CHAPTER 14

Flume

Hadoop is built for processing very large datasets. Often it is assumed that the data is already in HDFS, or can be copied there in bulk. However, there are many systems that don't meet this assumption. They produce streams of data that we would like to aggregate, store, and analyze using Hadoop—and these are the systems that Apache Flume is an ideal fit for.

Flume is designed for high-volume ingestion into Hadoop of event-based data. The canonical example is using Flume to collect logfiles from a bank of web servers, then moving the log events from those files into new aggregated files in HDFS for processing. The usual destination (or *sink* in Flume parlance) is HDFS. However, Flume is flexible enough to write to other systems, like HBase or Solr.

To use Flume, we need to run a Flume *agent*, which is a long-lived Java process that runs *sources* and *sinks*, connected by *channels*. A source in Flume produces *events* and delivers them to the channel, which stores the events until they are forwarded to the sink. You can think of the source-channel-sink combination as a basic Flume building block.

A Flume installation is made up of a collection of connected agents running in a distributed topology. Agents on the edge of the system (co-located on web server machines, for example) collect data and forward it to agents that are responsible for aggregating and then storing the data in its final destination. Agents are configured to run a collection of particular sources and sinks, so using Flume is mainly a configuration exercise in wiring the pieces together. In this chapter, we'll see how to build Flume topologies for data ingestion that you can use as a part of your own Hadoop pipeline.

Installing Flume

Download a stable release of the Flume binary distribution from the download page, and unpack the tarball in a suitable location:

```
% tar xzf apache-flume-x.y.z-bin.tar.gz
```

It's useful to put the Flume binary on your path:

```
% export FLUME_HOME=~/sw/apache-flume-x.y.z-bin
% export PATH=$PATH:$FLUME_HOME/bin
```

A Flume agent can then be started with the `flume-ng` command, as we'll see next.

An Example

To show how Flume works, let's start with a setup that:

1. Watches a local directory for new text files
2. Sends each line of each file to the console as files are added

We'll add the files by hand, but it's easy to imagine a process like a web server creating new files that we want to continuously ingest with Flume. Also, in a real system, rather than just logging the file contents we would write the contents to HDFS for subsequent processing—we'll see how to do that later in the chapter.

In this example, the Flume agent runs a single source-channel-sink, configured using a Java properties file. The configuration controls the types of sources, sinks, and channels that are used, as well as how they are connected together. For this example, we'll use the configuration in Example 14-1.

Example 14-1. Flume configuration using a spooling directory source and a logger sink

```
agent1.sources = source1
agent1.sinks = sink1
agent1.channels = channel1

agent1.sources.source1.channels = channel1
agent1.sinks.sink1.channel = channel1

agent1.sources.source1.type = spooldir
agent1.sources.source1.spoolDir = /tmp/spooldir

agent1.sinks.sink1.type = logger

agent1.channels.channel1.type = file
```

Property names form a hierarchy with the agent name at the top level. In this example, we have a single agent, called `agent1`. The names for the different components in an agent are defined at the next level, so for example `agent1.sources` lists the names of the sources that should be run in `agent1` (here it is a single source, `source1`). Similarly, `agent1` has a sink (`sink1`) and a channel (`channel1`).

The properties for each component are defined at the next level of the hierarchy. The configuration properties that are available for a component depend on the type of the component. In this case, `agent1.sources.source1.type` is set to `spooldir`, which is a spooling directory source that monitors a spooling directory for new files. The spooling directory source defines a `spoolDir` property, so for `source1` the full key is `agent1.sources.source1.spoolDir`. The source's channels are set with `agent1.sources.source1.channels`.

The sink is a `logger` sink for logging events to the console. It too must be connected to the channel (with the `agent1.sinks.sink1.channel` property).[1] The channel is a `file` channel, which means that events in the channel are persisted to disk for durability. The system is illustrated in Figure 14-1.

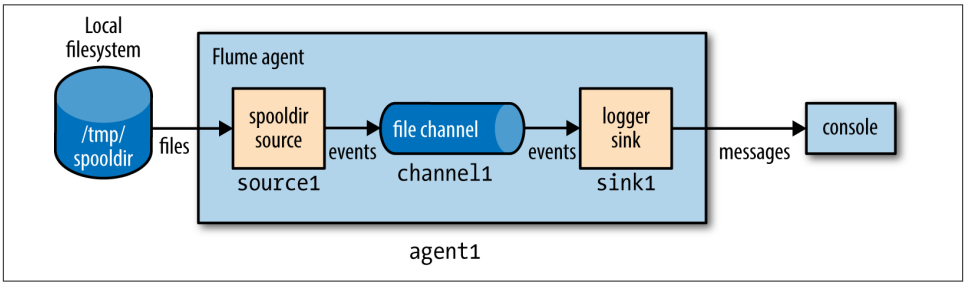

Figure 14-1. Flume agent with a spooling directory source and a logger sink connected by a file channel

Before running the example, we need to create the spooling directory on the local filesystem:

```
% mkdir /tmp/spooldir
```

Then we can start the Flume agent using the `flume-ng` command:

```
% flume-ng agent \
  --conf-file spool-to-logger.properties \
  --name agent1 \
  --conf $FLUME_HOME/conf \
  -Dflume.root.logger=INFO,console
```

The Flume properties file from Example 14-1 is specified with the `--conf-file` flag. The agent name must also be passed in with `--name` (since a Flume properties file can

1. Note that a source has a `channels` property (plural) but a sink has a `channel` property (singular). This is because a source can feed more than one channel (see "Fan Out" on page 388), but a sink can only be fed by one channel. It's also possible for a channel to feed multiple sinks. This is covered in "Sink Groups" on page 395.

define several agents, we have to say which one to run). The `--conf` flag tells Flume where to find its general configuration, such as environment settings.

