LOGSTASHER - VOL1

METRICS, METRICS, METRICS

BY LOGSTASHER TEAM

For B and C -

DEFINITELY NOT A FOREWARD BECAUSE NOBODY READS FOREWARDS

I'm not sure why I chose metrics for a starting point in Volume 1. It's probably because I heard that Peter Drucker quote at one point in time: "You can't improve what you don't measure". Another reason is that I dedicated a significant amount of my own personal time to messing around with metrics. Part of that was my own curiosity. A larger part was feeling the burn after putting out a fire at $dayjob. For those with experience in the industry that last line will sound all too familiar. For those of you new to the game, there will be fires. I hope they are few and I hope you persevere and prevail and hit that eureka moment when you solve the problem and save your ass.

There should not have to be a "eureka!" moment though. If the systems are designed well enough and the right metrics are collected, and the right people can interpret those metrics correctly, the cause of the fire should be clear as day. That should be the aim anyway. Maybe that's how things go down at the FAANGs of the world. For the rest of us that dwell in "too few people, too little time", that goal may seem further away.

If the reader is familiar with Amazon's "Well Architected Framework"[1], they will know that metrics align with Performance Efficiency (PE) pillar in the framework and that Monitoring is listed as a Best Practice under the PE pillar. Hopefully we can all agree that metrics are important. Having finally agreed, we are left with a question: "WHAT DO?!". What do we measure? How do we measure it? Do we need more tooling ? Where do the metrics go? Do I have to look at spreadsheets? "If I wanted to do math I would have been an accountant".

Here is where we get to the good news / bad news scenario. Are your organization's compute resources currently residing in one of the big cloud providers? If yes: good news, many of the baseline metrics (from a Virtual Machine / host perspective) are much easier to implement/obtain/monitor/alert on. If no: bad news. Your org may use something like Solarwinds to do baseline monitoring of hosts, or may not be doing any such monitoring at all and you may / may not have access to this data (sigh… le combat est réel).

For what it's worth, I consider the following to be baseline metrics in that they are 1) offered by most if not all cloud providers / monitoring software vendors, 2) are relevant at the host / Operating System level, and 3) are relevant regardless of whatever applications we may be tasked with maintaining / owning / monitoring.

- CPU Utilization
- Memory Utilization
- Network Utilization
- Disk Utilization

Cloud providers and monitoring software vendors will have some means of relaying that baseline information and granularity will vary. For example, AWS "*Disk**" metrics include things like "*DiskReadOps*", "*DiskWriteOps*", "*DiskReadBytes*", and "*DiskWriteBytes*".[2] On the other hand, Azure metrics

[1] AWS WELL ARCHITECTED FRAMEWORK: https://aws.amazon.com/architecture/well-architected/?wa-lens-whitepapers.sort-by=item.additionalFields.sortDate&wa-lens-whitepapers.sort-order=desc

[2] AWS EC2 Metrics:https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/viewing_metrics_with_cloudwatch.html

include things like “*Available Memory Bytes*”, “*Data Disk IOPS consumed percentage*”, and “*Data Disk Target Bandwidth*”.[3]

That’s good and well, but those metrics may not always give you the insight you need. I don’t think any lone metric can. Consider the following (since this is *Logstasher* after all): how relevant are “DiskRead/WriteOps” to your Logstash node in the following scenario?

- The node only has 1 TCP input
- The node only has 1 Elasticsearch output.
- The node does not use persistent queues.
- The node only writes normal application logs to /var/log/logstash
- These logs are purged weekly.
- The node has 500G of available disk (for the sake of the example, consider the node to not have any funky partition layouts, everything is mounted at /).
- Disk utilization has only ever peaked at 30G

The answer is “not very”. This brings us to application metrics. Logstash is a Java application and thus runs on the Java Virtual Machine (JVM). As such it’s possible to have completely fucked your Logstash or JVM settings and never notice it from a baseline (cpu / memory / disk / network) perspective. WOMP WOMP WOMP. Fortunately, with some discipline, practice, and curiosity we can do better to protect ourselves against this.

This issue of *LOGSTASHER* doesn’t go too deep on the JVM front but there is a section where we try to replicate one of Elastic’s examples of an unhealthy Logstash node / JVM from their docs.[4] Some of the examples have mainly application relevance (particularly the syslog section in which everything is generated on 1 —albeit fairly large— server). “But, but AKCHUALLY in a REAL network…”. Spare us. We know. We also don’t have massive networks to play with. But we do stumble upon servers from time to time. We also get a little cloudy (with a lean towards AWS; sorry Azure shills) so we’ve shared some CloudFormation templates.

The logical progression of each volume will start with small building blocks. From there we will build in complexity. We have found that this progression reflects reality in the lab as well as in production. We hope you find this issue useful / enjoyable / interesting. If that is the case, we humbly ask that you leave us review wherever you happened to pick up this little diddy. This helps us understand what sucked and what was good. With this understanding we can improve Logstasher and keep doing what we’re doing if our readers have found it worthwhile.

Thank you for your time,

- root@logstasher

P.S. My grades in discrete mathematics and stats were shit (but passing), so I’m not even going to pretend to get all nerdy with the numbers. Do what thou wilt.

[3] AZURE VM METRICS: https://docs.microsoft.com/en-us/azure/azure-monitor/essentials/metrics-supported#microsoftcomputevirtualmachines

[4] TUNING AND PROFILING LOGSTASH PERFORMANCE: https://www.elastic.co/guide/en/logstash/current/tuning-logstash.html

1. SYSLOG

Syslog is a good starting point. Every network device I've encountered in an enterprise can ship syslog.[5] In linux land, it can also be pretty straightforward to configure a host to send syslog to a remote target. Unfortunately most syslog syntax is REALLY bad so if you want well-structured data you'll likely spend a lot of time running GROKs or regex against that syslog. Perhaps if you're lucky, you'll be able to offload some of that parsing to say a Filebeat module (for example, the Filebeat Cisco modules just create an ingest pipeline at the Elasticsearch cluster, and offload all that grokking to the Elasticsearch nodes).

At the end of the day, pattern matching to extract relevant information is *Business As Usual*. There are only two things certain in life… death and using regex against syslog messages. But I digress.

