encryption should be used for the TCP connections. Defaults to false.

New in version 0.5.

- **tls (TlsConfig, optional):** A sub-section that specifies the settings to be used for any SSL/TLS encryption. This will only have any impact if `use_tls` is set to true. See *Configuring TLS*.

New in version 0.6.

- **local_address (string, optional):** A local IP address to use as the source address for outgoing traffic to this destination. Cannot currently be combined with TLS connections.
- **encoder (string, optional):** Specifies which of the registered encoders should be used for converting Heka messages to binary data that is sent out over the TCP connection. Defaults to the always available "ProtobufEncoder".
- **use_framing (bool, optional):** Specifies whether or not the encoded data sent out over the TCP connection should be delimited by Heka's *Stream Framing*. Defaults to true if a ProtobufEncoder is used, false otherwise.
- **keep_alive (bool):** Specifies whether or not TCP keepalive should be used for established TCP connections. Defaults to false.
- **keep_alive_period (int):** Time duration in seconds that a TCP connection will be maintained before keepalive probes start being sent. Defaults to 7200 (i.e. 2 hours).

New in version 0.10.

- **use_buffering (bool, optional):** Buffer records to a disk-backed buffer on the Heka server before sending them out over the TCP connection. Defaults to true.
- **buffering (QueueBufferConfig, optional):** All of the *buffering* config options are set to the standard default options, except for *cursor_update_count*, which is set to 50 instead of the standard default of 1.

Example:

```
[aggregator_output]
type = "TcpOutput"
address = "heka-aggregator.mydomain.com:55"
local_address = "127.0.0.1"
message_matcher = "Type != 'logfile' && Type !~ /^heka\./'"
```

UDP Output

New in version 0.7.

Plugin Name: **UdpOutput**

Output plugin that delivers Heka message data to a specified UDP or Unix datagram socket location.

Config:

- **net (string, optional):** Network type to use for communication. Must be one of "udp", "udp4", "udp6", or "unixgram". "unixgram" option only available on systems that support Unix datagram sockets. Defaults to "udp".
- **address (string):** Address to which we will be sending the data. Must be IP:port for net types of "udp", "udp4", or "udp6". Must be a path to a Unix datagram socket file for net type "unixgram".
- **local_address (string, optional):** Local address to use on the datagram packets being generated. Must be IP:port for net types of "udp", "udp4", or "udp6". Must be a path to a Unix datagram socket file for net type "unixgram".
- **encoder (string):** Name of registered encoder plugin that will extract and/or serialized data from the Heka message.
- **max_message_size (int):** Maximum size of message that is allowed to be sent via UdpOutput. Messages which exceeds this limit will be dropped. Defaults to 65507 (the limit for UDP packets in IPv4).

Example:

```
[PayloadEncoder]

[UdpOutput]
address = "myserver.example.com:34567"
encoder = "PayloadEncoder"
```

Whisper Output

Plugin Name: **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.

Config:

- **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 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 *Dashboard Output*.

2.11 Extending Heka

The core of the Heka engine is written in the Go programming language. Heka supports six different types of plugins (inputs, splitters, decoders, filters, encoders, 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 security sandboxed Lua code for implementing the core logic of every plugin type except splitters. This document only covers the development of Go plugins. You can learn more about sandboxed plugins in the *Sandbox* section.

2.11.1 Definitions

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

2.11.2 Overview

Each Heka plugin type performs a specific task. Inputs receive input from the outside world and inject the data into the Heka pipeline. Splitters slice the input stream into individual records. Decoders turn binary data into Message objects that Heka can process. Filters perform arbitrary processing (aggregation, collation, monitoring, etc.) of Heka message data. Encoders serialize Heka messages into arbitrary byte streams. 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 LogstreamerInput (reading files 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 all Heka plugins except encoders actually consist of two parts: the plugin itself, and an accompanying "plugin runner". Inputs have an InputRunner, splitters have a SplitterRunner, 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 first creates and configures a plugin instance of the appropriate type, then it creates a plugin runner instance of the appropriate type, passing in the plugin.

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 splitters, decoders and encoders, however. Configuration sections for splitter, decoder and encoder plugins register *possible* configurations, but actual running instances of these types aren't created until they are used by input or output plugins.

2.11.3 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 plugin 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. The input, filter, and output plugins will then be started so they can begin processing messages. The splitter, decoder, and encoder instances will be thrown away, with new ones created as needed when requested by input (for splitter and decoder) or output (for encoder) plugins.

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 contrived example. In practice, messages are routed to outputs using the *Message Matcher Syntax*.)

2.11.4 Restarting Plugins

If your plugin supports being restarted and either fails to initialize properly at startup, or fails during Run with an error (perhaps because a network connection dropped, a file became unavailable, etc) then Heka will attempt to reinitialize and restart it up until the specified max_retries value.

If the failure continues beyond the maximum number of retries, or if the plugin didn't support restarting in the first place, then Heka will either shut down or, if the plugin is an input, filter or an output with the `can_exit` setting set to true, the plugin will be removed from operation and Heka will continue to run.

To add restart support to your plugin, you must implement the `Restarting` interface defined in the config.go file:

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

The `CleanupForRestart` method will be called when the plugin's main run method exits, a single time. This allows you a place to perform any additional cleanup that might be necessary before attempting to reinitialize the plugin. After this, the runner will repeatedly call the plugin's 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.

2.11.5 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 look at the code for Heka's UdpOutput, which delivers messages to a UDP listener somewhere. The initialization code looks as follows:

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

// This is our plugin's config struct
type UdpOutputConfig struct {
    // Network type ("udp", "udp4", "udp6", or "unixgram"). Needs to match the
    // input type.
    Net string
    // String representation of the address of the network connection to which
    // we will be sending out packets (e.g. "192.168.64.48:3336").
    Address string
    // Optional address to use as the local address for the connection.
    LocalAddress string `toml:"local_address"`

    // Maximum size of message, plugin drops the data if it exceeds this limit.
    MaxMessageSize int `toml:"max_message_size"`
}

// Provides pipeline.HasConfigStruct interface.
func (o *UdpOutput) ConfigStruct() interface{} {
    return &UdpOutputConfig{
        Net: "udp",

        // Defines maximum size of udp data for IPv4.
        MaxMessageSize: 65507,
    }
}

// Initialize UDP connection
func (o *UdpOutput) Init(config interface{}) (err error) {
    o.UdpOutputConfig = config.(*UdpOutputConfig) // assert we have the right config type

    if o.UdpOutputConfig.MaxMessageSize < 512 {
                return fmt.Errorf("Maximum message size can't be smaller than 512 bytes.")
```

```
    }

    if o.Net == "unixgram" {
        if runtime.GOOS == "windows" {
            return errors.New("Can't use Unix datagram sockets on Windows.")
        }
        var unixAddr, lAddr *net.UnixAddr
        unixAddr, err = net.ResolveUnixAddr(o.Net, o.Address)
        if err != nil {
            return fmt.Errorf("Error resolving unixgram address '%s': %s", o.Address,
                err.Error())
        }
        if o.LocalAddress != "" {
            lAddr, err = net.ResolveUnixAddr(o.Net, o.LocalAddress)
            if err != nil {
                return fmt.Errorf("Error resolving local unixgram address '%s': %s",
                    o.LocalAddress, err.Error())
            }
        }
        if o.conn, err = net.DialUnix(o.Net, lAddr, unixAddr); err != nil {
            return fmt.Errorf("Can't connect to '%s': %s", o.Address,
                err.Error())
        }
    } else {
        var udpAddr, lAddr *net.UDPAddr
        if udpAddr, err = net.ResolveUDPAddr(o.Net, o.Address); err != nil {
            return fmt.Errorf("Error resolving UDP address '%s': %s", o.Address,
                err.Error())
        }
        if o.LocalAddress != "" {
            lAddr, err = net.ResolveUDPAddr(o.Net, o.LocalAddress)
            if err != nil {
                return fmt.Errorf("Error resolving local UDP address '%s': %s",
                    o.Address, err.Error())
            }
        }
        if o.conn, err = net.DialUDP(o.Net, lAddr, udpAddr); err != nil {
            return fmt.Errorf("Can't connect to '%s': %s", o.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 any common configuration options that are processed by Heka, such as the `synchronous_decode` option available to Input plugins, or the `ticker_interval`, `message_matcher`, and `buffering` values that are available to 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 plugin access to its configured name before the runner has started. The SandboxFilter plugin uses the name to locate/load any preserved state before being run:

```
type WantsName interface {
    SetName(name string)
```

```
}
```

There is also a similar `WantsPipelineConfig` interface that can be used if a plugin needs access to the active PipelineConfig or GlobalConfigStruct values in the ConfigStruct or Init methods:

```
type WantsPipelineConfig interface {
    SetPipelineConfig(pConfig *pipeline.PipelineConfig)
}
```

Note that, in the case of inputs, filters, and outputs, these interfaces only need to be implemented if you need this information *before* the plugin is started. Once started, the plugin runner and a plugin helper will be passed in to the Run or Prepare method, which make the plugin name and PipelineConfig struct available in other ways.

2.11.6 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 typically has three primary responsibilities:

1. Access some data or data stream from the outside world.
2. Provide acquired data or stream to a SplitterRunner for record extraction and further delivery.
3. (optional) Provide a "pack decorator" function to the SplitterRunner to populate the message object with any input-specific information.

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, you need to get a SplitterRunner to which you can feed your incoming data. This is available through the InputRunner's `NewSplitterRunner` method. NewSplitterRunner takes a single string argument called *token*. This token is used to differentiate multiple SplitterRunner instances from each other. If you have a simple input plugin that only needs a single SplitterRunner, you can just pass an empty string (i.e. `sr := ir.NewSplitterRunner("")`). In more complicated scenarios you might want multiple SplitterRunners, say one per goroutine, in which case you should pass a unique identifier string in to each NewSplitterRunner call.

Splitting records efficiently is a surprisingly complicated process so the SplitterRunner interface has a number of methods:

```
type SplitterRunner interface {
    PluginRunner
    SetInputRunner(ir InputRunner)
    Splitter() Splitter
    SplitBytes(data []byte, del Deliverer) error
    SplitStream(r io.Reader, del Deliverer) error
    GetRemainingData() (record []byte)
```

```
    GetRecordFromStream(r io.Reader) (int, []byte, error)
    DeliverRecord(record []byte, del Deliverer)
    KeepTruncated() bool
    UseMsgBytes() bool
    SetPackDecorator(decorator func(*PipelinePack))
    Done()
}
```

Don't let this scare you, however. SplitterRunner's expose some internal workings to be able to support advanced uses, but in most cases you only need to deal with a few of the exposed methods. Specifically, you care about either `SplitStream` or `SplitBytes`, possibly about `SetPackDecorator` and `UseMsgBytes`, and you need to call `Done` when the Splitter is no longer needed.

First we'll examine the "Split" methods. As mentioned above, you'll typically only want to use one or the other. Deciding which you want is straightforward. If your mechanism for getting data from the outside world is a stream object (an *io.Reader*, in Go terms), then you'll want SplitStream. If not and you just end up with a byte slice of binary data, then you'll want SplitBytes.

Note that both SplitStream and SplitBytes ask for a `Deliverer` object as their second argument. Again, in simple cases you don't need to worry about this. If you're only using a single SplitterRunner, you can just pass in nil and Heka will take care of delivering the message to a decoder and/or the message router appropriately. If you're using multiple goroutines (and therefore multiple SplitterRunners), however, you'll typically want multiple Deliverers, too. This is especially important if you want each separate goroutine to have its own Decoder, so decoding can happen in parallel, delegated to multiple cores on a single machine.

Like SplitterRunners, Deliverers are obtained from the InputRunner, using the `NewDeliverer` method. And, like SplitterRunners, NewDeliverer takes a single string identifier argument, which should be unique for each requested deliverer. Usually a single SplitterRunner will be using a single Deliverer, and the same token identifier will be used for each. You can see an example of this in the TcpInput's *handleConnection* code snippet a bit further down this page. It's also important to keep in mind that both SplitterRunners and Deliverers expose a `Done` method that should be called by the input plugin's code whenever the resource is no longer being used so Heka can free it up appropriately.

If you're using SplitBytes, then you'll want to call it each time you have a new payload of data to process. It will return the number of bytes successfully consumed from the provided slice, and any relevant errors occurred while processing. It is up to the calling code to decide what to do in error cases, or when all of the data isn't consumed.

If you're using SplitStream, then the SplitStream call will block for as long as it is consuming data. When data processing pauses or stops, SplitStream will exit and return control back to the input, returning either nil or any relevant errors. Typically if nil is returned, you'll want to call SplitStream again to continue processing the stream. Code such as the following is a common idiom:

```
var err error
for err == nil {
    err = sr.SplitStream(r, nil)
}
```

Any errors encountered while processing the stream, including io.EOF, will be returned from the SplitStream call. It is up to the input code to decide how to proceed.

Finally, we're ready for the third step, providing a "pack decorator" function to the SplitterRunner. Sometimes an input plugin would like to populate a Heka message with information specific to the input mechanism. The TcpInput, for instance, often wants to store the remote address of the TCP connection as a message's Hostname field. Any provided pack decorator function will be called immediately before the PipelinePack is passed on for delivery, allowing the input to mutate the pack's Message struct as desired. The TcpInput code that uses this feature looks like so:

```
func (t *TcpInput) handleConnection(conn net.Conn) {
    raddr := conn.RemoteAddr().String()
    host, _, err := net.SplitHostPort(raddr)
    if err != nil {
```

```
        host = raddr
    }

    deliverer := t.ir.NewDeliverer(host)
    sr := t.ir.NewSplitterRunner(host)

    defer func() {
        conn.Close()
        t.wg.Done()
        deliverer.Done()
        sr.Done()
    }()

    if !sr.UseMsgBytes() {
        packDec := func(pack *PipelinePack) {
            pack.Message.SetHostname(raddr)
        }
        sr.SetPackDecorator(packDec)
    }
```

The `if !sr.UseMsgBytes()` check before the SetPackDecorator call deserves some explanation. Generally Heka receives input data in one of two flavors. The first is standalone data, usually text, such as log files loaded from the file system using a LogstreamerInput. This data is stored within a Message struct, usually as the payload. Most decoder plugins, then, will expect to find the raw input data in the Message payload, and will parse this data and mutate the Message struct with extracted data.

The second flavor of input data is a binary blob, usually protocol buffers encoded, representing an entire Heka message. Clearly it doesn't make much sense to store data representing a serialized Message struct *within* a Message struct, since it would overwrite itself upon deserialization. For this reason, PipelinePacks have a MsgBytes attribute that is used as a buffer for storing binary data that will be converted to a message. Certain decoder plugins, most notably the ProtobufDecoder, will expect to find input data in the pack.MsgBytes buffer, and will use this to create a new Message struct from scratch.

Splitters can specify via a config setting whether the data records they parse should be placed in the message payload of an existing Message struct or in the MsgBytes attribute of the enclosing PipelinePack, depending on what the accompanying decoder plugin expects. The UseMsgBytes method on the SplitterRunner will return true if the contained splitter plugin is putting the data in the MsgBytes buffer, or false if it is putting the data in the Message's Payload field.

Now we can understand why the TcpInput is checking this before setting the pack decorator. When UseMsgBytes returns true, then the Message struct on that pack is going to be overwritten when decoding happens. There's not much value in setting the Hostname field when it's going to be clobbered shortly afterward.

Okay, that covers most of what you need to know about developing your own Heka input plugins. There's one important final possibility to consider, however. In some cases, an input might fail to retrieve any data at all, so it has nothing to hand to the Splitter. Even so, it might *still* want to deliver a message containing information about the data retrieval failure itself. The HttpInput does this when an HTTP request fails completely due to network or other errors, for instance.

When this happens the input must obtain a fresh PipelinePack, manually populate the contained Message struct, and manually hand it over for delivery. Here's the snippet in the HttpInput code that does this:

```
resp, err := httpClient.Do(req)
responseTime := time.Since(responseTimeStart)
if err != nil {
    pack := <-hi.ir.InChan()
    pack.Message.SetUuid(uuid.NewRandom())
    pack.Message.SetTimestamp(time.Now().UnixNano())
    pack.Message.SetType("heka.httpinput.error")
    pack.Message.SetPayload(err.Error())
```

```
    pack.Message.SetSeverity(hi.conf.ErrorSeverity)
    pack.Message.SetLogger(url)
    hi.ir.Deliver(pack)
    return
}
```

As you can see, the pattern is simple. The PipelinePack supply is exposed via a channel provided by the InputRunner's `InChan` method, so we pull from this channel to get a fresh pack. Then we populate the Message struct with any relevant data we want to include, and we finish up by passing the pack in to the InputRunner's `Deliver` method for delivery. If we were using separate Deliverers, then we would call the Deliver method on the relevant Deliverer instance instead of on the InputRunner.

One important detail about this pattern, however: if for any reason your plugin should pull a PipelinePack off of the input channel and *not* end up passing it on to one of the Deliver methods, you *must* call `pack.Recycle(nil)` to free the pack up to be used again. Failure to do so will eventually deplete the pool of PipelinePacks and will cause Heka to freeze.

2.11.7 Splitters

In contrast to the relatively complicated SplitterRunner interface that is discussed in the *Inputs* section above, the actual Splitter plugins themselves are very simple. The basic Splitter interface consists of a single method:

```
// Splitter plugin interface type.
type Splitter interface {
    FindRecord(buf []byte) (bytesRead int, record []byte)
}
```

The job of the `FindRecord` method is straightforward. It should scan through the provided byte slice, from the beginning, looking for any delimiters or appropriate indicators of a record boundary. It returns two values, the number of bytes consumed from the input buffer, and a slice that represents any record that was found. The `bytesRead` value should always be returned, whether a record slice is returned or not. If the entire buffer was scanned but no record was found, for instance, then bytesRead should be `len(buf)`.

Note that when a record is discovered, the returned slice can (and should, if possible) be a subsection of the input buffer. It's recommended that FindRecord not do any unnecessary copying of the input data.

In many cases this is all that is required of a splitter plugin. In some situations, however, records may include some headers and/or framing of some sort, and additional processing of those headers might be called for. For instance, Heka's native *Stream Framing* can embed HMAC authenticated message signing information in the message header, and the splitter needs to be able to decide whether or not the authentication is valid. For this reason, splitter plugins can implement an additional `UnframingSplitter` interface:

```
// UnframingSplitter is an interface optionally implemented by splitter
// plugins to remove and process any record framing that may have been used by
// the splitter.
type UnframingSplitter interface {
    UnframeRecord(framed []byte, pack *PipelinePack) []byte
}
```

The FindRecord method of an UnframingSplitter should return the full record, frame and all. Heka will then pass each framed record into the `UnframeRecord` method, along with the PipelinePack into which the record will be written. UnframeRecord should then extract the record framing, process it as needed, and return a byte slice containing the unframed record that is remaining. As with FindRecord, copying the data isn't necessary, the unframed record can safely refer to a subslice of the original framed record.

If the splitter examines the headers and decides that a given record is for some reason not valid, such as for the use of an incorrect authentication key, then it should return nil instead of the contained record. Additionally, signing information

can be written to the PipelinePack's `Signer` attribute, and this will be honored by the `message_signer` config setting available to *filter* and *output* plugins.

Note that if UnframeRecord returns nil it does *not* need to call `pack.Recycle(nil)`. Heka will recognize that the pack isn't going to be used and will recycle it itself.

2.11.8 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 and splitters, the `Decoder` interface is quite simple:

```
type Decoder interface {
    Decode(pack *PipelinePack) (packs []*PipelinePack, err error)
}
```

There are three additional optional interfaces a decoder might decide to implement. 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()
}
```

Understanding the third optional interface requires a bit of context. Heka's PipelinePack structs contain a Message attribute, which points to the actual instantiated Message struct, and a MsgBytes attribute, which is generally used to hold the protocol buffer encoding of the Message struct. Whenever a message is injected into the message router, Heka will protobuf encode that message and store the result in the MsgBytes attribute, also setting the pack's `TrustMsgBytes` attribute flag to `true`.

In some cases, however, a protobuf encoding of the message is already available. For instance, when a message is received in protobuf format and is not further mutated, as in the case when an input is using a single ProtobufDecoder, then the original incoming data is already a valid protobuf encoding. Any decoder that might already have access to or generate a valid protobuf encoding for the resulting message should implement the `EncodesMsgBytes` interface:

```
type EncodesMsgBytes interface {
    EncodesMsgBytes() bool
}
```

Heka will check for this method at startup and, if it exists, it will assume that the decoder plugin may populate the MsgBytes data with the encoded message data, and that it will set pack.TrustMsgBytes to true if it does.

A decoder's `Decode` method should extract raw message data from the provided pack. Depending on the nature of the decoder, this might be found either in the MsgBytes attribute of the PipelinePack, or in the contained Message struct's Payload field. Then it should try to deserialize and/or parse this raw data, using the contained information to overwrite or populate the pack's Message struct.

If the decoding / parsing operation concludes successfully then Decode should return a slice of PipelinePack pointers and a nil error value. The first item in the returned slice (i.e. `packs[0]`) should be the pack that was passed in to the method. If the decoding process needs to produce more than one output pack, additional ones can be obtained from the DecoderRunner's `NewPack` method, and they should be appended to the returned slice of packs.

If decoding fails for any reason, then Decode should return a nil value for the PipelinePack slice and an appropriate error value. Returning an error will cause Heka to log an error message about the decoding failure. Additionally, if the

associated input plugin's configuration set the `send_decode_failure` value to true, the message will be tagged with `decode_failure` and `decode_error` fields and delivered to the router.

2.11.9 About Message Mutation

All of the above plugin types (i.e. inputs, splitters, and decoders) come *before* the router in Heka's pipeline, and therefore they may safely mutate the message struct. Once a pack hits the router, however, it is no longer safe to mutate the message, because a) it might be concurrently processed by more that one filter and/or output plugin, leading to race conditions; and b) a protobuf encoding of the message will be stored in the pack.MsgBytes attribute, and mutating the message will cause this encoding to become out of sync with the actual message.

Filter, encoder, and output plugins should never mutate Heka messages. Sandbox plugins will prevent you from doing so. SandboxEncoders, in particular, expose the `write_message` API that appears to mutate a message, but it actually creates a new message struct rather than modifying the existing one (i.e. copy-on-write). If you implement your own filter, encoder, or output plugins in Go, you must take care to honor this requirement and not mutate any PipelinePack or Message structs.

New in version 0.10.

2.11.10 Transitional Filter / Output APIs for v0.10 Only

Heka's APIs for filter and output plugins have changed dramatically from version 0.9 to version 0.10, to be able to efficiently support disk buffering. The new (and future-proof) APIs for these plugin types are described below. For the 0.10.X series of Heka releases, however, a *very slightly modified version of the older APIs* will be available. This allows Heka users with a significant number of existing filter and output plugins to get them working with a minimal amount of effort.

All filter and output plugin code should soon be upgraded to support the new API code, however, because a) the older APIs incur a considerable performance penalty when *disk buffering* is in use, and b) the older APIs are deprecated and will be removed from Heka entirely in future releases.

2.11.11 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.

Relevant Interfaces

There are three interfaces related to filter plugin implementations. The first of these is the `Filter` interface:

```
type Filter interface {
    Prepare(fr FilterRunner, h PluginHelper) (err error)
    CleanUp()
}
```

The `Filter` interface provides two methods. The first, `Prepare`, will be called by Heka to finalize initialization and start any needed additional goroutines before message processing happens. The provided FilterRunner and PluginHelper interfaces give the filter access to the relevant Heka APIs. Any error returned will indicate that the initialization failed, preventing any messages from being delivered to the filter and possibly causing Heka to shut down, depending on the plugin's `can_exit` value. The `CleanUp` method will be called after message processing has stopped to allow the filter an opportunity to clean up any resources that might need to be freed when exiting.

The second relevant interface for filter plugins is the *MessageProcessor* interface. All filter plugins *must* implement this interface.

The third relevant interface is *TickerPlugin*. The TickerPlugin interface is not strictly required to be implemented by every filter plugin, but most of them will want to do so, and the failure to implement it will mean that your filter will not support the `ticker_interval` config setting.

Buffering

All filter plugins can be configured to support *disk buffering*, so they should regularly call the FilterRunner's `UpdateCursor` method as described *here* to advance the buffer's cursor in cases where buffering is used.

Message Injection

Filter plugins will often need to create new messages that should be injected into Heka's router for further processing by other filter or output plugins, from either the ProcessMessage method or (more often) from TimerEvent. In either case, the process is the same.

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 pack that you're already holding, if called from within ProcessMessage, or zero if called from within TimerEvent. 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 pack has been obtained, a filter plugin can populate its Message struct using any of its provided mutator methods. (Note that this is the *only* time that it is safe to mutate a Message struct from within filter plugin code, since we know that this message has not yet hit the router and there is no risk of a race condition.) The pack can then 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.

Note: In contrast to the Input plugin API, and older versions of the Filter plugin API, filter plugin code should *not* call the PipelinePacks' `Recycle` method when a message has completed its processing. Message recycling is now handled by the FilterRunner.

2.11.12 Encoders

Encoder plugins are the inverse of decoders. They convert Message structs into raw bytes that can be delivered to the outside world. Some encoders will serialize an entire Message struct, such as the *Protobuf Encoder* which uses Heka's native protocol buffers format. Other encoders extract data from the message and insert it into a different format such as plain text or JSON.

The `Encoder` interface consists of one method:

```
type Encoder interface {
    Encode(pack *PipelinePack) (output []byte, err error)
}
```

This method accepts a PiplelinePack containing a populated message object and returns a byte slice containing the data that should be sent out, or an error if serialization fails for some reason. If the encoder wishes to swallow an input message without generating any output (such as for batching, or because the message contains no new data) then nil should be returned for both the output and the error.

Unlike the other plugin types, encoders don't have a PluginRunner, nor do they run in their own goroutines. Outputs invoke encoders directly, by calling the Encode method exposed on the OutputRunner. This has the same signature as

the Encoder interface's Encode method, to which it will will delegate. If `use_framing` is set to true in the output's configuration, however, the OutputRunner will prepend Heka's *Stream Framing* to the generated binary data.

Outputs can also directly access their encoder instance by calling OutputRunner.Encoder(). Encoders themselves don't handle the stream framing, however, so it is recommended that outputs use the OutputRunner method instead.

Even though encoders don't run in their own goroutines, it is possible that they might need to perform some clean up at shutdown time. If this is so, the encoder can implement the `NeedsStopping` interface:

```
type NeedsStopping interface {
    Stop()
}
```

And the `Stop` method will be called during the shutdown sequence.

2.11.13 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. Output plugins interact with Heka nearly identically to filter plugins.

Relevant Interfaces

There are three interfaces related to output plugin implementations. The first of these is the `Output` interface:

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

The only difference between this and the *Filter* interface is that the first argument to `Prepare` is and `OutputRunner` and not a `FilterRunner`. The two interfaces are used in the same way, with Prepare being called before message processing starts and CleanUp being called after message processing has completed.

The other two interfaces relevant to output plugins are *MessageProcessor* and *TickerPlugin*, which are used in precisely the same manner as they are with filter plugins, supporting the same special return errors.

Buffering

Like filters, outputs can be configured to support *disk buffering*, and should call the OutputRunner's `UpdateCursor` method as described *here* to advance the buffer cursor when appropriate.

Message Encoding

The primary way that outputs differ from filters is that output plugins need to either serialize or extract data from the messages they receive and then send that serialized or extracted data to an external destination. The serialization extraction should typically be performed by the output's specified encoder plugin. The OutputRunner exposes the following methods to assist with this:

```
Encode(pack *PipelinePack) (output []byte, err error)
UsesFraming() bool
Encoder() (encoder Encoder)
```

The `Encode` method will use the specified encoder to convert the pack's message to binary data, then if `use_framing` was set to true in the output's configuration it will prepend Heka's *Stream Framing*. The `UsesFraming` method will tell you whether or not `use_framing` was set to true. Finally, the `Encoder` method

will return the actual encoder that was registered. This is useful to check to make sure that an encoder was actually registered, but generally you will want to use OutputRunner.Encode and not Encoder.Encode, since the latter will not honor the output's `use_framing` specification.

Note: In contrast to the Input plugin API, and older versions of the Output plugin API, output plugin code should *not* call the PipelinePacks' `Recycle` method when a message has completed its processing. Message recycling is now handled by the OutputRunner.

2.11.14 Registering Your Plugin

The last step you have to take after implementing your plugin is to register it with Heka 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", "Splitter", "Decoder", "Filter", "Encoder", 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`, `Splitter`, `Decoder`, etc).

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*.

2.11.15 MessageProcessor Interface

Filter and output plugins should both implement the `MessageProcessor` interface:

```
type MessageProcessor interface {
    ProcessMessage(pack *PipelinePack) (err error)
}
```

Once initialization for a filter or an output plugin has been finalized, the `ProcessMessage` method will be called repeatedly, once for every message that is delivered to the plugin. The plugin should process the provided message as needed and should signal the results of the processing with the return value. If the message is successfully processed, `nil` should be returned. If the message cannot be processed and should be dropped, then an error should be returned; Heka will emit the error message to the console and continue.

Special MessageProcessor Return Errors

Heka provides a couple of special error types to pass additional information about the results of a ProcessMessage call. The first is a `RetryMessageError`. This is for when a message can't be processed but future attempts may

succeed, like when an output tries to write to an external service which is down. You can instantiate such an error using the `pipeline.NewRetryMessageError` function, defined as:

```
func NewRetryMessageError(msg string, subs ...interface{}) RetryMessageError
```

The variadic `subs` argument can be used to inject `Printf` style substitutions into the error's message string.

When a RetryMessageError is returned, the next call to ProcessMessage will pass in the same PipelinePack, allowing the plugin to try again. If the error persists and ProcessMessage returns several RetryMessageErrors in a row, an exponential back-off time delay will happen between subsequent retries, to a maximum of one second. The other special errors type is a `PluginExitError`, for cases where something went wrong during processing from which the plugin cannot recover. They are created using `pipeline.NewPluginExitError`:

```
func NewPluginExitError(msg string, subs ...interface{}) PluginExitError
```

Returning such an error will cause the plugin's runner to exit its message processing loop entirely. If any *restarting behavior* has been configured, it will be applied after the exit.

2.11.16 TickerPlugin Interface

Most filter and output plugins will want to implement the `TickerPlugin` interface:

```
type TickerPlugin interface {
    TimerEvent() (err error)
}
```

Providing the `TickerPlugin` interface is optional for any filter or output plugin. If a plugin provides this interface, then the `TimerEvent` method will be called in regular intervals specified by the filter's `ticker_interval` configuration setting. If TimerEvent is implemented but the ticker_interval option is set to zero, then TimerEvent will never be called. If a filter does *not* provide the TickerPlugin interface but the config specifies a ticker_interval, then the configuration will be considered to be invalid and Heka will not start.

If the TimerEvent method returns an error, that error will be logged to Heka's console output. TimerEvent also supports returning the special *PluginExitError* type, in which case the plugin's runner will exit exactly as if such an error were returned from the ProcessMessage method.

Note that the ProcessMessage and TimerEvent methods will *never* be called concurrently for the same plugin. This means that you do not need to worry about race conditions between these two methods. Race conditions are possible, of course, between either of these methods and any additional goroutines that may have been started by the Prepare method; it is up to the developer to manage mutable state carefully to avoid such conditions.

2.11.17 Updating Buffer Cursor

All filter and output plugins support optional use of disk buffering for all messages delivered to the plugin. Each buffer maintains a cursor into the buffer's stream, which should be advanced past a given record when that record has been completely processed.

At first it might seem that the cursor should be updated automatically every time ProcessMessage is called and no error is returned. This doesn't work, however, because plugins will sometimes process in batches, and the cursor shouldn't be advanced past *any* of the messages in the batch until the entire batch is finalized.

For this reason plugins must manually update the queue cursor. Every message has a unique, opaque cursor string value, stored on the PipelinePack as the `QueueCursor` attribute. When a plugin has successfully finalized processing of a message, it should notify Heka by passing this QueueCursor value back to the runner's `UpdateCursor` method, like so:

```
runner.UpdateCursor(pack.QueueCursor)
```

Where `runner` is the FilterRunner or OutputRunner, depending on your plugin type.

When messages are being processed in batches, after each batch UpdateCursor should be called once, with the QueueCursor value from the last message in the batch.

Each UpdateCursor call should pass in a cursor for a message that came *later* in the stream than the previous UpdateCursor call. In other words, it is not possible to move the cursor backwards; if UpdateCursor is passed a cursor value for a messages that was earlier in the stream, an error will be logged to stderr and the cursor will not be updated.

2.12 Heka Message

2.12.1 Message Variables

- uuid (required, []byte) - 16 byte array containing a type 4 UUID.
- timestamp (required, int64) - Number of nanoseconds since the UNIX epoch.
- type (optional, string) - Type of message i.e. "WebLog".
- logger (optional, string) - Data source i.e. "Apache", "TCPInput", "/var/log/test.log".
- severity (optional, int32) - Syslog severity level.
- payload (optional, string) - Textual data i.e. log line, filename.
- env_version (optional, string) - Envelope version. Semantic version of the message content (http://semver.org/ (although in most cases it is just the major version)).
- pid (optional, int32) - Process ID that generated the message.
- hostname (optional, string) - Hostname that generated the message.
- fields (optional, Field) - Array of Field structures.

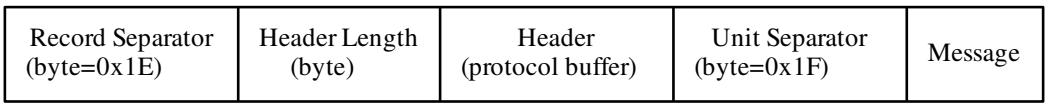
2.12.2 Field Variables

- name (required, string) - Name of the field (key).
- **value_type (optional, int32) - Type of the value stored in this field.**
  - STRING = 0 (default)
  - BYTES = 1
  - INTEGER = 2
  - DOUBLE = 3
  - BOOL = 4
- representation (optional, string) - Freeform metadata string where you can describe what the data in this field represents. This information might provide cues to assist with processing, labeling, or rendering of the data performed by downstream plugins or UI elements. Examples of common usage follow:
  - **Numeric value representation - In most cases it is the unit.**
    - count - It is a standard practice to use 'count' for raw values with no units.
    - KiB

        * mm
    - **String value representation - Ideally it should reference a formal specification but you are free to create you own voca**
        * date-time RFC 3339, section 5.6
        * email RFC 5322, section 3.4.1
        * hostname RFC 1034, section 3.1
        * ipv4 RFC 2673, section 3.2
        * ipv6 RFC 2373, section 2.2
        * uri RFC 3986
    - **How the representation is/can be used**
        * data parsing and validation
        * unit conversion i.e., B to KiB
        * presentation i.e., graph labels, HTML links
- value_* (optional, value_type) - Array of values, only one type will be active at a time.

2.12.3 Stream Framing

Heka has some custom framing that can be used to delimit records when generating a stream of binary data. The entire structure encapsulating a single message consists of a one byte record separator, one byte representing the header length, a protobuf encoded message header, a one byte unit separator, and the binary record content (usually a protobuf encoded Heka message). This message structure is indicated in this diagram:



The header schema is as follows:

- message_length (required, uint32) - length in bytes of the serialized message data
- hmac_hash_function (optional, int32) - enum indicating the hash function used to sign the message, 0 for MD5, 1, for SHA1
- hmac_signer (optional, string) - string token identifying HMAC signer
- hmac_key_version (optional, uint32) - version number of the provided HMAC key
- hmac (optional, []byte) - binary representation of provided HMAC key

Clients interested in decoding a Heka stream will need to read the header length byte to determine the length of the header, extract the encoded header data and decode this into a Header structure using an appropriate protobuf library. From this they can then extract the length of the encoded message data, which can then be extracted from the data stream and processed and/or decoded as needed.

2.13 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.

2.13.1 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%/
- Fields[widget] != NIL

2.13.2 Relational Operators

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

2.13.3 Logical Operators

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

2.13.4 Boolean

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

2.13.5 Constants

- **NIL used to test the existence (!=) or non-existence (==) of a field variable**
  - must be placed on the right side of the comparison e.g., Fields[widget] == NIL

2.13.6 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 e.g., Fields[foo] == 6 where 'foo' is a string

2.13.7 Quoted String

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

2.13.8 Regular Expression String

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

See also:

Regular Expression re2 syntax

2.14 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.)

2.14.1 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

2.14.2 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

API

Functions that must be exposed from the Lua sandbox

int, string process_message() This is the entry point for input plugins to start creating messages. For all other plugin types it is called by Heka when a message is available to the sandbox. The instruction_limit configuration parameter is applied to this function call for non input plugins.

Arguments none

Return

- **int**
  - < 0 for non-fatal failure (increments ProcessMessageFailures)
  - -2 for no output, but no error (encoders only)
  - 0 for success
  - > 0 for fatal error (terminates the sandbox)
- string optional error message

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 that have a ticker_interval configuration greater than zero.

Arguments

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

Return none

Core functions that are exposed to the Lua sandbox

See: https://github.com/mozilla-services/lua_sandbox/

require(libraryName)

Available In All plugin types

add_to_payload(arg1, arg2, ...argN) Appends the arguments to the payload buffer for incremental construction of the final payload output (inject_payload finalizes the buffer and sends the message to Heka). This function is simply a rename of the generic sandbox *output* function to improve the readability of the plugin code.

Arguments

- arg (number, string, bool, nil, circular_buffer)

Return none

Available In Decoders, filters, encoders

Heka specific 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

Available In All plugin types

read_message(variableName, fieldIndex, arrayIndex) Provides access to the Heka message data. Note that both *fieldIndex* and *arrayIndex* are zero-based (i.e. the first element is 0) as opposed to Lua's standard indexing, which is one-based.

Arguments

- **variableName (string)**
  - raw (accesses the raw MsgBytes in the PipelinePack)
  - 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_; zero indexed
- **arrayIndex (unsigned) only used in combination with the Fields variableName**
  - use to retrieve a specific element out of a field containing an array; zero indexed

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

Available In Decoders, filters, encoders, outputs

write_message(variableName, value, representation, fieldIndex, arrayIndex) New in version 0.5.

Mutates specified field value on the message that is being decoded.

Arguments

- **variableName (string)**
  - Uuid (accepts raw bytes or RFC4122 string representation)
  - Type (string)
  - Logger (string)
  - Payload (string)
  - EnvVersion (string)
  - Hostname (string)
  - **Timestamp (accepts Unix ns-since-epoch number or a handful of** parseable string representations.)
  - Severity (number or int-parseable string)
  - Pid (number or int-parseable string)
  - Fields[_name_] (field type determined by value type: bool, number, or string)
- **value (bool, number or string)**
  - value to which field should be set
- **representation (string) only used in combination with the Fields variableName**
  - representation tag to set
- **fieldIndex (unsigned) only used in combination with the Fields variableName**
  - use to set a specfic instance of a repeated field _name_
- **arrayIndex (unsigned) only used in combination with the Fields variableName**
  - use to set a specific element of a field containing an array

Return none

Available In Decoders, encoders

read_next_field() Deprecated since version 0.10.0: Use read_message("raw") instead e.g.,

```
local msg = decode_message(read_message("raw"))
if msg.Fields then
    for i, f in ipairs(msg.Fields) do
    -- process fields
    end
end
```

Iterates through the message fields returning the field contents or nil when the end is reached.

Arguments none

Return value_type, name, value, representation, count (number of items in the field array)

Available In Decoders, filters, encoders, outputs

inject_payload(payload_type, payload_name, arg3, ..., argN)

Creates a new Heka message using the contents of the payload buffer (pre-populated with *add_to_payload*) combined with any additional payload_args passed to this function. The output buffer is cleared after the injection. The payload_type and payload_name arguments are two pieces of optional metadata. If specified, they will be included as fields in the injected message e.g., 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 *global configuration 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.
- arg3 (**optional**) Same type restrictions as add_to_payload.
- ...
- argN

Return none

Available In Decoders, filters, encoders

inject_message(message) 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 if a 16 byte binary UUID is not provided.
- 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 (hash structure) can be represented in multiple forms and support the following primitive types: string, double, bool. These constructs can be added to the 'Fields' table in the message structure.
  - name=value e.g., foo="bar"; foo=1; foo=true
  - name={array} e.g., foo={"b", "a", "r"}
  - name={object} e.g., foo={value=1, representation="s"}; foo={value={1010, 2200, 1567}, value_type=2, representation="ms"}
    - value (required) may be a single value or an array of values

        * value_type (optional) value_type enum. This is most useful for specifying that numbers should be treated as integers as opposed defaulting to doubles.
        * representation (optional) metadata for display and unit management

    - **Fields (array structure)**
        - same as above but the hash key name is moved into the object as 'name' e.g., Fields = {{name="foo", value="bar"}}

Arguments

- message (table or string) A table with the message structure documented below or a string with a Heka protobuf encoded message.

Return none

Available In Inputs, decoders, filters, encoders

Notes Injection limits are only enforced on filter plugins. See `max_*_inject` in the *global configuration options*.

decode_message(heka_protobuf_string) Converts a Heka protobuf encoded message string into a Lua table.

Arguments

- heka_message (string) Lua variable containing a Heka protobuf encoded message

Return

- message (table) The array based version of the message structure with the value member always being an array (even if there is only a single item). This format makes working with the output more consistent. The wide variation in the inject table format is to ease the construction of the message especially when using an LPeg grammar transformation.

Lua Message Hash Based Field Structure

```
{
Uuid        = "data",               -- ignored if not 16 byte raw binary UUID
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
Payload     = "Test Payload",
EnvVersion  = "0.8",
Pid         = 1234,
Severity    = 6,
Fields      = {
            http_status     = 200, -- encoded as a double
            request_size    = {value=1413, value_type=2, representation="B"} -- encoded as an intege
            }
}
```

Lua Message Array Based Field Structure

```
{
-- same as above
Fields      = {
            {name="http_status", value=200}, -- encoded as a double
```

```
            {name="request_size", value=1413, value_type=2, representation="B"} -- encoded as an inte
            }
}
```

2.14.3 Lua Sandbox Tutorial

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)

```
require "string"

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)
    count = 0
    inject_payload("txt", "",
                   string.format("%d messages in the last minute; total=%d", count, total))
end
```

3. Setup the configuration

```
[demo_counter]
type = "SandboxFilter"
message_matcher = "Type == 'demo'"
ticker_interval = 60
filename = "counter.lua"
preserve_data = true
```

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

```
require "string"

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

2.14.4 Sandbox Input

New in version 0.9.

