heka-node Documentation

Release 0.5

Rob Miller, Victor Ng

March 22, 2016

Contents

heka-node is a Node.js client for the "Heka" system of application logging and metrics gathering developed by the Mozilla Services team. The Heka system is meant to make life easier for application developers with regard to generating and sending logging and analytics data to various destinations. It achieves this goal (we hope!) by separating the concerns of message generation from those of message delivery and analysis. Front end application code no longer has to deal directly with separate back end client libraries, or even know what back end data storage and processing tools are in use. Instead, a message is labeled with a type (and possibly other metadata) and handed to the Heka system, which then handles ultimate message delivery.

More information about how Mozilla Services is using Heka (including what is being used for a router and what endpoints are in use / planning to be used) can be found in the heka-docs repository on github.

You can find a pre-rendered version of that documentation on readthedocs.org at heka-docs.rtfd.org.

A pre-rendered version of the heka-node documentation can be found on readthedocs.org at heka-node.rtfd.org.

The primary component to the heka-node library, is the Basic Usage client class which exposes a *clientFromJsonConfig* factory function that will generate a configured client.

The HekaClient should be instantiated with the factory function.

Folks new to using Heka will probably find Heka Configuration a good place to get started.

CHAPTER 1

Welcome to heka-node's documentation!

Contents:

1.1 Getting Started

There are two primary components with which users of the heka-node library should be aware. The first is the Basic Usage clientFromJsonConfig factory function.

The HekaClient exposes the Heka API, and is generally your main point of interaction with the Heka system. The client doesn't do very much, however; it just provides convenience methods for constructing messages of various types and then passes the messages along. Actual message delivery is handled by a *stream*. Without a properly configured stream, a HekaClient is useless.

1.2 Heka Configuration

To assist with getting a working Heka set up, heka-node provides a Basic Usage module which will take declarative configuration info in JSON format and use it to configure a HekaClient instance.

1.2.1 JSON format

The *clientFromJsonConfig* function of the config module is used to create a HekaClient instance.

A minimal configuration that will instantiate a working Heka client may look like this

```
var heka = require('heka');
var heka_CONF = {
    'stream': {'factory': 'heka/streams:udpStreamFactory',
               'hosts': ['localhost'],
               'ports': [5565]
     },
    'logger': 'test',
    'severity': heka.SEVERITY.INFORMATIONAL
};
var jsonConfig = JSON.stringify(heka_CONF);
var log_client = heka.clientFromJsonConfig(jsonConfig);
```

There are several optional parameters you may use to specialize the heka-node client. A detailed description of each option follows:

logger Each heka message that goes out contains a *logger* value, which is simply a string token meant to identify the source of the message, usually the name of the application that is running. This can be specified separately for each message that is sent, but the client supports a default value which will be used for all messages that don't explicitly override. The *logger* config option specifies this default value. This value isn't strictly required, but if it is omitted '' (i.e. the empty string) will be used, so it is strongly suggested that a value be set.

severity Similarly, each heka message specifies a *severity* value corresponding to the integer severity values defined by RFC 3164. While each message can set its own severity value, if one is omitted the client's default value will be used. If no default is specified here, the default default (how meta!) will be 6, "Informational".

disabledTimers Heka natively supports "timer" behavior, which will calculate the amount of elapsed time taken by an operation and send that data along as a message to the back end. Each timer has a string token identifier. Because the act of calculating code performance actually impacts code performance, it is sometimes desirable to be able to activate and deactivate timers on a case by case basis. The *disabledTimers* value specifies a set of timer ids for which the client should NOT actually generate messages. Heka will attempt to minimize the run-time impact of disabled timers, so the price paid for having deactivated timers will be very small. Note that the various timer ids should be newline separated.

filters You can configure client side filters to restrict messages from going to the server.

1.2.2 Setting up a heka-node client

The following snippet demonstrates setting up a minimal heka-node client that writes out protocol buffer formatted messages to localhost on port 5565.

```
var heka = require('heka');
var config = {
    'stream': {'factory': 'heka/streams:udpStreamFactory',
               'hosts': ['localhost'],
               'ports': [5565]
    },
    'logger': 'test',
    'severity': heka.SEVERITY.INFORMATIONAL
};

var jsonConfig = JSON.stringify(config);
var client = heka.clientFromJsonConfig(jsonConfig);
```

1.2.3 Streams

The heka client supports different kinds of output streams.

Each stream allows at least the one parameter *hmc* which specifies the kind of HMAC signature to use when signing messages. By default, *hmc* is set to null and no signatures will be written into the header portion of the serialized message.

1.2.4 debugStreamFactory

Buffers messages into a list within the stream. This is useful if you want to capture your own messages for inspection within a unit test suite. Example usage can be found in the heka-node testsuite.

No extra configuartion parameters are supported.

Sample configuration

```
var heka = require('heka');
var config = {
    'stream': {'factory': 'heka/streams:debugStreamFactory'},
    'logger': 'test',
    'severity': heka.SEVERITY.INFORMATIONAL
};
```

1.2.5 fileStreamFactory

Write messages out into a filepath. The parent directory of the file must exist.

filepath is a required parameter. The parent directory of *filepath* must exist or the heka-client will error out during initialization.

Sample configuration

```
var heka = require('heka');
var config = {
    'stream': {'factory': 'heka/streams:fileStreamFactory',
               'filepath': '/tmp/some_output_file.txt'},
    'logger': 'test',
    'severity': heka.SEVERITY.INFORMATIONAL
};
```

1.2.6 stdoutStreamFactory

Writes messages directly to stdout. This is probably not useful to most people as all messages are serialized to protocolbuffer prior to being written to a stream. This output stream may be useful if you implement an encoder to replace the ProtobufEncoder.

