SYSADMIN

Solving Rsyslog Performance Issues

DAVID LANG

David Lang is a site reliability engineer at Google. He spent more than a decade at Intuit working in the Security Department for the Banking Division. He was introduced to Linux in 1993 and has been making his living with Linux since 1996. He is an Amateur Extra Class Radio Operator and served on the communications staff of the Civil Air Patrol, California Wing, where his duties included managing the statewide digital wireless network. He was awarded the 2012 Chuck Yerkes award for his participation on various open source mailing lists. david@lang.hm

Rsyslog is a very fast modern logging daemon, but with its capabilities comes complexity: When things go wrong, it's sometimes hard to tell what the problem is and how to fix it. To continue my series of logging articles [1], this time I will be talking about how to troubleshoot performance issues with rsyslog.

Common Bottlenecks and Solutions

By far, the most common solution to poor performance is to upgrade to the current version of rsyslog. Performance is considered a key feature of rsyslog, and the improvements from one major version to another can be drastic. In many common rulesets, going from 5.x to 7.x has resulted in >10x performance improvements.

The next most common performance bottleneck is name resolution. Rsyslog will try to do a reverse lookup for the IP of any system sending it log messages. With v7, it will cache the results, but if your name lookups time out, this doesn't help much. If you cannot get a fast name server or put the names into /etc/hosts, consider disabling DNS lookups with the -x command line flag.

For people using dynamically generated file names, a very common problem is failing to increase the number of filehandles that rsyslog keeps open. Historically, rsyslog keeps only ten dynamically generated output files open per action. If you commonly have logs arriving for more than this small number, rsyslog needs to close a file (flushing pending writes), open a new file, and write to that file for each new log line that it's processing. To fix this, set the $DynaFileCacheSize parameter to some number larger than the number of files that you expect to write to (and make sure your filehandle limits allow this).

The last of the common bottlenecks is contention on the queues. If every thread accessing the queue locks it for every message, it's very possible for contention for the queue logs to become a significant bottleneck. In v5, rsyslog gained the ability to process batches of messages, and over time more modules have been getting updated to support this mode. The default batch size ($ActionQueueDequeueBatchSize) was initially 16 messages at a time; however, on dedicated, central servers, it may be appropriate to set this limit much higher. In v8, the default is being changed to 1024, and even more may be appropriate on a dedicated server. The benefit of increasing this number tapers off with size, but setting it to 1024 or so to see if there is a noticeable difference is a very reasonable early step. The discussion below will help you determine if you want to go further.

Rsyslog and Threads

Beyond these most common causes, things get more difficult. It is necessary to track down what is actually the bottleneck and address it. This task is complex because rsyslog makes heavy use of threads to decouple pieces from each other and to take advantage of modern systems, but this structure also provides some handles to use to track down the issues.

Each input to rsyslog is through one or more threads, which gather the log messages and add them to the main queue. Worker threads then pull messages off the main queue and deliver them to their destinations and/or add the message to an action queue. If there are action

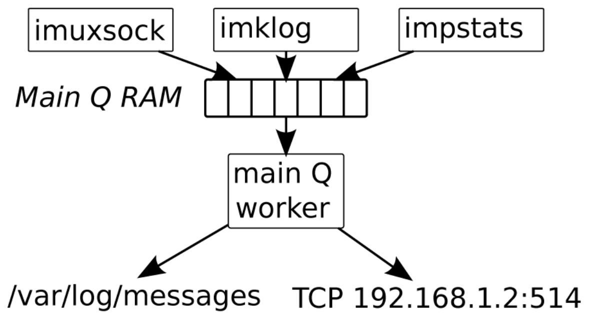

Figure 1: The flow of logs in a basic rsyslog configuration

queues, each one has its own set of worker threads pulling from the action queue and delivering to the destinations for that action.

If a worker is unable to deliver messages to a destination, all progress of that queue will block until that delivery is able to succeed (or it hits the retry limit and permanently fails). If you don't want this to block all log processing, you should make an action queue for that destination (or group of destinations).

As an example, I'll show a basic configuration that accepts logs from the kernel and from /dev/log, writes all the messages locally, and delivers them via TCP to a remote machine.

