

JAMES TURNBULL

THE
ART
OF
MONITORING



The Art of Monitoring

James Turnbull

March 25, 2019

Version: v1.0.4 (9364dd6)

Website: [The Art of Monitoring](#)



Some rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical or photocopying, recording, or otherwise, for commercial purposes without the prior permission of the publisher.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit [here](https://creativecommons.org/licenses/by-nc-nd/3.0/).

© Copyright 2018 - James Turnbull <james@lovedthanlost.net>



Contents

	Page
Chapter 3 Managing events and metrics with Riemann	1
Introducing Riemann	2
Riemann architecture and implementation	3
Installing Riemann	4
Configuring Riemann	12
Learning some Clojure	13
Riemann's base configuration	13
Events, streams, and the index	19
Configuring events, streams, and the index	22
Sending our first event to Riemann	27
Creating our first Riemann monitoring check	30
An interlude into Riemann filtering	31
Connecting Riemann servers	35
Configuring the upstream Riemann servers	37
Configuring the downstream Riemann server	41
Enabling the send of our Riemann events downstream	42
Alerting on the upstream Riemann servers	44
Throttling Riemann events	54
Rolling up Riemann events	54
Alternatives to email notifications	55
Testing your Riemann configuration	56
Validating Riemann configuration	60

Performance, scaling, and making Riemann highly available	61
Alternatives to Riemann	64
Summary	65
Appendix A An Introduction to Clojure and Functional Programming	66
A brief introduction to Clojure	68
Installing Leiningen	68
Clojure syntax and types	70
Clojure functions	71
Lists	75
Vectors	77
Sets	79
Maps	81
Strings	84
Creating our own functions	84
Creating variables	86
Creating named functions	88
Learning more Clojure	90
List of Figures	92
List of Listings	97
Index	98

Chapter 3

Managing events and metrics with Riemann

In Chapter 2 we talked about events, metrics, and logs, and how we're going to use them. In this chapter we're going to build the base of our monitoring framework: the routing engine we described that will input and process those events, metrics, and logs.

Our design for an event router is:

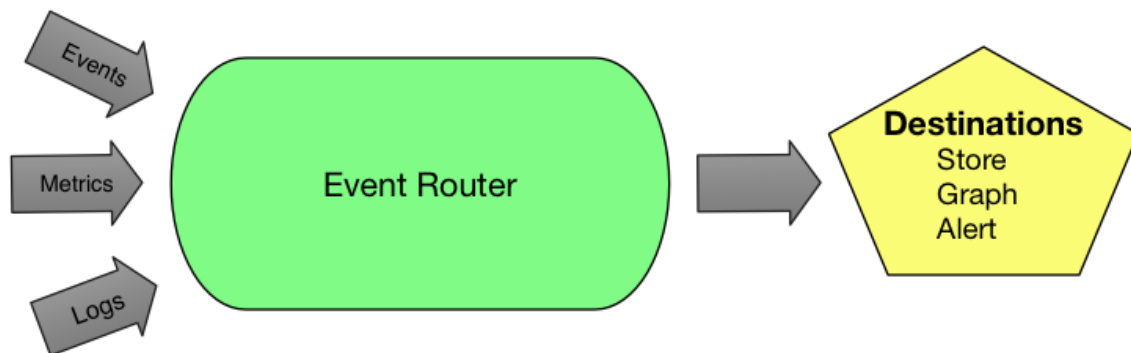


Figure 3.1: Event Routing

With this design we want our routing engine to:

- Receive our events and metrics (we'll talk more about logs in Chapter 8) including scaling as our environment grows.
- Maintain sufficient state to allow us to do event matching; provide context for notifications and for checks based on trending.
- Munge events including extracting metrics from events.
- Categorize and route data to be stored, graphed, alerted on, or sent to any other potential destinations.

To achieve these objectives we're going to look at a tool called [Riemann](#). Riemann is an event-based tool for monitoring distributed systems. It works in a push model, receiving events rather than polling for them. We're going to use Riemann as our routing engine. Our hosts, services, and applications will send their events into Riemann, and Riemann will make the necessary decisions about those events.

Introducing Riemann

If only I had the theorems! Then I should find the proofs easily enough.

— Bernard Riemann

So why Riemann? [Riemann](#) is a monitoring tool that aggregates events from hosts and applications and can feed them into a stream processing language to be manipulated, summarized, or actioned. The idea behind Riemann is to make monitoring and measuring events an easy default.

Riemann can also track the state of incoming events and allows us to build checks that take advantage of sequences or combinations of events. It provides notifications, the ability to send events onto other services and into storage, and a variety of other integrations.

Overall, Riemann has functionality that addresses all of our objectives. It is fast and highly configurable. Throughput depends on what you do with each event,

but stock Riemann on commodity x86 hardware can handle millions of events per second at sub-millisecond latencies.

Riemann is [open source](#) and licensed with the [Eclipse Public license](#). It is primarily authored by [Kyle Kingsbury](#) aka Aphyr. Riemann is written in Clojure and runs on top of the [JVM](#).

Riemann architecture and implementation

In the Introduction we talked a little about the topology of our Example.com environment with Production A, Production B, and Mission Control environments. We're going to deploy Riemann servers in each environment. We're also going to deploy downstream servers in the Mission Control environment to allow us to roll up events and "monitor the monitors." To do this we'll send Riemann's own status events downstream, and we'll setup notifications if these events stop flowing.

In Chapter 4 we're going to pair Riemann with Graphite, a real-time time-series data graphing engine, that can store and graph metrics generated from events. We'll also introduce Grafana, a tool to visualize the data we're collecting.

In summary we will:

- Install Riemann servers in the Production A and Production B environments.
- Install Graphite and Grafana servers in the Production A and Production B environments (coming up in Chapter 4).
- Configure the downstream Riemann, Graphite, and Grafana servers in our Mission Control environment.

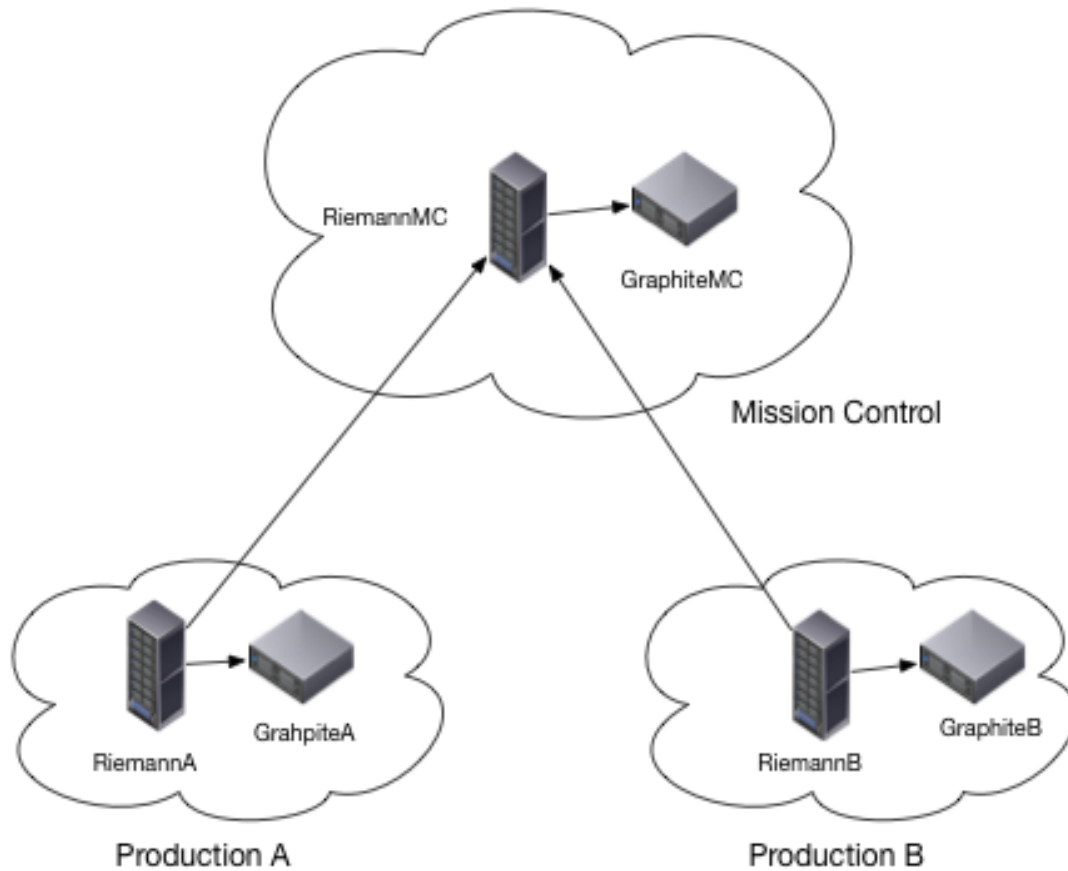


Figure 3.2: Metrics Architecture

Let's take a look at installing Riemann now.

Installing Riemann

We're going to install Riemann onto three hosts:

- An Ubuntu 14.04 host in Production A with a hostname of `riemanna.example.com` and an IP address of 10.0.0.110
- A Red Hat Enterprise Linux (RHEL) 7.0 host in Production B with a hostname of `riemannb.example.com` and an IP address of 10.0.0.120.

- An Ubuntu 14.04 host in Mission Control with a hostname of `riemannmc.example.com` and an IP address of 10.0.0.100.

NOTE Here, and throughout the book, we're going to assume you've configured DNS and local host resolution to find and resolve these hosts.

We're going to conduct a manual installation of the required prerequisites and packages to give you an understanding of how Riemann works, but in the real world we'd use a configuration management tool.

Installing Riemann on Ubuntu

We're going to do our first Riemann installation on the `riemanna.example.com` host which is an Ubuntu 14.04 host.

Prerequisites for Ubuntu

First, we'll need Java to run Riemann itself. For Java we're going to use the default OpenJDK available on Ubuntu.

Listing 3.1: Installing Java on Ubuntu

```
$ sudo apt-get -y install default-jre
```

Then let's check Java is installed correctly.

Listing 3.2: Checking Java is installed on Ubuntu

```
$ java -version
java version "1.7.0_65"
OpenJDK Runtime Environment (IcedTea 2.5.3) (7u71-2.5.3-0ubuntu0
.14.04.1)
OpenJDK 64-Bit Server VM (build 24.65-b04, mixed mode)
```

Installing the Riemann package on Ubuntu

Now we're going to install Riemann itself. We're going to use [the Riemann project's own DEB packages](#). Also available are RPM packages and tarballs. I am going to do a manual install so you can see the steps involved, but you could just as easily use the configuration management content above.

Let's grab the DEB package of the current release. You should check the Riemann site for the latest version and update this command to get that package.

Listing 3.3: Fetching the Riemann DEB package

```
$ wget https://aphyr.com/riemann/riemann_0.2.11_all.deb
```

And then install it via the `dpkg` command.

Listing 3.4: Installing the Riemann package on Ubuntu

```
$ sudo dpkg -i riemann_0.2.11_all.deb
```

The Riemann DEB package installs the `riemann` binary and supporting files, service management, and a default configuration file.

We'd then repeat this installation for the second Ubuntu host, `riemannmc.example.com`.

Installing Riemann on Red Hat


We're going to do our second installation on the `riemannb.example.com` host, which is a Red Hat Enterprise Linux (RHEL) 7 host.

Prerequisites for Red Hat

Again we need to have Java installed. Let's install the package we require.

Listing 3.5: Installing Java and prerequisites on RHEL

```
$ sudo yum install -y java-1.7.0-openjdk
```

 **TIP** On newer Red Hat and family versions the `yum` command has been replaced with the `dnf` command. The syntax is otherwise unchanged.

Here we've installed Java. Let's now test Java on this host.

Listing 3.6: Checking Java is installed on Red Hat

```
$ java -version
java version "1.7.0_75"
OpenJDK Runtime Environment (rhel-2.5.4.2.el7_0-x86_64 u75-b13)
OpenJDK 64-Bit Server VM (build 24.75-b04, mixed mode)
```

Installing the Riemann package on Red Hat

Now we're going to install Riemann itself. We're going to use [the Riemann project's own RPM packages](#). Again we're going to do a manual install but we could easily use configuration management.

Let's grab the RPM package of the current release. Check the Riemann site for the latest version and update this command to get that package.

Listing 3.7: Fetching the Riemann RPM package

```
$ wget https://aphyr.com/riemann/riemann-0.2.11-1.noarch.rpm
```

Then install it via the `rpm` command.

Listing 3.8: Installing the Riemann package on RHEL

```
$ sudo rpm -Uvh riemann-0.2.11-1.noarch.rpm
```

The Riemann RPM package installs the `riemann` binary and supporting files, service management, and a default configuration file.

Installing Riemann via configuration management

You could also install Riemann via a variety of configuration management tools like Puppet or Chef or via Docker or Vagrant.

