Apache Flume: Distributed Log Collection for Hadoop

Second Edition

Design and implement a series of Flume agents to send streamed data into Hadoop

Steve Hoffman

Apache Flume: Distributed Log Collection for Hadoop

Second Edition

Design and implement a series of Flume agents to send streamed data into Hadoop

Steve Hoffman

[PACKT] open source*
community experience distilled

PUBLISHING

Apache Flume: Distributed Log Collection for Hadoop
Second Edition

Credits

Author

Steve Hoffman

Reviewers

Sachin Handiekar

Michael Keane

Stefan Will

Commissioning Editor

Dipika Gaonkar

Acquisition Editor

Reshma Raman

Content Development Editor

Neetu Ann Mathew

Technical Editor

Menza Mathew

Copy Editors

Vikrant Phadke

Stuti Srivastava

Project Coordinator

Mary Alex

Proofreader

Simran Bhogal

Safis Editing

Indexer

Rekha Nair

Graphics

Sheetal Aute

Abhinash Sahu

Production Coordinator

Komal Ramchandani

Cover Work

Komal Ramchandani

About the Author

Steve Hoffman has 32 years of experience in software development, ranging from embedded software development to the design and implementation of large-scale, service-oriented, object-oriented systems. For the last 5 years, he has focused on infrastructure as code, including automated Hadoop and HBase implementations and data ingestion using Apache Flume. Steve holds a BS in computer engineering from the University of Illinois at Urbana-Champaign and an MS in computer science from DePaul University. He is currently a senior principal engineer at Orbitz Worldwide (`http://orbitz.com/`).

More information on Steve can be found at `http://bit.ly/bacoboy` and on Twitter at `@bacoboy`.

This is the first update to Steve's first book, *Apache Flume: Distributed Log Collection for Hadoop*, *Packt Publishing*.

> I'd again like to dedicate this updated book to my loving and supportive wife, Tracy. She puts up with a lot, and that is very much appreciated. I couldn't ask for a better friend daily by my side.
>
> My terrific children, Rachel and Noah, are a constant reminder that hard work does pay off and that great things can come from chaos.
>
> I also want to give a big thanks to my parents, Alan and Karen, for molding me into the somewhat satisfactory human I've become. Their dedication to family and education above all else guides me daily as I attempt to help my own children find their happiness in the world.

About the Reviewers

Sachin Handiekar is a senior software developer with over 5 years of experience in Java EE development. He graduated in computer science from the University of Greenwich, London, and currently works for a global consulting company, developing enterprise applications using various open source technologies, such as Apache Camel, ServiceMix, ActiveMQ, and ZooKeeper.

Sachin has a lot of interest in open source projects. He has contributed code to Apache Camel and developed plugins for Spring Social, which can be found at GitHub (`https://github.com/sachin-handiekar`).

He also actively writes about enterprise application development on his blog (`http://sachinhandiekar.com`).

Michael Keane has a BS in computer science from the University of Illinois at Urbana-Champaign. He has worked as a software engineer, coding almost exclusively in Java since JDK 1.1. He has also worked on the mission-critical medical device software, e-commerce, transportation, navigation, and advertising domains. He is currently a development leader for Conversant, where he maintains Flume flows of nearly 100 billion log lines per day.

Michael is a father of three, and besides work, he spends most of his time with his family and coaching youth softball.

Stefan Will is a computer scientist with a degree in machine learning and pattern recognition from the University of Bonn, Germany. For over a decade, he has worked for several start-ups in Silicon Valley and Raleigh, North Carolina, in the area of search and analytics. Presently, he leads the development of the search backend and real-time analytics platform at Zendesk, a provider of customer service software.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Hadoop is a great open source tool for shifting tons of unstructured data into something manageable so that your business can gain better insight into your customers' needs. It's cheap (mostly free), scales horizontally as long as you have space and power in your datacenter, and can handle problems that would crush your traditional data warehouse. That said, a little-known secret is that your Hadoop cluster requires you to feed it data. Otherwise, you just have a very expensive heat generator! You will quickly realize (once you get past the "playing around" phase with Hadoop) that you will need a tool to automatically feed data into your cluster. In the past, you had to come up with a solution for this problem, but no more! Flume was started as a project out of Cloudera, when its integration engineers had to keep writing tools over and over again for their customers to automatically import data. Today, the project lives with the Apache Foundation, is under active development, and boasts of users who have been using it in their production environments for years.

In this book, I hope to get you up and running quickly with an architectural overview of Flume and a quick-start guide. After that, we'll dive deep into the details of many of the more useful Flume components, including the very important file channel for the persistence of in-flight data records and the HDFS Sink for buffering and writing data into HDFS (the Hadoop File System). Since Flume comes with a wide variety of modules, chances are that the only tool you'll need to get started is a text editor for the configuration file.

By the time you reach the end of this book, you should know enough to build a highly available, fault-tolerant, streaming data pipeline that feeds your Hadoop cluster.

What this book covers

Chapter 1, *Overview and Architecture*, introduces Flume and the problem space that it's trying to address (specifically with regards to Hadoop). An architectural overview of the various components to be covered in later chapters is given.

Chapter 2, *A Quick Start Guide to Flume*, serves to get you up and running quickly. It includes downloading Flume, creating a "Hello, World!" configuration, and running it.

Chapter 3, *Channels*, covers the two major channels most people will use and the configuration options available for each of them.

Chapter 4, *Sinks and Sink Processors*, goes into great detail on using the HDFS Flume output, including compression options and options for formatting the data. Failover options are also covered so that you can create a more robust data pipeline.

Chapter 5, *Sources and Channel Selectors*, introduces several of the Flume input mechanisms and their configuration options. Also covered is switching between different channels based on data content, which allows the creation of complex data flows.

Chapter 6, *Interceptors, ETL, and Routing*, explains how to transform data in-flight as well as extract information from the payload to use with Channel Selectors to make routing decisions. Then this chapter covers tiering Flume agents using Avro serialization, as well as using the Flume command line as a standalone Avro client for testing and importing data manually.

Chapter 7, *Putting It All Together*, walks you through the details of an end-to-end use case from the web server logs to a searchable UI, backed by Elasticsearch as well as archival storage in HDFS.

Chapter 8, *Monitoring Flume*, discusses various options available for monitoring Flume both internally and externally, including Monit, Nagios, Ganglia, and custom hooks.

Chapter 9, *There Is No Spoon – the Realities of Real-time Distributed Data Collection*, is a collection of miscellaneous things to consider that are outside the scope of just configuring and using Flume.

What you need for this book

You'll need a computer with a Java Virtual Machine installed, since Flume is written in Java. If you don't have Java on your computer, you can download it from `http://java.com/`.

You will also need an Internet connection so that you can download Flume to run the Quick Start example.

This book covers Apache Flume 1.5.2.

Who this book is for

This book is for people responsible for implementing the automatic movement of data from various systems to a Hadoop cluster. If it is your job to load data into Hadoop on a regular basis, this book should help you to code yourself out of manual monkey work or from writing a custom tool you'll be supporting for as long as you work at your company.

Only basic knowledge of Hadoop and HDFS is required. Some custom implementations are covered, should your needs necessitate them. For this level of implementation, you will need to know how to program in Java.

Finally, you'll need your favorite text editor, since most of this book covers how to configure various Flume components via an agent's text configuration file.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and explanations of their meanings.

Code words in text are shown as follows: "If you want to use this feature, you set the `useDualCheckpoints` property to `true` and specify a location for that second checkpoint directory with the `backupCheckpointDir` property."

A block of code is set as follows:

```
agent.sinks.k1.hdfs.path=/logs/apache/access
agent.sinks.k1.hdfs.filePrefix=access
agent.sinks.k1.hdfs.fileSuffix=.log
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
agent.sources.s1.command=uptime
agent.sources.s1.restart=true
agent.sources.s1.restartThrottle=60000
```

Any command-line input or output is written as follows:

```
$ tar -zxf apache-flume-1.5.2.tar.gz
$ cd apache-flume-1.5.2
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "**Flume** was first introduced in Cloudera's **CDH3** distribution in 2011."

> Warnings or important notes appear in a box like this.

> Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Overview and Architecture

If you are reading this book, chances are you are swimming in oceans of data. Creating mountains of data has become very easy, thanks to Facebook, Twitter, Amazon, digital cameras and camera phones, YouTube, Google, and just about anything else you can think of being connected to the Internet. As a provider of a website, 10 years ago, your application logs were only used to help you troubleshoot your website. Today, this same data can provide a valuable insight into your business and customers if you know how to pan gold out of your river of data.

Furthermore, as you *are* reading this book, you are also aware that Hadoop was created to solve (partially) the problem of sifting through mountains of data. Of course, this only works if you can reliably load your Hadoop cluster with data for your data scientists to pick apart.

Getting data into and out of Hadoop (in this case, the **Hadoop File System**, or **HDFS**) isn't hard; it is just a simple command, such as:

```
% hadoop fs --put data.csv .
```

This works great when you have all your data neatly packaged and ready to upload.

However, your website is creating data all the time. How often should you batch load data to HDFS? Daily? Hourly? Whatever processing period you choose, eventually somebody always asks "can you get me the data sooner?" What you really need is a solution that can deal with streaming logs/data.

Turns out you aren't alone in this need. **Cloudera**, a provider of professional services for Hadoop as well as their own distribution of Hadoop, saw this need over and over when working with their customers. Flume was created to fill this need and create a standard, simple, robust, flexible, and extensible tool for data ingestion into Hadoop.

Flume 0.9

Flume was first introduced in Cloudera's **CDH3** distribution in 2011. It consisted of a federation of worker daemons (**agents**) configured from a centralized master (or **masters**) via **Zookeeper** (a federated configuration and coordination system). From the master, you could check the agent status in a web UI as well as push out configuration centrally from the UI or via a command-line shell (both really communicating via Zookeeper to the worker agents).

Data could be sent in one of three modes: **Best effort** (**BE**), **Disk Failover** (**DFO**), and **End-to-End** (**E2E**). The masters were used for the E2E mode acknowledgements and multimaster configuration never really matured, so you usually only had one master, making it a central point of failure for E2E data flows. The BE mode is just what it sounds like: the agent would try to send the data, but if it couldn't, the data would be discarded. This mode is good for things such as metrics, where gaps can easily be tolerated, as new data is just a second away. The DFO mode stores undeliverable data to the local disk (or sometimes, a local database) and would keep retrying until the data could be delivered to the next recipient in your data flow. This is handy for those planned (or unplanned) outages, as long as you have sufficient local disk space to buffer the load.

In June, 2011, Cloudera moved control of the Flume project to the Apache Foundation. It came out of the incubator status a year later in 2012. During the incubation year, work had already begun to refactor Flume under the Star-Trek-themed tag, **Flume-NG** (**Flume the Next Generation**).

Flume 1.X (Flume-NG)

There were many reasons why Flume was refactored. If you are interested in the details, you can read about them at `https://issues.apache.org/jira/browse/FLUME-728`. What started as a refactoring branch eventually became the main line of development as Flume 1.X.

The most obvious change in Flume 1.X is that the centralized configuration master(s) and Zookeeper are gone. The configuration in Flume 0.9 was overly verbose, and mistakes were easy to make. Furthermore, centralized configuration was really outside the scope of Flume's goals. Centralized configuration was replaced with a simple on-disk configuration file (although the configuration provider is pluggable so that it can be replaced). These configuration files are easily distributed using tools such as **cf-engine**, **Chef**, and **Puppet**. If you are using a Cloudera distribution, take a look at Cloudera Manager to manage your configurations. About two years ago, they created a free version with no node limit, so it may be an attractive option for you. Just be sure you don't manage these configurations manually, or you'll be editing these files manually forever.

Another major difference in Flume 1.X is that the reading of input data and the writing of output data are now handled by different worker threads (called **Runners**). In Flume 0.9, the input thread also did the writing to the output (except for failover retries). If the output writer was slow (rather than just failing outright), it would block Flume's ability to ingest data. This new asynchronous design leaves the input thread blissfully unaware of any downstream problem.

The first edition of this book covered all the versions of Flume up till Version 1.3.1. This second edition will cover till Version 1.5.2 (the current version at the time of writing this).

The problem with HDFS and streaming data/logs

HDFS isn't a real filesystem, at least not in the traditional sense, and many of the things we take for granted with normal filesystems don't apply here, such as being able to mount it. This makes getting your streaming data into Hadoop a little more complicated.

In a regular **POSIX**-style filesystem, if you open a file and write data, it still exists on the disk before the file is closed. That is, if another program opens the same file and starts reading, it will get the data already flushed by the writer to the disk. Furthermore, if this writing process is interrupted, any portion that made it to disk is usable (it may be incomplete, but it exists).

In HDFS, the file exists only as a directory entry; it shows zero length until the file is closed. This means that if data is written to a file for an extended period without closing it, a network disconnect with the client will leave you with nothing but an empty file for all your efforts. This may lead you to the conclusion that it would be wise to write small files so that you can close them as soon as possible.

The problem is that Hadoop doesn't like lots of tiny files. As the HDFS filesystem metadata is kept in memory on the NameNode, the more files you create, the more RAM you'll need to use. From a MapReduce prospective, tiny files lead to poor efficiency. Usually, each Mapper is assigned a single block of a file as the input (unless you have used certain compression codecs). If you have lots of tiny files, the cost of starting the worker processes can be disproportionally high compared to the data it is processing. This kind of block fragmentation also results in more Mapper tasks, increasing the overall job run times.

These factors need to be weighed when determining the rotation period to use when writing to HDFS. If the plan is to keep the data around for a short time, then you can lean toward the smaller file size. However, if you plan on keeping the data for a very long time, you can either target larger files or do some periodic cleanup to compact smaller files into fewer, larger files to make them more MapReduce friendly. After all, you only ingest the data once, but you might run a MapReduce job on that data hundreds or thousands of times.

Sources, channels, and sinks

The Flume agent's architecture can be viewed in this simple diagram. Inputs are called **sources** and outputs are called **sinks**. **Channels** provide the glue between sources and sinks. All of these run inside a daemon called an **agent**.
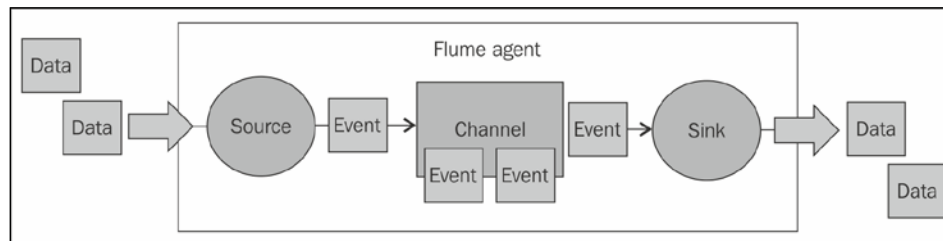
Keep in mind:

- A source writes events to one or more channels.
- A channel is the holding area as events are passed from a source to a sink.
- A sink receives events from one channel only.
- An agent can have many channels.

Flume events

The basic payload of data transported by Flume is called an event. An event is composed of zero or more headers and a body.

The headers are key/value pairs that can be used to make routing decisions or carry other structured information (such as the timestamp of the event or the hostname of the server from which the event originated). You can think of it as serving the same function as HTTP headers—a way to pass additional information that is distinct from the body.

The body is an array of bytes that contains the actual payload. If your input is comprised of tailed log files, the array is most likely a UTF-8-encoded string containing a line of text.
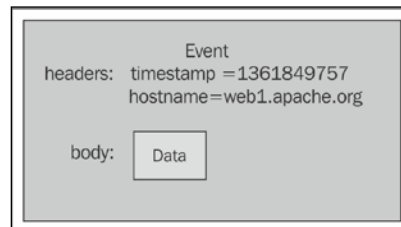
Flume may add additional headers automatically (like when a source adds the hostname where the data is sourced or creating an event's timestamp), but the body is mostly untouched unless you edit it en route using interceptors.

Interceptors, channel selectors, and sink processors

An **interceptor** is a point in your data flow where you can inspect and alter Flume events. You can chain zero or more interceptors after a source creates an event. If you are familiar with the **AOP Spring Framework**, think `MethodInterceptor`. In Java Servlets, it's similar to `ServletFilter`. Here's an example of what using four chained interceptors on a source might look like:
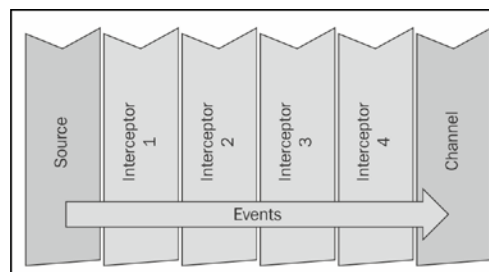
Channel selectors are responsible for how data moves from a source to one or more channels. Flume comes packaged with two channel selectors that cover most use cases you might have, although you can write your own if need be. A replicating channel selector (the default) simply puts a copy of the event into each channel, assuming you have configured more than one. In contrast, a multiplexing channel selector can write to different channels depending on some header information. Combined with some interceptor logic, this duo forms the foundation for routing input to different channels.

Finally, a **sink processor** is the mechanism by which you can create failover paths for your sinks or load balance events across multiple sinks from a channel.

Tiered data collection (multiple flows and/or agents)

You can chain your Flume agents depending on your particular use case. For example, you may want to insert an agent in a tiered fashion to limit the number of clients trying to connect directly to your Hadoop cluster. More likely, your source machines don't have sufficient disk space to deal with a prolonged outage or maintenance window, so you create a tier with lots of disk space between your sources and your Hadoop cluster.

In the following diagram, you can see that there are two places where data is created (on the left-hand side) and two final destinations for the data (the HDFS and ElasticSearch cloud bubbles on the right-hand side). To make things more interesting, let's say one of the machines generates two kinds of data (let's call them square and triangle data). You can see that in the lower-left agent, we use a multiplexing channel selector to split the two kinds of data into different channels. The rectangle channel is then routed to the agent in the upper-right corner (along with the data coming from the upper-left agent). The combined volume of events is written together in HDFS in Datacenter 1. Meanwhile, the triangle data is sent to the agent that writes to ElasticSearch in Datacenter 2. Keep in mind that data transformations can occur after any source. How all of these components can be used to build complicated data workflows will be become clear as we proceed.
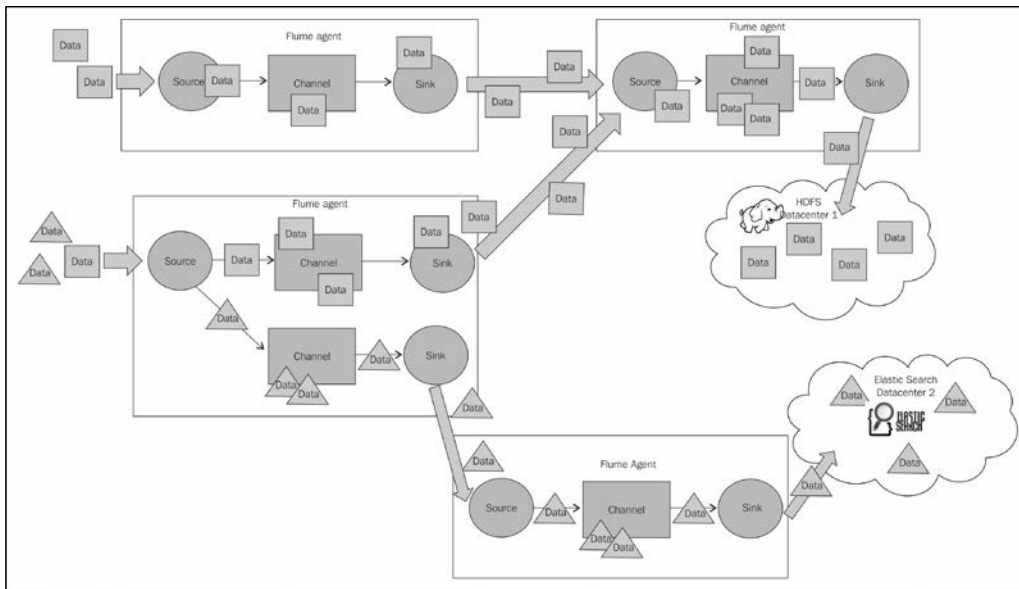
The Kite SDK

One of the new technologies incorporated in Flume, starting with Version 1.4, is something called a Morphline. You can think of a Morphline as a series of commands chained together to form a data transformation pipe.

If you are a fan of pipelining Unix commands, this will be very familiar to you. The commands themselves are intended to be small, single-purpose functions that when chained together create powerful logic. In many ways, using a Morphline command chain can be identical in functionality to the interceptor paradigm just mentioned. There is a Morphline interceptor we will cover in *Chapter 6, Interceptors, ETL, and Routing*, which you can use instead of, or in addition to, the included Java-based interceptors.

> To get an idea of how useful these commands can be, take a look at the handy `grok` command and its included extensible regular expression library at `https://github.com/kite-sdk/kite/blob/master/kite-morphlines/kite-morphlines-core/src/test/resources/grok-dictionaries/grok-patterns`

Many of the custom Java interceptors that I've written in the past were to modify the body (data) and can easily be replaced with an out-of-the-box Morphline command chain. You can get familiar with the Morphline commands by checking out their reference guide at `http://kitesdk.org/docs/current/kite-morphlines/morphlinesReferenceGuide.html`

Flume Version 1.4 also includes a Morphline-backed sink used primarily to feed data into Solr. We'll see more of this in *Chapter 4*, *Sinks and Sink Processors, Morphline Solr Search Sink*.

Morphlines are just one component of the KiteSDK included in Flume. Starting with Version 1.5, Flume has added experimental support for KiteData, which is an effort to create a standard library for datasets in Hadoop. It looks very promising, but it is outside the scope of this book.
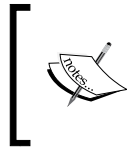
> Please see the project home page for more information, as it will certainly become more prominent in the Hadoop ecosystem as the technology matures. You can read all about the KiteSDK at `http://kitesdk.org`.

Summary

In this chapter, we discussed the problem that Flume is attempting to solve: getting data into your Hadoop cluster for data processing in an easily configured, reliable way. We also discussed the Flume agent and its logical components, including events, sources, channel selectors, channels, sink processors, and sinks. Finally, we briefly discussed Morphlines as a powerful new ETL (**Extract, Transform, Load**) library, starting with Version 1.4 of Flume.

The next chapter will cover these in more detail, specifically, the most commonly used implementations of each. Like all good open source projects, almost all of these components are extensible if the bundled ones don't do what you need them to do.

2

A Quick Start Guide to Flume

As we covered some of the basics in the previous chapter, this chapter will help you get started with Flume. So, let's start with the first step: downloading and configuring Flume.

Downloading Flume

Let's download Flume from `http://flume.apache.org/`. Look for the download link in the side navigation. You'll see two compressed `.tar` archives available along with the checksum and GPG signature files used to verify the archives. Instructions to verify the download are on the website, so I won't cover them here. Checking the checksum file contents against the actual checksum verifies that the download was not corrupted. Checking the signature file validates that all the files you are downloading (including the checksum and signature) came from Apache and not some nefarious location. Do you really need to verify your downloads? In general, it is a good idea and it is recommended by Apache that you do so. If you choose not to, I won't tell.

The binary distribution archive has `bin` in the name, and the source archive is marked with `src`. The source archive contains just the Flume source code. The binary distribution is much larger because it contains not only the Flume source and the compiled Flume components (jars, javadocs, and so on), but also all the dependent Java libraries. The binary package contains the same Maven POM file as the source archive, so you can always recompile the code even if you start with the binary distribution.

Go ahead, download and verify the binary distribution to save us some time in getting started.

Flume in Hadoop distributions

Flume is available with some Hadoop distributions. The distributions supposedly provide bundles of Hadoop's core components and satellite projects (such as Flume) in a way that ensures things such as version compatibility and additional bug fixes are taken into account. These distributions aren't better or worse; they're just different.

There are benefits to using a distribution. Someone else has already done the work of pulling together all the version-compatible components. Today, this is less of an issue since the Apache BigTop project started (`http://bigtop.apache.org/`). Nevertheless, having prebuilt standard OS packages, such as RPMs and DEBs, ease installation as well as provide startup/shutdown scripts. Each distribution has different levels of free and paid options, including paid professional services if you really get into a situation you just can't handle.

There are downsides, of course. The version of Flume bundled in a distribution will often lag quite a bit behind the Apache releases. If there is a new or bleeding-edge feature you are interested in using, you'll either be waiting for your distribution's provider to backport it for you, or you'll be stuck patching it yourself. Furthermore, while the distribution providers do a fair amount of testing, such as any general-purpose platform, you will most likely encounter something that their testing didn't cover, in which case, you are still on the hook to come up with a workaround or dive into the code, fix it, and hopefully, submit that patch back to the open source community (where, at a future point, it'll make it into an update of your distribution or the next version).

So, things move slower in a Hadoop distribution world. You can see that as good or bad. Usually, large companies don't like the instability of bleeding-edge technology or making changes often, as change can be the most common cause of unplanned outages. You'd be hard pressed to find such a company using the bleeding-edge Linux kernel rather than something like **Red Hat Enterprise Linux** (**RHEL**), CentOS, Ubuntu LTS, or any of the other distributions whose target is stability and compatibility. If you are a startup building the next Internet fad, you might need that bleeding-edge feature to get a leg up on the established competition.

If you are considering a distribution, do the research and see what you are getting (or not getting) with each. Remember that each of these offerings is hoping that you'll eventually want and/or need their Enterprise offering, which usually doesn't come cheap. Do your homework.

Here's a short, nondefinitive list of some of the more established players. For more information, refer to the following links:

- Cloudera: `http://cloudera.com/`
- Hortonworks: `http://hortonworks.com/`
- MapR: `http://mapr.com/`

An overview of the Flume configuration file

Now that we've downloaded Flume, let's spend some time going over how to configure an agent.

A Flume agent's default configuration provider uses a simple Java property file of key/value pairs that you pass as an argument to the agent upon startup. As you can configure more than one agent in a single file, you will need to additionally pass an **agent identifier** (called a **name**) so that it knows which configurations to use. In my examples where I'm only specifying one agent, I'm going to use the name `agent`.

By default, the configuration property file is monitored for changes every 30 seconds. If a change is detected, Flume will attempt to reconfigure itself. In practice, many of the configuration settings cannot be changed after the agent has started. Save yourself some trouble and pass the undocumented `--no-reload-conf` argument when starting the agent (except in development situations perhaps).

If you use the Cloudera distribution, the passing of this flag is currently not possible. I've opened a ticket to fix that at `https://issues.cloudera.org/browse/DISTRO-648`. If this is important to you, please vote it up.

Each agent is configured, starting with three parameters:

```
agent.sources=<list of sources>
agent.channels=<list of channels>
agent.sinks=<list of sinks>
```

Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Each source, channel, and sink also has a unique name within the context of that agent. For example, if I'm going to transport my Apache access logs, I might define a channel named `access`. The configurations for this channel would all start with the `agent.channels.access` prefix. Each configuration item has a type property that tells Flume what kind of source, channel, or sink it is. In this case, we are going to use an in-memory channel whose type is `memory`. The complete configuration for the channel named `access` in the agent named `agent` would be:

```
agent.channels.access.type=memory
```

Any arguments to a source, channel, or sink are added as additional properties using the same prefix. The memory channel has a capacity parameter to indicate the maximum number of Flume events it can hold. Let's say we didn't want to use the default value of 100; our configuration would now look like this:

```
agent.channels.access.type=memory
agent.channels.access.capacity=200
```

Finally, we need to add the `access` channel name to the `agent.channels` property so that the agent knows to load it:

```
agent.channels=access
```

Let's look at a complete example using the canonical Hello, World! example.

Starting up with "Hello, World!"

No technical book would be complete without a Hello, World! example. Here is the configuration file we'll be using:

```
agent.sources=s1
agent.channels=c1
agent.sinks=k1

agent.sources.s1.type=netcat
agent.sources.s1.channels=c1
agent.sources.s1.bind=0.0.0.0
agent.sources.s1.port=12345

agent.channels.c1.type=memory

agent.sinks.k1.type=logger
agent.sinks.k1.channel=c1
```

Here, I've defined one agent (called `agent`) who has a source named `s1`, a channel named `c1`, and a sink named `k1`.

The s1 source's type is netcat, which simply opens a socket listening for events (one line of text per event). It requires two parameters: a bind IP and a port number. In this example, we are using 0.0.0.0 for a bind address (the Java convention to specify listen on any address) and port 12345. The source configuration also has a parameter called channels (plural), which is the name of the channel(s) the source will append events to, in this case, c1. It is plural, because you can configure a source to write to more than one channel; we just aren't doing that in this simple example.

The channel named c1 is a memory channel with a default configuration.

The sink named k1 is of the logger type. This is a sink that is mostly used for debugging and testing. It will log all events at the INFO level using Log4j, which it receives from the configured channel, in this case, c1. Here, the channel keyword is singular because a sink can only be fed data from one channel.

Using this configuration, let's run the agent and connect to it using the Linux netcat utility to send an event.

First, explode the .tar archive of the binary distribution we downloaded earlier:

```
$ tar -zxf apache-flume-1.5.2-bin.tar.gz
$ cd apache-flume-1.5.2-bin
```

Next, let's briefly look at the help. Run the flume-ng command with the help command:

```
$ ./bin/flume-ng help
Usage: ./bin/flume-ng <command> [options]...


commands:
  help                  display this help text
  agent                 run a Flume agent
  avro-client           run an avro Flume client
  version               show Flume version info


global options:
  --conf,-c <conf>      use configs in <conf> directory
  --classpath,-C <cp>   append to the classpath
  --dryrun,-d           do not actually start Flume, just print the
command
  --plugins-path <dirs> colon-separated list of plugins.d directories.
See the
```

```
                        plugins.d section in the user guide for more
details.

                        Default: $FLUME_HOME/plugins.d
  -Dproperty=value      sets a Java system property value
  -Xproperty=value      sets a Java -X option


agent options:
  --conf-file,-f <file> specify a config file (required)
  --name,-n <name>      the name of this agent (required)
  --help,-h             display help text


avro-client options:
  --rpcProps,-P <file>   RPC client properties file with server
connection params
  --host,-H <host>       hostname to which events will be sent
  --port,-p <port>       port of the avro source
  --dirname <dir>        directory to stream to avro source
  --filename,-F <file>   text file to stream to avro source (default: std
input)
  --headerFile,-R <file> File containing event headers as key/value pairs
on each new line
  --help,-h              display help text


  Either --rpcProps or both --host and --port must be specified.


Note that if <conf> directory is specified, then it is always included
first in the classpath.
```

As you can see, there are two ways with which you can invoke the command (other than the simple `help` and `version` commands). We will be using the `agent` command. The use of `avro-client` will be covered later.

The `agent` command has two required parameters: a configuration file to use and the agent name (in case your configuration contains multiple agents).

Let's take our sample configuration and open an editor (`vi` in my case, but use whatever you like):

```
$ vi conf/hw.conf
```

Next, place the contents of the preceding configuration into the editor, save, and exit back to the shell.

Now you can start the agent:

```
$ ./bin/flume-ng agent -n agent -c conf -f conf/hw.conf -Dflume.root.
logger=INFO,console
```

The `-Dflume.root.logger` property overrides the root logger in `conf/log4j.properties` to use the console appender.

If we didn't override the root logger, everything would still work, but the output would go to the `log/flume.log` file instead of being based on the contents of the default configuration file. Of course, you can edit the `conf/log4j.properties` file and change the `flume.root.logger` property (or anything else you like). To change just the path or filename, you can set the `flume.log.dir` and `flume.log.file` properties in the configuration file or pass additional flags on the command line as follows:

```
$ ./bin/flume-ng agent -n agent -c conf -f conf/hw.conf -Dflume.root.
logger=INFO,console -Dflume.log.dir=/tmp -Dflume.log.file=flume-agent.log
```

You might ask why you need to specify the `-c` parameter, as the `-f` parameter contains the complete relative path to the configuration. The reason for this is that the Log4j configuration file should be included on the class path.

If you left the `-c` parameter off the command, you'll see this error:

```
Warning: No configuration directory set! Use --conf <dir> to override.
```

```
log4j:WARN No appenders could be found for logger (org.apache.flume.
lifecycle.LifecycleSupervisor).
```

```
log4j:WARN Please initialize the log4j system properly.
```

```
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for
more info.
```

But you didn't do that so you should see these key log lines:

```
2014-10-05 15:39:06,109 (conf-file-poller-0) [INFO - org.apache.flume.
conf.FlumeConfiguration.validateConfiguration(FlumeConfiguration.
java:140)] Post-validation flume configuration contains configuration for
agents: [agent]
```

This line tells you that your agent starts with the name `agent`.

Usually you'd look for this line only to be sure you started the right configuration when you have multiple configurations defined in your configuration file.

```
2014-10-05 15:39:06,076 (conf-file-poller-0) [INFO - org.apache.flume.
node.PollingPropertiesFileConfigurationProvider$FileWatcherRunnable.
run(PollingPropertiesFileConfigurationProvider.java:133)] Reloading
configuration file:conf/hw.conf
```

This is another sanity check to make sure you are loading the correct file, in this case our `hw.conf` file.