Legacy config:

```
$ModLoad imklog
$modLoad imuxsock
$SystemLogRateLimitInterval 0
$SystemLogSocketAnnotate on
*.* /var/log/messages
*.* @@192.168.1.1:514
```

Version 7 and later config:

```
module(load="imklog")
module(load="imuxsock" SysSock.RateLimit.Interval="0"
  SysSock.Annotate="on")
action(type="omfile" File="/var/log/messages")
action(type="omfwd" Target="192.168.2.11" Port="10514"
  Protocol="tcp")
```

This will create three threads in addition to the parent "housekeeping" thread. Figure 1 shows the data flow through rsyslog. The threads do not communicate directly with each other, and no one thread "owns" the queue. The housekeeping thread isn't shown here, because it doesn't have any role in the processing of log messages.

With top, you can see these threads by pressing H, and with ps, you can see these as well:

```
# ps  -eLl |grep `cat /var/run/rsyslogd.pid`
5 S 0 758  1 758  0  80   0 - 18365 poll_s ?   00:00:00 rsyslogd
1 S 0 758  1 763  0  80   0 - 18365 poll_s ?   00:00:05
  in:imuxsock
1 S 0 758  1 764  0  80   0 - 18365 syslog ?   00:00:00 in:imklog
1 S 0 758  1 765  0  80   0 - 18365 futex_ ?   00:00:02 rs:main
  Q:Reg
```

argv[0] is changed to tag each thread with what it's doing. This lets you see if any of the threads are pegging the CPU, or if the main Q worker thread is just doing nothing.

This is, of course, a "Hello World" configuration, but it shows some of the types of issues that are common for people to run into in larger setups. For example, this configuration is dependent on a remote system; if that system is down, no local logs will be processed and we will see logs queue up and eventually fill the queue, causing programs trying to write logs to block. If we want to continue logging locally, even if the remote system is down, we can create a default-sized action queue for the TCP output action. To handle longer outages, or restarts of rsyslog, the queue can be disk backed. A full discussion of queue configuration and management is a topic that will require its own article.

Legacy config:

```
$ModLoad imklog
$modLoad imuxsock
$SystemLogRateLimitInterval 0
$SystemLogSocketAnnotate on
*.* /var/log/messages
$ActionQueueType FixedArray
*.* @@192.168.1.1:514
```

Version 7 config:

```
module(load="imklog")
module(load="imuxsock" SysSock.RateLimit.Interval="0"
  SysSock.Annotate="on")
action(type="omfile" File="/var/log/messages")
action(type="omfwd" Target="192.168.2.11" Port="514"
  Protocol="tcp" queue.type="FixedArray")
```

Now, you can see an additional queue worker for the action queue:

```
# ps  -eLl |grep `cat /var/run/rsyslogd.pid`
    5 S 0 458  1 458  0  80   0 - 20414 poll_s ?  00:00:00 rsyslogd
    1 S 0 458  1 462  0  80   0 - 20414 poll_s ?  00:00:00 in:imuxsock
    1 S 0 458  1 463  0  80   0 - 20414 syslog ?  00:00:00 in:imklog
    5 S 0 458  1 464  0  80   0 - 20414 futex_ ?  00:00:00
      rs:main Q:Reg
    1 S 0 458  1 465  0  80   0 - 20414 futex_ ?  00:00:00
      rs:action  2 que
```

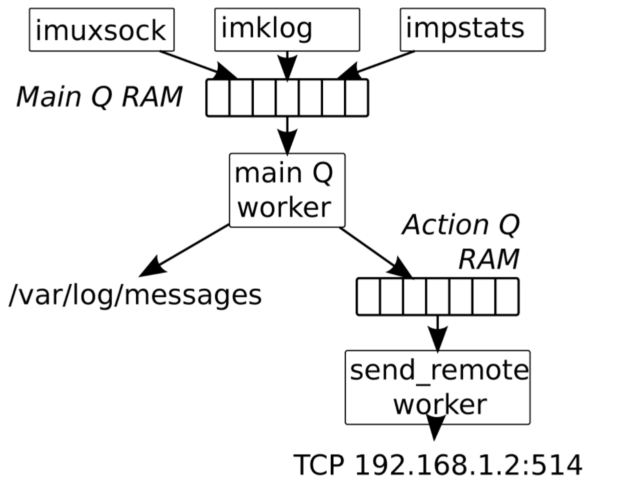

Figure 2: The flow of logs when an action queue is added to preventing blocking

With v7+, you can add a parameter to name the queue:

```
action(name="send_remote" type="omfwd" Target="192.168.2.11"
  Port="514"
  Protocol="tcp" queue.type="FixedArray" )
```