Plugin Name: **SandboxInput**

The SandboxInput provides a flexible execution environment for data ingestion and transformation without the need to recompile Heka. Like all other sandboxes it needs to implement a process_message function. However, it doesn't have to return until shutdown. If you would like to implement a polling interface process_message can return zero when complete and it will be called again the next time TickerInterval fires (if ticker_interval was set to zero it would simply exit after running once). See *Sandbox*. Config:

- All of the common input configuration parameters are ignored since the data processing (splitting and decoding) should happen in the plugin.
- ***Common Sandbox Parameters***
  - `instruction_limit` is always set to zero for SandboxInputs

Example

```
[MemInfo]
type = "SandboxInput"
filename = "meminfo.lua"

[MemInfo.config]
path = "/proc/meminfo"
```

Available Sandbox Inputs

- none

2.14.5 Sandbox Decoder

Plugin Name: **SandboxDecoder**

The SandboxDecoder provides an isolated execution environment for data parsing and complex transformations without the need to recompile Heka. See *Sandbox*. Config:

- *Common Sandbox Parameters*

Example

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

Available Sandbox Decoders

Apache Access Log Decoder

Parses the Apache access logs based on the Apache 'LogFormat' configuration directive. The Apache format specifiers are mapped onto the Nginx variable names where applicable e.g. %a -> remote_addr. This allows generic web filters and outputs to work with any HTTP server input.

Config:

- **log_format (string)** The 'LogFormat' configuration directive from the apache2.conf. %t variables are converted to the number of nanosecond since the Unix epoch and used to set the Timestamp on the message. http://httpd.apache.org/docs/2.4/mod/mod_log_config.html
- **type (string, optional, default nil):** Sets the message 'Type' header to the specified value
- **user_agent_transform (bool, optional, default false)** Transform the http_user_agent into user_agent_browser, user_agent_version, user_agent_os.
- **user_agent_keep (bool, optional, default false)** Always preserve the http_user_agent value if transform is enabled.
- **user_agent_conditional (bool, optional, default false)** Only preserve the http_user_agent value if transform is enabled and fails.
- **payload_keep (bool, optional, default false)** Always preserve the original log line in the message payload.

Example Heka Configuration

```
[TestWebserver]
type = "LogstreamerInput"
log_directory = "/var/log/apache"
file_match = 'access\.log'
decoder = "CombinedLogDecoder"

[CombinedLogDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/apache_access.lua"

[CombinedLogDecoder.config]
type = "combined"
user_agent_transform = true
# combined log format
log_format = '%h %l %u %t \"%r\" %>s %O \"%{Referer}i\" \"%{User-Agent}i\"'

# common log format
# log_format = '%h %l %u %t \"%r\" %>s %O'

# vhost_combined log format
# log_format = '%v:%p %h %l %u %t \"%r\" %>s %O \"%{Referer}i\" \"%{User-Agent}i\"'
```

```
# referer log format
# log_format = '%{Referer}i -> %U'
```

Example Heka Message

Timestamp 2014-01-10 07:04:56 -0800 PST

Type combined

Hostname test.example.com

Pid 0

UUID 8e414f01-9d7f-4a48-a5e1-ae92e5954df5

Logger TestWebserver

Payload

EnvVersion

Severity 7

Fields

name:"remote_user" value_string:"-"
name:"http_x_forwarded_for" value_string:"-"
name:"http_referer" value_string:"-"
name:"body_bytes_sent" value_type:DOUBLE representation:"B" value_double:82
name:"remote_addr" value_string:"62.195.113.219" representation:"ipv4"
name:"status" value_type:DOUBLE value_double:200
name:"request" value_string:"GET /v1/recovery_email/status HTTP/1.1"
name:"user_agent_os" value_string:"FirefoxOS"
name:"user_agent_browser" value_string:"Firefox"
name:"user_agent_version" value_type:DOUBLE value_double:29

Graylog Extended Log Format Decoder

Parses a payload containing JSON in the Graylog2 Extended Format specficiation. http://graylog2.org/resources/gelf/specification

Config:

- **type (string, optional, default nil):** Sets the message 'Type' header to the specified value
- **payload_keep (bool, optional, default false)** Always preserve the original log line in the message payload.

Example of Graylog2 Exteded Format Log

```
{
  "version": "1.1",
  "host": "rogueethic.com",
  "short_message": "This is a short message to identify what is going on.",
  "full_message": "An entire backtrace\ncould\ngo\nhere",
  "timestamp": 1385053862.3072,
  "level": 1,
  "_user_id": 9001,
  "_some_info": "foo",
  "_some_env_var": "bar"
}
```

Example Heka Configuration

```
[GELFLogInput]
type = "LogstreamerInput"
log_directory = "/var/log"
file_match = 'application\.gelf'
decoder = "GraylogDecoder"

[GraylogDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/graylog_decoder.lua"

    [GraylogDecoder.config]
    type = "gelf"
    payload_keep = true
```

Linux CPU Stats Decoder

Parses a payload containing the contents of file */proc/stat*.

Config:

- **payload_keep (bool, optional, default false)** Always preserve the original log line in the message payload.

Example Heka Configuration

```
[ProcStats]
type = "FilePollingInput"
ticker_interval = 1
file_path = "/proc/stat"
decoder = "ProcStatDecoder"

[ProcStatDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/linux_procstat.lua"
```

Example Heka Message

Timestamp 2014-12-10 22:38:24 +0000 UTC

Type stats.proc

Hostname yourhost.net

Pid 0

Uuid d2546942-7c36-4042-ad2e-f6bfdac11cdb

Logger

Payload

EnvVersion

Severity 7

Fields

name:"cpu" type:double value:[14384,125,3330,946000,333,0,356,0,0,0]
name:"cpu[1-#]" type:double value:[14384,125,3330,946000,333,0,356,0,0,0]
name:"ctxt" type:double value:2808304
name:"btime" type:double value:1423004780

name:"intr" type:double value:[14384,125,3330,0,0,0,0,0,0,0...0]
name:"processes" type:double value:3811
name:"procs_running" type:double value:1
name:"procs_blocked" type:double value:0
name:"softirq" type:double value:[288977,23,101952,19,13046,19217,7,...]

Cpu fields: 1 2 3 4 5 6 7 8 9 10 user nice system idle [iowait] [irq] [softirq] [steal] [guest] [guestnice]
Note: systems provide user, nice, system, idle. Other fields depend on kernel.

intr This line shows counts of interrupts serviced since boot time, for each of the possible system interrupts. The first column is the total of all interrupts serviced including unnumbered architecture specific interrupts; each subsequent column is the total for that particular numbered interrupt. Unnumbered interrupts are not shown, only summed into the total.

Linux Disk Stats Decoder

Parses a payload containing the contents of a */sys/block/$DISK/stat* file (where *$DISK* is a disk identifier such as *sda*) into a Heka message struct. This also tries to obtain the TickerInterval of the input it recieved the data from, by extracting it from a message field named *TickerInterval*.

Config:

- **payload_keep (bool, optional, default false)** Always preserve the original log line in the message payload.

Example Heka Configuration

```
[DiskStats]
type = "FilePollingInput"
ticker_interval = 1
file_path = "/sys/block/sda1/stat"
decoder = "DiskStatsDecoder"

[DiskStatsDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/linux_diskstats.lua"
```

Example Heka Message

Timestamp 2014-01-10 07:04:56 -0800 PST

Type stats.diskstats

Hostname test.example.com

Pid 0

UUID 8e414f01-9d7f-4a48-a5e1-ae92e5954df5

Payload

EnvVersion

Severity 7

Fields

name:"ReadsCompleted" value_type:DOUBLE value_double:"20123"
name:"ReadsMerged" value_type:DOUBLE value_double:"11267"
name:"SectorsRead" value_type:DOUBLE value_double:"1.094968e+06"
name:"TimeReading" value_type:DOUBLE value_double:"45148"
name:"WritesCompleted" value_type:DOUBLE value_double:"1278"

name:"WritesMerged" value_type:DOUBLE value_double:"1278"
name:"SectorsWritten" value_type:DOUBLE value_double:"206504"
name:"TimeWriting" value_type:DOUBLE value_double:"3348"
name:"TimeDoingIO" value_type:DOUBLE value_double:"4876"
name:"WeightedTimeDoingIO" value_type:DOUBLE value_double:"48356"
name:"NumIOInProgress" value_type:DOUBLE value_double:"3"
name:"TickerInterval" value_type:DOUBLE value_double:"2"
name:"FilePath" value_string:"/sys/block/sda/stat"

Linux Load Average Decoder

Parses a payload containing the contents of a */proc/loadavg* file into a Heka message.

Config:

- **payload_keep (bool, optional, default false)** Always preserve the original log line in the message payload.

Example Heka Configuration

```
[LoadAvg]
type = "FilePollingInput"
ticker_interval = 1
file_path = "/proc/loadavg"
decoder = "LoadAvgDecoder"

[LoadAvgDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/linux_loadavg.lua"
```

Example Heka Message

Timestamp 2014-01-10 07:04:56 -0800 PST

Type stats.loadavg

Hostname test.example.com

Pid 0

UUID 8e414f01-9d7f-4a48-a5e1-ae92e5954df5

Payload

EnvVersion

Severity 7

Fields

name:"1MinAvg" value_type:DOUBLE value_double:"3.05"
name:"5MinAvg" value_type:DOUBLE value_double:"1.21"
name:"15MinAvg" value_type:DOUBLE value_double:"0.44"
name:"NumProcesses" value_type:DOUBLE value_double:"11"
name:"FilePath" value_string:"/proc/loadavg"

Linux Memory Stats Decoder

Parses a payload containing the contents of a */proc/meminfo* file into a Heka message.

Config:

- **payload_keep (bool, optional, default false)** Always preserve the original log line in the message payload.

Example Heka Configuration

```
[MemStats]
type = "FilePollingInput"
ticker_interval = 1
file_path = "/proc/meminfo"
decoder = "MemStatsDecoder"

[MemStatsDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/linux_memstats.lua"
```

Example Heka Message

Timestamp 2014-01-10 07:04:56 -0800 PST

Type stats.memstats

Hostname test.example.com

Pid 0

UUID 8e414f01-9d7f-4a48-a5e1-ae92e5954df5

Payload

EnvVersion

Severity 7

Fields

name:"MemTotal" value_type:DOUBLE representation:"kB" value_double:"4047616"
name:"MemFree" value_type:DOUBLE representation:"kB" value_double:"3432216"
name:"Buffers" value_type:DOUBLE representation:"kB" value_double:"82028"
name:"Cached" value_type:DOUBLE representation:"kB" value_double:"368636"
name:"FilePath" value_string:"/proc/meminfo"

The total available fields can be found in *man procfs*. All fields are of type double, and the representation is in kB (except for the HugePages fields). Here is a full list of fields available:

MemTotal, MemFree, Buffers, Cached, SwapCached, Active, Inactive, Active(anon), Inactive(anon), Active(file), Inactive(file), Unevictable, Mlocked, SwapTotal, SwapFree, Dirty, Writeback, AnonPages, Mapped, Shmem, Slab, SReclaimable, SUnreclaim, KernelStack, PageTables, NFS_Unstable, Bounce, WritebackTmp, CommitLimit, Committed_AS, VmallocTotal, VmallocUsed, VmallocChunk, HardwareCorrupted, AnonHugePages, HugePages_Total, HugePages_Free, HugePages_Rsvd, HugePages_Surp, Hugepagesize, DirectMap4k, DirectMap2M, DirectMap1G.

Note that your available fields may have a slight variance depending on the system's kernel version.

MySQL Slow Query Log Decoder

Parses and transforms the MySQL slow query logs. Use mariadb_slow_query.lua to parse the MariaDB variant of the MySQL slow query logs.

Config:

- **truncate_sql (int, optional, default nil)** Truncates the SQL payload to the specified number of bytes (not UTF-8 aware) and appends "...". If the value is nil no truncation is performed. A negative value will truncate the specified number of bytes from the end.

Example Heka Configuration

```
[Sync-1_5-SlowQuery]
type = "LogstreamerInput"
log_directory = "/var/log/mysql"
file_match = 'mysql-slow\.log'
parser_type = "regexp"
delimiter = "\n(# User@Host:)"
delimiter_location = "start"
decoder = "MySqlSlowQueryDecoder"

[MySqlSlowQueryDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/mysql_slow_query.lua"

    [MySqlSlowQueryDecoder.config]
    truncate_sql = 64
```

Example Heka Message

Timestamp 2014-05-07 15:51:28 -0700 PDT

Type mysql.slow-query

Hostname 127.0.0.1

Pid 0

UUID 5324dd93-47df-485b-a88e-429f0fcd57d6

Logger Sync-1_5-SlowQuery

Payload /* [queryName=FIND_ITEMS] */ SELECT bso.userid, bso.collection, ...

EnvVersion

Severity 7

Fields

name:"Rows_examined" value_type:DOUBLE value_double:16458
name:"Query_time" value_type:DOUBLE representation:"s" value_double:7.24966
name:"Rows_sent" value_type:DOUBLE value_double:5001
name:"Lock_time" value_type:DOUBLE representation:"s" value_double:0.047038

Nginx Access Log Decoder

Parses the Nginx access logs based on the Nginx 'log_format' configuration directive.

Config:

- **log_format (string)** The 'log_format' configuration directive from the nginx.conf. $time_local or $time_iso8601 variable is converted to the number of nanosecond since the Unix epoch and used to set the Timestamp on the message. http://nginx.org/en/docs/http/ngx_http_log_module.html
- **type (string, optional, default nil):** Sets the message 'Type' header to the specified value

- **user_agent_transform (bool, optional, default false)** Transform the http_user_agent into user_agent_browser, user_agent_version, user_agent_os.
- **user_agent_keep (bool, optional, default false)** Always preserve the http_user_agent value if transform is enabled.
- **user_agent_conditional (bool, optional, default false)** Only preserve the http_user_agent value if transform is enabled and fails.
- **payload_keep (bool, optional, default false)** Always preserve the original log line in the message payload.

Example Heka Configuration

```
[TestWebserver]
type = "LogstreamerInput"
log_directory = "/var/log/nginx"
file_match = 'access\.log'
decoder = "CombinedLogDecoder"

[CombinedLogDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/nginx_access.lua"

[CombinedLogDecoder.config]
type = "combined"
user_agent_transform = true
# combined log format
log_format = '$remote_addr - $remote_user [$time_local] "$request" $status $body_bytes_sent "$http_re
```

Example Heka Message

Timestamp 2014-01-10 07:04:56 -0800 PST

Type combined

Hostname test.example.com

Pid 0

UUID 8e414f01-9d7f-4a48-a5e1-ae92e5954df5

Logger TestWebserver

Payload

EnvVersion

Severity 7

Fields

name:"remote_user" value_string:"-"
name:"http_x_forwarded_for" value_string:"-"
name:"http_referer" value_string:"-"
name:"body_bytes_sent" value_type:DOUBLE representation:"B" value_double:82
name:"remote_addr" value_string:"62.195.113.219" representation:"ipv4"
name:"status" value_type:DOUBLE value_double:200
name:"request" value_string:"GET /v1/recovery_email/status HTTP/1.1"
name:"user_agent_os" value_string:"FirefoxOS"
name:"user_agent_browser" value_string:"Firefox"
name:"user_agent_version" value_type:DOUBLE value_double:29

Nginx Error Log Decoder

Parses the Nginx error logs based on the Nginx hard coded internal format.

Config:

- **tz (string, optional, defaults to UTC)** The conversion actually happens on the Go side since there isn't good TZ support here.

Example Heka Configuration

```
[TestWebserverError]
type = "LogstreamerInput"
log_directory = "/var/log/nginx"
file_match = 'error\.log'
decoder = "NginxErrorDecoder"

[NginxErrorDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/nginx_error.lua"

[NginxErrorDecoder.config]
tz = "America/Los_Angeles"
```

Example Heka Message

Timestamp 2014-01-10 07:04:56 -0800 PST

Type nginx.error

Hostname trink-x230

Pid 16842

UUID 8e414f01-9d7f-4a48-a5e1-ae92e5954df5

Logger TestWebserverError

Payload using inherited sockets from "6;"

EnvVersion

Severity 5

Fields

name:"tid" value_type:DOUBLE value_double:0
name:"connection" value_type:DOUBLE value_double:8878

Rsyslog Decoder

Parses the rsyslog output using the string based configuration template.

Config:

- **hostname_keep (boolean, defaults to false)** Always preserve the original 'Hostname' field set by Logstreamer's 'hostname' configuration setting.
- **template (string)** The 'template' configuration string from rsyslog.conf. http://rsyslog-5-8-6-doc.neocities.org/rsyslog_conf_templates.html

- **tz (string, optional, defaults to UTC)** If your rsyslog timestamp field in the template does not carry zone offset information, you may set an offset to be applied to your events here. Typically this would be used with the "Traditional" rsyslog formats.

  Parsing is done by Go, supports values of "UTC", "Local", or a location name corresponding to a file in the IANA Time Zone database, e.g. "America/New_York".

Example Heka Configuration

```
[RsyslogDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/rsyslog.lua"

[RsyslogDecoder.config]
type = "RSYSLOG_TraditionalFileFormat"
template = '%TIMESTAMP% %HOSTNAME% %syslogtag%%msg:::sp-if-no-1st-sp%%msg:::drop-last-lf%\n'
tz = "America/Los_Angeles"
```

Example Heka Message

Timestamp 2014-02-10 12:58:58 -0800 PST

Type RSYSLOG_TraditionalFileFormat

Hostname trink-x230

Pid 0

UUID e0eef205-0b64-41e8-a307-5772b05e16c1

Logger RsyslogInput

Payload "imklog 5.8.6, log source = /proc/kmsg started."

EnvVersion

Severity 7

Fields

name:"programname" value_string:"kernel"

Externally Available Sandbox Decoders

- GZip Payload Decoder

2.14.6 Sandbox Filter

Plugin Name: **SandboxFilter**

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

Config:

- *Common Filter Parameters*
- *Common Sandbox Parameters*
- **timer_event_on_shutdown (bool):** True if the sandbox should have its timer_event function called on shutdown.

Example:

```
[hekabench_counter]
type = "SandboxFilter"
message_matcher = "Type == 'hekabench'"
ticker_interval = 1
filename = "counter.lua"
preserve_data = true
profile = false

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

Available Sandbox Filters

Circular Buffer Delta Aggregator

Collects the circular buffer delta output from multiple instances of an upstream sandbox filter (the filters should all be the same version at least with respect to their cbuf output). The purpose is to recreate the view at a larger scope in each level of the aggregation i.e., host view -> datacenter view -> service level view.

Config:

- **enable_delta (bool, optional, default false)** Specifies whether or not this aggregator should generate cbuf deltas.
- **anomaly_config(string) - (see *Anomaly Detection Module*)** A list of anomaly detection specifications. If not specified no anomaly detection/alerting will be performed.
- **preservation_version (uint, optional, default 0)** If *preserve_data = true* is set in the SandboxFilter configuration, then this value should be incremented every time the *enable_delta* configuration is changed to prevent the plugin from failing to start during data restoration.

Example Heka Configuration

```
[TelemetryServerMetricsAggregator]
type = "SandboxFilter"
message_matcher = "Logger == 'TelemetryServerMetrics' && Fields[payload_type] == 'cbufd'"
ticker_interval = 60
filename = "lua_filters/cbufd_aggregator.lua"
preserve_data = true

[TelemetryServerMetricsAggregator.config]
enable_delta = false
anomaly_config = 'roc("Request Statistics", 1, 15, 0, 1.5, true, false)'
preservation_version = 0
```

Circular Buffer Delta Aggregator (by hostname)

Collects the circular buffer delta output from multiple instances of an upstream sandbox filter (the filters should all be the same version at least with respect to their cbuf output). Each column from the source circular buffer will become its own graph. i.e., 'Error Count' will become a graph with each host being represented in a column.

Config:

- **max_hosts (uint)** Pre-allocates the number of host columns in the graph(s). If the number of active hosts exceed this value, the plugin will terminate.

- **rows (uint)** The number of rows to keep from the original circular buffer. Storing all the data from all the hosts is not practical since you will most likely run into memory and output size restrictions (adjust the view down as necessary).
- **host_expiration (uint, optional, default 120 seconds)** The amount of time a host has to be inactive before it can be replaced by a new host.
- **preservation_version (uint, optional, default 0)** If *preserve_data = true* is set in the SandboxFilter configuration, then this value should be incremented every time the *max_hosts* or *rows* configuration is changed to prevent the plugin from failing to start during data restoration.

Example Heka Configuration

```
[TelemetryServerMetricsHostAggregator]
type = "SandboxFilter"
message_matcher = "Logger == 'TelemetryServerMetrics' && Fields[payload_type] == 'cbufd'"
ticker_interval = 60
filename = "lua_filters/cbufd_host_aggregator.lua"
preserve_data = true

[TelemetryServerMetricsHostAggregator.config]
max_hosts = 5
rows = 60
host_expiration = 120
preservation_version = 0
```

CPU Stats Filter

Calculates deltas in /proc/stat data. Also emits CPU percentage utilization information.

Config:

- **whitelist (string, optional, default "")** Only process fields that fit the pattern, defaults to match all.
- **extras (boolean, optional, default false)** Process extra fields like ctxt, softirq, cpu fields.
- **percent_integer (boolean, optional, default true)** Process percentage as whole number.

Example Heka Configuration

```
[ProcStats]
type = "FilePollingInput"
ticker_interval = 1
file_path = "/proc/stat"
decoder = "ProcStatDecoder"

[ProcStatDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/linux_procstat.lua"

[ProcStatFilter]
type = "SandboxFilter"
filename = "lua_filters/procstat.lua"
preserve_data = true
message_matcher = "Type == 'stats.procstat'"
[ProcStatFilter.config]
    whitelist = "cpu$"
    extras = false
    percent_integer = true
```

Cpu fields: 1 2 3 4 5 6 7 8 9 10 user nice system idle [iowait] [irq] [softirq] [steal] [guest] [guestnice] Note: systems provide user, nice, system, idle. Other fields depend on kernel.

user: Time spent executing user applications (user mode). nice: Time spent executing user applications with low priority (nice). system: Time spent executing system calls (system mode). idle: Idle time. iowait: Time waiting for I/O operations to complete. irq: Time spent servicing interrupts. softirq: Time spent servicing soft-interrupts. steal: ticks spent executing other virtual hosts [virtualization setups] guest: Used in virtualization setups. guestnice: running a niced guest

intr This line shows counts of interrupts serviced since boot time, for each of the possible system interrupts. The first column is the total of all interrupts serviced including unnumbered architecture specific interrupts; each subsequent column is the total for that particular numbered interrupt. Unnumbered interrupts are not shown, only summed into the total.

ctxt 115315 The number of context switches that the system underwent.

btime 769041601 Boot time, in seconds since the Epoch, 1970-01-01 00:00:00 +0000 (UTC).

processes 86031 Number of forks since boot.

procs_running 6 Number of process in runnable state. (Linux 2.5.45 onward.)

procs_blocked 2 Number of process blocked waiting for I/O to complete. (Linux 2.5.45 onward.)

softirq 288977 23 101952 19 13046 19217 7 19125 92077 389 43122 Time spent servicing soft-interrupts.

Disk Stats Filter

Graphs disk IO stats. It automatically converts the running totals of Writes and Reads into rates of the values. The time based fields are left as running totals of the amount of time doing IO. Expects to receive messages with disk IO data embedded in a particular set of message fields which matches what is generated by *Linux Disk Stats Decoder*: WritesCompleted, ReadsCompleted, SectorsWritten, SectorsRead, WritesMerged, ReadsMerged, TimeWriting, TimeReading, TimeDoingIO, WeightedTimeDoingIO, TickerInterval.

Config:

- **rows (uint, optional, default 1440)** Sets the size of the sliding window i.e., 1440 rows representing 60 seconds per row is a 24 sliding hour window with 1 minute resolution.
- anomaly_config(string) - (see *Anomaly Detection Module*)

Example Heka Configuration

```
[DiskStatsFilter]
type = "SandboxFilter"
filename = "lua_filters/diskstats.lua"
preserve_data = true
message_matcher = "Type == 'stats.diskstats'"
ticker_interval = 10
```

Frequent Items

Calculates the most frequent items in a data stream.

Config:

- **message_variable (string)** The message variable name containing the items to be counted.
- **max_items (uint, optional, default 1000)** The maximum size of the sample set (higher will produce a more accurate list).

- **min_output_weight (uint, optional, default 100)** Used to reduce the long tail output by only outputting the higher frequency items.
- **reset_days (uint, optional, default 1)** Resets the list after the specified number of days (on the UTC day boundary). A value of 0 will never reset the list.

Example Heka Configuration

```
[FxaAuthServerFrequentIP]
type = "SandboxFilter"
filename = "lua_filters/frequent_items.lua"
ticker_interval = 60
preserve_data = true
message_matcher = "Logger == 'nginx.access' && Type == 'fxa-auth-server'"

[FxaAuthServerFrequentIP.config]
message_variable = "Fields[remote_addr]"
max_items = 10000
min_output_weight = 100
reset_days = 1
```

Heka Memory Statistics (self monitoring)

Graphs the Heka memory statistics using the heka.memstat message generated by pipeline/report.go.

Config:

- **rows (uint, optional, default 1440)** Sets the size of the sliding window i.e., 1440 rows representing 60 seconds per row is a 24 sliding hour window with 1 minute resolution.
- **sec_per_row (uint, optional, default 60)** Sets the size of each bucket (resolution in seconds) in the sliding window.
- **anomaly_config (string, optional)** See *Anomaly Detection Module*.
- **preservation_version (uint, optional, default 0)** If *preserve_data = true* is set in the SandboxFilter configuration, then this value should be incremented every time the *rows* or *sec_per_row* configuration is changed to prevent the plugin from failing to start during data restoration.

Example Heka Configuration

```
[HekaMemstat]
type = "SandboxFilter"
filename = "lua_filters/heka_memstat.lua"
ticker_interval = 60
preserve_data = true
message_matcher = "Type == 'heka.memstat'"
```

Heka Message Schema (Message Documentation)

Generates documentation for each unique message in a data stream. The output is a hierarchy of Logger, Type, EnvVersion, and a list of associated message field attributes including their counts (number in the brackets). This plugin is meant for data discovery/exploration and should not be left running on a production system.

Config:

<none>

Example Heka Configuration

```
[SyncMessageSchema]
type = "SandboxFilter"
filename = "lua_filters/heka_message_schema.lua"
ticker_interval = 60
preserve_data = false
message_matcher = "Logger != 'SyncMessageSchema' && Logger =~ /^Sync/"
```

Example Output

Sync-1_5-Webserver [54600]
    slf [54600]
        -no version- [54600]
            upstream_response_time (mismatch)
            http_user_agent (string)
            body_bytes_sent (number)
            remote_addr (string)
            request (string)
            upstream_status (mismatch)
            status (number)
            request_time (number)
            request_length (number)
Sync-1_5-SlowQuery [37]
    mysql.slow-query [37]
        -no version- [37]
            Query_time (number)
            Rows_examined (number)
            Rows_sent (number)
            Lock_time (number)

Heka Process Message Failures (self monitoring)

Monitors Heka's process message failures by plugin.

Config:

- **anomaly_config(string) - (see *Anomaly Detection Module*)** A list of anomaly detection specifications. If not specified a default of 'mww_nonparametric("DEFAULT", 1, 5, 10, 0.7)' is used. The "DEFAULT" settings are applied to any plugin without an explict specification.

Example Heka Configuration

```
[HekaProcessMessageFailures]
type = "SandboxFilter"
filename = "lua_filters/heka_process_message_failures.lua"
ticker_interval = 60
preserve_data = false # the counts are reset on Heka restarts and the monitoring should be too.
message_matcher = "Type == 'heka.all-report'"
```

HTTP Status Graph

Graphs HTTP status codes using the numeric Fields[status] variable collected from web server access logs.

Config:

- **sec_per_row (uint, optional, default 60)** Sets the size of each bucket (resolution in seconds) in the sliding window.
- **rows (uint, optional, default 1440)** Sets the size of the sliding window i.e., 1440 rows representing 60 seconds per row is a 24 sliding hour window with 1 minute resolution.
- **anomaly_config (string, optional)** See *Anomaly Detection Module*.
- **alert_throttle (uint, optional, default 3600)** Sets the throttle for the anomaly alert, in seconds.
- **preservation_version (uint, optional, default 0)** If *preserve_data = true* is set in the SandboxFilter configuration, then this value should be incremented every time the *sec_per_row* or *rows* configuration is changed to prevent the plugin from failing to start during data restoration.

Example Heka Configuration

```
[FxaAuthServerHTTPStatus]
type = "SandboxFilter"
filename = "lua_filters/http_status.lua"
ticker_interval = 60
preserve_data = true
message_matcher = "Logger == 'nginx.access' && Type == 'fxa-auth-server'"

[FxaAuthServerHTTPStatus.config]
sec_per_row = 60
rows = 1440
anomaly_config = 'roc("HTTP Status", 2, 15, 0, 1.5, true, false) roc("HTTP Status", 4, 15, 0, 1.5, tr
alert_throttle = 300
preservation_version = 0
```

Load Average Filter

Graphs the load average and process count data. Expects to receive messages containing fields entitled *1MinAvg*, *5MinAvg*, *15MinAvg*, and *NumProcesses*, such as those generated by the *Linux Load Average Decoder*.

Config:

- **sec_per_row (uint, optional, default 60)** Sets the size of each bucket (resolution in seconds) in the sliding window.
- **rows (uint, optional, default 1440)** Sets the size of the sliding window i.e., 1440 rows representing 60 seconds per row is a 24 sliding hour window with 1 minute resolution.
- **anomaly_config (string, optional)** See *Anomaly Detection Module*.
- **preservation_version (uint, optional, default 0)** If *preserve_data = true* is set in the SandboxFilter configuration, then this value should be incremented every time the *sec_per_row* or *rows* configuration is changed to prevent the plugin from failing to start during data restoration.

Example Heka Configuration

```
[LoadAvgFilter]
type = "SandboxFilter"
filename = "lua_filters/loadavg.lua"
ticker_interval = 60
```

```
preserve_data = true
message_matcher = "Type == 'stats.loadavg'"
```

Memory Stats Filter

Graphs memory usage statistics. Expects to receive messages with memory usage data embedded in a specific set of message fields, which matches the messages generated by *Linux Memory Stats Decoder*: MemFree, Cached, Active, Inactive, VmallocUsed, Shmem, SwapCached.

Config:

- **sec_per_row (uint, optional, default 60)** Sets the size of each bucket (resolution in seconds) in the sliding window.
- **rows (uint, optional, default 1440)** Sets the size of the sliding window i.e., 1440 rows representing 60 seconds per row is a 24 sliding hour window with 1 minute resolution.
- **anomaly_config (string, optional)** See *Anomaly Detection Module*.
- **preservation_version (uint, optional, default 0)** If *preserve_data = true* is set in the SandboxFilter configuration, then this value should be incremented every time the *sec_per_row* or *rows* configuration is changed to prevent the plugin from failing to start during data restoration.

Example Heka Configuration

```
[MemoryStatsFilter]
type = "SandboxFilter"
filename = "lua_filters/memstats.lua"
ticker_interval = 60
preserve_data = true
message_matcher = "Type == 'stats.memstats'"
```

MySQL Slow Query

Graphs MySQL slow query data produced by the *MySQL Slow Query Log Decoder*.

Config:

- **sec_per_row (uint, optional, default 60)** Sets the size of each bucket (resolution in seconds) in the sliding window.
- **rows (uint, optional, default 1440)** Sets the size of the sliding window i.e., 1440 rows representing 60 seconds per row is a 24 sliding hour window with 1 minute resolution.
- **anomaly_config (string, optional)** See *Anomaly Detection Module*.
- **preservation_version (uint, optional, default 0)** If *preserve_data = true* is set in the SandboxFilter configuration, then this value should be incremented every time the *sec_per_row* or *rows* configuration is changed to prevent the plugin from failing to start during data restoration.

Example Heka Configuration

```
[Sync-1_5-SlowQueries]
type = "SandboxFilter"
message_matcher = "Logger == 'Sync-1_5-SlowQuery'"
ticker_interval = 60
filename = "lua_filters/mysql_slow_query.lua"

    [Sync-1_5-SlowQueries.config]
```

```
    anomaly_config = 'mww_nonparametric("Statistics", 5, 15, 10, 0.8)'
    preservation_version = 0
```

Stats Graph

Converts stat values extracted from statmetric messages (see *Stat Accumulator Input*) to circular buffer data and periodically emits messages containing this data to be graphed by a DashboardOutput. Note that this filter expects the stats data to be available in the message fields, so the StatAccumInput *must* be configured with *emit_in_fields* set to true for this filter to work correctly.

Config:

- **title (string, optional, default "Stats"):** Title for the graph output generated by this filter.
- **rows (uint, optional, default 300):** The number of rows to store in our circular buffer. Each row represents one time interval.
- **sec_per_row (uint, optional, default 1):** The number of seconds in each circular buffer time interval.
- **stats (string):** Space separated list of stat names. Each specified stat will be expected to be found in the fields of the received statmetric messages, and will be extracted and inserted into its own column in the accumulated circular buffer.
- **stat_labels (string):** Space separated list of header label names to use for the extracted stats. Must be in the same order as the specified stats. Any label longer than 15 characters will be truncated.
- **anomaly_config (string, optional):** Anomaly detection configuration, see *Anomaly Detection Module*.
- **preservation_version (uint, optional, default 0):** If *preserve_data = true* is set in the SandboxFilter configuration, then this value should be incremented every time any edits are made to your *rows*, *sec_per_row*, *stats*, or *stat_labels* values, or else Heka will fail to start because the preserved data will no longer match the filter's data structure.
- **stat_aggregation (string, optional, default "sum"):**

  Controls how the column data is aggregated when combining multiple circular buffers. "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. "none" - No aggregation will be performed the column.
- **stat_unit (string, optional, default "count"):** 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).

Example Heka Configuration

```
[stat-graph]
type = "SandboxFilter"
filename = "lua_filters/stat_graph.lua"
ticker_interval = 10
preserve_data = true
message_matcher = "Type == 'heka.statmetric'"

  [stat-graph.config]
  title = "Hits and Misses"
  rows = 1440
  stat_aggregation = "none"
  stat_unit = "count"
  sec_per_row = 10
  stats = "stats.counters.hits.count stats.counters.misses.count"
```

```
  stat_labels = "hits misses"
  anomaly_config = 'roc("Hits and Misses", 1, 15, 0, 1.5, true, false) roc("Hits and Misses", 2, 15,
  preservation_version = 0
```

Unique Items

Counts the number of unique items per day e.g. active daily users by uid.

Config:

- **message_variable (string, required)** The Heka message variable containing the item to be counted.
- **title (string, optional, default "Estimated Unique Daily *message_variable*")** The graph title for the cbuf output.
- **enable_delta (bool, optional, default false)** Specifies whether or not this plugin should generate cbuf deltas. Deltas should be enabled when sharding is used; see: *Circular Buffer Delta Aggregator*.
- **preservation_version (uint, optional, default 0)** If *preserve_data = true* is set in the SandboxFilter configuration, then this value should be incremented every time the *enable_delta* configuration is changed to prevent the plugin from failing to start during data restoration.

Example Heka Configuration

```
[FxaActiveDailyUsers]
type = "SandboxFilter"
filename = "lua_filters/unique_items.lua"
ticker_interval = 60
preserve_data = true
message_matcher = "Logger == 'FxaAuth' && Type == 'request.summary' && Fields[path] == '/v1/certifica

    [FxaActiveDailyUsers.config]
    message_variable = "Fields[uid]"
    title = "Estimated Active Daily Users"
    preservation_version = 0
```

2.14.7 Sandbox Encoder

Plugin Name: **SandboxEncoder**

The SandboxEncoder provides an isolated execution environment for converting messages into binary data without the need to recompile Heka. See *Sandbox*. Config:

- *Common Sandbox Parameters*

Example

```
[custom_json_encoder]
type = "SandboxEncoder"
filename = "path/to/custom_json_encoder.lua"

    [custom_json_encoder.config]
    msg_fields = ["field1", "field2"]
```

Available Sandbox Encoders

Alert Encoder

Produces more human readable alert messages.

Config:

<none>

Example Heka Configuration

```
[FxaAlert]
type = "SmtpOutput"
message_matcher = "Type == 'heka.sandbox-output' && Fields[payload_type] == 'alert' && Logger =~ /^Fx
send_from = "heka@example.com"
send_to = ["alert@example.com"]
auth = "Plain"
user = "test"
password = "testpw"
host = "localhost:25"
encoder = "AlertEncoder"