```
2014-10-05 15:39:06,221 (conf-file-poller-0) [INFO - org.apache.
flume.node.Application.startAllComponents(Application.java:138)]
Starting new configuration:{ sourceRunners:{s1=EventDrivenSourceRu
nner: { source:org.apache.flume.source.NetcatSource{name:s1,state:I
DLE} }} sinkRunners:{k1=SinkRunner: { policy:org.apache.flume.sink.
DefaultSinkProcessor@442fbe47 counterGroup:{ name:null counters:{} } }}
channels:{c1=org.apache.flume.channel.MemoryChannel{name: c1}} }
```

Once all the configurations have been parsed, you will see this message, which shows you everything that was configured. You can see `s1`, `c1`, and `k1`, and which Java classes are actually doing the work. As you probably guessed, `netcat` is a convenience for `org.apache.flume.source.NetcatSource`. We could have used the class name if we wanted. In fact, if I had my own custom source written, I would use its class name for the source's `type` parameter. You cannot define your own short names without patching the Flume distribution.

```
2014-10-05 15:39:06,427 (lifecycleSupervisor-1-0) [INFO - org.apache.
flume.source.NetcatSource.start(NetcatSource.java:164)] Created
serverSocket:sun.nio.ch.ServerSocketChannelImpl[/0.0.0.0:12345]
```

Here, we see that our source is now listening on port 12345 for the input. So, let's send some data to it.

Finally, open a second terminal. We'll use the `nc` command (you can use Telnet or anything else similar) to send the `Hello World` string and press the Return (*Enter*) key to mark the end of the event:

```
% nc localhost 12345
Hello World
OK
```

The `OK` message came from the agent after we pressed the Return key, signifying that it accepted the line of text as a single Flume event. If you look at the agent log, you will see the following:

```
2014-10-05 15:44:11,215 (SinkRunner-PollingRunner-DefaultSinkProcessor)
[INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.
java:70)] Event: { headers:{} body: 48 65 6C 6C 6F 20 57 6F 72 6C 64
Hello World }
```

This log message shows you that the Flume event contains no headers (`NetcatSource` doesn't add any itself). The body is shown in hexadecimal along with a string representation (for us humans to read, in this case, our `Hello World` message).

If I send the following line and then press the *Enter* key, you'll get an `OK` message:

The quick brown fox jumped over the lazy dog.

You'll see this in the agent's log:

```
2014-10-05 15:44:57,232 (SinkRunner-PollingRunner-DefaultSinkProcessor)
[INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:70)]
Event: { headers:{} body: 54 68 65 20 71 75 69 63 6B 20 62 72 6F 77 6E 20
The quick brown  }
```

The event appears to have been truncated. The logger sink, by design, limits the body content to 16 bytes to keep your screen from being filled with more than what you'd need in a debugging context. If you need to see the full contents for debugging, you should use a different sink, perhaps the `file_roll` sink, which would write to the local filesystem.

Summary

In this chapter, we covered how to download the Flume binary distribution. We created a simple configuration file that included one source writing to one channel, feeding one sink. The source listened on a socket for network clients to connect to and to send it event data. These events were written to an in-memory channel and then fed to a Log4j sink to become the output. We then connected to our listening agent using the Linux netcat utility and sent some string events to our Flume agent's source. Finally, we verified that our Log4j-based sink wrote the events out.

In the next chapter, we'll take a detailed look at the two major channel types you'll most likely use in your data processing workflows: the memory channel and the file channel.

We will also take a look at a new experimental channel, introduced in Version 1.5 of Flume, called the Spillable Memory Channel, which attempts to be a hybrid of the other two.

For each type, we'll discuss all the configuration knobs available to you, when and why you might want to deviate from the defaults, and most importantly, why to use one over the other.

3
Channels

In Flume, a **channel** is the construct used between sources and sinks. It provides a buffer for your in-flight events after they are read from sources until they can be written to sinks in your data processing pipelines.

The primary types we'll cover here are a memory-backed/nondurable channel and a local-filesystem-backed/durable channel. Starting with Flume 1.5, an experimental hybrid memory and file channel called the Spillable Memory Channel is introduced. The durable file channel flushes all changes to disk before acknowledging the receipt of the event to the sender. This is considerably slower than using the nondurable memory channel, but it provides recoverability in the event of system or Flume agent restarts. Conversely, the memory channel is much faster, but failure results in data loss and it has much lower storage capacity when compared to the multiterabyte disks backing the file channel. This is why the Spillable Memory Channel was created. In theory, you get the benefits of memory speed until the memory fills up due to flow backpressure. At this point, the disk will be used to store the events—and with that comes much larger capacity. There are trade-offs here as well, as performance is now variable depending on how the entire flow is performing. Ultimately, the channel you choose depends on your specific use cases, failure scenarios, and risk tolerance.

That said, regardless of what channel you choose, if your rate of ingest from the sources into the channel is greater than the rate at which the sink can write data, you will exceed the capacity of the channel and throw a `ChannelException`. What your source does or doesn't do with that `ChannelException` is source-specific, but in some cases, data loss is possible, so you'll want to avoid filling channels by sizing things properly. In fact, you always want your sink to be able to write faster than your source input. Otherwise, you might get into a situation where once your sink falls behind, you can never catch up. If your data volume tracks with the site usage, you can have higher volumes during the day and lower volumes at night, giving your channels time to drain. In practice, you'll want to try and keep the channel depth (the number of events currently in the channel) as low as possible because time spent in the channel translates to a time delay before reaching the final destination.

The memory channel

A memory channel, as expected, is a channel where in-flight events are stored in memory. As memory is (usually) orders of magnitude faster than the disk, events can be ingested much more quickly, resulting in reduced hardware needs. The downside of using this channel is that an agent failure (hardware problem, power outage, JVM crash, Flume restart, and so on) results in the loss of data. Depending on your use case, this might be perfectly fine. System metrics usually fall into this category, as a few lost data points isn't the end of the world. However, if your events represent purchases on your website, then a memory channel would be a poor choice.

To use the memory channel, set the `type` parameter on your named channel to `memory`.

```
agent.channels.c1.type=memory
```

This defines a memory channel named `c1` for the agent named `agent`.

Here is a table of configuration parameters you can adjust from the default values:

Key	Required	Type	Default
type	Yes	String	memory
capacity	No	int	100
transactionCapacity	No	int	100
byteCapacityBufferPercentage	No	int (percent)	20%
byteCapacity	No	long (bytes)	80% of JVM Heap
keep-alive	No	int	3 (seconds)

The default capacity of this channel is 100 events. This can be adjusted by setting the capacity property as follows:

```
agent.channels.c1.capacity=200
```

Remember that if you increase this value, you will most likely have to increase your Java heap space using the `-Xmx`, and optionally `-Xms`, parameters.

Another capacity-related setting you can set is `transactionCapacity`. This is the maximum number of events that can be written, also called a **put**, by a source's `ChannelProcessor`, the component responsible for moving data from the source to the channel, in a single transaction. This is also the number of events that can be read, also called a **take**, in a single transaction by the SinkProcessor, which is the component responsible for moving data from the channel to the sink. You might want to set this higher in order to decrease the overhead of the transaction wrapper, which might speed things up. The downside to increasing this is that a source would have to roll back more data in the event of a failure.

> Flume only provides transactional guarantees for each channel in each individual agent. In a multiagent, multichannel configuration, duplicates and out-of-order delivery are likely but should not be considered the norm. If you are getting duplicates in nonfailure conditions, it means that you need to continue tuning your Flume configurations.

If you are using a sink that writes some place that benefits from larger batches of work (such as HDFS), you might want to set this higher. Like many things, the only way to be sure is to run performance tests with different values. This blog post from Flume committer Mike Percy should give you some good starting points: `http://bit.ly/flumePerfPt1`.

The `byteCapacityBufferPercentage` and `byteCapacity` parameters were introduced in `https://issues.apache.org/jira/browse/FLUME-1535` as a means to size the memory channel capacity using the number of bytes used rather than the number of events as well as trying to avoid `OutOfMemoryErrors`. If your events have a large variance in size, you might be tempted to use these settings to adjust the capacity, but be warned that calculations are estimated from the event's body only. If you have any headers, which you will, your actual memory usage will be higher than the configured values.

Finally, the `keep-alive` parameter is the time the thread writing data into the channel will wait when the channel is full, before giving up. As data is being drained from the channel at the same time, if space opens up before the timeout expires, the data will be written to the channel rather than throwing an exception back to the source. You might be tempted to set this value very high, but remember that waiting for a write to a channel will block the data flowing into your source, which might cause data to back up in an upstream agent. Eventually, this might result in events being dropped. You need to size for periodic spikes in traffic as well as temporary planned (and unplanned) maintenance.

The file channel

A file channel is a channel that stores events to the local filesystem of the agent. Though it's slower than the memory channel, it provides a durable storage path that can survive most issues and should be used in use cases where a gap in your data flow is undesirable.

This durability is provided by a combination of a **Write Ahead Log** (**WAL**) and one or more file storage directories. The WAL is used to track all input and output from the channel in an atomically safe way. This way, if the agent is restarted, the WAL can be replayed to make sure all the events that came into the channel (puts) have been written out (takes) before the stored data can be purged from the local filesystem.

Additionally, the file channel supports the encryption of data written to the filesystem if your data handling policy requires that all data on the disk (even temporarily) be encrypted. I won't cover this here, but should you need it, there is an example in the Flume User Guide (`http://flume.apache.org/FlumeUserGuide.html`). Keep in mind that using encryption will reduce the throughput of your file channel.

To use the file channel, set the `type` parameter on your named channel to `file`.

```
agent.channels.c1.type=file
```

This defines a file channel named `c1` for the agent named `agent`.

Here is a table of configuration parameters you can adjust from the default values:

Key	Required	Type	Default
`type`	Yes	`String`	`file`
`checkpointDir`	No	`String`	`~/.flume/file-channel/checkpoint`
`useDualCheckpoints`	No	`boolean`	`false`
`backupCheckpointDir`	No	`String`	No default, but must be different from `checkpointDir`
`dataDirs`	No	`String` (comma separated list)	`~/.flume/file-channel/data`
`capacity`	No	`int`	`1000000`
`keep-alive`	No	`int`	3 (seconds)
`transactionCapacity`	No	`int`	`10000`

Key	Required	Type	Default
checkpointInterval	No	long	30000 (milliseconds)
maxFileSize	No	long	2146435071 (bytes)
minimumRequiredSpace	No	long	524288000 (bytes)

To specify the location where the Flume agent should hold data, set the `checkpointDir` and `dataDirs` properties:

```
agent.channels.c1.checkpointDir=/flume/c1/checkpoint
agent.channels.c1.dataDirs=/flume/c1/data
```

Technically, these properties are not required and have sensible default values for development. However, if you have more than one file channel configured in your agent, only the first channel will start. For production deployments and development work with multiple file channels, you should use distinct directory paths for each file channel storage area and consider placing different channels on different disks to avoid IO contention. Additionally, if you are sizing a large machine, consider using some form of RAID that contains striping (RAID 10, 50, or 60) to achieve higher disk performance rather than buying more expensive 10K or 15K drives or SSDs. If you don't have RAID striping but have multiple disks, set `dataDirs` to a comma-separated list of each storage location. Using multiple disks will spread the disk traffic almost as well as striped RAID but without the computational overhead associated with RAID 50/60 as well as the 50 percent space waste associated with RAID 10. You'll want to test your system to see whether the RAID overhead is worth the speed difference. As hard drive failures are a reality, you might prefer certain RAID configurations to single disks in order to protect yourself from the data loss associated with single drive failures. RAID 6 across the maximum number of disks can provide the highest performance for the minimal amount of data protection when combined with redundant and reliable power sources (such as an uninterruptable power supply or UPS).

Using the JDBC channel is a bad idea as it would introduce a bottleneck and single point of failure in what should be designed as a highly distributed system. NFS storage should be avoided for the same reason.

> Be sure to set `HADOOP_PREFIX` and `JAVA_HOME` environment variables when using the file channel. While we seemingly haven't used anything Hadoop-specific (such as writing to HDFS), the file channel uses Hadoop `Writables` as an on-disk serialization format. If Flume can't find the Hadoop libraries, you might see this in your startup, so check your environment variables:
>
> `java.lang.NoClassDefFoundError: org/apache/hadoop/io/Writable`

Starting with Flume 1.4, the file channel supports a secondary checkpoint directory. In situations where a failure occurs while writing the checkpoint data, that information could become unusable and a full replay of the logs is necessary in order to recover the state of the channel. As only one checkpoint is updated at a time before flipping to the other, one should always be in a consistent state, thus shortening restart times. Without valid checkpoint information, the Flume agent can't know what has been sent and what has not been sent in `dataDirs`. As files in the data directories might contain large amounts of data already sent but not yet deleted, a lack of checkpoint information would result in a large number of records being resent as duplicates.

Incoming data from sources is written and acknowledged at the end of the most current file in the data directory. The files in the checkpoint directory keep track of this data when it's taken by a sink.

If you want to use this feature, set the `useDualCheckpoints` property to `true` and specify a location for that second checkpoint directory with the `backupCheckpointDir` property. For performance reasons, it is always preferred that this be on a different disk from the other directories used by the file channel:

```
agent.channels.c1.useDualCheckpoints=true
agent.channels.c1.backupCheckpointDir=/flume/c1/checkpoint2
```

The default file channel `capacity` is one million events regardless of the size of the event contents. If the channel capacity is reached, a source will no longer be able to ingest the data. This default should be fine for low volume cases. You'll want to size this higher if your ingestion is so heavy that you can't tolerate normal planned or unplanned outages. For instance, there are many configuration changes you can make in Hadoop that require a cluster restart. If you have Flume writing important data into Hadoop, the file channel should be sized to tolerate the time it takes to restart Hadoop (and maybe add a comfort buffer for the unexpected). If your cluster or other systems are unreliable, you can set this higher still to handle even larger amounts of downtime. At some point, you'll run into the fact that your disk space is a finite resource, so you will have to pick some upper limit (or buy bigger disks).

The `keep-alive` parameter is similar to memory channels. It is the maximum time the source will wait when trying to write into a full channel before giving up. If space becomes available before the timeout, the write is successful; otherwise, `ChannelException` is thrown back to the source.

The `transactionCapacity` property is the maximum number of events allowed in a single transaction. This might become important for certain sources that batch together events and pass them to the channel in a single call. Most likely, you won't need to change this from the default. Setting this higher allocates additional resources internally, so you shouldn't increase it unless you run into performance issues.

The `checkpointInterval` property is the number of milliseconds between performing a checkpoint (which also rolls the log files written to `logDirs`). If you do not set this, 30 seconds will be used.

Checkpoint files also roll based on the volume of data written to them using the `maxFileSize` property. You can lower this value for low traffic channels if you want to try and save some disk space. Let's say your maximum file size is 50,000 bytes but your channel only writes 500 bytes a day; it would take 100 days to fill a single log. Let's say that you were on day 100 and 2000 bytes came in all at once. Some data would be written to the old file and a new file would be started with the overflow. After the roll, Flume tries to remove any log files that aren't needed anymore. As the full log has unprocessed records, it cannot be removed yet. The next chance to clean up that old log file might not come for another 100 days. It probably doesn't matter if that old 50,000 byte file sticks around longer, but as the default is around 2 GB, you could have twice that (4 GB) disk space used per channel. Depending on how much disk you have available and the number of channels configured in your agent, this might or might not be a problem. If your machines have plenty of storage space, the default should be fine.

Finally, the `minimumRequiredSpace` property is the amount of space you do *not* want to use for writing logs. The default configuration will throw an exception if you attempt to use the last 500 MB of the disk associated with the `dataDir` path. This limit applies across all channels, so if you have three file channels configured, the upper limit is still 500 MB and not 1.5 GB. You can set this value as low as 1 MB, but generally speaking, bad things tend to happen when you push disk utilization towards 100 percent.

Spillable Memory Channel

Introduced in Flume 1.5, the Spillable Memory Channel is a channel that acts like a memory channel until it is full. At that point, it acts like a file channel that is configured with a much larger capacity than its memory counterpart but runs at the speed of your disks (which means orders of magnitude slower).

> The Spillable Memory Channel is still considered experimental. Use it at your own risk!

I have mixed feelings about this new channel type. On the surface, it seems like a good idea, but in practice, I can see problems. Specifically, having a variable channel speed that changes depending on how downstream entities in your data pipe behave makes for difficult capacity planning. As a memory channel is used under good conditions, this implies that the data contained in it can be lost. So why would I go through extra trouble to save some of it to the disk? The data is either very important for me to spool it to disk with a file-backed channel, or it's less important and can be lost, so I can get away with less hardware and use a faster memory-backed channel. If I really need memory speed but with the capacity of a hard drive, **Solid State Drive** (**SSD**) prices have come down enough in recent years for a file channel on SSD to now be a viable option for you rather than using this hybrid channel type. I do not use this channel myself for these reasons.

To use this channel configuration, set the `type` parameter on your named channel to `spillablememory`:

```
agent.channels.c1.type=spillablememory
```

This defines a Spillable Memory Channel named `c1` for the agent named `agent`.

Here is a table of configuration parameters you can adjust from the default values:

Key	Required	Type	Default
type	Yes	String	spillablememory
memoryCapacity	No	int	10000
overflowCapacity	No	int	100000000
overflowTimeout	No	int	3 (seconds)
overflowDeactivationThreshold	No	int	5 (percent)
byteCapacityBufferPercentage	No	int	20 (percent)
byteCapacity	No	long (bytes)	80% of JVM Heap
checkpointDir	No	String	~/.flume/ file-channel/ checkpoint
dataDirs	No	String (comma separated list)	~/.flume/file-channel/data
useDualCheckpoints	No	boolean	false
backupCheckpointDir	No	String	No default, but must be different than checkpointDir
transactionCapacity	No	int	10000

Key	Required	Type	Default
`checkpointInterval`	No	`long`	`30000` (milliseconds)
`maxFileSize`	No	`long`	`2146435071` (bytes)
`minimumRequiredSpace`	No	`long`	`524288000` (bytes)

As you can see, many of the fields match against the memory channel and file channel's properties, so there should be no surprises here. Let's start with the memory side.

The `memoryCapacity` property determines the maximum number of events held in the memory (this was just called `capacity` for the memory channel but was renamed here to avoid ambiguity). Also, the default value, if unspecified, is 10,000 records instead of 100. If you wanted to double the default capacity, the configuration might look something like this:

```
agent.channels.c1.memoryCapacity=20000
```

As mentioned previously, you will most likely need to increase the Java heap space allocated using the `-Xmx` and `-Xms` parameters. Like most Java programs, more memory usually helps the garbage collector run more efficiently, especially as an application such as Flume generates a lot of short-lived objects. Be sure to do your research and pick an appropriate JVM garbage collector based on your available hardware.

> You can set `memoryCapacity` to zero, which effectively turns it into a file channel. Don't do this. Just use the file channel and be done with it.

The `overflowCapacity` property determines the maximum number of events that can be written to the disk before an error is thrown back to the source feeding it. The default value is a generous 100,000,000 events (far larger than the file channel's default of 100,000). Most servers nowadays have multiterabyte disks, so space should not be a problem, but do the math against your average event size to be sure you don't fill your disks by accident. If your channels are filling this much, you are probably dealing with another issue downstream and the last thing you need is your data buffer layer filling up completely. For example, if you had a large 1 megabyte event payload, 100 million of these add up to 100 terabytes, which is probably bigger than the disk space on an average server. A 1 kilobyte payload would only take 100 gigabytes, which is probably fine. Just do the math ahead of time so you are not surprised.

> You can set `overflowCapacity` to zero, which effectively turns it into a memory channel. Don't do this. Just use the memory channel and be done with it.

The `transactionCapacity` property adjusts the batch size of events written to a channel in a single transaction. If this is set too low, it will lower the throughput of the agent on high volume flows because of the transaction overhead. For a high volume channel, you will probably need to set this higher, but the only way to be sure is to test your particular workflow. See *Chapter 8*, *Monitoring Flume*, to learn how to accomplish this.

Finally, the `byteCapacityBufferPercentage` and `byteCapacity` parameters are identical in functionality and defaults to the memory channel, so I won't waste your time repeating it here.

What is important is the `overflowTimeout` property. This is the number of seconds after which the memory part of the channel fills before data starts getting written to the disk-backed portion of the channel. If you want writes to start occurring immediately, you can set this to zero. You might wonder why you need to wait before starting to write to the disk portion of the channel. This is where the undocumented `overflowDeactivationThreshold` property comes into play. This is the amount of time that space has to be available in the memory path before it can switch back from disk writing. I believe this is an attempt to prevent flapping back and forth between the two. Of course, there really are no ordering guarantees in Flume, so I don't know why you would choose to append to the disk buffer if a spot is available in faster memory. Perhaps they are trying to avoid some kind of starvation condition, although the code appears to attempt to remove events in the order of arrival even when using both memory and disk queues. Perhaps it will be explained to us should it ever come out of experimental status.

> The `overflowDeactivationThreshold` property is stated to be for internal use only, so adjust it at your own peril. If you are considering it, be sure to get familiar with the source code so that you understand the implications of altering the default.

The rest of the properties on this channel are identical in name and functionality to its file channel counterpart, so please refer to the previous section.

Summary

In this chapter, we covered the two channel types you are most likely to use in your data processing pipelines.

The memory channel offers speed at the cost of data loss in the event of failure. Alternatively, the file channel provides a more reliable transport in that it can tolerate agent failures and restarts at a performance cost.

You will need to decide which channel is appropriate for your use cases. When trying to decide whether a memory channel is appropriate, ask yourself what the monetary cost is if you lose some data. Weigh that against the additional costs of more hardware to cover the difference in performance when deciding if you need a durable channel after all. Another consideration is whether or not the data can be resent. Not all data you might ingest into Hadoop will come from streaming application logs. If you receive "daily downloads" of data, you can get away with using a memory channel because if you encounter a problem, you can always rerun the import.

Finally, we covered the experimental Spillable Memory Channel. Personally, I think its creation is a bad idea, but like most things in computer science, everybody has an opinion on what is good or bad. I feel that the added complexity and nondeterministic performance make for difficult capacity planning, as you should always size things for the worst-case scenario. If your data is critical enough for you to overflow to the disk rather than discard the events, then you aren't going to be okay with losing even the small amount held in memory.

In the next chapter, we'll look at sinks, specifically, the HDFS sink to write events to HDFS, the Elastic Search sink to write events to Elastic Search, and the Morphline Solr sink to write events to Solr. We will also cover Event Serializers, which specify how Flume events are translated into output that's more suitable for the sink. Finally, we will cover sink processors and how to set up load balancing and failure paths in a tiered configuration for more robust data transport.

4
Sinks and Sink Processors

By now, you should have a pretty good idea where the sink fits into the Flume architecture. In this chapter, we will first learn about the most-used sink with Hadoop, the HDFS sink. We will then cover two of the newer sinks that support common **Near Real Time** (**NRT**) log processing: the ElasticSearchSink and the MorphlineSolrSink. As you'd expect, the first writes data into Elasticsearch and the latter to Solr. The general architecture of Flume supports many other sinks we won't have space to cover in this book. Some come bundled with Flume and can write to **HBase**, **IRC**, and, as we saw in *Chapter 2*, *A Quick Start Guide to Flume*, a log4j and file sink. Other sinks are available on the Internet and can be used to write data to **MongoDB**, **Cassandra**, **RabbitMQ**, **Redis**, and just about any other data store you can think of. If you can't find a sink that suits your needs, you can write one easily by extending the `org.apache.flume.sink.AbstractSink` class.

HDFS sink

The job of the HDFS sink is to continuously open a file in HDFS, stream data into it, and at some point, close that file and start a new one. As we discussed in *Chapter 1*, *Overview and Architecture*, the time between files rotations must be balanced with how quickly files are closed in HDFS, thus making the data visible for processing. As we've discussed, having lots of tiny files for input will make your MapReduce jobs inefficient.

To use the HDFS sink, set the `type` parameter on your named sink to `hdfs`.

```
agent.sinks.k1.type=hdfs
```

This defines a HDFS sink named `k1` for the agent named `agent`. There are some additional parameters you must specify, starting with the path in HDFS you want to write the data to:

```
agent.sinks.k1.hdfs.path=/path/in/hdfs
```

This HDFS path, like most file paths in Hadoop, can be specified in three different ways: absolute, absolute with server name, and relative. These are all equivalent (assuming your Flume agent is run as the `flume` user):

absolute	`/Users/flume/mydata`
absolute with server	`hdfs://namenode/Users/flume/mydata`
relative	`mydata`

I prefer to configure any server I'm installing Flume on with a working `hadoop` command line by setting the `fs.default.name` property in Hadoop's `core-site.xml` file. I don't keep persistent data in HDFS user directories but prefer to use absolute paths with some meaningful path name (for example, `/logs/apache/access`). The only time I would specify a NameNode specifically is if the target was a different Hadoop cluster entirely. This allows you to move configurations you've already tested in one environment into another without unintended consequences such as your production server writing data to your staging Hadoop cluster because somebody forgot to edit the target in the configuration. I consider externalizing environment specifics a good best practice to avoid situations such as these.

One final required parameter for the HDFS sink, actually *any* sink, is the channel that it will be doing `take` operations from. For this, set the channel parameter with the channel name to read from:

```
agent.sinks.k1.channel=c1
```

This tells the `k1` sink to read events from the `c1` channel.

Here is a mostly complete table of configuration parameters you can adjust from the default values:

Key	Required	Type	Default
`type`	Yes	`String`	`hdfs`
`channel`	Yes	`String`	
`hdfs.path`	Yes	`String`	
`hdfs.filePrefix`	No	`String`	`FlumeData`
`hdfs.fileSuffix`	No	`String`	
`hdfs.minBlockReplicas`	No	`int`	See the `dfs.replication` property in your inherited Hadoop configuration, usually, 3.
`hdfs.maxOpenFiles`	No	`long`	`5000`

Key	Required	Type	Default
`hdfs.closeTries`	No	`int`	`0` (0=try forever, otherwise a count)
`hdfs.retryInterval`	No	`int`	`180 Seconds` (0=don't retry)
`hdfs.round`	No	`boolean`	`false`
`hdfs.roundValue`	No	`int`	`1`
`hdfs.roundUnit`	No	`String (second, minute or hour)`	`second`
`hdfs.timeZone`	No	`String`	Local time
`hdfs.useLocalTimeStamp`	No	`boolean`	`False`
`hdfs.inUsePrefix`	No	`String`	Blank
`hdfs.inUseSuffix`	No	`String`	`.tmp`
`hdfs.rollInterval`	No	`long (seconds)`	`30 Seconds` (0=disable)
`hdfs.rollSize`	No	`long (bytes)`	`1024 bytes` (0=disable)
`hdfs.rollCount`	No	`long`	`10` (0=disable)
`hdfs.batchSize`	No	`long`	`100`
`hdfs.codeC`	No	`String`	

Remember to always check the Flume User Guide for the version you are using at `http://flume.apache.org/`, as things might change between the release of this book and the version you are actually using.

Path and filename

Each time Flume starts a new file at `hdfs.path` in HDFS to write data into, the filename is composed of the `hdfs.filePrefix`, a period character, the epoch timestamp at which the file was started, and optionally, a file suffix specified by the `hdfs.fileSuffix` property (if set), for example:

```
agent.sinks.k1.hdfs.path=/logs/apache/access
```

The preceding command would result in a file such as `/logs/apache/access/FlumeData.1362945258`

However, in the following configuration, your filenames would be more like `/logs/apache/access/access.1362945258.log`:

```
agent.sinks.k1.hdfs.path=/logs/apache/access
agent.sinks.k1.hdfs.filePrefix=access
agent.sinks.k1.hdfs.fileSuffix=.log
```

Over time, the `hdfs.path` directory will get very full, so you will want to add some kind of time element into the path to partition the files into subdirectories. Flume supports various time-based escape sequences, such as `%Y` to specify a four-digit year. I like to use sequences in the year/month/day/hour form (so that they are sorted oldest to newest), so I often use this for a path:

```
agent.sinks.k1.hdfs.path=/logs/apache/access/%Y/%m/%d/%H
```

This says I want a path like `/logs/apache/access/2013/03/10/18/`.

> For a complete list of time-based escape sequences, see the Flume User Guide.

Another handy escape sequence mechanism is the ability to use Flume header values in your path. For instance, if there was a header with a key of `logType`, I could split Apache access and error logs into different directories while using the same channel, by escaping the header's key as follows:

```
agent.sinks.k1.hdfs.path=/logs/apache/%{logType}/%Y/%m/%d/%H
```

The preceding line of code would result in `access` logs going to `/logs/apache/access/2013/03/10/18/`, and `error` logs going to `/logs/apache/error/2013/03/10/18/`. However, if I preferred both log types in the same directory path, I could have used `logType` in my `hdfs.filePrefix` instead, as follows:

```
agent.sinks.k1.hdfs.path=/logs/apache/%Y/%m/%d/%H
agent.sinks.k1.hdfs.filePrefix=%{logType}
```

Obviously, it is possible for Flume to write to multiple files at once. The `hdfs.maxOpenFiles` property sets the upper limit for how many can be open at once, with a default of 5000. If you should exceed this limit, the oldest file that's still open is closed. Remember that every open file incurs overhead both at the OS level and in HDFS (NameNode and DataNode connections).

Another set of properties you might find useful allow for rounding down event times at an hour, minute, or second granularity while still maintaining these elements in file paths. Let's say you had a path specification as follows:

```
agent.sinks.k1.hdfs.path=/logs/apache/%Y/%m/%d/%H%M
```

However, if you wanted only four subdirectories per day (at 00, 15, 30, and 45 past the hour, each containing 15 minutes of data), you could accomplish this by setting the following:

```
agent.sinks.k1.hdfs.round=true
agent.sinks.k1.hdfs.roundValue=15
agent.sinks.k1.hdfs.roundUnit=minute
```

This would result in logs between 01:15:00 and 01:29:59 on March 10, 2013 being written to files contained in `/logs/apache/2013/03/10/0115/`. Logs from 01:30:00 to 01:44:59 would be written in files contained in `/logs/apache/2013/03/10/0130/`.

The `hdfs.timeZone` property is used to specify the time zone that you want time interpreted for your escape sequences. The default is your computer's local time. If your local time is affected by daylight savings time adjustments, you will have twice as much data when `%H == 02` (in the fall) and no data when `%H == 02` (in the spring). I think it is a bad idea to introduce time zones into things that are meant for computers to read. I believe time zones are a concern for humans alone and computers should only converse in universal time. For this reason, I set this property on my Flume agents to make the time zone issue just go away:

```
-Duser.timezone=UTC
```

If you don't agree, you are free to use the default (local time) or set `hdfs.timeZone` to whatever you like. The value you passed is used in a call to `java.util.Timezone.getTimeZone(...)`, so check the Javadocs for acceptable values to be used here.

The other time-related property is the `hdfs.useLocalTimeStamp boolean` property. By default, its value is `false`, which tells the sink to use the event's `timestamp` header when calculating date-based escape sequences in file paths, as shown previously. If you set the property to `true`, the current system time will be used instead, effectively telling Flume to use the transport arrival time rather than the original event time. You would not set this in cases where HDFS was the final target for the streamed events. This way, delayed events will still be placed correctly (where users would normally look for them) regardless of their arrival time. However, there may be a use case where events are temporarily written to Hadoop and processed in batches, on some interval (perhaps daily). In this case, the transport time would be preferred, so your postprocessing job doesn't need to scan older folders for delayed data.

Remember that files in HDFS are broken into file blocks that are replicated across the DataNodes. The default number of replicas is usually three (as set in the Hadoop base configuration). You can override this value up or down for this sink with the `hdfs.minBlockReplicas` property. For example, if I have a data stream that I feel only needs two replicas instead of three, I can override this as follows:

```
agent.skinks.k1.hdfs.minBlockReplicas=2
```

> Don't set the minimum replica count higher than the number of data nodes you have, otherwise you'll create a degraded state HDFS. You also don't want to set it so high that a downed box for maintenance would trigger this situation. Personally, I've never set this higher than the default of three, but I have set it lower on less important data in order to save space.

Finally, while files are being written to HDFS, a `.tmp` extension is added. When the file is closed, the extension is removed. You can change the extension used by setting the `hdfs.inUseSuffix` property, but I've never had a reason to do so:

```
agent.sinks.k1.hdfs.inUseSuffix=flumeiswriting
```

This allows you to see which files are being written to simply by looking at a directory listing in HDFS. As you typically specify a directory for input in your MapReduce job (or because you are using Hive), the temporary files will often be picked up as empty or garbled input by mistake. To avoid having your temporary files picked up before being closed, set the prefix to either a dot or an underscore character as follows:

```
agent.sinks.k1.hdfs.inUsePrefix=_
```

That said, there are occasions where files were not closed properly due to some HDFS glitch, so you might see files with the in-use prefix/suffix that haven't been used in some time. A few new properties were added in Version 1.5 to change the default behavior of closing files. The first is the `hdfs.closeTries` property. The default of zero actually means "try forever", so it is a little confusing. Setting it to 4 means try 4 times before giving up. You can adjust the interval between retries by setting the `hdfs.retryInterval` property. Setting it too low could swamp your NameNode with too many requests, so be careful if you lower this from the default of 3 minutes. Of course, if you are opening files too quickly, you might need to lower this just to keep from going over the `hdfs.maxOpenFiles` setting which was covered previously. If you actually didn't want any retries, you can set `hdfs.retryInterval` to zero seconds (again, not to be confused with `closeTries=0`, which means try forever). Hopefully in a future version, they will use the more commonly used convention of a negative number (usually, `-1`) when infinite is desired.

File rotation

By default, Flume will rotate actively written-to files every 30 seconds, 10 events, or 1024 bytes. This is done by setting the `hdfs.rollInterval`, `hdfs.rollCount`, and `hdfs.rollSize` properties, respectively. One or more of these can be set to zero to disable this particular rolling mechanism. For instance, if you only wanted a time-based roll of 1 minute, you would set the following:

```
agent.sinks.k1.hdfs.rollInterval=60
agent.sinks.k1.hdfs.rollCount=0
agent.sinks.k1.hdfs.rollSize=0
```

If your output contains any amount of header information, the HDFS size per file can be larger than what you expect, because the `hdfs.rollSize` rotation scheme only counts the event body length. Clearly, you might not want to disable all three mechanisms for rotation at the same time, or you will have one directory in HDFS overflowing with files.

Finally, a related parameter is `hdfs.batchSize`. This is the number of events that the sink will read per transaction from the channel. If you have a large volume of data in your channel, you might see a performance increase by setting this higher than the default of 100, which decreases the transaction overhead per event.

Now that we've discussed the way files are managed and rolled in HDFS, let's look into how the event contents get written.

Compression codecs

Codecs (Coder/Decoders) are used to compress and decompress data using various compression algorithms. Flume supports `gzip`, `bzip2`, `lzo`, and `snappy`, although you might have to install `lzo` yourself, especially if you are using a distribution such as CDH, due to licensing issues.

If you want to specify compression for your data, set the `hdfs.codeC` property if you want the HDFS sink to write compressed files. The property is also used as the file suffix for the files written to HDFS. For example, if you specify the following, all files that are written will have a `.gzip` extension, so you don't need to specify the `hdfs.fileSuffix` property in this case:

```
agent.sinks.k1.hdfs.codeC=gzip
```

The codec you choose to use will require some research on your part. There are arguments for using `gzip` or `bzip2` for their higher compression ratios at the cost of longer compression times, especially if your data is written once but will be read hundreds or thousands of times. On the other hand, using snappy or `lzo` results in faster compression performance but results in a lower compression ratio. Keep in mind that the splitability of the file, especially if you are using plain text files, will greatly affect the performance of your MapReduce jobs. Go pick up a copy of *Hadoop Beginner's Guide*, *Garry Turkington*, *Packt Publishing* (http://amzn.to/14Dh6TA) or *Hadoop: The Definitive Guide*, *Tom White*, *O'Reilly* (http://amzn.to/16OsfIf) if you aren't sure what I'm talking about.

Event Serializers

An Event Serializer is the mechanism by which a `FlumeEvent` is converted into another format for output. It is similar in function to the `Layout` class in log4j. By default, the `text` serializer, which outputs just the Flume event body, is used. There is another serializer, `header_and_text`, which outputs both the headers and the body. Finally, there is an `avro_event` serializer that can be used to create an Avro representation of the event. If you write your own, you'd use the implementation's fully qualified class name as the `serializer` property value.

Text output

As mentioned previously, the default serializer is the `text` serializer. This will output only the Flume event body, with the headers discarded. Each event has a newline character appender unless you override this default behavior by setting the `serializer.appendNewLine` property to `false`.

Key	Required	Type	Default
`Serializer`	No	`String`	`text`
`serializer.appendNewLine`	No	`boolean`	`true`

Text with headers

The `text_with_headers` serializer allows you to save the Flume event headers rather than discard them. The output format consists of the headers, followed by a space, then the body payload, and finally, terminated by an optionally disabled newline character, for instance:

```
{key1=value1, key2=value2} body text here
```

Key	Required	Type	Default
`serializer`	No	`String`	`text_with_headers`
`serializer.appendNewLine`	No	`boolean`	`true`

Apache Avro

The Apache Avro project (`http://avro.apache.org/`) provides a serialization format that is similar in functionality to Google Protocol Buffers but is more Hadoop friendly as the container is based on Hadoop's `SequenceFile` and has some MapReduce integration. The format is also self-describing using JSON, making for a good long-term data storage format, as your data format might evolve over time. If your data has a lot of structure and you want to avoid turning it into Strings only to then parse them in your MapReduce job, you should read more about Avro to see whether you want to use it as a storage format in HDFS.

The `avro_event` serializer creates Avro data based on the Flume event schema. It has no formatting parameters as Avro dictates the format of the data, and the structure of the Flume event dictates the schema used:

Key	Required	Type	Default
`serializer`	No	`String`	`avro_event`
`serializer.compressionCodec`	No	`String` (`gzip`, `bzip2`, `lzo`, or `snappy`)	
`serializer.syncIntervalBytes`	No	`int` (bytes)	`2048000` (bytes)

If you want your data compressed before being written to the Avro container, you should set the `serializer.compressionCodec` property to the file extension of an installed codec. The `serializer.syncIntervalBytes` property determines the size of the data buffer used before flushing the data to HDFS, and therefore, this setting can affect your compression ratio when using a codec. Here is an example using `snappy` compression on Avro data using a 4 MB buffer:

```
agent.sinks.k1.serializer=avro_event
agent.sinks.k1.serializer.compressionCodec=snappy
agent.sinks.k1.serializer.syncIntervalBytes=4194304
agent.sinks.k1.hdfs.fileSuffix=.avro
```

For Avro files to work in an Avro MapReduce job, they *must* end in `.avro` or they will be ignored as input. For this reason, you need to explicitly set the `hdfs.fileSuffix` property. Furthermore, you would *not* set the `hdfs.codeC` property on an Avro file.

User-provided Avro schema

If you want to use a different schema from the Flume event schema used with the `avro_event` type, starting in Version 1.4, the closely named `AvroEventSerializer` will let you do this. Keep in mind that using this implementation only, the event's body is serialized and headers are not passed on.

Set the serializer type to the fully qualified `org.apache.flume.sink.hdfs.AvroEventSerializer` class name:

```
agent.sinks.k1.serializer=org.apache.flume.sink.hdfs.
AvroEventSerializer
```

Unlike the other serializers that take additional parameters in the Flume configuration file, this one requires that you pass the schema information via a Flume header. This is a byproduct of one of the Avro-aware sources we'll see in *Chapter 6, Interceptors, ETL, and Routing*, where schema information is sent from the source to the final destination via the event header. You can fake this if you are using a source that doesn't set these by using a static header interceptor. We'll talk more about interceptors in *Chapter 6, Interceptors, ETL, and Routing*, so flip back to this part later on.

To specify the schema directly in the Flume configuration file, use the `flume.avro.schema.literal` header as shown in this example (using a map of strings schema):

```
agent.sinks.k1.serializer=org.apache.flume.sink.hdfs.
AvroEventSerializer
agent.sinks.k1.interceptors=i1
agent.sinks.k1.interceptors.i1.type=static
agent.sinks.k1.interceptors.i1.key=flume.avro.schema.literal
agent.sinks.k1.interceptors.i1.value="{\"type\":\"map\",\"values\":\"
string\"}"
```

If you prefer to put the schema file in HDFS, use the `flume.avro.schema.url` header instead, as shown in this example:

```
agent.sinks.k1.serializer=org.apache.flume.sink.hdfs.
AvroEventSerializer
agent.sinks.k1.interceptors=i1
agent.sinks.k1.interceptors.i1.type=static
agent.sinks.k1.interceptors.i1.key=flume.avro.schema.url
agent.sinks.k1.interceptors.i1.value=hdfs://path/to/schema.avsc
```

Actually, in this second form, you can pass any URL including a `file://` URL, but this would indicate a file local to where you are running the Flume agent, which might create additional setup work for your administrators. This is also true of configuration served up by a HTTP web server or farm. Rather than creating additional setup dependencies, just use the dependency you cannot remove, which is HDFS, using a `hdfs://` URL.

Be sure to only set either the `flume.avro.schema.literal` header or the `flume.avro.schema.url` header both not both.

File type

By default, the HDFS sink writes data to HDFS as Hadoop's `SequenceFile`. This is a common Hadoop wrapper that consists of a key and value field separated by binary field and record delimiters. Usually, text files on a computer make assumptions like a newline character terminates each record. So, what do you do if your data contains a newline character, such as some XML? Using a sequence file can solve this problem because it uses nonprintable characters for delimiters. Sequence files are also splittable, which makes for better locality and parallelism when running MapReduce jobs on your data, especially on large files.

SequenceFile

When using a `SequenceFile` file type, you need to specify how you want the key and value to be written on the record in the `SequenceFile`. The key on each record will always be a `LongWritable` type and will contain the current timestamp, or if the `timestamp` event header is set, it will be used instead. By default, the format of the value is a `org.apache.hadoop.io.BytesWritable` type, which corresponds to the `byte[]` Flume body:

Key	Required	Type	Default
hdfs.fileType	No	String	SequenceFile
hdfs.writeFormat	No	String	writable

However, if you want the payload interpreted as a String, you can override the `hdfs.writeFormat` property, so `org.apache.hadoop.io.Text` will be used as the value field:

Key	Required	Type	Default
hdfs.fileType	No	String	SequenceFile
hdfs.writeFormat	No	String	text

DataStream

If you do not want to output a `SequenceFile` file because your data doesn't have a natural key, you can use a DataStream to output only the uncompressed value. Simply override the `hdfs.fileType` property:

```
agent.sinks.k1.hdfs.fileType=DataStream
```

This is the file type you would use with Avro serialization, as any compression should have been done in the Event Serializer. To serialize gzip-compressed Avro files, you would set these properties:

```
agent.sinks.k1.serializer=avro_event
agent.sinks.k1.serializer.compressionCodec=gzip
agent.sinks.k1.hdfs.fileType=DataStream
agent.sinks.k1.hdfs.fileSuffix=.avro
```

CompressedStream

`CompressedStream` is similar to a DataStream, except that the data is compressed when it's written. You can think of this as running the gzip utility on an uncompressed file, but all in one step. This differs from a compressed Avro file whose contents are compressed *and then* written into an uncompressed Avro wrapper:

```
agent.sinks.k1.hdfs.fileType=CompressedStream
```

Remember that only certain compressed formats are splittable in MapReduce should you decide to use `CompressedStream`. The compression algorithm selection doesn't have a Flume configuration but is dictated by the `zlib.compress.strategy` and `zlib.compress.level` properties in core Hadoop instead.

Timeouts and workers

Finally, there are two miscellaneous properties related to timeouts and two for worker pools that you can change:

Key	Required	Type	Default
`hdfs.callTimeout`	No	`long` (milliseconds)	10000
`hdfs.idleTimeout`	No	`int` (seconds)	0 (0=disable)
`hdfs.threadsPoolSize`	No	`int`	10
`hdfs.rollTimerPoolSize`	No	`int`	1

The `hdfs.callTimeout` property is the amount of time the HDFS sink will wait for HDFS operations to return a success (or failure) before giving up. If your Hadoop cluster is particularly slow (for instance, a development or virtual cluster), you might need to set this value higher in order to avoid errors. Keep in mind that your channel will overflow if you cannot sustain higher write throughput than the input rate of your channel.

The `hdfs.idleTimeout` property, if set to a nonzero value, is the time Flume will wait to automatically close an idle file. I have never used this as `hdfs.fileRollInterval` handles the closing of files for each roll period, and if the channel is idle, it will not open a new file. This setting seems to have been created as an alternative roll mechanism to the size, time, and event count mechanisms that have already been discussed. You might want as much data written to a file as possible and only close it when there really is no more data. In this case, you can use `hdfs.idleTimeout` to accomplish this rotation scheme if you also set `hdfs.rollInterval`, `hdfs.rollSize`, and `hdfs.rollCount` to zero.

The first property you can set to adjust the number of workers is `hdfs.threadsPoolSize` and it defaults to 10. This is the maximum number of files that can be written to at the same time. If you are using event headers to determine file paths and names, you might have more than 10 files open at once, but be careful when increasing this value too much so as not to overwhelm HDFS.

The last property related to worker pools is the `hdfs.rollTimerPoolSize`. This is the number of workers processing timeouts set by the `hdfs.idleTimeout` property. The amount of work to close the files is pretty small, so increasing this value from the default of one worker is unlikely. If you do not use a rotation based on `hdfs.idleTimeout`, you can ignore the `hdfs.rollTimerPoolSize` property, as it is not used.

Sink groups

In order to remove single points of failures in your data processing pipeline, Flume has the ability to send events to different sinks using either load balancing or failover. In order to do this, we need to introduce a new concept called a **sink group**. A sink group is used to create a logical grouping of sinks. The behavior of this grouping is dictated by something called the **sink processor**, which determines how events are routed.

There is a default sink processor that contains a single sink which is used whenever you have a sink that isn't part of any sink group. Our `Hello, World!` example in *Chapter 2*, *A Quick Start Guide to Flume*, used the default sink processor. No special configuration is required for single sinks.

In order for Flume to know about the sink groups, there is a new top-level agent property called `sinkgroups`. Similar to sources, channels, and sinks, you prefix the property with the agent name:

```
agent.sinkgroups=sg1
```

Here, we have defined a sink group called `sg1` for the agent named `agent`.

For each named sink group, you need to specify the sinks it contains using the `sinks` property consisting of a space-delimited list of sink names:

```
agent.sinkgroups.sg1.sinks=k1 k2
```

This defines that the `k1` and `k2` sinks are part of the `sg1` sink group for the agent named `agent`.

Often, sink groups are used in conjunction with the tiered movement of data to route around failures. However, they can also be used to write to different Hadoop clusters, as even a well-maintained cluster has periodic maintenance.

Load balancing

Continuing the preceding example, let's say you want to load balance traffic to `k1` and `k2` evenly. There are some additional properties you need to specify, as listed in this table:

Key	Type	Default
`processor.type`	`String`	`load_balance`
`processor.selector`	`String (round_robin, random)`	`round_robin`
`processor.backoff`	`boolean`	`false`

When you set `processor.type` to `load_balance`, round robin selection will be used, unless otherwise specified by the `processor.selector` property. This can be set to either `round_robin` or `random`. You can also specify your own load balancing selector mechanism, which we won't cover here. Consult the Flume documentation if you need this custom control.

The `processor.backoff` property specifies whether an exponential backup should be used when retrying a sink that threw an exception. The default is `false`, which means that after a thrown exception, the sink will be tried again the next time its turn is up based on round robin or random selection. If set to `true`, then the wait time for each failure is doubled, starting at 1 second up to a limit of around 18 hours (2^{16} seconds).

In an earlier version of Flume, the default in the code for `processor.backoff` was stated as `false`, but the documentation stated it as `true`. This error has been fixed, however, it may save you a headache by specifying what you want for property settings rather than relying on the defaults.

Failover

If you would rather try one sink and if that one fails to try another, then you want to set `processor.type` to `failover`. Next, you'll need to set additional properties to specify the order by setting the `processor.priority` property, followed by the sink name:

Key	Type	Default
`processor.type`	`String`	`failover`
`processor.priority.NAME`	`int`	
`processor.maxpenality`	`int (milliseconds)`	`30000`

Let's look at the following example:

```
agent.sinkgroups.sg1.sinks=k1 k2 k3
agent.sinkgroups.sg1.processor.type=failover
agent.sinkgroups.sg1.processor.priority.k1=10
agent.sinkgroups.sg1.processor.priority.k2=20
agent.sinkgroups.sg1.processor.priority.k3=20
```

Lower priority numbers come first, and in the case of a tie, order is arbitrary. You can use any numbering system that makes sense to you (by ones, fives, tens—whatever). In this example, the `k1` sink will be tried first, and if an exception is thrown, either `k2` or `k3` will be tried next. If `k3` was selected first for trial and it failed, `k2` will still be tried. If all sinks in the sink group fail, the transaction with the channel is rolled back.

Finally, `processor.maxPenality` sets an upper limit to an exponential backoff for failed sinks in the group. After the first failure, it will be 1 second before it can be used again. Each subsequent failure doubles the wait time until `processor.maxPenality` is reached.

MorphlineSolrSink

HDFS is not the only useful place to send your logs and data. Solr is a popular real-time search platform used to index large amounts of data, so full text searching can be performed almost instantaneously. Hadoop's horizontal scalability creates an interesting problem for Solr, as there is now more data than a single instance can handle. For this reason, a horizontally scalable version of Solr was created, called SolrCloud. Cloudera's Search product is also based on SolrCloud, so it should be no surprise that Flume developers created a new sink specifically to write streaming data into Solr.

Like most streaming data flows, you not only transport the data, but you also often reformat it into a form more consumable to the target of the flow. Typically, this is done in a Flume-only workflow by applying one or more interceptors just prior to the sink writing the data to the target system. This sink uses the Morphline engine to transform the data, instead of interceptors.

Internally, each Flume event is converted into a Morphline record and passed to the first command in the Morphline command chain. A record can be thought of as a set of key/value pairs with string keys and arbitrary object values. Each of the Flume headers is passed as a Record Field with the same header keys. A special Record Field key `_attachment_body` is used for the Flume event body. Keep in mind that the body is still a byte array (Java `byte[]`) at this point and must be specifically processed in the Morphline command chain.

Each command processes the record in turn, passing the output to the input of the next command in line with the final command responsible for terminating the flow. In many ways, it is similar in functionally to Flume's Interceptor functionality, which we'll see in *Chapter 6, Interceptors, ETL, and Routing*. In the case of writing to Solr, we use the `loadSolr` command to convert the Morphline record into a Solr Document and write to the Solr cluster. Here is what this simplified flow might look like in a picture form:
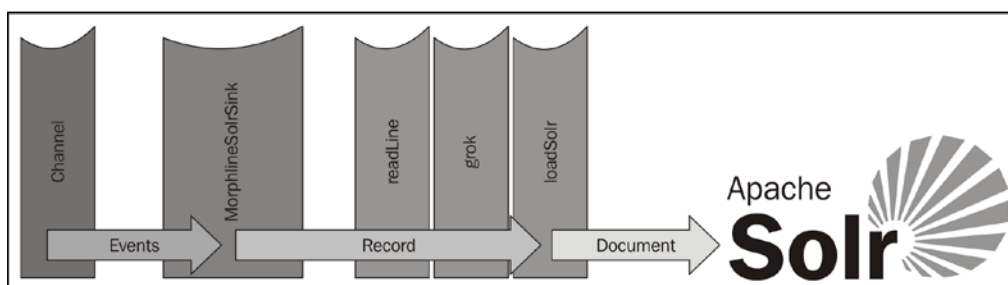
Morphline configuration files

Morphline configuration files use the HOCON format, which is similar to JSON but has a less strict syntax, making them less error-prone when used for configuration files over JSON.
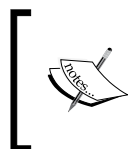
> HOCON is an acronym for Human Optimized Configuration Object Notation. You can read more about HOCON on this GitHub page: `https://github.com/typesafehub/config/blob/master/HOCON.md`

The configuration file contains a single key with the `morphlines` value. The value is an array of Morphline configurations. Each individual entry is comprised of three keys:

- `id`
- `importCommands`
- `commands`

If your configuration contains multiple Morphlines, the value of `id` must be provided to the Flume sink by way of the `morphlineId` property. The value of `importCommands` specifies the Java classes to import when the Morphine is evaluated. The double star indicates that all paths and classes from that point in the package hierarchy should be included. All classes that implement `com.cloudera.cdk.morphline.api.CommandBuilder` are interrogated for their names via the `getNames()` method. These names are the command names you use in the next section. Don't worry; you don't need to sift through the source code to find them, as they have a well-documented reference guide online. Finally, the `commands` key references a list of command dictionaries. Each command dictionary has a single key consisting of the name of the Morphline command followed by its specific properties.
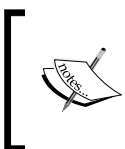
> For a list of Morphline commands and associated configuration properties, see the reference guide at `http://kitesdk.org/docs/current/kite-morphlines/morphlinesReferenceGuide.html`

Here is what a skeleton configuration file might look like:

```
morphlines : [
  {
    id : transform_my_data
    importCommands : [
      "com.cloudera.**",
      "org.apache.solr.**"
```

```
      ]
      commands : [
        {
          COMMAND_NAME1 : {
            property1 : value1
            property2 : value2
          }
        }
        { COMMAND_NAME2 : {
            property1 : value1
          }
        ]
      }
  ]
```

Typical SolrSink configuration

Here is the preceding skeleton configuration applied to our Solr use case. This is not meant to be complete, but it is sufficient to discuss the flow in the preceding diagram:

```
morphlines : [
  {
    id : solr_flow
    importCommands : [
      "com.cloudera.**",
      "org.apache.solr.**"
    ]
    commands : [
      {
        readLine : {
          charset : UTF-8
        }
      {
        grok : {
          GROK_PROPERTIES_HERE
        }
      }
      {
        loadSolr : {
          solrLocator : {
            collection : my_collection
            zkHost : "solr.example.com:2181/solr"
          }
        }
```

```
          }
      ]
    }
  ]
```

You can see the same boilerplate configuration where we define a single Morphline with the `solr_flow` identifier. The command sequence starts with the `readLine` command. This simply reads the event body from the `_attachment_body` field and converts `byte[]` to `String` using the configured encoding (in this case, UTF-8). The resulting `String` value is set to the field with the key `message`. The next command in the sequence, which is the `grok` command, uses regular expressions to extract additional fields to make a more interesting Solr Document. I couldn't possibly do this command justice by trying to explain everything you can do with it. For that, please see the KiteSDK documentation.
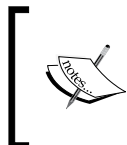
> See the reference guide for a complete list of Morphline commands, their properties, and usage information at `http://kitesdk.org/docs/current/kite-morphlines/morphlinesReferenceGuide.html`

Suffice to say, `grok` lets me take a webserver log line such as this:

```
10.4.240.176 - - [14/Mar/2014:12:02:17 -0500] "POST http://mysite.
com/do_stuff.php HTTP/1.1" 500 834
```

Then, it lets me turn it into more structured data like this:

```
{
  ip : 10.4.240.176
  timestamp : 1413306137
  method : POST
  url : http://mysite.com/do_stuff.php
  protocol : HTTP/1.1
  status_code : 500
  length : 834
}
```

If you wanted to search for all the times this page threw a 500 status code, having these fields broken out makes the task easy for Solr.

Finally, we call the `loadSolr` command to insert the record into our Solr cluster. The `solrLocator` property indicates the target Solr cluster (by way of its Zookeeper server(s)) and the data collection to write these documents into.

Sink configuration

Now that you have a basic idea of how to create a Morphline configuration file, let's apply this to the actual sink configuration.

The following table details the sink's parameters and default values:

Key	Required	Type	Default
type	Yes	String	org.apache.flume. sink.solr.morphline. MorphlineSolrSink
channel	Yes	String	
morphlineFile	Yes	String	
morphlineId	No	String	Required if the Morphline configuration file contains more than one Morphline.
batchSize	No	int	1000
batchDurationMillis	No	long	1000 (milliseconds)
handlerClass	No	String	org.apache.flume. sink.solr.morphline. MorphlineHandlerImpl

The MorphlineSolrSink does not have a short type alias, so set the type parameter on your named sink to org.apache.flume.sink.solr.morphline. MorphlineSolrSink:

```
agent.sinks.k1.type=org.apache.flume.sink.solr.morphline.
MorphlineSolrSink
```

This defines a MorphlineSolrSink named k1 for the agent named agent.

The next required parameter is the channel property. This specifies which channel to read events from for processing.

```
agent.sinks.k1.channel=c1
```

This tells the k1 sink to read events from the c1 channel.

The only other required parameter is the relative or absolute path to the Morphline configuration file. This cannot be a path in HDFS; it must accessible on the server the Flume agent is running on (local disk, NFS disk, and so on).

To specify the configuration file path, set the `morphlineFile` property:

```
agent.sinks.k1.morphlineFile=/path/on/local/system/morphline.conf
```

As a Morphline configuration file can contain multiple Morphlines, you must specify the identifier if more than one exists, using the `morphlineId` property:

```
agent.sinks.k1.morphlineId=transform_my_data
```

The next two properties are fairly common among sinks. They specify how many events to remove at a time for processing, also known as a batch. The `batchSize` property defaults to 1000 events, but you might need to set this higher if you aren't consuming events from the channel faster than they are being inserted. Clearly, you can only increase this so much, as the thing you are writing to — in this case, Solr — will have some record consumption limit. Only through testing will you be able to stress your systems to see where the limits are.

The related `batchDurationMillis` property specifies the maximum time to wait before the sink proceeds with the processing when fewer than the `batchSize` number of events have been read. The default value is 1 second and is specified in milliseconds in the configuration properties. In a situation with a light data flow (using the defaults, less than 1000 records per second), setting `batchDurationMillis` higher can make things worse. For instance, if you are using a memory channel with this sink, your Flume agent could be sitting there with data to write to the sink's target but is waiting for more, only to show up when a crash happens, resulting in lost data. That said, your downstream entity might perform better on larger batches, which might push both these configuration values higher, so there is no universally correct answer. Start with the defaults if you are unsure, and use hard data that you'll collect using techniques in *Chapter 8*, *Monitoring Flume*, to adjust based on facts and not guesses.

Finally, you should never need to touch the `handlerClass` property unless you plan to write an alternate implementation of the Morphline processing class. As there is only one Morphline engine implementation to date, I'm not really sure why this is a documented property in Flume. I'm just mentioning it for completeness.

ElasticSearchSink

Another common target to stream data to be searched in NRT is Elasticsearch. Elasticsearch is also a clustered searching platform based on Lucene, like Solr. It is often used along with the logstash project (to create structured logs) and the Kibana project (a web UI for searches). This trio is often referred to as the acronym ELK (Elasticsearch/Logstash/Kibana).

> Here are the project home pages for the ELK stack that can give you a much better overview than I can in a few short pages:
> - Elasticsearch: `http://elasticsearch.org/`
> - Logstash: `http://logstash.net/`
> - Kibana: `http://www.elasticsearch.org/overview/kibana/`

In Elasticsearch, data is grouped into indices. You can think of these as being equivalent to databases in a single MySQL installation. The indices are composed of types (similar to tables in databases), which are made up of documents. A document is like a single row in a database, so, each Flume event will become a single document in ElasticSearch. Documents have one or more fields (just like columns in a database).

This is by no means a complete introduction to Elasticsearch, but it should be enough to get you started, assuming you already have an Elasticsearch cluster at your disposal. As events get mapped to documents by the sink's serializer, the actual sink configuration needs only a few configuration items: where the cluster is located, which index to write to, and what type the record is.

This table summarizes the settings for ElasticSearchSink:

Key	Required	Type	Default
`type`	Yes	`String`	`org.apache.flume.sink.` `elasticsearch.ElasticSearchSink`
`hostNames`	Yes	`String`	A comma-separated list of Elasticsearch nodes to connect to. If the port is specified, use a colon after the name. The default port is 9300.
`clusterName`	No	`String`	`elasticsearch`
`indexName`	No	`String`	`flume`
`indexType`	No	`String`	`log`
`ttl`	No	`String`	Defaults to never expire. Specify the number and unit (5m = 5 minutes).
`batchSize`	No	`int`	100

With this information in mind, let's start by setting the sink's `type` property:

```
agent.sinks.k1.type=org.apache.flume.sink.elasticsearch.
ElasticSearchSink
```

Next, we need to set the list of servers and ports to establish connectivity using the `hostNames` property. This is a comma-separated list of `hostname:port` pairs. If you are using the default port of 9300, you can just specify the server name or IP, for example:

```
agent.sinks.k1.hostNames=es1.example.com,es2.example.com:12345
```

Now that we can communicate with the Elasticsearch servers, we need to tell them which cluster, index, and type to write our documents to. The cluster is specified using the `clusterName` property. This corresponds with the `cluster.name` property in Elasticsearch's `elasticsearch.yml` configuration file. It needs to be specified, as an Elasticsearch node can participate in more than one cluster. Here is how I would specify a nondefault cluster name called `production`:

```
agent.sinks.k1.clusterName=production
```

The `indexName` property is really a prefix used to create a daily index. This keeps any single index from becoming too large over time. If you use the default index name, the index on September 30, 2014 will be named `flume-2014-10-30`.

Lastly, the `indexType` property specifies the Elasticsearch type. If unspecified, the `log` default value will be used.

By default, data written into Elasticsearch will never expire. If you want the data to automatically expire, you can specify a time-to-live value on the records with the `ttl` property. Values are a numeric number in milliseconds or a number with units. The units are given in this table:

Unit string	Definition	Example
`ms`	Milliseconds	`5ms` = 5 milliseconds
`not specified`	Milliseconds	`10000` = 10 seconds
`m`	Minutes	`10m` = 10 minutes
`h`	Hours	`1h` = 1 hour
`d`	Days	`7d` = 7 days
`w`	Weeks	`4w` = 4 weeks

Keep in mind that you also need to enable the TTL features on the Elasticsearch cluster, as it disabled by default. See the Elasticsearch documentation for how to do this.

Finally, like the HDFS sink, the `batch` property is the number of events per transaction that the sink will read from the channel. If you have a large volume of data in your channel, you should see a performance increase by setting this higher than the default of 100, due to the reduced overhead per transaction.

The sink's serializer does the work of transforming the Flume event to the Elasticsearch document. There are two Elasticsearch serializers that come packaged with Flume, neither has additional configuration properties since they mostly use existing headers to dictate field mappings.

We'll see more of this sink in action in *Chapter 7*, *Putting It All Together*.

LogStash Serializer

The default serializer, if not specified, is `ElasticSearchLogStashEventSerializer`:

```
agent.sinks.k1.serializer=org.apache.flume.sink.elasticsearch.
ElasticSearchLogStashEventSerializer
```

It writes data in the same format that Logstash uses in conjunction with Kibana. Here is a table of the commonly used fields and their associated mappings from Flume events:

Elasticsearch field	Taken from the Flume header	Notes
`@timestamp`	`timestamp`	From the header, if present
`@source`	`source`	From the header, if present
`@source_host`	`source_host`	From the header, if present
`@source_path`	`source_path`	From the header, if present
`@type`	`type`	From the header, if present
`@host`	`host`	From the header, if present
`@fields`	all headers	A dictionary of all Flume headers, including the ones that might have been mapped to the other fields, such as the host
`@message`	Flume Body	

While you might think the document's `@type` field will be automatically set to the sink's `indexType` configuration property, you'd be incorrect. If you had only one type of log, it would be wasteful to write this over and over again for every document. However, if you had more than one log type going through your Flume channel, you can designate its type in Elasticsearch using the Static interceptor we'll see in *Chapter 6*, *Interceptors, ETL, and Routing*, to set the `type` (or `@type`) Flume header on the event.

Dynamic Serializer

Another serializer is the `ElasticSearchDynamicSerializer` serializer. If you use this serializer, the event's body is written to a field called `body`. All other Flume header keys are used as field names. Clearly, you want to avoid having a flume header key called `body`, as this will conflict with the actual event's body when transformed into the Elasticsearch document. To use this serializer, specify the fully qualified class name, as shown in this example:

