# Chapter 9. Log Management

Despite the best efforts of system administrators everywhere, logging in the cloud can quickly become more complicated (and more expensive) than logging in a physical hardware environment. Because EC2 instances come and go dynamically, the number of instances producing log files can grow and shrink at any time. Your logging system must therefore be designed with this in mind, to ensure that it keeps up with peaks in demand when processing log files.

Another area that requires some forward thinking is log storage. Running a large number of instances will produce large log files, which need to be stored somewhere. Without some advance planning, the storage requirements can grow rapidly, leading to an increase in costs.

This chapter presents some popular logging tools that can be useful in AWS environments and introduces some strategies for managing log files without breaking the bank. Logstash is used to demonstrate the concepts in this chapter, but the principles also apply to most other logging software.

# Central Logging

A common solution to the problem of viewing logs from multiple machines is to set up a central logging server to which all servers in your infrastructure send their logs. The central logging server is responsible for parsing and processing log files and managing the policies for log retention.

This pattern works well within AWS, with a few caveats. As mentioned earlier, it is critical to ensure that your logging system does not struggle to keep up when many instances are sending their log files to the central server.

Another potential issue is related to hostnames within EC2. Remember that, by default, most instances will set their hostname to match their internal IP address. This is not particularly useful when it comes to viewing log files. For this reason, many log-viewing tools provide a method of adding key/value pairs to log data. These can be used in conjunction with EC2 tags to make it easier to keep track of the source of a particular log entry.

Building a central logging system requires three main components. Log shippers and log receivers such as syslog, rsylog, and syslog-ng are responsible for sending and receiving log files. The third component is log viewers, such as Kibana and Graylog2, which handle the task of displaying this gathered data through a web interface. To further complicate things, most log shippers can also act as log receivers, and some packages provide all three components.

Unfortunately, a comparison of the various tools that can be used to build such a system is beyond the scope of this book, because the issues surrounding them are not really specific to AWS. However, one tool deserves special mention, because it has a few AWS-specific features.

Logstash is an open source log management tool that provides a framework for receiving, processing, and storing logs. It also includes a web interface for browsing logged data.

Like many log-receiving tools, it can accept data in the standard syslog format (RFC 3164). It also has a plug-in architecture that allows it to consume log data from a variety of additional sources. Most interestingly, it can read and write logs stored in S3 buckets, read logs from SQS queues, and write output messages to SNS topics. This integration with AWS services makes Logstash an ideal candidate for building a logging system within AWS.

Logstash itself consists of multiple components. It includes the core functionality of consuming and producing log files, as well as a built-in web interface (Kibana). It also comes with a built-in instance of Elasticsearch, which is a distributed search server based on Apache Lucene. Elasticsearch provides search capabilities, allowing you to quickly and easily find the log entries you are searching for.

All of these components can be run on a single machine for development and testing purposes. Once Logstash is dealing with a large amount of log data, these components can be moved to separate instances, so they can be scaled independently.

Unfortunately, Logstash is still relatively new. As a result, there is limited support for Logstash in popular package managers—it is not currently possible to install Logstash with a simple apt-get install logstash.

Of course, there are also many third-party logging services that can entirely obviate the need to build your own logging system. Many of these services provide convenient features, such as automatically retrieving EC2 tags and assigning them to log data, so that EC2 tags can be used to quickly drill down through log files.

## Logstash Configuration

To demonstrate Logstash in action, we will set up a simple centralized logging infrastructure suitable for use in EC2. Before going ahead and setting up the instances, we need to first prepare the security groups that will be used in the demonstration.

###### NOTE

To keep the demonstration simple, we will manually install and configure Logstash on the client and server instances. Of course, when it comes to moving this into production, the configuration should be handled by a tool such as Puppet. The Logstash documentation site contains links to [Logstash Puppet modules](http://cookbook.logstash.net/recipes/puppet-modules/) that can be used to install and configure the Logstash components.

Using the Management Console or command-line tools, create two security groups, named log\_client and log\_receiver. The rules for log\_client can be left empty.

The log\_receiver security group will be used for the Logstash server. It must be able to accept syslog traffic from logging clients and allow administrators to access the Kibana web interface and Elasticsearch API.