[AlertEncoder]
type = "SandboxEncoder"
filename = "lua_encoders/alert.lua"
```

Example Output

Timestamp 2014-05-14T14:20:18Z

Hostname ip-10-226-204-51

Plugin FxaBrowserIdHTTPStatus

Alert HTTP Status - algorithm: roc col: 1 msg: detected anomaly, standard deviation exceeds 1.5

CBUF Librato Encoder

Extracts data from SandboxFilter circular buffer output messages and uses it to generate time series JSON structures that will be accepted by Librato's POST API. It will keep track of the last time it's seen a particular message, keyed by filter name and output name. The first time it sees a new message, it will send data from all of the rows except the last one, which is possibly incomplete. For subsequent messages, the encoder will automatically extract data from all of the rows that have elapsed since the last message was received.

The SandboxEncoder *preserve_data* setting should be set to true when using this encoder, or else the list of received messages will be lost whenever Heka is restarted, possibly causing the same data rows to be sent to Librato multiple times.

Config:

- **message_key (string, optional, default "%{Logger}:%{payload_name}")** String to use as the key to differentiate separate cbuf messages from each other. Supports *message field interpolation*.

Example Heka Configuration

```
[cbuf_librato_encoder]
type = "SandboxEncoder"
filename = "lua_encoders/cbuf_librato.lua"
preserve_data = true
```

```
  [cbuf_librato_encoder.config]
  message_key = "%{Logger}:%{Hostname}:%{payload_name}"

[librato]
type = "HttpOutput"
message_matcher = "Type == 'heka.sandbox-output && Fields[payload_type] == 'cbuf'"
encoder = "cbuf_librato_encoder"
address = "https://metrics-api.librato.com/v1/metrics"
username = "username@example.com"
password = "SECRET"
    [librato.headers]
    Content-Type = ["application/json"]
```

Example Output

```
{"gauges":[{"value":12,"measure_time":1410824950,"name":"HTTP_200","source":"thor"},{"value":1,"measu
```

ESPayloadEncoder

Prepends ElasticSearch BulkAPI index JSON to a message payload.

Config:

- **index (string, optional, default "heka-%{%Y.%m.%d}")** String to use as the *_index* key's value in the generated JSON. Supports field interpolation as described below.
- **type_name (string, optional, default "message")** String to use as the *_type* key's value in the generated JSON. Supports field interpolation as described below.
- **id (string, optional)** String to use as the *_id* key's value in the generated JSON. Supports field interpolation as described below.
- **es_index_from_timestamp (boolean, optional)** If true, then any time interpolation (often used to generate the ElasticSeach index) will use the timestamp from the processed message rather than the system time.

Field interpolation:

All of the string config settings listed above support *message field interpolation*.

Example Heka Configuration

```
[es_payload]
type = "SandboxEncoder"
filename = "lua_encoders/es_payload.lua"
    [es_payload.config]
    es_index_from_timestamp = true
    index = "%{Logger}-%{%Y.%m.%d}"
    type_name = "%{Type}-%{Hostname}"

[ElasticSearchOutput]
message_matcher = "Type == 'mytype'"
encoder = "es_payload"
```

Example Output

```
{"index":{"_index":"mylogger-2014.06.05","_type":"mytype-host.domain.com"}}
{"json":"data","extracted":"from","message":"payload"}
```

Schema Carbon Line Encoder

Converts full Heka message contents to line protocol for Carbon Plaintext API Iterates through all of the dynamic fields to add as points (series), skipping any fields explicitly omitted using the *skip_fields* config option. All dynamic fields in the Heka message are converted to separate points separated by newlines that are submitted to Carbon.

Config:

- **name_prefix (string, optional, default nil)** String to use as the *name* key's prefix value in the generated line. Supports *message field interpolation*. *%{fieldname}*. Any *fieldname* values of "Type", "Payload", "Hostname", "Pid", "Logger", "Severity", or "EnvVersion" will be extracted from the the base message schema, any other values will be assumed to refer to a dynamic message field. Only the first value of the first instance of a dynamic message field can be used for name name interpolation. If the dynamic field doesn't exist, the uninterpolated value will be left in the name. Note that it is not possible to interpolate either the "Timestamp" or the "Uuid" message fields into the name, those values will be interpreted as referring to dynamic message fields.
- **name_prefix_delimiter (string, optional, default ".")** String to use as the delimiter between the name_prefix and the field name. This defaults to a "." to use Graphite naming convention.
- **skip_fields (string, optional, default nil)** Space delimited set of fields that should *not* be included in the Carbon records being generated. Any fieldname values of "Type", "Payload", "Hostname", "Pid", "Logger", "Severity", or "EnvVersion" will be assumed to refer to the corresponding field from the base message schema. Any other values will be assumed to refer to a dynamic message field. The magic value "**all_base**" can be used to exclude base fields from being mapped to the event altogether.
- **source_value_field (string, optional, default nil)** If the desired behavior of this encoder is to extract one field from the Heka message and feed it as a single line to Carbon, then use this option to define which field to find the value from. Make sure to set the name_prefix value to use fields from the message with field interpolation so the full metric path in Graphite is populated. When this option is present, no other fields besides this one will be sent to Carbon whatsoever.

Example Heka Configuration

```
[LinuxStatsDecoder]
type = "MultiDecoder"
subs = ["LoadAvgDecoder", "AddStaticFields"]
cascade_strategy = "all"
log_sub_errors = false

[LoadAvgPoller]
type = "FilePollingInput"
ticker_interval = 5
file_path = "/proc/loadavg"
decoder = "LinuxStatsDecoder"

[LoadAvgDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/linux_loadavg.lua"

[AddStaticFields]
type = "ScribbleDecoder"

    [AddStaticFields.message_fields]
    Environment = "dev"

[CarbonLineEncoder]
type = "SandboxEncoder"
filename = "lua_encoders/schema_carbon_line.lua"
```

```
    [CarbonLineEncoder.config]
    name_prefix = "%{Environment}.%{Hostname}.%{Type}"
    skip_fields = "**all_base** FilePath NumProcesses Environment TickerInterval"

[CarbonOutput]
type = "TcpOutput"
message_matcher = "Type =~ /stats.*/"
encoder = "CarbonLineEncoder"
address = "127.0.0.1:2003"
```

Example Output

```
dev.myhost.stats.loadavg.1MinAvg 0.12 1434932023
dev.myhost.stats.loadavg.15MinAvg 0.18 1434932023
dev.myhost.stats.loadavg.5MinAvg 0.11 1434932023
```

Schema InfluxDB Encoder

Converts full Heka message contents to JSON for InfluxDB HTTP API. Includes all standard message fields and iterates through all of the dynamically specified fields, skipping any bytes fields or any fields explicitly omitted using the *skip_fields* config option.

Note: This encoder is intended for use with InfluxDB versions *prior* to 0.9. If you're working with InfluxDB v0.9 or greater, you'll want to use the *Schema InfluxDB Write Encoder* instead.

Config:

- **series (string, optional, default "series")** String to use as the *series* key's value in the generated JSON. Supports interpolation of field values from the processed message, using *%{fieldname}*. Any *fieldname* values of "Type", "Payload", "Hostname", "Pid", "Logger", "Severity", or "EnvVersion" will be extracted from the the base message schema, any other values will be assumed to refer to a dynamic message field. Only the first value of the first instance of a dynamic message field can be used for series name interpolation. If the dynamic field doesn't exist, the uninterpolated value will be left in the series name. Note that it is not possible to interpolate either the "Timestamp" or the "Uuid" message fields into the series name, those values will be interpreted as referring to dynamic message fields.
- **skip_fields (string, optional, default "")** Space delimited set of fields that should *not* be included in the InfluxDB records being generated. Any fieldname values of "Type", "Payload", "Hostname", "Pid", "Logger", "Severity", or "EnvVersion" will be assumed to refer to the corresponding field from the base message schema. Any other values will be assumed to refer to a dynamic message field.
- **multi_series (boolean, optional, default false)** Instead of submitting all fields to InfluxDB as attributes of a single series, submit a series for each field that sets a "value" attribute to the value of the field. This also sets the name attribute to the series value with the field name appended to it by a ".". This is the recommended by InfluxDB for v0.9 onwards as it is found to provide better performance when querying and aggregating across multiple series.
- **exclude_base_fields (boolean, optional, default false)** Don't send the base fields to InfluxDB. This saves storage space by not including base fields that are mostly redundant and unused data. If skip_fields includes base fields, this overrides it and will only be relevant for skipping dynamic fields.

Example Heka Configuration

```
[influxdb]
type = "SandboxEncoder"
filename = "lua_encoders/schema_influx.lua"
```

```
    [influxdb.config]
    series = "heka.%{Logger}"
    skip_fields = "Pid EnvVersion"

[InfluxOutput]
message_matcher = "Type == 'influxdb'"
encoder = "influxdb"
type = "HttpOutput"
address = "http://influxdbserver.example.com:8086/db/databasename/series"
username = "influx_username"
password = "influx_password"
```

Example Output

```
[{"points":[[1.409378221e+21,"log","test","systemName","TcpInput",5,1,"test"]],"name":"heka.MyLogger"
```

Schema InfluxDB Line Encoder

Converts full Heka message contents to line protocol for InfluxDB HTTP write API (new in InfluxDB v0.9.0). Optionally includes all standard message fields as tags or fields and iterates through all of the dynamic fields to add as points (series), skipping any fields explicitly omitted using the *skip_fields* config option. It can also map any Heka message fields as tags in the request sent to the InfluxDB write API, using the *tag_fields* config option. All dynamic fields in the Heka message are converted to separate points separated by newlines that are submitted to InfluxDB.

Note: This encoder is intended for use with InfluxDB versions 0.9 or greater. If you're working with InfluxDB versions prior to 0.9, you'll want to use the *Schema InfluxDB Encoder* instead.

Config:

- **decimal_precision (string, optional, default "6")** String that is used in the string.format function to define the number of digits printed after the decimal in number values. The string formatting of numbers is forced to print with floating points because InfluxDB will reject values that change from integers to floats and vice-versa. By forcing all numbers to floats, we ensure that InfluxDB will always accept our numerical values, regardless of the initial format.
- **name_prefix (string, optional, default nil)** String to use as the *name* key's prefix value in the generated line. Supports *message field interpolation*. *%{fieldname}*. Any *fieldname* values of "Type", "Payload", "Hostname", "Pid", "Logger", "Severity", or "EnvVersion" will be extracted from the the base message schema, any other values will be assumed to refer to a dynamic message field. Only the first value of the first instance of a dynamic message field can be used for name name interpolation. If the dynamic field doesn't exist, the uninterpolated value will be left in the name. Note that it is not possible to interpolate either the "Timestamp" or the "Uuid" message fields into the name, those values will be interpreted as referring to dynamic message fields.
- **name_prefix_delimiter (string, optional, default nil)** String to use as the delimiter between the name_prefix and the field name. This defaults to a blank string but can be anything else instead (such as "." to use Graphite-like naming).
- **skip_fields (string, optional, default nil)** Space delimited set of fields that should *not* be included in the InfluxDB measurements being generated. Any *fieldname* values of "Type", "Payload", "Hostname", "Pid", "Logger", "Severity", or "EnvVersion" will be assumed to refer to the corresponding field from the base message schema. Any other values will be assumed to refer to a dynamic message field. The magic value "**all_base**" can be used to exclude base fields from being mapped to the event altogether (useful if you don't want to use tags and embed them in the name_prefix instead).

- **source_value_field (string, optional, default nil)** If the desired behavior of this encoder is to extract one field from the Heka message and feed it as a single line to InfluxDB, then use this option to define which field to find the value from. Be careful to set the name_prefix field if this option is present or no measurement name will be present when trying to send to InfluxDB. When this option is present, no other fields besides this one will be sent to InfluxDB as a measurement whatsoever.
- **tag_fields (string, optional, default "all_base")** Take fields defined and add them as tags of the measurement(s) sent to InfluxDB for the message. The magic values "**all**" and "**all_base**" are used to map all fields (including taggable base fields) to tags and only base fields to tags, respectively. If those magic values aren't used, then only those fields defined will map to tags of the measurement sent to InfluxDB. The tag_fields values are independent of the skip_fields values and have no affect on each other. You can skip fields from being sent to InfluxDB as measurements, but still include them as tags.
- **timestamp_precision (string, optional, default "ms")** Specify the timestamp precision that you want the event sent with. The default is to use milliseconds by dividing the Heka message timestamp by 1e6, but this math can be altered by specifying one of the precision values supported by the InfluxDB write API (ms, s, m, h). Other precisions supported by InfluxDB of n and u are not yet supported.
- **value_field_key (string, optional, default "value")** This defines the name of the InfluxDB measurement. We default this to "value" to match the examples in the InfluxDB documentation, but you can replace that with anything else that you prefer.

Example Heka Configuration

```
[LoadAvgPoller]
type = "FilePollingInput"
ticker_interval = 5
file_path = "/proc/loadavg"
decoder = "LinuxStatsDecoder"

[LoadAvgDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/linux_loadavg.lua"

[LinuxStatsDecoder]
type = "MultiDecoder"
subs = ["LoadAvgDecoder", "AddStaticFields"]
cascade_strategy = "all"
log_sub_errors = false

[AddStaticFields]
type = "ScribbleDecoder"

    [AddStaticFields.message_fields]
    Environment = "dev"

[InfluxdbLineEncoder]
type = "SandboxEncoder"
filename = "lua_encoders/schema_influx_line.lua"

    [InfluxdbLineEncoder.config]
    skip_fields = "**all_base** FilePath NumProcesses Environment TickerInterval"
    tag_fields = "Hostname Environment"
    timestamp_precision= "s"

[InfluxdbOutput]
type = "HttpOutput"
message_matcher = "Type =~ /stats.*/"
encoder = "InfluxdbLineEncoder"
```

```
address = "http://influxdbserver.example.com:8086/write?db=mydb&rp=mypolicy&precision=s"
username = "influx_username"
password = "influx_password"
```

Example Output

```
5MinAvg,Hostname=myhost,Environment=dev value=0.110000 1434932024
1MinAvg,Hostname=myhost,Environment=dev value=0.110000 1434932024
15MinAvg,Hostname=myhost,Environment=dev value=0.170000 1434932024
```

Statmetric Influx Encoder

Extracts data from message fields in *heka.statmetric* messages generated by a *Stat Accumulator Input* and generates JSON suitable for use with InfluxDB's HTTP API. StatAccumInput must be configured with *emit_in_fields = true* for this encoder to work correctly.

Config:

<none>

Example Heka Configuration

```
[statmetric-influx-encoder]
type = "SandboxEncoder"
filename = "lua_encoders/statmetric_influx.lua"

[influx]
type = "HttpOutput"
message_matcher = "Type == 'heka.statmetric'"
address = "http://myinfluxserver.example.com:8086/db/stats/series"
encoder = "statmetric-influx-encoder"
username = "influx_username"
password = "influx_password"
```

Example Output

```
[{"points":[[1408404848,78271]],"name":"stats.counters.000000.rate","columns":["time","value"]},{"poi
```

2.14.8 Sandbox Output

New in version 0.9.

Plugin Name: **SandboxOutput**

The SandboxOutput provides a flexible execution environment for data encoding and transmission without the need to recompile Heka. See *Sandbox*. Config:

- The common output configuration parameter 'encoder' is ignored since all data transformation should happen in the plugin.
- *Common Sandbox Parameters*
- **timer_event_on_shutdown (bool):** True if the sandbox should have its timer_event function called on shutdown.

Example

```
[SandboxFileOutput]
type = "SandboxOutput"
filename = "fileoutput.lua"

[SandboxFileOutput.config]
path = "mylog.txt"
```

Available Sandbox Outputs

- none

Externally Available Sandbox Outputs

- Example Redshift Output

2.14.9 Available Sandbox Modules

Alert Module

API

Stores the last alert time in the global *_LAST_ALERT* so alert throttling will persist between restarts.

queue(ns, msg) Queue an alert message to be sent.

Arguments

- ns (int64) current time in nanoseconds since the UNIX epoch.
- msg (string) alert payload.

Return

- true if the message is queued, false if it would be throttled.

send(ns, msg) Send an alert message.

Arguments

- ns (int64) current time in nanoseconds since the UNIX epoch.
- msg (string) alert payload.

Return

- true if the message is sent, false if it is throttled.

send_queue(ns) Sends all queued alert message as a single message.

Arguments

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

Return

- true if the queued messages are sent, false if they are throttled.

set_throttle(ns_duration) Sets the minimum duration between alert event outputs.

Arguments

- ns_duration (int64) minimum duration in nanoseconds between alerts.

Return

- none

throttled(ns) Test to see if sending an alert at this time would be throttled.

Arguments

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

Return

- true if a message would be throttled, false if it would be sent.

Note: Use a zero timestamp to override message throttling.

Annotation Module

API

add(name, ns, col, stext, text) Create an annotation in the global *_ANNOTATIONS* table.

Arguments

- name (string) circular buffer payload name.
- ns (int64) current time in nanoseconds since the UNIX epoch.
- col (uint) circular buffer column to annotate.
- stext (string) short text to display on the graph.
- text (string) long text to display in the rollover.

Return

- none

create(ns, col, stext, text) Helper function to create an annotation table but not add it to the global list of annotations.

Arguments

- ns (int64) current time in nanoseconds since the UNIX epoch.
- col (uint) circular buffer column to annotate.
- stext (string) short text to display on the graph.
- text (string) long text to display in the rollover.

Return

- annotation table

concat(name, annotations) Concatenates an array of annotation tables to the specified key in the global _ANNOTATIONS table.

Arguments

- name (string) circular buffer payload name.
- annotations (array) annotation tables.

Return

- none

prune(name, ns)

Arguments

- name (string) circular buffer payload name.
- ns (int64) current time in nanoseconds since the UNIX epoch.

Return

- The json encoded list of annotations.

remove(name) Entirely remove the payload name from the global *_ANNOTATIONS* table.

Arguments

- name (string) circular buffer payload name.

Return

- none

set_prune(name, ns_duration)

Arguments

- name (string) circular buffer payload name.
- ns_duration (int64) time in nanoseconds the annotation should remain in the list.

Return

- none

Anomaly Detection Module

API

parse_config(anomaly_config) Parses the anomaly_config into a Lua table. If the configuration is invalid an error is thrown.

Arguments

- anomaly_config (string or nil)

The configuration can specify any number of algorithm function calls (space delimited if desired, but they will also work back to back with no delimiter). This allows for analysis of multiple graphs, columns, and even specification of multiple algorithms per column.

Rate of change test

Only use this test on data with a normal (Gaussian http://en.wikipedia.org/wiki/Normal_distribution) distribution. It identifies rapid changes (spikes) in the data (increasing and decreasing) but ignores cyclic data that has a more gradual rise and fall. It is typically used for something like HTTP 200 status code analysis to detect a sudden increase/decrease in web traffic.

roc("payload_name", col, win, hwin, sd, loss_of_data, start_of_data)

- **payload_name (string)** Quoted string containing the *payload_name* value used in the inject_payload function call. If the payload name contains a double quote it should be escaped as two double quotes in a row.
- **col (uint)** The circular buffer column to perform the analysis on.

- **win (uint)** The number of intervals in an analysis window.
- **hwin (uint)** The number of intervals in the historical analysis window (0 uses the full history). Must be greater than or equal to 'win'.
- **sd (double)** The standard deviation threshold to trigger the anomaly.
- **loss_of_data (bool)** Alert if data stops.
- **start_of_data (bool)** Alert if data starts.

e.g. roc("Output1", 1, 15, 0, 2, true, false)

Mann-Whitney-Wilcoxon test http://en.wikipedia.org/wiki/Mann-Whitney

Parametric

Only use this test on data with a normal (Gaussian http://en.wikipedia.org/wiki/Normal_distribution) distribution. It identifies more gradual changes in the data (increasing, decreasing, or any). It is typically used with something like server memory analysis where the values are more stable and gradual changes are interesting (e.g., memory leak).

mww("payload_name", col, win, nwin, pvalue, trend)

- **payload_name (string)** Quoted string containing the *payload_name* value used in the inject_payload function call. If the payload name contains a double quote it should be escaped as two double quotes in a row.
- **col (uint)** The circular buffer column to perform the analysis on.
- **win (uint)** The number of intervals in an analysis window (should be at least 20).
- **nwin (uint)** The number of analysis windows to compare.
- **pvalue (double)** The pvalue threshold to trigger the prediction. http://en.wikipedia.org/wiki/P_value
- **trend (string)** (decreasing|increasing|any)

e.g. mww("Output1", 2, 60, 10, 0.0001, decreasing)

Non-parametric

This test can be used on data with a normal (Gaussian http://en.wikipedia.org/wiki/Normal_distribution) or non-normal (nonparametric http://en.wikipedia.org/wiki/Nonparametric_statistics) distribution. It identifies overlap/similarities between two data sets. It is typically used for something like detecting an increase in HTTP 500 status code errors.

mww_nonparametric("payload_name", col, win, nwin, pstat)

- **payload_name (string)** Quoted string containing the *payload_name* value used in the inject_payload function call. If the payload name contains a double quote it should be escaped as two double quotes in a row.
- **col (uint)** The circular buffer column to perform the analysis on.
- **win (uint)** The number of intervals in an analysis window.
- **nwin (uint)** The number of analysis windows to compare.
- **pstat (double)** Value between 0 and 1. Anything above 0.5 is an increasing trend anything below 0.5 is a decreasing trend. http://en.wikipedia.org/wiki/Mann-Whitney#.CF.81_statistic

e.g. mww_nonparametric("Output1", 2, 15, 10, 0.55)

Return Configuration table if parsing was successful or nil, if nil was passed in.

detect(ns, name, cbuf, anomaly_config) Detects anomalies in the circular buffer data returning any error messages for alert generation and array of annotations for the graph.

Arguments

- ns (int64) current time in nanoseconds since the UNIX epoch. It used to advance the circular buffer if necessary (i.e., if no data is being received). The anomaly detection is always performed on the newest data (ignoring the current interval since it is incomplete).
- name (string) circular buffer payload name
- cbuf (userdata) circular buffer
- anomaly_config (table) returned from the parse() method

Return

- string if an anomaly was detected, otherwise nil.
- array of annotation tables

Message Interpolation Module

New in version 0.9.

API

interpolate_from_msg(value, secs)

Interpolates values from the currently processed message into the provided string value. A *%{}* enclosed field name will be replaced by the field value from the current message. Supported default field names are "Type", "Hostname", "Pid", "UUID", "Logger", "EnvVersion", and "Severity". Any other values will be checked against the defined dynamic message fields. If no field matches, then a C strftime (on non-Windows platforms) or C89 strftime (on Windows) time substitution will be attempted. The time used for time substitution will be the seconds-from-epoch timestamp passed in as the *secs* argument, if provided. If *secs* is nil, local system time is used. Note that the message timestamp is *not* automatically used; if you want to use the message timestamp for time substitutions, then you need to manually extract it and convert it from nanoseconds to seconds (i.e. divide by 1e9).

Arguments

- **value (string)** String into which message values should be interpolated.
- **secs (number or nil)** Timestamp (in seconds since epoch) to use for time substitutions. If nil, system time will be used.

Return

- Original string value with any interpolated message values.

ElasticSearch Module

API

bulkapi_index_json(index, type_name, id, ns)

Returns a simple JSON 'index' structure satisfying the ElasticSearch BulkAPI

Arguments

- **index (string or nil)** String to use as the *_index* key's value in the generated JSON, or nil to omit the key. Supports field interpolation as described below.
- **type_name (string or nil)** String to use as the *_type* key's value in the generated JSON, or nil to omit the key. Supports field interpolation as described below.
- **id (string or nil)** String to use as the *_id* key' value in the generated JSON, or nil to omit the key. Supports field interpolation as described below.
- **ns (number or nil)** Nanosecond timestamp to use for any strftime field interpolation into the above fields. Current system time will be used if nil.

Field interpolation

All of the string arguments listed above support *message field interpolation*.

Return

- JSON string suitable for use as ElasticSearch BulkAPI index directive.

Field Utilities Module

Module contains utility functions for setting up fields for various purposes.

API

field_map(fields_str or nil) Returns a table of fields that match the space delimited input string of fields. This can be used to provide input to other functions such as a list of fields to skip or use for tags.

Arguments

- **fields_str (string or nil)** Space delimited list of fields. If this is empty or nil, all base fields will be returned.

Return Table with the fields found in the space delimited input string, boolean indicating all base fields are to be used, boolean indicating all fields are to be used.

message_timestamp(timestamp_precision) Returns the timestamp value after dividing it by a constant after mapping it from a precision value to convert it from the heka default precision of ns to a lower precision to work better with output endpoints.

Arguments

- **timestamp_precision (string or nil)** String that can have a value of "ms", "s", "m" or "h".

Return The timestamp value after converting it from ns to the indicated timestamp_precision.

used_base_fields(skip_fields) Returns a table of base fields that are not found in the input table. This is useful to provide a lookup table that is used to decide whether or not a field should be included in an output by performing a simple lookup against it.

Arguments

- **skip_fields (table)** Table of fields to be skipped from use.

Return A table of base fields that are not found in the input table.

Graphite Module

New in version 0.10.

Module contains various utility functions for metrics stored Graphite.

Currently module focuses on generators of metrics which can be later passed to Graphite.

API

count_rate(bucket, count, ticker_interval, now_sec) Generates string with count and rate metric for graphite.

Arguments

- bucket - node name in which metric will be stored.
- count - value of count.
- ticker_interval - base interval for calculation of rate.
- now_sec - timestamp (float) for metric.

Return String with count and rate metric for *stats.counters.<bucket>.count* and *stats.counters.<bucket>.rate* bucket with time given in <now_sec>.

multi_counts_rates(counts, ticker_interval, now_sec) Generates a multiline graphite count metric with their rates.

Arguments

- **counts - table, indices will be mapped to buckets and values to their** specific counts e.g {'bucket1': 1, 'bucket2':2}
- ticker_interval - base interval for calculation of rate.
- now_sec - timestamp (float) for metric.

Return String with multiple counts and rates returned via *return_count* function.

timeseries_metrics(bucket, times, ticker_interval, percent_thresh, now_sec) Generates string with metrics for given timeseries data to pass it to graphite.

Arguments

- bucket - node name in which metric will be stored.
- times - a table with times (float values).
- ticker_interval - base interval for calculation of rate.
- percent_treshold - base treshould for percentiles.
- now_sec - timestamp (float) for metric.

Return

Returns multiline graphite string with following metrics:

- stats.timers.<bucket>.count
- stats.timers.<bucket>.rate
- stats.timers.<bucket>.min
- stats.timers.<bucket>.max
- stats.timers.<bucket>.mean

- stats.timers.<bucket>.mean_percentile
- stats.timers.<bucket>.upper_percentile

multi_timeseries_metrics(timers, ticker_interval, percent_thresh, now_sec) Returns multline string with stats calculated for timeseries in their respective buckets

Arguments

- timers - tables with bucket names and tables of times inside e.g {'bucket': [1,2,3,4]}
- ticker_interval - base interval for calculation of rate.
- percent_treshold - base treshould for percentiles.
- now_sec - timestamp (float) for metric.

Return String with corresponding metric series and their respective buckets defined in timers table. Metrics are the same as for the *timeseries_metrics_function*.

function ns_to_sec(ns) Converts nanoseconds into seconds.

Arguments

- ns - nanoseconds

Return Seconds in float value.

Time Series Line Protocol Module

2.14.10 Lua Parsing Expression Grammars (LPeg)

Best practices (using Lpeg in the sandbox)

1. Read the LPeg reference
2. The 're' module is now available in the sandbox but the best practice is to use the LPeg syntax whenever possible (i.e., in Lua code). Why?
   - Consistency and readability of a single syntax.
   - Promotes more modular grammars.
   - Is easier to comment.
3. Do not use parentheses around function calls that take a single string argument.

```
-- prefer
lpeg.P"Literal"

-- instead of
lpeg.P("Literal")
```

4. When writing sub-grammars with an ordered choice (+) place each choice on its own line; this make it easier to pick out the alternates. Also, if possible order them from most frequent to least frequent use.

```
local date_month = lpeg.P"0" * lpeg.R"19"
                   + "1" * lpeg.R"02"

-- The exception: when grouping alternates together in a higher level grammar.

local log_grammar = (rfc3339 + iso8601) * log_severity * log_message
```

5. Use the locale patterns when matching standard character classes.

```
-- prefer
lpeg.digit

-- instead of
lpeg.R"09".
```

6. If a literal occurs within an expression avoid wrapping it in a function.

```
-- prefer
lpeg.digit * "Test"

-- instead of
lpeg.digit * lpeg.P"Test"
```

7. When creating a parser from an RFC standard mirror the ABNF grammar that is provided.
8. If creating a grammar that would also be useful to others, please consider contributing it back to the project, thanks.
9. Use the grammar tester http://lpeg.trink.com.

2.14.11 Sandbox Manager Filter

Plugin Name: **SandboxManagerFilter**

The SandboxManagerFilter provides dynamic control (start/stop) of sandbox filters in a secure manner without stopping the Heka daemon. Commands are sent to a SandboxManagerFilter 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.

SandboxManagerFilter Settings

- *Common Filter 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.
- **module_directory (string):** The directory where 'require' will attempt to load the external Lua modules from. Defaults to ${SHARE_DIR}/lua_modules.
- **max_filters (uint):** The maximum number of filters this manager can run.

New in version 0.5.

- **memory_limit (uint):** The number of bytes managed sandboxes are allowed to consume before being terminated (default 8MiB).
- **instruction_limit (uint):** The number of instructions managed sandboxes are allowed to execute during the process_message/timer_event functions before being terminated (default 1M).
- **output_limit (uint):** The number of bytes managed sandbox output buffers can hold before being terminated (default 63KiB). Warning: messages exceeding 64KiB will generate an error and be discarded by the standard output plugins (File, TCP, UDP) since they exceed the maximum message size.

Example

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

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 *Sandbox Filter*

Stopping a SandboxFilter

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

heka-sbmgr

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

Command Line Options

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

Configuration Variables

- ip_address (string): IP address of the Heka server.
- use_tls (bool): Specifies whether or not SSL/TLS encryption should be used for the TCP connections. Defaults to false.
- **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.
- tls (TlsConfig): A sub-section that specifies the settings to be used for any SSL/TLS encryption. This will only have any impact if *use_tls* is set to true. See *Configuring TLS*.

Example

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

[tls]
    cert_file = "heka.crt"
    key_file = "heka.key"
    client_auth = "NoClientCert"
    prefer_server_ciphers = true
    min_version = "TLS11"
```

heka-sbmgrload

Heka 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

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

Configuration Variables (same as heka-sbmgr)

2.14.12 Tutorial - How to use the dynamic sandboxes

SandboxManager/Heka Setup

1. Configure the SandboxManagerFilter.

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"
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"
parser_type = "message.proto"
decoder = "ProtobufDecoder"
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
```

SandboxFilter Setup

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

```
require "circular_buffer"

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)
    inject_payload("cbuf", "", data)
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
filename = ""
preserve_data = false
```

3. Load the filter using sbmgr.

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

If you are running the *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. During the unload/load process some data can be missed and gaps will be created. In the future we hope to remedy this but for now it is a limitation of the dynamic sandbox.

4. Unload the filter using sbmgr.

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

2.14.13 Sandbox Development

Decoders

Since decoders cannot be dynamically loaded and they stop Heka processing on fatal errors they must be developed outside of a production enviroment. Most Lua decoders are LPeg based as it is the best way to parse and transform data within the sandbox. The other alternatives are the built-in Lua pattern matcher or the JSON parser with a manual transformation.

1. Procure some sample data to be used as test input.

```
timestamp=time_t key1=data1 key2=data2
```

2. Configure a simple LogstreamerInput to deliver the data to your decoder.

```
[LogstreamerInput]
log_directory = "."
file_match = 'data\.log'
decoder = "SandboxDecoder"
```

3. Configure your test decoder.

```
[SandboxDecoder]
filename = "decoder.lua"
```

4. Configure the DasboardOutput for visibility into the decoder (performance, memory usage, messages processed/failed, etc.)

```
[DashboardOutput]
address = "127.0.0.1:4352"
ticker_interval = 10
working_directory = "dashboard"
static_directory = "/usr/share/heka/dasher"
```

5. Configure a LogOutput to display the generated messages.

```
[LogOutput]
message_matcher = "TRUE"
```

6. **Build the decoder.** The decoder will receive a message from an input plugin. The input may have set some additional message headers but the 'Payload' header contains the data for the decoder. The decoder can access the payload using read_message("Payload"). The payload can be used to construct an entirely new message, multiple messages or modify any part of the existing message (see inject_message, write_message in the *Lua Sandbox* API). Message headers not modified by the decoder are left intact and in the case of multiple message injections the initial message header values are duplicated for each message.

   (a) **LPeg grammar.** Incrementally build and test your grammar using http://lpeg.trink.com.

(b) **Lua pattern matcher.** Test match expressions using http://www.lua.org/cgi-bin/demo.

(c) **JSON parser.** For data transformation use the LPeg/Lua matcher links above. Something like simple field remapping i.e. *msg.Hostname = json.host* can be verified in the LogOutput.

7. Run Heka with the test configuration.
8. Inspect/verify the messages written by LogOutput.

Filters

Since filters can be dynamically loaded it is recommended you develop them in production with live data.

1. Read *Tutorial - How to use the dynamic sandboxes*

OR

1. If you are developing the filter in conjunction with the decoder you can add it to the test configuration.

```
[SandboxFilter]
filename = "filter.lua"
```

2. Debugging

(a) Watch for a dashboard sandbox termination report. The termination message provides the line number and cause of the failure. These are usually straight forward to correct and commonly caused by a syntax error in the script or invalid assumptions about the data (e.g. *cnt = cnt + read_message("Fields[counter]")* will fail if the counter field doesn't exist or is non-numeric due to a error in the data).

(b) No termination report and the output does not match expectations. These are usually a little harder to debug.

i. Check the Heka dasboard to make sure the router is sending messages to the plugin. If not, verify your message_matcher configuration.

ii. Visually review the the plugin for errors. Are the message field names correct, was the result of the cjson.decode tested, are the output variables actually being assigned to and output/injected, etc.

iii. Add a debug output message with the pertinent information.

```
require "string"
require "table"
local dbg = {}

-- table.insert(dbg, string.format("Entering function x arg1: %s", arg1))
-- table.insert(dbg, "Exiting function x")

inject_payload("txt", "debug", table.concat(dbg, "\n"))
```

i. LAST RESORT: Move the filter out of production, turn on preservation, run the tests, stop Heka, and review the entire preserved state of the filter.

2.14.14 Lua Sandbox Cookbooks

- **Decoders**
  - *JSON Payload Transform*
- **Presentation**

  - *Circular Buffer Graph Annotation (Alerts)*

2.15 Testing Heka

2.15.1 heka-flood

heka-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.

Command Line Options

- -config="flood.toml": Path to heka-flood config file
- -test="default": Name of config file defined test to run

Example:

```
heka-flood -config="/etc/flood.toml" -test="my_test_name"
```

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.
- **message_interval (string):** Duration of time to delay between the sending of each message. Accepts duration values as supported by Go's time.ParseDuration function. Default of 0 means no delay.
- **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.

New in version 0.5.

- **use_tls (bool):** Specifies whether or not SSL/TLS encryption should be used for the TCP connections. Defaults to false.

- **tls (TlsConfig):** A sub-section that specifies the settings to be used for any SSL/TLS encryption. This will only have any impact if *use_tls* is set to true. See *Configuring TLS*.

New in version 0.9.

- **max_message_size (uint32):** The maximum size of the message that will be sent by heka-flood.

New in version 0.10.

- **reconnect_on_error (bool):** Defines if *heka-flood* should try to reconnect with backend after connection error. Exits if reconnect_on_error is set to false. Defaults to false.
- **reconnect_interval (int):** Specifies interval (in seconds) after which *heka-flood* will try to recreate connection with backend. Defaults to 5s.

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
```

2.15.2 heka-inject

New in version 0.5.

heka-inject is a Heka client allowing for the injecting of arbitrary messages into the Heka pipeline. It is capable of generating a message of specified message variables with values. It allows for quickly testing plugins. Inject requires TcpInput with Protobufs encoder availability.

Command Line Options

- -heka: Heka instance to connect
- -hostname: message hostname
- -logger: message logger
- -payload: message payload
- -pid: message pid
- -severity: message severity
- -type: message type

Example:

```
heka-inject -payload="Test message with high severity." -severity=1
```

2.15.3 heka-cat

New in version 0.5.

A command-line utility for counting, viewing, filtering, and extracting Heka protobuf logs.

Command Line Options

- -format="txt": output format [txt|json|heka|count]
- -match="TRUE": message_matcher filter expression
- -offset=0: starting offset for the input file in bytes
- -output="": output filename, defaults to stdout
- -tail=false: don't exit on EOF
- *input filename*

Example:

```
heka-cat -format=count -match="Fields[status] == 404" test.log
```

Output:

```
Input:test.log  Offset:0  Match:Fields[status] == 404  Format:count  Tail:false  Output:
Processed: 1002646, matched: 15660 messages
```

2.16 Configuring TLS

Many input and output plugins that rely on TCP as the underlying transport for network communication also support the use of SSL/TLS encryption for their connections. Typically the TOML configuration for these plugins will support a boolean *use_tls* flag that specifies whether or not encryption should be used, and a *tls* sub-section that specifies the settings to be used for negotiating the TLS connections. If *use_tls* is not set to true, the *tls* section will be ignored.

Modeled after Go's stdlib TLS configuration struct, the same configuration structure is used for both client and server connections, with some of the settings being applicable for a client's configuration, some for a server's, and some for both. In the description of the TLS configuration settings below, each setting is marked as appropriate to client, server, or both as appropriate.

2.16.1 TLS configuration settings

- **server_name (string, client):** Name of the server being requested. Included in the client handshake to support virtual hosting server environments.
- **cert_file (string, both):** Full filesystem path to the certificate file to be presented to the other side of the connection.
- **key_file (string, both):** Full filesystem path to the specified certificate's associated private key file.
- **client_auth (string, server):** Specifies the server's policy for TLS client authentication. Must be one of the following values:
  - NoClientCert
  - RequestClientCert

  - RequireAnyClientCert
  - VerifyClientCertIfGiven
  - RequireAndVerifyClientCert

  Defaults to "NoClientCert".

- **ciphers ([]string, both):** List of cipher suites supported for TLS connections. Earlier suites in the list have priority over those following. Must only contain values from the following selection:
  - RSA_WITH_RC4_128_SHA
  - RSA_WITH_3DES_EDE_CBC_SHA
  - RSA_WITH_AES_128_CBC_SHA
  - RSA_WITH_AES_256_CBC_SHA
  - ECDHE_ECDSA_WITH_RC4_128_SHA
  - ECDHE_ECDSA_WITH_AES_128_CBC_SHA
  - ECDHE_ECDSA_WITH_AES_256_CBC_SHA
  - ECDHE_RSA_WITH_RC4_128_SHA
  - ECDHE_RSA_WITH_3DES_EDE_CBC_SHA
  - ECDHE_RSA_WITH_AES_128_CBC_SHA
  - ECDHE_RSA_WITH_AES_256_CBC_SHA
  - ECDHE_RSA_WITH_AES_128_GCM_SHA256
  - ECDHE_ECDSA_WITH_AES_128_GCM_SHA256

  If omitted, the implementation's default ordering will be used.

- **insecure_skip_verify (bool, client):** If true, TLS client connections will accept any certificate presented by the server and any host name in that certificate. This causes TLS to be susceptible to man-in-the-middle attacks and should only be used for testing. Defaults to false.

- **prefer_server_ciphers (bool, server):** If true, a server will always favor the server's specified cipher suite priority order over that requested by the client. Defaults to true.

- **session_tickets_disabled (bool, server):** If true, session resumption support as specified in RFC 5077 will be disabled.

- **session_ticket_key (string, server):** Used by the TLS server to provide session resumption per RFC 5077. If left empty, it will be filled with random data before the first server handshake.