SYSLOG LAB - 001

First, some host meta. Host details for this lab are as follows.

- Operating System

```
$ uname -a
Linux localbox 5.4.0-121-generic #137-Ubuntu SMP Wed Jun 15 13:33:07 UTC 2022
```

- CPU

```
$ lscpu
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   46 bits physical, 48 bits virtual
CPU(s):                          48
On-line CPU(s) list:             0-47
Thread(s) per core:              2
Core(s) per socket:              12
Socket(s):                       2
NUMA node(s):                    2
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           63
Model name:                      Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz
Stepping:                        2

                                  - - - - SNIP - - - -
```

[5] Parsing those logs suck. But any data is better than no data. Pretty data is better than "<136>Oct 1 1988 13:37:37 mothership001 ERROR birthd[13] baby inbound…."

MEMORY

```
$ free -h
              total        used        free      shared  buff/cache   available
Mem:          125Gi        38Gi        84Gi        14Mi       3.3Gi        86Gi
Swap:         8.0Gi          0B       8.0Gi
```

DISK

```
$ lsblk
NAME                          MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
loop0                           7:0    0  55.5M  1 loop /snap/core18/2409
loop1                           7:1    0  55.5M  1 loop /snap/core18/2344
loop2                           7:2    0  99.5M  1 loop /snap/go/9848
loop3                           7:3    0  61.9M  1 loop /snap/core20/1518
loop4                           7:4    0  44.7M  1 loop /snap/snapd/15534
loop5                           7:5    0    47M  1 loop /snap/snapd/16010
loop6                           7:6    0  61.9M  1 loop /snap/core20/1434
loop7                           7:7    0  67.2M  1 loop /snap/lxd/21835
loop8                           7:8    0  67.8M  1 loop /snap/lxd/22753
loop9                           7:9    0  99.4M  1 loop /snap/go/9760
sda                             8:0    0   931G  0 disk
├─sda1                          8:1    0     1M  0 part
├─sda2                          8:2    0     1G  0 part /boot
└─sda3                          8:3    0   930G  0 part
  └─ubuntu--vg--1-ubuntu--lv 253:5    0   200G  0 lvm
sdb                             8:16   0   931G  0 disk
└─sdb1                          8:17   0   931G  0 part
  └─VolGroup01-decometa       253:4    0   931G  0 lvm
sdc                             8:32   0  16.4T  0 disk /
sdd                             8:48   0  10.9T  0 disk
└─sdd1                          8:49   0  10.9T  0 part
  ├─VolGroup03-concroot       253:1    0    30G  0 lvm
  ├─VolGroup03-concsess       253:2    0     1T  0 lvm
  └─VolGroup03-concmeta       253:3    0   9.9T  0 lvm
sde                             8:64   0 744.6G  0 disk
└─sde1                          8:65   0 744.6G  0 part
  └─VolGroup04-concinde       253:0    0 744.6G  0 lvm
```

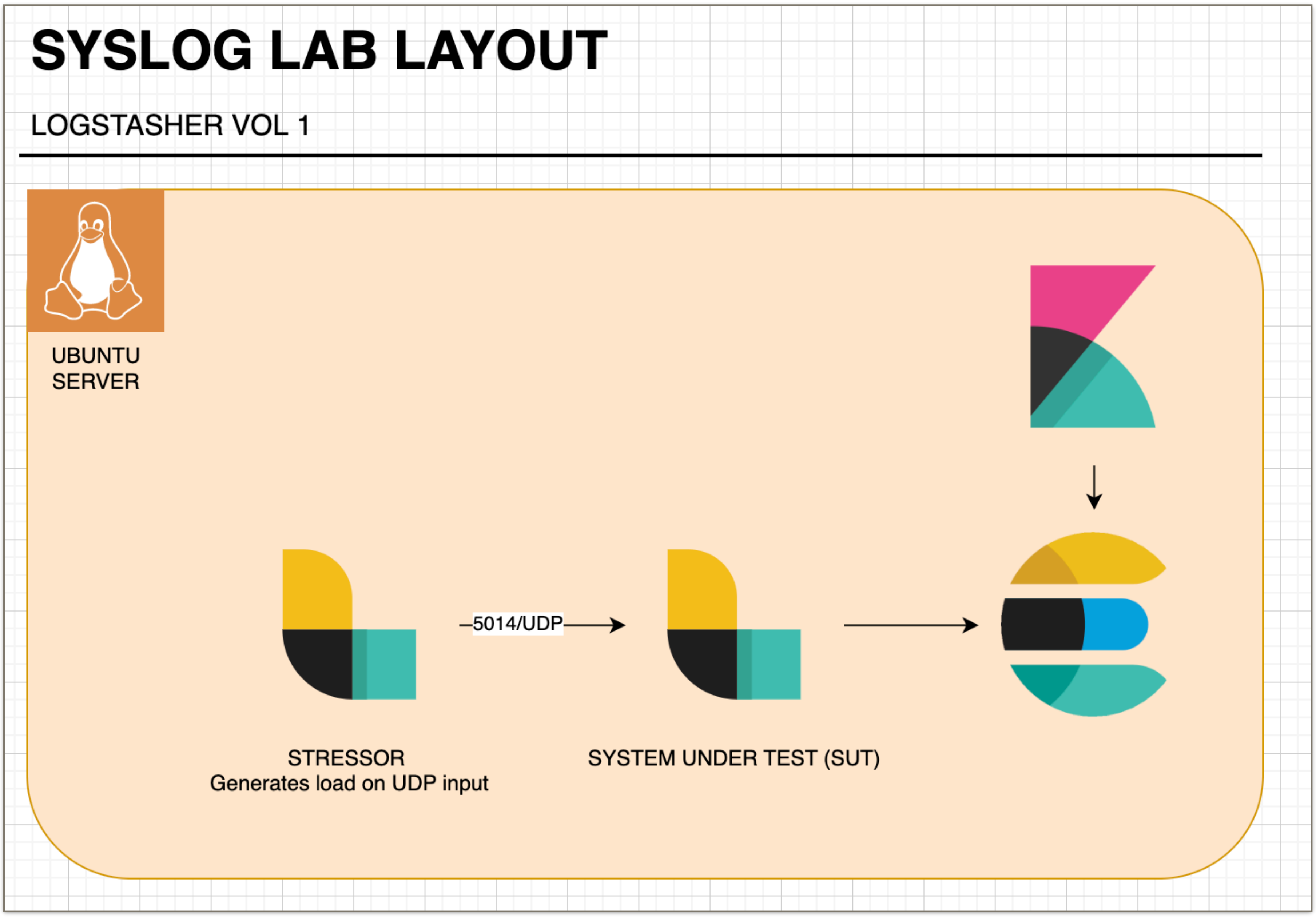
"THERE ARE ONLY TWO THINGS CERTAIN IN LIFE... DEATH AND USING REGEX AGAINST SYSLOG MESSAGES."