```
agent.sinks.k1.serializer=org.apache.flume.sink.elasticsearch.
ElasticSearchDynamicSerializer
```

For completeness, here is a table that shows you the breakdown of how Flume headers and body get mapped to Elasticsearch fields:

Flume entity	Elasticsearch field
All headers	same as Flume headers
Body	body

As the version of Elasticsearch can be different for each user, Flume doesn't package the Elasticsearch client and corresponding Lucene libraries. Find out from your administrator which versions should be included on the Flume classpath, or check out the Maven `pom.xml` file on GitHub for the corresponding version tag or branch at `https://github.com/elasticsearch/elasticsearch/blob/master/pom.xml`. Make sure the library versions used by Flume match with Elasticsearch or you might see serialization errors.

> As Solr and Elasticsearch have similar capabilities, check out Kelvin Tan's appropriately-named side-by-side detailed feature breakdown webpage. It should help get you started with what is most appropriate for your specific use case:
>
> `http://solr-vs-elasticsearch.com/`

Summary

In this chapter, we covered the HDFS sink in depth, which writes streaming data into HDFS. We covered how Flume can separate data into different HDFS paths based on time or contents of Flume headers. Several file-rolling techniques were also discussed, including time rotation, event count rotation, size rotation, and rotation on idle only.

Compression was discussed as a means to reduce storage requirements in HDFS, and should be used when possible. Besides storage savings, it is often faster to read a compressed file and decompress in memory than it is to read an uncompressed file. This will result in performance improvements in MapReduce jobs run on this data. The splitability of compressed data was also covered as a factor to decide when and which compression algorithm to use.

Event Serializers were introduced as the mechanism by which Flume events are converted into an external storage format, including text (body only), text and headers (headers and body), and Avro serialization (with optional compression).

Next, various file formats, including sequence files (Hadoop key/value files), Data Streams (uncompressed data files, like Avro containers), and Compressed Data Streams, were discussed.

Next, we covered sink groups as a means to route events to different sources using load balancing or failover paths, which can be used to eliminate single points of failure in routing data to its destination.

Finally, we covered two new sinks added in Flume 1.4 to write data to Apache Solr and Elastic Search in a **Near Real Time** (**NRT**) way. For years, MapReduce jobs have served us well, and will continue to do so, but sometimes it still isn't fast enough to search large datasets quickly and look at things from different angles without reprocessing data. KiteSDK Morphlines were also introduced as a way to prepare data for writing to Solr. We will revisit Morphlines again in *Chapter 6, Interceptors, ETL, and Routing*, when we look at a Morphline-powered interceptor.

In the next chapter, we will discuss various input mechanisms (sources) that will feed your configured channels which were covered back in *Chapter 3, Channels*.

5
Sources and Channel Selectors

Now that we have covered channels and sinks, we will now cover some of the more common ways to get data into your Flume agents. As discussed in *Chapter 1*, *Overview and Architecture*, the source is the input point for the Flume agent. There are many sources available with the Flume distribution as well as many open source options available. Like most open source software, if you can't find what you need, you can always write your own by extending the `org.apache.flume.source.AbstractSource` class. Since the primary focus of this book is ingesting files of logs into Hadoop, we'll cover a few of the more appropriate sources to accomplish this.

The problem with using tail

If you have used any of the Flume 0.9 releases, you'll notice that the `TailSource` is no longer a part of Flume. `TailSource` provided a mechanism to "tail" (`http://en.wikipedia.org/wiki/Tail_(Unix)`) any file on the system and create Flume events for each line of the file. It could also handle file rotations, so many used the filesystem as a handoff point between the application creating the data (for instance, log4j) and the mechanism responsible for moving those files someplace else (for instance, syslog).

As is the case with both channels and sinks, events are added and removed from a channel as part of a transaction. When you are tailing a file, there is no way to participate properly in a transaction. If failure to write successfully to a channel occurred, or if the channel was simply full (a more likely event than failure), the data couldn't be "put back" as rollback semantics dictate.

Furthermore, if the rate of data written to a file exceeds the rate Flume could read the data, it is possible to lose one or more log files of input outright. For example, say you were tailing `/var/log/app.log`. When that file reaches a certain size, it is rotated or renamed, to `/var/log/app.log.1`, and a new file called `/var/log/app.log` is created. Let's say you had a favorable review in the press and your application logs are much higher than usual. Flume may still be reading from the rotated file (`/var/log/app.log.1`) when another rotation occurs, moving `/var/log/app.log` to `/var/log/app.log.1`. The file Flume is reading is now renamed to `/var/log/app.log.2`. When Flume finishes with this file, it will move to what it thinks is the next file (`/var/log/app.log`), thus skipping the file that now resides at `/var/log/app.log.1`. This kind of data loss would go completely unnoticed and is something we want to avoid if possible.
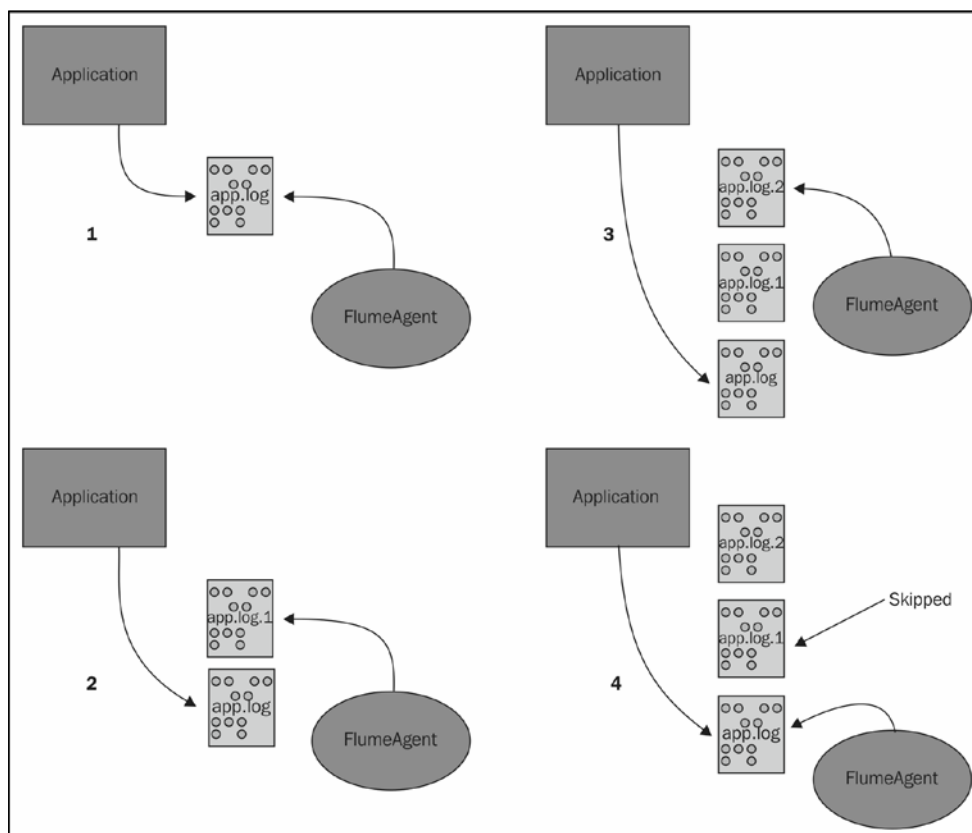
For these reasons, it was decided to remove the tail functionality from Flume when it was refactored. There are some workarounds for `TailSource` after it's removed, but it should be noted that no workaround can eliminate the possibility of data loss under load that occurs under these conditions.

The Exec source

The Exec source provides a mechanism to run a command outside Flume and then turn the output into Flume events. To use the Exec source, set the `type` property to `exec`:

```
agent.sources.s1.type=exec
```

All sources in Flume are required to specify the list of channels to write events to using the `channels` (plural) property. This is a space-separated list of one or more channel names:

```
agent.sources.s1.channels=c1
```

The only other required parameter is the `command` property, which tells Flume what command to pass to the operating system. Here is an example of the use of this property:

```
agent.sources=s1
agent.sources.s1.channels=c1
agent.sources.s1.type=exec
agent.sources.s1.command=tail -F /var/log/app.log
```

Here, I have configured a single source `s1` for an agent named `agent`. The source, an Exec source, will tail the `/var/log/app.log` file and follow any rotations that outside applications may perform on that log file. All events are written to the `c1` channel. This is an example of one of the workarounds for the lack of `TailSource` in Flume 1.x. This is not my preferred workaround to use tail, but just a simple example of the `exec` source type. I will show my preferred method in *Chapter 7, Pulling It All Together*.

> Should you use the `tail -F` command in conjunction with the Exec source, it is probable that the forked process will not shut down 100 percent of the time when the Flume agent shuts down or restarts. This will leave orphaned tail processes that will never exit. The `tail -F` command, by definition, has no end. Even if you delete the file being tailed (at least in Linux), the running tail process will keep the file handle open indefinitely. This keeps the file's space from actually being reclaimed until the tail process exits, which won't happen. I think you are beginning to see why Flume developers don't like tailing files.
>
> If you go this route, be sure to periodically scan the process tables for `tail -F` whose parent PID is 1. These are effectively dead processes and need to be killed manually.

Here is a list of other properties you can use with the Exec source:

Key	Required	Type	Default
`type`	Yes	`String`	`exec`
`channels`	Yes	`String`	space separated list of channels
`command`	Yes	`String`	
`shell`	No	`String`	shell command
`restart`	No	`boolean`	`false`
`restartThrottle`	No	`long` (milliseconds)	`10000` (milliseconds)
`logStdErr`	No	`boolean`	`false`
`batchSize`	No	`int`	`20`
`batchTimeout`	No	`long`	`3000` (milliseconds)

Not every command keeps running, either because it fails (for example, when the channel it is writing to is full) or because it is designed to exit immediately. In this example, we want to record the system load via the Linux `uptime` command, which prints out some system information to `stdout` and exits:

```
agent.sources.s1.command=uptime
```

This command will immediately exit, so you can use the `restart` and `restartThrottle` properties to run it periodically:

```
agent.sources.s1.command=uptime
agent.sources.s1.restart=true
agent.sources.s1.restartThrottle=60000
```

This will produce one event per minute. In the tail example, should the channel fill causing the Exec source to fail, you can use these properties to restart the Exec source. In this case, setting the `restart` property will start the tailing of the file from the beginning of the current file, thus producing duplicates. Depending on how long the `restartThrottle` property is, you may have missed some data due to a file rotation outside Flume. Furthermore, the channel may *still* be unable to accept data, in which case the source will fail again. Setting this value too low means giving less time to the channel to drain, and unlike some of the sinks we saw, there is not an option for exponential backoff.

If you need to use shell-specific features such as wildcard expansion, you can set the `shell` property as in this example:

```
agent.sources.s1.command=grep –i apache lib/*.jar | wc -l
agent.sources.s1.shell=/bin/bash -c
agent.sources.s1.restart=true
agent.sources.s1.restartThrottle=60000
```

This example will find the number of times the case-insensitive `apache` string is found in all the JAR files in the `lib` directory. Once per minute, that count will be sent as a Flume event payload.

While the command output written to `stdout` becomes the Flume event body, errors are sometimes written to `stderr`. If you want these lines included in the Flume agent's system logs, set the `logStdErr` property to `true`. Otherwise, they will be silently ignored, which is the default behavior.

Finally, you can specify the number of events to write per transaction by changing the `batchSize` property. You may need to set this value higher than the default of 20 if your input data is large and you realize that you cannot write to your channel fast enough. Using a higher batch size reduces the overall average transaction overhead per event. Testing with different values and monitoring the channel's `put` rate is the only way to know this for sure. The related `batchTimeout` property sets the maximum time to wait when records fewer than the batch size's number of records have been seen before, flushing a partial batch to the channel. The default setting for this is 3 seconds (specified in milliseconds).

Spooling Directory Source

In an effort to avoid all the assumptions inherent in tailing a file, a new source was devised to keep track of which files have been converted into Flume events and which still need to be processed. The `SpoolingDirectorySource` is given a directory to watch for new files appearing. It is assumed that files copied to this directory are complete. Otherwise, the source might try and send a partial file. It also assumes that filenames never change. Otherwise, on restarting, the source would forget which files have been sent and which have not. The filename condition can be met in log4j using `DailyRollingFileAppender` rather than `RollingFileAppender`. However, the currently open file would need to be written to one directory and copied to the spool directory after being closed. None of the log4j appenders shipping have this capability.

That said, if you are using the Linux `logrotate` program in your environment, this might be of interest. You can move completed files to a separate directory using a `postrotate` script. The final flow might look something like this:
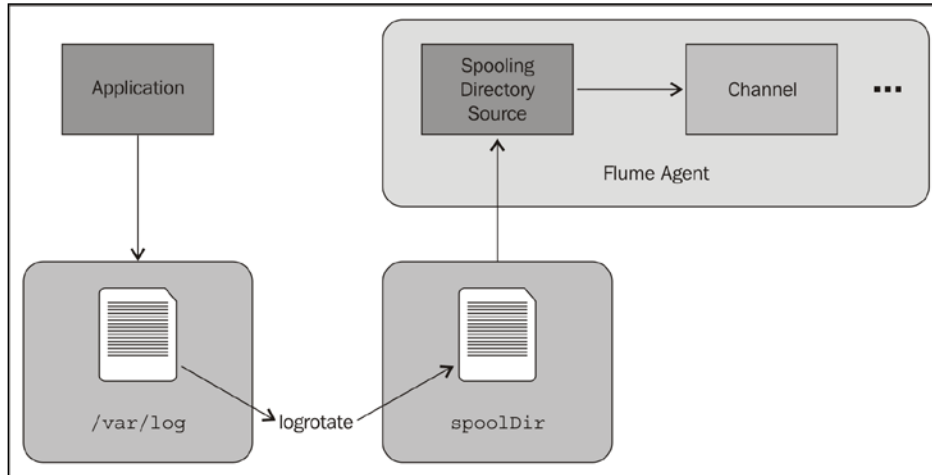
To create a Spooling Directory Source, set the `type` property to `spooldir`. You must specify the directory to watch by setting the `spoolDir` property:

```
agent.sources=s1
agent.sources.channels=c1
agent.sources.s1.type=spooldir
agent.sources.s1.spoolDir=/path/to/files
```

Here is a summary of the properties for the Spooling Directory Source:

Key	Required	Type	Default
type	Yes	String	spooldir
channels	Yes	String	space separated list of channels
spoolDir	Yes	String	path to directory to spool
fileSuffix	No	String	.COMPLETED
deletePolicy	No	String (never or immediate)	never
fileHeader	No	boolean	false
fileHeaderKey	No	String	file
basenameHeader	No	boolean	false

Key	Required	Type	Default
basenameHeaderKey	No	String	basename
ignorePattern	No	String	^$
trackerDir	No	String	${spoolDir}/.flumespool
consumeOrder	No	String (oldest, youngest, or random)	oldest
batchSize	No	int	10
bufferMaxLines	No	int	100
maxBufferLineLength	No	int	5000
maxBackoff	No	int	4000 (milliseconds)

When a file has been completely transmitted, it will be renamed with a `.COMPLETED` extension, unless overridden by setting the `fileSuffix` property, like this:

```
agent.sources.s1.fileSuffix=.DONE
```

Starting with Flume 1.4, a new property, `deletePolicy`, was created to remove completed files from the filesystem rather than just marking them as done. In a production environment, this is critical because your spool disk will fill up over time. Currently, you can only set this for immediate deletion or to leave the files forever. If you want delayed deletion, you'll need to implement your own periodic (cron) job, perhaps using the `find` command to find files in the spool directory with the COMPLETED file suffix and a modification time longer than some regular value, for example:

```
find /path/to/spool/dir -type f -name "*.COMPLETED" –mtime 7 –exec rm
{} \;
```

This will find all files completed more than seven days ago and delete them.

If you want the absolute file path attached to each event, set the `fileHeader` property to `true`. This will create a header with the `file` key unless set to something else using the `fileHeaderKey` property, like this would add the {sourceFile=/path/to/files/foo.1234.log} header if the event was read from the /path/to/files/foo.1234.log file:

```
agent.sources.s1.fileHeader=true
agent.sources.s1.fileHeaderKey=sourceFile
```

The related property, `basenameHeader`, if set to `true`, will add a header with the `basename` key, which contains just the filename. The `basenameHeaderKey` property allows you to change the key's value, as shown here:

```
agent.sources.s1.basenameHeader=true
agent.sources.s1.basenameHeaderKey=justTheName
```

This configuration would add the `{justTheName=foo.1234.log}` header if the event was read from the same file located at `/path/to/files/foo.1234.log`.

If there are certain file patterns that you do not want this source to read as input, you can pass a regular expression using the `ignorePattern` property. Personally, I won't copy any files I don't want transferred to Flume in the `spoolDir` in the first place. If this situation cannot be avoided, use the `ignorePattern` property to pass a regular expression to match filenames that should not be transferred as data. Furthermore, subdirectories and files that start with a `"."` (period) character are ignored, so you can avoid costly regular expression processing using this convention instead.

While Flume is sending data, it keeps track of how far it has gotten in each file by keeping a metadata file in the directory specified by the `trackerDir` property. By default, this file will be `.flumespool` under `spoolDir`. Should you want a location other than inside `spoolDir`, you can specify an absolute file path.

Files in the directory are processed in the "oldest first" manner as calculated by looking at the modification times of the files. You can change this behavior by setting the `consumeOrder` property. If you set this property to `youngest`, the newest files will be processed first. This may be desired if data is time sensitive. If you'd rather give equal precedence to all files, you can set this property to a value of `random`.

The `batchSize` property allows you to tune the number of events per transaction for writes to the channel. Increasing this may provide better throughput at the cost of larger transactions (and possibly larger rollbacks). The `bufferMaxLines` property is used to set the size of the memory buffer used in reading files by multiplying it with `maxBufferLineLength`. If your data is very short, you might consider increasing `bufferMaxLines` while reducing the `maxBufferLineLength` property. In this case, it will result in better throughput without increasing your memory overhead. That said, if you have events longer than 5000 characters, you'll want to set `maxBufferLineLength` higher.

If there is a problem writing data to the channel, a `ChannelException` is thrown back to the source, where it'll retry after an initial wait time of 250 ms. Each failed attempt will double this time up to a maximum of 4 seconds. To set this maximum time higher, set the `maxBackoff` property. For instance, if I wanted a maximum of 5 minutes, I can set it in milliseconds like this:

```
agent.sources.s1.maxBackoff=300000
```

Finally, you'll want to ensure that whatever mechanism is writing new files to your spooling directory creates unique filenames, such as adding a timestamp (and possibly more). Reusing a filename will confuse the source, and your data may not be processed.

As always, remember that restarts and errors will create duplicates due to retransmission of files partially sent, but not marked as complete, or because the metadata is incomplete.

Syslog sources

Syslog has been around for decades and is often used as an operating-system-level mechanism to capture and move logs around systems. In many ways, there are overlaps with some of the functionality Flume provides. There is even a Hadoop module for rsyslog, one of the more modern variants of syslog (`http://www.rsyslog.com/doc/rsyslog_conf_modules.html/omhdfs.html`). Generally, I don't like solutions that couple technologies that may version independently. If you use this rsyslog/Hadoop integration, you would be required to update the version of Hadoop you compiled into rsyslog at the same time you upgraded your Hadoop cluster to a new major version. This may be logistically difficult if you have a large number of servers and/or environments. Backward compatibility in Hadoop wire protocols is something that is being actively worked on in the Hadoop community, but currently, it isn't the norm. We'll talk more about this in *Chapter 8*, *Monitoring Flume*, when we discuss tiering data flows.

Syslog has an older UDP transport as well as a newer TCP protocol that can handle data larger than a single UDP packet can transmit (about 64 KB) and deal with network-related congestion events that might require the data to be retransmitted.

Finally, there are some undocumented properties of syslog sources that allow us to add more regular-expression pattern-matching for messages that do not conform to RFC standards. I won't be discussing these additional settings, but you should be aware of them if you run into frequent parsing errors. In this case, take a look at the source for `org.apache.flume.source.SyslogUtils` for implementation details to find the cause.

More details on syslog terms (such as a facility) and standard formats can be found in *RFC 3164* at `http://tools.ietf.org/html/rfc3164`.

The syslog UDP source

The UDP version of syslog is usually safe to use when you are receiving data from the server's local syslog process, provided the data is small enough (less than about 64 KB).
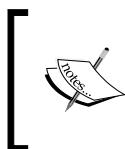
> The implementation for this source has chosen 2,500 bytes as the maximum payload size regardless of what your network can actually handle. So if your payload will be larger than this, use one of the TCP sources instead.

To create a Syslog UDP source, set the `type` property to `syslogudp`. You must set the port to `listen` on using the `port` property. The optional `host` property specifies the bind address. If no host is specified, all IPs for the server will be used, which is the same as specifying `0.0.0.0`. In this example, we will only listen for local UDP connections on port `5140`:

```
agent.sources=s1
agent.sources.channels=c1
agent.sources.s1.type=syslogudp
agent.sources.s1.host=localhost
agent.sources.s1.port=5140
```

If you want syslog to forward a tailed file, you can add a line like this to your syslog configuration file:

```
*.err;*.alert;*.crit;*.emerg;kern.*    @localhost:5140
```

This will send all error priority, critical priority, emergency priority, and kernel messages of any priority into your Flume source. The single `@` symbol designates that UDP protocol should be used.

Here is a summary of the properties of the Syslog UDP source:

Key	Required	Type	Default
type	Yes	String	syslogudp
channels	Yes	String	space separated list of channels
port	Yes	int	
host	No	String	0.0.0.0

Key	Required	Type	Default
`keepFields`	No	`boolean`	`false`

The `keepFields` property tells the source to include the syslog fields as part of the body. By default, these are simply removed, as they become Flume header values.

The Flume headers created by the Syslog UDP source are summarized here:

Header Key	Description
`Facility`	This is the syslog facility. See the syslog documentation.
`Priority`	This is the syslog priority. See the syslog documentation.
`timestamp`	This is the time of the syslog event, translated into an epoch timestamp. It's omitted if it's not parsed from one of the standard RFC formats.
`hostname`	This is the parsed hostname in the syslog message. It is omitted if it is not parsed.
`flume.syslog.status`	There was a problem parsing the syslog message's headers. This value is set to `Invalid` if the payload didn't conform to the RFCs, and set to `Incomplete` if the message was longer than the `eventSize` value (for UDP, this is set internally to 2,500 bytes). It's omitted if everything is fine.

The syslog TCP source

As previously mentioned, the Syslog TCP source provides an endpoint for messages over TCP, allowing for a larger payload size and TCP retry semantics that should be used for any reliable inter-server communications.

To create a Syslog TCP source, set the `type` property to `syslogtcp`. You must still set the bind address and port to listen on:

```
agent.sources=s1
agent.sources.s1.type=syslogtcp
agent.sources.s1.host=0.0.0.0
agent.sources.s1.port=12345
```

If your syslog implementation supports syslog over TCP, the configuration is usually the same, except that a double @ symbol is used to indicate TCP transport. Here is the same example using TCP, where I am forwarding the values to a Flume agent that is running on a different server named `flume-1`.

```
*.err;*.alert;*.crit;*.emerg;kern.*    @@flume-1:12345
```

There are some optional properties for the Syslog TCP source, as listed here:

Key	Required	Type	Default
`type`	Yes	`String`	`syslogtcp`
`channels`	Yes	`String`	space separated list of channels
`port`	Yes	`int`	
`host`	No	`String`	`0.0.0.0`
`keepFields`	No	`boolean`	`false`
`eventSize`	No	`int` (bytes)	2500 bytes

The `keepFields` property tells the source to include the syslog fields as part of the body. By default, these are simply removed, as they become Flume header values.

The Flume headers created by the Syslog TCP source are summarized here:

Header Key	Description
`Facility`	This is the syslog facility. See the syslog documentation.
`Priority`	This is the syslog priority. See the syslog documentation.
`timestamp`	This is the time of the syslog event translated into an epoch timestamp. It's omitted if not parsed from one of the standard RFC formats.
`hostname`	The parsed hostname in the syslog message. It's omitted if not parsed.
`flume.syslog.status`	There was a problem parsing the syslog message's headers. It's set to `Invalid` if the payload didn't conform to the RFCs and set to `Incomplete` if the message was longer than the configured `eventSize`. It's omitted if everything is fine.

The multiport syslog TCP source

The Multiport Syslog TCP source is nearly identical in functionality to the Syslog TCP source, except that it can listen to multiple ports for input. You may need to use this capability if you are unable to change which port syslog will use in its forwarding rules (it may not be your server at all). It is more likely that you will use this to read multiple formats using one source to write to different channels. We'll cover that in a moment in the *Channel Selectors* section. Under the hood, a high-performance asynchronous TCP library called Mina (`https://mina.apache.org/`) is used, which often provides better throughput on multicore servers even when consuming only a single TCP port.

To configure this source, set the `type` property to `multiport_syslogtcp`:

```
agent.sources.s1.type=multiport_syslogtcp
```

Like the other syslog sources, you need to specify the port, but in this case it is a space-separated list of ports. You can use this only if you have one port specified. The property for this is `ports` (plural):

```
agent.sources.s1.type=multiport_syslogtcp
agent.sources.s1.channels=c1
agent.sources.s1.ports=33333 44444
agent.sources.s1.host=0.0.0.0
```

This code configures the Multiport Syslog TCP source named `s1` to listen to any incoming connections on ports `33333` and `44444` and send them to channel `c1`.

In order to tell which event came from which port, you can set the optional `portHeader` property to the name of the key whose value will be the port number. Let's add this property to the configuration:

```
agent.sources.s1.portHeader=port
```

Then, any events received from port 33333 would have a header key/value of {`"port"="33333"`}. As you saw in *Chapter 4*, *Sinks and Sink Processors*, you can now use this value (or any header) as a part of your `HDFSSink` file path convention, like this:

```
agent.sinks.k1.hdfs.path=/logs/%{hostname}/%{port}/%Y/%m/%D/%H
```

Here is a complete table of the properties:

Key	Required	Type	Default
type	Yes	String	syslogtcp
channels	Yes	String	Space-separated list of channels
ports	Yes	int	Space-separated list of port numbers
host	No	String	0.0.0.0
keepFields	No	boolean	false
eventSize	No	int	2500 (bytes)
portHeader	No	String	
batchSize	No	int	100
readBufferSize	No	int (bytes)	1024
numProcessors	No	int	automatically detected
charset.default	No	String	UTF-8
charset.port.PORT#	No	String	

This TCP source has some additional tunable options over the standard TCP syslog source. The first is the `batchSize` property. This is the number of events processed per transaction with the channel. There is also the `readBufferSize` property. It specifies the internal buffer size used by an internal Mina library. Finally, the `numProcessors` property is used to size the worker thread pool in Mina. Before you tune these parameters, you may want to familiarize yourself with Mina (`http://mina.apache.org/`), and look at the source code before deviating from the defaults.

Finally, you can specify the default and per-port character encoding to use when converting between Strings and bytes:

```
agent.sources.s1.charset.default=UTF-16
agent.sources.s1.charset.port.33333=UTF-8
```

This sample configuration shows that all ports will be interpreted using UTF-16 encoding, except for port 33333 traffic, which will use UTF-8.

As you've already seen in the other syslog sources, the `keepFields` property tells the source to include the syslog fields as part of the body. By default, these are simply removed, as they become Flume header values.

The Flume headers created by this source are summarized here:

Header Key	Description
Facility	This is the syslog facility. See the syslog documentation.
Priority	This is the syslog priority. See the syslog documentation.
timestamp	This is the time of the syslog event translated into an epoch timestamp. Omitted if not parsed from one of the standard RFC formats.
hostname	This is the parsed hostname in the syslog message. Omitted if not parsed.
flume.syslog.status	There was a problem parsing the syslog message's headers. This is set to `Invalid` if the payload didn't conform to the RFCs, set to `Incomplete` if the message was longer than the configured `eventSize`, and omitted if everything is fine.

JMS source

Sometimes, data can originate from asynchronous message queues. For these cases, you can use Flume's JMS source to create events read from a JMS Queue or Topic. While it is theoretically possible to use any **Java Message Service** (**JMS**) implementation, Flume has only been tested with ActiveMQ, so be sure to test thoroughly if you use a different provider.

Like the previously covered ElasticSearch sink in *Chapter 4*, *Sinks and Sink Processors*, Flume does not come packaged with the JMS implementation you'll be using, as the versions need to match up, so you'll need to include the necessary implementation JAR files on the Flume agent's classpath. The preferred method is to use the `--plugins-dir` parameter mentioned in *Chapter 2*, *A Quick Start Guide to Flume*, which we'll cover in more detail in the next chapter.

> ActiveMQ is just one provider of the Java Message Service API. For more information, see the project homepage at `http://activemq.apache.org`.

To configure this source, set the `type` property to `jms`:

```
agent.sources.s1.type=jms
```

The first three properties are used to establish a connection with the JMS Server. They are `initialContextFactory`, `connectionFactory`, and `providerURL`. The `initialContextFactory` property for ActiveMQ will be `org.apache.activemq.jndi.ActiveMQInitialContextFactory`. The `connectionFactory` property will be the registered JNDI name, which defaults to `ConnectionFactory` if unspecified. Finally, `providerURL` is the connection String passed to the connection factory to actually establish a network connection. It is usually a URL-like String consisting of the server name and port information. If you aren't familiar with your JMS configuration, ask somebody who does what values to use in your environment.

This table summarizes these settings and others we'll discuss in a moment:

Key	Required	Type	Default
`type`	Yes	`String`	`jms`
`channels`	Yes	`String`	space separated list of channels
`initialContextFactory`	Yes	`String`	
`connectionFactory`	No	`String`	`ConnectionFactory`
`providerURL`	Yes	`String`	

Key	Required	Type	Default
userName	No	String	username for authentication
passwordFile	No	String	path to file containing password for authentication
destinationName	Yes	String	
destinationType	Yes	String	
messageSelector	No	String	
errorThreshold	No	int	10
pollTimeout	No	long	1000 (milliseconds)
batchSize	No	int	100

If your JMS Server requires authentication, pass the `userName` and `passwordFile` properties:

```
agent.sources.s1.userName=jms_bot_user
agent.sources.s1.passwordFile=/path/to/password.txt
```

Putting the password in a file you reference, rather than directly in the Flume configuration, allows you to keep the permissions on the configuration open for inspection, while storing the more sensitive password data in a separate file that is accessible only to the Flume agent (restrictions are commonly provided by the operating system's permissions system).

The `destinationName` property decides which message to read from our connection. Since JMS supports both queues and topics, you need to set the `destinationType` property to `queue` or `topic`, respectively. Here's what the properties might look like for a queue:

```
agent.source.s1.destinationName=my_cool_data
agent.source.s1.destinationType=queue
```

For a topic, it will look as follows:

```
agent.source.s1.destinationName=restart_events
agent.source.s1.destinationType=topic
```

The difference between a queue and a topic is basically the number of entities that will be read from the named destination. If you are reading from a queue, the message will be removed from the queue once it is written to Flume as an event. A topic, once read, is still available for other entities to read.

Should you need only a subset of messages published to a topic or queue, the optional `messageSelector` String property provides this capability. For example, to create Flume events on messages with a field called `Age` that is larger than 10, I can specify a `messageSelector` filter:

```
agent.source.s1.messageSelector="Age > 10"
```
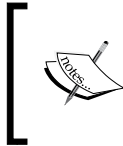
> Describing the selector capabilities and syntax in detail is far beyond the scope of this book. See `http://docs.oracle.com/javaee/1.4/api/javax/jms/Message.html` or pick up a book on JMS to become more familiar with JMS message selectors.

Should there be a problem in communicating with the JMS server, the connection will reset itself after a number of failures specified by the `errorThreshold` property. The default value of `10` is reasonable and will most likely not need to be changed.

Next, the JMS source has a property used to adjust the value of `batchSize`, which defaults to `100`. By now, you should be fairly familiar with adjusting batch sizes with other sources and sinks. For high-volume flows, you'll want to set this higher to consume data in larger chunks from your JMS server to get higher throughput. Setting this too high for low volume flows could delay processing. As always, testing is the only sure way to adjust this properly. The related `pollTimeout` property specifies how long to wait for new messages to appear before attempting to read a batch. The default, specified in milliseconds, is 1 second. If no messages are read before the poll timeout, the Source will go into an exponential backoff mode before attempting another read. This could delay the processing of messages until it wakes up again. Chances are that you won't need to change this value from the default.

When the message gets converted into a Flume event, the message properties become Flume headers. The message payload becomes the Flume body, but since JMS uses serialized Java objects, we need to tell the JMS source how to interpret the payload. We do this by setting the `converter.type` property, which defaults to the only implementation that is packaged with Flume using the `DEFAULT` String (implemented by the `DefaultJMSMessageConverer` class). It can deserialize JMS BytesMessages, TextMessages, and ObjectMessages (Java objects that implement the `java.io.DataOutput` interface). It does not handle StreamMessages or MapMessages, so if you need to process them, you'll need to implement your own converter type. To do this, you'll implement the `org.apache.flume.source.jms.JMSMessageConverter` interface, and use its fully qualified class name as the `converter.type` property value.

For completeness, here are these properties in tabular form:

Key	Required	Type	Default
converter.type	No	String	DEFAULT
converter.charset	No	String	UTF-8

Channel selectors

As we discussed in *Chapter 1*, *Overview and Architecture*, a source can write to one or more channels. This is why the property is plural (`channels` instead of `channel`). There are two ways multiple channels can be handled. The event can be written to all the channels or to just one channel, based on some Flume header value. The internal mechanism for this in Flume is called a **channel selector**.

The selector for any channel can be specified using the `selector.type` property. All selector-specific properties begin with the usual `Source` prefix: the agent name, keyword sources, and source name:

```
agent.sources.s1.selector.type=replicating
```

Replicating

If you do not specify a selector for a source, `replicating` is the default. The `replicating` selector writes the same event to all channels in the source's channels list:

```
agent.sources.s1.channels=c1 c2 c3
agent.sources.s1.selector.type=replicating
```

In this example, every event will be written to all three channels: `c1`, `c2`, and `c3`.

There is an optional property on this selector, called `optional`. It is a space-separated list of channels that are optional. Consider this modified example:

```
agent.sources.s1.channels=c1 c2 c3
agent.sources.s1.selector.type=replicating
agent.sources.s1.selector.optional=c2 c3
```

Now, any failure to write to channels `c2` or `c3` will not cause the transaction to fail, and any data written to `c1` will be committed. In the earlier example with no optional channels, any single channel failure would roll back the transaction for all channels.

Multiplexing

If you want to send different events to different channels, you should use a multiplexing channel selector by setting the value of `selector.type` to `multiplexing`. You also need to tell the channel selector which header to use by setting the `selector.header` property:

```
agent.sources.s1.selector.type=multiplexing
agent.sources.s1.selector.header=port
```

Let's assume we used the Multiport Syslog TCP source to listen on four ports—11111, 22222, 33333, and 44444—with a `portHeader` setting of `port`:

```
agent.sources.s1.selector.default=c2
agent.sources.s1.selector.mapping.11111=c1 c2
agent.sources.s1.selector.mapping.44444=c2
agent.sources.s1.selector.optional.44444=c3
```

This configuration will result in the traffic of port 22222 and port 33333 going to the `c2` channel only. The traffic of port 11111 will go to the `c1` and `c2` channels. A failure on either channel would result in nothing being added to either channel. The traffic of port `44444` will go to channels `c2` and `c3`. However, a failure to write to `c3` will still commit the transaction to `c2`, and `c3` will not be attempted again with that event.

Summary

In this chapter, we covered in depth the various sources that we can use to insert log data into Flume, including the Exec source, the Spooling Directory Source, Syslog sources (UDP, TCP, and multiport TCP), and the JMS source.

We discussed replicating the old `TailSource` functionality in Flume 0.9 and problems with using tail semantics in general.

We also covered channel selectors and sending events to one or more channels, specifically the replicating and multiplexing channel selectors.

Optional channels were also discussed as a way to only fail a put transaction for only some of the channels when more than one channel is used.

In the next chapter, we'll introduce interceptors that will allow in-flight inspection and transformation of events. Used in conjunction with channel selectors, interceptors provide the final piece to create complex data flows with Flume. Additionally, we will cover RPC mechanisms (source/sink pairs) between Flume agents using both Avro and Thrift, which can be used to create complex data flows.

6
Interceptors, ETL, and Routing

The final piece of functionality required in your data processing pipeline is the ability to inspect and transform events in flight. This can be accomplished using **interceptors**. Interceptors, as we discussed in *Chapter 1*, *Overview and Architecture*, can be inserted after a source creates an event, but before writing to the channel occurs.

Interceptors

An interceptor's functionality can be summed up with this method:

```
public Event intercept(Event event);
```

A Flume event is passed to it, and it returns a Flume event. It may do nothing, in which case, the same unaltered event is returned. Often, it alters the event in some useful way. If `null` is returned, the event is dropped.

To add interceptors to a source, simply add the `interceptors` property to the named source, for example:

```
agent.sources.s1.interceptors=i1 i2 i3
```

This defines three interceptors: `i1`, `i2`, and `i3` on the `s1` source for the agent named `agent`.
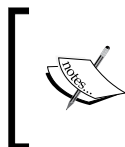
> Interceptors are run in the order in which they are listed. In the preceding example, `i2` will receive the output from `i1`. Then, `i3` will receive the output from `i2`. Finally, the channel selector receives the output from `i3`.

Now that we have defined the interceptor by name, we need to specify its type as follows:

```
agent.sources.s1.interceptors.i1.type=TYPE1
agent.sources.s1.interceptors.i1.additionalProperty1=VALUE
agent.sources.s1.interceptors.i2.type=TYPE2
agent.sources.s1.interceptors.i3.type=TYPE3
```

Let's look at some of the interceptors that come bundled with Flume to get a better idea of how to configure them.

Timestamp

The **Timestamp interceptor**, as its name suggests, adds a header with the `timestamp` key to the Flume event if one doesn't already exist. To use it, set the `type` property to `timestamp`.

If the event already contains a timestamp header, it will be overwritten with the current time unless configured to preserve the original value by setting the `preserveExisting` property to `true`.

Here is a table summarizing the properties of the Timestamp interceptor:

Key	Required	Type	Default
type	Yes	String	timestamp
preserveExisting	No	boolean	false

Here is what a total configuration for a source might look like if we only want it to add a timestamp header if none exists:

```
agent.sources.s1.interceptors=i1
agent.sources.s1.interceptors.i1.type=timestamp
agent.sources.s1.interceptors.i1.preserveExisting=true
```

Recall this HDFSSink path from *Chapter 4*, *Sinks and Sink Processors*, utilizing the event date:

```
agent.sinks.k1.hdfs.path=/logs/apache/%Y/%m/%d/%H
```

The `timestamp` header is what determines this path. If it is missing, you can be sure Flume will not know where to create the files, and you will not get the result you are looking for.

Host

Similar in simplicity to the Timestamp interceptor, the **Host interceptor** will add a header to the event containing the IP address of the current Flume agent. To use it, set the `type` property to `host`:

```
agent.sources.s1.interceptors=i1
agent.sources.s1.interceptors.type=host
```

The key for this header will be `host` unless you specify something else using the `hostHeader` property. Like before, an existing header will be overwritten, unless you set the `preserveExisting` property to `true`. Finally, if you want a reverse DNS lookup of the hostname to be used instead of the IP as a value, set the `useIP` property to `false`. Remember that reverse lookups will add processing time to your data flow.

Here is a table summarizing the properties of the Host interceptor:

Key	Required	Type	Default
type	Yes	String	host
hostHeader	No	String	host
preserveExisting	No	boolean	false
useIP	No	boolean	true

Here is what a total configuration for a source might look like if we only want it to add a `relayHost` header containing the DNS hostname of this agent to every event:

```
agent.sources.s1.interceptors=i1
agent.sources.s1.interceptors.i1.type=host
agent.sources.s1.interceptors.i1.hostHeader=relayHost
agent.sources.s1.interceptors.i1.useIP=false
```

This interceptor might be useful if you wanted to record the path your events took though your data flow, for instance. Chances are you are more interested in the origin of the event rather than the path it took, which is why I have yet to use this.

Static

The **Static interceptor** is used to insert a single key/value header into each Flume event processed. If more than one key/value is desired, you simply add additional Static interceptors. Unlike the interceptors we've looked at so far, the default behavior is to preserve existing headers with the same key. As always, my recommendation is to always specify what you want and not rely on the defaults.

I do not know why the key and value properties are not required, as the defaults are not terribly useful.

Here is a table summarizing the properties of the Static interceptor:

Key	Required	Type	Default
type	Yes	String	static
key	No	String	key
value	No	String	value
preserveExisting	No	boolean	true

Finally, let's look at an example configuration that inserts two new headers, provided they don't already exist in the event:

```
agent.sources.s1.interceptors=pos env
agent.sources.s1.interceptors.pos.type=static
agent.sources.s1.interceptors.pos.key=pointOfSale
agent.sources.s1.interceptors.pos.value=US
agent.sources.s1.interceptors.env.type=static
agent.sources.s1.interceptors.env.key=environment
agent.sources.s1.interceptors.env.value=staging
```

Regular expression filtering

If you want to filter events based on the content of the body, the **regular expression filtering interceptor** is your friend. Based on a regular expression you provide, it will either filter out the matching events or keep only the matching events. Start by setting the type interceptor to regex_filter. The pattern you want to match is specified using a Java-style regular expression syntax. See these javadocs for usage details at http://docs.oracle.com/javase/6/docs/api/java/util/regex/Pattern.html. The pattern string is set in the regex property. Be sure to escape backslashes in Java Strings. For instance, the \d+ pattern would need to be written with two backslashes: \\d+. If you wanted to match a backslash, the documentation says to type two backslashes, but each needs to be escaped, resulting in four, that is, \\\\. You will see the use of escaped backslashes throughout this chapter. Finally, you need to tell the interceptor if you want to exclude matching records by setting the excludeEvents property to true. The default (false) indicates that you want to only keep events that match the pattern.

Here is a table summarizing the properties of the regular expression filtering interceptor:

Key	Required	Type	Default
type	Yes	String	regex_filter
regex	No	String	.*
excludeEvents	No	boolean	false

In this example, any events containing the NullPointerException string will be dropped:

```
agent.sources.s1.interceptors=npe
agent.sources.s1.interceptors.npe.type=regex_filter
agent.sources.s1.interceptors.npe.regex=NullPointerException
agent.sources.s1.interceptors.npe.excludeEvents=true
```

Regular expression extractor

Sometimes, you'll want to extract bits of your event body into Flume headers so that you can perform routing via Channel Selectors. You can use the **regular expression extractor interceptor** to perform this function. Start by setting the type interceptor to regex_extractor:

```
agent.sources.s1.interceptors=e1
agent.sources.s1.interceptors.e1.type=regex_extractor
```

Like the regular expression filtering interceptor, the regular expression extractor interceptor too uses a Java-style regular expression syntax. In order to extract one or more fields, you start by specifying the regex property with group matching parentheses. Let's assume we are looking for error numbers in our events in the Error: N form, where N is a number:

```
agent.sources.s1.interceptors=e1
agent.sources.s1.interceptors.e1.type=regex_extractor
agent.sources.s1.interceptors.e1.regex=Error:\\s(\\d+)
```

As you can see, I put capture parentheses around the number, which may be one or more digit. Now that I've matched my desired pattern, I need to tell Flume what to do with my match. Here, we need to introduce **serializers**, which provide a pluggable mechanism for how to interpret each match. In this example, I've only got one match, so my space-separated list of serializer names has only one entry:

```
agent.sources.s1.interceptors=e1
agent.sources.s1.interceptors.e1.type=regex_extractor
agent.sources.s1.interceptors.e1.regex=Error:\\s(\\d+)
agent.sources.s1.interceptors.e1.serializers=ser1
agent.sources.s1.interceptors.e1.serializers.ser1.type=default
agent.sources.s1.interceptors.e1.serializers.ser1.name=error_no
```

The `name` property specifies the event key to use, where the value is the matching text from the regular expression. The type of `default` value (also the default if not specified) is a simple pass-through serializer. For this event body, look at the following:

```
NullPointerException: A problem occurred. Error: 123. TxnID: 5X2T9E.
```

The following header would be added to the event:

```
{ "error_no":"123" }
```

If I wanted to add the `TxnID` value as a header, I'd simply add another matching pattern group and serializer:

```
agent.sources.s1.interceptors=e1
agent.sources.s1.interceptors.e1.type=regex_extractor
agent.sources.s1.interceptors.e1.regex=Error:\\s(\\d+).*TxnID:\\s(\\
w+)
agent.sources.s1.interceptors.e1.serializers=ser1 ser2
agent.sources.s1.interceptors.e1.serializers.ser1.type=default
agent.sources.s1.interceptors.e1.serializers.ser1.name=error_no
agent.sources.s1.interceptors.e1.serializers.ser2.type=default
agent.sources.s1.interceptors.e1.serializers.ser2.name=txnid
```

Then, I would create these headers for the preceding input:

```
{ "error_no":"123", "txnid":"5x2T9E" }
```

However, take a look at what would happen if the fields were reversed as follows:

```
NullPointerException: A problem occurred. TxnID: 5X2T9E. Error: 123.
```

I would wind up with only a header for `txnid`. A better way to handle this kind of ordering would be to use multiple interceptors so that the order doesn't matter:

```
agent.sources.s1.interceptors=e1 e2
agent.sources.s1.interceptors.e1.type=regex_extractor
agent.sources.s1.interceptors.e1.regex=Error:\\s(\\d+)
agent.sources.s1.interceptors.e1.serializers=ser1
agent.sources.s1.interceptors.e1.serializers.ser1.type=default
agent.sources.s1.interceptors.e1.serializers.ser1.name=error_no
agent.sources.s1.interceptors.e2.type=regex_extractor
agent.sources.s1.interceptors.e2.regex=TxnID:\\s(\\w+)
agent.sources.s1.interceptors.e2.serializers=ser1
agent.sources.s1.interceptors.e2.serializers.ser1.type=default
agent.sources.s1.interceptors.e2.serializers.ser1.name=txnid
```

The only other type of serializer implementation that ships with Flume, other than the pass-through, is to specify the fully qualified class name of `org.apache.flume.interceptor.RegexExtractorInterceptorMillisSerializer`. This serializer is used to convert times into milliseconds. You need to specify a pattern property based on `org.joda.time.format.DateTimeFormat` patterns.

For instance, let's say you were ingesting Apache Web Server access logs, for example:

192.168.1.42 - - [29/Mar/2013:15:27:09 -0600] "GET /index.html HTTP/1.1"
200 1037

The complete regular expression for this might look like this (in the form of a Java String, with backslash and quotes escaped with an extra backslash):

^([\\d.]+) \\S+ \\S+ \\[([\\w:/]+\\s[+\\-]\\d{4})\\] \"(.+?)\" (\\d{3})
(\\d+)

The time pattern matched corresponds to the `org.joda.time.format.DateTimeFormat` pattern:

```
yyyy/MMM/dd:HH:mm:ss Z
```

Take a look at what would happen if we make our configuration something like this:

```
agent.sources.s1.interceptors=e1
agent.sources.s1.interceptors.e1.type=regex_extractor
```

```
agent.sources.s1.interceptors.e1.regex=^([\\d.]+) \\S+ \\S+ \\
[([\\w:/]+\\s[+\\-]\\d{4})\\] \"(.+?)\" (\\d{3}) (\\d+)
agent.sources.s1.interceptors.e1.serializers=ip dt url sc bc
agent.sources.s1.interceptors.e1.serializers.ip.name=ip_address
agent.sources.s1.interceptors.e1.serializers.dt.type=org.apache.flume.
interceptor.RegexExtractorInterceptorMillisSerializer
agent.sources.s1.interceptors.e1.serializers.dt.pattern=dd/MMM/
yyyy:HH:mm:ss Z
agent.sources.s1.interceptors.e1.serializers.dt.name=timestamp
agent.sources.s1.interceptors.e1.serializers.url.name=http_request
agent.sources.s1.interceptors.e1.serializers.sc.name=status_code
agent.sources.s1.interceptors.e1.serializers.bc.name=bytes_xfered
```

This would create the following headers for the preceding sample:

```
{ "ip_address":"192.168.1.42", "timestamp":"1364588829", "http_
request":"GET /index.html HTTP/1.1", "status_code":"200", "bytes_
xfered":"1037" }
```

The body content is unaffected. You'll also notice that I didn't specify `default` for the other type of serializers, as that is the default.

> There is no overwrite checking in this interceptor type. For instance, using the `timestamp` key will overwrite the event's previous time value if there was one.

You can implement your own serializers for this interceptor by implementing the `org.apache.flume.interceptor.RegexExtractorInterceptorSerializer` interface. However, if your goal is to move data from the body of an event to the header, you'll probably want to implement a custom interceptor so that you can alter the body contents in addition to setting the header value, otherwise the data will be effectively duplicated.

To summarize, let's review the properties for this interceptor:

Key	Required	Type	Default
type	Yes	String	regex_extractor
regex	Yes	String	
serializers	Yes	Space-separated list of serializer names	
serializers.NAME.name	Yes	String	

Key	Required	Type	Default
`serializers.NAME.type`	No	Default or FQCN of implementation	`default`
`serializers.NAME.PROP`	No	Serializer-specific properties	

Morphline interceptor

As we saw in *Chapter 4*, *Sinks and Sink Processors*, a powerful library of transformations backs the MorphlineSolrSink from the KiteSDK project. It should come as no surprise that you can also use these libraries in many places where you'd be forced to write a custom interceptor. Similar in configuration to its sink counterpart, you only need to specify the Morphline configuration file, and optionally, the Morphline unique identifier (if the configuration specifies more than one Morphline). Here is a summary table of the Flume interceptor configuration:

Key	Required	Type	Default

`type`	Yes	`String`	`org.apache.flume.`
			`sink.solr.morphline.`
			`MorphlineInterceptor$Builder`

`morphlineFile`	Yes	`String`	
`morphlineId`	No	`String`	Picks the first if not specified and an error if not specified; more than one exists.

```
Your Flume configuration might look something like this:
agent.sources.s1.interceptors=i1 m1
agent.sources.s1.interceptors.i1.type=timestamp
agent.sources.s1.interceptors.m1.type=org.apache.flume.sink.solr.
morphline.MorphlineInterceptor$Builder
agent.sources.s1.interceptors.m1.morphlineFile=/path/to/morph.conf
agent.sources.s1.interceptors.m1.morphlineId=goMorphy
```

In this example, we have specified the `s1` source on the agent named `agent`, which contains two interceptors, `i1` and `m1`, processed in that order. The first interceptor is a standard interceptor that inserts a `timestamp` header if none exists. The second will send the event through the Morphline processor specified by the Morphline configuration file corresponding to the `goMorphy` ID.

Events processed as an interceptor must only output one event for every input event. If you need to output multiple records from a single Flume event, you must do this in the MorphlineSolrSink. With interceptors, it is strictly one event in and one event out.

The first command will most likely be the `readLine` command (or `readBlob`, `readCSV`, and so on) to convert the event into a Morphline Record. From there, you run any other Morphline commands you like with the final Record at the end of the Morphline chain being converted back into a Flume event. We know from *Chapter 4*, *Sinks and Sink Processors*, that the `_attachment_body` special Record key should be `byte[]` (the same as the event's body). This means that the last command needs to do the conversion (such as `toByteArray` or `writeAvroToByteArray`). All other Record keys are converted to String values and set as headers with the same keys.
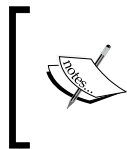
> Take a look at the reference guide for a complete list of Morphline commands, their properties and usage information at `http://kitesdk.org/docs/current/kite-morphlines/morphlinesReferenceGuide.html`

The Morphline configuration file syntax has already been discussed in detail in the *Morphline configuration file* section in *Chapter 4*, *Sinks and Sink Processors*.

Custom interceptors

If there is one piece of custom code you will add to your Flume implementation, it will most likely be a custom interceptor. As mentioned earlier, you implement the `org.apache.flume.interceptor.Interceptor` interface and the associated `org.apache.flume.interceptor.Interceptor.Builder` interface.

Let's say I needed to URLCode my event body. The code would look something like this:

```
public class URLDecode implements Interceptor {

 public void initialize() {}

 public Event intercept(Event event) {
  try {
   byte[] decoded = URLDecoder.decode(new String(event.getBody()),
"UTF-8").getBytes("UTF-8");
   event.setBody(decoded);
```

```
  } catch UnsupportedEncodingException e) {
   // Shouldn't happen. Fall through to unaltered event.
  }
  return event;
 }

 public List<Event> intercept(List<Event> events) {
  for (Event event:events) {
   intercept(event);
  }
  return events;
 }

 public void close() {}

 public static class Builder implements Interceptor.Builder {
  public Interceptor build() {
   return new URLDecode();
  }
  public void configure(Context context) {}
 }
}
```

Then, to configure my new interceptor, use the fully qualified class name for the `Builder` class as the type:

```
agent.sources.s1.interceptors=i1
agent.sources.s1.interceptors.i1.type=com.example.URLDecoder$Builder
```

For more examples of how to pass and validate properties, look at any of the existing interceptor implementations in the Flume source code.

Keep in mind that any heavy processing in your custom interceptor can affect the overall throughput, so be mindful of object churn or computationally intensive processing in your implementations.

The plugins directory

Custom code (sources, interceptors, and so on) can always be installed alongside the Flume core classes in the `$FLUME_HOME/lib` directory. You could also specify additional paths for `CLASSPATH` on startup by way of the `flume-env.sh` shell script (which is often sourced at startup time when using a packaged distribution of Flume). Starting with Flume 1.4, there is now a command-line option to specify a directory that contains the custom code. By default, if not specified, the `$FLUME_HOME/plugins.d` directory is used, but you can override this using the `--plugins-path` command-line parameter.

Within this directory, each piece of custom code is separated into a subdirectory whose name is of your choosing—pick something easy for you to keep track of things. In this directory, you can include up to three subdirectories: `lib`, `libext`, and `native`.

The `lib` directory should contain the JAR file for your custom component. Any dependencies it uses should be added to the `libext` subdirectory. Both of these paths are added to the Java `CLASSPATH` variable at startup.

> The Flume documentation implies that this directory separation by component allows for conflicting Java libraries to coexist. In truth, this is not possible unless the underlying implementation makes use of different class loaders within the JVM. In this case, the Flume startup code simply appends all of these paths to the startup `CLASSPATH` variable, so the order in which the subdirectories are processed will determine the precedence. You cannot even be guaranteed that the subdirectories will be processed in a lexicographic order, as the underlying bash shell `for` loop can give no such guarantees. In practice, you should always try and avoid conflicting dependencies. Code that depends too much on ordering tends to be buggy, especially if your classpath reorders itself from server installation to server installation.

The third directory, `native`, is where you put any native libraries associated with your custom component. This path, if it exists, gets added to `LD_LIBRARY_PATH` at startup so that the JVM can locate these native components.

So, if I had three custom components, my directory structure might look something like this:

$FLUME_HOME/plugins.d/base64-enc/lib/base64interceptor.jar

$FLUME_HOME/plugins.d/base64-enc/libext/base64-2.0.0.jar

$FLUME_HOME/plugins.d/base64-enc/native/libFoo.so

$FLUME_HOME/plugins.d/uuencode/lib/uuEncodingInterceptor.jar

```
$FLUME_HOME/plugins.d/my-avro-serializer/myAvroSerializer.jar
```

```
$FLUME_HOME/plugins.d/my-avro-serializer/native/libAvro.so
```

Keep in mind that all these paths get mashed together at startup, so conflicting versions of libraries should be avoided. This structure is an organization mechanism for ease of deployment for humans. Personally, I have not used it yet, as I use Chef (or Puppet) to install custom components on my Flume agents directly into `$FLUME_HOME/lib`, but if you prefer to use this mechanism, it is available.

Tiering flows

In *Chapter 1*, *Overview and Architecture*, we talked about tiering your data flows. There are several reasons for you to want to do this. You may want to limit the number of Flume agents that directly connect to your Hadoop cluster, to limit the number of parallel requests. You may also lack sufficient disk space on your application servers to store a significant amount of data while you are performing maintenance on your Hadoop cluster. Whatever your reason or use case, the most common mechanism to chain Flume agents is to use the Avro source/sink pair.

The Avro source/sink

We covered Avro a bit in *Chapter 4*, *Sinks and Sink Processors*, when we discussed how to use it as an on-disk serialization format for files stored in HDFS. Here, we'll put it to use in communication between Flume agents. A typical configuration might look something like this:
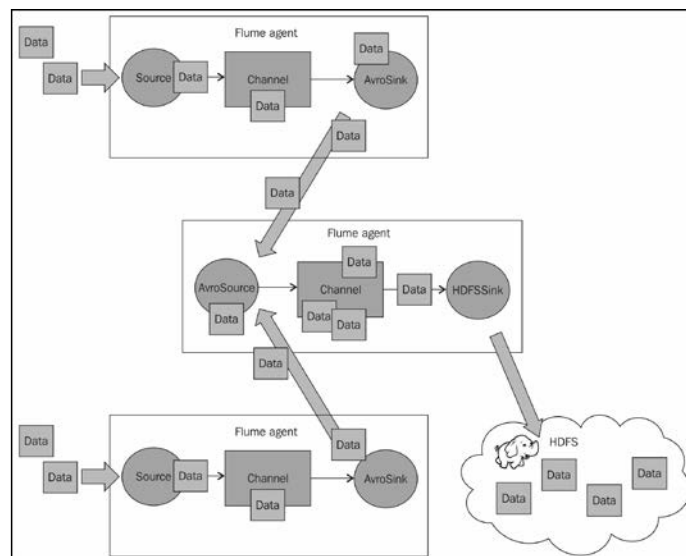
To use the Avro source, you specify the `type` property with a value of `avro`. You need to provide a bind address and port number to listen on:

```
collector.sources=av1
collector.sources.av1.type=avro
collector.sources.av1.bind=0.0.0.0
collector.sources.av1.port=42424
collector.sources.av1.channels=ch1
collector.channels=ch1
collector.channels.ch1.type=memory
collector.sinks=k1
collector.sinks.k1.type=hdfs
collector.sinks.k1.channel=ch1
collector.sinks.k1.hdfs.path=/path/in/hdfs
```

Here, we have configured the agent in the middle that listens on port 42424, uses a memory channel, and writes to HDFS. I've used the memory channel for brevity in this example configuration. Also note that I've given this agent a different name, `collector`, just to avoid confusion.

The agents on the top and bottom sides feeding the `collector` tier might have a configuration similar to this. I have left the sources off this configuration for brevity:

```
client.channels=ch1
client.channels.ch1.type=memory
client.sinks=k1
client.sinks.k1.type=avro
client.sinks.k1.channel=ch1
client.sinks.k1.hostname=collector.example.com
client.sinks.k1.port=42424
```

The hostname, `collector.example.com`, has nothing to do with the agent name on this machine; it is the hostname (or you can use an IP) of the target machine with the receiving Avro source. This configuration, named `client`, would be applied to both agents on the top and bottom sides, assuming both had similar source configurations.

As I don't like single points of failure, I would configure two collector agents with the preceding configuration and instead, set each client agent to round robin between the two, using a sink group. Again, I've left off the sources for brevity:

```
client.channels=ch1
client.channels.ch1.type=memory
client.sinks=k1 k2
client.sinks.k1.type=avro
client.sinks.k1.channel=ch1
client.sinks.k1.hostname=collectorA.example.com
client.sinks.k1.port=42424
client.sinks.k2.type=avro
client.sinks.k2.channel=ch1
client.sinks.k2.hostname=collectorB.example.com
client.sinks.k2.port=42424
client.sinkgroups=g1
client.sinkgroups.g1=k1 k2
client.sinkgroups.g1.processor.type=load_balance
client.sinkgroups.g1.processor.selector=round_robin
client.sinkgroups.g1.processor.backoff=true
```

There are four additional properties associated with the Avro sink that you may need to adjust from their sensible defaults.

The first is the `batch-size` property, which defaults to 100. In heavy loads, you may see better throughput by setting this higher, for example:

```
client.sinks.k1.batch-size=1024
```

The next two properties control network connection timeouts. The `connect-timeout` property, which defaults to 20 seconds (specified in milliseconds), is the amount of time required to establish a connection with an Avro source (receiver). The related `request-timeout` property, which also defaults to 20 seconds (specified in milliseconds), is the amount of time for a sent message to be acknowledged. If I wanted to increase these values to 1 minute, I could add these additional properties:

```
client.sinks.k1.connect-timeout=60000
client.sinks.k1.request-timeout=60000
```

Finally, the `reset-connection-interval` property can be set to force connections to reestablish themselves after some time period. This can be useful when your sink is connecting through a VIP (Virtual IP Address) or hardware load balancer to keep things balanced, as services offered behind the VIP may change over time due to failures or changes in capacity. By default, the connections will not be reset except in cases of failure. If you wanted to change this so that the connections reset themselves every hour, for example, you can specify this by setting this property with the number of seconds, as follows:

```
client.sinks.k1.reset-connection-interval=3600
```

Compressing Avro

Communication between the Avro source and sink can be compressed by setting the `compression-type` property to `deflate`. On the Avro sink, you can additionally, set the `compression-level` property to a number between 1 and 9, with the default being 6. Typically, there are diminishing returns at higher compression levels, but testing may prove a nondefault value that works better for you. Clearly, you need to weigh the additional CPU costs against the overhead to perform a higher level of compression. Typically, the default is fine.

Compressed communications are especially important when you are dealing with high latency networks, such as sending data between two data centers. Another common use case for compression is where you are charged for the bandwidth consumed, such as most public cloud services. In these cases, you will probably choose to spend CPU cycles, compressing and decompressing your flow rather than sending highly compressible data uncompressed.

> It is very important that if you set the `compression-type` property in a source/sink pair, you set this property at both ends. Otherwise, a sink could be sending data the source can't consume.

Continuing the preceding example, to add compression, you would add these additional property fields on both agents:

```
collector.sources.av1.compression-type=deflate
client.sinks.k1.compression-type=deflate
client.sinks.k1.compression-level=7
client.sinks.k2.compression-type=deflate
client.sinks.k2.compression-level=7
```

SSL Avro flows

Sometimes, communication is of a sensitive nature or may traverse untrusted network paths. You can encrypt your communications between an Avro sink and an Avro source by setting the ssl property to true. Additionally, you will need to pass SSL certificate information to the source (the receiver) and optionally, the sink (the sender).

I won't claim to be an expert in SSL and SSL certificates, but the general idea is that a certificate can either be self-signed or signed by a trusted third party such as **VeriSign** (called a certificate authority or CA). Generally, because of the cost associated with getting a verified certificate, people only do this for web browser certificates a customer might see in a web browser. Some organizations will have an internal CA who can sign certificates. For the purpose of this example, we'll be using self-signed certificates. This basically means that we'll generate a certificate but not have it signed by any authority. For this, we'll use the keytool utility that comes with all Java installations (the reference can be found at http://docs.oracle.com/javase/7/docs/technotes/tools/solaris/keytool.html). Here, I create a **Java Key Store** (**JKS**) file that contains a 2048 bit key. When prompted for a password, I'll use the password string. Make sure you use a real password in your nontest environments:

```
% keytool -genkey -alias flumey -keyalg RSA -keystore keystore.jks
-keysize 2048
Enter keystore password: password
Re-enter new password: password
What is your first and last name?
  [Unknown]: Steve Hoffman
What is the name of your organizational unit?
  [Unknown]: Operations
What is the name of your organization?
  [Unknown]: Me, Myself and I
What is the name of your City or Locality?
  [Unknown]: Chicago
What is the name of your State or Province?
  [Unknown]: Illinois
What is the two-letter country code for this unit?
  [Unknown]: US
Is CN=Steve Hoffman, OU=Operations, O="Me, Myself and I", L=Chicago,
ST=Illinois, C=US correct?
```

```
 [no]: yes
```

```
Enter key password for <flumey>
  (RETURN if same as keystore password):
```

I can verify the contents of the JKS by running the `list` subcommand:

```
% keytool -list -keystore keystore.jks
Enter keystore password: password

Keystore type: JKS
Keystore provider: SUN

Your keystore contains 1 entry

flumey, Nov 11, 2014, PrivateKeyEntry,
Certificate fingerprint (SHA1):
5C:BC:3C:7F:7A:E7:77:EB:B5:54:FA:E2:8B:DD:D3:66:36:86:DE:E4
```

Now that I have a key in my keystore file, I can set the additional properties on the receiving source:

```
collector.sources.av1.ssl=true
collector.sources.av1.keystore=/path/to/keystore.jks
collector.sources.av1.keystore-password=password
```

As I am using a self-signed certificate, the sink won't be sure that communications can be trusted, so I have two options. The first is to tell it to just trust all certificates. Clearly, you would only do this on a private network that has some reasonable assurances about its security. To ignore the dubious origin of my certificate, I can set the `trust-all-certs` property to `true` as follows:

```
client.sinks.k1.ssl=true
client.sinks.k1.trust-all-certs=true
```

If you didn't set this, you'd see something like this in the logs:

```
org.apache.flume.EventDeliveryException: Failed to send events
...
Caused by: javax.net.ssl.SSLHandshakeException: General SSLEngine
problem
...
Caused by: sun.security.validator.ValidatorException: No trusted
certificate found
...
```

For communications over the public Internet or in multitenant cloud environments, more care should be taken. In this case, you can provide a truststore to the sink. A truststore is similar to a keystore, except that it contains certificate authorities you are telling Flume can be trusted. For well-known certificate authorities, such as VeriSign and others, you don't need to specify a truststore, as their identities are already included in your Java distribution (at least for the major ones). You will need to have your certificate signed by a certificate authority and then add the signed certificate file to the source's keystore file. You will also need to include the signing CA's certificate in the keystore so that the certificate chain can be fully resolved.

Once you have added the key, signed certificate, and signing CA's certificate to the source, you need to configure the sink (the sender) to trust these authorities so that it will pass the validation step during the SSL handshake. To specify the truststore, set the `truststore` and `truststore-password` properties as follows:

```
client.sinks.k1.ssl=true
client.sinks.k1.truststore=/path/to/truststore.jks
client.sinks.k1.truststore-password=password
```

Chances are there is somebody in your organization responsible for obtaining the third-party certificates or someone who can issue an organizational CA-signed-certificate to you, so I'm not going to go into details about how to create your own certificate authority. There is plenty of information on the Internet if you choose to take this route. Remember that certificates have an expiration date (usually, a year), so you'll need to repeat this process every year or communications will abruptly stop when certificates or certificate authorities expire.

The Thrift source/sink

Another source/sink pair you can use to tier your data flows is based on Thrift (`http://thrift.apache.org/`). Unlike Avro data, which is self-documenting, Thrift uses an external schema, which can be compiled into just about every programming language on the planet. The configuration is almost identical to the Avro example already covered. The preceding `collector` configuration that uses Thrift would now look something like this:

```
collector.sources=th1
collector.sources.th1.type=thrift
collector.sources.th1.bind=0.0.0.0
collector.sources.th1.port=42324
collector.sources.th1.channels=ch1
collector.channels=ch1
```

```
collector.channels.ch1.type=memory
collector.sinks=k1
collector.sinks.k1.type=hdfs
collector.sinks.k1.channel=ch1
collector.sinks.k1.hdfs.path=/path/in/hdfs
```

There is one additional property that sets the maximum worker threads in the underlying thread pool. If unset, a value of zero is assumed, which makes the thread pool unbounded, so you should probably set this in your production configurations. Testing should provide you with a reasonable upper limit for your environment. To set the thread pool size to 10, for example, you would add the `threads` property:

```
collector.source.th1.threads=10
```

The "client" agents would use the corresponding Thrift sink as follows:

```
client.channels=ch1
client.channels.ch1.type=memory
client.sinks=k1
client.sinks.k1.type=thrift
client.sinks.k1.channel=ch1
client.sinks.k1.hostname=collector.example.com
client.sinks.k1.port=42324
```

Like its Avro counterpart, there are additional settings for the batch size, connection timeouts, and connection reset intervals that you may want to adjust based on your testing results. Refer to the *The Avro source/sink* section earlier in this chapter for details.

Using command-line Avro

The Avro source can also be used in conjunction with one of the command-line options you may have noticed back in *Chapter 2*, *A Quick Start Guide to Flume*. Rather than running `flume-ng` with the `agent` parameter, you can pass the `avro-client` parameter to send one or more files to an Avro source. These are the options specific to `avro-client` from the help text:

```
avro-client options:
 --dirname <dir>    directory to stream to avro source
 --host,-H <host>   hostname to which events will be sent (required)
 --port,-p <port>   port of the avro source (required)
 --filename,-F <file> text file to stream to avro source [default: std
input]
```

```
 --headerFile,-R <file> headerFile containing headers as key/value pairs
on each new line
 --help,-h        display help text
```

This variation is very useful for testing, resending data manually due to errors, or importing older data stored elsewhere.

Just like an Avro sink, you have to specify the hostname and port you will be sending data to. You can send a single file with the `--filename` option or all the files in a directory with the `--dirname` option. If you specify neither of these, `stdin` will be used. Here is how you might send a file named `foo.log` to the Flume agent we previously configured:

```
$ ./flume-ng avro-client --filename foo.log --host collector.example.com
--port 42424
```

Each line of the input will be converted into a single Flume event.

Optionally, you can specify a file containing key/value pairs to set Flume header values. The file uses Java property file syntax. Suppose I had a file named `headers.properties` containing:

```
pointOfSale=US
environment=staging
```

Then, including the `--headerFile` option would set these two headers on every event created:

```
$ ./flume-ng avro-client --filename foo.log --headerFile headers.
properties --host collector.example.com --port 42424
```

The Log4J appender

As we discussed in *Chapter 5*, *Sources and Channel Selectors*, there are issues that may arise from using a filesystem file as a source. One way to avoid this problem is to use the Flume Log4J Appender in your Java application(s). Under the hood, it uses the same Avro communication that the Avro sink uses, so you need only configure it to send data to an Avro source.

The Appender has two properties, which are shown here in XML:

```
<appender name="FLUME" class="org.apache.flume.clients.log4jappender.
Log4jAppender">
  <param name="Hostname" value="collector.example.com"/>
  <param name="Port" value="42424"/>
</appender>
```

The format of the body will be dictated by the Appender's configured layout (not shown). The log4j fields that get mapped to Flume headers are summarized in this table:

Flume header key	Log4J logging event field
`flume.client.log4j.logger.name`	`event.getLoggerName()`
`flume.client.log4j.log.level`	`event.getLevel()` as a number. See `org.apache.log4j.Level` for mappings.
`flume.client.log4j.timestamp`	`event.getTimeStamp()`
`flume.client.log4j.message.encoding`	N/A—always `UTF8`
`flume.client.log4j.logger.other`	Will only see this if there was a problem mapping one of the above fields, so normally, this won't be present.

Refer to `http://logging.apache.org/log4j/1.2/` for more details on using Log4J.

You will need to include the `flume-ng-sdk` JAR in the classpath of your Java application at runtime to use Flume's Log4J Appender.

Keep in mind that if there is a problem sending data to the Avro source, the appender will throw an exception and the log message will be dropped, as there is no place to put it. Keeping it in memory could quickly overload your JVM heap, which is usually considered worse than dropping the data record.

The Log4J load-balancing appender

I'm sure you noticed that the preceding Log4j Appender only has a single hostname/port in its configuration. If you wanted to spread the load across multiple collector agents, either for additional capacity or for fault tolerance, you can use the `LoadBalancingLog4jAppender`. This appender has a single required property named `Hosts`, which is a space-separated list of hostnames and port numbers separated by a colon, as follows:

```
<appender name="FLUME" class="org.apache.flume.clients.log4jappender.
LoadBalancingLog4jAppender">
 <param name="Hosts" value="server1:42424 server2:42424"/>
</appender>
```

There is an optional property, `Selector`, which specifies the method that you want to load balance. Valid values are `RANDOM` and `ROUND_ROBIN`. If not specified, the default is `ROUND_ROBIN`. You can implement your own selector, but that is outside the scope of this book. If you are interested, go have a look at the well-documented source code for the `LoadBalancingLog4jAppender` class.

Finally, there is another optional property to override the maximum time for exponential back off when a server cannot be contacted. Initially, if a server cannot be contacted, 1 second will need to pass before that server is tried again. Each time the server is unavailable, the retry time doubles, up to a default maximum of 30 seconds. If we wanted to increase this maximum to 2 minutes, we can specify a `MaxBackoff` property in milliseconds as follows:

```
<appender name="FLUME" class="org.apache.flume.clients.log4jappender.
LoadBalancingLog4jAppender">
 <param name="Hosts" value="server1:42424 server2:42424"/>
 <param name="Selector" value="RANDOM"/>
 <param name="MaxBackoff" value="120000"/>
</appender>
```

In this example, we have also overridden the default `round_robin` selector to use a random selection.

The embedded agent

If you are writing a Java program that creates data, you may choose to send the data directly as structured data using a special mode of Flume called the **Embedded Agent**. It is basically a simple single source/single channel Flume agent that you run inside your JVM.

There are benefits and drawbacks to this approach. On the positive side, you don't need to monitor an additional process on your servers to relay data. The embedded channel also allows for the data producer to continue executing its code immediately after queuing the event to the channel. The `SinkRunner` thread handles taking events from the channel and sending them to the configured sinks. Even if you didn't use embedded Flume to perform this handoff from the calling thread, you would most likely use some kind of synchronized queue (such as `BlockingQueue`) to isolate the sending of the data from the main execution thread. Using Embedded Flume provides the same functionality without having to worry whether you've written your multithreaded code correctly.

The major drawback to embedding Flume in your application is added memory pressure on your JVM's garbage collector. If you are using an in-memory channel, any unsent events are held in the heap and get in the way of cheap garbage collection by way of short-lived objects. However, this is why you set a channel size: to keep the maximum memory footprint to a known quantity. Furthermore, any configuration changes will require an application restart, which can be problematic if your overall system doesn't have sufficient redundancy built in to tolerate restarts.

Configuration and startup

Assuming you are not dissuaded (and you shouldn't be), you will first need to include the `flume-ng-embedded-agent` library (and dependencies) into your Java project. Depending on what build system you are using (Maven, Ivy, Gradle, and so on), the exact format will differ, so I'll just show you the Maven configuration here. You can look up the alternative formats at `http://mvnrepository.com/`:

```
<dependency>
  <groupId>org.apache.flume</groupId>
  <artifactId>flume-ng-embedded-agent</artifactId>
  <version>1.5.2</version>
</dependency>
```

Start by creating an `EmbeddedAgent` object in your Java code by calling the constructor (and passing a string name—used only in error messages):

```
EmbeddedAgent toHadoop = new EmbeddedAgent("myData");
```

Next, you have to set properties for the channel, sinks, and sink processor via the `configure()` method, as shown in this example. Most of this configuration should look very familiar to you at this point:

```
Map<String,String> config = new HashMap<String, String>;
config.put("channel.type","memory");
config.put("channel.capacity", "75");
config.put("sinks", "s1 s2");
config.put("sink.s1.type", "avro");
config.put("sink.s1.hostname", "foo.example.com");
config.put("sink.s1.port", "12345");
config.put("sink.s1.compression-type", "deflate");
config.put("sink.s2.type", "avro");
```

```
config.put("sink.s2.hostname", "bar.example.com");
config.put("sink.s2.port", "12345");
config.put("sink.s2.compression-type", "deflate");
config.put("processor.type", "failover");
config.put("processor.priority.s1", "10");
config.put("processor.priority.s2", "20");
toHadoop.configure(config);
```

Here, we define a memory channel with a capacity of 75 events along with two Avro sinks in an active/standby configuration (first, to `foo.example.com`, and if that fails, to `bar.example.com`) using Avro serialization with compression. Refer to *Chapter 3*, *Channels*, for specific settings for memory- or file-backed channel properties.

The sink processor only comes into play if you have more than one sink defined in your `sinks` property. Unlike the optional sink groups options covered in *Chapter 4*, *Sinks and Sink Processors*, you need to specify a sink list, even if it is just one. Clearly, there is no behavior difference between failover and load balance when there is only once sink. Refer to each specific sink's properties in *Chapter 4*, *Sinks and Sink Processors*, for specific configuration parameters.

Finally, before you start using this agent, you need to call the `start()` method to instantiate everything based on your properties and start all the background processing threads as follows:

```
toHadoop.start();
```

The class is now ready to start receiving and forwarding data.

You'll want to keep a reference to this object around for cleanup later, as well as to pass it to other objects that will be sending data as the underlying channel provides thread safety. Personally, I use Spring Framework's dependency injection to configure and pass references at startup rather than doing it programmatically, as I've shown in this example. Refer to the Spring website for more information (`http://spring.io/`), as proper use of Spring is a whole book unto itself, and it is not the only dependency injection framework available to you.

Sending data

Data can be sent either as single events or in batches. Batches can sometimes be more efficient in high volume data streams.

To send a single event, just call the `put()` method, as shown in this example:

```
Event e = EventBuilder.withBody("Hello Hadoop", Charset.
forName("UTF8"));
toHadoop.put(e);
```

Here, I'm using one of many methods available on the `org.apache.flume.event.` `EventBuilder` class. Here is a complete list of methods you can use to construct events with this helper class:

```
EventBuilder.withBody(byte[] body, Map<String, String> headers);
EventBuilder.withBody(byte[] body);
EventBuilder.withBody(String body, Charset charset, Map<String,
String> headers);
EventBuilder.withBody(String body, Charset charset);
```

In order to send several events in one batch, you can call the `putAll()` method with a list of events, as shown in this example, which also includes a time header:

```
Map<String,String> headers = new HashMap<String,String>;
headers.put("timestamp",Long.toString(System.currentTimeMillis());
List<Event> events = new ArrayList<Event>;
events.add(EventBuilder.withBody("First".getBytes("UTF-8"), headers);
events.add(EventBuilder.withBody("Second".getBytes("UTF-8"), headers);
toHadoop.putAll(events);
```

As interceptors are not supported, you will need to add any headers you want to add to the events programmatically in Java before you call `put()` or `putAll()`.

Shutdown

Finally, as there are background threads waiting for data to arrive on your embedded channel which are most likely holding persistent connections to configured destinations, when it is time to shut down your application, you'll want to have Flume do its cleanup as well. You simply call the `stop()` method on the configured Embedded Agent as follows:

```
toHadoop.stop();
```

Routing

The routing of data to different destinations based on content should be fairly straightforward now that you've been introduced to all the various mechanisms in Flume.

The first step is to get the data you want to switch on into a Flume header by means of a source-side interceptor if the header isn't already available. The second step is to use a Multiplexing Channel Selector on that header value to switch the data to an alternate channel.

For instance, let's say you wanted to capture all exceptions to HDFS. In this configuration, you can see events coming in on the s1 source via avro on port 42424. The event is tested to see whether the body contains the text Exception. If it does, it creates an exception header key (with the value of Exception). This header is used to switch these events to channel c1, and ultimately, HDFS. If the event didn't match the pattern, it would not have the exception header and would get passed to the c2 channel via the default selector where it would be forwarded via Avro serialization to port 12345 on server foo.example.com:

```
agent.sources=s1
agent.sources.s1.type=avro
agent.sources.s1.bind=0.0.0.0
agent.sources.s1.port=42424
agent.sources.s1.interceptors=i1
agent.sources.s1.interceptors.i1.type=regex_extractor
agent.sources.s1.interceptors.i1.regex=(Exception)
agent.sources.s1.interceptors.i1.serializers=ex
agent.sources.s1.intercetpros.i1.serializers.ex.name=exception
agent.sources.s1.selector.type=multiplexing
agent.sources.s1.selector.header=exception
agent.sources.s1.selector.mapping.Exception=c1
agent.sources.s1.selector.default=c2
agent.channels=c1 c2
agent.channels.c1.type=memory
agent.channels.c2.type=memory
agent.sinks=k1 k2
agent.sinks.k1.type=hdfs
agent.sinks.k1.channel=c1
agent.sinks.k1.hdfs.path=/logs/exceptions/%y/%M/%d/%H
agent.sinks.k2.type=avro
agent.sinks.k2.channel=c2
agent.sinks.k2.hostname=foo.example.com
agent.sinks.k2.port=12345
```

Summary

In this chapter, we covered various interceptors shipped with Flume, including:

- Timestamp: These are used to add a timestamp header, possibly overwriting an existing one.
- Host: This is used to add the Flume agent hostname or IP as a header in the event.
- Static: This is used to add static String headers.
- Regular expression filtering: This is used to include or exclude events based on a matched regular expression.
- Regular expression extractor: This is used to create headers from matched regular expressions. It's useful for routing with Channel Selectors.
- Morphline: This is used to delegate transformation to a Morphline command chain.
- Custom: This is used to create any custom transformations you need that you can't find elsewhere.

We also covered tiering data flows using the Avro source and sink. Optional compression and SSL with Avro flows were covered as well. Finally, Thrift sources and sinks were briefly covered, as some environments may already have Thrift data flows to integrate with.

Next, we introduced two Log4J Appenders, a single path and a load-balancing version, for direct integration with Java applications.

The Embedded Flume Agent was covered for those wishing to directly integrate basic Flume functionality into their Java applications.

Finally, we gave you an example of using interceptors in conjunction with a Channel Selector to provide routing decision logic.

In the next chapter, we will dive into an example to stitch together everything we have covered so far.

7
Putting It All Together

Now that we've walked through all the components and configurations, let's put together a working end-to-end configuration. This example is by no means exhaustive, nor does it cover every possible scenario you might need, but I think it should cover a couple of common use cases I've seen over and over:

- Finding errors by searching logs across multiple servers in near real time
- Streaming data to HDFS for long-term batch processing

In the first situation, your systems may be impaired, and you have multiple places where you need to search for problems. Bringing all of those logs to a single place that you can search means getting your systems restored quickly. In the second scenario, you are interested in capturing data in the long term for analytics and machine learning.

Web logs to searchable UI

Let's simulate a web application by setting up a simple web server whose logs we want to stream into some searchable application. In this case, we'll be using a Kibana UI to perform ad hoc queries against Elasticsearch.

For this example, I'll start three servers in Amazon's **Elastic Compute Cluster** (**EC2**), as shown in this diagram:
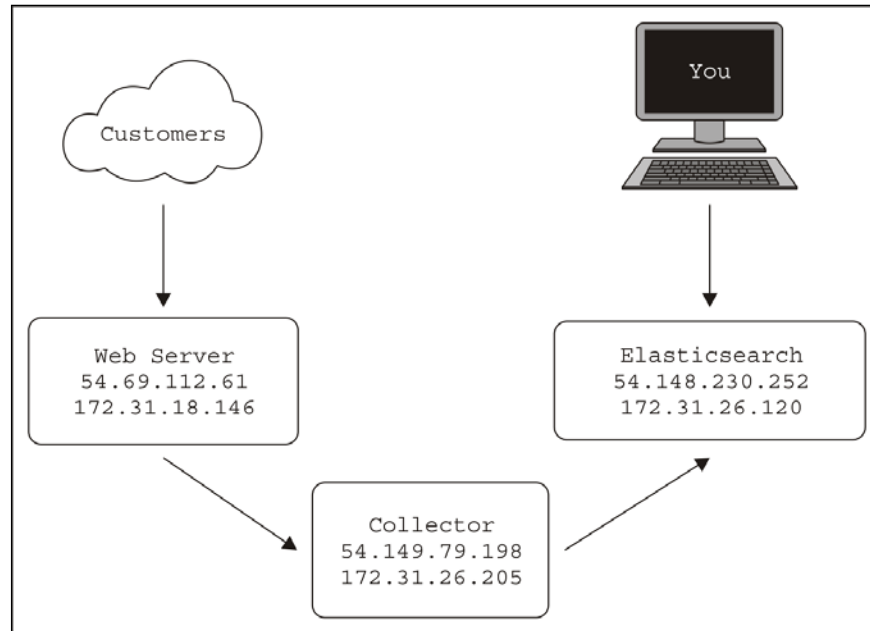
Each server has a public IP (starting with `54`) and a private IP (starting with `172`). For interserver communication, I'll be using the private IPs in my configurations. My personal interaction with the web server (to simulate traffic) and with Kibana and Elasticsearch will require the public IPs, since I'm sitting in my house and not in Amazon's data centers.
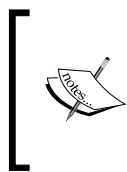
> Pay careful attention to the IP addresses in the shell prompts, as we will be jumping from machine to machine, and I don't want you to get lost. For instance, on the Collector box, the prompt will contain its private IP:
>
> `[ec2-user@ip-172-31-26-205 ~]$`

If you try this out yourself in EC2, you will get different IP assignments, so adjust the configuration files and URLs referenced as needed. For this example, I'll be using Amazon's Linux AMI for an operating system and the `t1.micro` size server. Also, be sure to adjust the security groups to allow network traffic between these servers (and yourself). When in doubt about connectivity, use utilities such as `telnet` or `nc` to test connections between servers and ports.

> While going through this exercise, I made mistakes in my security
> groups more than once, so perform these tests on connectivity
> exceptions.

Setting up the web server

Let's start by setting up the web server. For this, we'll install the popular Nginx web
server (`http://nginx.org`). Since web server logs are pretty much standardized
nowadays, I could have chosen any web server, but the point here is that this
application writes it logs (by default) to a file on the disk. Since I've warned you
from trying to use the tail program to stream them, we'll be using a combination of
`logrotate` and the Spooling Directory Source to ingest the data. First, let's log in to
the server. The default account with `sudo` for an Amazon Linux server is `ec2-user`.
You'll need to pass this in your `ssh` command:

```
% ssh 54.69.112.61 -l ec2-user
The authenticity of host '54.69.112.61 (54.69.112.61)' can't be
established.
RSA key fingerprint is dc:ee:7a:1a:05:e1:36:cd:a4:81:03:97:48:8d:b3:cc.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '54.69.112.61' (RSA) to the list of known
hosts.


     __|  __|_  )
     _|  (     /   Amazon Linux AMI
    ___|\___|___|


https://aws.amazon.com/amazon-linux-ami/2014.09-release-notes/
18 package(s) needed for security, out of 41 available
Run "sudo yum update" to apply all updates.
```

Now that you are logged in to the server, let's install Nginx (I'm not going to show
all of the output to save paper):

```
[ec2-user@ip-172-31-18-146 ~]$ sudo yum -y install nginx
Loaded plugins: priorities, update-motd, upgrade-helper
Resolving Dependencies
--> Running transaction check
---> Package nginx.x86_64 1:1.6.2-1.22.amzn1 will be installed
[SNIP]
```

```
Installed:
  nginx.x86_64 1:1.6.2-1.22.amzn1


Dependency Installed:
  GeoIP.x86_64 0:1.4.8-1.5.amzn1       gd.x86_64 0:2.0.35-11.10.amzn1
gperftools-libs.x86_64 0:2.0-11.5.amzn1     libXpm.x86_64 0:3.5.10-
2.9.amzn1
  libunwind.x86_64 0:1.1-2.1.amzn1


Complete!
```

Next, let's start the Nginx web server:

```
[ec2-user@ip-172-31-18-146 ~]$ sudo /etc/init.d/nginx start
```

```
Starting nginx:                                           [  OK  ]
```

At this point, the web server should be running, so let's use our computer's web browser to go to the public IP, `http://54.69.112.61/` in this case. If it is working, you should see the default welcome page, like this:
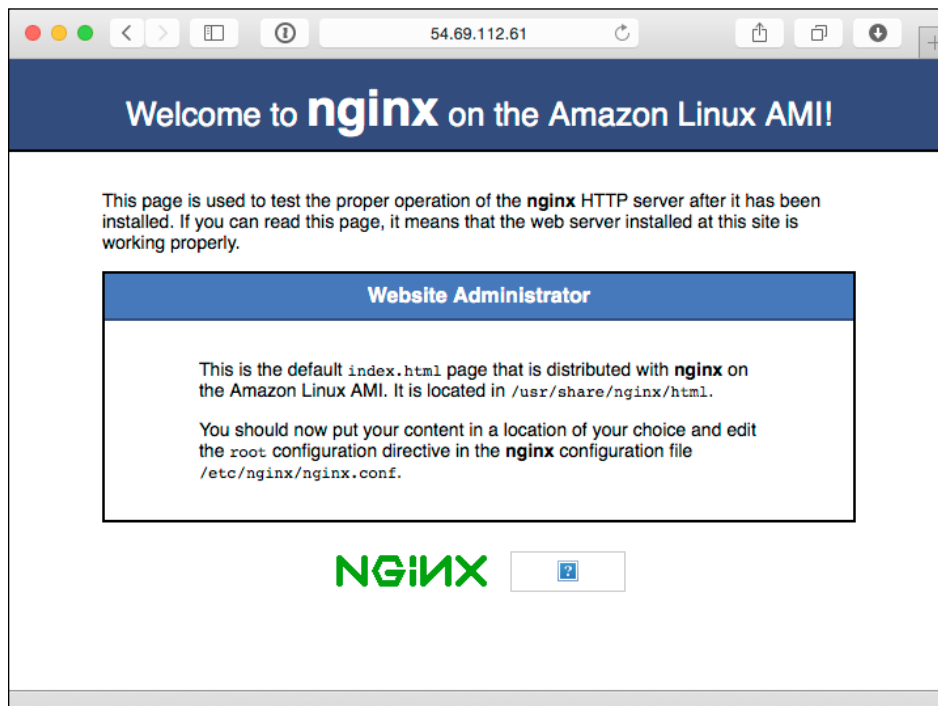
To help generate a sample input, you can use a load generator program such as Apache benchmark (`http://httpd.apache.org/docs/2.4/programs/ab.html`) or wrk (`https://github.com/wg/wrk`). Here is the command I used to generate data for 20 minutes at a time:

```
% wrk -c 2 -d 20m http://54.69.112.61
Running 20m test @ http://54.69.112.61
  2 threads and 2 connections
```

Configuring log rotation to the spool directory

By default, the server's access logs are written to `/var/log/nginx/access.log`. The problem we need to overcome here is to move the files to a separate directory while it is still open by the `nginx` process. This is where `logrotate` comes into the picture. Let's first create a spool directory that will become the input path for the Spooling Directory Source later on. For the purpose of this example, I'll just create the spool directory in the home directory of `ec2-user`. In a production environment, you would place the spool directory somewhere else:

```
[ec2-user@ip-172-31-18-146 ~]$ pwd
/home/ec2-user
[ec2-user@ip-172-31-18-146 ~]$ mkdir spool
```

Let's also create a `logrotate` script in the home directory of `ec2-user`, called `rotateAccess.conf`. Open an editor and paste these contents (after the prompt) in to a file called `accessRotate.conf`:

```
[ec2-user@ip-172-31-18-146 ~]$ cat accessRotate.conf
/var/log/nginx/access.log {
    missingok
    notifempty
    rotate 0
    copytruncate
    sharedscripts
    olddir /home/ec2-user/spool
    postrotate
        chown ec2-user:ec2-user /home/ec2-user/spool/access.log.1
        ts=$(date +%s)
        mv /home/ec2-user/spool/access.log.1 /home/ec2-user/spool/access.
log.$ts
    endscript
}
```

Let's go over this so that you understand what it is doing. This is by no means a complete introduction to the `logrotate` utility. Read the online documentation at `http://linuxconfig.org/logrotate` for more details.

When `logrotate` runs, it will copy `/var/log/nginx/access.log` to the `/home/ec2-user/spool` directory, but only if it has a nonzero length (meaning, there is data to send). The `rotate 0` command tells `logrotate` not to remove any files from the target directory (since the Flume source will do that *after* it has successfully transmitted each file). We'll make use of the `copytruncate` feature because it keeps the file open as far as the `nginx` process is concerned, while resetting it to zero length. This avoids the need to signal `nginx` that a rotation has just occurred. The destination of the rotated file in the spool directory is `/home/ec2-user/spool/access.log.1`. Next, in the `postrotate` script section, we will change the ownership of the file from the `root` user to `ec2-user` so that the Flume agent can read it, and later, delete it. Finally, we rename the file with a unique timestamp so that when the next rotation occurs, the `access.log.1` file will not exist. This also means that we'll need to add an exclusion rule to the Flume source later to keep it from reading `access.log.1` file until the copying process has completed. Once renamed, it is ready to be consumed by Flume.

Now let's try running it in debug mode, which will be just a dry run so that we can see what it can do. Normally, `logrotate` has a daily rotation setting that prevents it from doing anything unless enough time has passed:

```
[ec2-user@ip-172-31-18-146 ~]$ sudo /usr/sbin/logrotate -d /home/ec2-
user/accessRotate.conf
reading config file /home/ec2-user/accessRotate.conf
reading config info for /var/log/nginx/access.log
olddir is now /home/ec2-user/spool


Handling 1 logs


rotating pattern: /var/log/nginx/access.log  1048576 bytes (no old logs
will be kept)
olddir is /home/ec2-user/spool, empty log files are not rotated, old logs
are removed
considering log /var/log/nginx/access.log
  log does not need rotating
not running postrotate script, since no logs were rotated
```

Therefore, we will also include the `-f` (force) flag to override the preceding functionality:

```
[ec2-user@ip-172-31-18-146 ~]$ sudo /usr/sbin/logrotate -df /home/ec2-
user/accessRotate.conf
reading config file /home/ec2-user/accessRotate.conf
reading config info for /var/log/nginx/access.log
olddir is now /home/ec2-user/spool


Handling 1 logs


rotating pattern: /var/log/nginx/access.log forced from command line (no
old logs will be kept)
olddir is /home/ec2-user/spool, empty log files are not rotated, old logs
are removed
considering log /var/log/nginx/access.log
  log needs rotating
rotating log /var/log/nginx/access.log, log->rotateCount is 0
dateext suffix '-20141207'
glob pattern '-[0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]'
renaming /home/ec2-user/spool/access.log.1 to /home/ec2-user/spool/
access.log.2 (rotatecount 1, logstart 1, i 1),
renaming /home/ec2-user/spool/access.log.0 to /home/ec2-user/spool/
access.log.1 (rotatecount 1, logstart 1, i 0),
copying /var/log/nginx/access.log to /home/ec2-user/spool/access.log.1
truncating /var/log/nginx/access.log
running postrotate script
running script with arg /var/log/nginx/access.log : "
        chown ec2-user:ec2-user /home/ec2-user/spool/access.log.1
        ts=$(date +%s)
        mv /home/ec2-user/spool/access.log.1 /home/ec2-user/spool/access.
log.$ts
"
removing old log /home/ec2-user/spool/access.log.2
```

As you can see, `logrotate` will copy the log to the spool directory with the `.1` extension as expected, followed by our script block at the end to change permissions and rename it with a unique timestamp.

Now let's run it again, but this time for real, without the debug flag, and then we will list the source and target directories to see what happened:

```
[ec2-user@ip-172-31-18-146 ~]$ sudo /usr/sbin/logrotate -f /home/ec2-
user/accessRotate.conf
[ec2-user@ip-172-31-18-146 ~]$ ls -l spool/
total 188
-rw-r--r-- 1 ec2-user ec2-user 189344 Dec  7 17:27 access.log.1417973241
[ec2-user@ip-172-31-18-146 ~]$ ls -l /var/log/nginx/
total 4
-rw-r--r-- 1 root root    0 Dec  7 17:27 access.log
-rw-r--r-- 1 root root 520 Dec  7 16:44 error.log
[ec2-user@ip-172-31-18-146 ~]$
```

You can see the access log is empty and the old contents have been copied to the spool directory, with correct permissions and the filename ending in a unique timestamp.

Let's also verify that the empty access log will result in no action if run without any data. You'll need to include the debug flag to see what the process is thinking:

```
[ec2-user@ip-172-31-18-146 ~]$ sudo /usr/sbin/logrotate -df /home/ec2-
user/accessRotate.conf
reading config file accessRotate.conf
reading config info for /var/log/nginx/access.log
olddir is now /home/ec2-user/spool


Handling 1 logs


rotating pattern: /var/log/nginx/access.log forced from command line (1
rotations)
olddir is /home/ec2-user/spool, empty log files are not rotated, old logs
are removed
considering log /var/log/nginx/access.log
  log does not need rotating
```

Now that we have a working rotation script, we need something to run it periodically (not in debug mode) in some chosen interval. For this, we will use the cron daemon (http://en.wikipedia.org/wiki/Cron) by creating a file in the /etc/cron.d directory:

```
[ec2-user@ip-172-31-18-146 ~]$ cat /etc/cron.d/rotateLogsToSpool
# Move files to spool directory every 5 minutes
*/5 * * * * root /usr/sbin/logrotate -f /home/ec2-user/accessRotate.conf
```

Here, I've indicated a five-minute interval and to run as the `root` user. Since the root user owns the original log file, we need elevated access to reassign ownership to `ec2-user` so that it can be removed later by Flume.

Once you've saved this `cron` configuration file, you should see our script every 5 minutes (as well as other processes), by inspecting the `cron` daemon's log file:

```
[ec2-user@ip-172-31-18-146 ~]$ sudo tail /var/log/cron
Dec  7 17:45:01 ip-172-31-18-146 CROND[22904]: (root) CMD (/usr/sbin/
logrotate -f /home/ec2-user/accessRotate.conf)
Dec  7 17:50:01 ip-172-31-18-146 CROND[22929]: (root) CMD (/usr/sbin/
logrotate -f /home/ec2-user/accessRotate.conf)
Dec  7 17:54:01 ip-172-31-18-146 anacron[2426]: Job 'cron.weekly' started
Dec  7 17:54:01 ip-172-31-18-146 anacron[2426]: Job 'cron.weekly'
terminated
Dec  7 17:55:01 ip-172-31-18-146 CROND[22955]: (root) CMD (/usr/sbin/
logrotate -f /home/ec2-user/accessRotate.conf)
Dec  7 18:00:01 ip-172-31-18-146 CROND[23046]: (root) CMD (/usr/sbin/
logrotate -f /home/ec2-user/accessRotate.conf)
Dec  7 18:01:01 ip-172-31-18-146 CROND[23060]: (root) CMD (run-parts /
etc/cron.hourly)
Dec  7 18:01:01 ip-172-31-18-146 run-parts(/etc/cron.hourly)[23060]:
starting 0anacron
Dec  7 18:01:01 ip-172-31-18-146 run-parts(/etc/cron.hourly)[23069]:
finished 0anacron
Dec  7 18:05:01 ip-172-31-18-146 CROND[23117]: (root) CMD (/usr/sbin/
logrotate -f /home/ec2-user/accessRotate.conf)
```

An inspection of the spool directory also shows new files created on our arbitrarily chosen 5-minute interval:

```
[ec2-user@ip-172-31-18-146 ~]$ ls -l spool/
total 1072
-rw-r--r-- 1 ec2-user ec2-user 145029 Dec  7 18:02 access.log.1417975373
-rw-r--r-- 1 ec2-user ec2-user 164169 Dec  7 18:05 access.log.1417975501
-rw-r--r-- 1 ec2-user ec2-user 166779 Dec  7 18:10 access.log.1417975801
-rw-r--r-- 1 ec2-user ec2-user 613785 Dec  7 18:15 access.log.1417976101
```

Keep in mind that the Nginx RPM package also installed a rotation configuration located at `/etc/logrotate.d/nginx` to perform daily rotations. If you are going to use this example in production, you'll want to remove the configuration, since we don't want it clashing with our more frequently running cron script. I'll leave handling `error.log` as an exercise for you. You'll either want to send it someplace (and have Flume remove it) or rotate it periodically so that your disk doesn't fill up over time.

Setting up the target – Elasticsearch

Now let's move to the other server in our diagram and set up Elasticsearch, our destination for searchable data. I'm going to be using the instructions found at `http://www.elasticsearch.org/overview/elkdownloads/`. First, let's download and install the RPM package:

```
[ec2-user@ip-172-31-26-120 ~]$ wget https://download.elasticsearch.org/
elasticsearch/elasticsearch/elasticsearch-1.4.1.noarch.rpm

--2014-12-07 18:25:35--  https://download.elasticsearch.org/
elasticsearch/elasticsearch/elasticsearch-1.4.1.noarch.rpm

Resolving download.elasticsearch.org (download.elasticsearch.org)...
54.225.133.195, 54.243.77.158, 107.22.222.16, ...

Connecting to download.elasticsearch.org (download.elasticsearch.
org)|54.225.133.195|:443... connected.

HTTP request sent, awaiting response... 200 OK

Length: 26326154 (25M) [application/x-redhat-package-manager]

Saving to: 'elasticsearch-1.4.1.noarch.rpm'


100%[======================================================================
========================================>] 26,326,154  9.92MB/s   in 2.5s


2014-12-07 18:25:38 (9.92 MB/s) - 'elasticsearch-1.4.1.noarch.rpm' saved
[26326154/26326154]


[ec2-user@ip-172-31-26-120 ~]$ sudo rpm -ivh elasticsearch-1.4.1.noarch.
rpm
Preparing...                          ################################# [100%]
Updating / installing...
   1:elasticsearch-1.4.1-1            ################################# [100%]
```

```
### NOT starting on installation, please execute the following statements
to configure elasticsearch to start automatically using chkconfig
 sudo /sbin/chkconfig --add elasticsearch
### You can start elasticsearch by executing
 sudo service elasticsearch start
```

The installation is kind enough to tell me how to configure the service to automatically start on system boot-up, and it also tells me to launch the service now, so let's do that:

```
[ec2-user@ip-172-31-26-120 ~]$ sudo /sbin/chkconfig --add elasticsearch
[ec2-user@ip-172-31-26-120 ~]$ sudo service elasticsearch start
Starting elasticsearch:                                    [  OK  ]
```

Let's perform a quick test with `curl` to verify that it is running on the default port, which is 9200:

```
[ec2-user@ip-172-31-26-120 ~]$ curl http://localhost:9200/
{
  "status" : 200,
  "name" : "Siege",
  "cluster_name" : "elasticsearch",
  "version" : {
    "number" : "1.4.1",
    "build_hash" : "89d3241d670db65f994242c8e8383b169779e2d4",
    "build_timestamp" : "2014-11-26T15:49:29Z",
    "build_snapshot" : false,
    "lucene_version" : "4.10.2"
  },
  "tagline" : "You Know, for Search"
}
```

Since the indexes are created as data comes in, and we haven't sent any data, we expect to see no indexes created yet:

```
[ec2-user@ip-172-31-26-120 ~]$ curl http://localhost:9200/_cat/indices?v
health status index pri rep docs.count docs.deleted store.size pri.store.
size
```

All we see is the header with no indexes listed, so let's head over to the collector server and set up the Flume relay which will write data to Elasticsearch.

Setting up Flume on collector/relay

The Flume configuration on the Flume collector will be compressed Avro coming in and Elasticsearch going out. Go ahead and log in to the collector server (172.31.26.205).

Let's start by downloading the Flume binary from the Apache website. Follow the download link and select a mirror:

```
[ec2-user@ip-172-31-26-205 ~]$ wget http://apache.arvixe.com/flume/1.5.2/
apache-flume-1.5.2-bin.tar.gz

--2014-12-07 19:50:30--  http://apache.arvixe.com/flume/1.5.2/apache-
flume-1.5.2-bin.tar.gz

Resolving apache.arvixe.com (apache.arvixe.com)... 198.58.87.82

Connecting to apache.arvixe.com (apache.arvixe.com)|198.58.87.82|:80...
connected.