In a new terminal, create a file in the spooling directory. The spooling directory source expects files to be immutable. To prevent partially written files from being read by the source, we write the full contents to a hidden file. Then, we do an atomic rename so the source can read it:[2]

```
% echo "Hello Flume" > /tmp/spooldir/.file1.txt
% mv /tmp/spooldir/.file1.txt /tmp/spooldir/file1.txt
```

Back in the agent's terminal, we see that Flume has detected and processed the file:

```
Preparing to move file /tmp/spooldir/file1.txt to
 /tmp/spooldir/file1.txt.COMPLETED
Event: { headers:{} body: 48 65 6C 6C 6F 20 46 6C 75 6D 65         Hello Flume }
```

The spooling directory source ingests the file by splitting it into lines and creating a Flume event for each line. Events have optional headers and a binary body, which is the UTF-8 representation of the line of text. The body is logged by the logger sink in both hexadecimal and string form. The file we placed in the spooling directory was only one line long, so only one event was logged in this case. We also see that the file was renamed to *file1.txt.COMPLETED* by the source, which indicates that Flume has completed processing it and won't process it again.

Transactions and Reliability

Flume uses separate transactions to guarantee delivery from the source to the channel and from the channel to the sink. In the example in the previous section, the spooling directory source creates an event for each line in the file. The source will only mark the file as completed once the transactions encapsulating the delivery of the events to the channel have been successfully committed.

Similarly, a transaction is used for the delivery of the events from the channel to the sink. If for some unlikely reason the events could not be logged, the transaction would be rolled back and the events would remain in the channel for later redelivery.

The channel we are using is a *file channel*, which has the property of being durable: once an event has been written to the channel, it will not be lost, even if the agent restarts. (Flume also provides a *memory channel* that does not have this property, since events are stored in memory. With this channel, events are lost if the agent restarts. Depending on the application, this might be acceptable. The trade-off is that the memory channel has higher throughput than the file channel.)

2. For a logfile that is continually appended to, you would periodically roll the logfile and move the old file to the spooling directory for Flume to read it.

The overall effect is that every event produced by the source will reach the sink. The major caveat here is that every event will reach the sink *at least once*—that is, duplicates are possible. Duplicates can be produced in sources or sinks: for example, after an agent restart, the spooling directory source will redeliver events for an uncompleted file, even if some or all of them had been committed to the channel before the restart. After a restart, the logger sink will re-log any event that was logged but not committed (which could happen if the agent was shut down between these two operations).

At-least-once semantics might seem like a limitation, but in practice it is an acceptable performance trade-off. The stronger semantics of *exactly once* require a two-phase commit protocol, which is expensive. This choice is what differentiates Flume (at-least-once semantics) as a high-volume parallel event ingest system from more traditional enterprise messaging systems (exactly-once semantics). With at-least-once semantics, duplicate events can be removed further down the processing pipeline. Usually this takes the form of an application-specific deduplication job written in MapReduce or Hive.

Batching

For efficiency, Flume tries to process events in batches for each transaction, where possible, rather than one by one. Batching helps file channel performance in particular, since every transaction results in a local disk write and `fsync` call.

The batch size used is determined by the component in question, and is configurable in many cases. For example, the spooling directory source will read files in batches of 100 lines. (This can be changed by setting the `batchSize` property.) Similarly, the Avro sink (discussed in "Distribution: Agent Tiers" on page 390) will try to read 100 events from the channel before sending them over RPC, although it won't block if fewer are available.

The HDFS Sink

The point of Flume is to deliver large amounts of data into a Hadoop data store, so let's look at how to configure a Flume agent to deliver events to an HDFS sink. The configuration in Example 14-2 updates the previous example to use an HDFS sink. The only two settings that are required are the sink's type (`hdfs`) and `hdfs.path`, which specifies the directory where files will be placed (if, like here, the filesystem is not specified in the path, it's determined in the usual way from Hadoop's `fs.defaultFS` property). We've also specified a meaningful file prefix and suffix, and instructed Flume to write events to the files in text format.

Example 14-2. Flume configuration using a spooling directory source and an HDFS sink

```
agent1.sources = source1
agent1.sinks = sink1
```

```
agent1.channels = channel1

agent1.sources.source1.channels = channel1
agent1.sinks.sink1.channel = channel1

agent1.sources.source1.type = spooldir
agent1.sources.source1.spoolDir = /tmp/spooldir

agent1.sinks.sink1.type = hdfs
agent1.sinks.sink1.hdfs.path = /tmp/flume
agent1.sinks.sink1.hdfs.filePrefix = events
agent1.sinks.sink1.hdfs.fileSuffix = .log
agent1.sinks.sink1.hdfs.inUsePrefix = _
agent1.sinks.sink1.hdfs.fileType = DataStream

agent1.channels.channel1.type = file
```

Restart the agent to use the *spool-to-hdfs.properties* configuration, and create a new file in the spooling directory:

```
% echo -e "Hello\nAgain" > /tmp/spooldir/.file2.txt
% mv /tmp/spooldir/.file2.txt /tmp/spooldir/file2.txt
```

Events will now be delivered to the HDFS sink and written to a file. Files in the process of being written to have a *.tmp* in-use suffix added to their name to indicate that they are not yet complete. In this example, we have also set `hdfs.inUsePrefix` to be `_` (underscore; by default it is empty), which causes files in the process of being written to have that prefix added to their names. This is useful since MapReduce will ignore files that have a `_` prefix. So, a typical temporary filename would be *_events.1399295780136.log.tmp*; the number is a timestamp generated by the HDFS sink.