As you can see, we have plenty to work with. Now for the logstash meta.

1. **Logstash version:** 7.17.4
2. **Logstash install method:** deb
3. **Non default JVM settings:** heap at 2g

```
$ grep -E '^-Xm(s|x).g' /etc/logstash/jvm.options
-Xms2g
-Xmx2g
```

4. **Logstash pipeline size:** 125 (default)
5. **Logstash pipeline workers:** 2



IMG001 - A second logstash instance (STRESSOR) is used to generate traffic on the System Under Test (SUT) and increment the load until we see performance impact.

The base config for the SUT is listed on the next page. We have 1 UDP input on port 5014 (this is necessary if your main logstash process is running as a non root user) that has some metadata added to it. The first uses the builtin metrics filter to to track event rates and tag those metrics so we can segregate them in the output logic.[6] The second filter just tries to run a few groks against the message. The output dumps metric events to /var/log/syslog.metrics.json and sends them to an event rates index, and ships the regular processed syslog events to a syslog index. [7] We can fire this up and let it run while get the stressor setup.[8]

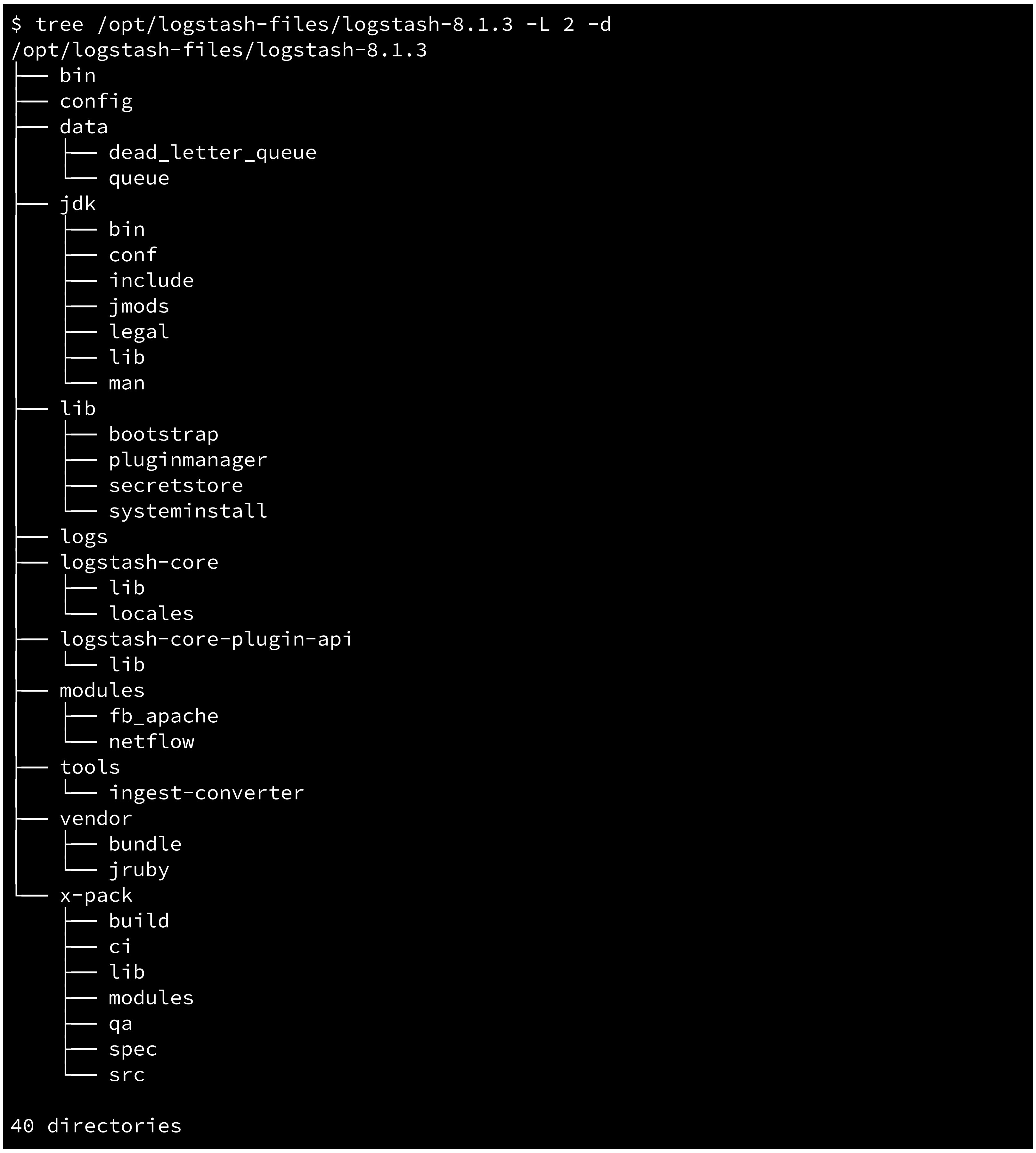
[6] METRICS FILTER PLUGIN: https://www.elastic.co/guide/en/logstash/current/plugins-filters-metrics.html

[7] We don't cover index configuration here. It's assumed the reader already knows how to do this.