- **min_version (string, both):** Specifies the mininum acceptable SSL/TLS version. Must be one of the following values:
  - SSL30
  - TLS10
  - TLS11
  - TLS12

  Defaults to SSL30.

- **max_version (string, both):** Specifies the maximum acceptable SSL/TLS version. Must be one of the following values:
  - SSL30

  - TLS10
  - TLS11
  - TLS12

  Defaults to TLS12.

- **client_cafile (string, server):** File for server to authenticate client TLS handshake. Any client certs recieved by server must be chained to a CA found in this PEM file.

  Has no effect when NoClientCert is set.

- **root_cafile (string, client):** File for client to authenticate server TLS handshake. Any server certs recieved by client must be must be chained to a CA found in this PEM file.

2.16.2 Sample TLS configuration

The following is a sample TcpInput configuration showing the use of TLS encryption.

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

    [TcpInput.tls]
    cert_file = "/usr/share/heka/tls/cert.pem"
    key_file = "/usr/share/heka/tls/cert.key"
    client_auth = "RequireAndVerifyClientCert"
    prefer_server_ciphers = true
    min_version = "TLS11"
```

2.17 Configuring Buffering

All filter and output plugins support the use of a disk based message queue. If `use_buffering` is set to true, then the router will deliver messages that match the plugin's *message matcher* to the queue buffer, and the plugin will read from the queue to get messages to process, instead of the handoff happening in the process RAM via Go channels. This improves message delivery reliability and allows plugins to reprocess messages from the queue in cases where upstream servers are down or Heka is recovering from a hard shutdown.

Each queue buffer supports a few configuration settings in addition to any options which the plugin might support. These can be specified in a sub-section of the plugin's TOML configuration section entitled `buffering`.

2.17.1 Buffering configuration settings

- max_file_size (uint64) The maximum size (in bytes) of a single file in the queue buffer. When a message would increase a queue file to greater than this size, the message will be written into a new file instead. Defaults to 128MiB. Value cannot be zero, if zero is specified the default will instead be used.
- max_buffer_size (uint64) Maximum amount of disk space (in bytes) that the entire queue buffer can consume. Defaults to 0, or no limit. The action taken when the maximum buffer size is reached is determined by the `full_action` setting.

- full_action (string) The action Heka will take if the queue buffer grows to larger than the maximum specified by the `max_buffer_size` setting. Must be one of the following values. Defaults to `shutdown`, although specific plugins might override this default with a default of their own:
  - `shutdown`: Heka will stop all processing and attempt a clean shutdown.
  - **`drop`: Heka will drop the current message and will continue to process** future messages.
  - **`block`: Heka will pause message delivery, applying back pressure through** the router to the inputs. Delivery will resume if and when the queue buffer size reduces to below the specified maximum.
- cursor_update_count (uint) A plugin is responsible for notifying the queue buffer when a message has been processed by calling an `UpdateCursor` method on the PluginRunner. Some plugins call this for every message, while others call it only periodically after processing a large batch of messages. This setting specifies how many `UpdateCursor` calls must be made before the cursor location is flushed to disk. Defaults to 1, although specific plugins might override this default with a default of their own. Value cannot be zero, if zero is specified the default will be used instead.

2.17.2 Buffering Default Values

Please note that if you provide a *buffering* subsection for your plugin configuration, it is best to specify *all* of the available settings. In cases where the plugin specifies a non-standard default for one or more of these values, that default will only be applied if you omit the *buffering* subsection altogether. If you specify any of the values, it is expected that you will specify all of the values.

2.17.3 Sample Buffering Configuration

The following is a sample TcpOutput configuration showing the use of buffering.

```
[TcpOutput]
message_matcher = "Type !~ /^heka/"
address = "upstream.example.com:5565"
keep_alive = true
use_buffering = true

    [TcpOutput.buffering]
    max_file_size = 268435456  # 256MiB
    max_buffer_size = 1073741824  # 1GiB
    full_action = "block"
    cursor_update_count = 100
```

2.18 Changelog

```
0.10.0b0 (2015-07-13)
=====================

Backwards Incompatibilities
---------------------------

* Major overhaul of filter and output plugin APIs to support disk buffering
  (#1378).

* Go 1.4 now required for building.
```

```
* Deprecated the read_next_field() sandbox API; it is targeted for removal in
  0.11.0 (#1602).

* Removed unused PipelinePack.Decoded attribute.

* LogOutput will write data to stdout instead of stderr (#1515).

* Using stftime literals for filenames during rotation in FileOutput plugin
  (#1469).

* Implemented stftime format codes in: filenames in FileOutput plugin,
  ESJsonEncoder, ESLogstashV0Encoder, Payload encoder (#1469, #1508).

* The package created by 'make deb' creates an "heka" user and ships an init
  script and a systemd unit.

* The 'make deb' target requires fakeroot and debhelper to be installed.

* SplitterRunner interface now provides a `Done` method that should be called
  whenever the splitter is no longer needed.

* `"Fields"` choice for the `fields` setting renamed to `"DynamicFields"` for
  both ESJsonEncoder and ESLogstashV0Encoder.

Features
--------

* Added support for `write_message` to delete fields when passed nil values.

* Added a timer_event_on_shutdown configuration option for Filter/Output
  sandboxes (#1460).

* Added `replace_dot` setting to StatFilter.

* A protobuf encoding of the current message is now always stored in
  pack.MsgBytes prior to injection into the message router (#265).

* ProtobufEncoder now just copies the pack.MsgBytes into a new byte slice and
  returns that.

* Added Linux CPU Stats Decoder and CPU Stats Filter.

* Centralized common functions used by the Schema Carbon Line Encoder and
  Schema InfluxDB Line Encoder into new ts_line_protocol and field_util modules.

* Added a new sandbox encoder, Schema Carbon Line Encoder, which provides
  support for converting fields in a Heka message to metrics formatted to
  send to Carbon/Graphite.

* Added a new sandbox encoder, Schema InfluxDB Line Encoder, which updates
  compatibility of sending data in Heka message fields to InfluxDB 0.9.0+
  write API. It is required to use this encoder when integrating Heka with an
  InfluxDB 0.9.0+ instance as the API to commit metrics has changed. This
  encoder was also updated to use the line protocol which is now the default
  series format for the write API.

* Added support for abstract Unix domain sockets to the UdpInput.
```

```
* Added support for `can_exit` to inputs (defaults to false, except on
  ProcessDirectoryInput spawned processes where it defaults to true)

* Added `field_mappings` setting to the ESJsonEncoder allowing custom names for
  Heka message fields in ElasticSearch.

* Termination messages are now generated for SandboxFilters that fail
  initialization by a SandboxManagerFilter.

* Added alert_throttle setting to HTTP status graph filter.

* Added DockerEventInput.

* Added splitters to the reports displayed by Heka after SIGUSR1 signal.

* Added `graphite` module with helpers allowing to generate graphite metrics
  for counters and timeseries (#1461).

* Added Basic Auth and API key authentication to HttpListenInput (#1533).

* Added SSL/TLS support to HttpListenInput (#1534)

* Allow to overwrite ContainerName using a Container environment variable from
  within the Docker container for the DockerLogInput. (#1545)

* Allow extracting environment variables from a Docker container to use as
  fields for the DockerLogInput. (#1569)

* Dashboard now shows sandbox plugins in alphabetical order.

* Added 'dynamic_fields' config to ESJsonEncoder for selecting a subset of
  Fields values

Bug Handling
------------

* Fixed visibility of synchronous decoders in reports (#1312).

* Fixed hang on SandboxFilter termination (#1509)

* More ProcessInput/ProcessDirectoryInput retry logic fixes (#1412 & #1418).

* ProcessInput fixed to no longer leak decoder goroutines when reconfigured
  via ProcessDirectoryInput-driven config changes (#1444).

* Check configuration, ticker_interval in Stat Accumulator Input  must be
  greater than 0 (#1474).

* Switch from unmaintained `crowdmob` fork of GoAMZ dependency to `AdRoll`
  (#1458).

* Added ability to reconnect after lost connection in `heka-flood` (#1536).

* Respect ElasticSearch URL path (#1558).

0.9.3 (2015-??-??)
==================
```

```
Bug Handling
------------

* Correctly honor "user-agent" heading config in HttpInput (#1520).

* Removed state from PluginMaker to remove race conditions during plugin
  construction (#1532).

* Get decoder lock before cleaning up decoders during pipeline shutdown to
  avoid race condition panics during exit (#1531).

0.9.2 (2015-04-22)
==================

Bug Handling
------------

* Added bounds check for truncated inputs lines to StatsdInput.

* Fixed ElasticSearch recovery after full queue when `queue_full_action` is
  set to "drop" or "block".

* Fixed TcpOutput recovery after full queue when `queue_full_action` is set to
  "drop" or "block" (#1484).

* Fixed bug where LogstreamerInput wasn't honoring `oldest_duration` setting
  (#1437).

* ElasticSearch payload encoder will ensure there is a newline in the
  end of the payload in order for the bulk API to work correctly (#1457).

* Fix a gzipped file seeking in logstreamer may cause an OOM exception.

* Fixed config parsing typo bug in heka-logstreamer command (#1436).

* Always check for decoder existence when a decoder is specified for an input
  plugin (#1439).

* Check `IsStoppable()` on all input, filter, output plugins if they error at
  startup time, only shut down Heka if false.

* Fixed typo causing panic in AMQPInput when splitter returns an error
  (#1453).

* Handle chunked error responses in http_output.

* ElasticSearchOutput buffering now only retries messages when it's clear that
  the problem related to failed communication with the ElasticSearch server
  (#1401).

* Added handling of incomplete/trailing data for SplitBytes() method in
  SplitterRunner (#1455)

0.9.1 (2015-03-13)
==================

Features
--------
```

```
* Added configurable max_message_size to heka-cat.

* Added `count` setting to TokenSplitter to allow splitting on every Nth
  instance of the given delimiter.

* Added `deliver_incomplete_final` setting to SplitterRunner.

* Added `max_message_size` setting to UdpOutput.

Bug Handling
------------

* Fix the message matcher parser to clear the yySymType structure on each call
  (#1409).

* Protect read_message("raw") from reading an empty pack.MsgBytes (#1405).

* Moved `SetSplitterRunner` out of the config's MakeRunner and into the
  `NewSplitterRunner` function to make sure a splitter always has access to a
  SplitterRunner if needed.

* Fixed SplitterRunner buffer readPos so it only increments when a read call
  doesn't return an error (#1367).

* Ensured plugin list file is closed after writing to it in Dashboard plugin.

* Fixed exiting / restart behavior for ProcessInput to match documentation
  (#1395).

* UdpOutput drops message if its size exceeds allowed UDP data size (#1393).

0.9.0 (2015-02-25)
==================

Backwards Incompatibilities
---------------------------

* Major overhaul of Heka's configuration loading code. This doesn't impact
  most plugins, it's only a breaking change for plugins that happen to
  instantiate and manage the lifecycles of other embedded plugins, e.g.
  MultiDecoder, ProcessDirectoryInput, SandboxManagerFilter.

* All inputs now support `decoder`, `synchronous_decode`, and
  `send_decode_failures` config options, automatically extracted by Heka's
  config system.

* InputRunner now handles message decoding and delivery to the router,
  according to the specifications of the input's configuration. This is
  accomplished either directly via the `Deliver` method or, in cases where
  decoding might need to happen in separate goroutines, through `Deliverer`
  objects available from the `NewDeliverer` method.

* Hekad and all clients started using stdout for informational messages and
  stderr for error messages.

* Stats Accum Input treats gauge inputs as float64 rather than int64 to match
  statsd spec and other statsd implementations (#850).
```

```
* Introduction of Splitter plugins, accompanied by major changes to how input
  plugins work to support the use of Splitter plugins. (#424)

* TcpInput and UdpInput now set a non-protobuf encoded message's Type field to
  the configured input plugin name instead of the hard-coded string
  "NetworkInput".

Bug Handling
------------

* Fix the shutdown hang caused by a Filter/Output plugin failure (#1324).

* Fix the RPM dependency errors on Centos7 (#1311).

* Don't output empty messages when use_framing = true (#1326).

* KafkaInput uses the oldest offset available instead of zero when no
  checkpoint file exists and the offset method is set to manual. KafkaInput
  also correctly closes the checkpoint file before removing it when it is
  invalid (#1325).

* Handle empty byte fields in the sandbox interface (#1284).

* Reset header when discarding valid but oversized messages (#1221).

* Prevent the protobuf stream encoder from creating messages over
  MAX_MESSAGE_SIZE (#1204).

* AMQPOutput now uses defined constants for delivery mode instead of hard
  coded (and wrong) integer values (#1235).

* Hekad now respects --config parameter and loads default config path
  instead of printing help and exit (#1239).

* MessageProtoParser now handles EOF gracefully, and returns all
  complete records up to the end of the stream without having to
  use GetRemainingData() (#1305).

* HttpListenInput no longer URL unescapes the HTTP request body (#1124).

* SmtpOutput now encodes email subject when necessary (#1277).

* All config files are now pre-loaded before any of the config is actually
  loaded to prevent default plugins from being registered for each separate
  config file, possibly overriding custom config.

Features
--------

* Added decode_message to the sandbox API (#1344).

* Added a SandboxInput plugin (#1333).

* Added a SandboxOutput plugin (#1303).

* `heka/plugins/amqp` package now exposes `GetAmqpHub` and `NewAMQPDialer`
  functions for use by external packages.
```

```
* Added heartbeat monitoring sandbox filter plugin.

* Include the new lua_sandbox with support for shared library modules.

* Allow the sandbox process_message function to set the last error string
  when it returns (#1191).

* Improve the sandbox inject_* error messages (#1156).

* Added ability to specify unit type and aggregation methods for stats graph
  filter.

* Added option (immediate_start) to start process immediately in ProcessInput
  (#1131).

* Added `hostname` setting to global `[hekad]` config options to allow
  overriding the hostname value provided by Go's `os.Hostname` call (#1123).

* Added 'connect_timeout' option in ElasticSearch output plugin.

* Slightly improve error output in ElasticSearch plugin.

* Added support for HTTP authentication to ElasticSearchOutput.

* TcpOutput's disk buffering now supports specification of a max buffer size
  and an action to take if the max size is reached, from 'shutdown', 'drop',
  or 'block' (#1110).

* Added `send_interval` setting to SmtpOutput.

* Added `timestamp` format setting to ESLogstashV0Encoder (#1142).

* LogstreamerInput now uses first two "magic" bytes to identify gzip files
  instead of relying on `.gz` suffix.

* Added on-disk buffering to ElasticSearchOutput. It is enabled by default.

* Added TLS support to ElasticSearchOutput (#1259).

* Added ability for HttpListenInput to capture specified HTTP request headers
  and write them as message fields.

* Added ability for Statsd input handler to treat a malformed stat line
  one at time without skipping all the "good" stats in a multi-line input
  (more like statsd itself).

* Added `message_interval` setting support to heka-flood test configuration.

* Added `cert_path` setting for TLS support to DockerLogInput.

* Added new fields to messages generated by HttpListenInput, including
  RemoteAddr, Path, EnvVersion, Hostname (the name of the server that
  handled the request), and Host (the host to which the client sent the
  request) (#1328).

* Added `hostname_keep` option to RsyslogDecoder (#1338).

* Added an option to Schema InfluxDB Encoder to send the data to InfluxDB as
```

```
  a list of series rather than the default of a single series for all fields
  in the message.

* Added an option to Schema InfluxDB Encoder that excludes all base fields
  from being sent to InfluxDB to reduce the network traffic and storage
  demands if these fields aren't useful.

* Added `read_only` setting to AMQPInput to support read-only users.

* Added ability to specify MAX_MESSAGE_SIZE via configuration file of hekad
  and heka-flood (#1208).

* Changed default plugin_chansize setting from 50 to 30 (#1242).

0.8.3 (2015-01-08)
==================

Bug Handling
------------

* Fixed LogstreamerInput to use a separate DecoderRunner for every
  LogstreamInput created, rather than sending all of the streams through a
  single decoder.

0.8.2 (2015-01-06)
==================

Bug Handling
------------

* Fix rsyslog sandbox decoder test to use current year and location for
  timestamp parsing, since syslog timestamp format makes that assumption.

* Ensure that geoip_decoder is not included in release binaries.

0.8.1 (2014-12-17)
==================

Bug Handling
------------

* Fix leaky file descriptor bug on http_output.go.

* Only stamp PipelinePack diagnostics in cases when a pack actually matches a
  message matcher, instead of for every matcher every time (#1167).

* StatsdInput allows to specify (via max_msg_size option) size of message read
  from UDP (#1165).

* AMQPOutput now recycles packs even when a publish error causes the output to
  exit / restart (#1178).

* Fixed HttpOutput TLS section parsing (#1163).

* Fixed LogstreamInput extraneous journal saves that caused high disk IO
  when EOF is reached.

0.8.0 (2014-10-29)
```

```
==================

Backwards Incompatibilities
---------------------------

* Audited the use of Logger and Type headers on internal message (#1024). These
  changes may break some existing message matchers with regards to
  plugin/sandbox termination messages.

* Sandbox decoder linux_cpustats has been renamed to linux_loadavg and the
  corresponding filter cpustats was renamed to loadavg (#1094).

* ESLogstashv0 encoder will now set the @type field to match the ES record
  type specified in the `type_name` setting. This is Logstash behaviour in
  V0. You can revert back to the old method with `use_message_type = true`

Bug Handling
------------

* linux_diskstats.lua will now parse a line with no leading space (#1141).

* Protect globals.Stopping with a RWMutex.

* Added support for `fields` config setting to ESLogstashV0Encoder; it was
  already documented, but hadn't been implemented (#1096).

* Fixed deadlock race condition in AMQPOutput shutdown sequence (#824).

Features
--------

* Added Kafka Input/Output plugins (#1148).

* Optimized prefix/suffix testing in the message matcher (#1040).

* LogstreamerInput now supports seekjournal file hashing for files that are
  less than 500 bytes in length; hash is generated against file contents with
  0 bytes prepended to make 500 bytes of hash content (#972).

* Support environment variables in config files (#1023).

* Add Dockerfiles and example Docker usage to repo

* Encoders can now return (nil, nil) to express that they've swallowed the
  input message without generating any output.

* SandboxEncoder now supports return value of -2 from the process_message call
  to specify that no output is being generated.

* Added schema_influx.lua encoder.

* Added CBUF Librato Encoder.

* Added option to disable re-using TCP connections to the ElasticSearch output

* Added separate env loading file and support for NUM_JOBS env var to Windows
  build (#971).
```

```
* Added DockerLogInput (issue #1092).

* Added support for "Epoch", "EpochMilli", "EpochMicro", and "EpochNano"
  timestamp formats to the time parsing code used by the Payload*Decoder
  plugins (#963).

* Added ability to drop big message entirely or to keep first part of it (#1134)

0.7.3 (2014-10-28)
==================

Bug Handling
------------

* Fail gracefully on LogstreamerInput missing `file_match` setting (#1105).

* Fixed issue where ProcessInput wasn't propagating environment variables and
  working directory to repeatedly run (vs. one time, long running) processes
  (#1108).

* UdpInput now removes a Unix datagram socket file at shutdown time if one is
  created at startup.

* Fixed issue with keeping previous EOF state in LogStreamer (#1119).

* ProcessInput now honors documented restart behavior (#1117).

0.7.2 (2014-10-01)
==================

Bug Handling
------------

* Fail more gracefully if user doesn't specify a scheme in the server URL for
  the ElasticSearchOutput (#1069).

* Correctly compute http timeout interval for ElasticSearchOutput when using
  HTTP indexing.

* Fixed issue w/ inaccurate payload sizes in heka-flood.

* Allow severity to be modified by the MessageTemplate logic for use in
  ScribbleDecoder (#1084).

* Render pack diagnostic idle time in seconds (as specified) instead of
  nanoseconds.

* Prevent orphaned matchers from remaining in the router's matcher lists in
  cases where a filter or output registered in one TOML file overrides a
  filter or output of the same name in an earlier TOML file.

0.7.1 (2014-09-02)
==================

Bug Handling
------------

* Fixed handling of TcpInput and TcpOutput keep_alive_period default value
```

```
  handling, and added docs (#1054).

* Fixed OSX lua_sandbox build error.

* Fixed load ordering of nested MultiDecoders (#1045).

0.7.0 (2014-08-27)
==================

Backwards Incompatibilities
---------------------------

* Switched to using fork of gomock, import location changed from
  `code.google.com/p/gomock` to `github.com/rafrombrc/gomock`.

* Move *GlobalConfigStruct out of pipeline package's global namespace, it is
  now stored as an attribute on PipelineConfig. Any code that used to call
  pipeline.Globals() now must instead access *PipelineConfig.Globals. The
  `WantsPipelineConfig` interface has been introduced to give plugins access
  to the active *PipelineConfig if it's needed in the ConfigStruct or Init
  methods (#951).

* Removed deprecated `pipeline.GetHekaConfigDir` API call, which has been
  replaced by `PrependBaseDir` and `PrependShareDir` for a few versions
  now.

* Removed the regular expression template support from the message matcher
  (issue #960).

* Updated the AMQP input / output configuration values to use underscore
  delimited words for long config options (issue #953).

* LogstreamerInput now errors on nonexistent `log_directory` instead of
  creating the folder (issue #1066).

Bug Handling
------------

* Fixed the MySQL slow query grammar to handle logs with no newline.

* Fixed packet tracking idle packs error output formatting.

* Prevent panics during shutdown when a restarting plugin has been restarted
  (issue #957).

* Added support for a Logstreamer symbolically linked log_directory root
  (issue #741).

* Fixed lots of race conditions in the tests.

* Eliminated race condition in use of global AMQPHub in AMQP input / output
  plugins (#953).

Features
--------

* Added StatMetric Influx Encoder.
```

```
* Added RstEncoder.

* Added support for '__ignore_root' tag in add_external_plugin to allow
  external packages to be added which do not have any .go files
  in their root directory (issue #955).

* Switched from using goprotobuf to gogoprotobuf with marshal_all and
  unmarshal_all extensions for significant performance increase.

* Added stats graph sandbox filter to make it easier to generate graphs of
  statsd / graphite metrics (issue #966).

* Added support for the majority of repository URI formats when cloning external
  Go packages (issue #937).

* Added the ability to test for field existence in message matcher (issue #958).

* Added support to ESJSonEncoder and ESLogstashV0Encoder to properly encode
  field arrays.

* Added a filter to monitor all process message failures (issue #948).

* Added UdpOutput.

* Added IrcOutput.

* Added support for filter and output plugins to optionally exit without
  causing Heka to shutdown via config option.

* Added support for custom HTTP Headers in HttpListenInput and DashboardOutput.

0.6.1 (2014-08-27)
==================

Bug Handling
------------

* CarbonOutput using UDP transport now uses multiple packets to send stats
  data to Carbon server when output would be longer than the 64KiB max size
  for UDP packets (issue #1035).

* StatsdInput now trims all whitespace around incoming stat names instead of
  just trailing newlines (issue #1011).

* Fixed silent failures in ElasticSearchOutput bulk indexing operations.

* Fixed permanent loss of HTTP connection in certain cases when ElasticSearch
  has been restarted (issue #1008).

* Fixed default sandbox script type when loading dynamic plugins.

0.6.0 (2014-07-09)
==================

Backwards Incompatibilities
---------------------------

* Go 1.3 now required for building.
```

```
* Changed handling of message stream framing. Before stream framing was
  presumed to always be required when encoding with protocol buffers, and not
  required when using other encodings. Now all outputs support an optional
  `use_framing` setting that will determine whether or not framing will be
  applied when using the OutputRunner's Encode method (issue #922).

* All files in a config directory that do not end in ".toml" will now be
  ignored and not loaded as a part of the Heka configuration (issue #750).

* Removed the PayloadJsonDecoder which is replaced by the Lua sandbox cjson
  module (issue 826).

* Removed LogfileInput and LogfileDirectoryManagerInput which were replaced
  by LogstreamerInput in 0.5.0 (issue 914).

* MultiDecoder no longer sets prepends name of the decoder instance to the
  message type value for every message. This was impacting performance, is
  almost never what is needed, and is almost always overwritten by the nested
  subdecoders anyway.

* MultiDecoder now has a `subs` option that refers to other top level decoder
  config definitions, instead of nested sub-configs and a separate `order`
  option specifying the order. This makes for less typing and the ability to
  reuse decoder definitions as both standalone decoders and across multiple
  MultiDecoders (issue #485).

* ElasticSearchOutput now uses an Encoder plugin instead of MessageFormatters.
  Created ESJsonEncoder, ESLogstashV0Encoder, and ESPayloadEncoder
  (implemented as a SandboxEncoder in es_payload.lua) to replicate prior
  formatter behavior.

* Changed the Lua sandbox API.  inject_message is no longer overloaded, output
  has been renamed to add_to_payload, and inject_payload has been introduced.

* Changed TcpOutput, FileOutput, and LogOutput to use Encoder plugins for
  output formatting instead of bespoke formatting implementations.

* The rsyslog decoder now further parses the %SYSLOGTAG% variable. The
  fields.syslogtag no longer exists and is replaced by fields.programname and
  message.Pid (issue #677).

* Removed no-longer-used 'decoder_poolsize' global config setting.

Bug Handling
------------

* Fixed the ElasticSearchOutput to always use UTC times in the bulk API header
  (issue #504).

* Fixed the SandboxDecoder panic when failing the Decode() after successfully
  injecting a message (issue #910).

* StatsdInput no longer spins up a new goroutine for each stat (issue #359).

* MultiDecoder no longer prevents shutdown if a nested SandboxDecoder crashes
  during startup (issue #896).

* MultiDecoder using `cascade_strategy = "all"` now passes all generated packs
```

```
  to all nested decoders instead of skipping the remainder if an earlier pack
  in the sequence fails to decode (issue #896).

* MultiDecoder no longer tries to recycle the original pack, leaving that job
  to the DecoderRunner as intended (issue #896).

* Updated SmtpOutput to use a slice instead of a map to generate SMTP headers
  so the order will remain consistent when using Go 1.3 or later.

* Use FindProcess instead of syscall.Kill to make the 'pid_file' configuration
  setting work on Windows (issue #807)

* Fix the SandboxFilter ReportMsg panic on termination/shutdown (issue #816)

* MultiDecoder with no 'order' set now fails on init instead of panicking on
  first message.

* Fix panic on SIGUSR1 caused by no reports in a given plugin category (issue
  #832).

* Fix file_match config of plugin logstreamer_input (issue #893).

* StatFilter now correctly handles values from integer or float fields (issue
  #612).

Features
--------

* Added general purpose HttpOutput (issue #820).

* Added message processing stat reporting to the MultiDecoder for each
  subdecoder and in aggregate (issue #719).

* Added memory statistics to the Heka report output, for self monitoring.

* Added a message type configuration option to the rsyslog decoder (issue #907).

* Added `elasticsearch` Lua module to generate BulkAPI index JSON (issue #875).

* Added support for Lua sandbox preservation versioning (issue #701).

* Added support for unix datagram sockets to the UdpInput using net "unixgram"
  (issue #790).

* Add LogstreamInput seekjournal reporting to dashboard output (issue #445).

* Added a HyperLogLog library to the Lua sandbox using the Redis
  implementation http://antirez.com/news/75.

* Added an alert encoder to make the alert messages easier to read.

* Introduced Encoder plugin type (issue #417).

* Added a bloom filter to the Lua sandbox and created a unique items filter.

* Turned the MySQL slow query log examples into a deployed decoder and filter.

* Added an Nginx error log decoder (issue #785)
```

```
* Added the ability to preserve the webserver log line in message payload
  (issue #784)

* Added the optional 'pid_file' configuration setting (issue #777).

* Add anomaly, alert, annotation modules (issue #677)

* Added 'sample_denominator' global config setting to allow tweaking the
  sample interval when computing timing of certain operations, replacing prior
  hard-coded DURATION_SAMPLE_DENOMINATOR constant (issue #625).

* Added an Apache access log decoder based on the Apache 'LogFormat'
  configuration directive.

* Added BufferedOutput. Extracts the queuing functionality out of TcpOutput.go
  into a general purpose lib for use in any output module. Callers get
  messages buffered to disk while another goroutine consumes and forwards
  data. Any errors encountered can cause the sending goroutine to backoff and
  resend data.

* Added http_timeout to elasticsearch output to prevent slow or stale
  connections from holding up the flow. (issue #769)

* Add query parameters to the Message as Fields in the HttpListenInput

* Add QueueTTL option to AMQPInput to allow specifying message expiration.

* Added optional tls sub-section to AMQPInput and AMQPOutput for configuring
  AMQPS TLS settings.

* Add detection and handling of gzipped files to LogstreamerInput (issue #648).

* Added optional TCP Keep-Alive parameters to TcpInput/TcpOutput plugins.

0.5.2 (2014-05-16)
==================

Bug Handling
------------

* FileOutput no longer panics when using `format = "text"` and payload is nil
  (issue #843).

* Fix SandboxDecoder pass-through case so decoders that only use write_message
  and not inject_message will emit messages correctly (issue #844).

* Fixed TcpInput so it will only override the default delimiter for a
  RegexpParser if a delimiter is specified in the config.

* Fix the FileOutput panic when HUP'ed.  Modified the CheckWritePermission
  utility function to use a unique filename for each check. (issue #808)

* Terminated plugins are now removed from the SandboxManager quota (issue
  #774).

* Fixed SIGUSR1 triggered text report to stdout to work with updated
  heka.all-report JSON data structure (issue #762).
```

```
* MultiDecoder now uses message.GetType() instead of Message.Type so
  emitted message type will be generated correctly (issue #761).

* LogstreamerInput no longer causes Heka to fail to start on empty or
  whitespace-only journal files (issue #755).

* The cbufd_host_aggregator filter now properly reclaims expired hosts.

* Pull in a new lua_sandbox to fix JSON encoding of empty strings in the
  sandbox plugin output() call.

* Escape regexp meta characters (notably '\') to prevent a panic in the LogstreamerInput on Windows

0.5.1 (2014-03-18)
==================

Bug Handling
------------

* Skip *.bak, *.tmp, *~, and .files in a config dir as a non-breaking band-aid
  until we require an explicit naming convention in 0.6 (see issue #750).

* heka-logstreamer command now supports config directories in addition to
  single config files, just like hekad itself (issue #742).

* Logstreamer package's NewLogstreamSet function no longer lowercases the
  match part names when constructing match translation maps since the
  PopulateMatchParts method doesn't actually expect the names to be
  lowercased.

* Added support for use of "missing" value in Logstreamer translation maps to
  allow users to place missing values at the end of the list instead of the
  beginning (issue #735).

0.5.0 (2014-03-06)
==================

Backwards Incompatibilities
---------------------------

* ProcessInput no longer supports a separate'name' config setting, it uses the
  specified plugin name from the TOML section header like all of the other
  plugins.

* Removed Stdout_chan and Stderr_chan from ManagedCmd and CommandChain, client
  code now has direct access to Stdout_r and Stderr_r io.Reader providers.

* The PluginHelper interface DecoderRunner prototype has changed (issue #717)
  to allow for the base name e.g. "ProtobufDecoder" and a full instance name
  e.g. "MyInput-ProtobufDecoder" of the decoder to be specified.  This allows
  multiple decoders of the same type to show up on the DashboardOutput and
  sandbox decoder state preservation to work properly. Also the DecoderRunner
  UUID interface method was removed.

* FileOutput `flushinterval` config setting changed to `flush_interval` to
  match config naming conventions.

* SandboxDecoder, SandboxFilter, and SandboxManagerFilter now all use
```

```
  `${SHARE_DIR}/lua_modules` as the default `module_directory` setting.
  SandboxDecoder and SandboxFilter both now interpret relative paths to lua
  source code to be relative to ${SHARE_DIR} instead of ${BASE_DIR}.

* DashboardOutput `static_directory` setting now defaults to
  `${SHARE_DIR}/dasher` instead of `${BASE_DIR}/dashboard_static`.

* The sandbox preservation data is now stored in the
  {base_dir}/sandbox_preservation directory instead of with the plugin source.
  On the initial restart no preserved data will be restored unless it is
  manually moved to this directory first. (issue #626)

* The Heka utilities (flood, sbmgr, sbmgrload, inject) have been namespaced
  with a `heka-` prefix for their respective binaries. I.e. Flood has been
  renamed heka-flood., etc.

* MultiDecoder now gets its name from the TOML section name like the rest of
  the plugins, instead of a separate 'name' config option.

* Major reorganization of the `pipeline` package, moving the implementation of
  most plugins to sub-packages of a separate `plugins` package.

* Removed the wrapper 'table' element from the JSON serialization (issue #525)
  i.e., {"table":{"value":1}} is now simply {"value":1}.  The change also
  removes the special '_name' metadata arribute; the top level _name element
  should be created in the Lua structure if it is required.

* In the process of removing the core sandbox code from Heka (issue #464), the
  sandbox was streamlined to only load the base library by default. All sandbox
  plugins must now explictly load additional libraries with the require
  function.

* Removed DecoderSet method from PluginHelper interface (and DecoderSet
  abstraction entirely) and replaced it with Decoder and DecoderRunner methods
  that return a Decoder or a DecoderRunner by name.

* Changed Decoder interface to support one input pack generating multiple
  output packs.

Bug Handling
------------

* Network parsers now return all records available in the parse buffer
  (issue #732).

* TcpInput now stops a given connection's decoder when the connection is
  closed, preventing memory pooling (issue #713).

* StatsdInput now doesn't fail with multiple stats in a single UDP packet.

* Set default StatAccumInput stat namespace prefix values even when
  `legacy_namespaces` is set to false (issue #630).

* Fixed cpuprof file from being closed right after opening so no data was
  being logged.

* Fixed LogfileInput so it will only override the default delimiter for a
  RegexpParser if a delimiter is specified in the config.
```

```
* PluginWrapper will now check for the WantsName interface when creating a
  plugin, and will set the plugin's name if appropriate.

* SandboxDecoder now explicitly logs a fatal error before triggering shutdown
  to ensure error message is actually emitted.

* Message severity now defaults to the highest RFC 5424 value (i.e. 7) implies
  low severity, rather than zero, which implies `emergency`, (issue #518).

* 'flood' command now outputs every send error (even 'connection refused'), and
  always increments 'messages sent' count even when there is a sending error
  (so setting 'num_messages' still works even if hekad stops responding,
  etc.).

* stat_accum_input will not fail when flushing a timer where the percentile
  is equal to the min value.

Features
--------

* Added ProcessDirectoryInput.

* InputRunners now support being specified as 'transient', meaning their
  lifespan should be managed by the code that creates the InputRunner and not
  Heka's pipeline code.

* HttpInput: now supports configuring the HTTP method, HTTP headers and HTTP
  Basic Authentication

* TLS Listeners can specify a 'client_cafile' which limits the CAs that a
  client cert can be chained to. This provides a mechanism for TLS Client AUTH.

* TLS Senders can specify a 'root_cafile' which limits the CAs that a
  server cert can be chained to. This provides a mechanims for TLS Server AUTH.

* Added StopDecoderRunner function to PluginHelper API so inputs can manually
  decommission decoders when they're no longer being used.

* The SandboxManagerFilter can now control the sandbox usage limits (issue #95)

* Added support for send_nsca to NagiosOutput as an alternative to direct http
  submission; also a way to explicitely specify service_description and host
  to match Nagios config

* Added configurable network types to TcpInput "tcp", "tcp4", "tcp6", "unix" and
 "unixpacket" (issue #539)

* Added configurable network types to UdpInput "udp", "udp4", "udp6"
  (issue #539)

* Added flush_count config setting to FileOutput to complement existing
  flush_interval. Also added flush_operator setting which can be "AND" or "OR"
  to specify how flush_count and flush_interval should be combined.

* Introduced `share_dir` global config setting, which specifies where Heka's
  read only resources should live. Defaults to `/usr/share/heka`. Also added
  `pipeline.PrependShareDir()` function for use within plugin initialization
  code.
```

```
* Added an rsyslog decoder based on the rsyslog string configuration template
  (issue #432).

* Added an Nginx access log decoder based on the Nginx 'log_format'
  configuration directive.

* heka-cat: A command-line utility for counting, viewing, filtering, and
  extracting Heka protobuf logs.

* TcpOutput has been redesigned to handle unavailable endpoints and dropped
  connections without data loss (issue #355).

* CarbonOutput now supports submitting messages via UDP, persistent TCP
  connection.

* Added Logstreamer Input [LogstreamerInput]: An input that replaces the
  Logfile and Logfile Directory Inputs and supports sequential reading of
  logstreams that span sets of ordered logfiles (issue #372).

* TcpInput, TcpOutput, and flood client now all support TLS encrypted TCP
  connections using Go's crypto/tls package.

* Added Http Listen Input [HttpListenInput]: An input that listens for
  HTTP requests on the specified address. If a decoder is not specified the
  input generates a message with the HTML request body as the payload.
  This input is especially useful for consuming and processing webhooks.
  (Issue #431)

* Added support for local external packages (issue #393)

* Inject: A command-line utility for injecting arbitrary messages into
  a Heka pipeline.

* Added Go ScribbleDecoder for efficient setting of purely static message
  field values.

* Exposed `write_message` API function to Lua decoders to allow mutation of
  existing decoded message (issue #577).

* HttpInput: Added urls (array) option.

* HttpInput: Failed and successful HTTP GET actions produce messages of Type
  "heka.httpinput.data", Logger = Request URL, and severity appropriate to
  HTTP status 200 or not. I.e. Connections responding with a status of 200
  produce a message with Severity 6, non-200 statuses Severity 1. Failure to
  connect produces a message with Severity 1 and Type "heka.httpinput.error".

* HttpInput: Fields[ResponseTime] populates with time duration in seconds for
  HTTP GET of URL, Fields[Status] with HTTP Status, Fields[Protocol] with HTTP
  protocol and version, Fields[StatusCode] with HTTP Response Status Code for
  successful HTTP GETs. The Circular Buffer Graph Annotation (Alerts)
  (http://hekad.readthedocs.org/en/latest/sandbox/graph_annotation.html) plugin
  is compatible with the HttpInput plugin.

* HttpInput: Added success_severity and error_severity options for GET actions.

* HttpInput: Messages now set Logger, UUID, and Timestamp.
```

```
* Added log_errors option to PayloadregexDecoder to allow skipping invalid
  payloads.

* Added "id" flag to elasticsearch output (issue #386).

* Added SmtpOutput (issue #472)

* Added preserve_data option to SandboxDecoder (issue #668).

* Added delete_idle_stats to StatAccumInput.

* Added sum and count_ps metrics to timers in stat_accum_input.

0.4.2 (2013-12-02)
==================

Bug Handling
------------

* Changed CPack configuration such that the 'make package' target no longer
  creates deb packages, and added a new 'make deb' target that creates debs
  with the filename expected by the deb package naming conventions (see
  https://github.com/mozilla-services/heka/issues/545).

* Doc clarifications re: required use of ProtobufDecoder (see
  https://github.com/mozilla-services/heka/issues/550).

* Explicitly exclude system level folders from those that CPack will include
  as part of the Heka RPM (requires CMake >= 2.8.12) (see
  https://github.com/mozilla-services/heka/issues/548).

0.4.1 (2013-11-05)
==================

Bug Handling
------------

* Updated Mozilla Sphinx theme submodule reference and configuration paths to
  work around bug in ReadTheDocs rendering (see
  https://github.com/rtfd/readthedocs.org/issues/529).

* Changed default DashboardOutput.StaticDirectory value to
  `/usr/share/heka/dasher`, to match where the packages put the files.

Features
--------

* add require_all_fields flag to PayloadJsonDecoder
  (issue #528)

0.4.0 (2013-10-30)
==================

Backwards Incompatibilities
---------------------------

* DashboardOutput now requires a `static_directory` setting to specify where
  the Heka dashboard source code can be found.
```

```
* AMQPInput now only supports a single decoder to be specified (issue #414).