This changes the data flow to what you see in Figure 2, and the ps/top display shows the additional threads as well:

```
# ps  -eLl |grep `cat /var/run/rsyslogd.pid`
5 S 0 971  2807 971  0  80   0 - 20414 poll_s ?  00:00:00
  rsyslogd
1 S 0 971  2807 972  0  80   0 - 20414 poll_s ?  00:00:00
  in:imuxsock
1 S 0 971  2807 973  0  80   0 - 20414 syslog ?  00:00:00
  in:imklog
5 S 0 971  2807 974  0  80   0 - 20414 futex_ ?  00:00:00
  rs:main Q:Reg
1 S 0 971  2807 975  0  80   0 - 20414 futex_ ?  00:00:00
  rs:send_remote:
```

Keeping track of all the pieces can be a bit difficult when you have multiple queues in use.

Through the rest of the article, I will just give the v7 format because of the increasing complexity of specifying options with the legacy config style. Not all of the features described are going to be available on older versions.

Although ps/top lets you see how much CPU is being used by the threads, it doesn't tell you what is being done. Rsyslog includes the impstats module, which produces a lot of information about what's going on inside rsyslog.

To load impstats, add the following line to the top of your config file (it needs to be ahead of many other configuration parameters, so it's easiest to make it the very first line).

```
module(load="impstats" interval="60" resetCounters="on"
   format="legacy")
```

This statement will create a set of outputs every minute, resetting counters every time, which makes it very easy to see if a queue is backing up. A sample output looks like the following (timestamp and hostname trimmed for space):

```
rsyslogd-pstats: imuxsock: submitted=3 ratelimit.discarded=0
  ratelimit.numratelimiters=2

rsyslogd-pstats: action 1: processed=4 failed=0
  suspended=0 suspended.duration=0 resumed=0

rsyslogd-pstats: send_remote: processed=4 failed=0
  suspended=0 suspended.duration=0 resumed=0

rsyslogd-pstats: resource-usage: utime=1536 stime=60107
  maxrss=1280 minflt=386 majflt=0
  inblock=0 oublock=0 nvcsw=22 nivcsw=32

rsyslogd-pstats: send_remote: size=0 enqueued=4 full=0
  discarded.full=0 discarded.nf=0

maxqsize=1
rsyslogd-pstats: main Q: size=5 enqueued=9 full=0
  discarded.full=0
    discarded.nf=0 maxqsize=5
```

You can use an automated analyzer to find the most common types of problems. Upload your pstats logs to http://www.rsyslog.com/impstats-analyzer/, and the script will highlight several common types of problems.

If you are using JSON-formatted messages, you can change format from "legacy" to "cee" and then use the mmjsonparse module to break this down into individual variables for analysis. In this format, the logs look like:

```
rsyslogd-pstats: @cee: {"name":"imuxsock","submitted":7,
  "ratelimit.discarded":0,"ratelimit.numratelimiters":3}

rsyslogd-pstats: @cee: {"name":"action 1","processed":8,
  "failed":0,"suspended":0, "suspended.duration":0,
  "resumed":0}

rsyslogd-pstats: @cee: {"name":"action 2","processed":8,
  "failed":0,"suspended":0, "suspended.duration":0,
  "resumed":0}

rsyslogd-pstats: @cee: {"name":"send_remote","processed":8,"
  failed":0, "suspended":0, "suspended.duration":0,
  "resumed":0}
```

```
rsyslogd-pstats: @cee: {"name":"resource-usage","utime":2917,
  "stime":3237,"maxrss":1520, "minflt":406,"majflt":0,"inblock":0,
  "oublock":0,"nvcsw":30,"nivcsw":6}

rsyslogd-pstats: @cee: {"name":"send_remote","size":0,
   "enqueued":8,
    "full":0,"discarded.full":0, "discarded.nf":0,"maxqsize":1}

rsyslogd-pstats: @cee: {"name":"main Q","size":6,"enqueued":14,
  "full":0, "discarded.full":0,"discarded.nf":0,"maxqsize":6}
```

So far, you've just put the pstats logs into the main queue along with all the other logs in the system. If something causes that queue to back up, it will delay the pstats logs as well.