[8] NOTE: the default number of worker threads for UDP input is 2: https://www.elastic.co/guide/en/logstash/current/plugins-inputs-udp.html

```
$ cat /etc/logstash/conf.syslog/syslog.cong
#### syslog base config
input {
    udp {
        port => 5014
        add_field => {'[logstash][input]' => 'udp5014'}
        add_field => {'[@metadata][index]' => 'syslog-rollover'}
        add_field => {'[logstash][type]' => 'syslog'}
    }
}
#### metrics filter
filter {
    #### logstash outputs these generated messages as json by default so convert
    json { source => ‘message' }
    mutate { remove_field => 'host'}
    if [logstash][input] == 'udp5014' {
        metrics {
            meter => 'events'
            add_tag => 'metric-syslog'
        }
    }
}
filter {
    if [logstash][type] == 'syslog' {
        grok {
            match => {'message' => '^<%{NUMBER:[syslog][priority]}>%
{GREEDYDATA:therest_001}'}
            tag_on_failure => 'gpf_001'
        }
        if [therest_001] {
            grok {
                match => {'therest_001' => '^%{WORD:[syslog_month]}\s%
{NUMBER:syslog_daynum}\s+%{TIME:syslog_time}\s+%{WORD:syslog_host}\s+%
{GREEDYDATA:therest_002}'}
                tag_on_failure => 'gpf_002'
            }
        }
    }
}
output {
    if 'metric-syslog' in [tags] {
        file {
          path => '/var/log/logstash/syslog.metrics.json'
        }
        elasticsearch {
            hosts => 'localhost:9200'
            index => 'eventrates-rollover'
            user => 'elastic'
            password => 'PASSHERE'
            http_compression => true
        }
    }
    if [@metadata][index] {
        elasticsearch {
            hosts => 'localhost:9200'
            index => '%{[@metadata][index]}'
            user => 'elastic'
            password => 'PASSHERE'
            http_compression => true
        }
    }
}
```

For the stressor Logstash instance, we've download a tarball of Logstash 8.1.3 and extracted that in /



```
$ tree /opt/logstash-files/logstash-8.1.3 -L 2 -d
/opt/logstash-files/logstash-8.1.3
├── bin
├── config
├── data
│   ├── dead_letter_queue
│   └── queue
├── jdk
│   ├── bin
│   ├── conf
│   ├── include
│   ├── jmods
│   ├── legal
│   ├── lib
│   └── man
├── lib
│   ├── bootstrap
│   ├── pluginmanager
│   ├── secretstore
│   └── systeminstall
├── logs
├── logstash-core
│   ├── lib
│   └── locales
├── logstash-core-plugin-api
│   └── lib
├── modules
│   ├── fb_apache
│   └── netflow
├── tools
│   └── ingest-converter
├── vendor
│   ├── bundle
│   └── jruby
└── x-pack
    ├── build
    ├── ci
    ├── lib
    ├── modules
    ├── qa
    ├── spec
    └── src

40 directories
```

opt/logstas-files resulting in the following tree structure.

Nothing fancy here really. What we need is a simple way to run this second logstash instance, with variable settings that will generate output to localhost:5014/UDP (where the SUT is listening for our "syslog" events). We can accomplish this by making a bash script in /opt/logstash-files/logstash-8.1.3/

bin that'll run logstash from the command line with our desired settings. Version one of this is on the next page. Apologies in advance for the one-liner config, I know it looks ugly, but I've never been a fan of

```
#!/bin/bash
GEN_THREADS=$1
BATCH_SIZE=$2
WORKERS=$3

FAKEMESSAGE="<161>Jul  2 13:37:00 localbox logstasher[1337]: LOGSTASHER VOLUME ONE
- METRICS,METRICS,METRICS"


/opt/logstash-files/logstash-8.1.3/bin/logstash -e "input { generator { message =>
'$FAKEMESSAGE' threads => $GEN_THREADS }} output {  udp { host => 'localhost' port
=> '5014' } }" --pipeline.batch.size=$BATCH_SIZE --pipeline.workers=$WORKERS --
path.data=/tmp/data
```

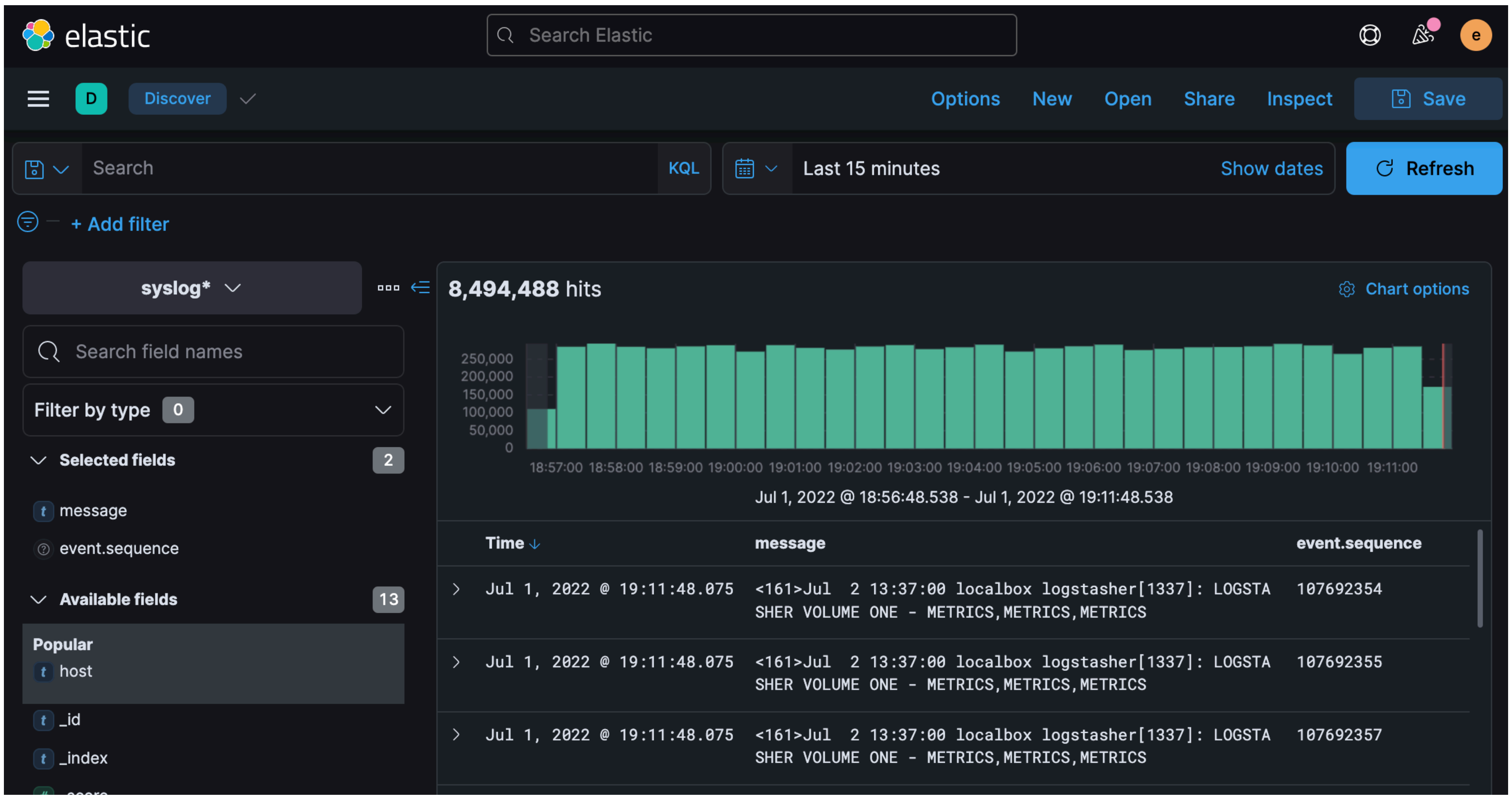
the whole escaped line breaks thing in bash scripts, so here we are.