* UdpInput configuration now requires a parser_type and decoder to be specified
  (issue #411).

* TcpInput configuration now requires a parser_type and decoder to be specified
  (issue #165).

* Removed the LogfileInput decoders array configuration in favor of a single
  decoder (issue #308).

* SandboxManagerFilter now auto-generates a default working directory based on
  the plugin name, in the ${BASE_DIR}/sbxmgrs folder.

* LogfileInput seek journals now support a boolean `use_seek_journal` config
  flag, and only the name of the journal file is now settable via
  `seek_journal_name`, replacing the `seekjournal` option which specified the
  full path.

* Build is now entirely cmake based and lives directly in the heka repository,
  replacing the separate primarily gnu make driven separate heka-build repo.

* Core Heka plugins now default to writing data to /var/cache instead of
  /var/run since /var/run is usually deleted on server restart.

* StatAccumInput now defaults to emitting stats data in the payload instead of
  in the message fields.

* Renamed LoglineDecoder to PayloadRegexDecoder.

Bug Handling
------------

* Added check for nil RemoteAddr value since UDP connections won't have one
  (issue #462).

* Statsd messages are now parsed manually (issue #44).

* Added mDRunner so MultiDecoder will work with sub-decoders that implement
  the WantsDecoderRunner interface.

* Removed all code that attempted to catch and recover from panics in called
  plugin code, since a) the recovery often left Heka in an undefined, broken
  state b) the safety was a lie, any panic in a separate goroutine would not
  be caught and c) panic traceback actually makes it easier to debug.

* WhisperOutput and CarbonOutput no longer require stats to use the `stats`
  namespace prefix.

* LogfileInput now truncates and overwrites the seek journals instead of
  appending.

* `protoc` now an optional dependency for the build, only required if the
  protobuf definition is actually changed.

* MessageTemplate field name interpolation was only supporting alphabetic
  characters for field names, now supports all "word" characters (i.e.
  alphanumeric plus underscore).
```

```
* ElasticSearchOutput now always sends timestamps in expected UTC time zone
  format.

* Fixed a CarbonOutput memory / socket leak.

* Fixed panic caused by an invalid config file path.

Features
--------

* Added a sandbox message field iterator (issue #460)

* Added ProcessInput. (issue #406)

* Replaced DashboardOutput UI with new Backbone.js-based `dasher` interface.
  (issue #378)

* Add delta output to circular buffers. (issue #467)

* Added json decoding to the Lua sandbox (issue #463). If you have an old build
  directory you will have to run make clean to pick up the new patch.

* Added a 'config' variable section to the Sandbox decoder/filter plugin.
  (issue #444)

* Added StatsToFieldsDecoder that will convert a message with statsd formatted
  string in the payload to one that also has the data available as message
  fields, in the same format that would be generated by a StatAccumInput with
  `emit_in_fields` set to true.

* MultiDecoder now supports `cascade_strategy` config options of `first-wins`
  and `all`, defaults to `first-wins`.

* Added stream parsing to UdpInput. (issue #411)

* Added stream parsing to TcpInput. (issue #165)

* Added SandboxDecoder. (issue #370)

* Support Heka protobuf message data in the LogfileInput plugin. (issue #261)

* Added "payload" format option to ElasticSearchOutput to pass message payload
  through to ES unchanged.

* Allow LogfileInput to use configurable delimiters including regexp (issue #263)
  This includes a backwards incompatible change for the journal file, old
  journals will not be honored and the new journal will overwrite it.

* Allow add_external_plugin to specify sub-packages. (issue #384)

* Added `base_dir` global (i.e. `[hekad]` section) config option to specify a
  single location Heka plugins should use to persist data to the file system.

* The `hekad` commands `config` flag now supports use of a config directory in
  addition to a single config file. If a directory path is specified, every
  file in the directory will be parsed and they will all be merged to a single
  running configuration.
```

```
* Added LogfileDirectoryManagerInput (issue #344).

* LPEG library added to Lua sandbox (issue #144).

* ElasticSearchOutput now supports `logstash_v0` output format that is modeled
  after the original Logstash ElasticSearch message schema.

* Added PayloadJsonDecoder that can extract information from JSON text in a
  message payload during decoding.

* Make index type_name option for ElasticSearchOutput definable from Field value (issue #356)

0.3.0 (2013-07-16)
==================

* Fixed default (and only supported, for now) sample rate in the stats
  generated by the StatFilter.

* Fixes to the platform specific build constraints around signal handling.

* Added HttpInput that can make HTTP requests to a remote server, placing
  the response body into a message payload.

* Filters and outputs now require a specified message_matcher or they will
  raise an error applying the configuration.

* ticker_interval now exposed as a global config option for inputs, as well
  as outputs and filters.

* Overhaul of StatAccumInput implementation to fix bugs and allow for better
  handling of output format options.

* LoglineDecoder will now default to the current day, month, and year when
  parsing text that contains bare timestamps w/ no date information.

* Added permission checks to FileOutput and WhisperOutput plugins to fail
  gracefully when they've been asked to write to a location where they do not
  have write permission.

* Fixed folder creation permission bugs and allow configurable folder
  permission settings in FileOutput and WhisperOutput.

* `hekad` now emits help message if started w/ no command line options.

* Consistently use underscore_separated_words for plugin TOML configuration
  options.

* Use strings instead of integers to specify octal permission values in the
  config, since TOML doesn't support octal integer values.

* LoglineDecoder config now allows specification of a time zone in case the
  parsed text contains non-UTC timestamps that don't include embedded time
  zone information.

* Removed match capture group support from the router's message matching
  functionality, since this is better done in the decoder layer where the
  back-pressure won't slow down all message delivery. This gets rid of the
  `PipelineCapture` object altogether, so now filter and output plugins deal
```

```
  w/ PipelinePacks directly.

* Added ElasticSearch output.

* Lua filters can now emit tables, which will be serialized to JSON and
  put in an injected message payload.

* Custom dashboard javascript now generated by Heka instead of hosted
  remotely.

* Improvements to flood, a Heka protocol load test client.

* Config loading now fails if a config section specifies an unrecognized
  configuration option.

* StatAccumulator now supports stat metric messages with stats data embedded
  in message fields as well as message payload.

* Added support for plugin code to provide message_matcher and ticker_interval
  default values.

* Reimplemented StatMonitor as StatAccumInput, providing a StatAccumulator
  interface.

* SIGUSR1 signals now generate Heka report on stdout instead of sending
  a message so it can be viewed even when message delivery is hosed.

* Added explicit Close() calls to whisperdb files in WhisperOutput.

* Redesigned message field metadata to use string specifiers instead of
  an enum.

* Lua filters can now emit multiple named outputs to the Heka dashboard.

* LogfileInput now supports just one file per input.

* Removed race conditions exposed by Go's race detector.

* Improved self monitoring / dashboard output.

* Improved shutdown message flushing.

* Added Nagios output.

* Added support for LogfileInput to resume parsing where it left off.

* Added AMQP input and output plugins.

* Improved control over sandbox filter message injection restrictions.

* Added support for restartable plugins.

* Moved regular expression and capture group parsing out of the filter layer
  (i.e. in the TransformFilter) and into the decoder layer (i.e.
  LoglineDecoder) to prevent back pressure from impacting the entire router.

0.2.0 (2013-06-26)
==================
```

```
* Fix stat name regex to capture "." characters in the name.

0.2.0rc2 (2013-05-23)
=====================

* Fix Lua sandbox C code to work on 32 bit systems.

* Trivial release documentation ReST formatting fix.

0.2.0rc1 (2013-05-21)
=====================

* Use non-https links in docs to prevent bad rendering due to mixed http/https
  content.

* A number of documentation tweaks and updates.

* Static linking of cgo wrapped Lua environment.

* Added LICENSE.txt and CHANGES.txt for better project hygiene.

* Changed default interval for log file reading from 1ms to 500ms so we don't
  churn the machine in default configuration.

* Moved StatPacket channel setup into StatsdInput's Init method (from Run) to
  avoid race conditions.

* Added support for mingw-based Windows build.

* Perform message injection in a goroutine to prevent blocking the router when
  the plugin that is doing the injection has a full in channel.

* Added required pack recycling to TcpOutput.

0.2.0b1 (2013-04-30)
====================

* Initial public release
```

2.19 Transitional Filter and Output APIs

Heka's APIs for filter and output plugins have changed drastically between the v0.9.X series and v0.10.0. To ease the transition, however, a very slightly modified version of the older APIs will be supported throughout the v0.10.X release series. These should allow users with pre-existing custom filter and output plugins to upgrade their plugin code to work with the newly introduced *disk buffering* support with minimal effort.

These transitional APIs are documented in full below, but a brief summary is that the APIs are identical to the filter and output APIs supported by the v0.9.X series, with two exceptions:

- The plugin code must call the runner's *UpdateCursor* method as appropriate to advance the disk buffer's cursor, exactly as required by the newer APIs.
- The `pack.Recycle` method now requires an `error` argument which signals to Heka whether or not processing of the message in question was successful. Using nil for this argument implies that the message was successfully processed. Using a regular, non-nil error (such as is returned by `errors.New` or `fmt.Errorf`) implies

that the message was unable to be processed but should not be tried again. Using a `RetryMessageError` (as is returned by `pipeline.NewRetryMessageError`, see *here* implies that the message should be retried.

In other words, the argument passed in to the Recycle call is used similarly to the return value of the *MessageProcessor interface's* `ProcessMessage` function. The behavior is not identical, however. Use of a *PluginExitError* will not cause the plugin to exit, exiting is still signaled by returning from the plugin's `Run` method. Also, the RetryMessageError will only cause a message to be retried when buffering is in use. If a plugin does not have buffering turned on, then the value returned to the Recycle method will have no impact.

2.19.1 Filters (transitional API)

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 transitional filter plugin interface is just a single method:

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

Like input plugins, old-style 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. The similarities end there, however.

Filters should call `runner.InChan()` to gain access to the plugin's input channel. A filter's input channel provides pointers to PipelinePack objects, defined in pipeline_runner.go, each of which contains what should be by now a fully populated Message struct from which the filter can extract any desired information.

Sometimes, while processing a message, a filter plugin will need to generate one or more *new* messages, which can be handed back to the router to be checked against all registered plugins' `message_matcher` rules.

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 pack 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 pack has been obtained, a filter plugin can populate its Message struct using the various provided mutator methods. The pack can then 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 old style 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 PipelinePacks. 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 are two very important points that all authors of old style filter plugins should keep in mind. The first has to do with updating the disk buffer's queue cursor for cases when *buffering* is in use. Old style filters should extract

the QueueCursor values from packs and call `FilterRunner.UpdateCursor` as appropriate, just like filters that support the newer APIs.

The second important point is that an old style filter's code *must* call `pack.Recycle(err error)` to tell Heka that it is 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. The error value passed in to Recycle should be `nil` if the message was processed successfully. The error value should be non-nil if the message processing failed and the message should be dropped. The error value should be an instance of *RetryMessageError* if the processing failure was transient and message delivery should be tried again. Note that redelivery will only happen in cases where the filter's configuration has `use_buffering` set to true.

2.19.2 Outputs (transitional API)

Output plugins are responsible for receiving Heka messages and using them to generate interactions with the outside world. The `OldOutput` interface is nearly identical to the `OldFilter` interface:

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

In fact, there are many ways in which old style filter and output plugins are similar. Like filters, outputs should call the `InChan` method on the provided runner to get an input channel, which will feed PipelinePacks. Like filters, outputs should listen on this channel until it is closed, at which time they should perform any necessary clean-up and then return. And, like filters, any old style 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 `pack.Recycle` (passing in an appropriate error value) when they finish w/ a pack so that Heka knows the pack is freed up for reuse.

The primary way that outputs differ from filters, of course, is that outputs need to serialize (or extract data from) the messages they receive and then send that data to an external destination. The serialization (or data extraction) should typically be performed by the output's specified encoder plugin. The OutputRunner exposes the following methods to assist with this:

```
Encode(pack *PipelinePack) (output []byte, err error)
UsesFraming() bool
Encoder() (encoder Encoder)
```

The `Encode` method will use the specified encoder to convert the pack's message to binary data, then if `use_framing` was set to true in the output's configuration it will prepend Heka's *Stream Framing*. The `UsesFraming` method will tell you whether or not `use_framing` was set to true. Finally, the `Encoder` method will return the actual encoder that was registered. This is useful to check to make sure that an encoder was actually registered, but generally you will want to use OutputRunner.Encode and not Encoder.Encode, since the latter will not honor the output's `use_framing` specification.

2.20 Glossary

hekad Daemon that routes messages from inputs to their outputs applying filters as configured.

Message A message is the atomic unit of data that Hekad deals with. It is a data structure related to a single event happening in the outside world, such as a log file entry, a counter increment, an application exception, a notification message, etc. It is specified as a *Message* struct in the *heka/message* packages message.go file.

Message matcher A configuration option for filter and output plugins that specifies which messages that plugin accepts for processing. The Heka router will evaluate the message matchers against every message to and will deliver the message when the match is positive.

Pipeline Messages being processed by Hekad are passed through a specific set of plugins. A set of plugins to be applied to a message is often called (somewhat informally) a Heka pipeline.

PipelinePack In addition to the core message data, Hekad needs to track some related state and configuration information for each message. To this end there is a *PipelinePack* struct defined in the *heka/pipeline* package's pipeline_runner.go file. *PipelinePack* objects are what get passed in to the various Hekad plugins as messages flow through the pipelines.

Plugin Hekad plugins are functional units that perform specific actions on or with messages. There are six types of plugins: inputs, splitters, decoders, filters, encoders, and ouputs.

PluginChanSize A Heka configuration setting which specifies the size of the input channel buffer for the various Heka plugins. Defaults to 50.

PluginHelper An interface that provides access to certain Heka internals that may be required by plugins in the course of their activity. Defined in config.go.

PluginRunner A plugin-specific helper object that manages the lifespan of a given plugin and handles most details of interaction with the greater Heka environment. Comes in five variants, each tailored to a specific plugin type: *InputRunner*, *SplitterRunner*, *DecoderRunner*, *FilterRunner*, and *OutputRunner*.

PoolSize A Heka configuration setting which specifies the number of *PipelinePack* structs that will be created. This value specifies the maximum number of incoming messages that Heka can be processing at any one time.

Router Component in the Heka pipeline that accepts messages and delivers them to the appropriate filter and output plugins, as specified by the plugins' message matcher values.

2.21 Circular Buffer Graph Annotation (Alerts)

This plugin detects anomalies in the data. When an anomaly is detected an alert is generated and the graph is visually annotated at the time of the alert. See dygraphs Annotations for the available annotation properties.

```
-- 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/.

--[[
Collects the circular buffer delta output from multiple instances of an upstream
sandbox filter (the filters should all be the same version at least with respect
to their cbuf output). The purpose is to recreate the view at a larger scope in
each level of the aggregation i.e., host view -> datacenter view -> service
level view.

Config:

- enable_delta (bool, optional, default false)
    Specifies whether or not this aggregator should generate cbuf deltas.

- anomaly_config(string) - (see :ref:`sandbox_anomaly_module`)
    A list of anomaly detection specifications.  If not specified no anomaly
    detection/alerting will be performed.

- preservation_version (uint, optional, default 0)
    If `preserve_data = true` is set in the SandboxFilter configuration, then
    this value should be incremented every time the `enable_delta`
    configuration is changed to prevent the plugin from failing to start
    during data restoration.
```

```
*Example Heka Configuration*

.. code-block:: ini

    [TelemetryServerMetricsAggregator]
    type = "SandboxFilter"
    message_matcher = "Logger == 'TelemetryServerMetrics' && Fields[payload_type] == 'cbufd'"
    ticker_interval = 60
    filename = "lua_filters/cbufd_aggregator.lua"
    preserve_data = true

    [TelemetryServerMetricsAggregator.config]
    enable_delta = false
    anomaly_config = 'roc("Request Statistics", 1, 15, 0, 1.5, true, false)'
    preservation_version = 0
--]]
_PRESERVATION_VERSION = read_config("preservation_version") or 0

local alert     = require "alert"
local annotation= require "annotation"
local anomaly   = require "anomaly"
local cbufd     = require "cbufd"
require "circular_buffer"

local enable_delta = read_config("enable_delta") or false
local anomaly_config = anomaly.parse_config(read_config("anomaly_config"))

cbufs = {}

local function init_cbuf(payload_name, data)
    local ok, h = pcall(cjson.decode, data.header)
    if not ok then
        return nil
    end

    local cb = circular_buffer.new(h.rows, h.columns, h.seconds_per_row, enable_delta)
    for i,v in ipairs(h.column_info) do
        cb:set_header(i, v.name, v.unit, v.aggregation)
    end
    annotation.set_prune(payload_name, h.rows * h.seconds_per_row * 1e9)

    cbufs[payload_name] = cb
    return cb
end

function process_message ()
    local payload = read_message("Payload")
    local payload_name = read_message("Fields[payload_name]") or ""
    local data = cbufd.grammar:match(payload)
    if not data then
        return -1
    end

    local cb = cbufs[payload_name]
    if not cb then
        cb = init_cbuf(payload_name, data)
        if not cb then
            return -1
```

```
        end
    end

    for i,v in ipairs(data) do
        for col, value in ipairs(v) do
            if value == value then -- NaN test, only aggregrate numbers
                local n, u, agg = cb:get_header(col)
                if  agg == "sum" then
                    cb:add(v.time, col, value)
                elseif agg == "min" or agg == "max" then
                    cb:set(v.time, col, value)
                end
            end
        end
    end
    return 0
end

function timer_event(ns)
    for k,v in pairs(cbufs) do
        if anomaly_config then
            if not alert.throttled(ns) then
                local msg, annos = anomaly.detect(ns, k, v, anomaly_config)
                if msg then
                    alert.queue(ns, msg)
                    annotation.concat(k, annos)
                end
            end
            inject_payload("cbuf", k, annotation.prune(k, ns), v)
        else
            inject_payload("cbuf", k, v)
        end

        if enable_delta then
            inject_payload("cbufd", k, v:format("cbufd"))
        end
    end
    alert.send_queue(ns)
end
```

2.22 JSON Payload Transform

Alters a date/time string in the JSON payload to be RFC3339 compliant. In this example the JSON is parsed, transformed and re-injected into the payload.

```
-- 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/.

require "string"
require "cjson"

-- sample input {"name":"android_app_created","created_at":"2013-11-15 22:37:34.709739275"}

local date_pattern = '^(%d+-%d+-%d+) (%d+:%d+:%d+%.%d+)'
function process_message ()
```

```
    local ok, json = pcall(cjson.decode, read_message("Payload"))
    if not ok then
        return -1
    end
    local d, t = string.match(json.created_at, date_pattern)
    if d then
        json.created_at = string.format("%sT%sZ", d, t)
        inject_payload("json", "transformed timestamp", cjson.encode(json))
    end
    return 0
end

-- sample output
--2013/11/15 15:25:56 <
--      Timestamp: 2013-11-15 15:25:56.826184879 -0800 PST
--      Type: logfile
--      Hostname: trink-x230
--      Pid: 0
--      UUID: ef5de908-822a-4fe1-a564-ad3b5a9631c6
--      Logger: test.log
--      Payload: {"name":"android_app_created","created_at":"2013-11-15T22:37:34.709739275Z"}
--
--      EnvVersion: 0.8
--      Severity: 0
--      Fields: [name:"payload_type" value_type:STRING representation:"file-extension" value_string:
```

Alters a date/time string in the JSON payload to be RFC3339 compliant. In this example a search and replace is performed on the JSON text and re-injected into the payload.

```
-- 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/.

require "cjson"
require "string"
-- sample input {"name":"android_app_created","created_at":"2013-11-15 22:37:34.709739275"}

local date_pattern = '("created_at":)"(%d+-%d+-%d+) (%d+:%d+:%d+%.%d+)"'

function process_message ()
    local pl = read_message("Payload")
    local json, cnt =  string.gsub(pl, date_pattern, '%1"%2T%3Z"', 1)
    if cnt == 0 then
        return -1
    end

    inject_payload("json", "transformed timestamp S&R", cjson.encode(json))
    return 0
end

-- sample output
--2013/11/18 09:20:41 <
--      Timestamp: 2013-11-18 09:20:41.252096692 -0800 PST
--      Type: logfile
--      Hostname: trink-x230
--      Pid: 0
--      UUID: e8298865-fbc2-422c-873e-1210ce8efd9f
--      Logger: test.log
```

```
--      Payload: {"name":"android_app_created","created_at":"2013-11-15T22:37:34.709739275Z"}
--
--      EnvVersion: 0.8
--      Severity: 0
--      Fields: [name:"payload_type" value_type:STRING representation:"file-extension" value_string:
```

2.23 Common Sandbox Parameters

These are the configuration options that are universally available to all Sandbox plugins. The are consumed by Heka when it initializes the plugin.

- **script_type (string):** The language the sandbox is written in. Currently the only valid option is 'lua' which is the default.
- **filename (string):** The path to the sandbox code; if specified as a relative path it will be appended to Heka's global share_dir.
- **preserve_data (bool):** True if the sandbox global data should be preserved/restored on plugin shutdown/startup. When true this works in conjunction with a global Lua _PRESERVATION_VERSION variable which is examined during restoration; if the previous version does not match the current version the restoration will be aborted and the sandbox will start cleanly. _PRESERVATION_VERSION should be incremented any time an incompatible change is made to the global data schema. If no version is set the check will always succeed and a version of zero is assumed.
- **memory_limit (uint):** The number of bytes the sandbox is allowed to consume before being terminated (default 8MiB).
- **instruction_limit (uint):** The number of instructions the sandbox is allowed to execute during the process_message/timer_event functions before being terminated (default 1M).
- **output_limit (uint):** The number of bytes the sandbox output buffer can hold before being terminated (default 63KiB). Warning: messages exceeding 64KiB will generate an error and be discarded by the standard output plugins (File, TCP, UDP) since they exceed the maximum message size.
- **module_directory (string):** The directory where 'require' will attempt to load the external Lua modules from. Defaults to ${SHARE_DIR}/lua_modules.
- **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.

2.24 Logstreamer

New in version 0.5.

The Logstreamer plugin scans, sorts, and reads logstreams in a sequential user-defined order, differentiating multiple logstreams found in a search based on a user-defined differentiator.

A "logstream" is a single, linear data stream that is spread across one or more sequential log files. For instance, an Apache or nginx server typically generates two logstreams for each domain: an access log and an error log. Each stream might be written to a single log file that is periodically truncated (ick!) or rotated (better), with some number of historical versions being kept (e.g. access-example.com.log, access-example.com.log.0, access-example.com.log.1, etc.). Or, better yet, the server might periodically create new timestamped files so that the 'tip' of the logstream jumps from file to file (e.g. access- example.com-2014.01.28.log, access-example.com-2014.01.27.log, access- example.com-2014.01.26.log, etc.). The job of Heka's Logstreamer plugin is to understand the file naming and ordering conventions for a single type of logstream (e.g. "all of the nginx server's domain access logs"), and to use

that to watch the specified directories and load the right files in the right order. The plugin will also track its location in the stream so it can resume from where it left off after a restart, even in cases where the file may have rotated during the downtime.

To make it easier to parse multiple logstreams, the Logstreamer plugin can be specified a single time with a single decoder for all the logstreams that should be parsed with it.

2.24.1 Standard Configurations

Given the flexibility of the Logstreamer, configuration can be more complex for the more advanced use-cases. We'll start with the simplest use-case and work towards the most complex.

Single Rotating Logfile

This is the basic use-case where a single logfile should be read that the system may rotate/truncate at some time (hopefully not using truncation though that condition is handled). Log rotation inherently has a risk that some loglines written may be missed if the program reading the log happens to die at exactly the wrong time that the rotation is occuring.

An example of a single rotating logfile would be the case where you want to watch /var/log/system.log for all new entries. Here's what the configuration for such a case looks like:

```
[syslog]
type = "LogstreamerInput"
log_directory = "/var/log"
file_match = 'system\.log'
```

Note: The `file_match` config value above is delimited with single quotes instead of double quotes (i.e. *'system\.log'* vs. *"system\.log"*) because single quotes indicate raw strings that do not require backslashes to be escaped. If you use double quotes around your regular expressions you'll need to escape backslashes by doubling them up, e.g. *"system\\.log"*.

We start with the highest directory to start scanning for files under, in this case `/var/log`. Then the files under that directory (recursively searching in sub-directories) are matched against the `file_match`.

The `log_directory` should be the most specific directory of files to match to prevent excessive file scanning to locate the `file_match`'s.

Multiple Single Rotating Logfiles

This use-case is similar to the single rotating logfile above except there are multiple separate files with the same policy.

An example of multiple single rotating logfiles would be a system that logs the access for each domain name to a separate access log. In this case to differentiate them, we will need to indicate what part of the `file_match` indicates its a separate logfile (using the domain name as the differentiator).

```
[accesslogs]
type = "LogstreamerInput"
log_directory = "/var/log/nginx"
file_match = '(?P<DomainName>[^/]+)-access\.log'
differentiator = ["nginx.", "DomainName", ".access"]
```

Note that we included two strings in the differentiator that don't correspond to a part in the `file_match` regular expression. These two parts will be included as is to create the logger name attached to each message. So a file:

`/var/log/nginx/hekathings.com-access.log`

Will have all its messages in heka with the logger name set to `nginx.hekathings.com.access`.

Single Sequential (Rotating) Logfile

What happens if you have a log structure like this?

```
/var/log/nginx/access.log
/var/log/nginx/access.log.1
/var/log/nginx/access.log.2
/var/log/nginx/access.log.3
```

Or perhaps like this?

```
/var/log/nginx/2014/08/1.access.log
/var/log/nginx/2014/08/2.access.log
/var/log/nginx/2014/08/3.access.log
/var/log/nginx/2014/08/4.access.log
```

Or a combination of them?

```
/var/log/nginx/2014/08/access.log
/var/log/nginx/2014/08/access.log.1
/var/log/nginx/2014/08/access.log.2
/var/log/nginx/2014/08/access.log.3
```

(Hopefully your setup isn't worse than any of these... but even if it is then Logstreamer can handle it.)

Handling a single access log that is sequential and rotated (the first example) can be tricky. The second case where rotation doesn't occur and new logfiles are written every day with new months/years result in new directories was previously quite difficult to handle. Both of these cases can be handled by the LogstreamerInput.

The other (fun) problem with the second case is that if you use a raw string listing of the directory then `11.access.log` will come before `2.access.log` which is not good if you expect the logs to be in order.

Let's look at the config for the first case, note that the numbers incrementing in this case represent the files getting older (the higher the number, the older the log data):

```
[accesslogs]
type = "LogstreamerInput"
log_directory = "/var/log/nginx"
file_match = 'access\.log\.?(?P<Seq>\d*)'
priority = ["^Seq"]
```

When handling sequential logfiles in a logstream, we need to indicate a list of matched parts in the `file_match` that will be used to sort the files matching in order from oldest -> newest. By default, the numbers are sorted in ascending order (which properly reflects oldest first if the number represents the year, month, or day). To indicate that we should sort in descending order we use the `^` in front of the matched part to sort on (`Seq`).

Here's what a configuration for the second case:

```
[accesslogs]
type = "LogstreamerInput"
log_directory = "/var/log/nginx"
file_match = '(?P<Year>\d+)/(?P<Month>\d+)/(?P<Day>\d+)\.access\.log'
priority = ["Year", "Month", "Day"]
```

First we match the portions to be sorted on, and then we specify the priority of matched portions to sort with. In this case the lower numbers represent older data so none of them need to be prefixed with `^`.

Finally, the last configuration is a mix of the prior two:

```
[accesslogs]
type = "LogstreamerInput"
log_directory = "/var/log/nginx"
file_match = '(?P<Year>\d+)/(?P<Month>\d+)/access\.log\.?(?P<Seq>\d*)'
priority = ["Year", "Month", "^Seq"]
```

Multiple Sequential (Rotating) Logfiles

Same as before, except now we need to differentiate the sequential streams. We're only introducing a single parameter here that we've seen before to handle the differentiation. Lets take the last case from above and consider it a multiple sequential source.

Example directory layout:

```
/var/log/nginx/frank.com/2014/08/access.log
/var/log/nginx/frank.com/2014/08/access.log.1
/var/log/nginx/frank.com/2014/08/access.log.2
/var/log/nginx/frank.com/2014/08/access.log.3
/var/log/nginx/george.com/2014/08/access.log
/var/log/nginx/george.com/2014/08/access.log.1
/var/log/nginx/george.com/2014/08/access.log.2
/var/log/nginx/george.com/2014/08/access.log.3
/var/log/nginx/sally.com/2014/08/access.log
/var/log/nginx/sally.com/2014/08/access.log.1
/var/log/nginx/sally.com/2014/08/access.log.2
/var/log/nginx/sally.com/2014/08/access.log.3
```

In this case we have multiple sequential logfiles for each domain name that are incrementing in date along with rotation when a logfile gets too large (causing rotation of the file within the directory).

Configuration for this case:

```
[accesslogs]
type = "LogstreamerInput"
log_directory = "/var/log/nginx"
file_match = '(?P<DomainName>[^/]+)/(?P<Year>\d+)/(?P<Month>\d+)/access\.log\.?(?P<Seq>\d*)'
priority = ["Year", "Month", "^Seq"]
differentiator = ["nginx-", "DomainName", "-access"]
```

As in the case for a non-sequential logfile, we supply a differentiator that will be used to file each sequential set of logfiles into a separate logstream.

See also:

Full set of configuration options

2.24.2 String-based Order Mappings

In the standard configurations above, the assumption has been that any part matched for sorting will be digit(s). This is because the Logstreamer by default will attempt to coerce a matched portion used for sorting into an integer in the event a mapping isn't available. LogstreamerInput comes with several built-in mappings and allows you to define your own so that matched parts can be translated to integers for sorting purposes.

Built-in Mappings

There are several special regex grouping names you can use that will indicate to the LogstreamerInput that a default mapping should be used:

- **MonthName:** English full month name or 3-letter version to the appropriate integer.
- **DayName:** English full day name or 3-letter version to the appropriate integer.

If the last example above looked like this:

```
/var/log/nginx/frank.com/2014/Sep/access.log
/var/log/nginx/frank.com/2014/Oct/access.log.1
/var/log/nginx/frank.com/2014/Nov/access.log.2
/var/log/nginx/frank.com/2014/Dec/access.log.3
/var/log/nginx/sally.com/2014/Sep/access.log
/var/log/nginx/sally.com/2014/Oct/access.log.1
/var/log/nginx/sally.com/2014/Nov/access.log.2
/var/log/nginx/sally.com/2014/Dec/access.log.3
```

Using the default mappings would provide us a simple configuration:

```
[accesslogs]
type = "LogstreamerInput"
log_directory = "/var/log/nginx"
file_match = '(?P<Domain>[^/]+)/(?P<Year>\d+)/(?P<MonthName>\s+)/access\.log\.?(?P<Seq>\d*)'
priority = ["Year", "MonthName", "^Seq"]
differentiator = ["nginx-", "Domain", "-access"]
```

LogstreamerInput will translate the 3-letter month names automatically before sorting (If used in the differentiator, you will still get the original matched string).

Custom Mappings

What if your logfiles (for reasons we won't speculate about) happened to use Pharsi month names but Spanish day names such that it looked like this?

```
/var/log/nginx/sally.com/2014/Hadukannas/lunes/access.log
/var/log/nginx/sally.com/2014/Turmar/miercoles/access.log
/var/log/nginx/sally.com/2014/Karmabatas/jueves/access.log
/var/log/nginx/sally.com/2014/Karbasiyas/sabado/access.log
```

It would be easier if the logging scheme just used month and day integers but changing existing systems isn't always an option, so lets work with this somewhat odd scheme.

The first chunk of our configuration:

```
[accesslogs]
type = "LogstreamerInput"
log_directory = "/var/log/nginx"
file_match = '(?P<Domain>[^/]+)/(?P<Year>\d+)/(?P<Month>\s+)/(?P<Day>[^/]+/access\.log'
priority = ["Year", "Month", "Day"]
differentiator = ["nginx-", "Domain", "-access"]
```

Now to supply the important mapping of how to translate `Month` and `Day` into sortable integers. We'll add this:

```
[accesslogs.translation.Month]
hadukannas = 1
turmar = 2
karmabatas = 4
```

```
karbasiyas = 6

[accesslogs.translation.Day]
lunes = 1
miercoles = 3
jueves = 4
sabado = 6
```

Note: The matched values used are all lowercased before comparison, so 'lunes' in the example above would match captured values of 'lunes', 'Lunes', and 'LuNeS' equivalently.

We left off the rest of the month names and day names not used for example purposes. Note that if you prefer the week to begin on a Saturday instead of Monday you can configure it with a custom mapping.

Mappings with Missing Values

In the examples above, the years and months were embedded in the file path as directory names, but what if the date was embedded into the filenames themselves, with a file naming schema like so?

```
/var/log/nginx/sally.com/access.log
/var/log/nginx/sally.com/access-20140803.log
/var/log/nginx/sally.com/access-20140804.log
/var/log/nginx/sally.com/access-20140805.log
/var/log/nginx/sally.com/access-20140806.log
/var/log/nginx/sally.com/access-20140807.log
/var/log/nginx/sally.com/access-20140808.log
```

Notice how the currently active log file contains no date information at all. As long as you construct your file_match regex correctly this will be fine, Logstreamer will capture all of the files and won't complain about entries that are missing the match portions. The following config would work to capture all of these files:

```
[accesslogs]
type = "LogstreamerInput"
log_directory = "/var/log/nginx"
file_match = '(?P<Domain>[^/]+)/access-?(?P<Year>\d4)(?P<Month>\d2)(?P<Day>\d2)\.log'
priority = ["Year", "Month", "Day"]
differentiator = ["nginx-", "Domain", "-access"]
```

This works to match all of the files because match groups are implicitly optional and we explicitly made the hyphen separator optional by following it with a question mark (i.e. *-?*). We still have a problem, however. Heka will automatically assign a missing match a sort value of -1. Because we're sorting by date values, which sort naturally in ascending order, the -1 value will come before every other value, it will be considered the oldest file in the stream. This is clearly incorrect, since the currently active file is actually the newest file in the stream.

It is possible to fix this by using a custom translation map to explicitly associate a sort index with the 'missing' value, like so:

```
[accesslogs.translation.Year]
missing = 9999
```

Note: If you create a translation map with only one key, that key *must* be 'missing'. It's possible to use the 'missing' value in a translation map that also contains other keys, but if you have any other key in the map you must include *all* possible match values, or else Heka will raise an error when it finds a match value that can't be converted.

2.24.3 Verifying Settings

Given the configuration complexity for more advanced use-cases, the Logstreamer includes a command line tool that lets you verify options and shows you what logstreams were found, the name, and the order they'll be parsed in. For convenience the same heka toml config file may be passed in to `heka-logstreamer` and `LogstreamerInput` sections will be located and parsed showing you how they were interpreted.

An example configuration that locates logfiles on an OSX system:

```
[osx-logfiles]
type = "LogstreamerInput"
log_directory = "/var/log"
file_match = '(?P<FileName>[^/]+).log'
differentiator = ["osx-", "FileName", "-logs"]
```

Running this through `heka-logstreamer` shows the following:

```
$ heka-logstreamer -config=test.toml
Found 10 Logstream(s) for section [osx-logfiles].

Logstream name: osx-appstore-logs
Files: 1 (printing oldest to newest)
    /var/log/appstore.log

.... more output ....

Logstream name: osx-bookstore-logs
Files: 1 (printing oldest to newest)
    /var/log/bookstore.log

Logstream name: osx-install-logs
Files: 1 (printing oldest to newest)
    /var/log/install.log
```

It's recommended to always run `heka-logstreamer` first to ensure the configuration behaves as desired.

2.25 hekad

2.25.1 Description

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

If hekad's config file is specified to be a directory, all contained files with a filename ending in ".toml" will be loaded and merged into a single config. Files that don't end with ".toml" will be ignored. Merging will happen in alphabetical order, settings specified later in the merge sequence will win conflicts.

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"
splitter = "HekaFramingSplitter"
```

```
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"
splitter = "HekaFramingSplitter"
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*).

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.

Full documentation on available plugins and settings for each one are in the hekad.plugin(5) pages.

2.25.2 Example hekad.toml File

```
[hekad]
maxprocs = 4

# Heka dashboard for internal metrics and time series graphs
[Dashboard]
type = "DashboardOutput"
address = ":4352"
ticker_interval = 15

# Email alerting for anomaly detection
[Alert]
type = "SmtpOutput"
message_matcher = "Type == 'heka.sandbox-output' && Fields[payload_type] == 'alert'"
send_from = "acme-alert@example.com"
send_to = ["admin@example.com"]
auth = "Plain"
user = "smtp-user"
password = "smtp-pass"
host = "mail.example.com:25"
encoder = "AlertEncoder"

# User friendly formatting of alert messages
[AlertEncoder]
type = "SandboxEncoder"
filename = "lua_encoders/alert.lua"
```

```
# Nginx access log reader
[AcmeWebserver]
type = "LogstreamerInput"
log_directory = "/var/log/nginx"
file_match = 'access\.log'
decoder = "CombinedNginxDecoder"

# Nginx access 'combined' log parser
[CombinedNginxDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/nginx_access.lua"

    [CombinedNginxDecoder.config]
    user_agent_transform = true
    user_agent_conditional = true
    type = "combined"
    log_format = '$remote_addr - $remote_user [$time_local] "$request" $status $body_bytes_sent "$htt

# Collection and visualization of the HTTP status codes
[AcmeHTTPStatus]
type = "SandboxFilter"
filename = "lua_filters/http_status.lua"
ticker_interval = 60
preserve_data = true
message_matcher = "Logger == 'AcmeWebserver'"

    # rate of change anomaly detection on column 1 (HTTP 200)
    [AcmeHTTPStatus.config]
    anomaly_config = 'roc("HTTP Status", 1, 15, 0, 1.5, true, false)'
```

2.25.3 Configuring Restarting Behavior

Plugins that support being restarted have a set of options that govern how a restart is handled if they exit with an error. If preferred, the plugin can be configured to not restart, or it could be restarted only 100 times, or restart attempts can proceed forever. Once the *max_retries* have been exceeded the plugin will be unregistered, potentially triggering hekad to shutdown (depending on the plugin's *can_exit* configuration).

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

Config:

- **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 exiting the plugin. 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). Defaults to -1.

Example:

```
[AMQPOutput]
url = "amqp://guest:guest@rabbitmq/"
```

```
exchange = "testout"
exchange_type = "fanout"
message_matcher = 'Logger == "TestWebserver"'

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

2.25.4 See Also

hekad(1), hekad.plugin(5)

2.26 hekad

2.26.1 Description

Available hekad plugins compiled with this version of hekad.

2.27 Inputs

2.27.1 Common Input Parameters

New in version 0.9.

There are some configuration options that are universally available to all Heka input 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.

- **decoder (string, optional):** Decoder to be used by the input. This should refer to the name of a registered decoder plugin configuration. If supplied, messages will be decoded before being passed on to the router when the InputRunner's *Deliver* method is called.
- **synchronous_decode (bool, optional):** If *synchronous_decode* is false, then any specified decoder plugin will be run by a DecoderRunner in its own goroutine and messages will be passed in to the decoder over a channel, freeing the input to start processing the next chunk of incoming or available data. If true, then any decoding will happen synchronously and message delivery will not return control to the input until after decoding has completed. Defaults to false.
- **send_decode_failures (bool, optional):** If false, then if an attempt to decode a message fails then Heka will log an error message and then drop the message. If true, then in addition to logging an error message, decode failure will cause the original, undecoded message to be tagged with a *decode_failure* field (set to true) and delivered to the router for possible further processing.
- **can_exit (bool, optional):** If false, the input plugin exiting will trigger a Heka shutdown. If set to true, Heka will continue processing other plugins. Defaults to false on most inputs.
- **splitter (string, optional)** Splitter to be used by the input. This should refer to the name of a registered splitter plugin configuration. It specifies how the input should split the incoming data stream into individual records prior to decoding and/or injection to the router. Typically defaults to "NullSplitter", although certain inputs override this with a different default value.