There are two ways to address this: First, you can configure rsyslog to write the pstat logs to a local file in addition by adding the log.file="/path/to/local/file" parameter to the module load line. The second approach is a bit more complicated, but it serves to show an example of another feature of rsyslog—rulesets and the ability to bind a ruleset to a specific input.

Take the example config and extend it to be the following:

```
module(load="impstats" interval="10" resetCounters="on"
  format="legacy" ruleset="high_p")
module(load="imklog")
module(load="imuxsock" SysSock.RateLimit.Interval="0"
  SysSock.Annotate="on")
action(type="omfile" File="/var/log/messages")
action(name="send_remote" type="omfwd" Target="192.168.2.11"
  Port="514" Protocol="tcp"queue.type="FixedArray" )

ruleset(name="high_p" queue.type="FixedArray"){
  action(type="omfile" File="/var/log/pstats")
  action(name="send_HP" type="omfwd" Target="192.168.2.11"
    Port="514"
      Protocol="tcp" queue.type="FixedArray" )
}
```

All pstat log entries will now go into a separate "main queue" named "high_p" with its own worker thread and its own separate queue to send the messages remotely. This is effectively the same as starting another stand-alone instance of rsyslog just to process these messages. There is no interaction (other than the house-keeping thread) between the threads processing the pstat messages and the threads processing other messages (see Figure 3).

```
# ps  -eLlww |grep `cat /var/run/rsyslogd.pid`
5 S 0 827  2807 827  0  80 0 - 31181 poll_s ?  00:00:00 rsyslogd
5 S 0 827  2807 828  0  80 0 - 31181 poll_s ?  00:00:00
  in:impstats
1 S 0 827  2807 829  0  80 0 - 31181 syslog ?  00:00:00 in:imklog
1 S 0 827  2807 830  0  80 0 - 31181 poll_s ?  00:00:00
  in:imuxsock
```

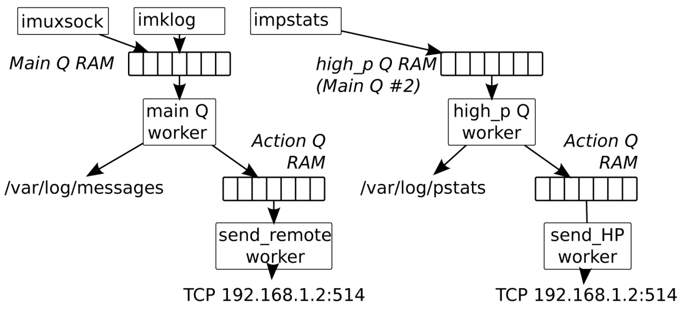

Figure 3: The flow of logs through the threads and queues with impstats bound to a ruleset

```
5 S 0 827  2807 831  0  80 0 - 31181 futex_ ?  00:00:00
  rs:main  Q:Reg
1 S 0 827  2807 832  0  80 0 - 31181 futex_ ?  00:00:00
  rs:send_remote:
5 S 0 827  2807 843  0  80 0 - 31181 futex_ ?  00:00:00
  rs:high_p:Reg
1 S 0 827  2807 844  0  80 0 - 31181 futex_ ?  00:00:00
  rs:send_  HP:Reg
```

Pstats output also shows the additional queues:

```
rsyslogd-pstats: imuxsock: submitted=0 ratelimit.discarded=0
  ratelimit.numratelimiters=0

rsyslogd-pstats: action 1: processed=0 failed=0 suspended=0
  suspended.duration=0 resumed=0

rsyslogd-pstats: send_remote: processed=0 failed=0
  suspended=0 suspended.duration=600 resumed=0

rsyslogd-pstats: action 3: processed=10 failed=0 suspended=0
  suspended.duration=0 resumed=0

rsyslogd-pstats: send_HP: processed=10 failed=0 suspended=0
  suspended.duration=600 resumed=0

rsyslogd-pstats: resource-usage: utime=26978 stime=26416
  maxrss=2056 minflt=857 majflt=0 inblock=0 oublock=400
  nvcsw=412 nivcsw=11

rsyslogd-pstats: send_remote: size=0 enqueued=0 full=0
  discarded.full=0 discarded.nf=0 maxqsize=2

rsyslogd-pstats: send_HP: size=0 enqueued=10 full=0 discarded.
  full=0 discarded.nf=0 maxqsize=10

rsyslogd-pstats: high_p: size=8 enqueued=10 full=0 discarded.
  full=0 discarded.nf=0 maxqsize=10

rsyslogd-pstats: main Q: size=0 enqueued=0 full=0 discarded.
  full=0 discarded.nf=0 maxqsize=2
```

Once you find the actual bottleneck in your configuration, what can you do about it? It boils down to a two-pronged attack.