You can find a Chef cookbook for Riemann at:

- <https://github.com/hudl/riemann-cookbook>

You can find a Puppet module for Riemann at:

- <https://forge.puppetlabs.com/garethr/riemann>

You can find an Ansible role for Riemann at:

- <https://github.com/dhruvbansal/riemann-server-ansible-role>

You can find Docker images for Riemann at:

- <https://hub.docker.com/search/?q=riemann>

You can find a Vagrant configuration for Riemann at:

- <https://github.com/garethr/riemann-vagrant>

Running Riemann

Now that we've installed Riemann on our hosts, we'll run it. Riemann can run interactively via the command line or as a daemon. If we're running it as a daemon we use the service management commands:

Listing 3.9: Starting and stopping Riemann

```
$ sudo service riemann start
$ sudo service riemann stop
. . .
```

We can also run Riemann interactively using the `riemann` binary. To do this we need to specify a configuration file. Conveniently the installation process has added one at `/etc/riemann/riemann.config`.

Listing 3.10: Running Riemann interactively

```
$ sudo riemann /etc/riemann/riemann.config
loading bin
INFO [2014-12-21 18:13:21,841] main - riemann.bin - PID 18754
INFO [2014-12-21 18:13:22,056] clojure-agent-send-off-pool-2 -
riemann.transport.websockets - Websockets server 127.0.0.1 5556
online
INFO [2014-12-21 18:13:22,091] clojure-agent-send-off-pool-4 -
riemann.transport.tcp - TCP server 127.0.0.1 5555 online
INFO [2014-12-21 18:13:22,099] clojure-agent-send-off-pool-3 -
riemann.transport.udp - UDP server 127.0.0.1 5555 16384 online
INFO [2014-12-21 18:13:22,102] main - riemann.core - Hyperspace
core online
```

We see that Riemann has been started and a couple of servers have also been started: a WebSockets server on port 5556, and TCP and UDP servers on port 5555. By default Riemann binds to `localhost`.

NOTE Don't use UDP to send events to Riemann. UDP has no guarantee of delivery, ordering, or duplicate protection. You will lose events and data.

The default configuration logs to `/var/log/riemann/riemann.log` and you can also follow the daemon's activity there.

You can stop the interactive Riemann server with a `Ctrl-C` on the command line.

NOTE The Riemann packages also add a `riemann` user and group that Riemann runs by default.

Installing Riemann's supporting tools

Lastly, let's install a final supporting piece on all our Riemann hosts: the Riemann tools.

The Riemann tools are a collection of small programs that can be used to submit events to Riemann. They include tools for monitoring web services, local hosts, applications, and databases. We're going to use these tools for some local testing. You can see the repository for the Riemann tool on GitHub [here](#).

 **NOTE** In Chapter 5 we'll explore host monitoring using collectd.

To install the tools we need to install Ruby and a compiler on our host. We can remove it afterward if we're concerned it's a security risk on the host. On Ubuntu we would install:

Listing 3.11: Installing supporting tools prerequisites on Ubuntu

```
$ sudo apt-get -y install ruby ruby-dev build-essential zlib1g-dev
```

Or on Red Hat distributions we would install:


Listing 3.12: Installing supporting tools prerequisites on RHEL

```
$ sudo yum install -y ruby ruby-devel gcc libxml2-devel
```

We're going to install the supporting tools via Ruby Gems.

Listing 3.13: Installing Riemann’s supporting tools

```
$ sudo gem install --no-ri --no-rdoc riemann-tools
```

 **TIP** After installation you can remove the build tools we installed if required. Don’t remove Ruby—you’ll need that to run the supporting tools.


There’s also a Riemann dashboard available that is provided by the [riemann-dash](#) gem. It’s basic and you can find its source code [on GitHub](#). We’re not going to use it in the book, but it’s useful for creating graphs and viewing events locally on the Riemann host—for example, when quickly doing diagnostics or checking out state and status. You can find out more in [the Riemann Dashboard documentation](#).

Configuring Riemann

Riemann is configured using a Clojure-based domain-specific language, or DSL, configuration file. This means your configuration file is actually processed as a Clojure program. To process events and send notifications and metrics you’ll be writing Clojure. Don’t panic—you won’t need to become a full-fledged Clojure developer to use Riemann. We’ll teach you what you need to know. Riemann also comes with a lot of helpers and shortcuts that make it easier to write Clojure to do what we need to process our events.

Learning some Clojure

Your first step in learning how to configure Riemann is learning a little bit of Clojure, just enough to get started and build our first few monitoring checks. Later in the book we'll introduce you to further concepts and syntax that you'll find useful. To start learning, please flip over to **Appendix A - An Introduction to Clojure and Functional Programming** at the back of the book. Alternatively, Kyle Kingsbury, the author of Riemann, has written an excellent series called [Clojure from the ground up](#) that should greatly help you understand Clojure.

 **TIP** We **strongly** recommend you read the appendix before continuing. It'll help you understand how Riemann's configuration DSL works and help you get started using it.

Riemann's base configuration

Now that we've installed Riemann let's look at how to configure it. The package installation installs a default configuration file at `/etc/riemann/riemann.config`. We're going to replace that file with a new initial configuration.

To do this edit the `/etc/riemann/riemann.config` file and add the following content.


Listing 3.14: New `/etc/riemann/riemann.config` configuration file

```
(logging/init {:file "/var/log/riemann/riemann.log"})

(let [host "127.0.0.1"]
  (tcp-server {:host host})
  (udp-server {:host host})
  (ws-server  {:host host}))

(periodically-expire 5)


(let [index (index)]
  (streams
    (default :ttl 60
      index
      #(info %))))
```

 **TIP** Any line or string prefixed with a `;` is a comment.

We see the file is broken into a few stanzas. The first stanza sets up Riemann's logging to a file: `/var/log/riemann/riemann.log`.

Listing 3.15: Riemann logging stanza

```
(logging/init {:file "/var/log/riemann/riemann.log"})
```

 **TIP** I strongly recommend you manage your Riemann configuration file with a configuration management tool and/or version control. Also useful is if your

editor supports syntax highlighting and validation. Clojure uses a lot of braces, brackets, and parentheses and ensuring they are in order and matched can be tricky.

In this case we're calling a function, `logging/init`. We've specified the namespace of the function, `logging`, and the name of the function, `init`, and then any subsequent arguments. Namespaces are a way of organizing code in Clojure and we'll talk more about them later in this chapter. In this case our argument is a `map`. Our map contains any options we want to pass to our `logging/init` function.

Inside our map we've specified a single option, `:file`, with a value of `/var/log/riemann/riemann.log`. The `:file` option is a `Clojure keyword`. A keyword is a label, much like a Ruby symbol, and it's commonly used inside collections like maps to mark the key in a key/value pair.

In summary we're calling the `logging/init` function and passing it a map, in this case containing only one option: the name of the file in which to write our logs.

The second stanza controls Riemann's interfaces. Riemann generally listens on TCP, UDP, and a WebSockets interface. By default, the TCP, UDP, and WebSockets interfaces are bound to the `127.0.0.1` or `localhost`.

- TCP is on port 5555.
- UDP is on port 5555.
- WebSockets is on port 5556.

We see that the definition of our interface configuration is inside a stanza starting with `let`. We're going to see `let` quite a bit in our configuration. The `let` expression creates lexically scoped immutable aliases for values. Or, in more simple terms, it defines a meaning for a symbol or symbols within a specific expression.

What does this mean? Let's look at our interface configuration.

Listing 3.16: The let form

```
(let [host "127.0.0.1"]
  (tcp-server {:host host})
  (udp-server {:host host})
  (ws-server  {:host host}))
```

The `let` expression takes a [vector](#) of one or more bindings. Bindings are pairs of symbols and values that are bound for that expression. [Symbols are pointers to values](#)—for example, the symbol `host` has a value of `127.0.0.1`. The binding is only valid locally to our `let` expression. This is useful because it allows you to do things like override existing values of symbols inside the current expression.

We use these bindings in the subsequent expressions. In our interface example we’re saying:

“Let the symbol `host` be `127.0.0.1` and then call the `tcp-server`, `udp-server`, and `ws-server` functions with that symbol as the value of the `:host` option.”

This sets the host interface of the TCP, UDP, and WebSockets servers to `127.0.0.1`.

A `let` binding is lexically scoped, i.e., limited in scope to the expression itself. Outside of this expression the `host` symbol would be undefined. The `host` symbol is also immutable inside the expression in which it is defined. You cannot change the value of `host` inside this expression. This is an excellent example of why functional programming is useful. Nothing inside the expression can change the value (state) of `host`, which ensures that every time the expression is evaluated the same result will be achieved.

The `let` expression is useful for configuring Riemann because it is simple, readable, and—because of the clear scope and immutable state—reloads cleanly when we want to change configuration.

For our purposes having Riemann bound to only the `localhost` isn’t overly useful. Let’s make a quick change here to bind these servers to all available interfaces.

Listing 3.17: Exposing Riemann on all interfaces

```
(let [host "0.0.0.0"]
  (tcp-server {:host host})
  (udp-server {:host host})
  (ws-server  {:host host}))
```


We've updated the value of the `host` symbol from `127.0.0.1` to `0.0.0.0`. This means if one of your interfaces is on the Internet then your Riemann server is now on the Internet and accessible to everyone. If you need more security you can also [configure Riemann with TLS](#).

We could also adjust the ports being used by Riemann by adding the `:port` argument to our map.

Listing 3.18: Changing the Riemann port

```
(let [host "0.0.0.0"
      tcp-port 5555]
  (tcp-server {:host host :port tcp-port}))
```

In addition to the other servers, Riemann also has a built-in REPL server you can use to test your Riemann configuration. You can add it to the servers you've enabled by adding `(repl-server {:host host})` to your server configuration. You may want to bind it to the localhost as `(repl-server {:host "127.0.0.1"})` to prevent inappropriate access. You can connect to it using the `lein` binary from inside a checkout of the Riemann source code.

 **TIP** We talk about REPL servers in Appendix A. They are useful for testing with Riemann and Clojure.

Listing 3.19: Connecting to the Riemann REPL server

```
$ git clone git://github.com/riemann/riemann.git
$ cd riemann
$ lein repl :connect 127.0.0.1:5557
```

To make these changes we need to reload or restart Riemann. If we're reloading Riemann then it will respond to `SIGHUP`, using `kill -HUP <Riemann PID>` or from inside the REPL server.

Listing 3.20: SIGHUP from the Riemann REPL server

```
user=> (riemann.bin/reload!)
```

We strongly recommend using `SIGHUP` to reload the configuration. Riemann has hot loading of configuration and in most cases this will allow your event flow to be less impacted. You'll then see a message about reloading the config in the Riemann log file. You could also use the service management tools on your host.

Listing 3.21: Restarting Riemann

```
$ sudo service riemann reload
```

💡 TIP If you make a configuration mistake or a syntax error then Riemann will continue running with the old config and won't apply the new one. It'll also log an error message detailing the issue so you can fix it up.

The next stanza configures indexing and streams. Both of these topics need special attention because they are at the heart of what makes Riemann so powerful. Let's look at each of these concepts now.

Events, streams, and the index

Riemann is an event processing engine. There are three concepts we need to understand if we're going to make use of Riemann: events, streams, and the index. Let's start by looking at events.

Events

The event is the base construct of Riemann. Events flow into Riemann and can be processed, counted, collected, manipulated, or exported to other systems. A Riemann event is a [struct](#) that Riemann treats as an immutable map.

Here's an example of a Riemann event.

Listing 3.22: Example Riemann event

```
{:host riemanna, :service riemann streams rate, :state ok,  
:description nil, :metric 0.0, :tags [riemann],  
:time 355740372471/250, :ttl 20}
```

Each event generally contains the following fields.

Field	Description
host	A hostname, e.g. <code>riemanna</code> .
service	The service, e.g. <code>riemann streams rate</code> .

Field	Description
state	A string describing state, e.g. <code>ok</code> , <code>warning</code> , <code>critical</code> .
time	The time of the event in Unix epoch seconds.
description	Freeform description of the event.
tags	Freeform list of tags.
metric	A number associated with this event, e.g. the number of reqs/sec.
ttl	A floating-point time in seconds, for which this event is valid.

A Riemann event can also be supplemented with optional custom fields. You can configure additional fields when you create the event or you can add additional fields to the event as it is being processed—for example, you could add a field containing a summary or derived metrics to an event.

Inside our Riemann configuration we'll generally refer to an event field using keywords. Remember that keywords are often used to identify the key in a key/value pair in a map and that our event is an immutable map. We identify keywords by their `:` prefix. So, the `host` field would be referenced as `:host`.

The next layer above events are streams.

Streams

Each arriving event is added to one or more streams. You define streams in the `(streams` section of your Riemann configuration. Streams are functions you can pass events to for aggregation, modification, or escalation. Streams can also have child streams that they can pass events to. This allows for filtering or partitioning of the event stream, such as by only selecting events from specific hosts or services.

Listing 3.23: Child streams example

```
(streams
  (childstream
    (childstream)))
```

You can think of streams like plumbing in the real world. Events enter the plumbing system, flow through pipes and tunnels, collect in tanks and dams, and are filtered by grates and drains.

You can have as many streams as you like and Riemann provides a powerful stream processing language that allows you to select the events relevant to a specific stream. For example, you could select events from a specific host or service that meets some other criteria.

Like your plumbing though, streams are designed for events to flow through them and for limited or no state to be retained. For many purposes, however, we do need to retain some state. To manage this state Riemann has the index.

The Riemann index

The index is a table of the current state of all services being tracked by Riemann. You tell Riemann to specifically index events that you wish to track. Riemann creates a new service for each indexed event by mapping its `:host` and `:service` fields. The index then retains the most recent event for that service. You can think about the index as Riemann's worldview and source of truth for state. You can query the index from streams or even from external services.