This script allows us to screw with the throughput settings a little bit. *GEN_THREADS* is relevant to the generator input only, while *BATCH_SIZE* and *WORKERS* you'll recognize as being relevant to the output. We set *path.data* to /tmp/data so the stressor doesn't try to use any default paths and have a conflict with the existing data path of the logstash process that is already running. In it's current state, the process will continue to generate events until we send SIGINT to the process (CTRL-C). For now we

```
$ ./stressd.sh 1 125 2
                              - - - - SNIP    - - - -
[2022-07-02T01:37:49,592][INFO ][logstash.javapipeline    ][main] Starting
pipeline {:pipeline_id=>"main", "pipeline.workers"=>2, "pipeline.batch.size"=>125,
"pipeline.batch.delay"=>50, "pipeline.max_inflight"=>250,
"pipeline.sources"=>["config string"], :thread=>"#<Thread:0x54c7f73 run>"}
[2022-07-02T01:37:50,217][INFO ][logstash.javapipeline    ][main] Pipeline Java
execution initialization time {"seconds"=>0.62}
[2022-07-02T01:37:50,238][INFO ][logstash.javapipeline    ][main] Pipeline started
{"pipeline.id"=>"main"}
[2022-07-02T01:37:50,292][INFO ][logstash.agent           ] Pipelines running
{:count=>1, :running_pipelines=>[:main], :non_running_pipelines=>[]}
```

can start it with 1 generator thread, a batch size of 125 (default) and 2 workers.

After letting this setup run for a while we can check how many events get ingested into elasticsaerch over a 15 minute window by viewing the syslog index in Kibana. In the screenshot on the next page you'll see that the hit count is about 8.5 Million. With a little math we can get our 1 minute event rate: (8,500,000 / 15 / 60 = 9444.44 events per minute / 60 = 157.4 events per second). Not bad… or is it? Is that abnormally low? Is that high? Don't know…. Need more data.



```
Time                        message                                                    event.sequence
Jul 1, 2022 @ 19:11:48.075  <161>Jul  2 13:37:00 localbox logstasher[1337]: LOGSTA     107692354
                            SHER VOLUME ONE - METRICS,METRICS,METRICS
Jul 1, 2022 @ 19:11:48.075  <161>Jul  2 13:37:00 localbox logstasher[1337]: LOGSTA     107692355
                            SHER VOLUME ONE - METRICS,METRICS,METRICS
Jul 1, 2022 @ 19:11:48.075  <161>Jul  2 13:37:00 localbox logstasher[1337]: LOGSTA     107692357
                            SHER VOLUME ONE - METRICS,METRICS,METRICS
```

IMG002 - With the base syslog config and base stressor, we're seeing about 8.5 Million events in a 15 minute window.