A file is kept open by the HDFS sink until it has either been open for a given time (default 30 seconds, controlled by the `hdfs.rollInterval` property), has reached a given size (default 1,024 bytes, set by `hdfs.rollSize`), or has had a given number of events written to it (default 10, set by `hdfs.rollCount`). If any of these criteria are met, the file is closed and its in-use prefix and suffix are removed. New events are written to a new file (which will have an in-use prefix and suffix until it is rolled).

After 30 seconds, we can be sure that the file has been rolled and we can take a look at its contents:

```
% hadoop fs -cat /tmp/flume/events.1399295780136.log
Hello
Again
```

The HDFS sink writes files as the user who is running the Flume agent, unless the `hdfs.proxyUser` property is set, in which case files will be written as that user.

Partitioning and Interceptors

Large datasets are often organized into partitions, so that processing can be restricted to particular partitions if only a subset of the data is being queried. For Flume event data, it's very common to partition by time. A process can be run periodically that transforms completed partitions (to remove duplicate events, for example).

It's easy to change the example to store data in partitions by setting `hdfs.path` to include subdirectories that use time format escape sequences:

```
agent1.sinks.sink1.hdfs.path = /tmp/flume/year=%Y/month=%m/day=%d
```

Here we have chosen to have day-sized partitions, but other levels of granularity are possible, as are other directory layout schemes. (If you are using Hive, see "Partitions and Buckets" on page 491 for how Hive lays out partitions on disk.) The full list of format escape sequences is provided in the documentation for the HDFS sink in the Flume User Guide.

The partition that a Flume event is written to is determined by the `timestamp` header on the event. Events don't have this header by default, but it can be added using a Flume *interceptor*. Interceptors are components that can modify or drop events in the flow; they are attached to sources, and are run on events before the events have been placed in a channel.[3] The following extra configuration lines add a timestamp interceptor to `source1`, which adds a `timestamp` header to every event produced by the source:

```
agent1.sources.source1.interceptors = interceptor1
agent1.sources.source1.interceptors.interceptor1.type = timestamp
```

Using the timestamp interceptor ensures that the timestamps closely reflect the times at which the events were created. For some applications, using a timestamp for when the event was written to HDFS might be sufficient—although, be aware that when there are multiple tiers of Flume agents there can be a significant difference between creation time and write time, especially in the event of agent downtime (see "Distribution: Agent Tiers" on page 390). For these cases, the HDFS sink has a setting, `hdfs.useLocalTimeStamp`, that will use a timestamp generated by the Flume agent running the HDFS sink.

File Formats

It's normally a good idea to use a binary format for storing your data in, since the resulting files are smaller than they would be if you used text. For the HDFS sink, the file format used is controlled using `hdfs.fileType` and a combination of a few other properties.

3. Table 14-1 describes the interceptors that Flume provides.

If unspecified, `hdfs.fileType` defaults to `SequenceFile`, which will write events to a sequence file with `LongWritable` keys that contain the event timestamp (or the current time if the `timestamp` header is not present) and `BytesWritable` values that contain the event body. It's possible to use `Text` Writable values in the sequence file instead of `BytesWritable` by setting `hdfs.writeFormat` to `Text`.

The configuration is a little different for Avro files. The `hdfs.fileType` property is set to `DataStream`, just like for plain text. Additionally, `serializer` (note the lack of an `hdfs.` prefix) must be set to `avro_event`. To enable compression, set the `serializer.compressionCodec` property. Here is an example of an HDFS sink configured to write Snappy-compressed Avro files:

```
agent1.sinks.sink1.type = hdfs
agent1.sinks.sink1.hdfs.path = /tmp/flume
agent1.sinks.sink1.hdfs.filePrefix = events
agent1.sinks.sink1.hdfs.fileSuffix = .avro
agent1.sinks.sink1.hdfs.fileType = DataStream
agent1.sinks.sink1.serializer = avro_event
agent1.sinks.sink1.serializer.compressionCodec = snappy
```

An event is represented as an Avro record with two fields: `headers`, an Avro map with string values, and `body`, an Avro bytes field.

If you want to use a custom Avro schema, there are a couple of options. If you have Avro in-memory objects that you want to send to Flume, then the `Log4jAppender` is appropriate. It allows you to log an Avro Generic, Specific, or Reflect object using a log4j `Logger` and send it to an Avro source running in a Flume agent (see "Distribution: Agent Tiers" on page 390). In this case, the `serializer` property for the HDFS sink should be set to `org.apache.flume.sink.hdfs.AvroEventSerializer$Builder`, and the Avro schema set in the header (see the class documentation).

Alternatively, if the events are not originally derived from Avro objects, you can write a custom serializer to convert a Flume event into an Avro object with a custom schema. The helper class `AbstractAvroEventSerializer` in the `org.apache.flume.serialization` package is a good starting point.

Fan Out

Fan out is the term for delivering events from one source to multiple channels, so they reach multiple sinks. For example, the configuration in Example 14-3 delivers events to both an HDFS sink (`sink1a` via `channel1a`) and a logger sink (`sink1b` via `channel1b`).