We saw in our event definition above that each event can contain a TTL or Time-to-Live field. This field measures the amount of time for which an event is valid. Events in the index longer than their TTL are expired and deleted. For each expiration a new event is created for the indexed service with its `:state` field set to

`expired`. The new event is then injected back into the stream.

Let's take a closer look at this. Here's an example event:

Listing 3.24: Example Apache Riemann event

```
{:host www, :service apache connections, :state nil, :  
description nil, :metric 100.0, :tags [www], :time 466741572492,  
:ttl 20}
```

It's from a host called `www` and is for a service called `apache connections`. It has a TTL of 20 seconds. If we index this event then Riemann will create a service by mapping `www` and `apache connections`. If events keep coming into Riemann then the index will track the latest event from this service. If the events stop flowing then sometime after 20 seconds have passed the event will be expired in the index. A new event will be generated for this service with a `:state` of `expired`, like so:

Listing 3.25: Example expired Apache Riemann event

```
{:host www, :service apache connections, :state expired, :  
description nil, :metric 100.0, :time 466741573456, :ttl 20}
```

This event will then be injected back into streams where we can make use of it. This behavior is going to be pretty useful to us as we use Riemann for monitoring our applications and services. Instead of polling or checking for failed services, we'll monitor for services whose events have expired.

Configuring events, streams, and the index

Now that we know a bit more about Riemann let's take another look at the second half of our default configuration which contains our streams and an index

configuration.

Listing 3.26: More of our default riemann.config configuration file

```
(periodically-expire 5)

(let [index (index)]
  (streams
    (default :ttl 60
             index
             #(info %))))
```

The first function in our configuration, `(periodically-expire 5)`, removes any events that have expired from the index. It's an event reaper that runs every five seconds and acts on any events with expired TTL by deleting them from the index. For every event reaped a new event is created for that indexed host and service. That event is then put onto the stream with a `:state` of `expired`.

By default, Riemann copies the `:host` and `:service` fields to the expired event. You can control what other fields from events are also copied onto expired events by passing the `:keep-keys` option to the `periodically-expire` function. For example, we'd like to add the `:tags` field to expired events.

Listing 3.27: Copying more keys into expired events

```
(periodically-expire 5 {:keep-keys [:host :service :tags]})
```

This will copy the `:host`, `:service`, and `:tags` fields from the event being expired into the new event being injected into the stream.

The `let` expression that follows our `(periodically-expire)` function defines a new symbol called `index`. This symbol has a value of `index`, which is the func-

tion that sends events to Riemann’s index. We’re going to use this symbol to tell Riemann when to index specific events.

The `let` expression also wraps our streams. Next inside our `let` expression (note the bracket is not closed yet) we’ve specified that what follows are streams. We’ve done this using the `streams` function. Each stream is a Clojure function that takes an event. The `streams` function means “here is a list of functions that you should call when new events arrive”.

The first thing we’ve done inside our streams is to set a default TTL for our events of 60 seconds. We’ve done this using the `default` function. The `default` function takes a field from an event and allows you to specify a default value for that field.

Listing 3.28: Using the Riemann default function

```
(default :field default_value)
```

This TTL will determine how long an event will be valid within the index. In this case, after 60 seconds, events which do not already have a TTL will be expired.

Next the configuration calls our `index` symbol. This means all incoming events will be automatically added to Riemann’s index.

The last item in our configuration prints any events to our log file.

Listing 3.29: Logging to the Riemann log file

```
 #(info %)
```

The `info` function writes our event and some logging data to the `/var/log/riemann/riemann.log` log file and to `STDOUT`. You’ll see events like the following in the log file when Riemann is running:

Listing 3.30: A Riemann log event

```
INFO [2015-03-22 21:40:37,287] Thread-5 - riemann.config - #
riemann.codec.Event{:host riemanna, :service riemann streams
rate, :state nil, :description nil, :metric 7.739079374131467, :
tags [riemann], :time 1427060437213/1000, :ttl 20}
```

Also available is the `#{warn %}` function that emits events with a level of `WARN` rather than `INFO`.

You can adjust this logging output to log additional information, such as by adding a prefix for debugging.

Listing 3.31: Adding a prefix to Riemann logs entries

```
#{info "prefix" %}
```

This will prefix any log entries with the word `prefix`. You can also limit the log output to specific fields in an event, for example:

Listing 3.32: Limiting Riemann log entries

```
#{info (:host %) (:service %)}
```

This will only send the contents of the `:host` and `:service` fields to the log file.

Listing 3.33: A filtered Riemann log event

```
INFO [2015-03-22 21:55:35,172] Thread-6 - riemann.config -  
riemanna riemann streams rate
```

If you just want to print to `STDOUT` because, for example, you're running Riemann interactively to test something, you can use the `prn` function.

Listing 3.34: The Riemann `prn` function

```
; Print event to stdout  
prn  
  
; Print "output", then the event  
#(prn "Output: " %)
```

Now we need to reload Riemann to enable our new configuration.

Listing 3.35: Reloading Riemann to enable our new configuration

```
$ sudo service riemann reload
```

You should start to see events in your `/var/log/riemann/riemann.log` file. These events are Riemann's own internal status reporting.


Listing 3.36: Riemann internal events

```
INFO [2015-02-03 06:04:50,031] Thread-7 - riemann.config - #
riemann.codec.Event{:host riemanna, :service riemann streams
rate, :state nil, :description nil, :metric 0.0, :tags [riemann],
:time 355740372471/250, :ttl 20}
INFO [2015-02-03 06:04:50,034] Thread-7 - riemann.config - #
riemann.codec.Event{:host riemanna, :service riemann streams
latency 0.0, :state nil, :description nil, :metric nil, :tags [
riemann], :time 355740372471/250, :ttl 20}
INFO [2015-02-03 06:04:50,035] Thread-7 - riemann.config - #
riemann.codec.Event{:host riemanna, :service riemann streams
latency 0.5, :state nil, :description nil, :metric nil, :tags [
riemann], :time 355740372471/250, :ttl 20}
. . .
```

They include the rates and latency of your streams and TCP, UDP, and WebSockets servers. We can use this data to report on the state of Riemann.


Sending our first event to Riemann

Let's test that Riemann is receiving events from external sources too. You can send data to Riemann in a number of ways, including via its own set of tools and a variety of client native language bindings.

 **NOTE** You can find a full list of the clients [on the Riemann website](#).

The set of tools are written in Ruby and available via the `riemann-tools` gem we installed earlier. Each tool ships as a separate binary, and you can see a list of the available tools [in the riemann-tools repository on GitHub](#). They include basic

health checks, web services like Apache and Nginx, Cloud services like AWS, and more.

 **TIP** We can also query and send events to Riemann using the Riemann C client. You can install it on Ubuntu and Red Hat via the `riemann-c-client` package and use it via the `riemann-client` binary.

The easiest of these tools to test with is `riemann-health`. It sends CPU, memory, and load statistics to Riemann. Open up a new terminal and launch it now on a Riemann server.

Listing 3.37: The `riemann-health` command

```
$ riemann-health
```

You can either run it locally on the same host where you're running Riemann, or you can run it on a remote server and point it at a Riemann server using the `--host` flag.

Listing 3.38: The `riemann-health --host` option

```
$ riemann-health --host riemanna.example.com
```

The `riemann-health` command will now start emitting events into Riemann's TCP server on port `5555`, or you can specify an alternate port with the `--port` flag. In our current configuration we're sending all incoming events into Riemann's log file via the `$(info %)` function. Let's look at our incoming data in the Riemann log file: `/var/log/riemann/riemann.log`.

Listing 3.39: Our incoming Riemann data

```
$ tail -f /var/log/riemann/riemann.log
INFO [2015-12-23 17:23:47,050] pool-1-thread-16 - riemann.config
- #riemann.codec.Event{:host riemanna.example.com, :service
disk /, :state ok, :description 11% used, :metric 0.11, :tags
nil, :time 1419373427, :ttl 60.0}
INFO [2015-12-23 17:23:47,055] pool-1-thread-18 - riemann.config
- #riemann.codec.Event{:host riemanna.example.com, :service
load, :state ok, :description 1-minute load average/core is 0.11,
:metric 0.11, :tags nil, :time 1419373427, :ttl 60.0}
. . .
```

Here we see two events, one for disk space and another for load. Let's look at the event itself a bit more closely.

Listing 3.40: A Riemann-health disk event

```
{:host riemanna.example.com, :service disk /, :state ok,
:description 11% used, :metric 0.11, :tags nil,
:time 1419373427, :ttl 10.0}
```

Here we have an event from the host `riemanna.example.com` from the service `disk /`. This measures the disk space used on the root or `/` filesystem. It has a state of `ok`, a description that tells us the percentage of disk space consumed, and an associated metric with that percentage as a float, as well as the time the event was recorded. Lastly, the event has a Time-to-Live or TTL that controls for how long the event is valid.

Now we know our Riemann server is working and can receive events.

NOTE We're going to collect and monitor a lot more of these host-level

events and metrics in Chapters 5 and 6.

Creating our first Riemann monitoring check

Now we'll get to the core of why we're here. We're going to build a monitoring check using one of our `riemann-health` events. Let's open up our `/etc/riemann/riemann.config` configuration file and add our first check.

Listing 3.41: Our first monitoring check

```
(let [index (index)]
  (streams
    (default :ttl 60
      index

      ;#(info %)
      (where (and (service "disk /") (> metric 0.10))
        #(info "Disk space on / is over 10%!" %)))
```

Here we've added some new Clojure code for our first check. You'll note we've used a `;` to comment out the `#(info %)` function. This stops Riemann from emitting every event to the log file. Next we've specified a new stream called `where`. The `where` stream selects events based on criteria then passes them to a child stream where you can do something else with the event. In our `where` stream we are matching on two criteria, combined with a Boolean `and` statement:

- The `:service` field of the event is `disk /`.
- The `:metric` field of the event is greater than `0.10` or 10%.

If these two criteria match then the matched event is sent to the `#(info %)` function, so it can then be sent to the log file. We've also added a prefix to our log

message detailing why the event has been matched and outputted.

Let's look at what a matched and outputted event would look like in our `/var/log/riemann/riemann.log` log file.

Listing 3.42: Our prefixed warning event

```
Disk space on / is over 10%! #riemann.codec.Event{:host riemanna,  
:service disk /, :state ok, :description 24% used, :metric 0.24,  
:tags nil, :time 1449184188, :ttl 60.0}
```

We've stripped off some initial boilerplate from the event but you can see that our event has disk space usage of `24%` and hence been filtered by our `where` stream. It's been prefixed with our helpful explanatory message and then printed to the log file.

Congratulations—you've just built a Riemann monitoring check! Yes, this check is basic, doesn't go anywhere terribly useful, and isn't telling us anything critical about our environment. But it starts to show us what we can do with Riemann. We'll see many more complex checks in later chapters as we build out our monitoring environment. Now let's explore some more ways we can filter events.

An interlude into Riemann filtering

Before we go on, and because filtering is going to be key to how we manage events inside Riemann, let's look at some more examples of how to use filtering streams.

In our first example we're going to match events using regular expressions. It's a common use case and a good starting point.

Listing 3.43: Using the where stream with a regular expression

```
(where (service #"^nginx"))
```

In Clojure, regular expressions are prefixed with `#` and wrapped in double quotes. Here the `where` stream matches all events where the `:service` field starts with `nginx`.

We can also use Boolean operators to match events.

Listing 3.44: Using the where stream with booleans

```
(where (and (tagged "www") (state "ok")))
```

In this case our `where` stream matches events that are tagged with `www` and have a `:state` of `ok`. This example also combines the `where` stream and a new stream called `tagged`.

The `tagged` stream selects all events where the `:tags` field contains `www`. The `tagged` stream is shorthand for the `tagged-all` function which matches all tags specified. There's another function called `tagged-any` that allows you to match any one of a series of tags.

Listing 3.45: The tagged-any stream

```
(tagged-any ["www" "app1"]  
  #(info %))
```

Here we've used the `tagged-any` stream to match any event which has either the `www` or the `app1` tag, and then send the event to be logged.

We can also combine tags, booleans, and regular expressions to do complex matching.

Listing 3.46: Using the where stream for complex matches

```
(where (and (tagged "www") (state "ok") (service #"^apache*")))
```

Here we've matched events that are tagged with `www`, have a state of `ok`, and are from services starting with `apache`.

Up until now we've matched events using "standard" fields like `:service` and `:state`. You'll note we've referred to these fields by their names, minus the `:`. This is some useful syntactic sugar provided by the `where` filtering stream. Any references to the "standard" fields like `:service`, `:host`, `:tags`, `:metric`, `:description`, `:time`, and `:ttl` can use these name shortcuts. If, however, you want to refer to an optional field then you need to refer to it like so:

Listing 3.47: Referring to an optional example field in Riemann

```
(:field_name event)
```

We prefix the field name with a colon and tell Riemann it belongs to `event`, which is Riemann's shorthand for the event being processed. For example, with the field named `type` it would look like:

Listing 3.48: Referring to an optional field in Riemann

```
(:type event)
```

Let's see this in action.

Listing 3.49: The optional type field in Riemann

```
(where (and (tagged "www") (= (:type event) "load")))
```

Here the `where` stream matches all events tagged with `www` and the value of the `:type` field is `load`. You can see that the second `:type` field match does it using a combination of an operator, the name of the field, and a value, constructed like so:

Listing 3.50: Referring to an optional field in Riemann

```
(operator (:field_name event) value)
```

Using this operator-field-value syntax we can also do math operations to match events.