Create four rules in the log\_receiver group:

| **Rule** | **Description** |
| --- | --- |
| Inbound TCP 9292 from your network | Kibana web interface |
| Inbound TCP 9300 from your network | Elasticsearch API |
| Inbound TCP 5514 from log\_client security group | Syslog |
| Inbound TCP 22 from your network | SSH |

Create a single rule in the log\_client group:

| **Rule** | **Description** |
| --- | --- |
| Inbound TCP 22 from your network | SSH |

Once you save these changes, the security group configuration is complete.

### CREATING AND CONFIGURING A LOGSTASH SERVER

After creating the security groups, you can launch and configure the Logstash server instance. Launch a new EC2 instance using the Ubuntu LTS AMI (or your favorite distribution). Optionally, use Route 53 to set up a DNS record for this instance, so that logging.example.com points to the public DNS name of the new instance.

When the instance is ready, connect to it via SSH and download the Logstash application:

wget http://logstash.objects.dreamhost.com/release/logstash-1.2.1-flatjar.jar

Ensure that you have a recent version of Java:

apt-get -y install openjdk-7-jre-headless

Logstash is configured by way of a configuration file, the path to which is specified when launching Logstash. Create a file named logstash-central.conf containing the following contents:

input {

tcp {

type => syslog

port => 5000

}

}

output {

stdout { codec => rubydebug }

elasticsearch { embedded => true }

}

The Logstash config file consists of three sections, two of which are shown here. The inputsection specifies sources of logging data that Logstash will consume. The output section controls what Logstash does with this data after it has been filtered. The final section (filter) is not required for this simple setup. Filters are used to control the flow of messages and to add supplementary data to log entries, and are not used here.

###### NOTE

For more information on using filters to parse and modify syslog data, see the [Logstash documentation](http://cookbook.logstash.net/recipes/syslog-pri/).

In this case, all data is output to an instance of Elasticsearch, as well as stdout. We output to stdout so we can easily see logged messages on the console. The use of the console is for development purposes only and should be removed when moving into production.

The embedded => true parameter tells Logstash to use an embedded instance of Elasticsearch. That is, Logstash will take care of launching Elasticsearch and forwarding log data to it.

With the configuration file saved, launch Logstash with the following command:

java -jar logstash-1.2.1-flatjar.jar agent -f logstash-simple.conf -- web

This command launches the main Logstash process and the additional processes required to run the Kibana web interface (as denoted by the final two -- web arguments). Thus, we are effectively running two commands simultaneously here: one to launch the Logstash agent, and another to launch the Logstash web interface.

Launching Logstash might take some time, as it needs to extract the JAR file to the working directory before launching the processes required for the Logstash agent and web interface, as well as the embedded Elasticsearch instance. Once Logstash has launched, you should see something similar to the following:

< logstash startup output >

In another SSH session, verify that Logstash is listening for incoming syslog data on TCP port 5000 by using the netstat command:

root@ip-10-80-1-109:~# netstat -anlp | grep 5000

tcp6 0 0 :::5000 :::\* LISTEN 6544/java

Finally, use the netcat command to send a test message to Logstash. This message should be printed in the terminal session in which you started the Logstash agent:

echo "testing logging" | nc localhost 5000

Once this command is executed, you should see some output printed to the Logstash console. This is a JSON representation of the logged message, for example:

{

"message" => "testing logging",

"@timestamp" => "2013-09-09T09:10:51.402Z",

"@version" => "1",

"type" => "syslog",

"host" => "127.0.0.1:35804"

}

Finally, browse to the Kibana web interface with your web browser. If you set up a Route 53 record to point logging.example.com to your instance, you can visit the interface athttp://logging.example.com:9292/. Otherwise, use the public DNS name of your EC2 instance, such as the following: http://ec2-54-217-126-48.eu-west-1.compute.amazonaws.com:9292/.

Using Kibana’s search function, search for the string testing. The results window should then show you the test message you logged with the logger program.

### CONFIGURING THE LOGSTASH CLIENTS

With a server listening, you can move on to configuring the logging client that will represent the other servers in your infrastructure.