Example 14-3. Flume configuration using a spooling directory source, fanning out to an HDFS sink and a logger sink

```
agent1.sources = source1
agent1.sinks = sink1a sink1b
agent1.channels = channel1a channel1b
```

```
agent1.sources.source1.channels = channel1a channel1b
agent1.sinks.sink1a.channel = channel1a
agent1.sinks.sink1b.channel = channel1b

agent1.sources.source1.type = spooldir
agent1.sources.source1.spoolDir = /tmp/spooldir

agent1.sinks.sink1a.type = hdfs
agent1.sinks.sink1a.hdfs.path = /tmp/flume
agent1.sinks.sink1a.hdfs.filePrefix = events
agent1.sinks.sink1a.hdfs.fileSuffix = .log
agent1.sinks.sink1a.hdfs.fileType = DataStream

agent1.sinks.sink1b.type = logger

agent1.channels.channel1a.type = file
agent1.channels.channel1b.type = memory
```

The key change here is that the source is configured to deliver to multiple channels by setting `agent1.sources.source1.channels` to a space-separated list of channel names, `channel1a` and `channel1b`. This time, the channel feeding the logger sink (`channel1b`) is a memory channel, since we are logging events for debugging purposes and don't mind losing events on agent restart. Also, each channel is configured to feed one sink, just like in the previous examples. The flow is illustrated in Figure 14-2.

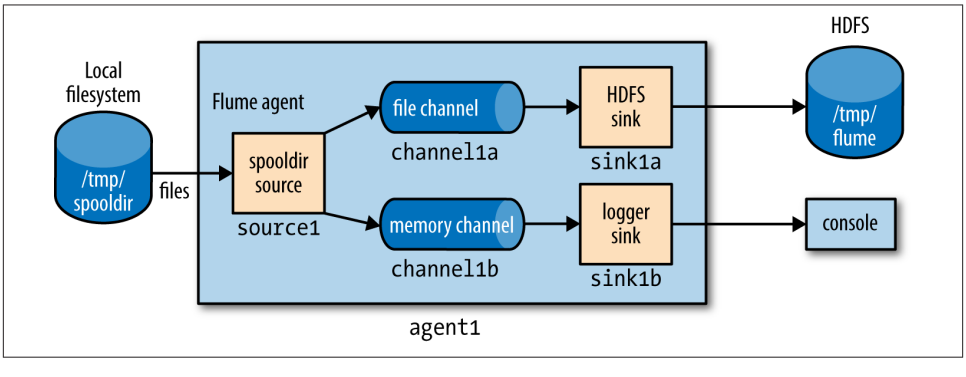

Figure 14-2. Flume agent with a spooling directory source and fanning out to an HDFS sink and a logger sink

Delivery Guarantees

Flume uses a separate transaction to deliver each batch of events from the spooling directory source to each channel. In this example, one transaction will be used to deliver to the channel feeding the HDFS sink, and then another transaction will be used to deliver the same batch of events to the channel for the logger sink. If either of these

transactions fails (if a channel is full, for example), then the events will not be removed from the source, and will be retried later.

In this case, since we don't mind if some events are not delivered to the logger sink, we can designate its channel as an *optional* channel, so that if the transaction associated with it fails, this will not cause events to be left in the source and tried again later. (Note that if the agent fails before *both* channel transactions have committed, then the affected events will be redelivered after the agent restarts—this is true even if the uncommitted channels are marked as optional.) To do this, we set the `selector.optional` property on the source, passing it a space-separated list of channels:

```
agent1.sources.source1.selector.optional = channel1b
```

Near-Real-Time Indexing

Indexing events for search is a good example of where fan out is used in practice. A single source of events is sent to both an HDFS sink (this is the main repository of events, so a required channel is used) and a Solr (or Elasticsearch) sink, to build a search index (using an optional channel).

The `MorphlineSolrSink` extracts fields from Flume events and transforms them into a Solr document (using a Morphline configuration file), which is then loaded into a live Solr search server. The process is called *near real time* since ingested data appears in search results in a matter of seconds.

Replicating and Multiplexing Selectors

In normal fan-out flow, events are replicated to all channels—but sometimes more selective behavior might be desirable, so that some events are sent to one channel and others to another. This can be achieved by setting a *multiplexing* selector on the source, and defining routing rules that map particular event header values to channels. See the Flume User Guide for configuration details.

Distribution: Agent Tiers

How do we scale a set of Flume agents? If there is one agent running on every node producing raw data, then with the setup described so far, at any particular time each file being written to HDFS will consist entirely of the events from one node. It would be better if we could aggregate the events from a group of nodes in a single file, since this would result in fewer, larger files (with the concomitant reduction in pressure on HDFS, and more efficient processing in MapReduce; see "Small files and CombineFileInputFormat" on page 226). Also, if needed, files can be rolled more often since they are being

fed by a larger number of nodes, leading to a reduction between the time when an event is created and when it's available for analysis.

Aggregating Flume events is achieved by having *tiers* of Flume agents. The first tier collects events from the original sources (such as web servers) and sends them to a smaller set of agents in the second tier, which aggregate events from the first tier before writing them to HDFS (see Figure 14-3). Further tiers may be warranted for very large numbers of source nodes.