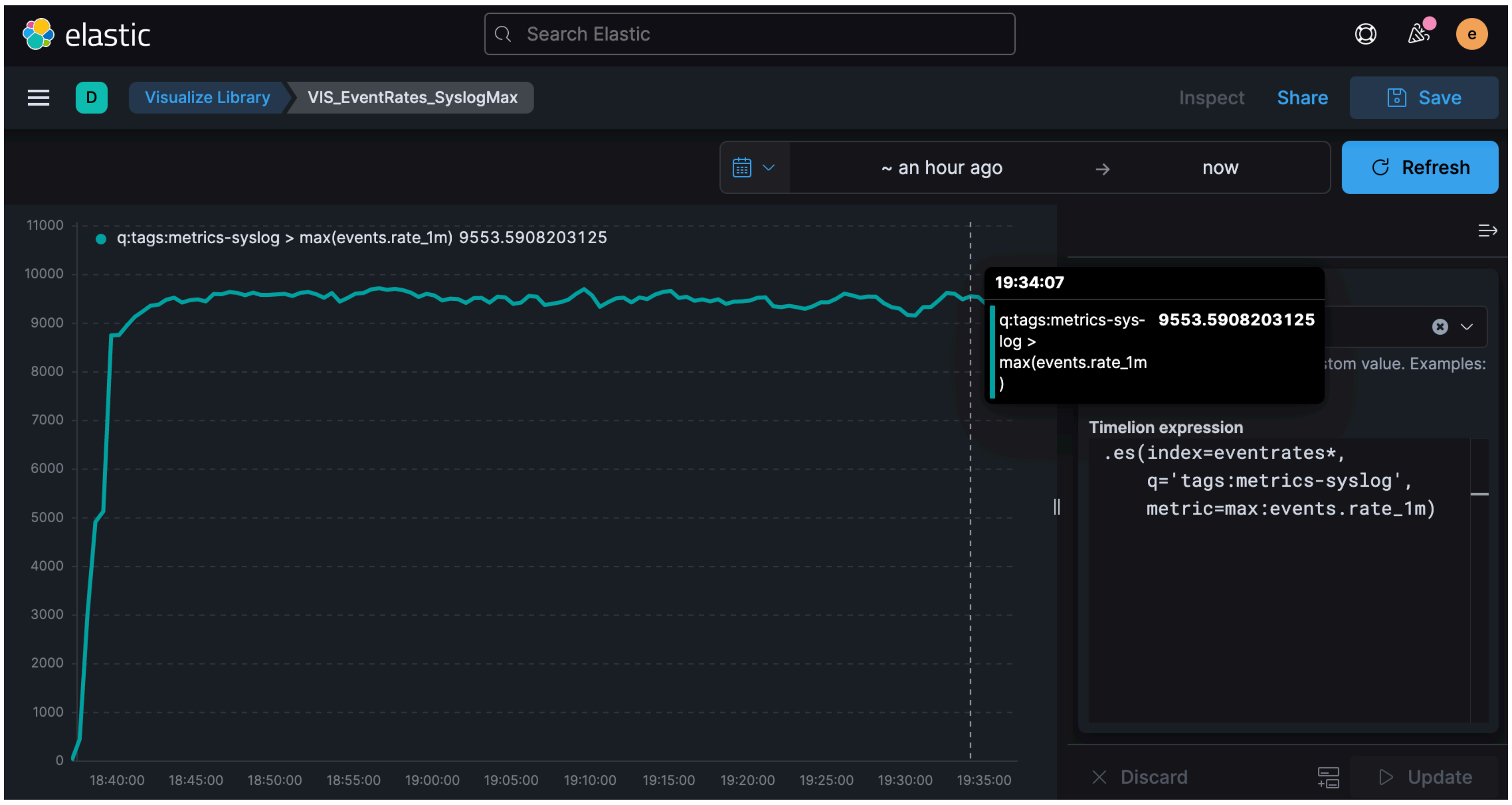
Recall that the config is also dumping event rate metrics to syslog.metrics.json. We can check this file

```
$ tail -n 20 /var/log/logstash/syslog.metrics.json | jq .events.rate_1m | sort -rn
9336.487809147882
9330.968342568867
9323.966401722813
9298.897997136572
9273.906749372241
9261.868007578634
9239.000771878627
9237.72091719613
9234.704919180995
9230.136011461975
9229.35448946115
9193.648130609885
9186.280887468307
9159.608373713842
9151.16202781002
9123.019196637968
9077.222430805858
9053.51695542433
9049.716496527128
9027.75137823632
```

and see if those numbers reflect our loose math from earlier.

As it turns out, the loose math is retty accurate. The last 20 metrics (sampled every 5 seconds by default) are all about 9K and some change which means our loose math wasn't far off. But our base SUT config was also shipping events to the cluster. What do these event rates look like over time? Below is a timelion expression and visualization used to see *events.rate_1m* over time.

```
.es(index=eventrates*,q='tags:metrics-syslog',metric=max:events.rate_1m)
```



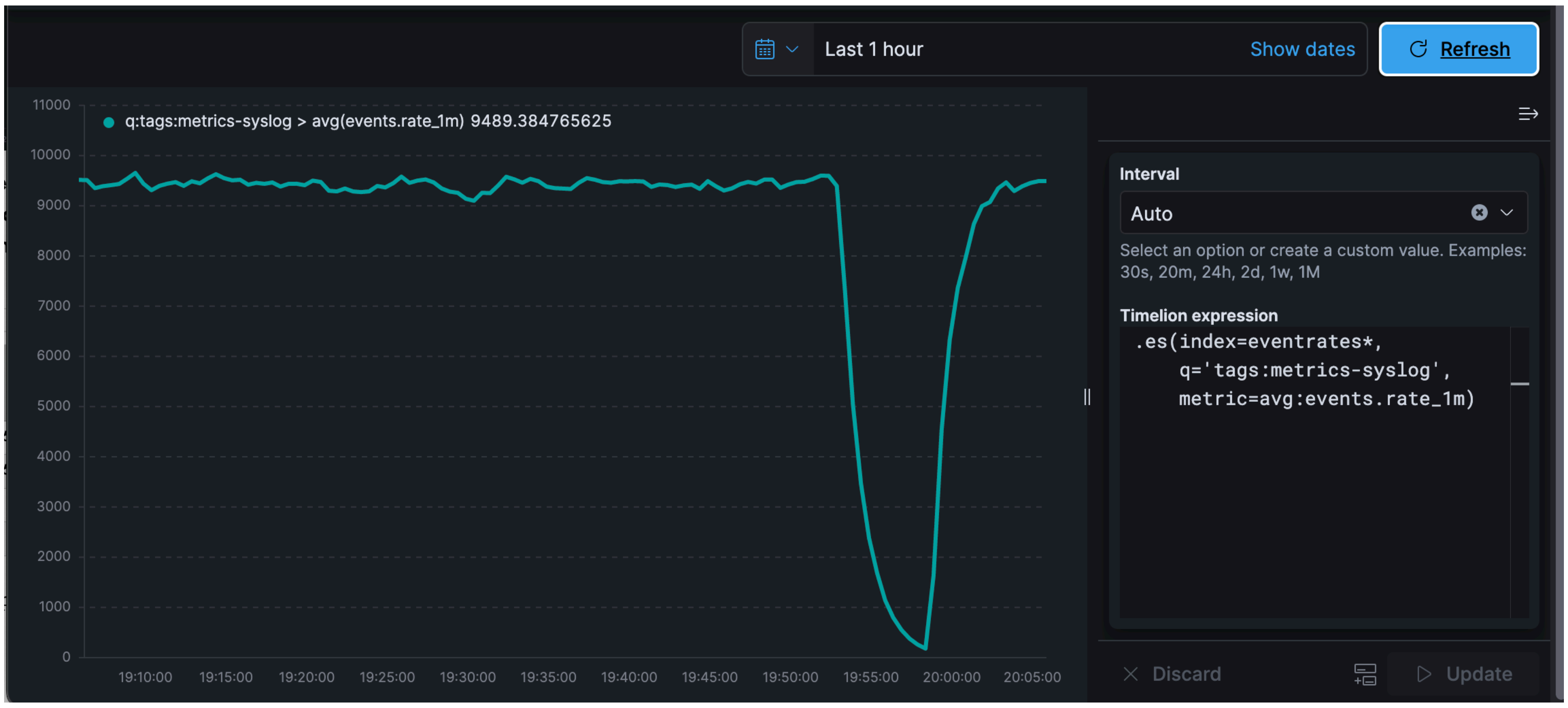
```
.es(index=eventrates*,
    q='tags:metrics-syslog',
    metric=max:events.rate_1m)
```