2.27.2 AMQP Input

Plugin Name: **AMQPInput**

Connects to a remote AMQP broker (RabbitMQ) and retrieves messages from the specified queue. As AMQP is dynamically programmable, the broker topology needs to be specified in the plugin configuration.

Config:

- **url (string):** An AMQP connection string formatted per the RabbitMQ URI Spec.
- **exchange (string):** AMQP exchange name
- **exchange_type (string):** AMQP exchange type (*fanout*, *direct*, *topic*, or *headers*).
- **exchange_durability (bool):** Whether the exchange should be configured as a durable exchange. Defaults to non-durable.
- **exchange_auto_delete (bool):** Whether the exchange is deleted when all queues have finished and there is no publishing. Defaults to auto-delete.
- **routing_key (string):** The message routing key used to bind the queue to the exchange. Defaults to empty string.
- **prefetch_count (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.
- **queue_durability (bool):** Whether the queue is durable or not. Defaults to non-durable.
- **queue_exclusive (bool):** Whether the queue is exclusive (only one consumer allowed) or not. Defaults to non-exclusive.
- **queue_auto_delete (bool):** Whether the queue is deleted when the last consumer un-subscribes. Defaults to auto-delete.
- **queue_ttl (int):** Allows ability to specify TTL in milliseconds on Queue declaration for expiring messages. Defaults to undefined/infinite.
- **retries (RetryOptions, optional):** A sub-section that specifies the settings to be used for restart behavior. See *Configuring Restarting Behavior*

New in version 0.6.

- **tls (TlsConfig):** An optional sub-section that specifies the settings to be used for any SSL/TLS encryption. This will only have any impact if *URL* uses the *AMQPS* URI scheme. See *Configuring TLS*.

New in version 0.9.

- **read_only (bool):** Whether the AMQP user is read-only. If this is true the exchange, queue and binding must be declared before starting Heka. Defaults to false.

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"
exchange_type = "fanout"
```

Or you might use a PayloadRegexDecoder to parse OSX syslog messages with the following:

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

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

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

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

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

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

2.27.3 Docker Event Input

New in version 0.10.0.

Plugin Name: **DockerEventInput**

The DockerEventInput plugin connects to the Docker daemon and watches the Docker events API, sending all events to the Heka pipeline. See: Docker Events API Messages will be populated as follows:

- Uuid: Type 4 (random) UUID generated by Heka.
- Timestamp: Time when the event was received by the plugin.
- Type: *DockerEvent*.
- Hostname: Hostname of the machine on which Heka is running.
- Payload: The event id, status, from and time. Example: *id:47e08ded0abb57ca263136291f14ed7689de8b6ec519f01ea76958fe512abff9 status:create from:gliderlabs/alpine:3.1 time:1429555298*
- Logger: The id provided in the event
- Fields["ID"] (string): The ID provided in the docker event.
- Fields["Status"] (string): The Status in the docker event.
- Fields["From"] (string): The From in the docker event.
- Fields["Time"] (string): The timestamp in the docker event.

Config:

- **endpoint (string):** A Docker endpoint. Defaults to "unix:///var/run/docker.sock".
- **cert_path (string, optional):** Path to directory containing client certificate and keys. This value works in the same way as DOCKER_CERT_PATH.

Example:

```
[DockerEventInput]
type = "DockerEventInput"

[PayloadEncoder]
append_newlines = false

[LogOutput]
type = "LogOutput"
message_matcher = "Type == 'DockerEvent'"
encoder = "PayloadEncoder"
```

2.27.4 Docker Log Input

New in version 0.8.

Plugin Name: **DockerLogInput**

The DockerLogInput plugin attaches to all containers running on a host and sends their logs messages into the Heka pipeline. The plugin is based on Logspout by Jeff Lindsay. Messages will be populated as follows:

- Uuid: Type 4 (random) UUID generated by Heka.
- Timestamp: Time when the log line was received by the plugin.
- Type: *DockerLog*.
- Hostname: Hostname of the machine on which Heka is running.
- Payload: The log line received from a Docker container.
- Logger: *stdout* or *stderr*, depending on source.
- Fields["ContainerID"] (string): The container ID.
- Fields["ContainerName"] (string): The container name.

Note: Logspout expects to be dealing exclusively with textual log file data, and always assumes that the file data is newline delimited, i.e. one line in the log file equals one logical unit of data. For this reason, the DockerLogInput currently does *not* support the use of alternate splitter plugins. Any splitter setting specified in a DockerLogInput's configuration will be ignored.

Config:

- **endpoint (string):** A Docker endpoint. Defaults to "unix:///var/run/docker.sock".
- **decoder (string):** The name of the decoder used to further transform the message into a structured hekad message. No default decoder is specified.

New in version 0.9.

- **cert_path (string, optional):** Path to directory containing client certificate and keys. This value works in the same way as DOCKER_CERT_PATH.

New in version 0.10.

- **name_from_env_var (string, optional):** Overwrite the ContainerName with this environment variable on the Container if exists. If left empty the container name will still be used.
- **fields_from_env (array[string], optional):** A list of environment variables to extract from the container and add as fields.

Example:

```
[nginx_log_decoder]
type = "SandboxDecoder"
filename = "lua_decoders/nginx_access.lua"

[nginx_log_decoder.config]
type = "nginx.access"
user_agent_transform = true
log_format = '$remote_addr - $remote_user [$time_local] "$request" $status $body_bytes_sent "$http_re

[DockerLogInput]
decoder = "nginx_log_decoder"
fields_from_env = [ "MESOS_TASK_ID" ]
```

2.27.5 File Polling Input

New in version 0.7.

Plugin Name: **FilePollingInput**

FilePollingInputs periodically read (unbuffered) the contents of a file specified, and creates a Heka message with the contents of the file as the payload.

Config:

- **file_path(string):** The absolute path to the file which the input should read.
- **ticker_interval (unit):** How often, in seconds to input should read the contents of the file.

Example:

```
[MemStats]
type = "FilePollingInput"
ticker_interval = 1
file_path = "/proc/meminfo"
decoder = "MemStatsDecoder"
```

2.27.6 HTTP Input

Plugin Name: **HttpInput**

HttpInput plugins intermittently poll remote HTTP URLs for data and populate message objects based on the results of the HTTP interactions. Messages will be populated as follows:

- Uuid: Type 4 (random) UUID generated by Heka.
- Timestamp: Time HTTP request is completed.
- **Type:** ***heka.httpinput.data*** **or** ***heka.httpinput.error*** **depending on whether or** not the request completed. (Note that a response returned with an HTTP error code is still considered complete and will generate type *heka.httpinput.data*.)
- Hostname: Hostname of the machine on which Heka is running.

- Payload: Entire contents of the HTTP response body.
- **Severity: HTTP response 200 uses *success_severity* config value, all other** results use *error_severity* config value.
- Logger: Fetched URL.
- Fields["Status"] (string): HTTP status string value (e.g. "200 OK").
- Fields["StatusCode"] (int): HTTP status code integer value.
- Fields["ResponseSize"] (int): Value of HTTP Content-Length header.
- **Fields["ResponseTime"] (float64): Clock time elapsed for HTTP request, in** seconds.
- **Fields["Protocol"] (string): HTTP protocol used for the request (e.g.** "HTTP/1.0")

The *Fields* values above will only be populated in the event of a completed HTTP request. Also, it is possible to specify a decoder to further process the results of the HTTP response before injecting the message into the router.

Config:

- **url (string):** A HTTP URL which this plugin will regularly poll for data. This option cannot be used with the urls option. No default URL is specified.
- **urls (array):** New in version 0.5.

  An array of HTTP URLs which this plugin will regularly poll for data. This option cannot be used with the url option. No default URLs are specified.
- **method (string):** New in version 0.5.

  The HTTP method to use for the request. Defaults to "GET".
- **headers (subsection):** New in version 0.5.

  Subsection defining headers for the request. By default the User-Agent header is set to "Heka"
- **body (string):** New in version 0.5.

  The request body (e.g. for an HTTP POST request). No default body is specified.
- **username (string):** New in version 0.5.

  The username for HTTP Basic Authentication. No default username is specified.
- **password (string):** New in version 0.5.

  The password for HTTP Basic Authentication. No default password is specified.
- **ticker_interval (uint):** Time interval (in seconds) between attempts to poll for new data. Defaults to 10.
- **success_severity (uint):** New in version 0.5.

  Severity level of successful HTTP request. Defaults to 6 (information).
- **error_severity (uint):** New in version 0.5.

  Severity level of errors, unreachable connections, and non-200 responses of successful HTTP requests. Defaults to 1 (alert).

Example:

```
[HttpInput]
url = "http://localhost:9876/"
ticker_interval = 5
success_severity = 6
error_severity = 1
decoder = "MyCustomJsonDecoder"
```

```
    [HttpInput.headers]
    user-agent = "MyCustomUserAgent"
```

2.27.7 HTTP Listen Input

New in version 0.5.

Plugin Name: **HttpListenInput**

HttpListenInput plugins start a webserver listening on the specified address and port. If no decoder is specified data in the request body will be populated as the message payload. Messages will be populated as follows:

- Uuid: Type 4 (random) UUID generated by Heka.
- Timestamp: Time HTTP request is handled.
- Type: *heka.httpdata.request*
- Hostname: The remote network address of requester.
- Payload: Entire contents of the HTTP request body.
- Severity: 6
- Logger: HttpListenInput
- Fields["UserAgent"] (string): Request User-Agent header (e.g. "GitHub Hookshot dd0772a").
- Fields["ContentType"] (string): Request Content-Type header (e.g. "application/x-www-form-urlencoded").
- **Fields["Protocol"] (string): HTTP protocol used for the request (e.g.** "HTTP/1.0")

New in version 0.6.

All query parameters are added as fields. For example, a request to "127.0.0.1:8325?user=bob" will create a field "user" with the value "bob".

Config:

- **address (string):** An IP address:port on which this plugin will expose a HTTP server. Defaults to "127.0.0.1:8325".

New in version 0.7.

- **headers (subsection, optional):** It is possible to inject arbitrary HTTP headers into each outgoing response by adding a TOML subsection entitled "headers" to you HttpOutput config section. All entries in the subsection must be a list of string values.

New in version 0.9.

- **request_headers ([]string):** Add additional request headers as message fields. Defaults to empty list.

New in version 0.10.

- **auth_type (string, optional):** If requiring Authentication specify "Basic" or "API" To use "API" you must set a header called "X-API-KEY" with the value of the "api_key" config.
- **username (string, optional):** Username to check against if auth_type = "Basic".
- **password (string, optional):** Password to check against if auth_type = "Basic".
- **api_key (string, optional):** String to validate the "X-API-KEY" header against when using auth_type = "API"
- **use_tls (bool):** Specifies whether or not SSL/TLS encryption should be used for the TCP connections. Defaults to false.

- **tls (TlsConfig):** A sub-section that specifies the settings to be used for any SSL/TLS encryption. This will only have any impact if *use_tls* is set to true. See *Configuring TLS*.

Example:

```
[HttpListenInput]
address = "0.0.0.0:8325"
```

With Basic Auth:

```
[HttpListenInput]
address = "0.0.0.0:8325"
auth_type = "Basic"
username = "foo"
password = "bar"
```

With API Key Auth:

```
[HttpListenInput]
address = "0.0.0.0:8325"
auth_type = "API"
api_key = "1234567"
```

2.27.8 Kafka Input

Plugin Name: **KafkaInput**

Connects to a Kafka broker and subscribes to messages from the specified topic and partition.

Config:

- **id (string)** Client ID string. Default is the hostname.
- **addrs ([]string)** List of brokers addresses.
- **metadata_retries (int)** How many times to retry a metadata request when a partition is in the middle of leader election. Default is 3.
- **wait_for_election (uint32)** How long to wait for leader election to finish between retries (in milliseconds). Default is 250.
- **background_refresh_frequency (uint32)** How frequently the client will refresh the cluster metadata in the background (in milliseconds). Default is 600000 (10 minutes). Set to 0 to disable.
- **max_open_reqests (int)** How many outstanding requests the broker is allowed to have before blocking attempts to send. Default is 4.
- **dial_timeout (uint32)** How long to wait for the initial connection to succeed before timing out and returning an error (in milliseconds). Default is 60000 (1 minute).
- **read_timeout (uint32)** How long to wait for a response before timing out and returning an error (in milliseconds). Default is 60000 (1 minute).
- **write_timeout (uint32)** How long to wait for a transmit to succeed before timing out and returning an error (in milliseconds). Default is 60000 (1 minute).
- **topic (string)** Kafka topic (must be set).
- **partition (int32)** Kafka topic partition. Default is 0.
- **group (string)** A string that uniquely identifies the group of consumer processes to which this consumer belongs. By setting the same group id multiple processes indicate that they are all part of the same consumer group. Default is the *id*.

- **default_fetch_size (int32)** The default (maximum) amount of data to fetch from the broker in each request. The default is 32768 bytes.
- **min_fetch_size (int32)** The minimum amount of data to fetch in a request - the broker will wait until at least this many bytes are available. The default is 1, as 0 causes the consumer to spin when no messages are available.
- **max_message_size (int32)** The maximum permittable message size - messages larger than this will return MessageTooLarge. The default of 0 is treated as no limit.
- **max_wait_time (uint32)** The maximum amount of time the broker will wait for min_fetch_size bytes to become available before it returns fewer than that anyways. The default is 250ms, since 0 causes the consumer to spin when no events are available. 100-500ms is a reasonable range for most cases.
- **offset_method (string)** The method used to determine at which offset to begin consuming messages. The valid values are:
  - *Manual* Heka will track the offset and resume from where it last left off (default).
  - *Newest* Heka will start reading from the most recent available offset.
  - *Oldest* Heka will start reading from the oldest available offset.
- **event_buffer_size (int)** The number of events to buffer in the Events channel. Having this non-zero permits the consumer to continue fetching messages in the background while client code consumes events, greatly improving throughput. The default is 16.

Example 1: Read Fxa messages from partition 0.

```
[FxaKafkaInputTest]
type = "KafkaInput"
topic = "Fxa"
addrs = ["localhost:9092"]
```

Example 2: Send messages between two Heka instances via a Kafka broker.

```
# On the producing instance
[KafkaOutputExample]
type = "KafkaOutput"
message_matcher = "TRUE"
topic = "heka"
addrs = ["kafka-broker:9092"]
encoder = "ProtobufEncoder"
```

```
# On the consuming instance
[KafkaInputExample]
type = "KafkaInput"
topic = "heka"
addrs = ["kafka-broker:9092"]
splitter = "KafkaSplitter"
decoder = "ProtobufDecoder"

[KafkaSplitter]
type = "NullSplitter"
use_message_bytes = true
```

2.27.9 Logstreamer Input

New in version 0.5.

Plugin Name:: **LogstreamerInput**

Tails a single log file, a sequential single log source, or multiple log sources of either a single logstream or multiple logstreams.

See also:

Complete documentation with examples

Config:

- **hostname (string):** The hostname to use for the messages, by default this will be the machine's qualified hostname. This can be set explicitly to ensure it's the correct name in the event the machine has multiple interfaces/hostnames.
- **oldest_duration (string):** A time duration string (e.x. "2s", "2m", "2h"). Logfiles with a last modified time older than `oldest_duration` ago will not be included for parsing.
- **journal_directory (string):** The directory to store the journal files in for tracking the location that has been read to thus far. By default this is stored under heka's base directory.
- **log_directory (string):** The root directory to scan files from. This scan is recursive so it should be suitably restricted to the most specific directory this selection of logfiles will be matched under. The log_directory path will be prepended to the file_match.
- **rescan_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.
- **file_match (string):** Regular expression used to match files located under the `log_directory`. This regular expression has `$` added to the end automatically if not already present, and `log_directory` as the prefix. WARNING: file_match should typically be delimited with single quotes, indicating use of a raw string, rather than double quotes, which require all backslashes to be escaped. For example, *'access\.log'* will work as expected, but *"access\.log"* will not, you would need *"access\\.log"* to achieve the same result.
- **priority (list of strings):** When using sequential logstreams, the priority is how to sort the logfiles in order from oldest to newest.
- **differentiator (list of strings):** When using multiple logstreams, the differentiator is a set of strings that will be used in the naming of the logger, and portions that match a captured group from the `file_match` will have their matched value substituted in.
- **translation (hash map of hash maps of ints):** A set of translation mappings for matched groupings to the ints to use for sorting purposes.
- **splitter (string, optional):** Defaults to "TokenSplitter", which will split the log stream into one Heka message per line.

2.27.10 Process Input

Plugin Name: **ProcessInput**

Executes one or more external programs on an interval, creating messages from the output. Supports a chain of commands, where stdout from each process will be piped into the stdin for the next process in the chain. ProcessInput creates Fields[ExitStatus] and Fields[SubcmdErrors]. Fields[ExitStatus] represents the platform dependent exit status of the last command in the command chain. Fields[SubcmdErrors] represents errors from each sub command, in the format of "Subcommand[<subcommand ID>] returned an error: <error message>".

Config:

- **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 below). ProcessInput expects the commands to be indexed by integers starting with 0, where 0 is the first process in the chain.
- **ticker_interval (uint):** The number of seconds to wait between each run of *command*. Defaults to 15. A ticker_interval of 0 indicates that the command is run only once, and should only be used for long running processes that do not exit. If ticker_interval is set to 0 and the process exits, then the ProcessInput will exit, invoking the restart behavior (see *Configuring Restarting Behavior*). Ignored when used in conjunction with *Process Directory Input*, where *ticker_interval* value is instead parsed from the directory path.
- **immediate_start (bool):** If true, heka starts process immediately instead of waiting for first interval defined by ticker_interval to pass. Defaults to false.
- **stdout (bool):** If true, for each run of the process chain a message will be generated with the last command in the chain's stdout as the payload. Defaults to true.
- **stderr (bool):** If true, for each run of the process chain a message will be generated with the last command in the chain's stderr as the payload. Defaults to false.
- **timeout (uint):** Timeout in seconds before any one of the commands in the chain is terminated.
- **retries (RetryOptions, optional):** A sub-section that specifies the settings to be used for restart behavior. See *Configuring Restarting Behavior*

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.
- **env ([]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.

Example:

```
[on_space]
type = "TokenSplitter"
delimiter = " "

[DemoProcessInput]
type = "ProcessInput"
ticker_interval = 2
splitter = "on_space"
stdout = true
stderr = false

    [DemoProcessInput.command.0]
    bin = "/bin/cat"
    args = ["../testsupport/process_input_pipes_test.txt"]

    [DemoProcessInput.command.1]
    bin = "/usr/bin/grep"
    args = ["ignore"]
```

2.27.11 Process Directory Input

Plugin Name: **ProcessDirectoryInput**

New in version 0.5.

The ProcessDirectoryInput periodically scans a filesystem directory looking for ProcessInput configuration files. The ProcessDirectoryInput will maintain a pool of running ProcessInputs based on the contents of this directory, refreshing the set of running inputs as needed with every rescan. This allows Heka administrators to manage a set of data collection processes for a running hekad server without restarting the server.

Each ProcessDirectoryInput has a *process_dir* configuration setting, which is the root folder of the tree where scheduled jobs are defined. It should contain exactly one nested level of subfolders, named with ASCII numeric characters indicating the interval, in seconds, between each process run. These numeric folders must contain TOML files which specify the details regarding which processes to run.

For example, a process_dir might look like this:

```
-/usr/share/heka/processes/
 |-5
   |- check_myserver_running.toml
 |-61
   |- cat_proc_mounts.toml
   |- get_running_processes.toml
 |-302
   |- some_custom_query.toml
```

This indicates one process to be run every five seconds, two processes to be run every 61 seconds, and one process to be run every 302 seconds.

Note that ProcessDirectoryInput will ignore any files that are not nested one level deep, are not in a folder named for an integer 0 or greater, and do not end with '.toml'. Each file which meets these criteria, such as those shown in the example above, should contain the TOML configuration for exactly one *Process Input*, matching that of a standalone ProcessInput with the following restrictions:

- The section name *must* be *ProcessInput*. Any TOML sections named anything other than ProcessInput will be ignored.
- Any specified *ticker_interval* value will be *ignored*. The ticker interval value to use will be parsed from the directory path.

By default, if the specified process fails to run or the ProcessInput config fails for any other reason, ProcessDirectoryInput will log an error message and continue, as if the ProcessInput's *can_exit* flag has been set to true. If the managed ProcessInput's *can_exit* flag is manually set to *false*, it will trigger a Heka shutdown.

Config:

- **ticker_interval (int, optional):** Amount of time, in seconds, between scans of the process_dir. Defaults to 300 (i.e. 5 minutes).
- **process_dir (string, optional):** This is the root folder of the tree where the scheduled jobs are defined. Absolute paths will be honored, relative paths will be computed relative to Heka's globally specified share_dir. Defaults to "processes" (i.e. "$share_dir/processes").
- **retries (RetryOptions, optional):** A sub-section that specifies the settings to be used for restart behavior of the ProcessDirectoryInput (not the individual ProcessInputs, which are configured independently). See *Configuring Restarting Behavior*

Example:

```
[ProcessDirectoryInput]
process_dir = "/etc/hekad/processes.d"
ticker_interval = 120
```

2.27.12 Sandbox Input

New in version 0.9.

Plugin Name: **SandboxInput**

The SandboxInput provides a flexible execution environment for data ingestion and transformation without the need to recompile Heka. Like all other sandboxes it needs to implement a process_message function. However, it doesn't have to return until shutdown. If you would like to implement a polling interface process_message can return zero when complete and it will be called again the next time TickerInterval fires (if ticker_interval was set to zero it would simply exit after running once). See *Sandbox*. Config:

- All of the common input configuration parameters are ignored since the data processing (splitting and decoding) should happen in the plugin.
- ***Common Sandbox Parameters***
  - `instruction_limit` is always set to zero for SandboxInputs

Example

```
[MemInfo]
type = "SandboxInput"
filename = "meminfo.lua"

[MemInfo.config]
path = "/proc/meminfo"
```

2.27.13 Stat Accumulator Input

Plugin Name: **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.

Config:

- **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".
- **legacy_namespaces (bool):** If set to true, then use the older format for namespacing counter stats, with rates recorded under *stats.*<*counter_name*> and absolute count recorded under *stats_counts.*<*counter_name*>. See statsd metric namespacing. Defaults to false.
- **global_prefix (string):** Global prefix to use for sending stats to graphite. Defaults to "stats".
- **counter_prefix (string):** Secondary prefix to use for namespacing counter metrics. Has no impact unless *legacy_namespaces* is set to false. Defaults to "counters".

- **timer_prefix (string):** Secondary prefix to use for namespacing timer metrics. Defaults to "timers".
- **gauge_prefix (string):** Secondary prefix to use for namespacing gauge metrics. Defaults to "gauges".
- **statsd_prefix (string):** Prefix to use for the statsd *numStats* metric. Defaults to "statsd".
- **delete_idle_stats (bool):** Don't emit values for inactive stats instead of sending 0 or in the case of gauges, sending the previous value. Defaults to false.

Example:

```
[StatAccumInput]
emit_in_fields = true
delete_idle_stats = true
ticker_interval = 5
```

2.27.14 Statsd Input

Plugin Name: **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.

Config:

- **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".
- **max_msg_size (uint):** Size of a buffer used for message read from statsd. In some cases, when statsd sends a lots in single message of stats it's required to boost this value. All over-length data will be truncated without raising an error. Defaults to 512.

Example:

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

2.27.15 TCP Input

Plugin Name: **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.

Config:

- **address (string):** An IP address:port on which this plugin will listen.

New in version 0.4.

- **decoder (string):** Defaults to "ProtobufDecoder".

New in version 0.5.

- **use_tls (bool):** Specifies whether or not SSL/TLS encryption should be used for the TCP connections. Defaults to false.

- **tls (TlsConfig):** A sub-section that specifies the settings to be used for any SSL/TLS encryption. This will only have any impact if *use_tls* is set to true. See *Configuring TLS*.
- **net (string, optional, default: "tcp")** Network value must be one of: "tcp", "tcp4", "tcp6", "unix" or "unixpacket".

New in version 0.6.

- **keep_alive (bool):** Specifies whether or not TCP keepalive should be used for established TCP connections. Defaults to false.
- **keep_alive_period (int):** Time duration in seconds that a TCP connection will be maintained before keepalive probes start being sent. Defaults to 7200 (i.e. 2 hours).

New in version 0.9.

- **splitter (string):** Defaults to "HekaFramingSplitter".

Example:

```
[TcpInput]
address = ":5565"
```

2.27.16 UDP Input

Plugin Name: **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.

Config:

- **address (string):** An IP address:port or Unix datagram socket file path 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 0.5.

- **net (string, optional, default: "udp")** Network value must be one of: "udp", "udp4", "udp6", or "unixgram".

Example:

```
[UdpInput]
address = "127.0.0.1:4880"
splitter = "HekaFramingSplitter"
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"
```

New in version 0.9.

2.28 Splitters

:

2.28.1 Common Splitter Parameters

There are some configuration options that are universally available to all Heka splitter 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.

- **keep_truncated (bool, optional):** If true, then any records that exceed the capacity of the input buffer will still be delivered in their truncated form. If false, then these records will be dropped. Defaults to false.
- **use_message_bytes (bool, optional):** Most decoders expect to find the raw, undecoded input data stored as the payload of the received Heka Message struct. Some decoders, however, such as the ProtobufDecoder, expect to receive a blob of bytes representing an entire Message struct, not just the payload. In this case, the data is expected to be found on the MsgBytes attribute of the Message's PipelinePack. If *use_message_bytes* is true, then the data will be written as a byte slice to the MsgBytes attribute, otherwise it will be written as a string to the Message payload. Defaults to false in most cases, but defaults to true for the HekaFramingSplitter, which is almost always used with the ProtobufDecoder.
- **min_buffer_size (uint, optional):** The initial size, in bytes, of the internal buffer that the SplitterRunner will use for buffering data streams. Must not be greater than the globally configured max_message_size. Defaults to 8KiB, although certain splitters may specify a different default.
- **deliver_incomplete_final (bool, optional):** When a splitter is used to split a stream, that stream can end part way through a record. It's sometimes appropriate to drop that data, but in other cases the incomplete data can still be useful. If 'deliver_incomplete_final' is set to true, then when the SplitterRunner's SplitStream method is used a delivery attempt will be made with any partial record data that may come through immediately before an EOF. Defaults to false.

2.28.2 Heka Framing Splitter

Plugin Name: **HekaFramingSplitter**

A HekaFramingSplitter is used to split streams of data that use Heka's built- in *Stream Framing*, with a protocol buffers encoded message header supporting HMAC key authentication.

A default configuration of the HekaFramingSplitter is automatically registered as an available splitter plugin as "HekaFramingSplitter", so it is only necessary to add an additional TOML section if you want to use an instance of the splitter with settings other than the default.

Config:

- **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.
- **use_message_bytes (bool, optional):** The HekaFramingSplitter is almost always used in concert with an instance of ProtobufDecoder, which expects the protocol buffer message data to be available in the PipelinePack's MsgBytes attribute, so *use_message_bytes* defaults to true.

- **skip_authentication (bool, optional):** Usually if a HekaFramingSplitter identifies an incorrectly signed message, that message will be silently dropped. In some cases, however, such as when loading a stream of protobuf encoded Heka messages from a file system file, it may be desirable to skip authentication altogether. Setting this to true will do so. Defaults to false.

Example:

```
[acl_splitter]
type = "HekaFramingSplitter"

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

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

[tcp_control]
type = "TcpInput"
address = ":5566"
splitter = "acl_splitter"
```

2.28.3 Null Splitter

Plugin Name: **NullSplitter**

The NullSplitter is used in cases where the incoming data is already naturally divided into logical messages, such that Heka doesn't need to do any further splitting. For instance, when used in conjunction with a UdpInput, the contents of each UDP packet will be made into a separate message.

Note that this means generally the NullSplitter should not be used with a stream oriented input transport, such as with TcpInput or LogstreamerInput. If this is done then the splitting will be arbitrary, each message will contain whatever happens to be the contents of a particular read operation.

The NullSplitter has no configuration options, and is automatically registered as an available splitter plugin of the name "NullSplitter", so it doesn't require a separate TOML configuration section.

2.28.4 Regex Splitter

Plugin Name: **RegexSplitter**

A RegexSplitter considers any text that matches a specified regular expression to represent a boundary on which records should be split. The regular expression may consist of exactly one capture group. If a capture group is specified, then the captured text will be included in the returned record. If not, then the returned record will not include the text that caused the regular expression match.

Config:

- **delimiter (string)** Regular expression to be used as the record boundary. May contain zero or one specified capture groups.
- **delimiter_eol (bool, optional):** Specifies whether the contents of a delimiter capture group should be appended to the end of a record (true) or prepended to the beginning (false). Defaults to true. If the delimiter expression does not specify a capture group, this will have no effect.

Example:

```
[mysql_slow_query_splitter]
type = "RegexSplitter"
delimiter = '\n(# User@Host:)'
delimiter_eol = false
```

2.28.5 Token Splitter

Plugin Name: **TokenSplitter**

A TokenSplitter is used to split an incoming data stream on every occurrence (or every Nth occurrence) of a single, one byte token character. The token will be included as the final character in the returned record.

A default configuration of the TokenSplitter (i.e. splitting on every newline) is automatically registered as an available splitter plugin as "TokenSplitter", so additional TOML sections don't need to be added unless you want to use different settings.

Config:

- **delimiter (string, optional):** String representation of the byte token to be used as message delimiter. Defaults to *"\n"*.
- **count (uint, optional):** Number of instances of the delimiter that should be encountered before returning a record. Defaults to 1. Setting to 0 has no effect, 0 and 1 will be treated identically. Often used in conjunction with the *deliver_incomplete_final* option set to true, to ensure trailing partial records are still delivered.

Example:

```
[split_on_space]
type = "TokenSplitter"
delimiter = " "

[split_every_50th_newline_keep_partial]
type = "TokenSplitter"
count = 50
deliver_incomplete_final = true
```

2.29 Decoders

2.29.1 Apache Access Log Decoder

New in version 0.6.

Plugin Name: **SandboxDecoder**
File Name: **lua_decoders/apache_access.lua**

Parses the Apache access logs based on the Apache 'LogFormat' configuration directive. The Apache format specifiers are mapped onto the Nginx variable names where applicable e.g. %a -> remote_addr. This allows generic web filters and outputs to work with any HTTP server input.

Config:

- **log_format (string)** The 'LogFormat' configuration directive from the apache2.conf. %t variables are converted to the number of nanosecond since the Unix epoch and used to set the Timestamp on the message. http://httpd.apache.org/docs/2.4/mod/mod_log_config.html
- **type (string, optional, default nil):** Sets the message 'Type' header to the specified value
- **user_agent_transform (bool, optional, default false)** Transform the http_user_agent into user_agent_browser, user_agent_version, user_agent_os.
- **user_agent_keep (bool, optional, default false)** Always preserve the http_user_agent value if transform is enabled.
- **user_agent_conditional (bool, optional, default false)** Only preserve the http_user_agent value if transform is enabled and fails.
- **payload_keep (bool, optional, default false)** Always preserve the original log line in the message payload.

Example Heka Configuration

```
[TestWebserver]
type = "LogstreamerInput"
log_directory = "/var/log/apache"
file_match = 'access\.log'
decoder = "CombinedLogDecoder"

[CombinedLogDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/apache_access.lua"

[CombinedLogDecoder.config]
type = "combined"
user_agent_transform = true
# combined log format
log_format = '%h %l %u %t \"%r\" %>s %O \"%{Referer}i\" \"%{User-Agent}i\"'

# common log format
# log_format = '%h %l %u %t \"%r\" %>s %O'

# vhost_combined log format
# log_format = '%v:%p %h %l %u %t \"%r\" %>s %O \"%{Referer}i\" \"%{User-Agent}i\"'

# referer log format
# log_format = '%{Referer}i -> %U'
```

Example Heka Message

Timestamp 2014-01-10 07:04:56 -0800 PST

Type combined

Hostname test.example.com

Pid 0

UUID 8e414f01-9d7f-4a48-a5e1-ae92e5954df5

Logger TestWebserver

Payload

EnvVersion

Severity 7

Fields

name:"remote_user" value_string:"-"
name:"http_x_forwarded_for" value_string:"-"
name:"http_referer" value_string:"-"
name:"body_bytes_sent" value_type:DOUBLE representation:"B" value_double:82
name:"remote_addr" value_string:"62.195.113.219" representation:"ipv4"
name:"status" value_type:DOUBLE value_double:200
name:"request" value_string:"GET /v1/recovery_email/status HTTP/1.1"
name:"user_agent_os" value_string:"FirefoxOS"
name:"user_agent_browser" value_string:"Firefox"
name:"user_agent_version" value_type:DOUBLE value_double:29

2.29.2 Graylog Extended Log Format Decoder

New in version 0.8.

Plugin Name: **SandboxDecoder**
File Name: **lua_decoders/graylog_extended.lua**

Parses a payload containing JSON in the Graylog2 Extended Format specficiation. http://graylog2.org/resources/gelf/specification

Config:

- **type (string, optional, default nil):** Sets the message 'Type' header to the specified value
- **payload_keep (bool, optional, default false)** Always preserve the original log line in the message payload.

Example of Graylog2 Exteded Format Log

```
{
  "version": "1.1",
  "host": "rogueethic.com",
  "short_message": "This is a short message to identify what is going on.",
  "full_message": "An entire backtrace\ncould\ngo\nhere",
  "timestamp": 1385053862.3072,
  "level": 1,
  "_user_id": 9001,
  "_some_info": "foo",
  "_some_env_var": "bar"
}
```

Example Heka Configuration

```
[GELFLogInput]
type = "LogstreamerInput"
log_directory = "/var/log"
file_match = 'application\.gelf'
decoder = "GraylogDecoder"

[GraylogDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/graylog_decoder.lua"
```

```
[GraylogDecoder.config]
type = "gelf"
payload_keep = true
```

2.29.3 Geo IP Decoder

New in version 0.6.

Plugin Name: **GeoIpDecoder**

Decoder plugin that generates GeoIP data based on the IP address of a specified field. It uses the GeoIP Go project as a wrapper around MaxMind's geoip-api-c library, and thus assumes you have the library downloaded and installed. Currently, only the GeoLiteCity database is supported, which you must also download and install yourself into a location to be referenced by the db_file config option. By default the database file is opened using "GEOIP_MEMORY_CACHE" mode. This setting is hard- coded into the wrapper's geoip.go file. You will need to manually override that code if you want to specify one of the other modes listed here.

Note: Due to external dependencies, this plugin is not compiled in to the released Heka binaries. It will automatically be included in a *source build* if GeoIP.h is available in the include path during build time. The generated binary will then only work on machines with the appropriate GeoIP shared library (e.g. *libGeoIP.so.1*) installed.

Note: If you are using this with the ES output you will likely need to specify the raw_bytes_field option for the target_field specified. This is required to preserve the formatting of the JSON object.

Config:

- **db_file:** The location of the GeoLiteCity.dat database. Defaults to "/var/cache/hekad/GeoLiteCity.dat"
- **source_ip_field:** The name of the field containing the IP address you want to derive the location for.
- **target_field:** The name of the new field created by the decoder. The decoder will output a JSON object with the following elements:
  - latitute: string,
  - longitude: string,
  - **location: [float64, float64],**
    - GeoJSON format intended for use as a geo_point for ES output. Useful when using Kibana's Bettermap panel
  - coordinates: [string, string],
  - countrycode: string,
  - countrycode3: string,
  - region: string,
  - city: string,
  - postalcode: string,
  - areacode: int,
  - charset: int,
  - continentalcode: string

```
[apache_geoip_decoder]
type = "GeoIpDecoder"
db_file="/etc/geoip/GeoLiteCity.dat"
source_ip_field="remote_host"
target_field="geoip"
```

2.29.4 MultiDecoder

Plugin Name: **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.

Config:

- **subs ([]string):** An ordered list of subdecoders to which the MultiDecoder will delegate. Each item in the list should specify another decoder configuration section by section name. Must contain at least one entry.
- **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.

Here is a slightly contrived example where we have protocol buffer encoded messages coming in over a TCP connection, with each message containin a single nginx log line. Our MultiDecoder will run each message through two decoders, the first to deserialize the protocol buffer and the second to parse the log text:

```
[TcpInput]
address = ":5565"
parser_type = "message.proto"
decoder = "shipped-nginx-decoder"

[shipped-nginx-decoder]
type = "MultiDecoder"
subs = ['ProtobufDecoder', 'nginx-access-decoder']
cascade_strategy = "all"
log_sub_errors = true

[ProtobufDecoder]

[nginx-access-decoder]
type = "SandboxDecoder"
filename = "lua_decoders/nginx_access.lua"

    [nginx-access-decoder.config]
    type = "combined"
    user_agent_transform = true
    log_format = '$remote_addr - $remote_user [$time_local] "$request" $status $body_bytes_sent "$htt
```

2.29.5 Linux CPU Stats Decoder

New in version 0.10.

Plugin Name: **SandboxDecoder**
File Name: **lua_decoders/linux_procstat.lua**

Parses a payload containing the contents of file */proc/stat*.

Config:

- **payload_keep (bool, optional, default false)** Always preserve the original log line in the message payload.

Example Heka Configuration

```
[ProcStats]
type = "FilePollingInput"
ticker_interval = 1
file_path = "/proc/stat"
decoder = "ProcStatDecoder"

[ProcStatDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/linux_procstat.lua"
```

Example Heka Message

Timestamp 2014-12-10 22:38:24 +0000 UTC

Type stats.proc

Hostname yourhost.net

Pid 0

Uuid d2546942-7c36-4042-ad2e-f6bfdac11cdb

Logger

Payload

EnvVersion

Severity 7

Fields

name:"cpu" type:double value:[14384,125,3330,946000,333,0,356,0,0,0]
name:"cpu[1-#]" type:double value:[14384,125,3330,946000,333,0,356,0,0,0]
name:"ctxt" type:double value:2808304
name:"btime" type:double value:1423004780
name:"intr" type:double value:[14384,125,3330,0,0,0,0,0,0,0...0]
name:"processes" type:double value:3811
name:"procs_running" type:double value:1
name:"procs_blocked" type:double value:0
name:"softirq" type:double value:[288977,23,101952,19,13046,19217,7,...]

Cpu fields: 1 2 3 4 5 6 7 8 9 10 user nice system idle [iowait] [irq] [softirq] [steal] [guest] [guestnice]
Note: systems provide user, nice, system, idle. Other fields depend on kernel.

intr This line shows counts of interrupts serviced since boot time, for each of the possible system interrupts. The first column is the total of all interrupts serviced including unnumbered architecture specific interrupts; each subsequent column is the total for that particular numbered interrupt. Unnumbered interrupts are not shown, only summed into the total.

2.29.6 Linux Disk Stats Decoder

New in version 0.7.

Plugin Name: **SandboxDecoder**
File Name: **lua_decoders/linux_diskstats.lua**

Parses a payload containing the contents of a */sys/block/$DISK/stat* file (where *$DISK* is a disk identifier such as *sda*) into a Heka message struct. This also tries to obtain the TickerInterval of the input it recieved the data from, by extracting it from a message field named *TickerInterval*.

Config:

- **payload_keep (bool, optional, default false)** Always preserve the original log line in the message payload.

Example Heka Configuration

```
[DiskStats]
type = "FilePollingInput"
ticker_interval = 1
file_path = "/sys/block/sda1/stat"
decoder = "DiskStatsDecoder"

[DiskStatsDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/linux_diskstats.lua"
```

Example Heka Message

Timestamp 2014-01-10 07:04:56 -0800 PST

Type stats.diskstats

Hostname test.example.com

Pid 0

UUID 8e414f01-9d7f-4a48-a5e1-ae92e5954df5

Payload

EnvVersion

Severity 7

Fields

name:"ReadsCompleted" value_type:DOUBLE value_double:"20123"
name:"ReadsMerged" value_type:DOUBLE value_double:"11267"
name:"SectorsRead" value_type:DOUBLE value_double:"1.094968e+06"
name:"TimeReading" value_type:DOUBLE value_double:"45148"
name:"WritesCompleted" value_type:DOUBLE value_double:"1278"
name:"WritesMerged" value_type:DOUBLE value_double:"1278"
name:"SectorsWritten" value_type:DOUBLE value_double:"206504"
name:"TimeWriting" value_type:DOUBLE value_double:"3348"
name:"TimeDoingIO" value_type:DOUBLE value_double:"4876"
name:"WeightedTimeDoingIO" value_type:DOUBLE value_double:"48356"
name:"NumIOInProgress" value_type:DOUBLE value_double:"3"

name:"TickerInterval" value_type:DOUBLE value_double:"2"
name:"FilePath" value_string:"/sys/block/sda/stat"

2.29.7 Linux Load Average Decoder

New in version 0.7.

Plugin Name: **SandboxDecoder**
File Name: **lua_decoders/linux_loadavg.lua**

Parses a payload containing the contents of a */proc/loadavg* file into a Heka message.

Config:

- **payload_keep (bool, optional, default false)** Always preserve the original log line in the message payload.

Example Heka Configuration

```
[LoadAvg]
type = "FilePollingInput"
ticker_interval = 1
file_path = "/proc/loadavg"
decoder = "LoadAvgDecoder"

[LoadAvgDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/linux_loadavg.lua"
```

Example Heka Message

Timestamp 2014-01-10 07:04:56 -0800 PST

Type stats.loadavg

Hostname test.example.com

Pid 0

UUID 8e414f01-9d7f-4a48-a5e1-ae92e5954df5

Payload

EnvVersion

Severity 7

Fields

name:"1MinAvg" value_type:DOUBLE value_double:"3.05"
name:"5MinAvg" value_type:DOUBLE value_double:"1.21"
name:"15MinAvg" value_type:DOUBLE value_double:"0.44"
name:"NumProcesses" value_type:DOUBLE value_double:"11"
name:"FilePath" value_string:"/proc/loadavg"

2.29.8 Linux Memory Stats Decoder

New in version 0.7.

Plugin Name: **SandboxDecoder** File Name: **lua_decoders/linux_memstats.lua**

Parses a payload containing the contents of a */proc/meminfo* file into a Heka message.

Config:

- **payload_keep (bool, optional, default false)** Always preserve the original log line in the message payload.

Example Heka Configuration

```
[MemStats]
type = "FilePollingInput"
ticker_interval = 1
file_path = "/proc/meminfo"
decoder = "MemStatsDecoder"

[MemStatsDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/linux_memstats.lua"
```

Example Heka Message

Timestamp 2014-01-10 07:04:56 -0800 PST

Type stats.memstats

Hostname test.example.com

Pid 0

UUID 8e414f01-9d7f-4a48-a5e1-ae92e5954df5

Payload

EnvVersion

Severity 7

Fields

name:"MemTotal" value_type:DOUBLE representation:"kB" value_double:"4047616"
name:"MemFree" value_type:DOUBLE representation:"kB" value_double:"3432216"
name:"Buffers" value_type:DOUBLE representation:"kB" value_double:"82028"
name:"Cached" value_type:DOUBLE representation:"kB" value_double:"368636"
name:"FilePath" value_string:"/proc/meminfo"

The total available fields can be found in *man procfs*. All fields are of type double, and the representation is in kB (except for the HugePages fields). Here is a full list of fields available:

MemTotal, MemFree, Buffers, Cached, SwapCached, Active, Inactive, Active(anon), Inactive(anon), Active(file), Inactive(file), Unevictable, Mlocked, SwapTotal, SwapFree, Dirty, Writeback, AnonPages, Mapped, Shmem, Slab, SReclaimable, SUnreclaim, KernelStack, PageTables, NFS_Unstable, Bounce, WritebackTmp, CommitLimit, Committed_AS, VmallocTotal, VmallocUsed, VmallocChunk, HardwareCorrupted, AnonHugePages, HugePages_Total, HugePages_Free, HugePages_Rsvd, HugePages_Surp, Hugepagesize, DirectMap4k, DirectMap2M, DirectMap1G.

Note that your available fields may have a slight variance depending on the system's kernel version.

2.29.9 MySQL Slow Query Log Decoder

New in version 0.6.

Plugin Name: **SandboxDecoder**
File Name: **lua_decoders/mysql_slow_query.lua**

Parses and transforms the MySQL slow query logs. Use mariadb_slow_query.lua to parse the MariaDB variant of the MySQL slow query logs.

Config:

- **truncate_sql (int, optional, default nil)** Truncates the SQL payload to the specified number of bytes (not UTF-8 aware) and appends "...". If the value is nil no truncation is performed. A negative value will truncate the specified number of bytes from the end.

Example Heka Configuration

```
[Sync-1_5-SlowQuery]
type = "LogstreamerInput"
log_directory = "/var/log/mysql"
file_match = 'mysql-slow\.log'
parser_type = "regexp"
delimiter = "\n(# User@Host:)"
delimiter_location = "start"
decoder = "MySqlSlowQueryDecoder"

[MySqlSlowQueryDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/mysql_slow_query.lua"

    [MySqlSlowQueryDecoder.config]
    truncate_sql = 64
```

Example Heka Message

Timestamp 2014-05-07 15:51:28 -0700 PDT

Type mysql.slow-query

Hostname 127.0.0.1

Pid 0

UUID 5324dd93-47df-485b-a88e-429f0fcd57d6

Logger Sync-1_5-SlowQuery

Payload /* [queryName=FIND_ITEMS] */ SELECT bso.userid, bso.collection, ...

EnvVersion

Severity 7

Fields

name:"Rows_examined" value_type:DOUBLE value_double:16458
name:"Query_time" value_type:DOUBLE representation:"s" value_double:7.24966
name:"Rows_sent" value_type:DOUBLE value_double:5001
name:"Lock_time" value_type:DOUBLE representation:"s" value_double:0.047038

2.29.10 Nginx Access Log Decoder

New in version 0.5.

Plugin Name: **SandboxDecoder**
File Name: **lua_decoders/nginx_access.lua**

Parses the Nginx access logs based on the Nginx 'log_format' configuration directive.

Config:

- **log_format (string)** The 'log_format' configuration directive from the nginx.conf. $time_local or $time_iso8601 variable is converted to the number of nanosecond since the Unix epoch and used to set the Timestamp on the message. http://nginx.org/en/docs/http/ngx_http_log_module.html
- **type (string, optional, default nil):** Sets the message 'Type' header to the specified value
- **user_agent_transform (bool, optional, default false)** Transform the http_user_agent into user_agent_browser, user_agent_version, user_agent_os.
- **user_agent_keep (bool, optional, default false)** Always preserve the http_user_agent value if transform is enabled.
- **user_agent_conditional (bool, optional, default false)** Only preserve the http_user_agent value if transform is enabled and fails.
- **payload_keep (bool, optional, default false)** Always preserve the original log line in the message payload.

Example Heka Configuration

```
[TestWebserver]
type = "LogstreamerInput"
log_directory = "/var/log/nginx"
file_match = 'access\.log'
decoder = "CombinedLogDecoder"

[CombinedLogDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/nginx_access.lua"

[CombinedLogDecoder.config]
type = "combined"
user_agent_transform = true
# combined log format
log_format = '$remote_addr - $remote_user [$time_local] "$request" $status $body_bytes_sent "$http_re
```

Example Heka Message

Timestamp 2014-01-10 07:04:56 -0800 PST

Type combined

Hostname test.example.com

Pid 0

UUID 8e414f01-9d7f-4a48-a5e1-ae92e5954df5

Logger TestWebserver

Payload

EnvVersion

Severity 7

Fields

name:"remote_user" value_string:"-"
name:"http_x_forwarded_for" value_string:"-"
name:"http_referer" value_string:"-"
name:"body_bytes_sent" value_type:DOUBLE representation:"B" value_double:82
name:"remote_addr" value_string:"62.195.113.219" representation:"ipv4"
name:"status" value_type:DOUBLE value_double:200
name:"request" value_string:"GET /v1/recovery_email/status HTTP/1.1"
name:"user_agent_os" value_string:"FirefoxOS"
name:"user_agent_browser" value_string:"Firefox"
name:"user_agent_version" value_type:DOUBLE value_double:29

2.29.11 Nginx Error Log Decoder

New in version 0.6.

Plugin Name: **SandboxDecoder**
File Name: **lua_decoders/nginx_error.lua**

Parses the Nginx error logs based on the Nginx hard coded internal format.

Config:

- **tz (string, optional, defaults to UTC)** The conversion actually happens on the Go side since there isn't good TZ support here.

Example Heka Configuration

```
[TestWebserverError]
type = "LogstreamerInput"
log_directory = "/var/log/nginx"
file_match = 'error\.log'
decoder = "NginxErrorDecoder"

[NginxErrorDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/nginx_error.lua"

[NginxErrorDecoder.config]
tz = "America/Los_Angeles"
```

Example Heka Message

Timestamp 2014-01-10 07:04:56 -0800 PST

Type nginx.error

Hostname trink-x230

Pid 16842

UUID 8e414f01-9d7f-4a48-a5e1-ae92e5954df5

Logger TestWebserverError

Payload using inherited sockets from "6;"

EnvVersion

Severity 5

Fields

name:"tid" value_type:DOUBLE value_double:0
name:"connection" value_type:DOUBLE value_double:8878

2.29.12 Payload Regex Decoder

Plugin Name: **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.

Note: The Go regular expression tester is an invaluable tool for constructing and debugging regular expressions to be used for parsing your input data.

Config:

- **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. See *Heka Message*.
- **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 (see *Field Variables*).

  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 documentation. In addition to the Go time formatting, special *timestamp_layout* values of "Epoch", "EpochMilli", "EpochMicro", and "EpochNano" are supported for Unix style timestamps represented in seconds, milliseconds, microseconds, and nanoseconds since the Epoch, respectively.
- **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*. This setting will have no impact if one of the supported "Epoch*" values is used as the *timestamp_layout* setting.

- **log_errors (bool):** New in version 0.5.

  If set to false, payloads that can not be matched against the regex will not be logged as errors. Defaults to true.

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]
timestamp_layout = "02/Jan/2006:15:04:05 -0700"

# severities in this case would work only if a (?P<Severity>...) matching
# group was present in the regex, and the log file contained this information.
[apache_transform_decoder.severity_map]
DEBUG = 7
INFO = 6
WARNING = 4

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

2.29.13 Payload XML Decoder

Plugin Name: **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 pipeline pack message using XPath syntax using the xmlpath library.

Config:

- **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. See *Heka Message*.
- **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 (see *Field Variables*).

  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 documentation. The default layout is ISO8601 - the same as Javascript. In addition to the Go time formatting, special *timestamp_layout* values of "Epoch", "EpochMilli", "EpochMicro", and "EpochNano" are supported for Unix style timestamps represented in seconds, milliseconds, microseconds, and nanoseconds since the Epoch, respectively.
- **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*. This setting will have no impact if one of the supported "Epoch*" values is used as the *timestamp_layout* setting.

Example:

```
[myxml_decoder]
type = "PayloadXmlDecoder"

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

[myxml_decoder.severity_map]
DEBUG = 7
INFO = 6
WARNING = 4

[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

2.29.14 Protobuf Decoder

Plugin Name: **ProtobufDecoder**

The ProtobufDecoder is used for Heka message objects that have been serialized into protocol buffers format. This is the format that Heka uses to communicate with other Heka instances, so one will always be included in your Heka

configuration under the name "ProtobufDecoder", whether specified or not. The ProtobufDecoder has no configuration options.

The hekad protocol buffers message schema in defined in the *message.proto* file in the *message* package.

Example:

```
[ProtobufDecoder]
```

See also:

Protocol Buffers - Google's data interchange format

2.29.15 Rsyslog Decoder

New in version 0.5.

Plugin Name: **SandboxDecoder**
File Name: **lua_decoders/rsyslog.lua**

Parses the rsyslog output using the string based configuration template.

Config:

- **hostname_keep (boolean, defaults to false)** Always preserve the original 'Hostname' field set by Logstreamer's 'hostname' configuration setting.
- **template (string)** The 'template' configuration string from rsyslog.conf. http://rsyslog-5-8-6-doc.neocities.org/rsyslog_conf_templates.html
- **tz (string, optional, defaults to UTC)** If your rsyslog timestamp field in the template does not carry zone offset information, you may set an offset to be applied to your events here. Typically this would be used with the "Traditional" rsyslog formats.

  Parsing is done by Go, supports values of "UTC", "Local", or a location name corresponding to a file in the IANA Time Zone database, e.g. "America/New_York".

Example Heka Configuration

```
[RsyslogDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/rsyslog.lua"

[RsyslogDecoder.config]
type = "RSYSLOG_TraditionalFileFormat"
template = '%TIMESTAMP% %HOSTNAME% %syslogtag%%msg:::sp-if-no-1st-sp%%msg:::drop-last-lf%\n'
tz = "America/Los_Angeles"
```

Example Heka Message

Timestamp 2014-02-10 12:58:58 -0800 PST

Type RSYSLOG_TraditionalFileFormat

Hostname trink-x230

Pid 0

UUID e0eef205-0b64-41e8-a307-5772b05e16c1

Logger RsyslogInput

Payload "imklog 5.8.6, log source = /proc/kmsg started."

EnvVersion

Severity 7

Fields

name:"programname" value_string:"kernel"

2.29.16 Sandbox Decoder

Plugin Name: **SandboxDecoder**

The SandboxDecoder provides an isolated execution environment for data parsing and complex transformations without the need to recompile Heka. See *Sandbox*. Config:

- *Common Sandbox Parameters*

Example

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

2.29.17 Scribble Decoder

New in version 0.5.

Plugin Name: **ScribbleDecoder**

The ScribbleDecoder is a trivial decoder that makes it possible to set one or more static field values on every decoded message. It is often used in conjunction with another decoder (i.e. in a MultiDecoder w/ cascade_strategy set to "all") to, for example, set the message type of every message to a specific custom value after the messages have been decoded from Protocol Buffers format. Note that this only supports setting the exact same value on every message, if any dynamic computation is required to determine what the value should be, or whether it should be applied to a specific message, a *Sandbox Decoder* using the provided *write_message* API call should be used instead.

Config:

- **message_fields:** Subsection defining message fields to populate. Optional representation metadata can be added at the end of the field name using a pipe delimiter i.e. *host|ipv4 = "192.168.55.55"* will create Fields[Host] containing an IPv4 address. 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 (see *Field Variables*). Does not support Timestamp or Uuid.

Example (in MultiDecoder context)

```
[mytypedecoder]
type = "MultiDecoder"
subs = ["ProtobufDecoder", "mytype"]
cascade_strategy = "all"
log_sub_errors = true

[ProtobufDecoder]

[mytype]
type = "ScribbleDecoder"
```

```
    [mytype.message_fields]
    Type = "MyType"
```

2.29.18 Stats To Fields Decoder

New in version 0.4.

Plugin Name: **StatsToFieldsDecoder**

The StatsToFieldsDecoder will parse time series statistics data in the graphite message format and encode the data into the message fields, in the same format produced by a *Stat Accumulator Input* plugin with the *emit_in_fields* value set to true. This is useful if you have externally generated graphite 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 messages 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.

Example:

```
[StatsToFieldsDecoder]
```

2.30 Filters

2.30.1 Common Filter Parameters

There are some configuration options that are universally available to all Heka filter 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.

New in version 0.7.

- **can_exit (bool, optional)** Whether or not this plugin can exit without causing Heka to shutdown. Defaults to false for non-sandbox filters, and true for sandbox filters.

New in version 0.10.

- **use_buffering (bool, optional)** If true, all messages delivered to this filter will be buffered to disk before delivery, preventing back pressure and allowing retries in cases of message processing failure. Defaults to false, unless otherwise specified by the individual filter's documentation.
- **buffering (QueueBufferConfig, optional)** A sub-section that specifies the settings to be used for the buffering behavior. This will only have any impact if *use_buffering* is set to true. See *Configuring Buffering*.

2.30.2 Circular Buffer Delta Aggregator

New in version 0.5.

Plugin Name: **SandboxFilter**
File Name: **lua_filters/cbufd_aggregator.lua**

Collects the circular buffer delta output from multiple instances of an upstream sandbox filter (the filters should all be the same version at least with respect to their cbuf output). The purpose is to recreate the view at a larger scope in each level of the aggregation i.e., host view -> datacenter view -> service level view.

Config:

- **enable_delta (bool, optional, default false)** Specifies whether or not this aggregator should generate cbuf deltas.
- **anomaly_config(string) - (see *Anomaly Detection Module*)** A list of anomaly detection specifications. If not specified no anomaly detection/alerting will be performed.
- **preservation_version (uint, optional, default 0)** If *preserve_data = true* is set in the SandboxFilter configuration, then this value should be incremented every time the *enable_delta* configuration is changed to prevent the plugin from failing to start during data restoration.

Example Heka Configuration

```
[TelemetryServerMetricsAggregator]
type = "SandboxFilter"
message_matcher = "Logger == 'TelemetryServerMetrics' && Fields[payload_type] == 'cbufd'"
ticker_interval = 60
filename = "lua_filters/cbufd_aggregator.lua"
preserve_data = true

[TelemetryServerMetricsAggregator.config]
enable_delta = false
anomaly_config = 'roc("Request Statistics", 1, 15, 0, 1.5, true, false)'
preservation_version = 0
```

2.30.3 CBuf Delta Aggregator By Hostname

New in version 0.5.

Plugin Name: **SandboxFilter**
File Name: **lua_filters/cbufd_host_aggregatory.lua**

Collects the circular buffer delta output from multiple instances of an upstream sandbox filter (the filters should all be the same version at least with respect to their cbuf output). Each column from the source circular buffer will become its own graph. i.e., 'Error Count' will become a graph with each host being represented in a column.

Config:

- **max_hosts (uint)** Pre-allocates the number of host columns in the graph(s). If the number of active hosts exceed this value, the plugin will terminate.

- **rows (uint)** The number of rows to keep from the original circular buffer. Storing all the data from all the hosts is not practical since you will most likely run into memory and output size restrictions (adjust the view down as necessary).
- **host_expiration (uint, optional, default 120 seconds)** The amount of time a host has to be inactive before it can be replaced by a new host.
- **preservation_version (uint, optional, default 0)** If *preserve_data = true* is set in the SandboxFilter configuration, then this value should be incremented every time the *max_hosts* or *rows* configuration is changed to prevent the plugin from failing to start during data restoration.

Example Heka Configuration

```
[TelemetryServerMetricsHostAggregator]
type = "SandboxFilter"
message_matcher = "Logger == 'TelemetryServerMetrics' && Fields[payload_type] == 'cbufd'"
ticker_interval = 60
filename = "lua_filters/cbufd_host_aggregator.lua"
preserve_data = true

[TelemetryServerMetricsHostAggregator.config]
max_hosts = 5
rows = 60
host_expiration = 120
preservation_version = 0
```

2.30.4 Counter Filter

Plugin Name: **CounterFilter**

Once per ticker interval 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 interval (i.e. it counts the messages the plugin received). Every ten intervals an extra message (also of type *heka.counter-output*) goes out, containing an aggregate count and average per second throughput of messages received.

Config:

- **ticker_interval (int, optional):** Interval between generated counter messages, in seconds. Defaults to 5.

Example:

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

2.30.5 CPU Stats Filter

New in version 0.10.

Plugin Name: **SandboxFilter**
File Name: **lua_filters/procstat.lua**

Calculates deltas in /proc/stat data. Also emits CPU percentage utilization information.

Config:

- **whitelist (string, optional, default "")** Only process fields that fit the pattern, defaults to match all.

- **extras (boolean, optional, default false)** Process extra fields like ctxt, softirq, cpu fields.
- **percent_integer (boolean, optional, default true)** Process percentage as whole number.

Example Heka Configuration

```
[ProcStats]
type = "FilePollingInput"
ticker_interval = 1
file_path = "/proc/stat"
decoder = "ProcStatDecoder"

[ProcStatDecoder]
type = "SandboxDecoder"
filename = "lua_decoders/linux_procstat.lua"

[ProcStatFilter]
type = "SandboxFilter"
filename = "lua_filters/procstat.lua"
preserve_data = true
message_matcher = "Type == 'stats.procstat'"
[ProcStatFilter.config]
    whitelist = "cpu$"
    extras = false
    percent_integer = true
```

Cpu fields: 1 2 3 4 5 6 7 8 9 10 user nice system idle [iowait] [irq] [softirq] [steal] [guest] [guestnice] Note: systems provide user, nice, system, idle. Other fields depend on kernel.

user: Time spent executing user applications (user mode). nice: Time spent executing user applications with low priority (nice). system: Time spent executing system calls (system mode). idle: Idle time. iowait: Time waiting for I/O operations to complete. irq: Time spent servicing interrupts. softirq: Time spent servicing soft-interrupts. steal: ticks spent executing other virtual hosts [virtualization setups] guest: Used in virtualization setups. guestnice: running a niced guest

intr This line shows counts of interrupts serviced since boot time, for each of the possible system interrupts. The first column is the total of all interrupts serviced including unnumbered architecture specific interrupts; each subsequent column is the total for that particular numbered interrupt. Unnumbered interrupts are not shown, only summed into the total.

ctxt 115315 The number of context switches that the system underwent.

btime 769041601 Boot time, in seconds since the Epoch, 1970-01-01 00:00:00 +0000 (UTC).

processes 86031 Number of forks since boot.

procs_running 6 Number of process in runnable state. (Linux 2.5.45 onward.)

procs_blocked 2 Number of process blocked waiting for I/O to complete. (Linux 2.5.45 onward.)

softirq 288977 23 101952 19 13046 19217 7 19125 92077 389 43122 Time spent servicing soft-interrupts.

2.30.6 Disk Stats Filter

New in version 0.7.

Plugin Name: **SandboxFilter**
File Name: **lua_filters/diskstats.lua**

Graphs disk IO stats. It automatically converts the running totals of Writes and Reads into rates of the values. The time based fields are left as running totals of the amount of time doing IO. Expects to receive messages with disk IO data embedded in a particular set of message fields which matches what is generated by *Linux Disk Stats Decoder*: WritesCompleted, ReadsCompleted, SectorsWritten, SectorsRead, WritesMerged, ReadsMerged, TimeWriting, TimeReading, TimeDoingIO, WeightedTimeDoingIO, TickerInterval.

Config:

- **rows (uint, optional, default 1440)** Sets the size of the sliding window i.e., 1440 rows representing 60 seconds per row is a 24 sliding hour window with 1 minute resolution.
- anomaly_config(string) - (see *Anomaly Detection Module*)

Example Heka Configuration

```
[DiskStatsFilter]
type = "SandboxFilter"
filename = "lua_filters/diskstats.lua"
preserve_data = true
message_matcher = "Type == 'stats.diskstats'"
ticker_interval = 10
```

2.30.7 Frequent Items

New in version 0.5.

Plugin Name: **SandboxFilter**
File Name: **lua_filters/frequent_items.lua**

Calculates the most frequent items in a data stream.

Config:

- **message_variable (string)** The message variable name containing the items to be counted.
- **max_items (uint, optional, default 1000)** The maximum size of the sample set (higher will produce a more accurate list).
- **min_output_weight (uint, optional, default 100)** Used to reduce the long tail output by only outputting the higher frequency items.
- **reset_days (uint, optional, default 1)** Resets the list after the specified number of days (on the UTC day boundary). A value of 0 will never reset the list.

Example Heka Configuration

```
[FxaAuthServerFrequentIP]
type = "SandboxFilter"
filename = "lua_filters/frequent_items.lua"
ticker_interval = 60
preserve_data = true
message_matcher = "Logger == 'nginx.access' && Type == 'fxa-auth-server'"

[FxaAuthServerFrequentIP.config]
message_variable = "Fields[remote_addr]"
max_items = 10000
min_output_weight = 100
reset_days = 1
```

2.30.8 Heka Memory Statistics

New in version 0.6.

Plugin Name: **SandboxFilter**
File Name: **lua_filters/heka_memstat.lua**

Graphs the Heka memory statistics using the heka.memstat message generated by pipeline/report.go.

Config:

- **rows (uint, optional, default 1440)** Sets the size of the sliding window i.e., 1440 rows representing 60 seconds per row is a 24 sliding hour window with 1 minute resolution.
- **sec_per_row (uint, optional, default 60)** Sets the size of each bucket (resolution in seconds) in the sliding window.
- **anomaly_config (string, optional)** See *Anomaly Detection Module*.
- **preservation_version (uint, optional, default 0)** If *preserve_data = true* is set in the SandboxFilter configuration, then this value should be incremented every time the *rows* or *sec_per_row* configuration is changed to prevent the plugin from failing to start during data restoration.

Example Heka Configuration

```
[HekaMemstat]
type = "SandboxFilter"
filename = "lua_filters/heka_memstat.lua"
ticker_interval = 60
preserve_data = true
message_matcher = "Type == 'heka.memstat'"
```

2.30.9 Heka Message Schema

New in version 0.5.

Plugin Name: **SandboxFilter**
File Name: **lua_filters/heka_message_schema.lua**

Generates documentation for each unique message in a data stream. The output is a hierarchy of Logger, Type, EnvVersion, and a list of associated message field attributes including their counts (number in the brackets). This plugin is meant for data discovery/exploration and should not be left running on a production system.

Config:

<none>

Example Heka Configuration

```
[SyncMessageSchema]
type = "SandboxFilter"
filename = "lua_filters/heka_message_schema.lua"
ticker_interval = 60
```

```
preserve_data = false
message_matcher = "Logger != 'SyncMessageSchema' && Logger =~ /^Sync/"
```

Example Output

```
Sync-1_5-Webserver [54600]
    slf [54600]
        -no version- [54600]
            upstream_response_time (mismatch)
            http_user_agent (string)
            body_bytes_sent (number)
            remote_addr (string)
            request (string)
            upstream_status (mismatch)
            status (number)
            request_time (number)
            request_length (number)
Sync-1_5-SlowQuery [37]
    mysql.slow-query [37]
        -no version- [37]
            Query_time (number)
            Rows_examined (number)
            Rows_sent (number)
            Lock_time (number)
```

2.30.10 HTTP Status Graph

New in version 0.5.

Plugin Name: **SandboxFilter**
File Name: **lua_filters/http_status.lua**

Graphs HTTP status codes using the numeric Fields[status] variable collected from web server access logs.

Config:

- **sec_per_row (uint, optional, default 60)** Sets the size of each bucket (resolution in seconds) in the sliding window.
- **rows (uint, optional, default 1440)** Sets the size of the sliding window i.e., 1440 rows representing 60 seconds per row is a 24 sliding hour window with 1 minute resolution.
- **anomaly_config (string, optional)** See *Anomaly Detection Module*.
- **alert_throttle (uint, optional, default 3600)** Sets the throttle for the anomaly alert, in seconds.
- **preservation_version (uint, optional, default 0)** If *preserve_data = true* is set in the SandboxFilter configuration, then this value should be incremented every time the *sec_per_row* or *rows* configuration is changed to prevent the plugin from failing to start during data restoration.

Example Heka Configuration

```
[FxaAuthServerHTTPStatus]
type = "SandboxFilter"
filename = "lua_filters/http_status.lua"
ticker_interval = 60
preserve_data = true
message_matcher = "Logger == 'nginx.access' && Type == 'fxa-auth-server'"

[FxaAuthServerHTTPStatus.config]
sec_per_row = 60
rows = 1440
anomaly_config = 'roc("HTTP Status", 2, 15, 0, 1.5, true, false) roc("HTTP Status", 4, 15, 0, 1.5, t
alert_throttle = 300
preservation_version = 0
```

2.30.11 Load Average Filter

New in version 0.7.

Plugin Name: **SandboxFilter**
File Name: **lua_filters/loadavg.lua**

Graphs the load average and process count data. Expects to receive messages containing fields entitled *1MinAvg*, *5MinAvg*, *15MinAvg*, and *NumProcesses*, such as those generated by the *Linux Load Average Decoder*.

Config:

- **sec_per_row (uint, optional, default 60)** Sets the size of each bucket (resolution in seconds) in the sliding window.
- **rows (uint, optional, default 1440)** Sets the size of the sliding window i.e., 1440 rows representing 60 seconds per row is a 24 sliding hour window with 1 minute resolution.
- **anomaly_config (string, optional)** See *Anomaly Detection Module*.
- **preservation_version (uint, optional, default 0)** If *preserve_data = true* is set in the SandboxFilter configuration, then this value should be incremented every time the *sec_per_row* or *rows* configuration is changed to prevent the plugin from failing to start during data restoration.

Example Heka Configuration

```
[LoadAvgFilter]
type = "SandboxFilter"
filename = "lua_filters/loadavg.lua"
ticker_interval = 60
preserve_data = true
message_matcher = "Type == 'stats.loadavg'"
```

2.30.12 Memory Stats Filter

New in version 0.7.

Plugin Name: **SandboxFilter**

File Name: **lua_filters/memstats.lua**

Graphs memory usage statistics. Expects to receive messages with memory usage data embedded in a specific set of message fields, which matches the messages generated by *Linux Memory Stats Decoder*: MemFree, Cached, Active, Inactive, VmallocUsed, Shmem, SwapCached.

Config:

- **sec_per_row (uint, optional, default 60)** Sets the size of each bucket (resolution in seconds) in the sliding window.
- **rows (uint, optional, default 1440)** Sets the size of the sliding window i.e., 1440 rows representing 60 seconds per row is a 24 sliding hour window with 1 minute resolution.
- **anomaly_config (string, optional)** See *Anomaly Detection Module*.
- **preservation_version (uint, optional, default 0)** If *preserve_data = true* is set in the SandboxFilter configuration, then this value should be incremented every time the *sec_per_row* or *rows* configuration is changed to prevent the plugin from failing to start during data restoration.

Example Heka Configuration

```
[MemoryStatsFilter]
type = "SandboxFilter"
filename = "lua_filters/memstats.lua"
ticker_interval = 60
preserve_data = true
message_matcher = "Type == 'stats.memstats'"
```

2.30.13 MySQL Slow Query

New in version 0.6.

Plugin Name: **SandboxFilter**
File Name: **lua_filters/mysql_slow_query.lua**

Graphs MySQL slow query data produced by the *MySQL Slow Query Log Decoder*.

Config:

- **sec_per_row (uint, optional, default 60)** Sets the size of each bucket (resolution in seconds) in the sliding window.
- **rows (uint, optional, default 1440)** Sets the size of the sliding window i.e., 1440 rows representing 60 seconds per row is a 24 sliding hour window with 1 minute resolution.
- **anomaly_config (string, optional)** See *Anomaly Detection Module*.
- **preservation_version (uint, optional, default 0)** If *preserve_data = true* is set in the SandboxFilter configuration, then this value should be incremented every time the *sec_per_row* or *rows* configuration is changed to prevent the plugin from failing to start during data restoration.

Example Heka Configuration

```
[Sync-1_5-SlowQueries]
type = "SandboxFilter"
message_matcher = "Logger == 'Sync-1_5-SlowQuery'"
ticker_interval = 60
filename = "lua_filters/mysql_slow_query.lua"

    [Sync-1_5-SlowQueries.config]
    anomaly_config = 'mww_nonparametric("Statistics", 5, 15, 10, 0.8)'
    preservation_version = 0
```

2.30.14 Sandbox Filter

Plugin Name: **SandboxFilter**

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

Config:

- *Common Filter Parameters*
- *Common Sandbox Parameters*
- **timer_event_on_shutdown (bool):** True if the sandbox should have its timer_event function called on shutdown.

Example:

```
[hekabench_counter]
type = "SandboxFilter"
message_matcher = "Type == 'hekabench'"
ticker_interval = 1
filename = "counter.lua"
preserve_data = true
profile = false

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

2.30.15 Sandbox Manager Filter

Plugin Name: **SandboxManagerFilter**

The SandboxManagerFilter provides dynamic control (start/stop) of sandbox filters in a secure manner without stopping the Heka daemon. Commands are sent to a SandboxManagerFilter 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.

SandboxManagerFilter Settings

- *Common Filter 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.

- **module_directory (string):** The directory where 'require' will attempt to load the external Lua modules from. Defaults to ${SHARE_DIR}/lua_modules.
- **max_filters (uint):** The maximum number of filters this manager can run.

New in version 0.5.

- **memory_limit (uint):** The number of bytes managed sandboxes are allowed to consume before being terminated (default 8MiB).
- **instruction_limit (uint):** The number of instructions managed sandboxes are allowed to execute during the process_message/timer_event functions before being terminated (default 1M).
- **output_limit (uint):** The number of bytes managed sandbox output buffers can hold before being terminated (default 63KiB). Warning: messages exceeding 64KiB will generate an error and be discarded by the standard output plugins (File, TCP, UDP) since they exceed the maximum message size.

Example

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

2.30.16 Stat Filter

Plugin Name: **StatFilter**

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

Config:

- Metric:

  Subsection defining a single metric to be generated. Both the *name* and *value* fields for each metric support interpolation of message field values (from 'Type', 'Hostname', 'Logger', 'Payload', or any dynamic field name) with the use of %% delimiters, so *%Hostname%* would be replaced by the message's Hostname field, and %Foo% would be replaced by the first value of a dynamic field called "Foo":

  - **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.
  - **replace_dot (boolean):** Replace all dots . per an underscore _ during the string interpolation. It's useful if you output this result in a graphite instance.
- **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:

```
[StatAccumInput]
ticker_interval = 5

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

```
[Hits]
type = "StatFilter"
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 StatAccumInput to be running.

2.30.17 Stats Graph

New in version 0.7.

Plugin Name: **SandboxFilter**
File Name: **lua_filters/stat_graph.lua**

Converts stat values extracted from statmetric messages (see *Stat Accumulator Input*) to circular buffer data and periodically emits messages containing this data to be graphed by a DashboardOutput. Note that this filter expects the stats data to be available in the message fields, so the StatAccumInput *must* be configured with *emit_in_fields* set to true for this filter to work correctly.

Config:

- **title (string, optional, default "Stats"):** Title for the graph output generated by this filter.
- **rows (uint, optional, default 300):** The number of rows to store in our circular buffer. Each row represents one time interval.
- **sec_per_row (uint, optional, default 1):** The number of seconds in each circular buffer time interval.
- **stats (string):** Space separated list of stat names. Each specified stat will be expected to be found in the fields of the received statmetric messages, and will be extracted and inserted into its own column in the accumulated circular buffer.
- **stat_labels (string):** Space separated list of header label names to use for the extracted stats. Must be in the same order as the specified stats. Any label longer than 15 characters will be truncated.
- **anomaly_config (string, optional):** Anomaly detection configuration, see *Anomaly Detection Module*.
- **preservation_version (uint, optional, default 0):** If *preserve_data = true* is set in the SandboxFilter configuration, then this value should be incremented every time any edits are made to your *rows*, *sec_per_row*, *stats*, or *stat_labels* values, or else Heka will fail to start because the preserved data will no longer match the filter's data structure.
- **stat_aggregation (string, optional, default "sum"):**

  Controls how the column data is aggregated when combining multiple circular buffers. "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. “none” - No aggregation will be performed the column.

- **stat_unit (string, optional, default “count”):** 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).

Example Heka Configuration

```
[stat-graph]
type = "SandboxFilter"
filename = "lua_filters/stat_graph.lua"
ticker_interval = 10
preserve_data = true
message_matcher = "Type == 'heka.statmetric'"

  [stat-graph.config]
  title = "Hits and Misses"
  rows = 1440
  stat_aggregation = "none"
  stat_unit = "count"
  sec_per_row = 10
  stats = "stats.counters.hits.count stats.counters.misses.count"
  stat_labels = "hits misses"
  anomaly_config = 'roc("Hits and Misses", 1, 15, 0, 1.5, true, false) roc("Hits and Misses", 2, 15,
  preservation_version = 0
```

2.30.18 Unique Items

New in version 0.6.

Plugin Name: **SandboxFilter**
File Name: **lua_filters/unique_items.lua**

Counts the number of unique items per day e.g. active daily users by uid.

Config:

- **message_variable (string, required)** The Heka message variable containing the item to be counted.
- **title (string, optional, default “Estimated Unique Daily *message_variable*”)** The graph title for the cbuf output.
- **enable_delta (bool, optional, default false)** Specifies whether or not this plugin should generate cbuf deltas. Deltas should be enabled when sharding is used; see: *Circular Buffer Delta Aggregator*.
- **preservation_version (uint, optional, default 0)** If *preserve_data = true* is set in the SandboxFilter configuration, then this value should be incremented every time the *enable_delta* configuration is changed to prevent the plugin from failing to start during data restoration.

Example Heka Configuration

```
[FxaActiveDailyUsers]
type = "SandboxFilter"
filename = "lua_filters/unique_items.lua"
ticker_interval = 60
preserve_data = true
```

```
message_matcher = "Logger == 'FxaAuth' && Type == 'request.summary' && Fields[path] == '/v1/certifica

    [FxaActiveDailyUsers.config]
    message_variable = "Fields[uid]"
    title = "Estimated Active Daily Users"
    preservation_version = 0
```

2.31 Encoders

2.31.1 Alert Encoder

Plugin Name: **SandboxEncoder**
File Name: **lua_encoders/alert.lua**

Produces more human readable alert messages.

Config:

<none>

Example Heka Configuration

```
[FxaAlert]
type = "SmtpOutput"
message_matcher = "Type == 'heka.sandbox-output' && Fields[payload_type] == 'alert' && Logger =~ /^Fx
send_from = "heka@example.com"
send_to = ["alert@example.com"]
auth = "Plain"
user = "test"
password = "testpw"
host = "localhost:25"
encoder = "AlertEncoder"

[AlertEncoder]
type = "SandboxEncoder"
filename = "lua_encoders/alert.lua"
```

Example Output

Timestamp 2014-05-14T14:20:18Z

Hostname ip-10-226-204-51

Plugin FxaBrowserIdHTTPStatus

Alert HTTP Status - algorithm: roc col: 1 msg: detected anomaly, standard deviation exceeds 1.5

2.31.2 CBUF Librato Encoder

New in version 0.8.

Plugin Name: **SandboxEncoder**
File Name: **lua_encoders/cbuf_librato.lua**

Extracts data from SandboxFilter circular buffer output messages and uses it to generate time series JSON structures that will be accepted by Librato's POST API. It will keep track of the last time it's seen a particular message, keyed by filter name and output name. The first time it sees a new message, it will send data from all of the rows except the last one, which is possibly incomplete. For subsequent messages, the encoder will automatically extract data from all of the rows that have elapsed since the last message was received.

The SandboxEncoder *preserve_data* setting should be set to true when using this encoder, or else the list of received messages will be lost whenever Heka is restarted, possibly causing the same data rows to be sent to Librato multiple times.

Config:

- **message_key (string, optional, default "%{Logger}:%{payload_name}")** String to use as the key to differentiate separate cbuf messages from each other. Supports *message field interpolation*.

Example Heka Configuration

```
[cbuf_librato_encoder]
type = "SandboxEncoder"
filename = "lua_encoders/cbuf_librato.lua"
preserve_data = true
  [cbuf_librato_encoder.config]
  message_key = "%{Logger}:%{Hostname}:%{payload_name}"