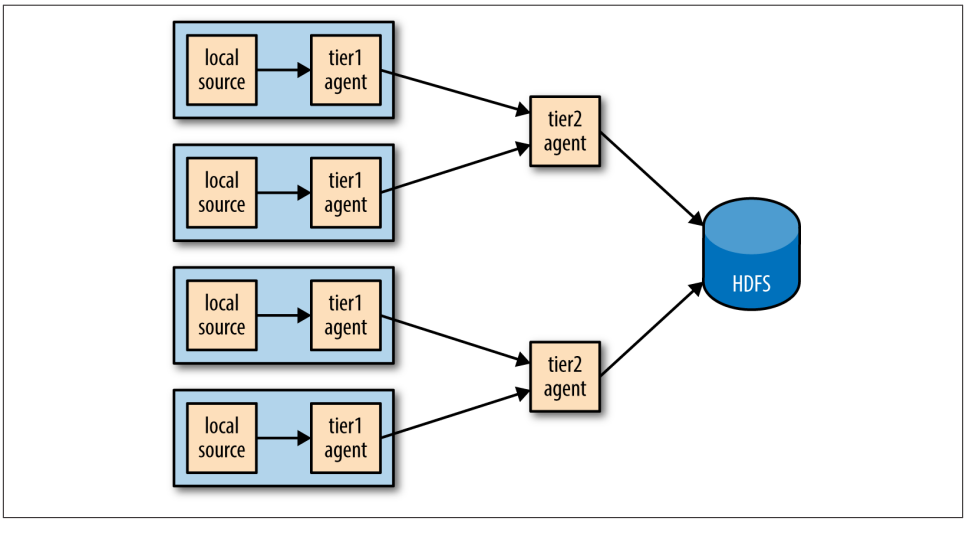

Figure 14-3. Using a second agent tier to aggregate Flume events from the first tier

Tiers are constructed by using a special sink that sends events over the network, and a corresponding source that receives events. The Avro sink sends events over Avro RPC to an Avro source running in another Flume agent. There is also a Thrift sink that does the same thing using Thrift RPC, and is paired with a Thrift source.[4]

Don't be confused by the naming: Avro sinks and sources do not provide the ability to write (or read) Avro files. They are used only to distribute events between agent tiers, and to do so they use Avro RPC to communicate (hence the name). If you need to write events to Avro files, use the HDFS sink, described in "File Formats" on page 387.

4. The Avro sink-source pair is older than the Thrift equivalent, and (at the time of writing) has some features that the Thrift one doesn't provide, such as encryption.

Example 14-4 shows a two-tier Flume configuration. Two agents are defined in the file, named `agent1` and `agent2`. An agent of type `agent1` runs in the first tier, and has a `spooldir` source and an Avro sink connected by a file channel. The `agent2` agent runs in the second tier, and has an Avro source that listens on the port that `agent1`'s Avro sink sends events to. The sink for `agent2` uses the same HDFS sink configuration from Example 14-2.

Notice that since there are two file channels running on the same machine, they are configured to point to different data and checkpoint directories (they are in the user's home directory by default). This way, they don't try to write their files on top of one another.

Example 14-4. A two-tier Flume configuration using a spooling directory source and an HDFS sink

```
# First-tier agent

agent1.sources = source1
agent1.sinks = sink1
agent1.channels = channel1

agent1.sources.source1.channels = channel1
agent1.sinks.sink1.channel = channel1

agent1.sources.source1.type = spooldir
agent1.sources.source1.spoolDir = /tmp/spooldir

agent1.sinks.sink1.type = avro
agent1.sinks.sink1.hostname = localhost
agent1.sinks.sink1.port = 10000

agent1.channels.channel1.type = file
agent1.channels.channel1.checkpointDir=/tmp/agent1/file-channel/checkpoint
agent1.channels.channel1.dataDirs=/tmp/agent1/file-channel/data

# Second-tier agent

agent2.sources = source2
agent2.sinks = sink2
agent2.channels = channel2

agent2.sources.source2.channels = channel2
agent2.sinks.sink2.channel = channel2

agent2.sources.source2.type = avro
agent2.sources.source2.bind = localhost
agent2.sources.source2.port = 10000

agent2.sinks.sink2.type = hdfs
agent2.sinks.sink2.hdfs.path = /tmp/flume
agent2.sinks.sink2.hdfs.filePrefix = events
```

```
agent2.sinks.sink2.hdfs.fileSuffix = .log
agent2.sinks.sink2.hdfs.fileType = DataStream

agent2.channels.channel2.type = file
agent2.channels.channel2.checkpointDir=/tmp/agent2/file-channel/checkpoint
agent2.channels.channel2.dataDirs=/tmp/agent2/file-channel/data
```

The system is illustrated in Figure 14-4.