Listing 3.51: Using the where stream with math

```
(where (and (tagged "www") (>= (* metric 10) 5)))
```


Here we've matched all events with the `www` tag and those in which the value of the `:metric` field multiplied by `10` is greater than or equal to `5`. You can use the normal collection of operators: greater than, equal to, less than, and so on in these statements.

We also use a similar syntax to do a range query.

Listing 3.52: Using the where stream for a range query

```
(where (and (tagged "www") (< 5 metric 10)))
```

Here we've matched all events tagged with `www` and those where the `:metric` field has a value between 5 and 10.

 **TIP** You can learn more about event filtering on the [Riemann website](#).

Connecting Riemann servers

Now that we know our individual Riemann servers are working, let's hook them together. In our architecture we have two upstream Riemann servers: `riemanna.example.com` and `riemannb.example.com`. We also have a downstream Riemann server `riemannmc.example.com`. We're assuming you've used the steps above to install Riemann onto each of these hosts.

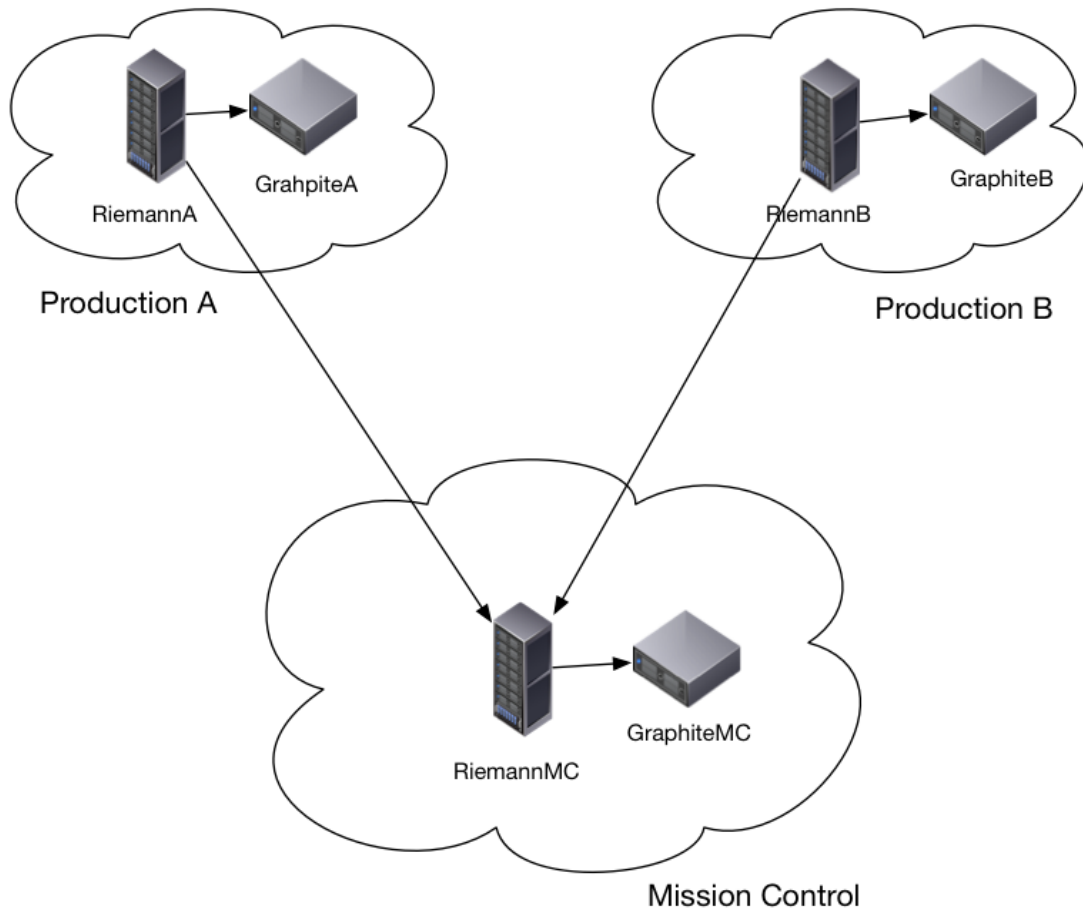


Figure 3.3: Connecting Riemann servers

In our architecture we want to send escalations and status updates for our production environments downstream to our Mission Control environment, riemannmc.example.com. But initially we're just going to send information about Riemann itself. This will allow us to detect if our upstream Riemann servers are operational. We're going to connect the upstream servers to the downstream server and then test our connection.

Configuring the upstream Riemann servers

First, we need to define our downstream Riemann server to our upstream servers. We do this by first specifying a new stream called `async-queue!`. The `async-queue!` stream creates an asynchronous `thread pool` queue. That queue accepts events and passes those events to child streams via that thread pool asynchronously.

So why pass the events asynchronously? Well Riemann is fast but anytime a stream connects to an external service there is a risk that your processing will be blocked waiting for that external service. With an asynchronous queue we tell Riemann to queue events and return to the mainline without blocking. In this case we're then going to use Riemann's own TCP client as one of those asynchronous streams to send events to our downstream server.

⚠ WARNING Asynchronous streams look like an easy fix for connecting to potentially blocking services but they come with some caveats. You should carefully read the [documentation](#) before using them wildly.

Now let's look at an updated configuration file.

Listing 3.53: Updated Riemann configuration

```
(logging/init {:file "/var/log/riemann/riemann.log"})

(require 'riemann.client)

. . .

(let [index (index)
      downstream (batch 100 1/10
                       (async-queue! :agg { :queue-size      1e3
                                           :core-pool-size  4
                                           :max-pool-size  32}
                                     (forward
                                      (riemann.client/tcp-client :host "riemannmc"))))]

  (streams
   (default :ttl 60
            index

            #(info %)

            (where (service #"^riemann.*")
                   downstream))))
```

You can see we've added some new configuration that adds the Riemann client. The `require` function is much like Ruby's `require` method. It loads additional code we might need to do certain actions.

Listing 3.54: Requiring the Riemann client

```
(require 'riemann.client)
```

Here our `require` loads the Riemann TCP client. We're going to use this client to send events downstream.

NOTE We're going to talk about `require` in a lot more detail a little later in this chapter.

Next, we've added another binding to our `let` expression (remember, a binding is a symbol-value pair). This configures the destination and process for sending events.

Listing 3.55: Added downstream binding to Riemann

```
(let [index (index)
      downstream (batch 100 1/10
                       (async-queue! :agg { :queue-size    1e3
                                           :core-pool-size 4
                                           :max-pool-size 32}
                                       (forward
                                        (riemann.client/tcp-client :host "riemannmc"))))]
      . . .
    )
```

Our binding defines a symbol called `downstream`. The value of this symbol is a series of streams. Our first stream is called `batch`. This batches up events to send. Each batch is sent when 100 events or 1/10th of a second has passed. The `batch` stream passed the events into our `async-queue!`, which we've called `:agg` (for aggregation).

We've defined a `queue-size` for our `async-queue!` in exponential notation, `1e3` or 1000. We set `core-pool-size` (the number of threads in the pool) to 4 and `max-pool-size` (the maximum number of threads in the pool) to 32. This should generally work for most scenarios.

NOTE You can read a bit more about Java-based ThreadPooling [here](#). This provides some useful information about the interaction of queue and pool sizing.

Our queue takes the incoming event batches and passes them to another child stream, `forward`. The `forward` stream sends events through a Riemann client—here the TCP client—to our `riemannmc` host.

Listing 3.56: Riemann client forwarding configuration

```
(riemann.client/tcp-client :host "riemannmc")
```

NOTE This assumes you've configured DNS, added the various Riemann servers to `/etc/hosts`, or provided some other way for Riemann to resolve the `riemannmc` hostname.

This is a complex configuration so let's walk through the whole process to make sure it's clear.

1. We define the `downstream` symbol. When that symbol is referenced events are passed into it.
2. Events first go into the `batch` stream. Every 100 events or 1/10th of a second, events are batched and sent on.
3. The batched events are passed to the `async-queue!` stream.
4. The `async-queue!` stream passes the events to the `forward` stream which

sends them to the `riemannmc` server.

Lastly, we've added a `where` stream to select the events we want to send.

Listing 3.57: The `where` filtering stream for forwards

```
(where (service #"^riemann.*")
  downstream)
```

As we discovered earlier, the `where` filtering stream selects events based on specific criteria—for example, from a particular host or service—via a regular expression or from the result of executing a function of some kind.

Here our `where` stream selects any events that match the regular expression `^riemann.*`. This is any events whose `:service` field starts with `riemann..` These events are then passed to the `downstream` symbol which forwards them to the `riemannmc` server.

We then add the configuration we created to both the `riemanna` and `riemannb` servers.


Configuring the downstream Riemann server

Now let's look at the configuration on our downstream `riemannmc` server.

Listing 3.58: The downstream riemannmc server

```
. . .  
  
(let [index (index)]  
  ; Inbound events will be passed to these streams:  
  (streams  
    (default :ttl 60  
      ; Index all events immediately.  
      index  
  
      #(info %))))
```

You can see it's basically the Riemann configuration we've just created except that it lacks our `downstream` sending configuration.

 **NOTE** We've included all example configuration and code in the book [on GitHub](#).

Enabling the send of our Riemann events downstream

We now restart Riemann on all our servers to enable the sending of our events downstream.

Listing 3.59: Restarting Riemann to enable forwarding

```
riemanna$ sudo service riemann restart
riemannb$ sudo service riemann restart
riemannmc$ sudo service riemann restart
```

We should now see some new events for our queue in the `/var/log/riemann/riemann.log` log file on our upstream servers.

Listing 3.60: Riemann agg events on riemanna or riemannb

```
INFO [2015-02-03 15:29:10,911] Thread-7 - riemann.config - #
riemann.codec.Event{:host riemanna, :service riemann executor
agg accepted rate, :state ok, :description nil, :metric 250/2507,
:tags nil, :time 711497675449/500, :ttl 20}
INFO [2015-02-03 15:29:10,911] Thread-7 - riemann.config - #
riemann.codec.Event{:host riemanna, :service riemann executor
agg completed rate, :state ok, :description nil, :metric
250/2507, :tags nil, :time 711497675449/500, :ttl 20}
INFO [2015-02-03 15:29:10,911] Thread-7 - riemann.config - #
riemann.codec.Event{:host riemanna, :service riemann executor
agg rejected rate, :state ok, :description nil, :metric 0N, :
tags nil, :time 711497675449/500, :ttl 20}
. . .
```

These are useful to allow us to track the state, performance, and status of our forwarding.


We'll also start to see events on our downstream server: `riemannmc`.

You'll note we don't have to change anything on the `riemannmc` server. It's already set up to receive events. If we look in the `/var/log/riemann/riemann.log` log file we'll find events from `riemanna` and `riemannb` as well as the local events from `riemannmc`.

Listing 3.61: Combined events from upstream and downstream

```
INFO [2015-02-03 08:35:58,507] Thread-6 - riemann.config - #
riemann.codec.Event{:host riemannMC, :service riemann server ws
0.0.0.0:5556 in latency 0.999, :state ok, :description nil, :
metric nil, :tags nil, :time 1422970558489/1000, :ttl 20}
. . .
INFO [2015-02-03 08:36:01,495] defaultEventExecutorGroup-2-1 -
riemann.config - #riemann.codec.Event{:host riemannb.
lovedthanlost.net, :service riemann streams rate, :state nil, :
description nil, :metric 3.9884721385215447, :tags [riemann], :
time 1422970561, :ttl 20.0}
. . .
INFO [2015-02-03 08:36:14,314] defaultEventExecutorGroup-2-1 -
riemann.config - #riemann.codec.Event{:host riemanna, :service
riemann streams latency 0.5, :state nil, :description nil, :
metric 0.222681, :tags [riemann], :time 1422970574, :ttl 20.0}
. . .
```

Here we see events from all three servers indicating that they are successfully connected.

 **NOTE** We've included this example configuration with all the required files in the book's code [on GitHub](#).

Alerting on the upstream Riemann servers

So now the downstream `riemannmc` Riemann server knows about the upstream `riemanna` and `riemannb` servers. But we also want to know when something goes wrong with those upstream servers. To do that we're going to take advantage of

Riemann's index.

Remember the index is a table of the current state of all services being tracked by Riemann. Each event you tell Riemann to index is added as a service mapped by its `:host` and `:service` fields. The index retains the most recent event for that service. Each indexed event has a Time-to-Live or TTL. The TTL can be set by the event's `:ttl` field, or if no TTL is present then a default can be specified. Our default TTL is 60 seconds set using the `default` function in our Riemann configuration.

If a service fails it stops submitting events to Riemann. In our Riemann configuration we have a `periodically-expire` function that runs every 5 seconds. This is the event reaper for the index. It checks the TTL of events in the index and expires and deletes events from the index if their TTL is expired. When an indexed event is expired a new event is created for the indexed service with a `:state` of `expired` and sent back to the stream.

We then monitor the stream for events with a `:state` of `expired`, which means the service isn't reporting, and notify on those.

Let's create some configuration on the `riemannmc` server to catch expired events from the `riemanna` and `riemannb` servers. This means that if these servers stop sending events, we'll get notified.