IMG003 - In this timelion visualization we see our event rate spike rapidly (when we first started the stressor script) and then level off at around 9K+ for the metric *events.rate_1m*. This indicates that our throughput max has been reached with the current settings.

Let's record our current testing results thus far.

	STRESS:BATCH	STRESS:WRKRS	STRESS:INPUT THREADS	SUT:BATCH	SUT:WRKRS	SUT:INPUT THREADS	SUT:VAR?	RATE 1m (nearest 100)
1	125	2	1	125	2	2		9400

CAN WE GO HIGHER!??? Let's find out…. What happens if we bump the number of input threads on the stressor to 2?!!! Womp the graph looks the same. As you'll see on the next page..



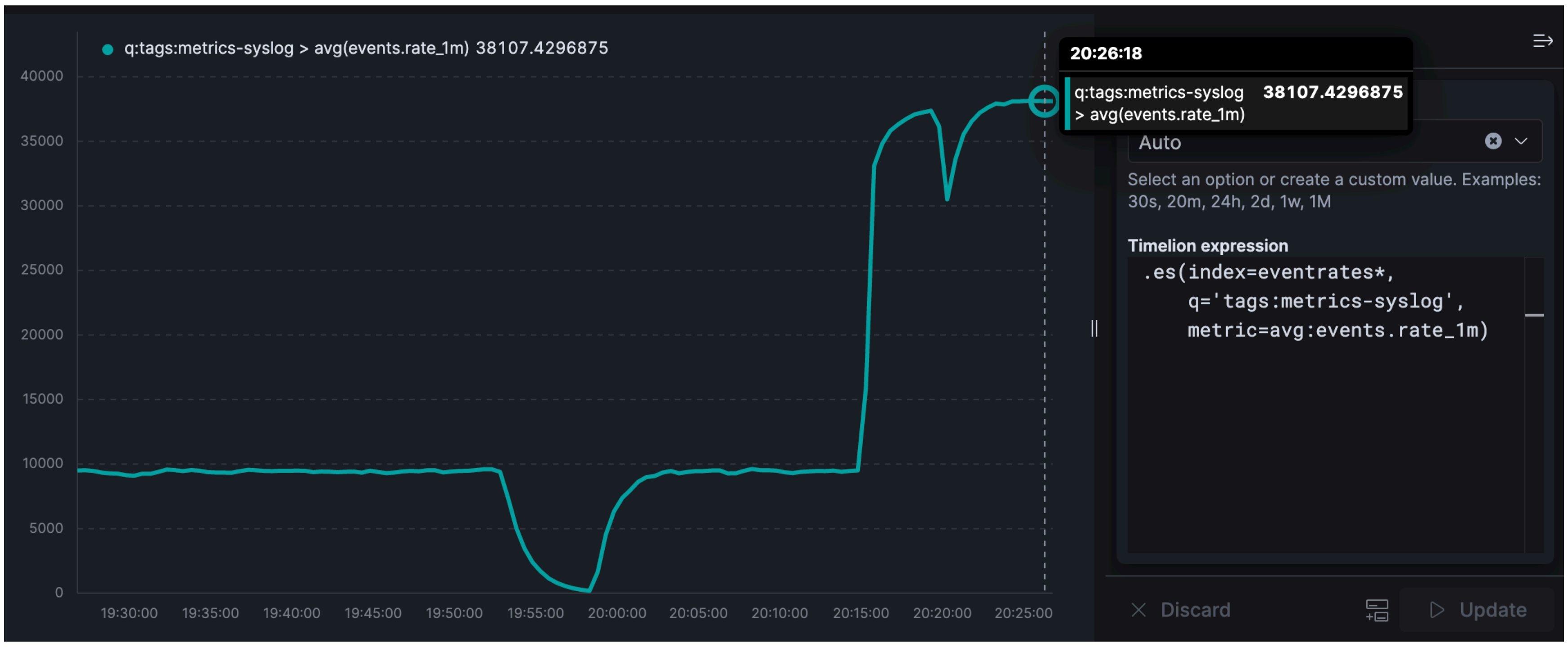
```
.es(index=eventrates*,
    q='tags:metrics-syslog',
    metric=avg:events.rate_1m)
```

With a slight adjustment (bumping the number of stressor input threads to 2) the event rate of the SUT does not deviate significantly from the previous levels.

We can record a new entry on our table.

	STRESS:BATCH	STRESS:WRKRS	STRESS:INPUT THREADS	SUT:BATCH	SUT:WRKRS	SUT:INPUT THREADS	SUT:VAR?	RATE 1m (nearest 100)
1	125	2	1	125	2	2		9400
2	125	2	2	125	2	2		9400

YAAAAAAAAAAWWWN…and that's how metrics go dear readers. Let's cheat a little bit… What happens if we bump the workers on our SUT to 8? Alas… we begin to get somewhere. Now the event rate has climbed closer to 38,100 minus the small blip in the graph where I realized I had left the stressor at 2 input threads while changing the SUT workers to 8 (a big no no). This is a big no no because we should only change one variable at a time when performing the tests. Otherwise we invite chaos.


q:tags:metrics-syslog > avg(events.rate_1m) 38107.4296875
20:26:18
q:tags:metrics-syslog > avg(events.rate_1m) 38107.4296875
Auto
Select an option or create a custom value. Examples: 30s, 20m, 24h, 2d, 1w, 1M
Timelion expression
Discard
Update

```
.es(index=eventrates*,
    q='tags:metrics-syslog',
    metric=avg:events.rate_1m)
```

IMG004 - With the SUT workers at 8, the event rate has climbed to 38,100. We increased throughput by a factor > 4.