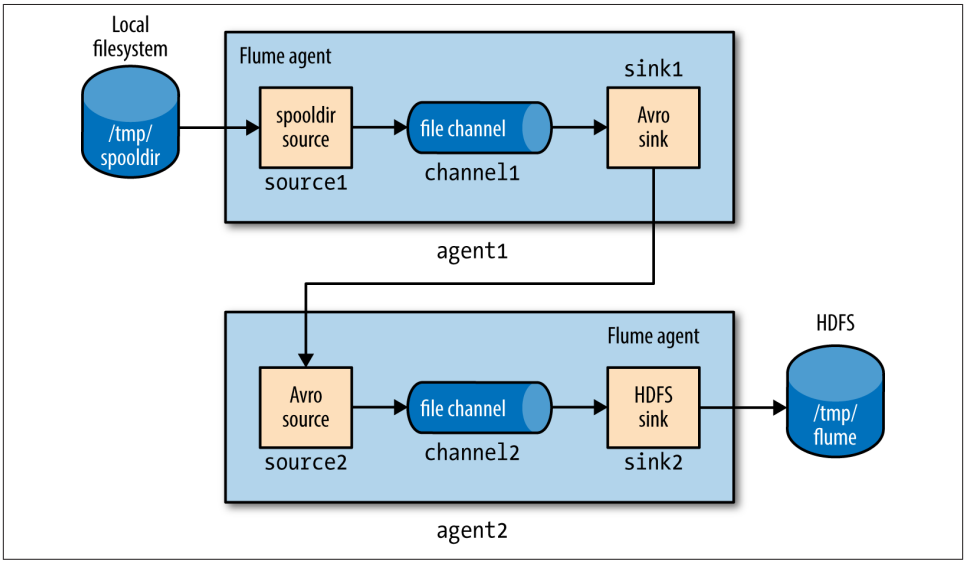

Figure 14-4. Two Flume agents connected by an Avro sink-source pair

Each agent is run independently, using the same `--conf-file` parameter but different agent `--name` parameters:

```
% flume-ng agent --conf-file spool-to-hdfs-tiered.properties --name agent1 ...
```

and:

```
% flume-ng agent --conf-file spool-to-hdfs-tiered.properties --name agent2 ...
```

Delivery Guarantees

Flume uses transactions to ensure that each batch of events is reliably delivered from a source to a channel, and from a channel to a sink. In the context of the Avro sink-source connection, transactions ensure that events are reliably delivered from one agent to the next.

The operation to read a batch of events from the file channel in `agent1` by the Avro sink will be wrapped in a transaction. The transaction will only be committed once the Avro

sink has received the (synchronous) confirmation that the write to the Avro source's RPC endpoint was successful. This confirmation will only be sent once `agent2`'s transaction wrapping the operation to write the batch of events to its file channel has been successfully committed. Thus, the Avro sink-source pair guarantees that an event is delivered from one Flume agent's channel to another Flume agent's channel (at least once).

If either agent is not running, then clearly events cannot be delivered to HDFS. For example, if `agent1` stops running, then files will accumulate in the spooling directory, to be processed once `agent1` starts up again. Also, any events in an agent's own file channel at the point the agent stopped running will be available on restart, due to the durability guarantee that file channel provides.

If `agent2` stops running, then events will be stored in `agent1`'s file channel until `agent2` starts again. Note, however, that channels necessarily have a limited capacity; if `agent1`'s channel fills up while `agent2` is not running, then any new events will be lost. By default, a file channel will not recover more than one million events (this can be overridden by its `capacity` property), and it will stop accepting events if the free disk space for its checkpoint directory falls below 500 MB (controlled by the `minimumRequiredSpace` property).

Both these scenarios assume that the agent will eventually recover, but that is not always the case (if the hardware it is running on fails, for example). If `agent1` doesn't recover, then the loss is limited to the events in its file channel that had not been delivered to `agent2` before `agent1` shut down. In the architecture described here, there are multiple first-tier agents like `agent1`, so other nodes in the tier can take over the function of the failed node. For example, if the nodes are running load-balanced web servers, then other nodes will absorb the failed web server's traffic, and they will generate new Flume events that are delivered to `agent2`. Thus, no new events are lost.

An unrecoverable `agent2` failure is more serious, however. Any events in the channels of upstream first-tier agents (`agent1` instances) will be lost, and all new events generated by these agents will not be delivered either. The solution to this problem is for `agent1` to have multiple redundant Avro sinks, arranged in a *sink group*, so that if the destination `agent2` Avro endpoint is unavailable, it can try another sink from the group. We'll see how to do this in the next section.

Sink Groups

A sink group allows multiple sinks to be treated as one, for failover or load-balancing purposes (see Figure 14-5). If a second-tier agent is unavailable, then events will be delivered to another second-tier agent and on to HDFS without disruption.

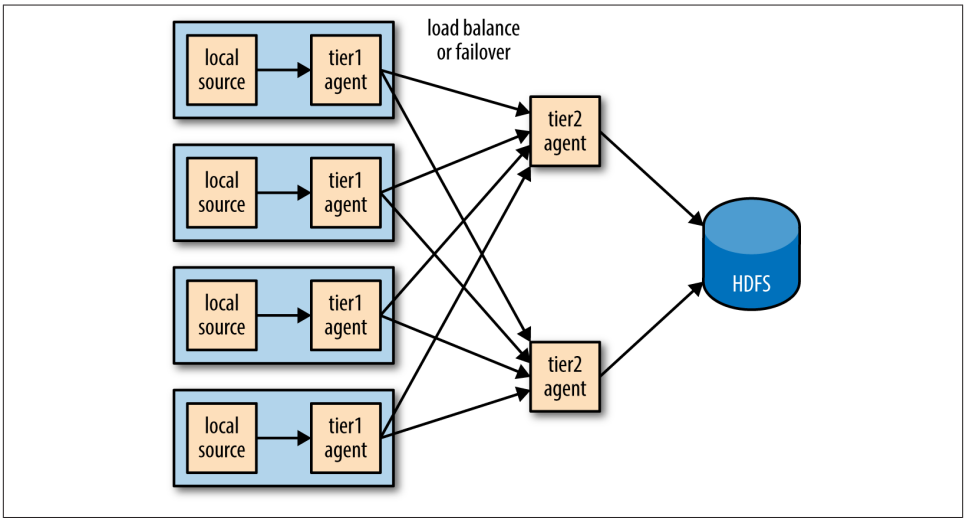

Figure 14-5. Using multiple sinks for load balancing or failover

To configure a sink group, the agent's `sinkgroups` property is set to define the sink group's name; then the sink group lists the sinks in the group, and also the type of the sink processor, which sets the policy for choosing a sink. Example 14-5 shows the configuration for load balancing between two Avro endpoints.