HTTP request sent, awaiting response... 200 OK

Length: 25323459 (24M) [application/x-gzip]

Saving to: 'apache-flume-1.5.2-bin.tar.gz'


100%[=========================>] 25,323,459  8.38MB/s   in 2.9s


2014-12-07 19:50:33 (8.38 MB/s) - 'apache-flume-1.5.2-bin.tar.gz' saved
[25323459/25323459]
```

Next, expand and change directories:

```
[ec2-user@ip-172-31-26-205 ~]$ tar -zxf apache-flume-1.5.2-bin.tar.gz

[ec2-user@ip-172-31-26-205 ~]$ cd apache-flume-1.5.2-bin

[ec2-user@ip-172-31-26-205 apache-flume-1.5.2-bin]$
```

Create the configuration file, called `collector.conf`, in Flume's configuration directory using your favorite editor:

```
[ec2-user@ip-172-31-18-146 apache-flume-1.5.2-bin]$ cat conf/collector.
conf

collector.sources = av

collector.channels = m1

collector.sinks = es


collector.sources.av.type=avro

collector.sources.av.bind=0.0.0.0
```

```
collector.sources.av.port=12345

collector.sources.av.compression-type=deflate

collector.sources.av.channels=m1


collector.channels.m1.type=memory

collector.channels.m1.capacity=10000


collector.sinks.es.type=org.apache.flume.sink.elasticsearch.
ElasticSearchSink

collector.sinks.es.channel=m1

collector.sinks.es.hostNames=172.31.26.120
```

Here, you can see that we are using a simple memory channel configured with a capacity of 10,000 events.

The source is configured to accept compressed Avro on port 12345 and pass it to our memory channel.

Finally, the sink is configured to write to Elasticsearch on the server we just set up at the `172.31.26.120` private IP. We are using the default settings, which means it'll write to the index named `flume-YYYY-MM-DD` with the `log` type.

Let's try running the Flume agent:

```
[ec2-user@ip-172-31-26-205 apache-flume-1.5.2-bin]$ ./bin/flume-
ng agent -n collector -c conf -f conf/collector.conf -Dflume.root.
logger=INFO,console
```

You'll see an exception in the log, including something like this:

```
2014-12-07 20:00:13,184 (conf-file-poller-0) [ERROR - org.apache.flume.
node.PollingPropertiesFileConfigurationProvider$FileWatcherRunnable.run
(PollingPropertiesFileConfigurationProvider.java:145)] Failed to start
agent because dependencies were not found in classpath. Error follows.

java.lang.NoClassDefFoundError: org/elasticsearch/common/io/BytesStream
```

The Flume agent can't find the Elasticsearch classes. Remember that these are not packaged with Flume, as the libraries need to be compatible with the version of Elasticsearch you are running. Looking back at the Elasticsearch server, we can get an idea of what we need. Remember that this list includes many runtime server dependencies, so it is probably more than what you'll need for a functional Elasticsearch client:

```
[ec2-user@ip-172-31-26-120 ~]$ rpm -qil elasticsearch | grep jar
/usr/share/elasticsearch/lib/elasticsearch-1.4.1.jar
/usr/share/elasticsearch/lib/groovy-all-2.3.2.jar
/usr/share/elasticsearch/lib/jna-4.1.0.jar
/usr/share/elasticsearch/lib/jts-1.13.jar
/usr/share/elasticsearch/lib/log4j-1.2.17.jar
/usr/share/elasticsearch/lib/lucene-analyzers-common-4.10.2.jar
/usr/share/elasticsearch/lib/lucene-core-4.10.2.jar
/usr/share/elasticsearch/lib/lucene-expressions-4.10.2.jar
/usr/share/elasticsearch/lib/lucene-grouping-4.10.2.jar
/usr/share/elasticsearch/lib/lucene-highlighter-4.10.2.jar
/usr/share/elasticsearch/lib/lucene-join-4.10.2.jar
/usr/share/elasticsearch/lib/lucene-memory-4.10.2.jar
/usr/share/elasticsearch/lib/lucene-misc-4.10.2.jar
/usr/share/elasticsearch/lib/lucene-queries-4.10.2.jar
/usr/share/elasticsearch/lib/lucene-queryparser-4.10.2.jar
/usr/share/elasticsearch/lib/lucene-sandbox-4.10.2.jar
/usr/share/elasticsearch/lib/lucene-spatial-4.10.2.jar
/usr/share/elasticsearch/lib/lucene-suggest-4.10.2.jar
/usr/share/elasticsearch/lib/sigar/sigar-1.6.4.jar
/usr/share/elasticsearch/lib/spatial4j-0.4.1.jar
```

Really, you only need the `elasticsearch.jar` file and its dependencies, but we are going to be lazy and just download the RPM again in the collector machine, and copy the JAR files to Flume:

```
[ec2-user@ip-172-31-26-205 ~]$ wget https://download.elasticsearch.org/
elasticsearch/elasticsearch/elasticsearch-1.4.1.noarch.rpm
--2014-12-07 20:03:38--  https://download.elasticsearch.org/
elasticsearch/elasticsearch/elasticsearch-1.4.1.noarch.rpm
Resolving download.elasticsearch.org (download.elasticsearch.org)...
54.225.133.195, 54.243.77.158, 107.22.222.16, ...
Connecting to download.elasticsearch.org (download.elasticsearch.
org)|54.225.133.195|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 26326154 (25M) [application/x-redhat-package-manager]
Saving to: 'elasticsearch-1.4.1.noarch.rpm'
```

```
100%[======================================================================
=======================================>] 26,326,154  9.70MB/s   in 2.6s


2014-12-07 20:03:41 (9.70 MB/s) - 'elasticsearch-1.4.1.noarch.rpm' saved
[26326154/26326154]


[ec2-user@ip-172-31-26-205 ~]$ sudo rpm -ivh elasticsearch-1.4.1.noarch.
rpm
Preparing...                          ################################
[100%]
Updating / installing...
   1:elasticsearch-1.4.1-1             ################################
[100%]
### NOT starting on installation, please execute the following statements
to configure elasticsearch to start automatically using chkconfig
 sudo /sbin/chkconfig --add elasticsearch
### You can start elasticsearch by executing
 sudo service elasticsearch start
```

This time we will *not* configure the service to start. Instead, we'll copy the JAR files we need to Flume's plugins directory architecture, which we learned about in the previous chapter:

```
[ec2-user@ip-172-31-26-205 ~]$ cd apache-flume-1.5.2-bin
[ec2-user@ip-172-31-26-205 apache-flume-1.5.2-bin]$ mkdir -p plugins.d/
elasticsearch/libext
[ec2-user@ip-172-31-26-205 apache-flume-1.5.2-bin]$ cp /usr/share/
elasticsearch/lib/*.jar plugins.d/elasticsearch/libext/
```

Now try running the Flume agent again:

```
[ec2-user@ip-172-31-26-205 apache-flume-1.5.2-bin]$ ./bin/flume-
ng agent -n collector -c conf -f conf/collector.conf -Dflume.root.
logger=INFO,console
```

No exceptions this time around, but still no data. Let's go back to the web server machine and set up the final Flume agent.

Setting up Flume on the client

On the first server, the web server, let's download the Flume binaries again and expand the package:

```
[ec2-user@ip-172-31-18-146 ~]$ wget http://apache.arvixe.com/flume/1.5.2/
apache-flume-1.5.2-bin.tar.gz

--2014-12-07 20:12:53--  http://apache.arvixe.com/flume/1.5.2/apache-
flume-1.5.2-bin.tar.gz

Resolving apache.arvixe.com (apache.arvixe.com)... 198.58.87.82

Connecting to apache.arvixe.com (apache.arvixe.com)|198.58.87.82|:80...
connected.

HTTP request sent, awaiting response... 200 OK

Length: 25323459 (24M) [application/x-gzip]

Saving to: 'apache-flume-1.5.2-bin.tar.gz'


100%[=========================>] 25,323,459  8.13MB/s   in 3.0s


2014-12-07 20:12:56 (8.13 MB/s) - 'apache-flume-1.5.2-bin.tar.gz' saved
[25323459/25323459]


[ec2-user@ip-172-31-18-146 ~]$ tar -zxf apache-flume-1.5.2-bin.tar.gz

[ec2-user@ip-172-31-18-146 ~]$ cd apache-flume-1.5.2-bin
```

This time, our Flume configuration takes the spool directory we set up before, with `logrotate` as input, and it needs to write compressed Avro as the collector server. Open an editor and create the `client.conf` file using your favorite editor:

```
[ec2-user@ip-172-31-18-146 apache-flume-1.5.2-bin]$ cat conf/client.conf

client.sources = sd

client.channels = m1

client.sinks = av


client.sources.sd.type=spooldir

client.sources.sd.spoolDir=/home/ec2-user/spool

client.sources.sd.deletePolicy=immediate

client.sources.sd.ignorePattern=access.log.1$

client.sources.sd.channels=m1


client.channels.m1.type=memory
```

```
client.channels.m1.capacity=10000


client.sinks.av.type=avro
client.sinks.av.hostname=172.31.26.205
client.sinks.av.port=12345
client.sinks.av.compression-type=deflate
client.sinks.av.channel=m1
```

Again, for simplicity, we are using a memory channel with 10,000-record capacity.

For the source, we configure the Spooling Directory Source with `/home/ec2-user/spool` as the input. Additionally, we configure the deletion policy to remove the files after sending is complete. This is also where we set up the exclusion rule for the `access.log.1` filename pattern mentioned earlier. Note the dollar sign at the end of the filename, denoting the end of the line. Without this, the exclusion pattern would also exclude valid files, such as `access.log.1417975373`.

Finally, an Avro sink is configured to point at the collector's private IP and port 12345. Additionally, we set the compression so that it matches the receiving Avro source's settings.

Now let's try running the agent:

```
[ec2-user@ip-172-31-18-146 apache-flume-1.5.2-bin]$ ./bin/flume-ng agent
-n client -c conf -f conf/client.conf -Dflume.root.logger=INFO,console
```

No exceptions! But more importantly, I see the log files in the spool directory being processed and deleted:

```
2014-12-07 20:59:04,041 (pool-4-thread-1) [INFO - org.apache.flume.
client.avro.ReliableSpoolingFileEventReader.deleteCurrentFile(ReliableSp
oolingFileEventReader.java:390)] Preparing to delete file /home/ec2-user/
spool/access.log.1417976401

2014-12-07 20:59:05,319 (pool-4-thread-1) [INFO - org.apache.flume.
client.avro.ReliableSpoolingFileEventReader.deleteCurrentFile(ReliableSp
oolingFileEventReader.java:390)] Preparing to delete file /home/ec2-user/
spool/access.log.1417976701

2014-12-07 20:59:06,245 (pool-4-thread-1) [INFO - org.apache.flume.
client.avro.ReliableSpoolingFileEventReader.deleteCurrentFile(ReliableSp
oolingFileEventReader.java:390)] Preparing to delete file /home/ec2-user/
spool/access.log.1417977001

2014-12-07 20:59:06,245 (pool-4-thread-1) [INFO - org.apache.
flume.source.SpoolDirectorySource$SpoolDirectoryRunnable.
run(SpoolDirectorySource.java:254)] Spooling Directory Source runner has
shutdown.
```

```
2014-12-07 20:59:06,746 (pool-4-thread-1) [INFO - org.apache.
flume.source.SpoolDirectorySource$SpoolDirectoryRunnable.
run(SpoolDirectorySource.java:254)] Spooling Directory Source runner has
shutdown.
```

Once all the files have been processed, the last lines are repeated every 500 milliseconds. This is a known bug in Flume (`https://issues.apache.org/jira/browse/FLUME-2385`). It has already been fixed and is slated for the 1.6.0 release, so be sure to set up log rotation on your Flume agent and clean up this mess before you run out of disk space.

On the collector box, we see writes occurring to Elasticsearch:

```
2014-12-07 22:10:03,694 (SinkRunner-PollingRunner-DefaultSinkProcessor)
[INFO - org.apache.flume.sink.elasticsearch.client.
ElasticSearchTransportClient.execute(ElasticSearchTransportClient.
java:181)] Sending bulk to elasticsearch cluster
```

Querying the Elasticsearch REST API, we can see an index with records:

```
[ec2-user@ip-172-31-26-120 ~]$ curl http://localhost:9200/_cat/indices?v

health status index              pri rep docs.count docs.deleted store.size
pri.store.size
yellow open    flume-2014-12-07  5   1      32102             0       1.3mb
1.3mb
```

Let's read the first 5 records:

```
[ec2-user@ip-172-31-26-120 elasticsearch]$ curl -XGET 'http://
localhost:9200/flume-2014-12-07/_search?pretty=true&q=*.*&size=5'
{
  "took" : 26,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 32102,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "flume-2014-12-07",
      "_type" : "log",
```

```
        "_id" : "AUomjGVVbObD75ecNqJg",

        "_score" : 1.0,

        "_source":{"@message":"207.222.127.224 - - [07/Dec/2014:18:01:47
+0000] \"GET / HTTP/1.1\" 200 3770 \"-\" \"-\" \"-\"","@fields":{}}

    }, {

        "_index" : "flume-2014-12-07",

        "_type" : "log",

        "_id" : "AUomjGVWbObD75ecNqJj",

        "_score" : 1.0,

        "_source":{"@message":"207.222.127.224 - - [07/Dec/2014:18:01:47
+0000] \"GET / HTTP/1.1\" 200 3770 \"-\" \"-\" \"-\"","@fields":{}}

    }, {

        "_index" : "flume-2014-12-07",

        "_type" : "log",

        "_id" : "AUomjGVWbObD75ecNqJo",

        "_score" : 1.0,

        "_source":{"@message":"207.222.127.224 - - [07/Dec/2014:18:01:47
+0000] \"GET / HTTP/1.1\" 200 3770 \"-\" \"-\" \"-\"","@fields":{}}

    }, {

        "_index" : "flume-2014-12-07",

        "_type" : "log",

        "_id" : "AUomjGVWbObD75ecNqJt",

        "_score" : 1.0,

        "_source":{"@message":"207.222.127.224 - - [07/Dec/2014:18:01:47
+0000] \"GET / HTTP/1.1\" 200 3770 \"-\" \"-\" \"-\"","@fields":{}}

    }, {

        "_index" : "flume-2014-12-07",

        "_type" : "log",

        "_id" : "AUomjGVWbObD75ecNqJy",

        "_score" : 1.0,

        "_source":{"@message":"207.222.127.224 - - [07/Dec/2014:18:01:47
+0000] \"GET / HTTP/1.1\" 200 3770 \"-\" \"-\" \"-\"","@fields":{}}

    } ]
  }
}
```

As you can see, the log lines are in there under the `@message` field, and they can now be searched. However, we can do better. Let's break that message down into searchable fields.

Creating more search fields with an interceptor

Let's borrow some code from what we covered earlier in this book to extract some Flume headers from this common log format, knowing that all Flume headers will become fields in Elasticsearch. Since we are creating fields to be searched by in Elasticsearch, I'm going to add them to the collector's configuration rather than the web server's Flume agent.

Change the agent configuration on the collector to include a Regular Expression Extractor interceptor:

```
[ec2-user@ip-172-31-18-146 apache-flume-1.5.2-bin]$ cat conf/collector.
conf
collector.sources = av
collector.channels = m1
collector.sinks = es


collector.sources.av.type=avro
collector.sources.av.bind=0.0.0.0
collector.sources.av.port=12345
collector.sources.av.compression-type=deflate
collector.sources.av.channels=m1
collector.sources.av.interceptors=e1
collector.sources.av.interceptors.e1.type=regex_extractor
collector.sources.av.interceptors.e1.regex=^([\\d.]+) \\S+ \\S+ \\
[([\\w:/]+\\s[+\\-]\\d{4})\\] \"(.+?)\" (\\d{3}) (\\d+)
collector.sources.av.interceptors.e1.serializers=ip dt url sc bc
collector.sources.av.interceptors.e1.serializers.ip.name=source
collector.sources.av.interceptors.e1.serializers.dt.type=org.apache.
flume.interceptor.RegexExtractorInterceptorMillisSerializer
collector.sources.av.interceptors.e1.serializers.dt.pattern=dd/MMM/
yyyy:dd:HH:mm:ss Z
collector.sources.av.interceptors.e1.serializers.dt.name=timestamp
collector.sources.av.interceptors.e1.serializers.url.name=http_request
```

```
collector.sources.av.interceptors.e1.serializers.sc.name=status_code
collector.sources.av.interceptors.e1.serializers.bc.name=bytes_xfered


collector.channels.m1.type=memory
collector.channels.m1.capacity=10000


collector.sinks.es.type=org.apache.flume.sink.elasticsearch.
ElasticSearchSink
collector.sinks.es.channel=m1
collector.sinks.es.hostNames=172.31.26.120
```

To follow the Logstash convention, I've renamed the hostname header to `source`. Now, to make the new format easy to find, I'm going to delete the existing index:

```
[ec2-user@ip-172-31-26-120 ~]$ curl -XDELETE 'http://localhost:9200/
flume-2014-12-07/'
{"acknowledged":true}
[ec2-user@ip-172-31-26-120 ~]$ curl -XGET 'http://localhost:9200/
flume-2014-12-07/_search?pretty=true&q=*.*&size=5'
{
  "error" : "IndexMissingException[[flume-2014-12-07] missing]",
  "status" : 404
}
```

Next, I create more traffic on my web server and wait for it to appear at the other end. Then I query some records to see what it looks like:

```
[ec2-user@ip-172-31-26-120 ~]$ curl -XGET 'http://localhost:9200/
flume-2014-12-07/_search?pretty=true&q=*.*&size=5'
{
  "took" : 95,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 12083,
```

```
      "max_score" : 1.0,
      "hits" : [ {
        "_index" : "flume-2014-12-07",
        "_type" : "log",
        "_id" : "AUomq4FnbObD75ecNx_H",
        "_score" : 1.0,
        "_source":{"@message":"207.222.127.224 - - [07/Dec/2014:18:13:15
+0000] \"GET / HTTP/1.1\" 200 3770 \"-\" \"-\" \"-\"","@timestamp":"2014-
12-07T18:13:15.000Z","@source":"207.222.127.224","@fields":{"timestamp"
:"1417975995000","status_code":"200","source":"207.222.127.224","http_
request":"GET / HTTP/1.1","bytes_xfered":"3770"}}
      }, {
        "_index" : "flume-2014-12-07",
        "_type" : "log",
        "_id" : "AUomq4FnbObD75ecNx_M",
        "_score" : 1.0,
        "_source":{"@message":"207.222.127.224 - - [07/Dec/2014:18:13:16
+0000] \"GET / HTTP/1.1\" 200 3770 \"-\" \"-\" \"-\"","@timestamp":"2014-
12-07T18:13:16.000Z","@source":"207.222.127.224","@fields":{"timestamp"
:"1417975996000","status_code":"200","source":"207.222.127.224","http_
request":"GET / HTTP/1.1","bytes_xfered":"3770"}}
      }, {
        "_index" : "flume-2014-12-07",
        "_type" : "log",
        "_id" : "AUomq4FnbObD75ecNx_R",
        "_score" : 1.0,
        "_source":{"@message":"207.222.127.224 - - [07/Dec/2014:18:13:16
+0000] \"GET / HTTP/1.1\" 200 3770 \"-\" \"-\" \"-\"","@timestamp":"2014-
12-07T18:13:16.000Z","@source":"207.222.127.224","@fields":{"timestamp"
:"1417975996000","status_code":"200","source":"207.222.127.224","http_
request":"GET / HTTP/1.1","bytes_xfered":"3770"}}
      }, {
        "_index" : "flume-2014-12-07",
        "_type" : "log",
        "_id" : "AUomq4FnbObD75ecNx_W",
        "_score" : 1.0,
```

```
        "_source":{"@message":"207.222.127.224 - - [07/Dec/2014:18:13:16
+0000] \"GET / HTTP/1.1\" 200 3770 \"-\" \"-\" \"-\"","@timestamp":"2014-
12-07T18:13:16.000Z","@source":"207.222.127.224","@fields":{"timestamp"
:"1417975996000","status_code":"200","source":"207.222.127.224","http_
request":"GET / HTTP/1.1","bytes_xfered":"3770"}}
    }, {
        "_index" : "flume-2014-12-07",
        "_type" : "log",
        "_id" : "AUomq4FnbObD75ecNx_a",
        "_score" : 1.0,
        "_source":{"@message":"207.222.127.224 - - [07/Dec/2014:18:13:16
+0000] \"GET / HTTP/1.1\" 200 3770 \"-\" \"-\" \"-\"","@timestamp":"2014-
12-07T18:13:16.000Z","@source":"207.222.127.224","@fields":{"timestamp"
:"1417975996000","status_code":"200","source":"207.222.127.224","http_
request":"GET / HTTP/1.1","bytes_xfered":"3770"}}
    } ]
  }
}
```

As you can see, we now have additional fields that we can search by. Additionally, you can see that Elasticsearch has taken our millisecond-based `timestamp` field and created its own `@timestamp` field in ISO-8601 format.

Now we can do some more interesting queries such as finding out how many successful (status `200`) pages we saw:

```
[ec2-user@ip-172-31-26-120 elasticsearch]$ curl -XGET 'http://
localhost:9200/flume-2014-12-07/_count' -d '{"query":{"term":{"status_
code":"200"}}}'
{"count":46018,"_shards":{"total":5,"successful":5,"failed":0}}
```

Setting up a better user interface – Kibana

While it appears as if we are done, there is one more thing we should do. The data is all there, but you need to be an expert in querying Elasticsearch to make good use of the information. After all, if the data is difficult to consume and gets ignored, then why bother collecting it at all? What you really need is a nice, searchable web interface that humans with non-technical backgrounds can use. For this, we are going to set up Kibana. In a nutshell, Kibana is a web application that runs as a dynamic HTML page on your browser, making calls for data to Elasticsearch when necessary. The result is an interactive web interface that doesn't require you to learn the details of the Elasticsearch query API. This is not the only option available to you; it is just what I'm using in this example. Let's download Kibana 3 from the Elasticsearch website, and install this on the same server (although you could easily serve this from another HTTP server):

```
[ec2-user@ip-172-31-26-120 ~]$ wget https://download.elasticsearch.org/
kibana/kibana/kibana-3.1.2.tar.gz

--2014-12-07 18:31:16--  https://download.elasticsearch.org/kibana/
kibana/kibana-3.1.2.tar.gz

Resolving download.elasticsearch.org (download.elasticsearch.org)...
54.225.133.195, 54.243.77.158, 107.22.222.16, ...

Connecting to download.elasticsearch.org (download.elasticsearch.
org)|54.225.133.195|:443... connected.

HTTP request sent, awaiting response... 200 OK

Length: 1074306 (1.0M) [application/octet-stream]

Saving to: 'kibana-3.1.2.tar.gz'


100%[======================================================================
=======================================>] 1,074,306   1.33MB/s   in 0.8s


2014-12-07 18:31:17 (1.33 MB/s) - 'kibana-3.1.2.tar.gz' saved
[1074306/1074306]


[ec2-user@ip-172-31-26-120 ~]$ tar -zxf kibana-3.1.2.tar.gz

[ec2-user@ip-172-31-26-120 ~]$ cd kibana-3.1.2
```

At the time of writing this book, a newer version of Kibana (Kibana 4) is in beta. These instructions may be outdated by the time this book is released, but rest assured that somewhere in the Kibana setup, you can edit a configuration file to point to your Elasticsearch server. I have not tried out the newer version yet, but there is a good overview of it on the Elasticsearch blog at `http://www.elasticsearch.org/blog/kibana-4-beta-3-now-more-filtery`.

Open the `config.js` file to edit the line with the `elasticsearch` key. This needs to be the URL of the *public* name or IP of the Elasticsearch API. For our example, this line should look as shown here:

```
elasticsearch: 'http://ec2-54-148-230-252.us-west-2.compute.amazonaws.com:9200',
```

Now we need to provide this directory with a web browser. Let's download and install Nginx again:

```
[ec2-user@ip-172-31-26-120 ~]$ sudo yum install nginx
```

By default, the root directory is `/usr/share/nginx/html`. We can change this configuration in Nginx, but to make things easy, let's just create a symbolic link to point to the right location. First, move the original path out of the way by renaming it:

```
[ec2-user@ip-172-31-26-120 ~]$ sudo mv /usr/share/nginx/html /usr/share/nginx/html.dist
```

Next, link the configured Kibana directory as the new web root:

```
[ec2-user@ip-172-31-26-120 ~]$ sudo ln -s ~/kibana-3.1.2 /usr/share/nginx/html
```

Finally, start the web server:

```
[ec2-user@ip-172-31-26-120 ~]$ sudo /etc/init.d/nginx start
Starting nginx:                                            [  OK  ]
```

From your computer, go to `http://54.148.230.252/`. If you see this page, it means you may have made a mistake in your Kibana configuration:
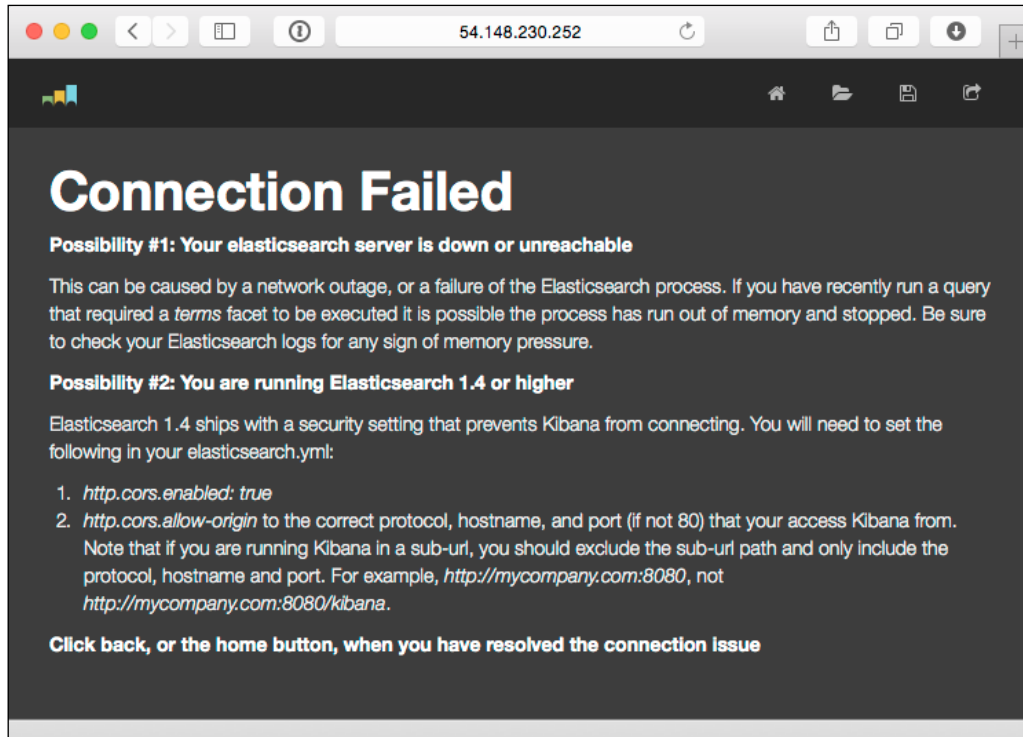
This error page means your web browser can't connect to Elasticsearch. You may need to clear your browser cache if you fixed the configuration, as web pages are typically cached locally for some period of time. If you got it right, the screen should look like this:

Go ahead and select **Sample Dashboard**, the first option. You should see something like what is shown in the next screenshot. This includes some of the data we ingested, record counts in the center, and the filtering fields in the left-hand margin.
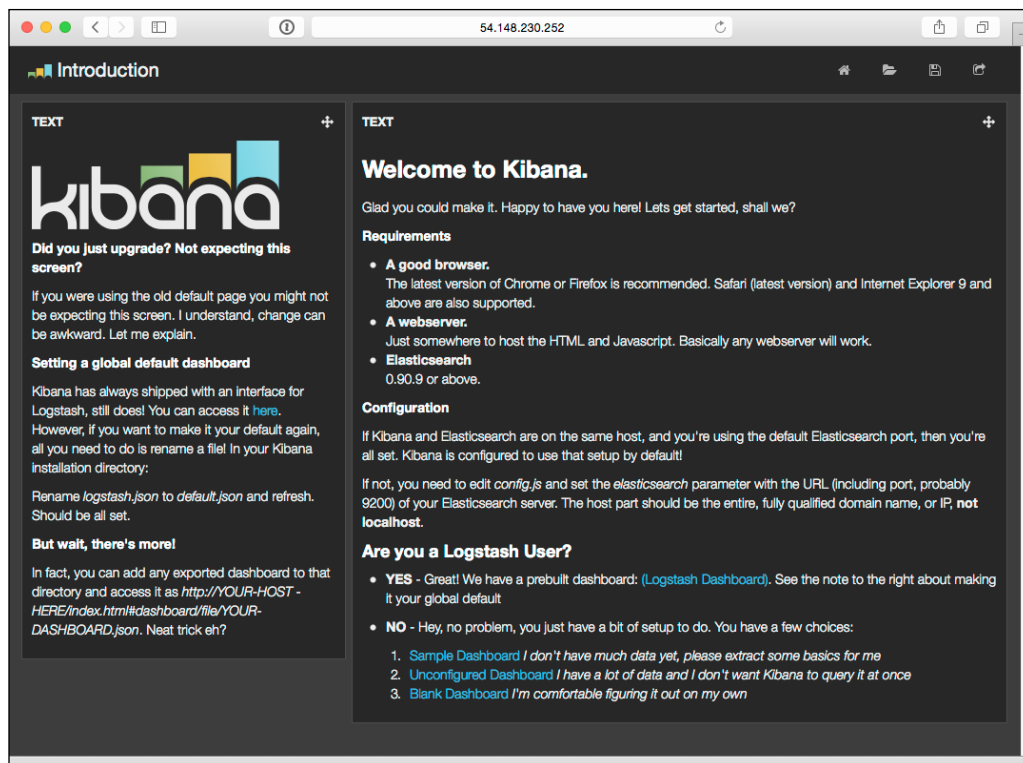
I'm not going to claim to be a Kibana expert (I'm far from that), so I'll leave further customization of this to you. Use this as a base to go back and make additional modifications to the data to make it easier to consume, search, or filter. To do that, you'll probably need to get more familiar with how Elasticsearch works, but that's okay because knowledge isn't a bad thing. A copious amount of documentation is waiting for you at `http://www.elasticsearch.org/guide/en/kibana/current/index.html`.

At this point, we have completed an end-to-end implementation of data from a web server streamed in a near real-time fashion to a web-based tool for searching. Since both the source and target formats were dictated by others, we used an interceptor to transform the data en route. This use case is very good for short-term troubleshooting, but it's clearly not very "Hadoopy" since we have yet to use core Hadoop.

Archiving to HDFS

When people speak of Hadoop, they usually refer to storing lots of data for a long time, usually in HDFS, so more interesting data science or machine learning can be done later. Let's extend our use case by splitting the data flow at the collector to store an extra copy in HDFS for later use.

So, back in Amazon AWS, I start a fourth server to run Hadoop. If you plan on doing all your work in Hadoop, you'll probably want to write this data to S3, but for this example, let's stick with HDFS. Now our server diagram looks like this:
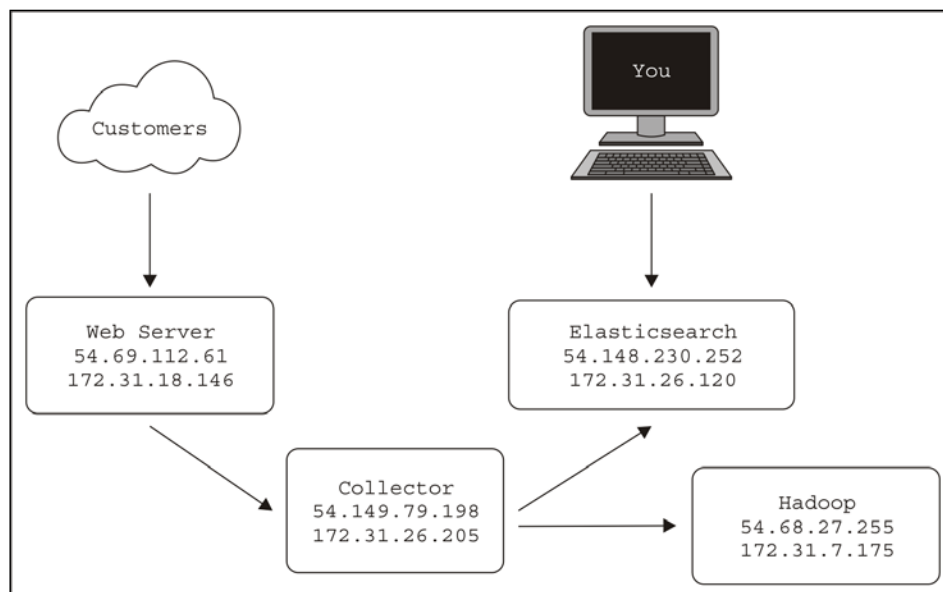
I used Cloudera's one-line installation instructions to speed up the setup. It's instructions can be found at `http://www.cloudera.com/content/cloudera/en/ documentation/core/latest/topics/cdh_qs_mrv1_pseudo.html`.

Since the Amazon AMI is compatible with Enterprise Linux 6, I selected the EL6 RPM Repository and imported the corresponding GPG key:

```
[ec2-user@ip-172-31-7-175 ~]$ sudo rpm -ivh http://archive.cloudera.com/
cdh5/one-click-install/redhat/6/x86_64/cloudera-cdh-5-0.x86_64.rpm
Retrieving http://archive.cloudera.com/cdh5/one-click-install/redhat/6/
x86_64/cloudera-cdh-5-0.x86_64.rpm
Preparing...                          #################################
[100%]
Updating / installing...
   1:cloudera-cdh-5-0                  #################################
[100%]
[ec2-user@ip-172-31-7-175 ~]$ sudo rpm --import http://archive.cloudera.
com/cdh5/redhat/6/x86_64/cdh/RPM-GPG-KEY-cloudera
```

Next, I installed the pseudo-distributed configuration to run in a single-node Hadoop cluster:

```
[ec2-user@ip-172-31-7-175 ~]$ sudo yum install -y hadoop-0.20-conf-pseudo
```

This might take a while as it downloads all the Cloudera Hadoop distribution dependencies.

Since this configuration is for single-node use, we need to adjust the `fs.defaultFS` property in `/etc/hadoop/conf/core-site.xml` to advertise our private IP instead of `localhost`. If we don't do this, the `namenode` process will bind to `127.0.0.1`, and other servers, such as our collector's Flume agent, will not be able to contact it:

```
    <property>
      <name>fs.defaultFS</name>
      <value>hdfs://172.31.7.175:8020</value>
    </property>
```

Next, we format the new HDFS volume and start the HDFS daemon (since that is all we need for this example):

```
[ec2-user@ip-172-31-7-175 ~]$ sudo -u hdfs hdfs namenode -format
[ec2-user@ip-172-31-7-175 ~]$ for x in 'cd /etc/init.d ; ls hadoop-
hdfs-*' ; do sudo service $x start ; done
starting datanode, logging to /var/log/hadoop-hdfs/hadoop-hdfs-datanode-
ip-172-31-7-175.out
```

```
Started Hadoop datanode (hadoop-hdfs-datanode):        [  OK  ]
starting namenode, logging to /var/log/hadoop-hdfs/hadoop-hdfs-namenode-
ip-172-31-7-175.out
Started Hadoop namenode:                               [  OK  ]
starting secondarynamenode, logging to /var/log/hadoop-hdfs/hadoop-hdfs-
secondarynamenode-ip-172-31-7-175.out
Started Hadoop secondarynamenode:                      [  OK  ]
[ec2-user@ip-172-31-7-175 ~]$ hadoop fs -df
Filesystem                      Size    Used    Available  Use%
hdfs://172.31.7.175:8020  8318783488  24576  6661824512    0%
```

Now that HDFS is running, let's go back to the collector box configuration to create a second channel and an HDFS Sink by adding these lines:

```
collector.channels.h1.type=memory
collector.channels.h1.capacity=10000
collector.sinks.hadoop.type=hdfs
collector.sinks.hadoop.channel=h1
collector.sinks.hadoop.hdfs.path=hdfs://172.31.7.175/access_
logs/%Y/%m/%d/%H
collector.sinks.hadoop.hdfs.filePrefix=access
collector.sinks.hadoop.hdfs.rollInterval=60
collector.sinks.hadoop.hdfs.rollSize=0
collector.sinks.hadoop.hdfs.rollCount=0
```

Then we modify the top-level channels and sinks keys:

```
collector.channels = m1 h1
collector.sinks = es hadoop
```

As you can see, I've gone with a simple memory channel again, but feel free to use a durable file channel if you need it. For the HDFS configuration, I'll be using a dated file path from the /access_logs root directory with a 60-second rotation regardless of size. We are not altering the source just yet, so don't worry.

If we attempt to start the collector now, we see this exception:

```
java.lang.NoClassDefFoundError: org/apache/hadoop/io/
SequenceFile$CompressionType
```

Remember that for Apache Flume to speak to HDFS, we need compatible HDFS classes and dependencies for the version of Hadoop we are speaking to. Let's get some help from our friends at Cloudera and install the Hadoop client RPM (output removed to save paper):

```
[ec2-user@ip-172-31-26-205 ~]$ sudo rpm -ivh http://archive.cloudera.com/
cdh5/one-click-install/redhat/6/x86_64/cloudera-cdh-5-0.x86_64.rpm
```

```
[ec2-user@ip-172-31-26-205 ~]$ sudo rpm --import http://archive.cloudera.
com/cdh5/redhat/6/x86_64/cdh/RPM-GPG-KEY-cloudera
```

```
[ec2-user@ip-172-31-26-205 ~]$ sudo yum install hadoop-client
```

Test whether RPM works by creating the destination directory for our data. We'll also set permissions to match the account we'll be running the Flume agent under (`ec2-user` in this case):

```
[ec2-user@ip-172-31-26-205 ~]$ sudo -u hdfs hadoop fs -mkdir
hdfs://172.31.7.175/access_logs
```

```
[ec2-user@ip-172-31-26-205 ~]$ sudo -u hdfs hadoop fs -chown ec2-
user:ec2-user hdfs://172.31.7.175/access_logs
```

```
[ec2-user@ip-172-31-26-205 ~]$ hadoop fs -ls hdfs://172.31.7.175/
Found 1 items
drwxr-xr-x   - ec2-user ec2-user          0 2014-12-11 04:35
hdfs://172.31.7.175/access_logs
```

Now, if we run the collector Flume agent, we see no exceptions due to missing HDFS classes. The Flume startup script detects that we have a local Hadoop installation, and appends its class path to its own.

Finally, we can go back to the Flume configuration file, and split the incoming data at the source by listing both channels on the source as destination channels:

```
    collector.sources.av.channels=m1 h1
```

By default, a replicating channel selector is used. This is what we want, so that no further configuration is needed. Save the configuration file and restart the Flume agent.

When data is flowing, you should see the expected HDFS activity in the Flume collector agent's logs:

```
2014-12-11 05:05:04,141 (SinkRunner-PollingRunner-DefaultSinkProcessor)
[INFO - org.apache.flume.sink.hdfs.BucketWriter.open(BucketWriter.
java:261)] Creating hdfs://172.31.7.175/access_logs/2014/12/11/05/
access.1418274302083.tmp
```

You should also see data appearing in HDFS:

```
[ec2-user@ip-172-31-7-175 ~]$ ls -R /access_logs
drwxr-xr-x   - ec2-user ec2-user          0 2014-12-11 05:05 /access_
logs/2014
drwxr-xr-x   - ec2-user ec2-user          0 2014-12-11 05:05 /access_
logs/2014/12
```

```
drwxr-xr-x   - ec2-user ec2-user          0 2014-12-11 05:05 /access_
logs/2014/12/11
drwxr-xr-x   - ec2-user ec2-user          0 2014-12-11 05:06 /access_
logs/2014/12/11/05
-rw-r--r--   3 ec2-user ec2-user      10486 2014-12-11 05:05 /access_
logs/2014/12/11/05/access.1418274302082
-rw-r--r--   3 ec2-user ec2-user      10486 2014-12-11 05:05 /access_
logs/2014/12/11/05/access.1418274302083
-rw-r--r--   3 ec2-user ec2-user       6429 2014-12-11 05:06 /access_
logs/2014/12/11/05/access.1418274302084
```

If you run this command from a server other than the Hadoop node, you'll need to specify the full HDFS URI (`hdfs://172.31.7.175/access_logs`) or set the `default.FS` property in the `core-site.xml` configuration file in Hadoop.
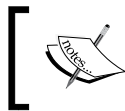
> In retrospect, I probably should have configured the `default.FS` property on any installed Hadoop client that will use HDFS to save myself a lot of typing. Live and learn!

Like the preceding Elasticsearch implementation, you can now use this example as a base end-to-end configuration for HDFS. The next step will be to go back and modify the format, compression, and so on, on the HDFS Sink to better match what you'll do with the data later.

Summary

In this chapter, we iteratively assembled an end-to-end data flow. We started by setting up an Nginx web server to create access logs. We also configured cron to execute a `logrotate` configuration periodically to safely rotate old logs to a spooling directory.

Next, we installed and configured a single-node Elasticsearch server and tested some insertions and deletions. Then we configured a Flume client to read input from our spooling directory filled with web logs, and relay them to a Flume collector using compressed Avro serialization. The collector then relayed the incoming data to our Elasticsearch server.

Once we saw data flowing from one end to another, we set up a single-node HDFS server and modified our collector configuration to split the input data feed and relay a copy of the message to HDFS, simulating archival storage. Finally, we set up a Kibana UI in front of our Elasticsearch instance to provide an easy-search function for nontechnical consumers.

In the next chapter, we will cover monitoring Flume data flows using Ganglia.

8
Monitoring Flume

The user guide for Flume states:

> *Monitoring in Flume is still a work in progress. Changes can happen very often.*
> *Several Flume components report metrics to the JMX platform MBean server.*
> *These metrics can be queried using Jconsole.*

While JMX is fine for casual browsing of metric values, the number of eyeballs looking at Jconsole doesn't scale when you have hundreds or even thousands of servers sending data all over the place. What you need is a way to watch everything at once. However, what are the important things to look for? That is a very difficult question, but I'll try and cover several of the items that are important, as we cover monitoring options in this chapter.

Monitoring the agent process

The most obvious type of monitoring you'll want to perform is **Flume agent process monitoring**, that is, making sure the agent is still running. There are many products that do this kind of process monitoring, so there is no way we can cover them all. If you work at a company of any reasonable size, chances are there is already a system in place for this. If this is the case, do not go off and build your own. The last thing operations wants is yet another screen to watch 24/7.

Monit

If you do not already have something in place, one freemium option is **Monit** (`http://mmonit.com/monit/`). The developers of Monit have a paid version that provides more bells and whistles you may want to consider. Even in the free form, it can provide you with a way to check whether the Flume agent is running, restart it if it isn't, and send you an e-mail when this happens so that you can look into why it died.

Monit does much more, but this functionality is what we will cover here. If you are smart, and I know you are, you will add checks to the disk, CPU, and memory usage as a minimum, in addition to what we cover in this chapter.

Nagios

Another option for Flume agent process monitoring is **Nagios** (`http://www.nagios.org/`). Like Monit, you can configure it to watch your Flume agents and alert you via a web UI, e-mail, or an SNMP trap. That said, it doesn't have restart capabilities. The community is quite strong, and there are many plugins for other available applications.

My company uses this to check the availability of Hadoop web UIs. While not a complete picture of health, it does provide more information to the overall monitoring of our Hadoop ecosystem.

Again, if you already have tools in place at your company, see whether you can reuse them before bringing in another tool.

Monitoring performance metrics

Now that we have covered some options for process monitoring, how do you know whether your application is actually doing the work you think it is? On many occasions, I've seen a stuck `syslog-ng` process that appears to be running, but it just wasn't sending any data. I'm not picking on `syslog-ng` specifically; all software does this when conditions that are not designed for occur.

When talking about Flume data flows, you need to monitor the following:

- Data entering sources is within expected rates
- Data isn't overflowing your channels
- Data is exiting sinks at expected rates

Flume has a pluggable monitoring framework, but as mentioned at the beginning of the chapter, it is still very much a work in progress. This does not mean you shouldn't use it, as that would be foolish. It means you'll want to prepare extra testing and integration time anytime you upgrade.
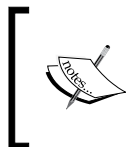
> While not covered in the Flume documentation, it is common to enable JMX in your Flume JVM (`http://bit.ly/javajmx`) and use the Nagios JMX plugin (`http://bit.ly/nagiosjmx`) to alert you about performance abnormalities in your Flume agents.

Ganglia

One of the available monitoring options to watch Flume internal metrics is Ganglia integration. Ganglia (`http://ganglia.sourceforge.net/`) is an open source monitoring tool that is used to collect metrics and display graphs, and it can be tiered to handle very large installations. To send your Flume metrics to your Ganglia cluster, you need to pass some properties to your agent at startup time:

Java property	Value	Description
`flume.monitoring.type`	`ganglia`	Set to `ganglia`
`flume.monitoring.hosts`	`host1:port1,` `host2:port2`	A comma-separated list of `host:port` pairs for your `gmond` process(es)
`flume.monitoring.pollFrequency`	`60`	The number of seconds between sending of data (default 60 seconds)
`flume.monitoring.isGanglia3`	`false`	Set to `true` if using older Ganglia 3 protocol. The default process is to send data using v3.1 protocol.

Look at each instance of `gmond` within the same network broadcast domain (as reachability is based on multicast packets), and find the `udp_recv_channel` block in `gmond.conf`. Let's say I had two nearby servers with these two corresponding configuration blocks:

```
udp_recv_channel {
  mcast_join = 239.2.14.22
  port = 8649
  bind = 239.2.14.22
  retry_bind = true
}
udp_recv_channel {
  mcast_join = 239.2.11.71
  port = 8649
  bind = 239.2.11.71
  retry_bind = true
}
```

In this case the IP and port are `239.2.14.22/8649` for the first server and `239.2.11.71/8649` for the second, leading to these startup properties:

```
-Dflume.monitoring.type=ganglia
-Dflume.monitoring.hosts=239.2.14.22:8649,239.2.11.71:8649
```

Here, I am using defaults for the poll interval, and I'm also using the newer Ganglia wire protocol.

> While receiving data via TCP is supported in Ganglia, the current Flume/Ganglia integration only supports sending data using multicast UDP. If you have a large/complicated network setup, you'll want to get educated by your network engineers if things don't work as you expect.

Internal HTTP server

You can configure the Flume agent to start an HTTP server that will output JSON that can use queries by outside mechanisms. Unlike the Ganglia integration, an external entity has to call the Flume agent to poll the data. In theory, you can use Nagios to poll this JSON data and alert on certain conditions, but I have personally never tried it. Of course, this setup is very useful in development and testing, especially if you are writing custom Flume components to be sure they are generating useful metrics. Here is a summary of the Java properties you'll need to set at the start up of the Flume agent:

Java property	Value	Description
`flume.monitoring.type`	`http`	The value is set to `http`
`flume.monitoring.port`	`PORT`	This is the port number to bind the HTTP server

The URL for metrics will be `http://SERVER_OR_IP_OF_AGENT:PORT/metrics`.

Let's look at the following Flume configuration:

```
agent.sources = s1
agent.channels = c1
agent.sinks = k1
agent.sources.s1.type=avro
agent.sources.s1.bind=0.0.0.0
agent.sources.s1.port=12345
agent.sources.s1.channels=c1
agent.channels.c1.type=memory
agent.sinks.k1.type=avro
agent.sinks.k1.hostname=192.168.33.33
agent.sinks.k1.port=9999
agent.sinks.k1.channel=c1
```

Start the Flume agent with these properties:

```
-Dflume.monitoring.type=http
-Dflume.monitoring.port=44444
```

Now, when you go to `http://SERVER_OR_IP:44444/metrics`, you might see something like this:

```
{
  "SOURCE.s1":{
    "OpenConnectionCount":"0",
    "AppendBatchAcceptedCount":"0",
    "AppendBatchReceivedCount":"0",
    "Type":"SOURCE",
    "EventAcceptedCount":"0",
    "AppendReceivedCount":"0",
    "StopTime":"0",
    "EventReceivedCount":"0",
    "StartTime":"1365128622891",
    "AppendAcceptedCount":"0"},
  "CHANNEL.c1":{
    "EventPutSuccessCount":"0",
    "ChannelFillPercentage":"0.0",
    "Type":"CHANNEL",
    "StopTime":"0",
    "EventPutAttemptCount":"0",
    "ChannelSize":"0",
    "StartTime":"1365128621890",
    "EventTakeSuccessCount":"0",
    "ChannelCapacity":"100",
    "EventTakeAttemptCount":"0"},
  "SINK.k1":{
    "BatchCompleteCount":"0",
    "ConnectionFailedCount":"4",
    "EventDrainAttemptCount":"0",
    "ConnectionCreatedCount":"0",
    "BatchEmptyCount":"0",
    "Type":"SINK",
    "ConnectionClosedCount":"0",
    "EventDrainSuccessCount":"0",
    "StopTime":"0",
```

```
    "StartTime":"1365128622325",
    "BatchUnderflowCount":"0"}
}
```

As you can see, each source, sink, and channel are broken out separately with their corresponding metrics. Each type of source, channel, and sink provide their own set of metric keys, although there is some commonality, so be sure to check what looks interesting. For instance, this Avro source has `OpenConnectionCount`, that is, the number of connected clients (who are most likely sending data in). This may help you decide whether you have the expected number of clients relying on that data or, perhaps, too many clients, and you need to start tiering your agents.

Generally speaking, the channel's `ChannelSize` or `ChannelFillPercentage` metrics will give you a good idea whether the data is coming in faster than it is going out. It will also tell you whether you have it set large enough for maintenance/ outages of your data volume.

Looking at the sink, `EventDrainSuccessCount` versus `EventDrainAttemptCount` will tell you how often output is successful when compared to the times tried. In this example, I am configuring an Avro sink to a nonexistent target. As you can see, the `ConnectionFailedCount` metric is growing, which is a good indicator of persistent connection problems. Even a growing `ConnectionCreatedCount` metric can indicate that connections are dropping and reopening too often.

Really, there are no hard and fast rules besides watching `ChannelSize`/ `ChannelFillPercentage`. Each use case will have its own performance profile, so start small, set up your monitoring, and learn as you go.

Custom monitoring hooks

If you already have a monitoring system, you may want to take the extra effort to develop a custom monitoring reporting mechanism. You may think this is as simple as implementing the `org.apache.flume.instrumentation.MonitorService` interface. You do need to do this, but looking at the interface, you will only see a `start()` and `stop()` method. Unlike the more obvious interceptor paradigm, the agent expects that your `MonitorService` implementation will start/stop a thread to send data on the expected or configured interval if it is the type to send data to a receiving service. If you are going to operate a service, such as the HTTP service, then start/stop would be used to start and stop your listening service. The metrics themselves are published internally to JMX by the various sources, sinks, channels, and interceptors using object names that start with `org.apache.flume`. Your implementation will need to read these from `MBeanServer`.

The best advice I can give you, should you decide to implement your own, is to look at the source of two existing implementations (included in the source download referenced in *Chapter 2, A Quick Start Guide to Flume*) and do what they do. To use your monitoring hook, set the `flume.monitoring.type` property to the fully qualified class name of your implementation class. Expect to have to rework any custom hooks with new Flume versions until the framework matures and stabilizes.

Summary

In this chapter, we covered monitoring Flume agents both from the process level and the monitoring of internal metrics (whether it is working).

Monit and Nagios were introduced as open source options for process watching.

Next, we covered the Flume agent internal monitoring metrics with Ganglia and JSON over HTTP implementations that ship with Apache Flume.

Finally, we covered how to integrate a custom monitoring implementation if you need to directly integrate to some other tool that's not supported by Flume by default.

In our final chapter, we will discuss some general considerations for your Flume deployment.

9
There Is No Spoon – the Realities of Real-time Distributed Data Collection

In this last chapter, I thought we should cover some of the less concrete, random thoughts I have around data collection into Hadoop. There's no hard science behind some of this, and you should feel perfectly alright to disagree with me.

While Hadoop is a great tool to consume vast quantities of data, I often think of a picture of the logjam that occurred in 1886 in the St. Croix River in Minnesota (`http://www.nps.gov/sacn/historyculture/stories.htm`). When dealing with too much data, you want to make sure you don't jam your river. Be sure to take the previous chapter on monitoring seriously and not just as nice-to-have information.

Transport time versus log time

I had a situation where data was being placed using date patterns in the filename and/or the path in HDFS didn't match the contents of the directories. The expectation was that the data in the `2014/12/29` directory path contained all the data for December 29, 2014. However, the reality was that the date was being pulled from the transport. It turns out that the version of `syslog` we were using was rewriting the header, including the date portion, causing the data to take on the transport time and not reflect the original time of the record. Usually, the offsets were tiny, just a second or two, so nobody really took notice. However, one day, one of the relay servers died and when the data that had got stuck on upstream servers was finally sent, it had the current time. In this case, it was shifted by a couple of days, causing a significant data cleanup effort.

Be sure this isn't happening to you if you are placing data by date. Check the date edge cases to see that they are what you expect, and make sure you test your outage scenarios *before* they happen for real in production.

As I mentioned previously, these retransmits due to planned or unplanned maintenance (or even a tiny network hiccup) will most likely cause duplicate and out-of-order events to arrive, so be sure to account for this when processing raw data. There are no single delivery or ordering guarantees in Flume. If you need that, use a transactional database or distributed transaction log such as Apache Kafka (`http://kafka.apache.org/`) instead. Of course, if you are going to use Kafka, you would probably only use Flume for the final leg of your data path, with your source consuming events from Kafka (`https://github.com/baniuyao/flume-ng-kafka-source`).

> Remember that you can always work around duplicates in your data at query time as long as you can uniquely identify your events from one another. If you cannot distinguish events easily, you can add a **Universally Unique Identifier** (**UUID**) (`http://en.wikipedia.org/wiki/Universally_unique_identifier`) header using the bundled interceptor, `UUIDInterceptor` (configuration details are in the Flume User Guide).

Time zones are evil

In case you missed my bias against using local time in *Chapter 4*, *Sinks and Sink Processors*, I'll repeat it here a little stronger: time zones are evil—evil like Dr. Evil (`http://en.wikipedia.org/wiki/Dr._Evil`)—and let's not forget about his Mini Me counterpart, (`http://en.wikipedia.org/wiki/Mini-Me`)—Daylight Savings Time.

We live in a global world now. You are pulling data from all over the place into your Hadoop cluster. You may even have multiple data centers in different parts of the country (or the world). The last thing you want to be doing while trying to analyze your data is to deal with askew data. Daylight Savings Time changes at least somewhere on Earth a dozen times in a year. Just look at the history: `ftp://ftp.iana.org/tz/releases/`. Save yourself the headache and just normalize it to UTC. If you want to convert it to "local time" on its way to human eyeballs, feel free. However, while it lives in your cluster, keep it normalized to UTC.

> Consider adopting UTC everywhere via this Java startup parameter (if you can't set it system-wide): `-Duser.timezone=UTC`
>
> Also, use the **ISO 8601** (`http://en.wikipedia.org/wiki/ISO_8601`) time standard where possible and be sure to include time zone information (even if it is UTC). Every modern tool on the planet supports this format and will save you pain down the road.

I live in Chicago, and our computers at work use Central Time, which adjusts for daylight savings. In our Hadoop cluster, we like to keep data in a `YYYY/MM/DD/HH` directory layout. Twice a year, some things break slightly. In the fall, we have twice as much data in our 2 a.m. directory. In the spring, there is no 2 a.m. directory. Madness!

Capacity planning

Regardless of how much data you think you have, things will change over time. New projects will pop up and data creation rates for your existing projects will change (up or down). Data volume will usually ebb and flow with the traffic of the day. Finally, the number of servers feeding your Hadoop cluster will change over time.

There are many schools of thought on how much extra storage capacity you should keep in your Hadoop cluster (we use the totally unscientific value of 20 percent, which means that we usually plan for 80 percent full when ordering additional hardware but don't start to panic until we hit the 85-90 percent utilization number). Generally, you want to keep enough extra space so that the failure and/or maintenance of a server or two won't cause the HDFS block replication to consume all the remaining space.

You may also need to set up multiple flows inside a single agent. The source and sink processors are currently single-threaded, so there is some limit to what tuning batch sizes can accomplish when under heavy data volumes. Be very careful in these situations where you split your data flow at the source using a replicating channel selector to multiple channels/sinks. If one of the path's channels fills up, an exception is thrown back to the source. If that full channel is not marked as optional and the data is dropped, the source will stop consuming new data. This effectively jams the agent for all other channels attached to that source. You may not want to drop the data (marking the channel as optional) because the data is important. Unfortunately, this is the only fan-out mechanism provided in Flume to send to multiple destinations, so make sure you catch issues quickly so that all your data flows are not impaired due to a cascade backup of events.

For a number of Flume agents feeding Hadoop, this too should be adjusted based on real numbers. Watch the channel size to see how well the writes are keeping up under normal loads. Adjust the maximum channel capacity to handle whatever amount of overhead makes you feel good. You can always purchase way more hardware than you need, but even a prolonged outage may overflow even the most conservative estimates. This is when you have to pick and choose which data is more important to you and adjust your channel capacities to reflect that. This way, if you exceed your limits, the least important data will be the first to be dropped.

Chances are your company doesn't have an infinite amount of money and at some point, the value of the data versus the cost of continuing to expand your cluster will start to be questioned. This is why setting limits on the volume of data collected is very important. This is just one aspect of your data retention policy, where cost is the driving factor. In a moment, we'll discuss some of the compliance aspects of this policy. Suffice to say, any project sending data into Hadoop should be able to say what the value of that data is and what the loss is if we delete the older stuff. This is the only way the people writing the checks can make an informed decision.

Considerations for multiple data centers

If you run your business out of multiple data centers and have a large volume of data collected, you may want to consider setting up a Hadoop cluster in each data center rather than sending all your collected data back to a single data center. There may be regulatory implications regarding data crossing certain geographic boundaries. Chances are there is somebody in your company who knows much more about compliance than you or I, so seek them out before you start copying data across borders. Of course, not collating your data will make it more difficult to analyze it, as you can't just run one MapReduce job against all the data. Instead, you would have to run parallel jobs and then combine the results in a second pass. Adjusting your data processing procedures is better than potentially breaking the law. Be sure to do your homework.

Pulling all your data into a single cluster may also be more than your networking can handle. Depending on how your data centers are connected to each other, you simply may not be able to transmit the desired volume of data. If you use public cloud services, there are surely data transfer costs between data centers. Finally, consider that a complete cluster failure or corruption may wipe out everything, as most clusters are usually too big to back up everything except high value data. Having some of the old data in this case is sometimes better than having nothing. With multiple Hadoop clusters, you have the ability to use a FailoverSinkProcessor to forward data to a different cluster if you don't want to wait to send to the local one.

If you do choose to send all your data to a single destination, consider adding a large disk capacity machine as a relay server for the data center. This way, if there is a communication issue or extended cluster maintenance, you can let data pile up on a machine that's different from the ones trying to service your customers. This is sound advice even in a single data center situation.

Compliance and data expiry

Remember that the data your company is collecting from your customers should be considered sensitive information. You may be bound by additional regulatory limitations on accessing data such as:

- **Personally identifiable information** (**PII**): How you handle and safeguard customer's identities `http://en.wikipedia.org/wiki/Personally_identifiable_information`

- **Payment Card Industry Data Security Standard** (**PCI DSS**): How you safeguard credit card information `http://en.wikipedia.org/wiki/PCI_DSS`

- **Service Organization Control** (**SOC-2**): How you control access to information/systems `http://www.aicpa.org/InterestAreas/FRC/AssuranceAdvisoryServices/Pages/AICPASOC2Report.aspx`

- **Statements on Standards for Attestation Engagements** (**SSAE-16**): How you manage changes `http://www.aicpa.org/Research/Standards/AuditAttest/DownloadableDocuments/AT-00801.pdf`

- **Sarbanes Oxley** (**SOX**): `http://en.wikipedia.org/wiki/Sarbanes%E2%80%93Oxley_Act`

This is by no means a definitive list, so be sure to seek out your company's compliance experts for what does and doesn't apply to your situation. If you aren't properly handling access to this data in your cluster, the government will lean on you, or worse, you won't have customers anymore if they feel you aren't protecting their personal information. Consider scrambling, trimming, or obfuscating your data of personal information. Chances are the business insight you are looking falls more into the category of "how many people who search for "hammer" actually buy one?" rather than "how many customers are named Bob?" As you saw in *Chapter 6, Interceptors, ETL, and Routing*, it would be very easy to write an interceptor to obfuscate PII as you move it around.

Your company probably has a document retention policy that includes the data you are putting into Hadoop. Make sure you remove data that your policy says you aren't supposed to be keeping around anymore. The last thing you want is a visit from the lawyers.

Summary

In this chapter, we covered several real-world considerations you need to think about when planning your Flume implementation, including:

- Transport time does not always match event time
- The mayhem introduced with Daylight Savings Time to certain time-based logic
- Capacity planning considerations
- Items to consider when you have more than one data center
- Data compliance
- Data retention and expiration

I hope you enjoyed this book. Hopefully, you will be able to apply much of this information directly in your application/Hadoop integration efforts.

Thanks, this was fun!

Index

Thank you for buying
Apache Flume: Distributed Log Collection for Hadoop
Second Edition

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.
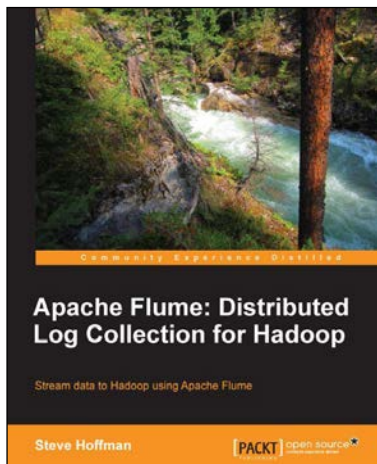
Apache Flume: Distributed Log Collection for Hadoop

ISBN: 978-1-78216-791-4   Paperback: 108 pages

Stream data to Hadoop using Apache Flume

1. Integrate Flume with your data sources.

2. Transcode your data en-route in Flume.

3. Route and separate your data using regular expression matching.

4. Configure failover paths and load-balancing to remove single points of failure.
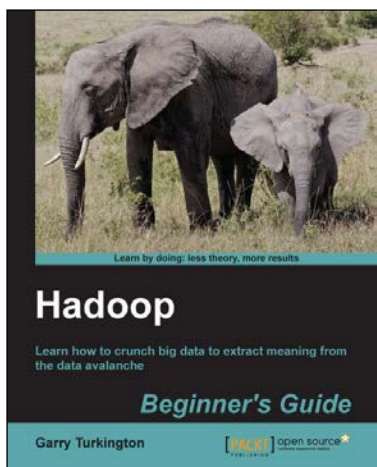
Hadoop Beginner's Guide

ISBN: 978-1-84951-730-0   Paperback: 398 pages

Learn how to crunch big data to extract meaning from the data avalanche

1. Learn tools and techniques that let you approach big data with relish and not fear.

2. Shows how to build a complete infrastructure to handle your needs as your data grows.

3. Hands-on examples in each chapter give the big picture while also giving direct experience.

Please check **www.PacktPub.com** for information on our titles
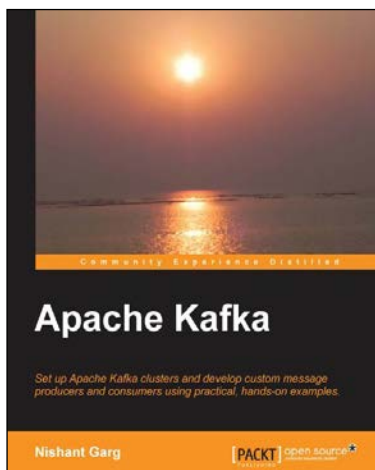
Apache Kafka

ISBN: 978-1-78216-793-8   Paperback: 88 pages

Set up Apache Kafka clusters and develop custom message producers and consumers using practical, hand-on examples

1. Write custom producers and consumers with message partition techniques.

2. Integrate Kafka with Apache Hadoop and Storm for use cases such as processing streaming data.

3. Provide an overview of Kafka tools and other contributions that work with Kafka in areas such as logging, packaging, and so on.
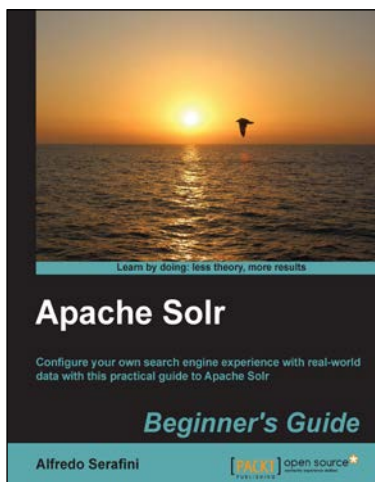
Apache Solr Beginner's Guide

ISBN: 978-1-78216-252-0   Paperback: 324 pages

Configure your own search engine experience with real-world data with this practical guide to Apache Solr

1. Learn to use Solr in real-world contexts, even if you are not a programmer, using simple configuration examples.

2. Define simple configurations for searching data in several ways in your specific context, from suggestions to advanced faceted navigation.

3. Teaches you in an easy-to-follow style, full of examples, illustrations, and tips to suit the demands of beginners.

Please check **www.PacktPub.com** for information on our titles