Launch a second EC2 instance and make it a member of the log\_client security group. Once the instance is ready to accept SSH connections, you can log in to download the Logstash agent and install Java:

wget http://logstash.objects.dreamhost.com/release/logstash-1.2.1-flatjar.jar

apt-get -y install openjdk-7-jre-headless

The Logstash client configuration will differ slightly from the central Logstash configuration. Instead of outputting your log files to Elasticsearch, you will instead send them to the central Logstash instance using the syslog protocol.

Create a client configuration file named logstash-client.conf with the following contents:

input {

file {

type => "syslog"

path => [ "/var/log/\*.log", "/var/log/syslog" ]

}

}

output {

stdout { codec => rubydebug }

tcp {

host => "logging.example.com"

port => 5000

}

}

As before, this configuration file defines the inputs and outputs that Logstash will use. The input in this example is a list of files. Logstash will read these files and ingest log entries as they are written. The path attribute accepts a list of paths, which can either be glob paths (including an asterisk or other wildcard) or paths to single files, as in the case of /var/log/syslog.

###### NOTE

If you did not set up a Route 53 DNS record for your central log server, you will need to add the instance’s public DNS name to the client configuration file, instead of logging.example.com.

Therefore, this configuration file will cause Logstash to monitor the specified files and send their input via TCP to the logging.example.com host, where syslog is listening on port 5000. We are not yet using any filters to modify or parse the logged data.

On the client server, start the Logstash agent:

java -jar logstash-1.2.1-flatjar.jar agent -f logstash-client.conf

Use the logger command to write some example text to the local syslog system:

echo "testing client logging" | logger

If everything is configured correctly, you should see the test message repeated in the Logstash receiver console. In addition, this message will be passed to Elasticsearch, so it can also be viewed by searching for testing in the Kibana web interface.

With those steps complete, the basic central logging system is up and running. Any log messages produced on the client system and written to one of the monitored files will be passed to the central Logstash server.

## Logging to S3

Like any critical component, the logging system should be loosely coupled to the other moving parts in your infrastructure. That is, a failure in the logging system should not propagate and cause other services to fail.

In the previous example, we set up a central Logstash server that will accept log messages from clients via TCP. What happens if the central Logstash instance crashes or otherwise becomes unavailable? In that case, clients will be unable to send log messages. Any attempt to do so would result in a broken pipe error message, because the client is unable to open a TCP connection to the central server. Fortunately, the Logstash client will recognize this failure and spool the log messages locally. Once the central Logstash server is back in action, these spooled messages will be resent.

###### NOTE

If we were using UDP, transmissions would fail silently, and messages will be lost instead of being stored at the senders. This may or may not be acceptable, depending on your log retention policies.

This process of storing the messages locally until they can be sent to the master is known as store and forward, and is similar to the way in which systems like email work. If the Logstash agent is only temporarily unavailable, storing the messages temporarily on the client will not cause any problems. However, prolonged outages might cause an excessive amount of spooled data to pile up on the client instance. This can cause problems if the temporary files grow so large that they interfere with the proper running of the client instance.

In this case, it can be helpful to have an intermediary storage location for your log files, to further decouple the client/master Logstash instances. One method of doing this is to use S3 as temporary storage for your log files: instead of sending its log files directly to the central Logstash server, the client writes all log files to an S3 bucket. The central Logstash agent is then responsible for regularly downloading these log files from S3 and processing them as usual.

The S3 intermediary has several benefits, the primary one being decoupling. Even if the central Logstash is unavailable for an extended period of time, you can be safe in the knowledge that your log files will be queued up on S3 and processed after the central log server is back in action.

A secondary benefit relates to scaling your logging system. Consider what would happen if the number of instances sending their log files to the central Logstash instance were to grow rapidly. Because log messages are sent to Logstash as soon as they are generated, logs are effectively processed in real time. Sending too many logs could cause the central instance to become overloaded.

By temporarily storing the files in S3, you can remove the instantaneous nature of the processing. Instead, the central server has more control over when log files are pulled from S3 for processing. While the amount of work remains the same, the peaks and troughs are evened out by storing the data on S3 and allowing the central server to pull it.