Use More Cores to Perform the Work

This approach is tricky. In many cases enabling more worker threads will help, but you will need to check the documentation for the module that is your bottleneck to find what options you have. In many cases, the modules end up needing to serialize (e.g., you only want one thread writing to a file at a time), and if your bottleneck is in one of these areas, just adding more workers won't help. Writing to a file can be split up by moving most of the work away from the worker thread that is doing the testing of conditions, using a second thread to format the lines for the file, and, if needed, using a third thread to write the data to the file, potentially compressing it in the process.

Restructure the Configuration to Reduce the Amount of Work

This approach involves standard simplification and refactoring work for the most part. Rsyslog has lots of flexibility in terms of what modules can be written to do; so, in an extreme case, a custom module may end up being written to address a problem. In most cases, however, it's simplifying regex expressions, refactoring to reduce the number of tests needed or to make it easier for the config optimizer to detect the patterns. Like most programming, algorithmic changes usually produce gains that dwarf other optimization work, so it's worth spending time looking at ways to restructure your configuration.

Some Examples of Restructuring

If you have several actions that are related to one destination, instead of creating a separate queue for each action, you can create a ruleset containing all the actions, and then call the ruleset with a queue.

```
ruleset(name="rulesetname" queue/type="FixedArray"){
  action(type="omfwd" Target="192.168.2.11" Port="514"
    Protocol="tcp")
}
```

Then, in the main ruleset, you can replace the existing actions with:

```
call rulesetname
```

A ruleset can contain any tests and actions that you can have in a normal rsyslog ruleset, including calls to other rulesets.

With the v7 config optimizer, zero overhead is incurred in using a ruleset that doesn't have a queue, so you can also use rulesets to clarify and simplify your rulesets. If you find that you have a lot of format rules:

```
if $hostname == "host1" and $programname = "apache" then {
  /var/log/apache/host1.log
  stop
}
if $hostname == "host1" and $programname = "apache" then {
  /var/log/postfix/host2.log
  stop
}
if $hostname == "host2" and $programname = "postfix" then {
  /var/log/apache/host1.log
  stop
}
if $hostname == "host2" and $programname = "postfix" then {
  /var/log/postfix/host2.log
  stop
}
/var/log/other-logs
```

you can simplify it into:

```
$template multi_test='/var/log/%programname%/%hostname%.log'
ruleset(name="inner_test"){
  if $programname == "apache" then {
    ?multi_test
    stop
  }
  if $programname == "postfix" then {
    ?multi_test
    stop
  }
}
if $hostname = "host1" then call inner_test
if $hostname = "host2" then call inner_test
/var/log/other-logs
```

This change defines a template to be used for the file name to be written to (the Dynafile capability mentioned earlier), then defines a ruleset to use—similar to a subroutine that checks that the program name is one of the known ones. If so, it writes the log out to a file whose name is defined by the template and stops processing the log message. Then, finally, you go through a list of hosts. If the host is known, you call the ruleset subroutine to check the program name. If either the hostname or program name is not in your list of known entities, then the tests will not match and the stop action will never be reached, resulting in the log entry being put into the /var/log/other-logs file.

This specific case can be simplified further by using the rsyslog array match capability. This approach requires that the entries to be matched in the array be sorted, but it can further reduce the configuration size and speed up the processing.

```
$template multi_test='/var/log/%programname%/%hostname%.log'
ruleset(name="inner_test"){
  if $programname == ["apache", "postfix"] then {
    ?multi_test
    stop
  }
}
if $hostname = ["host1","host2"] then call inner_test
/var/log/other-logs
```

When troubleshooting performance problems with rsyslog, usually the biggest problem is finding where the bottleneck is; once the bottleneck is found, it's usually not that complicated to remove it. Something can always be done. In extreme conditions, it's even possible to use a custom module to do extensive string processing that is expensive to do in the config language. You can always ask for help on the rsyslog-users mailing list at rsyslog@lists.adiscon.com; the volunteers there are always interested in new problems to solve.

Reference

[1] Links to previous articles by David Lang about logging: https://www.usenix.org/login/david-lang-series.