Take a look at our `riemannmc` Riemann configuration now. Our `/etc/riemann/riemann.config` file looks like:

Listing 3.62: The downstream riemannmc server configuration

```
(logging/init {:file "/var/log/riemann/riemann.log"})

(let [host "0.0.0.0"]
  (repl-server {:host "127.0.0.1"})
  (tcp-server {:host host})
  (udp-server {:host host})
  (ws-server  {:host host}))

(periodically-expire 10 {:keep-keys [:host :service :tags, :
state, :description, :metric]})

(let [index (index)]
  ; Inbound events will be passed to these streams:
  (streams
   (default :ttl 60
    ; Index all events immediately.
    index

    #(info %))))
```

It's similar to our upstream Riemann servers, minus the configuration that sends events downstream.

We're going to add some configuration to:

1. Identify expired events.
2. Select only the Riemann-specific events.
3. Email a notification on these Riemann-specific expired events.

Let's look at this additional configuration. First we're going to configure a notification mechanism. We're going to start with something simple: email. If we detect an expired Riemann event, we'll send an email notification.

To do this we need to configure the `mailer` plugin. The `mailer` plugin allows us to send email from Riemann. We're going to configure the `mailer` plugin in a names-

pace. Namespaces are a way of organizing code and functions. You can consider namespaces a more advanced method of organizing and including streams into our Riemann configuration.

Riemann uses Clojure's built-in [namespacing](#) to do this. In Clojure it's generally recommended you use a carefully defined namespace that won't overlap with anything else. In most cases this is a format like:

Listing 3.63: Clojure namespace format

```
[organization].[library|app].[group-of-functions]
```

In our case we're going to use [examplecom](#) as our organization. You might use [mycorpname](#) or a department name or something similar. We're then going to call our library [etc](#) because that's broadly its function: useful functions we're going to regularly use in our Riemann configuration. Finally, we're going to call the group of function: [email](#). Literally [examplecom.etc.email](#) or Example.com Etc Email.

Our namespace also shapes where we store our code on our Riemann server. Riemann expects to find the code in a directory structure matching the namespace, underneath the [/etc/riemann](#) directory. So let's start by creating that directory structure.

Listing 3.64: Creating the examplecom.etc namespace path

```
$ sudo mkdir -p /etc/riemann/examplecom/etc
```

Let's then create a file to hold our Riemann code and functions.

Listing 3.65: Creating the email.clj file

```
$ sudo touch /etc/riemann/examplecom/etc/email.clj
```

We can see our directory structure and file follows the namespace:

`examplecom/etc/email`

Our Riemann code and functions will go inside the `email.clj` file.

 **NOTE** `.clj` is the extension for a Clojure code file.

Let's look at the code in this file to get started.

Listing 3.66: Requiring the Riemann functions

```
(ns examplecom.etc.email
  (:require [riemann.email :refer :all]))

(def email (mailer {:from "reimann@example.com"}))
```

Firstly, we declare a name using the `ns` function. The `ns` function creates a new namespace. We've called our namespace: `examplecom.etc.email`. This name is how we'll reference our namespace in our Riemann configuration.

 **TIP** Namespaces are the standard Clojure of organizing code, libraries and functions. You can read more about namespacing in the [Riemann HOWTO](#).

Next we've specified an argument, `:require`, to be passed into our namespace. The `:require` statement is closely related to the `require` function we used earlier to include Riemann's TCP client. It performs the same function inside a namespace. The `:require` argument here includes the `riemann.email` library of functions. The `:require` ensures that the namespace to be required exists, is ready to be used and evaluates the corresponding namespace. By evaluating the namespace, it also defines and makes available any functions in that namespace.

Inside our `examplecom.etc.email` namespace we can now refer to functions inside the `riemann.email` namespace, like so:

```
riemann.email/mailer
```

Referring to functions with the fully-qualified namespace is a little unwieldy though. We've added an argument to our `:require` directive called `:refer`. The `:refer` function means you don't have to use fully qualified names to reference the functions inside a namespace. You can refer one, many or all functions in a namespace. Here we've used the `:all` option to refer all functions inside `riemann.email`.

Alternatively, if you only want to use one or more functions from that namespace you can specify only those.

Listing 3.67: Referring functions

```
(ns examplecom.etc.email
  (:require [riemann.email :refer [mailer]]))
. . .
```

This would only refer the `mailer` function from the `riemann.email` namespace.

We can now reference a function inside our namespace without needing to prefix it with: `riemann.email`.

NOTE Be careful with this. If you define a symbol with the same name inside two namespaces and refer both of them then you will get a conflict and Riemann will fail to start. You cannot have `examplecom.etc.email/foo` and `examplecom.etc.fish/foo` defined and referred.

Now let's look at our `mailer` configuration in this file.

Listing 3.68: Configuring email notifications in Riemann

```
(def email (mailer {:from "reimann@example.com"}))
```

You can see we've used the `def` statement. As we learned in Appendix A, the `def` statement declares a symbol and a `var`.

In our case we're giving the `email` symbol a value of `mailer`. This tells Riemann that anywhere we specify `email` it should call the `mailer` function from the `riemann.email` namespace. As we referred to `:all` functions in this namespace we don't have to prefix `mailer` with `riemann.email`.

The `mailer` function sends email. We've also specified one option for the `mailer` function: `:from`, which controls the source email address for any emails from Riemann.


The `mailer` function uses a standard Clojure email library called `Postal` under the covers to send email. By default it uses the local `sendmail` binary but it can also be configured to use an SMTP server.

TIP A quick way to add the `sendmail` binary and set up local mail sending is to install the `mailutils` package on Ubuntu, or the `mailx` package on Red Hat

and related distributions.

Listing 3.69: Configuring an SMTP server for Postal

```
(def email (mailer {:host "smtp.example.com"
                   :user "james"
                   :pass "password"
                   :from "riemann@example.com"}))
```

 **TIP** We should add this `email.clj` file and its configuration to all our Riemann hosts. It's going to be useful in future chapters to send notifications.

Now we've got a way to notify on our event, let's add that capability to our Riemann configuration. To use our new `email` function we need to tell Riemann about it in our `riemann.config` file. To do this we again use the `require` function but slightly differently this time.

Listing 3.70: Adding our email function to Riemann

```
(logging/init {:file "/var/log/riemann/riemann.log"})

(require 'riemann.client)
(require '[examplecom.etc.email :refer :all])

. . .
```

We've added a new `require` to our `riemann.config` file below our `riemann.client`. It includes the `examplecom.etc.email` namespace and uses the `:refer :all`

directive to refer all functions in that namespace.

On startup, Riemann automatically searches for our namespace underneath the `/etc/riemann` directory, evaluates the namespace and the functions in it. In our case this will cause Riemann to search for a directory:

```
/etc/riemann/examplecom/etc/
```

And then load the file at:

```
/etc/riemann/examplecom/etc/email.clj
```

The `:refer` then allows us to reference the `email` function inside our Riemann configuration, without needing to prefix it with the `examplecom.etc.email` namespace.

We can now specify a stream to grab the relevant events.

Listing 3.71: Our expired Riemann event filter stream

```
(expired
  (where (service #"^riemann.*")
    (email "james@example.com")))
```

Here we've used the `expired` filtering stream. The `expired` stream matches events expired from the index. You can think about it like a `where` stream with the match on the `:state` field of `expired` preconfigured.

We've used a `where` stream to do a regular expression match for any services that start with `riemann.`. Any events matched by the `where` stream will be passed to the `email` var and mailed via the `mailer` function to `james@example.com`.

To activate this new `email` function we now need to reload or restart Riemann. This will load, evaluate and refer the `examplecom.etc.email` namespace.

So if we were to stop Riemann on the `riemanna` host now, after the TTL had expired, then the Riemann index on the `riemannmc` host would generate some

events like this:

Listing 3.72: The riemann expired streams event

```
{:ttl 60, :time 713456271631/500, :state expired, :service  
riemann streams latency 0.999, :host riemann, :tags [riemann]}
```

We see the event has `:state` of `expired` and contains the `:service`, `:host`, and `:tags` fields, all of which were copied to the new expired event when the event reaper ran over the index. The event also contains the default TTL of 60 seconds, and the time the event was generated.

This matched event would then be passed to the `email` var, which would trigger an email to be sent. That email will look something like this:

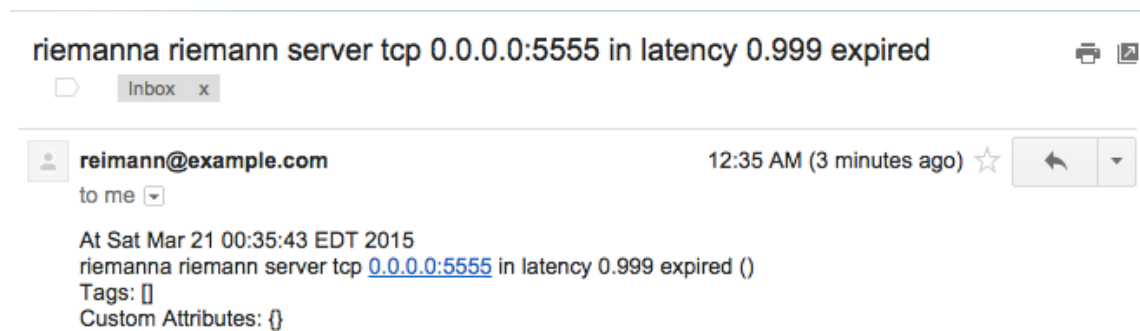


Figure 3.4: Email notification

It's great we're going to be getting emails when the Riemann server fails... But there's a problem. We'll get an email notification for every Riemann metric that is being collected—yes, one for each event. This could mean a lot of emails letting us know that one of our Riemann servers is down.

Throttling Riemann events

To prevent this flood of emails we're going to add a throttle to our notifications. Let's add some throttling to our configuration.

Listing 3.73: Our throttled expired Riemann event filter

```
(expired
  (where (service #"^riemann.*")
    (throttle 1 600
      (email "james@example.com"))))
```

You'll see we've added a new stream called `throttle`. The `throttle` allows a number of events through and then ignores any others for a period of time.

Listing 3.74: The throttle stream

```
(throttle 1 600
```

This throttle works by allowing one event through and then dropping and ignoring all other events for 600 seconds or 10 minutes. It's a pretty crude mechanism but works for services where we only care about a limited number of notifications.

Rolling up Riemann events

An alternative to `throttle` is the `rollup` stream. The `rollup` stream will allow a few events to pass through, then it will collect and hold the others. It holds those events for a period of time and then sends a summary of them.

Listing 3.75: A rollup of our expired Riemann events

```
(expired
  (where (service #"^riemann.*")
    (rollup 5 3600
      (email "james@example.com"))))
```

The `rollup` stream here will only send five emails per 3,600 seconds (one hour). You'll get four emails immediately and then, after an hour, you'll get a final email with a summary of any other events received during that hour period.

⚠ WARNING The `rollup` stream accumulates events in memory. The more events there are, the more memory is consumed. You should be careful to ensure it doesn't exhaust memory on the host holding the rolled-up events.

Alternatives to email notifications

Obviously email is not always an ideal notification mechanism—we all already have folders full of emails—so Riemann provides a wide variety of other mechanisms including [PagerDuty](#) and a variety of chat applications like [Slack](#).

We'll talk a lot more about these mechanisms and about notifications in Chapter 10.

■ NOTE We've included the `riemannmc` configuration in the book's code [here](#).

Testing your Riemann configuration

One of the advantages of Riemann's configuration being a Clojure program is that you can write tests to confirm that your configuration is working correctly.

To test Riemann events, we tap into points where we'd like to observe events and then write tests that confirm the right behavior is occurring and the right events are being seen or generated. In production Riemann will ignore these taps so you don't experience a performance hit from adding them.

Let's look at an example. In our current `riemannmc` configuration we accept incoming events and index them immediately.

Listing 3.76: Revisting our `riemannmc` configuration

```
(logging/init {:file "/var/log/riemann/riemann.log"})

(let [host "0.0.0.0"]
  (repl-server {:host "127.0.0.1"})
  (tcp-server {:host host})
  (udp-server {:host host})
  (ws-server  {:host host}))

(periodically-expire 10 {:keep-keys [:host :service :tags, :
state, :description, :metric]})

(let [index (index)]

  (streams
   (default :ttl 60
    ; Index all events immediately.
    index)))
```

So our test is going to confirm that events coming into the stream are being indexed. To support this test we need to wrap our `index` stream in a `riemann.test /tap` function. This will allow us to watch the events being indexed.

Listing 3.77: Adding a tap to our riemannmc index

```
(logging/init {:file "/var/log/riemann/riemann.log"})

(let [host "0.0.0.0"]
  (repl-server {:host "127.0.0.1"})
  (tcp-server {:host host})
  (udp-server {:host host})
  (ws-server  {:host host}))

(periodically-expire 10 {:keep-keys [:host :service :tags, :
state, :description, :metric]})

(let [index (tap :index (index))]

  (streams
   (default :ttl 60
    ; Index all events immediately.
    index)))
```

You see that we've added a tap around our `index`. We've called that tap `:index`. We then create one or more tests that sample incoming events to the `index`. We do this by defining a `tests` stream and using the standard Clojure test function: `deftest`.