Because Logstash has built-in support for some Amazon services, including S3, modifying our existing system to support the new setup is extremely straightforward. We need to make two changes to the system. First, instead of clients sending files directly to the central server, they should be written to an S3 bucket. On the central server side of things, we need to tell Logstash to read incoming log files from the S3 bucket, instead of listening for TCP connections.

To begin with, we need to create a new S3 bucket to store our log files. Using the Management Console or command-line tools, create a new bucket with a name like logs-example-com. Remember that S3 bucket names must be unique, so you will not be able to use this exact example.

Once the bucket is created, create a new IAM user named logging. This will be used on both the client and central Logstash instances to read and write to the bucket.

Assign an IAM policy to the new user, granting it permissions to read from and write to the logging bucket. Here is an example IAM policy:

{

"Statement": [

{

"Action": "s3:\*",

"Effect": "Allow",

"Resource": [

"arn:aws:s3:::my-s3-bucket",

"arn:aws:s3:::my-s3-bucket/\*"

]

}

]

}

Note that this policy explicitly references the S3 bucket and IAM users. You will need to change the example to match the name of your S3 bucket and the ID of your IAM user.

Once the policy has been assigned to the IAM user, the AWS side of the configuration is complete, and you can return to the EC2 instances running your Logstash client and server.

Begin by configuring the Logstash client so that it writes log entries to the S3 bucket. Create a configuration file named logstash-client-s3.conf with the following contents:

input {

file {

type => "syslog"

path => [ "/var/log/\*.log", "/var/log/syslog" ]

}

}

output {

s3 {

access\_key\_id => "my-aws-access-key-id"

secret\_access\_key => "my-aws-secret-access-key"

endpoint\_region => "eu-west-1"

bucket => "logging-example-com"

}

}

You will, of course, need to update this example to reflect the name of your S3 bucket and the AWS IAM access credentials.

On the central Logstash instance, create a configuration file named logstash-central-s3.confwith the following contents:

input {

s3 {

bucket => "logging-example-com"

credentials => ["my-aws-access-key-id", "my-aws-secret-access-key"]

region => "eu-west-1"

}

}

output {

stdout { codec => rubydebug }

elasticsearch { embedded => true }

}

The Logstash S3 extension downloads data from the bucket using the credentials you configure here. So the server’s input section is totally new in this example, but the output section is the same as our original one. Again, you will need to replace the S3 bucket name and IAM access credentials with your own data.

###### NOTE

Please note the different configuration file options required for the S3 input and S3 output. Logstash has not yet settled on conventions for naming attributes in plug-ins, so each plug-in author chooses his own variable naming scheme. The documentation for plug-ins also varies dramatically in terms of quality and professionalism. Hopefully, both of these will improve as Logstash continues to grow in popularity.

With these changes in place, you will need to stop and restart the Logstash agents on both the client and central servers. Start the Logstash process on the client:

java -jar logstash-1.2.1-flatjar.jar agent -f logstash-client-s3.conf

The central Logstash instance can be started as follows:

java -jar logstash-1.2.1-flatjar.jar agent -f logstash-central-s3.conf

Once both processes are running, create some log file entries on the client server with a logger command:

echo "testing s3 logger" | logger

After a few minutes, this log message should be printed in the central Logstash agent log. This will take a little longer than the preceding examples, because you will need to wait for the Logstash client to write this message to the S3 bucket, and then wait again for the central Logstash agent to retrieve the updated file and process its contents.

This method of using S3 as a temporary storage location greatly increases the reliability of this logging system, because it is no longer dependent on having the central Logstash agent running. Of course, without the central agent running, log files will not be processed or visible in the Kibana web interface. However, a failure in the central agent will have no effect on client instances, which will happily continue shipping their log files to the temporary S3 bucket for later processing.

# AWS Service Logs

So far, we have been looking at application and operating system logs, but another class of logs must also be considered. Many of the AWS services produce log files that might need to be stored and reviewed. For example, a CloudFront distribution will produce log files providing details about requests it receives, such as the URL that was requested or the resulting HTTP response code.

All of Amazon’s services use the same basic logging methodology: logs are written to a specified S3 bucket at regular intervals. This makes retrieving and processing the log files very simple. You just need to regularly download and process the files. The kind of decoupling described in the previous section is already built into this system: if you do not process the log files, they will pile up in the S3 bucket, but CloudFront will continue to function as normal.