[librato]
type = "HttpOutput"
message_matcher = "Type == 'heka.sandbox-output && Fields[payload_type] == 'cbuf'"
encoder = "cbuf_librato_encoder"
address = "https://metrics-api.librato.com/v1/metrics"
username = "username@example.com"
password = "SECRET"
    [librato.headers]
    Content-Type = ["application/json"]
```

Example Output

```
{"gauges":[{"value":12,"measure_time":1410824950,"name":"HTTP_200","source":"thor"},{"value":1,"measu
```

2.31.3 ElasticSearch JSON Encoder

Plugin Name: **ESJsonEncoder**

This encoder serializes a Heka message into a clean JSON format, preceded by a separate JSON structure containing information required for ElasticSearch BulkAPI indexing. The JSON serialization is done by hand, without the use of Go's stdlib JSON marshalling. This is so serialization can succeed even if the message contains invalid UTF-8 characters, which will be encoded as U+FFFD. Config:

- **index (string):** Name of the ES index into which the messages will be inserted. Supports interpolation of message field values (from 'Type', 'Hostname', 'Pid', 'UUID', 'Logger', 'EnvVersion', 'Severity', a field name, or a timestamp format) with the use of '%{}' chars, so '%{Hostname}-%{Logger}-data' would add the records to an ES index called 'some.example.com-processname-data'. Allows to use strftime format codes. Defaults to 'heka-%{%Y.%m.%d}'.
- **type_name (string):** Name of ES record type to create. Supports interpolation of message field values (from 'Type', 'Hostname', 'Pid', 'UUID', 'Logger', 'EnvVersion', 'Severity', field name, or a timestamp format) with the use of '%{}' chars, so '%{Hostname}-stat' would create an ES record with a type of 'some.example.com-stat'. Defaults to 'message'.

- **fields ([]string):** The 'fields' parameter specifies 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 "DynamicFields" (where "DynamicFields" causes the inclusion of dynamically specified message fields, see `dynamic_fields`). Defaults to including all of the supported message fields.
- **timestamp (string):** Format to use for timestamps in generated ES documents. Allows to use strftime format codes. Defaults to "%Y-%m-%dT%H:%M:%S".
- **es_index_from_timestamp (bool):** When generating the index name use the timestamp from the message instead of the current time. Defaults to false.
- **id (string):** Allows you to optionally specify the document id for ES to use. Useful for overwriting existing ES documents. If the value specified is placed within %{}, it will be interpolated to its Field value. Default is allow ES to auto-generate the id.
- **raw_bytes_fields ([]string):** This specifies a set of fields which will be passed through to the encoded JSON output without any processing or escaping. This is useful for fields which contain embedded JSON objects to prevent the embedded JSON from being escaped as normal strings. Only supports dynamically specified message fields.
- **field_mappings (map[string]string):** Maps Heka message fields to custom ES keys. Can be used to implement a custom format in ES or implement Logstash V1. The available fields are "Timestamp", "Uuid", "Type", "Logger", "Severity", "Payload", "EnvVersion", "Pid" and "Hostname".
- **dynamic_fields ([]string):** This specifies which of the message's dynamic fields should be included in the JSON output. Defaults to including all of the messages dynamic fields. If `dynamic_fields` is non-empty, then the `fields` list *must* contain "DynamicFields" or an error will be raised.

Example

```
[ESJsonEncoder]
index = "%{Type}-%{2006.01.02}"
es_index_from_timestamp = true
type_name = "%{Type}"
    [ESJsonEncoder.field_mappings]
    Timestamp = "@timestamp"
    Severity = "level"

[ElasticSearchOutput]
message_matcher = "Type == 'nginx.access'"
encoder = "ESJsonEncoder"
flush_interval = 50
```

2.31.4 ElasticSearch Logstash V0 Encoder

Plugin Name: **ESLogstashV0Encoder**

This encoder serializes a Heka message into a JSON format, preceded by a separate JSON structure containing information required for ElasticSearch BulkAPI indexing. The message JSON structure uses the original (i.e. "v0") schema popularized by Logstash. Using this schema can aid integration with existing Logstash deployments. This schema also plays nicely with the default Logstash dashboard provided by Kibana.

The JSON serialization is done by hand, without using Go's stdlib JSON marshalling. This is so serialization can succeed even if the message contains invalid UTF-8 characters, which will be encoded as U+FFFD. Config:

- **index (string):** Name of the ES index into which the messages will be inserted. Supports interpolation of message field values (from 'Type', 'Hostname', 'Pid', 'UUID', 'Logger', 'EnvVersion', 'Severity', a field name, or a timestamp format) with the use of '%{}' chars, so '%{Hostname}-%{Logger}-data' would

add the records to an ES index called 'some.example.com-processname-data'. Defaults to 'logstash-%{2006.01.02}'.

- **type_name (string):** Name of ES record type to create. Supports interpolation of message field values (from 'Type', 'Hostname', 'Pid', 'UUID', 'Logger', 'EnvVersion', 'Severity', field name, or a timestamp format) with the use of '%{}' chars, so '%{Hostname}-stat' would create an ES record with a type of 'some.example.com-stat'. Defaults to 'message'.
- **use_message_type (bool):** If false, the generated JSON's @type value will match the ES record type specified in the type_name setting. If true, the message's Type value will be used as the @type value instead. Defaults to false.
- **fields ([]string):** The 'fields' parameter specifies 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 "DynamicFields" (where "DynamicFields" causes the inclusion of dynamically specified message fields, see `dynamic_fields`). Defaults to including all of the supported message fields. The "Payload" field is sent to ElasticSearch as "@message".
- **timestamp (string):** Format to use for timestamps in generated ES documents. Allows to use strftime format codes. Defaults to "%Y-%m-%dT%H:%M:%S".
- **es_index_from_timestamp (bool):** When generating the index name use the timestamp from the message instead of the current time. Defaults to false.
- **id (string):** Allows you to optionally specify the document id for ES to use. Useful for overwriting existing ES documents. If the value specified is placed within %{}, it will be interpolated to its Field value. Default is allow ES to auto-generate the id.
- **raw_bytes_fields ([]string):** This specifies a set of fields which will be passed through to the encoded JSON output without any processing or escaping. This is useful for fields which contain embedded JSON objects to prevent the embedded JSON from being escaped as normal strings. Only supports dynamically specified message fields.
- **dynamic_fields ([]string):** This specifies which of the message's dynamic fields should be included in the JSON output. Defaults to including all of the messages dynamic fields. If `dynamic_fields` is non-empty, then the `fields` list *must* contain "DynamicFields" or an error will be raised.

Example

```
[ESLogstashV0Encoder]
es_index_from_timestamp = true
type_name = "%{Type}"

[ElasticSearchOutput]
message_matcher = "Type == 'nginx.access'"
encoder = "ESLogstashV0Encoder"
flush_interval = 50
```

2.31.5 ElasticSearch Payload Encoder

Plugin Name: **SandboxEncoder**
File Name: **lua_encoders/es_payload.lua**

Prepends ElasticSearch BulkAPI index JSON to a message payload.

Config:

- **index (string, optional, default "heka-%{%Y.%m.%d}")** String to use as the *_index* key's value in the generated JSON. Supports field interpolation as described below.
- **type_name (string, optional, default "message")** String to use as the *_type* key's value in the generated JSON. Supports field interpolation as described below.
- **id (string, optional)** String to use as the *_id* key's value in the generated JSON. Supports field interpolation as described below.
- **es_index_from_timestamp (boolean, optional)** If true, then any time interpolation (often used to generate the ElasticSeach index) will use the timestamp from the processed message rather than the system time.

Field interpolation:

All of the string config settings listed above support *message field interpolation*.

Example Heka Configuration

```
[es_payload]
type = "SandboxEncoder"
filename = "lua_encoders/es_payload.lua"
    [es_payload.config]
    es_index_from_timestamp = true
    index = "%{Logger}-%{%Y.%m.%d}"
    type_name = "%{Type}-%{Hostname}"

[ElasticSearchOutput]
message_matcher = "Type == 'mytype'"
encoder = "es_payload"
```

Example Output

```
{"index":{"_index":"mylogger-2014.06.05","_type":"mytype-host.domain.com"}}
{"json":"data","extracted":"from","message":"payload"}
```

2.31.6 Payload Encoder

Plugin Name: **PayloadEncoder**

The PayloadEncoder simply extracts the payload from the provided Heka message and converts it into a byte stream for delivery to an external resource. Config:

- **append_newlines (bool, optional):** Specifies whether or not a newline character (i.e. *n*) will be appended to the captured message payload before serialization. Defaults to true.
- **prefix_ts (bool, optional):** Specifies whether a timestamp will be prepended to the captured message payload before serialization. Defaults to false.
- **ts_from_message (bool, optional):** If true, the prepended timestamp will be extracted from the message that is being processed. If false, the prepended timestamp will be generated by the system clock at the time of message processing. Defaults to true. This setting has no impact if *prefix_ts* is set to false.
- **ts_format (string, optional):** Specifies the format that should be used for prepended timestamps, using Go's standard time format specification strings. Defaults to *[2006/Jan/02:15:04:05 -0700]*. If the specified format string does not end with a space character, then a space will be inserted between the formatted timestamp and the payload.

Example

```
[PayloadEncoder]
append_newlines = false
prefix_ts = true
ts_format = "2006/01/02 3:04:05PM MST"
```

2.31.7 Protobuf Encoder

Plugin Name: **ProtobufEncoder**

The ProtobufEncoder is used to serialize Heka message objects back into Heka's standard protocol buffers format. This is the format that Heka uses to communicate with other Heka instances, so one will always be included in your Heka configuration using the default "ProtobufEncoder" name whether specified or not.

The hekad protocol buffers message schema is defined in the *message.proto* file in the *message* package.

Config:

<none>

Example:

```
[ProtobufEncoder]
```

See also:

Protocol Buffers - Google's data interchange format

2.31.8 Restructured Text Encoder

Plugin Name: **RstEncoder**

The RstEncoder generates a reStructuredText rendering of a Heka message, including all fields and attributes. It is useful for debugging, especially when coupled with a *Log Output*.

Config:

<none>

Example:

```
[RstEncoder]

[LogOutput]
message_matcher = "TRUE"
encoder = "RstEncoder"
```

2.31.9 Sandbox Encoder

Plugin Name: **SandboxEncoder**

The SandboxEncoder provides an isolated execution environment for converting messages into binary data without the need to recompile Heka. See *Sandbox*. Config:

- *Common Sandbox Parameters*

Example

```
[custom_json_encoder]
type = "SandboxEncoder"
filename = "path/to/custom_json_encoder.lua"

    [custom_json_encoder.config]
    msg_fields = ["field1", "field2"]
```

2.31.10 Schema InfluxDB Encoder

New in version 0.8.

Plugin Name: **SandboxEncoder**
File Name: **lua_encoders/schema_influx.lua**

Converts full Heka message contents to JSON for InfluxDB HTTP API. Includes all standard message fields and iterates through all of the dynamically specified fields, skipping any bytes fields or any fields explicitly omitted using the *skip_fields* config option.

Note: This encoder is intended for use with InfluxDB versions *prior* to 0.9. If you're working with InfluxDB v0.9 or greater, you'll want to use the *Schema InfluxDB Write Encoder* instead.

Config:

- **series (string, optional, default "series")** String to use as the *series* key's value in the generated JSON. Supports interpolation of field values from the processed message, using *%{fieldname}*. Any *fieldname* values of "Type", "Payload", "Hostname", "Pid", "Logger", "Severity", or "EnvVersion" will be extracted from the the base message schema, any other values will be assumed to refer to a dynamic message field. Only the first value of the first instance of a dynamic message field can be used for series name interpolation. If the dynamic field doesn't exist, the uninterpolated value will be left in the series name. Note that it is not possible to interpolate either the "Timestamp" or the "Uuid" message fields into the series name, those values will be interpreted as referring to dynamic message fields.
- **skip_fields (string, optional, default "")** Space delimited set of fields that should *not* be included in the InfluxDB records being generated. Any fieldname values of "Type", "Payload", "Hostname", "Pid", "Logger", "Severity", or "EnvVersion" will be assumed to refer to the corresponding field from the base message schema. Any other values will be assumed to refer to a dynamic message field.
- **multi_series (boolean, optional, default false)** Instead of submitting all fields to InfluxDB as attributes of a single series, submit a series for each field that sets a "value" attribute to the value of the field. This also sets the name attribute to the series value with the field name appended to it by a ".". This is the recommended by InfluxDB for v0.9 onwards as it is found to provide better performance when querying and aggregating across multiple series.
- **exclude_base_fields (boolean, optional, default false)** Don't send the base fields to InfluxDB. This saves storage space by not including base fields that are mostly redundant and unused data. If skip_fields includes base fields, this overrides it and will only be relevant for skipping dynamic fields.

Example Heka Configuration

```
[influxdb]
type = "SandboxEncoder"
filename = "lua_encoders/schema_influx.lua"
    [influxdb.config]
    series = "heka.%{Logger}"
```

```
    skip_fields = "Pid EnvVersion"

[InfluxOutput]
message_matcher = "Type == 'influxdb'"
encoder = "influxdb"
type = "HttpOutput"
address = "http://influxdbserver.example.com:8086/db/databasename/series"
username = "influx_username"
password = "influx_password"
```

Example Output

```
[{"points":[[1.409378221e+21,"log","test","systemName","TcpInput",5,1,"test"]],"name":"heka.MyLogger"
```

2.31.11 Schema InfluxDB Write Encoder

New in version 0.10.

Plugin Name: **SandboxEncoder**
File Name: **lua_encoders/schema_influx_write.lua**

2.31.12 StatMetric InfluxDB Encoder

New in version 0.7.

Plugin Name: **SandboxEncoder**
File Name: **lua_encoders/statmetric_influx.lua**

Extracts data from message fields in *heka.statmetric* messages generated by a *Stat Accumulator Input* and generates JSON suitable for use with InfluxDB's HTTP API. StatAccumInput must be configured with *emit_in_fields = true* for this encoder to work correctly.

Config:

<none>

Example Heka Configuration

```
[statmetric-influx-encoder]
type = "SandboxEncoder"
filename = "lua_encoders/statmetric_influx.lua"

[influx]
type = "HttpOutput"
message_matcher = "Type == 'heka.statmetric'"
address = "http://myinfluxserver.example.com:8086/db/stats/series"
encoder = "statmetric-influx-encoder"
username = "influx_username"
password = "influx_password"
```

Example Output

[{"points":[[1408404848,78271]],"name":"stats.counters.000000.rate","columns":["time","value"]},{"poi

2.32 Outputs

:

2.32.1 Common Output Parameters

There are some configuration options that are universally available to all Heka 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.
- **encoder (string, optional):** New in version 0.6.

  Encoder to be used by the output. This should refer to the name of an encoder plugin section that is specified elsewhere in the TOML configuration. Messages can be encoded using the specified encoder by calling the OutputRunner's *Encode()* method.
- **use_framing (bool, optional):** New in version 0.6.

  Specifies whether or not Heka's *Stream Framing* should be applied to the binary data returned from the OutputRunner's *Encode()* method.
- **can_exit (bool, optional)** New in version 0.7.

  Whether or not this plugin can exit without causing Heka to shutdown. Defaults to false.

2.32.2 AMQP Output

Plugin Name: **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.

Config:

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

- **routing_key (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.
- **retries (RetryOptions, optional):** A sub-section that specifies the settings to be used for restart behavior. See *Configuring Restarting Behavior*

New in version 0.6.

- **content_type (string):** MIME content type of the payload used in the AMQP header. Defaults to "application/hekad".
- **encoder (string, optional)** Specifies which of the registered encoders should be used for converting Heka messages to binary data that is sent out over the AMQP connection. Defaults to the always available "ProtobufEncoder".
- **use_framing (bool, optional):** Specifies whether or not the encoded data sent out over the TCP connection should be delimited by Heka's *Stream Framing*. Defaults to true.

New in version 0.6.

- **tls (TlsConfig):** An optional sub-section that specifies the settings to be used for any SSL/TLS encryption. This will only have any impact if *URL* uses the *AMQPS* URI scheme. See *Configuring TLS*.

Example (that sends log lines from the logger):

```
[AMQPOutput]
url = "amqp://guest:guest@rabbitmq/"
exchange = "testout"
exchange_type = "fanout"
message_matcher = 'Logger == "TestWebserver"'
```

2.32.3 Carbon Output

Plugin Name: **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 or UDP socket using the plaintext protocol.

Config:

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

New in version 0.5.

- **protocol (string):** "tcp" or "udp" (default: "tcp")
- **tcp_keep_alive (bool)** if set, keep the TCP connection open and reuse it until a failure; then retry (default: false)

Example:

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

2.32.4 Dashboard Output

Plugin Name: **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.

Config:

- **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 *$(BASE_DIR)/dashboard*.
- **static_directory (string):** File system directory where the Heka dashboard source code can be found. The Heka process must have read access to this directory. Relative paths will be evaluated relative to the Heka base directory. Defaults to *${SHARE_DIR}/dasher*.

New in version 0.7.

- **headers (subsection, optional):** It is possible to inject arbitrary HTTP headers into each outgoing response by adding a TOML subsection entitled "headers" to you HttpOutput config section. All entries in the subsection must be a list of string values.

Example:

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

2.32.5 ElasticSearch Output

Plugin Name: **ElasticSearchOutput**

Output plugin that uses HTTP or UDP to insert records into an ElasticSearch database. Note that it is up to the specified encoder to both serialize the message into a JSON structure *and* to prepend that with the appropriate ElasticSearch BulkAPI indexing JSON. Usually this output is used in conjunction with an ElasticSearch-specific encoder plugin, such as *ElasticSearch JSON Encoder*, *ElasticSearch Logstash V0 Encoder*, or *ElasticSearch Payload Encoder*.

Config:

- **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.
- **server (string):** ElasticSearch server URL. Supports http://, https:// and udp:// urls. Defaults to "http://localhost:9200".
- **connect_timeout (int):** Time in milliseconds to wait for a server name resolving and connection to ES. It's included in an overall time (see 'http_timeout' option), if they both are set. Default is 0 (no timeout).
- **http_timeout (int):** Time in milliseconds to wait for a response for each http post to ES. This may drop data as there is currently no retry. Default is 0 (no timeout).

- **http_disable_keepalives (bool):** Specifies whether or not re-using of established TCP connections to ElasticSearch should be disabled. Defaults to false, that means using both HTTP keep-alive mode and TCP keep-alives. Set it to true to close each TCP connection after 'flushing' messages to ElasticSearch.
- **username (string):** The username to use for HTTP authentication against the ElasticSearch host. Defaults to "" (i. e. no authentication).
- **password (string):** The password to use for HTTP authentication against the ElasticSearch host. Defaults to "" (i. e. no authentication).

New in version 0.9.

- **tls (TlsConfig):** An optional sub-section that specifies the settings to be used for any SSL/TLS encryption. This will only have any impact if *URL* uses the *HTTPS* URI scheme. See *Configuring TLS*.
- **use_buffering (bool, optional):** Buffer records to a disk-backed buffer on the Heka server before writing them to ElasticSearch. Defaults to true.
- **buffering (QueueBufferConfig, optional):** All of the *buffering* config options are set to the standard default options.

Example:

```
[ElasticSearchOutput]
message_matcher = "Type == 'sync.log'"
server = "http://es-server:9200"
flush_interval = 5000
flush_count = 10
encoder = "ESJsonEncoder"
```

2.32.6 File Output

Plugin Name: **FileOutput**

Writes message data out to a file system.

Config:

- **path (string):** Full path to the output file. If date rotation is in use, then the output file path can support strftime syntax to embed timestamps in the file path: http://strftime.org
- **perm (string, optional):** File permission for writing. A string of the octal digit representation. Defaults to "644".
- **folder_perm (string, optional):** Permissions to apply to directories created for FileOutput's parent directory if it doesn't exist. Must be a string representation of an octal integer. Defaults to "700".
- **flush_interval (uint32, optional):** Interval at which accumulated file data should be written to disk, in milliseconds (default 1000, i.e. 1 second). Set to 0 to disable.
- **flush_count (uint32, optional):** Number of messages to accumulate until file data should be written to disk (default 1, minimum 1).
- **flush_operator (string, optional):** Operator describing how the two parameters "flush_interval" and "flush_count" are combined. Allowed values are "AND" or "OR" (default is "AND").

New in version 0.6.

- **use_framing (bool, optional):** Specifies whether or not the encoded data sent out over the TCP connection should be delimited by Heka's *Stream Framing*. Defaults to true if a ProtobufEncoder is used, false otherwise.

New in version 0.9.

- **rotation_interval (uint32, optional):** Interval at which the output file should be rotated, in hours. Only the following values are allowed: 0, 1, 4, 12, 24 (set to 0 to disable). The files will be named relative to midnight of the day. Defaults to 0, i.e. disabled.

Example:

```
[counter_file]
type = "FileOutput"
message_matcher = "Type == 'heka.counter-output'"
path = "/var/log/heka/counter-output.log"
perm = "666"
flush_count = 100
flush_operator = "OR"
encoder = "PayloadEncoder"
```

New in version 0.6.

2.32.7 HTTP Output

Plugin Name: **HttpOutput**

A very simple output plugin that uses HTTP GET, POST, or PUT requests to deliver data to an HTTP endpoint. When using POST or PUT request methods the encoded output will be uploaded as the request body. When using GET the encoded output will be ignored.

This output doesn't support any request batching; each received message will generate an HTTP request. Batching can be achieved by use of a filter plugin that accumulates message data, periodically emitting a single message containing the batched, encoded HTTP request data in the payload. An HttpOutput can then be configured to capture these batch messages, using a *Payload Encoder* to extract the message payload.

For now the HttpOutput only supports statically defined request parameters (URL, headers, auth, etc.). Future iterations will provide a mechanism for dynamically specifying these values on a per-message basis.

Config:

- **address (string):** URL address of HTTP server to which requests should be sent. Must begin with "http://" or "https://".
- **method (string, optional):** HTTP request method to use, must be one of GET, POST, or PUT. Defaults to POST.
- **username (string, optional):** If specified, HTTP Basic Auth will be used with the provided user name.
- **password (string, optional):** If specified, HTTP Basic Auth will be used with the provided password.
- **headers (subsection, optional):** It is possible to inject arbitrary HTTP headers into each outgoing request by adding a TOML subsection entitled "headers" to you HttpOutput config section. All entries in the subsection must be a list of string values.
- **http_timeout(uint, optional):** Time in milliseconds to wait for a response for each http request. This may drop data as there is currently no retry. Default is 0 (no timeout)
- **tls (subsection, optional):** A sub-section that specifies the settings to be used for any SSL/TLS encryption. This will only have any impact if an "https://" address is used. See *Configuring TLS*.

Example:

```
[PayloadEncoder]

[influxdb]
type = "HttpOutput"
```

```
message_matcher = "Type == 'influx.formatted'"
address = "http://influxdb.example.com:8086/db/stats/series"
encoder = "PayloadEncoder"
username = "MyUserName"
password = "MyPassword"
```

2.32.8 IRC Output

Plugin Name: **IrcOutput**

Connects to an Irc Server and sends messages to the specified Irc channels. Output is encoded using the specified encoder, and expects output to be properly truncated to fit within the bounds of an Irc message before being receiving the output.

Config:

- **server (string):** A host:port of the irc server that Heka will connect to for sending output.
- **nick (string):** Irc nick used by Heka.
- **ident (string):** The Irc identity used to login with by Heka.
- **password (string, optional):** The password used to connect to the Irc server.
- **channels (list of strings):** A list of Irc channels which every matching Heka message is sent to. If there is a space in the channel string, then the part after the space is expected to be a password for a protected irc channel.
- **timeout (uint, optional):** The maximum amount of time (in seconds) to wait before timing out when connect, reading, or writing to the Irc server. Defaults to 10.
- **tls (TlsConfig, optional):** A sub-section that specifies the settings to be used for any SSL/TLS encryption. This will only have any impact if *use_tls* is set to true. See *Configuring TLS*.
- **queue_size (uint, optional):** This is the maximum amount of messages Heka will queue per Irc channel before discarding messages. There is also a queue of the same size used if all per-irc channel queues are full. This is used when Heka is unable to send a message to an Irc channel, such as when it hasn't joined or has been disconnected. Defaults to 100.
- **rejoin_on_kick (bool, optional):** Set this if you want Heka to automatically re-join an Irc channel after being kicked. If not set, and Heka is kicked, it will not attempt to rejoin ever. Defaults to false.
- **ticker_interval (uint, optional):** How often (in seconds) heka should send a message to the server. This is on a per message basis, not per channel. Defaults to 2.
- **time_before_reconnect (uint, optional):** How long to wait (in seconds) before reconnecting to the Irc server after being disconnected. Defaults to 3.
- **time_before_rejoin (uint, optional):** How long to wait (in seconds) before attempting to rejoin an Irc channel which is full. Defaults to 3.
- **max_join_retries (uint, optional):** The maximum amount of attempts Heka will attempt to join an Irc channel before giving up. After attempts are exhausted, Heka will no longer attempt to join the channel. Defaults to 3.
- **verbose_irc_logging (bool, optional):** Enable to see raw internal message events Heka is receiving from the server. Defaults to false.
- **encoder (string):** Specifies which of the registered encoders should be used for converting Heka messages into what is sent to the irc channels.

- **retries (RetryOptions, optional):** A sub-section that specifies the settings to be used for restart behavior. See *Configuring Restarting Behavior*

Example:

```
[IrcOutput]
message_matcher = 'Type == "alert"'
encoder = "PayloadEncoder"
server = "irc.mozilla.org:6667"
nick = "heka_bot"
ident = "heka_ident"
channels = [ "#heka_bot_irc testkeypassword" ]
rejoin_on_kick = true
queue_size = 200
ticker_interval = 1
```

2.32.9 Kafka Output

Plugin Name: **KafkaOutput**

Connects to a Kafka broker and sends messages to the specified topic.

Config:

- **id (string)** Client ID string. Default is the hostname.
- **addrs ([]string)** List of brokers addresses.
- **metadata_retries (int)** How many times to retry a metadata request when a partition is in the middle of leader election. Default is 3.
- **wait_for_election (uint32)** How long to wait for leader election to finish between retries (in milliseconds). Default is 250.
- **background_refresh_frequency (uint32)** How frequently the client will refresh the cluster metadata in the background (in milliseconds). Default is 600000 (10 minutes). Set to 0 to disable.
- **max_open_reqests (int)** How many outstanding requests the broker is allowed to have before blocking attempts to send. Default is 4.
- **dial_timeout (uint32)** How long to wait for the initial connection to succeed before timing out and returning an error (in milliseconds). Default is 60000 (1 minute).
- **read_timeout (uint32)** How long to wait for a response before timing out and returning an error (in milliseconds). Default is 60000 (1 minute).
- **write_timeout (uint32)** How long to wait for a transmit to succeed before timing out and returning an error (in milliseconds). Default is 60000 (1 minute).
- **partitioner (string)** Chooses the partition to send messages to. The valid values are *Random*, *RoundRobin*, *Hash*. Default is Random.
- **hash_variable (string)** The message variable used for the Hash partitioner only. The variables are restricted to *Type*, *Logger*, *Hostname*, *Payload* or any of the message's dynamic field values. All dynamic field values will be converted to a string representation. Field specifications are the same as with the *Message Matcher Syntax* e.g. Fields[foo][0][0].
- **topic_variable (string)** The message variable used as the Kafka topic (cannot be used in conjunction with the 'topic' configuration). The variable restrictions are the same as the hash_variable.
- **topic (string)** A static Kafka topic (cannot be used in conjunction with the 'topic_variable' configuration).

- **required_acks (string)** The level of acknowledgement reliability needed from the broker. The valid values are *NoResponse*, *WaitForLocal*, *WaitForAll*. Default is WaitForLocal.
- **timeout (uint32)** The maximum duration the broker will wait for the receipt of the number of RequiredAcks (in milliseconds). This is only relevant when RequiredAcks is set to WaitForAll. Default is no timeout.
- **compression_codec (string)** The type of compression to use on messages. The valid values are *None*, *GZIP*, *Snappy*. Default is None.
- **max_buffer_time (uint32)** The maximum duration to buffer messages before triggering a flush to the broker (in milliseconds). Default is 1.
- **max_buffered_bytes (uint32)** The threshold number of bytes buffered before triggering a flush to the broker. Default is 1.
- **back_pressure_threshold_bytes (uint32)** The maximum number of bytes allowed to accumulate in the buffer before back-pressure is applied to QueueMessage. Without this, queueing messages too fast will cause the producer to construct requests larger than the MaxRequestSize (100 MiB). Default is 50 * 1024 * 1024 (50 MiB), cannot be more than (MaxRequestSize - 10 KiB).

Example (send various Fxa messages to a static Fxa topic):

```
[FxaKafkaOutput]
type = "KafkaOutput"
message_matcher = "Logger == 'FxaAuthWebserver' || Logger == 'FxaAuthServer'"
topic = "Fxa"
addrs = ["localhost:9092"]
encoder = "ProtobufEncoder"
```

2.32.10 Log Output

Plugin Name: **LogOutput**

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

Config:

<none>

Example:

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

2.32.11 Nagios Output

Plugin Name: **NagiosOutput**

Specialized output plugin that listens for Nagios external command message types and delivers passive service check results to Nagios using either HTTP requests made to the Nagios cmd.cgi API or the use of the *send_ncsa* binary. The message payload must consist of a state followed by a colon and then the message e.g., "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.

Config:

- **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 empty string.
- **password (string, optional):** Password used to authenticate with the Nagios web interface. Defaults to empty string.
- **response_header_timeout (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.
- **nagios_service_description (string, optional):** Must match Nagios service's service_description attribute. Defaults to the name of the output.
- **nagios_host (string, optional):** Must match the hostname of the server in nagios. Defaults to the Hostname attribute of the message.
- **send_nsca_bin (string, optional):** New in version 0.5.

  Use send_nsca program, as provided, rather than sending HTTP requests. Not supplying this value means HTTP will be used, and any other send_nsca_* settings will be ignored.
- **send_nsca_args ([]string, optional):** New in version 0.5.

  Arguments to use with send_nsca, usually at least the nagios hostname, e.g. *["-H", "nagios.somehost.com"]*. Defaults to an empty list.
- **send_nsca_timeout (int, optional):** New in version 0.5.

  Timeout for the send_nsca command, in seconds. Defaults to 5.
- **use_tls (bool, optional):** New in version 0.5.

  Specifies whether or not SSL/TLS encryption should be used for the TCP connections. Defaults to false.
- **tls (TlsConfig, optional):** New in version 0.5.

  A sub-section that specifies the settings to be used for any SSL/TLS encryption. This will only have any impact if *use_tls* is set to true. See *Configuring TLS*.

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:

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

2.32.12 Sandbox Output

New in version 0.9.

Plugin Name: **SandboxOutput**

The SandboxOutput provides a flexible execution environment for data encoding and transmission without the need to recompile Heka. See *Sandbox*. Config:

- The common output configuration parameter 'encoder' is ignored since all data transformation should happen in the plugin.
- *Common Sandbox Parameters*

- **timer_event_on_shutdown (bool):** True if the sandbox should have its timer_event function called on shutdown.

Example

```
[SandboxFileOutput]
type = "SandboxOutput"
filename = "fileoutput.lua"

[SandboxFileOutput.config]
path = "mylog.txt"
```

2.32.13 SMTP Output

Plugin Name: **SmtpOutput**

New in version 0.5.

Outputs a Heka message in an email. The message subject is the plugin name and the message content is controlled by the payload_only setting. The primary purpose is for email alert notifications e.g., PagerDuty.

Config:

- **send_from (string)** The email address of the sender. (default: "heka@localhost.localdomain")
- **send_to (array of strings)** An array of email addresses where the output will be sent to.
- **subject (string)** Custom subject line of email. (default: "Heka [SmtpOutput]")
- **host (string)** SMTP host to send the email to (default: "127.0.0.1:25")
- **auth (string)** SMTP authentication type: "none", "Plain", "CRAMMD5" (default: "none")
- **user (string, optional)** SMTP user name
- **password (string, optional)** SMTP user password

New in version 0.9.

- **send_interval (uint, optional)** Minimum time interval between each email, in seconds. First email in an interval goes out immediately, subsequent messages in the same interval are concatenated and all sent when the interval expires. Defaults to 0, meaning all emails are sent immediately.

Example:

```
[FxaAlert]
type = "SmtpOutput"
message_matcher = "((Type == 'heka.sandbox-output' && Fields[payload_type] == 'alert') || Type == 'he
send_from = "heka@example.com"
send_to = ["alert@example.com"]
auth = "Plain"
user = "test"
password = "testpw"
host = "localhost:25"
encoder = "AlertEncoder"
```

2.32.14 TCP Output

Plugin Name: **TcpOutput**

Output plugin that delivers Heka message data 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, or to any other arbitrary listening TCP server that knows how to process the encoded data.

Config:

- **address (string):** An IP address:port to which we will send our output data.
- **use_tls (bool, optional):** Specifies whether or not SSL/TLS encryption should be used for the TCP connections. Defaults to false.

New in version 0.5.

- **tls (TlsConfig, optional):** A sub-section that specifies the settings to be used for any SSL/TLS encryption. This will only have any impact if `use_tls` is set to true. See *Configuring TLS*.

New in version 0.6.

- **local_address (string, optional):** A local IP address to use as the source address for outgoing traffic to this destination. Cannot currently be combined with TLS connections.
- **encoder (string, optional):** Specifies which of the registered encoders should be used for converting Heka messages to binary data that is sent out over the TCP connection. Defaults to the always available "ProtobufEncoder".
- **use_framing (bool, optional):** Specifies whether or not the encoded data sent out over the TCP connection should be delimited by Heka's *Stream Framing*. Defaults to true if a ProtobufEncoder is used, false otherwise.
- **keep_alive (bool):** Specifies whether or not TCP keepalive should be used for established TCP connections. Defaults to false.
- **keep_alive_period (int):** Time duration in seconds that a TCP connection will be maintained before keepalive probes start being sent. Defaults to 7200 (i.e. 2 hours).

New in version 0.10.

- **use_buffering (bool, optional):** Buffer records to a disk-backed buffer on the Heka server before sending them out over the TCP connection. Defaults to true.
- **buffering (QueueBufferConfig, optional):** All of the *buffering* config options are set to the standard default options, except for *cursor_update_count*, which is set to 50 instead of the standard default of 1.

Example:

```
[aggregator_output]
type = "TcpOutput"
address = "heka-aggregator.mydomain.com:55"
local_address = "127.0.0.1"
message_matcher = "Type != 'logfile' && Type !~ /^heka\./'"
```

2.32.15 UDP Output

New in version 0.7.

Plugin Name: **UdpOutput**

Output plugin that delivers Heka message data to a specified UDP or Unix datagram socket location.

Config:

- **net (string, optional):** Network type to use for communication. Must be one of "udp", "udp4", "udp6", or "unixgram". "unixgram" option only available on systems that support Unix datagram sockets. Defaults to "udp".
- **address (string):** Address to which we will be sending the data. Must be IP:port for net types of "udp", "udp4", or "udp6". Must be a path to a Unix datagram socket file for net type "unixgram".
- **local_address (string, optional):** Local address to use on the datagram packets being generated. Must be IP:port for net types of "udp", "udp4", or "udp6". Must be a path to a Unix datagram socket file for net type "unixgram".
- **encoder (string):** Name of registered encoder plugin that will extract and/or serialized data from the Heka message.
- **max_message_size (int):** Maximum size of message that is allowed to be sent via UdpOutput. Messages which exceeds this limit will be dropped. Defaults to 65507 (the limit for UDP packets in IPv4).

Example:

```
[PayloadEncoder]

[UdpOutput]
address = "myserver.example.com:34567"
encoder = "PayloadEncoder"
```

2.32.16 Whisper Output

Plugin Name: **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.

Config:

- **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"
```

2.32.17 See Also

hekad(1), hekad.config(5)

2.33 hekad

2.33.1 Synopsis

hekad [`-version`] [`-config` *config_file*]

2.33.2 Description

Heka is an open source stream processing software system developed by Mozilla. Heka is a "Swiss Army Knife" type tool for data processing, useful for a wide variety of different tasks, such as:

- Loading and parsing log files from a file system.
- Accepting statsd type metrics data for aggregation and forwarding to upstream time series data stores such as graphite or InfluxDB.
- Launching external processes to gather operational data from the local system.
- Performing real time analysis, graphing, and anomaly detection on any data flowing through the Heka pipeline.
- Shipping data from one location to another via the use of an external transport (such as AMQP) or directly (via TCP).
- Delivering processed data to one or more persistent data stores.

The following resources are available to those who would like to ask questions, report problems, or learn more:

- Mailing List: https://mail.mozilla.org/listinfo/heka
- Issue Tracker: https://github.com/mozilla-services/heka/issues
- Github Project: https://github.com/mozilla-services/heka/
- IRC: #heka channel on irc.mozilla.org

Heka is a heavily plugin based system. Common operations such as adding data to Heka, processing it, and writing it out are implemented as plugins. Heka ships with numerous plugins for performing common tasks.

There are six different types of Heka plugins:

Inputs

Input plugins acquire data from the outside world and inject it into the Heka pipeline. They can do this by reading files from a file system, actively making network connections to acquire data from remote servers, listening on a network socket for external actors to push data in, launching processes on the local system to gather arbitrary data, or any other mechanism.

Input plugins must be written in Go.

Splitters

Splitter plugins receive the data that is being acquired by an input plugin and slice it up into individual records. They must be written in Go.

Decoders

Decoder plugins convert data that comes in through the Input plugins to Heka's internal Message data structure. Typically decoders are responsible for any parsing, deserializing, or extracting of structure from unstructured data that needs to happen.

Decoder plugins can be written entirely in Go, or the core logic can be written in sandboxed Lua code.

Filters

Filter plugins are Heka's processing engines. They are configured to receive messages matching certain specific characteristics (using Heka's *Message Matcher Syntax*) and are able to perform arbitrary monitoring, aggregation, and/or processing of the data. Filters are also able to generate new messages that can be reinjected into the Heka pipeline, such as summary messages containing aggregate data, notification messages in cases where suspicious anomalies are detected, or circular buffer data messages that will show up as real time graphs in Heka's dashboard.

Filters can be written entirely in Go, or the core logic can be written in sandboxed Lua code. It is also possible to configure Heka to allow Lua filters to be dynamically injected into a running Heka instance without needing to reconfigure or restart the Heka process, nor even to have shell access to the server on which Heka is running.

Encoders

Encoder plugins are the inverse of Decoders. They generate arbitrary byte streams using data extracted from Heka Message structs. Encoders are embedded within Output plugins; Encoders handle the serialization, Outputs handle the details of interacting with the outside world.

Encoder plugins can be written entirely in Go, or the core logic can be written in sandboxed Lua code.

Outputs

Output plugins send data that has been serialized by an Encoder to some external destination. They handle all of the details of interacting with the network, filesystem, or any other outside resource. They are, like Filters, configured using Heka's *Message Matcher Syntax* so they will only receive and deliver messages matching certain characteristics.

Output plugins must be written in Go.

Information about developing plugins in Go can be found in the *Extending Heka* section. Details about using Lua sandboxes for Decoder, Filter, and Encoder plugins can be found in the *Sandbox* section.

2.33.3 Options

`-version` Output the version number, then exit.

`-config` ***config_path*** Specify the configuration file or directory to use; the default is /etc/hekad.toml. If *config_path* resolves to a directory, all files in that directory must be valid TOML files. (See hekad.config(5).)

2.33.4 Files

/etc/hekad.toml configuration file

2.33.5 See Also

hekad.config(5), hekad.plugin(5)

CHAPTER 3

Indices and tables

- search
- *Glossary*
- *Changelog*

H

M

P

R