Listing 3.78: Adding tests to our riemannmc configuration

```
(logging/init {:file "/var/log/riemann/riemann.log"})

(let [host "0.0.0.0"]
  (repl-server {:host "127.0.0.1"})
  (tcp-server {:host host})
  (udp-server {:host host})
  (ws-server  {:host host}))

(periodically-expire 10 {:keep-keys [:host :service :tags, :
state, :description, :metric]})

(let [index (tap :index (index))]

  (streams
   (default :ttl 60
    ; Index all events immediately.
    index)))

(tests
 (deftest index-test
  (is (= (inject! [{:service "test"
                   :time    1}])
         {:index [{:service "test"
                   :time    1
                   :ttl     60}]})))))
```

We see a new set of `tests` streams in our configuration now. We've defined one test, `index-test`. The test is a simple process:

- Inject an event into the stream.
- Monitor the `:index` tap and see if that event arrives.

The `inject!` function injects an event into the stream. Here we're injecting a single event with a `:service` field set to `test` and a `:time` field of `1`. We're then

asking if the `:index` tap sees a like event. We've added the `:ttl` field of `60` because the `default` function sets that when we index an event. Without it our test will fail because the event we're watching for will not match the event we've injected.

We then run our tests using the `riemann` binary. Riemann must be stopped when we run the tests because our whole configuration, including components like interface bindings, is run when the tests are run.


Listing 3.79: Running the Riemann tests

```
$ sudo riemann test riemann.config
INFO [2015-07-15 17:40:01,236] main - riemann.repl - REPL server
{:port 5557, :host 127.0.0.1} online

Testing riemann.config-test

Ran 1 tests containing 1 assertions.
0 failures, 0 errors.
```

We see that our single test, containing a single assertion, has been run and has passed successfully. This means we've injected an event into the stream, watched the tap, and then seen the corresponding event appear in the tap.

 **TIP** You can read more about the testing of your configuration in the [Riemann documentation](#).

Validating Riemann configuration

Riemann configuration looks pretty scary at first. But to make it easier to build your Riemann configuration there is a syntax checker available. To use it you need to download and build it. This assumes you have Java, Git, and Leiningen installed.

Listing 3.80: Download the Riemann syntax checker

```
$ git clone https://github.com/samn/riemann-syntax-check.git
```

Then build the syntax checker as a Jar file.

Listing 3.81: Build the Riemann syntax checker

```
$ cd riemann-syntax-check
$ lein uberjar
. . .
```

You should see some dependencies downloaded and some Jar files created. You can then use one of these Jar files to syntax check a Riemann configuration.

Listing 3.82: Syntax check a Riemann configuration

```
$ cd /etc/riemann/
$ java -jar riemann-syntax-check-0.2.0-standalone.jar riemann.
config
```

Here we're syntax checking the `riemann.config` configuration file. Any errors or issues will be reported so you can fix them before reloading or restarting Riemann.

Performance, scaling, and making Riemann highly available

The performance of Riemann is largely memory-bound. The only disk IO it uses is logging. You should ensure any hosts running Riemann have generous allocations of memory.

You can configure how much additional memory goes to the Riemann process by configuring the Java heap size options that are passed when Riemann is launched. To do this we add any required options to the `/etc/default/riemann` file on Ubuntu and `/etc/sysconfig/riemann` file on Red Hat and similar distributions. In both files, uncomment the line starting with `EXTRA_JAVA_OPTS` and add the `-Xms` and `-Xmx` flags with an appropriate amount of additional memory. These flags control the initial and maximum Java heap size. Like so:

Listing 3.83: Configuring additional RAM

```
EXTRA_JAVA_OPTS="-Xms4096m -Xmx4096m"
```

Oracle recommends that you set the initial and maximum heap size to the same value to minimize garbage collections.

NOTE You can read a bit more about the available `-X` options in the [Java documentation](#).

You can also view the JVM tuning options in the [Debian and RPM packages](#) in the `AGGRESSIVE_OPTS` environment variable for other ideas on how to tune Riemann.

At this time there isn't an out-of-the-box solution for high availability with Rie-

mann. Riemann servers don't come with a highly available, fail-over solution or configuration. This isn't all that dire though. Riemann doesn't maintain much state—it's mostly being used as a routing engine for events and metrics. Losing that capability isn't great, but because of the statelessness of Riemann it's easy to rebuild or add a new Riemann host to replace any unavailable hosts.

Alternatively you could run one or more Riemann hosts in a warm standby and replace missing hosts with new ones relatively quickly. You could also supplement this with a shared proxy like HAProxy and add multiple Riemann servers to your back-end host pool. Then if a failure occurred on one host another would be available.

For scaling it's also not yet possible to automatically scale and distribute Riemann workloads in any sort of cluster or distributed manner. For scaling Riemann it is largely a matter of horizontal scaling by adding Riemann servers, perhaps distributed by application, rack, stack, data center, or site. Again using a solution like a shared proxy to distribute events and metrics to a series of back-end Riemann hosts depending on their source is also an option.

Or you could also create a sharded configuration. This configuration would run two or more Riemann servers in parallel. Every event would be assigned a shard ID from the Riemann server that received it. They would then be consolidated downstream, filtering out the shard ID and using a merge algorithm for each set of events.

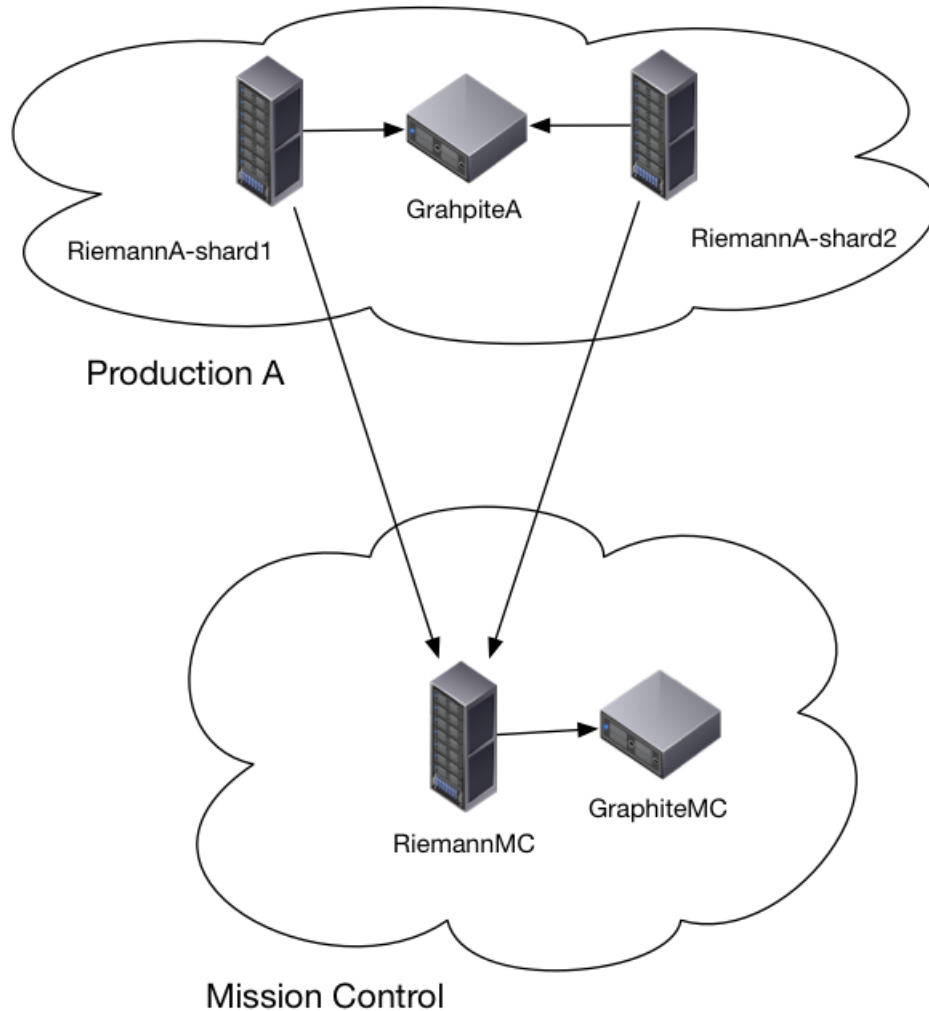


Figure 3.5: Sharded Riemann

This book does not currently cover either high availability or scaling in any depth, but will be updated if better native solutions become available.

💡 TIP There's also [a plugin to enable JMX-based monitoring](#) of Riemann. You can see read more about using JMX to monitor the JVM in Chapters 8 and 12.

Alternatives to Riemann

There are a few event-based or message queue systems that work like Riemann. This is not a definitive list but it's a sampling of the more interesting tools you could use if the choices in the book aren't suitable or to your taste.

- [Apache Samza](#) and [Apache Kafka](#) — A distributed real-time computation system and a publish-subscribe messaging system with a cluster-centric design.
- [Prometheus](#) — Prometheus is an open-source systems monitoring and alerting toolkit originally built at SoundCloud.
- [Heron](#) - Heron is realtime analytics platform developed by Twitter.
- [Bosun](#) — An open-source, MIT-licensed, monitoring and alerting system built by Stack Exchange.
- [Anthracite](#) — An event and change logging and management application.
- The [ELK Stack](#) — Elasticsearch, Logstash, Kibana. A powerful logging tool that can also be adapted to handle events and metrics. We'll look at it more closely in Chapter 8.
- [Heka](#) — Another logging tool, this one released by the Mozilla team. Sadly, [no longer maintained](#). There is a potential successor called [Hindsight](#).
- [Godot](#) — An event streamer modeled on Riemann but rewritten in node.js. Instead of Clojure, it's written in Javascript. It's somewhat slower than Riemann but shares many concepts.
- [Ganglia](#) — A monitoring tool with a focus on clusters and grids.
- [Munin](#) — A popular metric and monitoring tool that uses [RRDTool](#).
- [Snap](#) - Open source by Intel, Snap is a telemetry collection and processing tool. It also supports publishing data to Riemann.

Summary

In this chapter we've learned how to install Riemann. We installed Riemann on three hosts: `riemanna` and `riemannb` in our production data centers, and `riemannmc` in our Mission Control environment. We've also had an introduction to configuring and running it.

We've also connected our upstream Riemann production servers with the downstream Riemann server in Mission Control. We then set up notifications in the downstream Riemann server to detect if there are any issues with the upstream servers.

These Riemann servers will form the basis of our monitoring framework: event-centric routing engines that will allow us to collect, process, and send events and metrics. We'll track the state of our hosts, services, and applications centrally in the Riemann index, notify on any critical or important events (or the absence of those events), and then graph any related metrics.

In the next chapter we're going to continue our monitoring build by installing and configuring the graphing engine Graphite in our environment. We'll then send some initial metrics from Riemann to Graphite.

Chapter A

An Introduction to Clojure and Functional Programming

Riemann is configured using a Clojure-based configuration file. This means your configuration file is actually processed as a Clojure program. So, to process events and send notifications and metrics, you'll be writing Clojure. Don't panic! You won't need to become a full-fledged Clojure developer to use Riemann. We'll teach you what you need to know in order to use Riemann. Additionally, Riemann comes with a lot of helpers and shortcuts that make it easier to write Clojure to do what we need to process our events.

Let's learn a bit more about Clojure and help you get started with Riemann. Clojure is a dynamic programming language that targets the Java Virtual Machine. It's a dialect of Lisp and is largely a [functional programming language](#).

[Functional programming](#) is a programming style that focuses on the evaluation of mathematical functions and steers away from changing state and mutable data. It's highly declarative, meaning you build programs from expressions that describe "what" a program should accomplish rather than "how" it accomplishes things.

NOTE Languages that describe more of the “how” are called imperative languages.

Examples of declarative programming languages include SQL, CSS, regular expressions, and configuration management languages like Puppet and Chef. Let’s look at a simple example.

Listing A.1: A declarative statement

```
SELECT user_id FROM users WHERE user_name = "Alice"
```

In this SQL query we’re asking for the `user_id` for `user_name` of `Alice` from the `users` table. The statement is asking a declarative “what” question. We don’t really care about the “how”; the database engine takes care of those details.

In addition to their declarative nature, functional programming languages try to eliminate all side effects from changing state. In a functional language, when you call a function its output value depends only on the inputs to the function. So if you repeatedly call function `f` with the same value for argument `x`, `f(x)`, it will produce the same result every time. This makes functional programs easy to understand, test, and predict. Functional programming languages call functions that operate like this “pure” functions.

The best way to get started with Clojure is to understand the basics of its syntax and types. Let’s get an introduction now.

WARNING This is going to be a high-level introduction to Clojure. It’s designed to give you the knowledge and recognition of various syntax and expressions to allow you to work with Riemann. We will not be teaching you how to

develop in Clojure in this book.

A brief introduction to Clojure

Let's step through the Clojure basic syntax and types. We'll also show you a tool called REPL that can help you test and build your Clojure snippets. REPL (short for read-eval-print loop) is an interactive programming shell that takes single expressions, evaluates them, and returns the results. It's a great way to get to know Clojure.

NOTE If you're from the Ruby world then REPL is just like `irb`. Or, in Python, it's like when you launch the `python` binary interactively.

We install REPL via a tool called [Leiningen](#). Leiningen is an automation tool for Clojure that helps you automate the build and management of Clojure projects.