No extra configuration parameters are supported.

Sample configuration

```
var heka = require('heka');
var config = {
    'stream': {'factory': 'heka/streams:stdoutStreamFactory'},
    'logger': 'test',
    'severity': heka.SEVERITY.INFORMATIONAL
};
```

1.2.7 udpStreamFactory

Writes messages to one or more hosts.

udpStreamFactory expects *hosts* and *ports* to be defined.

Sample configuration

```
var heka = require('heka');
var config = {
    'stream': {'factory': 'heka/streams:udpStreamFactory',
               'hosts': ['localhost'],
               'ports': [5565],
    },
    'logger': 'test',
```

```
        'severity': heka.SEVERITY.INFORMATIONAL
    };
```

1.2.8 Filters

Filters can be used to suppress the client from emitting messages which match specific criteria. We currently provide the following filters :

typeBlacklistProvider Suppress any messages where the *type* attribute matches one of the *types* in the provider.

Sample Configuration :: var config = {'types': {'foo': {'severity': 3}}};

typeWhitelistProvider Only allow messages to pass through where the *type* matches one of the *types* in the provider.

severityMaxProvider Only allow message to pass through if the severity of the message is strictly greater than the *severity* in the provider.

typeSeverityMaxProvider Given a dictionary of type to severity, only allow message to pass through for a given type if the severity of the message is strictly greater than the one specified in the configuration.

For messages where the *type* is not specified, allow the message through regardless of the severity.

Example usage for each of these filter is available in the filters.spec.js testsuite

1.2.9 Disabling Timers

The heka client will let you disable calls to the *timer()* method. Each call to *timer()* requires a timer name in the second positional argument. Passing in a list of names, or a wildcard ('*') will disable any timer calls where the timer name matches at least one of the disabled timer names.

The configuration expects either a list of message *type* names which match timer messages that will be excluded. You can also use a wildcard * to disable all timer code.

Example configuration

```
var config = {
    'stream': {'factory': 'heka/streams:debugStreamFactory'},
    'logger': 'test',
    'severity': 5,
    'disabledTimers': ['some_disabled_type'],
};
var jsonConfig = JSON.stringify(config);
var client = configModule.clientFromJsonConfig(jsonConfig);
```

1.2.10 Plugins

Plugins can be bound to the heka-node client using the *plugins* key of the configuration dictionary. You must provide at least a *provider* key which will be resolved into a factory function to bind a new method onto the heka-node client. Any additional key/value pairs in the plugin configuration are passed into the factory function to configure the plugin.

Example configuration

```
var config = {
    'stream': {'factory': 'heka/streams:debugStreamFactory'},
    'logger': 'test',
    'severity': 5,
    'plugins': {'showLogger': {'provider': './tests/plugins.spec.js:showLoggerProvider',
```

```
                                'label': 'some_custom_label'}}
};
var jsonConfig = JSON.stringify(config);
var client = configModule.clientFromJsonConfig(jsonConfig);
```

1.3 Basic Usage

After instantiating your logger, you can start sending counters and timers using the heka-node client.

1.3.1 Increment counters

Counters increment a named value. You may increment the counter by values other than 1, and you can take random samples instead of sending an increment message every time.

Typically, these messages go into statsd.

Signature

```
incr(name, opts={}, sample_rate=1.0)
```

The simplest way to increment a counter is to simply name the counter

log.incr('demo.node.incr_thing');

Options:

The incr() method takes two optional arguments, an opts dictionary and a sample_rate.

name is the name of the counter you are incrementing.

The opts dictionary may include an integer *count* and dictionary *fields*. *count* represents the number to increment by in case you want to increase the counters by more than the default of 1.

fields is a dictionary of data. By default, heka-node will create this dictionary for you and autopopulate the name and sampling rate into the dictionary for you. If you supply fields, you can supply additional key value pairs to store into fields.

If the sample_rate is supplied, it must be a float from 0 to 1.0. heka-node will compute a random number. If the random number is greater than the sample_rate, then *no* message is delivered.

Note that because Javascript has only a single number type, you will need to use the *heka.BoxedFloat* type to ensure that the *sample_rate* is properly encoded into a double precision number to protect against a rate of 1.0 being encoded into an integer.

Example

```
log.incr('some_counter', {count: 2, my_meta: 42}, new heka.BoxedFloat(0.25))
```

Will send a message ~25% of the time to hekad. Each increment will increase the count of 'some_counter' by 2 and will also send a field 'my_meta' with a value of 42 to the server.

1.3.2 Timers

The timer method provides a way to decorate functions so that you will emit timing messages whenever the function is invoked.

Typically, these messages go into statsd.

Signature

```
timer(fn, name, opts={})
```

Options:

fn is the function to be called *name* is the name of the event you are measuring. *opts* may contain a *rate* attribute which specifies a sampling rate for timer messages.

The return value of *timer* is your decorated function.

Note that because Javascript has only a single number type, you will need to use the *heka.BoxedFloat* type to ensure that the *rate* is properly encoded into a double precision number to protect against a rate of 1.0 being encoded into an integer.

1.3.3 Encoders

The heka wire format for 0.2 currently uses ProtocolBuffers to encode the header and you may use ProtocolBuffer or JSON to encode the payload.

At this time, please use the JSON encoder only. There are known bugs when the ProtocolBuffer encoder is applied to the payload of the message body.

1.3.4 Example

heka-node includes a complete example that exercises the *timer()* and the *incr()* methods using a thing HTTP REST API server. You can find the source in the heka-node repository on github in the example directory.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search