Given that we already have a system for processing log files that have been written to an S3 bucket, we can reuse the example from the previous section to read CloudFront logs, as well as our application and operating system logs. Logstash is already configured to process logs from an S3 bucket, so we can easily add another section to our central Logstash agent’s configuration file to make it process CloudFront log files.

The first step is to create a CloudFront distribution to serve some static or dynamic files and configure that distribution to store its access logs in an S3 bucket. This is described in Amazon’s [CloudFront documentation](http://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/AccessLogs.html). During this process, you will have the option to create a new S3 bucket in which to store the logs, or enter the name of an existing bucket. Either way, keep track of the bucket name you choose, because this will be required in the following steps. For the demonstration, I have used a bucket named cloudfront-logs-example-com.

Once these steps in the Amazon documentation have been completed, you will have a CloudFront distribution that writes its access logs to an S3 bucket periodically. Next, you can configure Logstash to consume these logs, feeding them into the same system that processes your application and operating system logs.

On the central Logstash instance, stop the logstash process if it is still running from the previous example. Modify the logstash-central-s3.conf file so that it matches the following:

input {

s3 {

bucket => "logging-example-com"

credentials => ["my-aws-access-key-id", "my-aws-secret-access-key"]

region => "eu-west-1"

}

s3 {

bucket => "cloudfront-logs-example-com"

credentials => ["my-aws-access-key-id", "my-aws-secret-access-key"]

region => "eu-west-1"

type => "cloudfront"

}

}

output {

stdout { codec => rubydebug }

elasticsearch { embedded => true }

}

The newly inserted s3 section configures Logstash so that it will read log files from the CloudFront log bucket, as well as the original logging-example-com bucket.

All logs retrieved from the cloudfront-logs-example-com bucket will have their type attribute set to cloudfront. You can refer to this to keep track of the source of log data, and the type will be visible when these logs are viewed in the Kibana web interface.

After saving the file, start Logstash again with this command:

java -jar logstash-1.2.1-flatjar.jar agent -f logstash-central-s3.conf

To see this in action, you will need to wait for CloudFront to write the first log file, which it will do after receiving a certain number of HTTP requests. For testing, it can be helpful to use cURL to quickly make a large number of requests to your CloudFront distribution, which will cause the access log to be written to. Once this file reaches a certain size, it will be written to S3, at which point Logstash will notice the new file and process the logs contained therein.

Other Amazon services, such as Elastic Beanstalk and S3 itself, use the same mechanism for storing access logs, so this technique can also be reused for those services.

# S3 Life Cycle Management

Managing ever-growing log files is an old problem for systems administrators. Working in the AWS cloud introduces some additional challenges, but the same principles can be used to solve the problem. On an individual system, logrotate is used to ensure that log files are regularly rotated and deleted. Without logrotate, log files might grow to the point where they exhaust all available space on the system, causing problems for running applications.

Storing logs on S3 creates a different problem: instead of worrying about shrinking available capacity, you need to worry about an increasing AWS bill. Constantly throwing log files into S3 buckets and ignoring them will lead to an unnecessarily high bill at the end of the month.

S3 life cycles can be used to manage this problem, by allowing you to create rules that control when your data is automatically archived to Glacier or permanently deleted. If you are logging to S3, you should ensure that your life cycle rules are configured to automatically delete objects when they are no longer required.

Life cycle rules can also be used to potentially increase the security of your log files after they have been moved to storage. In some scenarios, log files should be considered read-only after they have been written. In strict cases, this is enforced using WORM (write once, read many) drives, which provide physical protection to prevent modification of files after they have been written.

While life cycle rules cannot provide this level of protection, they can be used to separate the credentials used for reading and writing operations. For example, imagine you write your log files to an S3 bucket and they are stored in /backups/logs/. Your logging application uses a set of IAM credentials that give it permission to write to this location in the bucket.

You would like to ensure that, once log files have been written, it would be difficult for a malicious user or application to overwrite them. This can be done by configuring life cycle rules to Archive and Delete the objects after a certain time frame. After this interval, the log files would be moved to Glacier, where they would be inaccessible to the IAM credentials used to create the files.