Example 14-5. A Flume configuration for load balancing between two Avro endpoints using a sink group

```
agent1.sources = source1
agent1.sinks = sink1a sink1b
agent1.sinkgroups = sinkgroup1
agent1.channels = channel1

agent1.sources.source1.channels = channel1
agent1.sinks.sink1a.channel = channel1
agent1.sinks.sink1b.channel = channel1

agent1.sinkgroups.sinkgroup1.sinks = sink1a sink1b
agent1.sinkgroups.sinkgroup1.processor.type = load_balance
agent1.sinkgroups.sinkgroup1.processor.backoff = true
```

```
agent1.sources.source1.type = spooldir
agent1.sources.source1.spoolDir = /tmp/spooldir

agent1.sinks.sink1a.type = avro
agent1.sinks.sink1a.hostname = localhost
agent1.sinks.sink1a.port = 10000

agent1.sinks.sink1b.type = avro
agent1.sinks.sink1b.hostname = localhost
agent1.sinks.sink1b.port = 10001

agent1.channels.channel1.type = file
```

There are two Avro sinks defined, `sink1a` and `sink1b`, which differ only in the Avro endpoint they are connected to (since we are running all the examples on localhost, it is the port that is different; for a distributed install, the hosts would differ and the ports would be the same). We also define `sinkgroup1`, and set its sinks to `sink1a` and `sink1b`.

The processor type is set to `load_balance`, which attempts to distribute the event flow over both sinks in the group, using a round-robin selection mechanism (you can change this using the `processor.selector` property). If a sink is unavailable, then the next sink is tried; if they are all unavailable, the event is not removed from the channel, just like in the single sink case. By default, sink unavailability is not remembered by the sink processor, so failing sinks are retried for every batch of events being delivered. This can be inefficient, so we have set the `processor.backoff` property to change the behavior so that failing sinks are blacklisted for an exponentially increasing timeout period (up to a maximum period of 30 seconds, controlled by `processor.selector.maxTimeOut`).

There is another type of processor, `failover`, that instead of load balancing events across sinks uses a preferred sink if it is available, and fails over to another sink in the case that the preferred sink is down. The failover sink processor maintains a priority order for sinks in the group, and attempts delivery in order of priority. If the sink with the highest priority is unavailable the one with the next highest priority is tried, and so on. Failed sinks are blacklisted for an increasing timeout period (up to a maximum period of 30 seconds, controlled by `processor.maxpenalty`).

The configuration for one of the second-tier agents, `agent2a`, is shown in Example 14-6.

Example 14-6. Flume configuration for second-tier agent in a load balancing scenario

```
agent2a.sources = source2a
agent2a.sinks = sink2a
agent2a.channels = channel2a

agent2a.sources.source2a.channels = channel2a
agent2a.sinks.sink2a.channel = channel2a

agent2a.sources.source2a.type = avro
agent2a.sources.source2a.bind = localhost
agent2a.sources.source2a.port = 10000

agent2a.sinks.sink2a.type = hdfs
agent2a.sinks.sink2a.hdfs.path = /tmp/flume
agent2a.sinks.sink2a.hdfs.filePrefix = events-a
agent2a.sinks.sink2a.hdfs.fileSuffix = .log
agent2a.sinks.sink2a.hdfs.fileType = DataStream

agent2a.channels.channel2a.type = file
```

The configuration for `agent2b` is the same, except for the Avro source port (since we are running the examples on localhost) and the file prefix for the files created by the HDFS sink. The file prefix is used to ensure that HDFS files created by second-tier agents at the same time don't collide.

In the more usual case of agents running on different machines, the hostname can be used to make the filename unique by configuring a host interceptor (see Table 14-1) and including the `%{host}` escape sequence in the file path, or prefix:

```
agent2.sinks.sink2.hdfs.filePrefix = events-%{host}
```

A diagram of the whole system is shown in Figure 14-6.