Installing Leiningen


In order to install Leiningen we'll need to have Java installed on the host. The prerequisite Java packages we installed on Ubuntu and Red Hat for Reimann will be sufficient for Leiningen too.

We're going to download a Leiningen binary called `lein` to install it. Let's download that into a `bin` directory under our home directory.

Listing A.2: Getting lein

```
$ mkdir -p ~/bin
$ cd ~/bin
$ curl -o lein https://raw.githubusercontent.com/technomancy/leiningen/stable/bin/lein
$ chmod a+x lein
$ export PATH=$PATH:$HOME/bin
```

Here we've created a new directory called `~/bin` and changed into it. We've then used the `curl` command to download the `lein` binary and the `chmod` command to make it executable. Lastly, we've added our `~/bin` directory to our path so that we can find the `lein` binary.

 **TIP** The addition of the `~/bin` directory assumes you're in a Bash shell. It's also temporary to your current shell. You'd need to add the path to your `.bashrc` or a similar setup for your shell.

Next we need to run `lein` to auto-install its supporting libraries.

Listing A.3: Auto-installing lein

```
$ lein
. . .
```

This will download Leiningen's supporting Jar file.

Finally, we run REPL using the `lein repl` sub-command.

Listing A.4: Launching REPL

```
$ lein repl
. . .
user=>
```

This will download Clojure itself (in the form of its Jar file) and launch our interactive Clojure shell.

Clojure syntax and types

Let's use this interactive shell to look at some of the syntax and functions we've just learned about. We'll start by opening our shell.

Listing A.5: The REPL shell

```
user=>
```

Now we try a simple expression.

Listing A.6: Our first Clojure value

```
user=> nil
nil
```

The `nil` expression is the simplest value in Clojure. It represents literally nothing. We can also specify an integer value.

Listing A.7: Our first Clojure integer

```
user=> 1  
1
```

Or a string.

Listing A.8: Our first Clojure string

```
user=> "hello Ms Event"  
"hello Ms Event"
```

Or Boolean values.

Listing A.9: Our first Clojure Booleans


```
user=> true  
true  
user=> false  
false
```

Clojure functions

While interesting, these values aren't exciting on their own. To do some more interesting things we can use Clojure functions. A function is structured like this:

Listing A.10: The Clojure function syntax

```
(function argument argument)
```

 **TIP** If you're used to the Ruby or Python world, a function is broadly the equivalent of a method.

Let's look at a function in action by doing something with some values: adding two integers together.

Listing A.11: Our first Clojure function

```
user=> (+ 1 1)
2
```

In this case we've used the `+` function and added `1` and `1` together to get `2`.

But there's something about this structure that might look familiar to you if you've used other programming languages. Our function looks just like [a list](#). This is because it is! Our expression might add two numbers together, but it's also a list of three items in a valid list data structure.

 **NOTE** Technically it's an [s-expression](#).

This is a feature of Clojure called [homoiconicity](#), sometimes described as: “code

is data, data is code.” This concept is inherited from Clojure’s parent language, [Lisp](#).

Homoiconicity means that the program’s structure is similar to its syntax. In this case Clojure programs are written in the form of lists. Hence you can gain insight into the program’s internal workings by reading its code. This also makes [metaprogramming](#) really easy because Clojure’s source code is a data structure and the language can treat it like one.

Now let’s look more closely at the `+` function. Each function is a symbol. A symbol is a bare string of characters, like `+` or `inc`. Symbols have short names and full names. The short name is used to refer to it locally—for example, `+`. The full name, or perhaps more accurately the fully qualified name, gives you a way to refer to the symbol unambiguously from anywhere. The fully qualified name of the `+` symbol is `clojure.core/+`. The `clojure.core` is the fundamental library of the Clojure language. We refer to `+` in its fully qualified form here:

Listing A.12: The fully qualified `+` function

```
user=> (clojure.core/+ 1 1)
2
```

Symbols refer to other things; generally they point to values. Think about them as a name or identifier that points to a concept: `+` is the name, “adding” is the concept. When Clojure encounters a symbol it evaluates it by looking up its meaning. If it can’t find a meaning it’ll generate an error message, for example:

Listing A.13: Unable to resolve symbol

```
user=> (bob 1 2)
CompilerException java.lang.RuntimeException: Unable to resolve
symbol: bob in this context, compiling:(NO_SOURCE_PATH:1:1)
```

Clojure also has a syntax for stopping that evaluation. This is called quoting, and it is achieved by prefixing the expression with a quotation mark: `'`.

Listing A.14: Quoting a symbol

```
user=> '(+ 1 1)
(+ 1 1)
```

This returns the symbol itself without evaluating it. This is important because often we want to do things, review things, or test things without evaluating.

For example, if we need to determine what type of thing something is in Clojure we use the `type` function and quote the function like so:

Listing A.15: The type function

```
user=> (type '+)
clojure.lang.Symbol
```

Here we see that `+` is a Clojure language symbol.

Lists

Clojure also has a variety of data structures. Especially useful to us will be collections. Collections are groups of values, for example, a list or a map.

Let's start by looking at lists. Lists are core to all Lisp-based languages (Lisp means "LISt Processing"). As we discovered above, Clojure programs are essentially lists. So we're going to see a lot of them!

Lists have zero or more elements and are wrapped in parentheses.

Listing A.16: A Clojure list

```
user=> '(a b c)
(a b c)
```

Here we've created a list containing the elements `a`, `b`, and `c`. We've quoted it because we don't want it evaluated. If we didn't quote it then evaluation would fail because none of the elements, `a`, `b`, etc., are defined. Let's see that now.

Listing A.17: An unquoted Clojure list

```
user=> (a b c)
CompilerException java.lang.RuntimeException: Unable to resolve
symbol: a in this context, compiling:(NO_SOURCE_PATH:1:1)
```

We can do a few neat things with lists, such as adding an element using the `conj` function.

Listing A.18: Adding an element to a list

```
user=> (conj '(a b c) 'd)
(d a b c)
```

You can see we've added a new element, **d**, to the front of the list. Why the front? Because a list is really a **linked list** and focuses on providing immediate access to the first value in the list. Lists are most useful for small collections of elements and when you need to read elements in a linear fashion.


We can also return values from a list using a variety of functions.

Listing A.19: Working with lists

```
user=> (first '(a b c))
a
user=> (second '(a b c))
b
user=> (nth '(a b c) 2)
c
```

Here we've pulled out the first element, second element, and using the **nth** function, the third element.

This last, **nth**, function shows us a multi-argument function. The first argument is the list, **'(a b c)**, and the second argument is the index value of the element we want to return, here **2**.

 **TIP** Like most programming languages Clojure starts counting from **0**.

We can also create a list with the `list` function.

Listing A.20: Creating a list

```
user=> (list 1 2 3)
(1 2 3)
```

Vectors

Another collection available to us is the vector. Vectors are like lists but they are optimized for random access to the elements by index. Vectors are created by adding zero or more elements inside square brackets.

Listing A.21: A Clojure vector

```
user=> '[a b c]
[a b c]
```

Like lists, we again use `conj` to add to a vector.

Listing A.22: Adding an element to a vector

```
user=> (conj '[a b c] 'd)
[a b c d]
```

You'll note the `d` element is added at the end because a vector isn't focused on sequential access like a list.

There are some other useful functions we can use on lists and vectors. For example, to get the last element in a list or vector:

Listing A.23: Getting the last element in a vector

```
user=> (last '[a b c d])
d
```

Or count the elements:

Listing A.24: Counting elements in a vector

```
user=> (count '[a b c d])
4
```

Because vectors are designed to look up elements by index, we can also use them directly as functions, for example:

Listing A.25: Using a vector as a function

```
user=> ([1 2 3] 1)
2
```

Here we've retrieved the value, `2`, at index `1`.

We can create a vector with the `vector` function or change another data structure into a vector with the `vec` function.

Listing A.26: Creating or converting vectors

```
user=> (vector 1 2 3)
[1 2 3]
user=> (vec (list 1 2 3))
[1 2 3]
```

Sets

There's a final collection related to lists and vectors called a set. Sets are unordered collections of values, prefixed with `#` and wrapped in curly braces, `{ }`. They are most useful for collections of values where you want to check if a value or values are present.

Listing A.27: A Clojure set

```
user=> '#{a b c}
#{a c b}
```

You'll notice the set was returned in a different order. This is because sets are focused on presence lookups so order doesn't matter quite so much.

Like lists and vectors we use the `conj` function to add an element to a set.

Listing A.28: Adding to a set

```
user=> (conj '#{a b c} 'd)
#{a c b d}
```

Sets can never contain an element more than once, so adding an element that's already present does nothing. You can remove elements with the `disj` function.

Listing A.29: Removing an element from a set

```
user=> (disj '#{a b c d} 'd)
#{a c b}
```

The most common operation with a set is to check for the presence of a specific value. For this we use the `contains?` function.

Listing A.30: Checking for a value inside a set

```
user=> (contains? '#{a b c} 'c)
true
user=> (contains? '#{a b c} 'd)
false
```

Like a vector, you can also use the set itself as a function. This returns the value if it is present, or `nil` if it is not.

Listing A.31: Using the set as a function

```
user=> ('#{a b c} 'c)
c
user=> ('#{a b c} 'd)
nil
```

You can make a set out of any other collection with the `set` function.

Listing A.32: Making a set

```
user=> (set '[a b c])
#{a c b}
```

Here we've made a set out of a vector.

Maps

The last data structure we're going to look at is the map. Maps are key/value pairs enclosed in braces. You can think about them as being equivalent to a hash.

Listing A.33: A Clojure map

```
user=> {:a 1 :b 2}
{:a 1, :b 2}
```

Here we've defined a map with two key/value pairs: `:a 1` and `:b 2`.

You'll note each key is prefixed with a `:`. This denotes another type of Clojure syntax: the keyword. A keyword is much like a symbol, but instead of referencing another value it is merely a name or label. It's highly useful in data structures like maps to do lookups—you look up the keyword and return the value.

We can use the `get` function to retrieve a value.

Listing A.34: Getting a Clojure map value

```
(get {:a 1 :b 2} :a)
1
```

Here we've specified the keyword `:a` and asked Clojure if it is inside our map. It's returned the value in the key/value pair, `1`.

If the key doesn't exist in the map then Clojure returns `nil`.

Listing A.35: Getting a missing Clojure map value

```
user=> (get {:a 1 :b 2} :c)
nil
```

The `get` function can also take a default value to return instead of `nil` if the key doesn't exist in that map.

Listing A.36: Getting a default value from a map

```
user=> (get {:a 1 :b 2} :c :novalue)
:novalue
```

We can use the map itself as a function, as well.

Listing A.37: Using a map as a function

```
user=> ({:a 1 :b 2} :a)
1
```

And we can use keywords as functions to look themselves up in a map.

Listing A.38: Using a keyword as a function

```
user=> (:a {:a 1 :b 2})  
1
```

To add a key/value pair to a map we use the `assoc` function.

Listing A.39: Using `assoc` to add a key/value

```
user=> (assoc {:a 1 :b 2} :c 3)  
{:a 1, :b 2, :c 3}
```

If a key isn't present then `assoc` adds it. If the key is present then `assoc` replaces the value.

Listing A.40: Replacing a key/value with `assoc`

```
user=> (assoc {:a 1 :b 2} :b 3)  
{:a 1, :b 3}
```

To remove a key we use the `dissoc` function.

Listing A.41: Removing a key/value with `dissoc`

```
user=> (dissoc {:a 1 :b 2} :b)  
{:a 1}
```

NOTE If you've come from the Ruby or Python world, the terms list, set,

vector, and map might be a little new. But the syntax probably looks familiar. You can think about lists, vectors, and sets as being similar to arrays, and maps being hashes.

Strings

We can also work with strings. Clojure lets you turn pretty much any value into a string using the `str` function.

Listing A.42: The `str` function

```
user=> (str "holiday")
"holiday"
```

The `str` function turns anything specified into a string. We can also use it to concatenate strings.

Listing A.43: Concatenating a string

```
user=> (str "james needs " 2 " holidays")
"james needs 2 holidays"
```

Creating our own functions

Up until now we've run functions as stand-alone expressions. For example, here's the `inc` function that increments arguments passed to it:

Listing A.44: The inc function again

```
user=> (inc 1)
2
```

This isn't overly practical except to demonstrate how a function works. If we want to do more with Clojure we need to be able to define our own functions. To do this, Clojure provides a function called `fn`. Let's construct our first function.

Listing A.45: The fn function

```
user=> (fn [a] (+ a 1))
```

So what's going on here? We've used the `fn` function to create a new function. The `fn` function takes a vector as an argument. This vector contains any arguments being passed to our function. Then we specify the actual action our function is going to perform. In our case we're mimicking the behavior of the `inc` function. The function will take the value of `a` and add `1` to it.

If we run this code now nothing will happen because `a` is currently unbound—we haven't defined a value for it. Let's run our function now.

Listing A.46: Running our first fn function

```
user=> ((fn [a] (+ a 1)) 2)
3
```

Here we've evaluated our function and passed in an argument of `2`. This is assigned to our `a` symbol and passed to the function. The function adds `a`, now set to `2`, and `1` and returns the resulting value: `3`.

There's also a shorthand for writing functions that we'll see occasionally in the book.

Listing A.47: The fn function shortcut

```
user=> #(+ % 1)
```

This shorthand function is the equivalent of `(fn [x] (+ x 1))` and we can call it to see the result.

Listing A.48: Calling the fn function shortcut

```
user=> ((+ % 1) 2)
3
```

Creating variables

But we're still a step from a named function, and we're missing an important piece: how do we define our own variables to hold values? Clojure has a function called `def` that allows us to do this.

Listing A.49: Creating a var

```
user=> (def smoker "joker")
#'user/smoker
```

The `def` statement does two things:

- It creates a new type of object called a var. Vars, like symbols, are references

to other values. You can see our new var `#'user/smoker` returned as output of the `def` function.

- It binds a symbol to that var. Here the symbol `smoker` is bound to a var with a value of the string `joker`.

When we evaluate a symbol pointing to a var it is replaced by the var's value. But because `def` also creates a symbol, we can refer to our var like that too.

Listing A.50: Evaluating a symbol

```
user=> user/smoker
"joker"
user=> smoker
"joker"
```

Where did this `user/` come from? It's a Clojure namespace. Namespaces are a way Clojure organizes code and program structure. In this case the REPL creates a namespace called `user/` by default. Remember, we learned earlier that a symbol has both a short name—for example, `smoker`—that can be used locally to refer to it, and a full name. That full name, here `user/smoker`, would be used to refer to this symbol from another namespace.

We'll talk more about namespaces and use them to organize our Riemann configuration in other parts of this book. If you'd like to read more about them, there is an excellent explanation [in this post](#).

We can also use the `type` function to see the type of value the symbol references.

Listing A.51: Using the type function on the symbol

```
user=> (type smoker)
java.lang.String
```

Here we see that the value `smoker` resolves to is a string.

Creating named functions

Now, with the combination of `def` and `fn`, we can create our own named functions.

Listing A.52: Creating our first named function

```
user=> (def grow (fn [number] (* number 2)))  
#'user/grow
```

Firstly, we've defined a var (and symbol) called `grow`. Inside that we've defined a function. Our function takes a single argument, `number`, and passes that number to the `*` function, the mathematical multiplication operator in Clojure, and multiplies it by `2`.

Let's call our function now.

Listing A.53: Calling our grow function

```
user=> (grow 10)  
20
```

Here we've called the `grow` function and passed it a value of `10`. The `grow` function multiplies that value and returns the result: `20`. Pretty awesome eh?

But the syntax is a little cumbersome. Thankfully Clojure offers a shortcut to creating a var and binding it to a function called `defn`. Let's rewrite our function using this form.

Listing A.54: Using the defn form

```
user=> (defn grow [number] (* number 2))
#'user/grow
```

That's a little neater and easier to read. Now how about we add a second argument? Let's make both the number to be multiplied and the multiplier arguments.

Listing A.55: Adding a second argument

```
user=> (defn grow [number multiple] (* number multiple))
#'user/grow
```

Let's call our `grow` function again.

Listing A.56: Calling our grow function again

```
user=> (grow 10)
ArityException Wrong number of args (1) passed to: user/grow
clojure.lang.AFn.throwArity (AFn.java:429)
```

Oops, not enough arguments! Let's add the second argument.

Listing A.57: Calling grow with our second argument

```
user=> (grow 10 4)
40
```

We can also add a doc string to our function to help us articulate what it does.

Listing A.58: Adding a second argument

```
(defn grow
  "Multiplies numbers - can specify the number and multiplier"
  [number multiple]
  (* number multiple)
)
```

We can access a function's doc string using the `doc` function.

Listing A.59: Using the doc function

```
user=> (doc grow)
-----
user/grow
([number multiple]
  Multiplies numbers - can specify the number and multiplier
nil)
```

The `doc` function tells us the full name of the function, the arguments it accepts, and returns the docstring.

That's the end of our introduction. We'll see and learn more Clojure elsewhere in the book.


Learning more Clojure

We recommend trying to get an understanding of the basics of Clojure to get the most out of Riemann. If you'd like to start to learn a bit about Clojure, Kyle Kingsbury's excellent [Clojure from the ground up](#) series is a great place to start. This section is much an abbreviated summary of that tutorial, and we can't thank Kyle

enough for writing it. A reading of his tutorial will greatly add to the knowledge we've shared here. For the purposes of this book, we recommend at least a solid reading of the first three posts in the series:

- The [Welcome](#) post.
- The post on [Basic types](#).
- The post on [Functions](#).

Also useful to writing good code is the Clojure [Style Guide](#).

 **TIP** If you're interested in learning a bit more about the basics of Clojure, another good resource is [Learn Clojure](#).

List of Figures

- 3.1 Event Routing 1
- 3.2 Metrics Architecture 4
- 3.3 Connecting Riemann servers 36
- 3.4 Email notification 53
- 3.5 Sharded Riemann 63

Listings

3.1 Installing Java on Ubuntu	5
3.2 Checking Java is installed on Ubuntu	6
3.3 Fetching the Riemann DEB package	6
3.4 Installing the Riemann package on Ubuntu	6
3.5 Installing Java and prerequisites on RHEL	7
3.6 Checking Java is installed on Red Hat	7
3.7 Fetching the Riemann RPM package	8
3.8 Installing the Riemann package on RHEL	8
3.9 Starting and stopping Riemann	9
3.10 Running Riemann interactively	10
3.11 Installing supporting tools prerequisites on Ubuntu	11
3.12 Installing supporting tools prerequisites on RHEL	11
3.13 Installing Riemann's supporting tools	12
3.14 New /etc/riemann/riemann.config configuration file	14
3.15 Riemann logging stanza	14
3.16 The let form	16
3.17 Exposing Riemann on all interfaces	17
3.18 Changing the Riemann port	17
3.19 Connecting to the Riemann REPL server	18
3.20 SIGHUP from the Riemann REPL server	18
3.21 Restarting Riemann	18
3.22 Example Riemann event	19

3.23 Child streams example	21
3.24 Example Apache Riemann event	22
3.25 Example expired Apache Riemann event	22
3.26 More of our default riemann.config configuration file	23
3.27 Copying more keys into expired events	23
3.28 Using the Riemann default function	24
3.29 Logging to the Riemann log file	24
3.30 A Riemann log event	25
3.31 Adding a prefix to Riemann logs entries	25
3.32 Limiting Riemann log entries	25
3.33 A filtered Riemann log event	26
3.34 The Riemann prn function	26
3.35 Reloading Riemann to enable our new configuration	26
3.36 Riemann internal events	27
3.37 The riemann-health command	28
3.38 The riemann-health -host option	28
3.39 Our incoming Riemann data	29
3.40 A Riemann-health disk event	29
3.41 Our first monitoring check	30
3.42 Our prefixed warning event	31
3.43 Using the where stream with a regular expression	32
3.44 Using the where stream with booleans	32
3.45 The tagged-any stream	32
3.46 Using the where stream for complex matches	33
3.47 Referring to an optional example field in Riemann	33
3.48 Referring to an optional field in Riemann	33
3.49 The optional type field in Riemann	34
3.50 Referring to an optional field in Riemann	34
3.51 Using the where stream with math	34
3.52 Using the where stream for a range query	35
3.53 Updated Riemann configuration	38

3.54	Requiring the Riemann client	38
3.55	Added downstream binding to Riemann	39
3.56	Riemann client forwarding configuration	40
3.57	The where filtering stream for forwards	41
3.58	The downstream riemannmc server	42
3.59	Restarting Riemann to enable forwarding	43
3.60	Riemann agg events on riemannb	43
3.61	Combined events from upstream and downstream	44
3.62	The downstream riemannmc server configuration	46
3.63	Clojure namespace format	47
3.64	Creating the examplecom.etc namespace path	47
3.65	Creating the email.clj file	48
3.66	Requiring the Riemann functions	48
3.67	Referring functions	49
3.68	Configuring email notifications in Riemann	50
3.69	Configuring an SMTP server for Postal	51
3.70	Adding our email function to Riemann	51
3.71	Our expired Riemann event filter stream	52
3.72	The riemannb expired streams event	53
3.73	Our throttled expired Riemann event filter	54
3.74	The throttle stream	54
3.75	A rollup of our expired Riemann events	55
3.76	Revisiting our riemannmc configuration	56
3.77	Adding a tap to our riemannmc index	57
3.78	Adding tests to our riemannmc configuration	58
3.79	Running the Riemann tests	59
3.80	Download the Riemann syntax checker	60
3.81	Build the Riemann syntax checker	60
3.82	Syntax check a Riemann configuration	60
3.83	Configuring additional RAM	61
A.1	A declarative statement	67

A.2 Getting lein	69
A.3 Auto-installing lein	69
A.4 Launching REPL	70
A.5 The REPL shell	70
A.6 Our first Clojure value	70
A.7 Our first Clojure integer	71
A.8 Our first Clojure string	71
A.9 Our first Clojure Booleans	71
A.10 The Clojure function syntax	72
A.11 Our first Clojure function	72
A.12 The fully qualified + function	73
A.13 Unable to resolve symbol	74
A.14 Quoting a symbol	74
A.15 The type function	74
A.16 A Clojure list	75
A.17 An unquoted Clojure list	75
A.18 Adding an element to a list	76
A.19 Working with lists	76
A.20 Creating a list	77
A.21 A Clojure vector	77
A.22 Adding an element to a vector	77
A.23 Getting the last element in a vector	78
A.24 Counting elements in a vector	78
A.25 Using a vector as a function	78
A.26 Creating or converting vectors	79
A.27 A Clojure set	79
A.28 Adding to a set	79
A.29 Removing an element from a set	80
A.30 Checking for a value inside a set	80
A.31 Using the set as a function	80
A.32 Making a set	81

A.33 A Clojure map	81
A.34 Getting a Clojure map value	81
A.35 Getting a missing Clojure map value	82
A.36 Getting a default value from a map	82
A.37 Using a map as a function	82
A.38 Using a keyword as a function	83
A.39 Using assoc to add a key/value	83
A.40 Replacing a key/value with assoc	83
A.41 Removing a key/value with dissoc	83
A.42 The str function	84
A.43 Concatenating a string	84
A.44 The inc function again	85
A.45 The fn function	85
A.46 Running our first fn function	85
A.47 The fn function shortcut	86
A.48 Calling the fn function shortcut	86
A.49 Creating a var	86
A.50 Evaluating a symbol	87
A.51 Using the type function on the symbol	87
A.52 Creating our first named function	88
A.53 Calling our grow function	88
A.54 Using the defn form	89
A.55 Adding a second argument	89
A.56 Calling our grow function again	89
A.57 Calling grow with our second argument	89
A.58 Adding a second argument	90
A.59 Using the doc function	90

Index

- Chef, 8
- Clojure, 3, 12, 47, 66
 - *, 88
 - +, 72
 - assoc, 83
 - conj, 75, 77, 79
 - contains?, 80
 - count, 78
 - def, 50, 86
 - defn, 88
 - deftest, 57
 - disj, 80
 - dissoc, 83
 - doc, 90
 - first, 76
 - fn, 85
 - get, 81
 - hash, 81
 - homoiconicity, 73
 - last, 77
 - Leiningen, 68
 - list, 72, 75
 - map, 81
 - namespaces, 48, 87
 - ns, 48
 - nth, 76
 - quoting, 74
 - second, 76
 - set, 79, 80
 - str, 84
 - style, 91
 - symbols, 73
 - type, 87
 - types, 70
 - var, 50, 87
 - vec, 78
 - vector, 77
- Configuration management, 5, 6, 8, 14, 67
- deftest, 57
- Docker, 8
- Events, 19
- Functional programming, 66
- Ganglia, 64

-
- Git, 60
 - Grafana, 3
 - Graphite, 3
 - index, 21
 - Java, 3
 - JVM, 3
 - Leiningen, 60, 68
 - Munin, 64
 - PagerDuty, 55
 - Postal, 50
 - Puppet, 8
 - Reimann
 - REPL, 68
 - REPL, 68
 - Riemann, 2, 4
 - /var/log/riemann/riemann.log, 24
 - AGGRESSIVE_OPTS, 61
 - async-queue, 37
 - asynchronous streams, 37
 - C client, 28
 - client, 37
 - Clojure DSL, 12
 - configuration, 9, 13
 - connecting servers, 35
 - dashboard, 12
 - default, 24, 45
 - email, 47
 - event, 19
 - expired stream, 52
 - EXTRA_JAVA_OPTS, 61
 - failover, 62
 - filtering streams, 32
 - forwarding events, 37
 - high availability, 62
 - HUP, 18
 - index, 21, 24, 45
 - installation, 5
 - JMX, 63
 - Leiningen, 68
 - logging, 10, 14, 15, 26
 - mailer, 47
 - namespacing, 48
 - network, 15
 - ns, 48
 - PagerDuty, 55
 - Performance, 61
 - ports, 15
 - Postal, 50
 - prn, 26
 - REPL, 17
 - require, 38
 - rollup, 54
 - scaling, 62
 - SIGHUP, 18
 - Slack, 55
 - SSL, 17
 - Streams, 21
 - streams, 24
 - syntax checking, 60

tagged, 32
tagged-all, 32
tagged-any, 32
tap, 57
testing, 57
throttle, 54
TLS, 17
tools, 11
TTL, 21
where, 32, 41, 52

sendmail, 50

Slack, 55

Streams, 21

Throttle, 54

Vagrant, 8

Websockets, 15

Thanks! I hope you enjoyed the book.

© Copyright 2018 - James Turnbull <james@lovedthanlost.net>