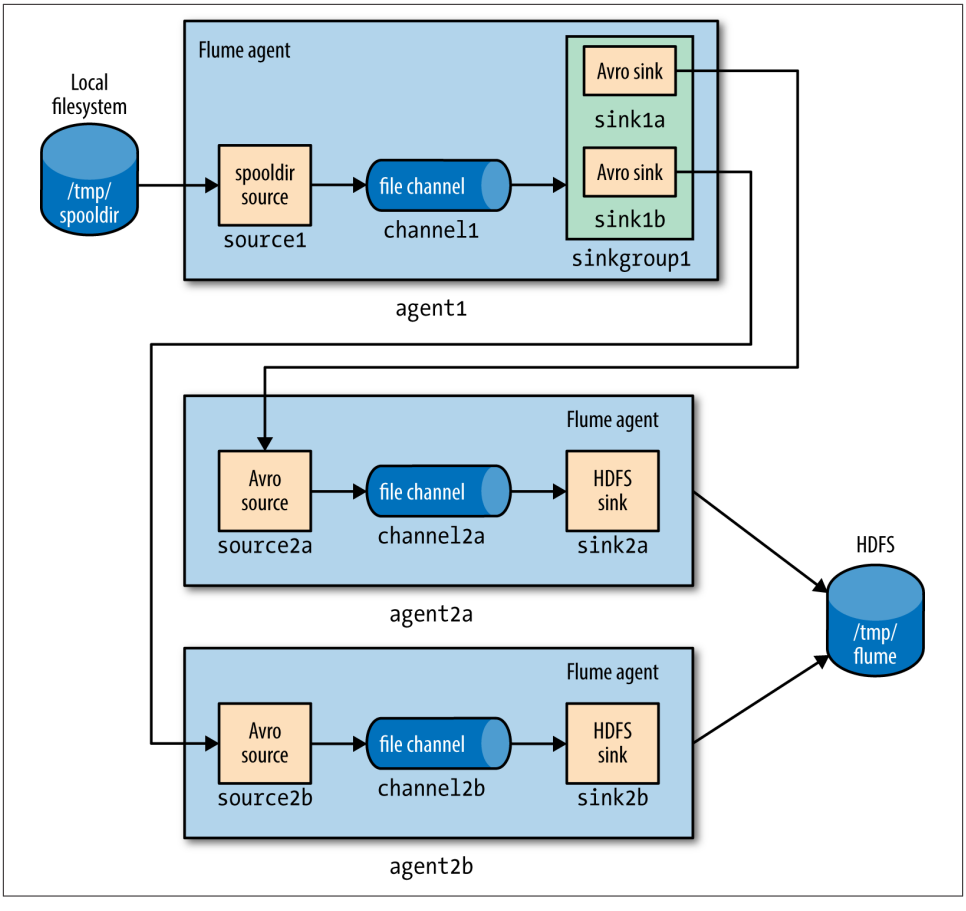

Figure 14-6. Load balancing between two agents

Integrating Flume with Applications

An Avro source is an RPC endpoint that accepts Flume events, making it possible to write an RPC client to send events to the endpoint, which can be embedded in any application that wants to introduce events into Flume.

The *Flume SDK* is a module that provides a Java `RpcClient` class for sending `Event` objects to an Avro endpoint (an Avro source running in a Flume agent, usually in another tier). Clients can be configured to fail over or load balance between endpoints, and Thrift endpoints (Thrift sources) are supported too.

The Flume *embedded agent* offers similar functionality: it is a cut-down Flume agent that runs in a Java application. It has a single special source that your application sends Flume `Event` objects to by calling a method on the `EmbeddedAgent` object; the only sinks

that are supported are Avro sinks, but it can be configured with multiple sinks for failover or load balancing.

Both the SDK and the embedded agent are described in more detail in the Flume Developer Guide.

Component Catalog

We've only used a handful of Flume components in this chapter. Flume comes with many more, which are briefly described in Table 14-1. Refer to the Flume User Guide for further information on how to configure and use them.

Table 14-1. Flume components

Category	Component	Description
Source	Avro	Listens on a port for events sent over Avro RPC by an Avro sink or the Flume SDK.
	Exec	Runs a Unix command (e.g., `tail -F`*`/path/to/file`*) and converts lines read from standard output into events. Note that this source cannot guarantee delivery of events to the channel; see the spooling directory source or the Flume SDK for better alternatives.
	HTTP	Listens on a port and converts HTTP requests into events using a pluggable handler (e.g., a JSON handler or binary blob handler).
	JMS	Reads messages from a JMS queue or topic and converts them into events.
	Netcat	Listens on a port and converts each line of text into an event.
	Sequence generator	Generates events from an incrementing counter. Useful for testing.
	Spooling directory	Reads lines from files placed in a spooling directory and converts them into events.
	Syslog	Reads lines from syslog and converts them into events.
	Thrift	Listens on a port for events sent over Thrift RPC by a Thrift sink or the Flume SDK.
	Twitter	Connects to Twitter's streaming API (1% of the firehose) and converts tweets into events.
Sink	Avro	Sends events over Avro RPC to an Avro source.
	Elasticsearch	Writes events to an Elasticsearch cluster using the Logstash format.
	File roll	Writes events to the local filesystem.
	HBase	Writes events to HBase using a choice of serializer.
	HDFS	Writes events to HDFS in text, sequence file, Avro, or a custom format.
	IRC	Sends events to an IRC channel.
	Logger	Logs events at `INFO` level using SLF4J. Useful for testing.
	Morphline (Solr)	Runs events through an in-process chain of Morphline commands. Typically used to load data into Solr.
	Null	Discards all events.
	Thrift	Sends events over Thrift RPC to a Thrift source.

Category	Component	Description
Channel	File	Stores events in a transaction log stored on the local filesystem.
	JDBC	Stores events in a database (embedded Derby).
	Memory	Stores events in an in-memory queue.
Interceptor	Host	Sets a `host` header containing the agent's hostname or IP address on all events.
	Morphline	Filters events through a Morphline configuration file. Useful for conditionally dropping events or adding headers based on pattern matching or content extraction.
	Regex extractor	Sets headers extracted from the event body as text using a specified regular expression.
	Regex filtering	Includes or excludes events by matching the event body as text against a specified regular expression.
	Static	Sets a fixed header and value on all events.
	Timestamp	Sets a `timestamp` header containing the time in milliseconds at which the agent processes the event.
	UUID	Sets an `id` header containing a universally unique identifier on all events. Useful for later deduplication.

Further Reading

This chapter has given a short overview of Flume. For more detail, see *Using Flume* by Hari Shreedharan (O'Reilly, 2014). There is also a lot of practical information about designing ingest pipelines (and building Hadoop applications in general) in *Hadoop Application Architectures* by Mark Grover, Ted Malaska, Jonathan Seidman, and Gwen Shapira (O'Reilly, 2014).