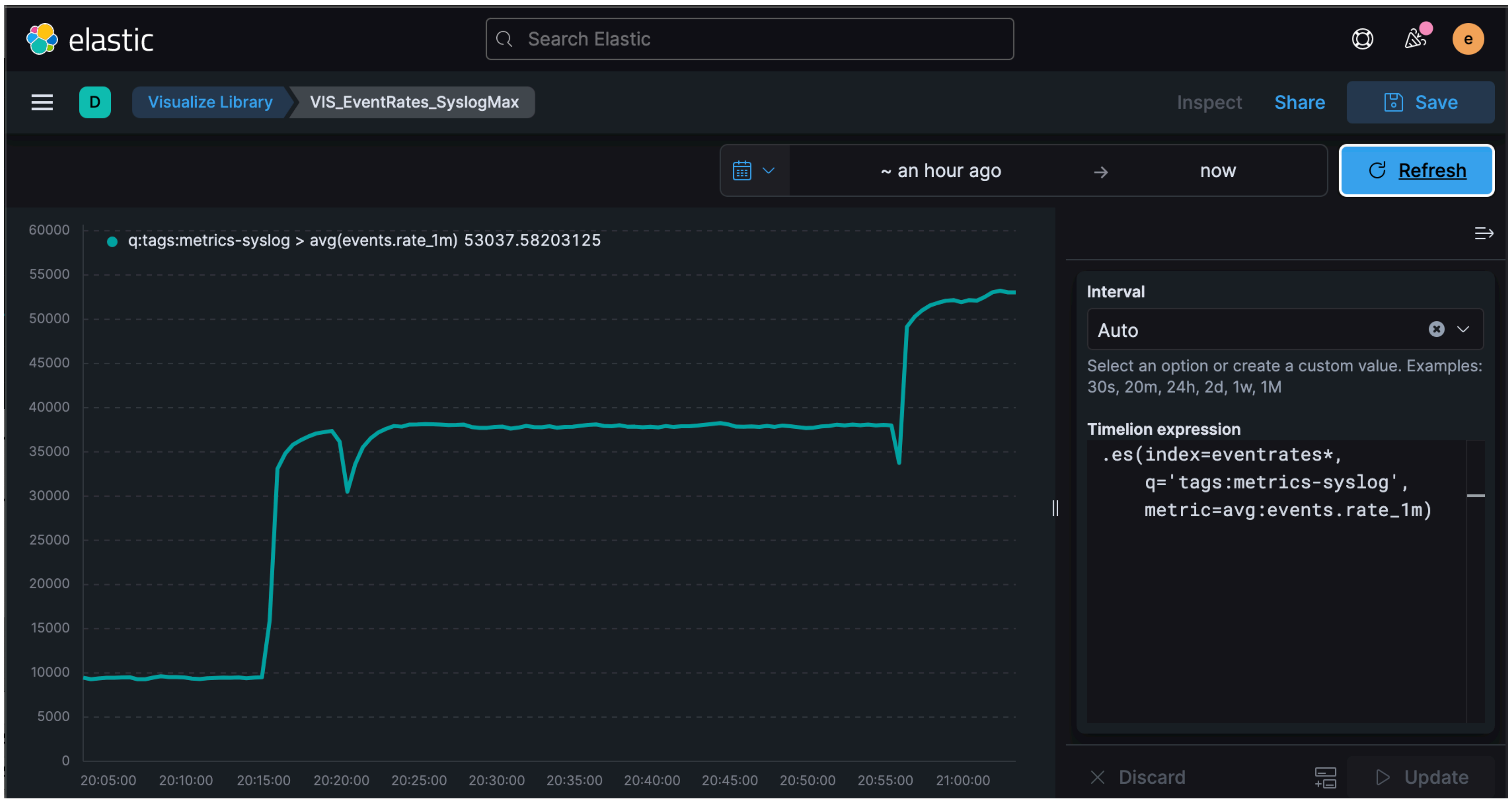
Sweet we got higher. *Can you take me higher, dear reader?*[9] More will be revealed… but first. A riveting new entry for our data table.

	STRESS:BATCH	STRESS:WRKRS	STRESS:INPUT THREADS	SUT:BATCH	SUT:WRKRS	SUT:INPUT THREADS	SUT:VAR?	RATE 1m (nearest 100)
BASE-1	**125**	**2**	**1**	**125**	**2**	**2**		**9400**
2	125	2	2	125	2	2		9400
3	125	2	1	125	8	2		38100

At this point we can keep increasing the SUT worker count until we see ceiling be reached with the event rate, however we go about it is left as an exercise for the reader. Could be we keep doubling the workers. Could be we tie workers to CPU by percentage (i.e. workers == 50% of CPU == 50% of 48 == 24). Whatever's clever. You can review the graph on the next page which is what we saw when we bumped SUT workers to 48 (the number of cpus).

"CAN YOU TAKE ME HIGHER? TO A PLACE WHERE BLIND MEN SEE?" - CREED

[9] "**Higher**" is a song by American rock band Creed. It was released on August 31, 1999, as the lead single from their second studio album, *Human Clay*. The song became the bands breakthrough hit as it was their first song to reach the top ten on the US *Billboard* Hot 100 where it peaked at number seven in July 2000. It spent a total of 57 weeks upon the survey, the longest stay for any Creed song on the Hot 100. "Higher" also became the band's second chart-topping hit on rock radio as it topped both the Modern Rock and Mainstream Rock charts, for a then-record of 17 weeks



● q:tags:metrics-syslog > avg(events.rate_1m) 53037.58203125

Interval: Auto

Select an option or create a custom value. Examples: 30s, 20m, 24h, 2d, 1w, 1M

Timelion expression

```
.es(index=eventrates*,
    q='tags:metrics-syslog',
    metric=avg:events.rate_1m)
```

IMG005 - To the far left of the graph is the event rate seen (~53,000) with SUT workers set to 48.

SYSLOG LAB - 002

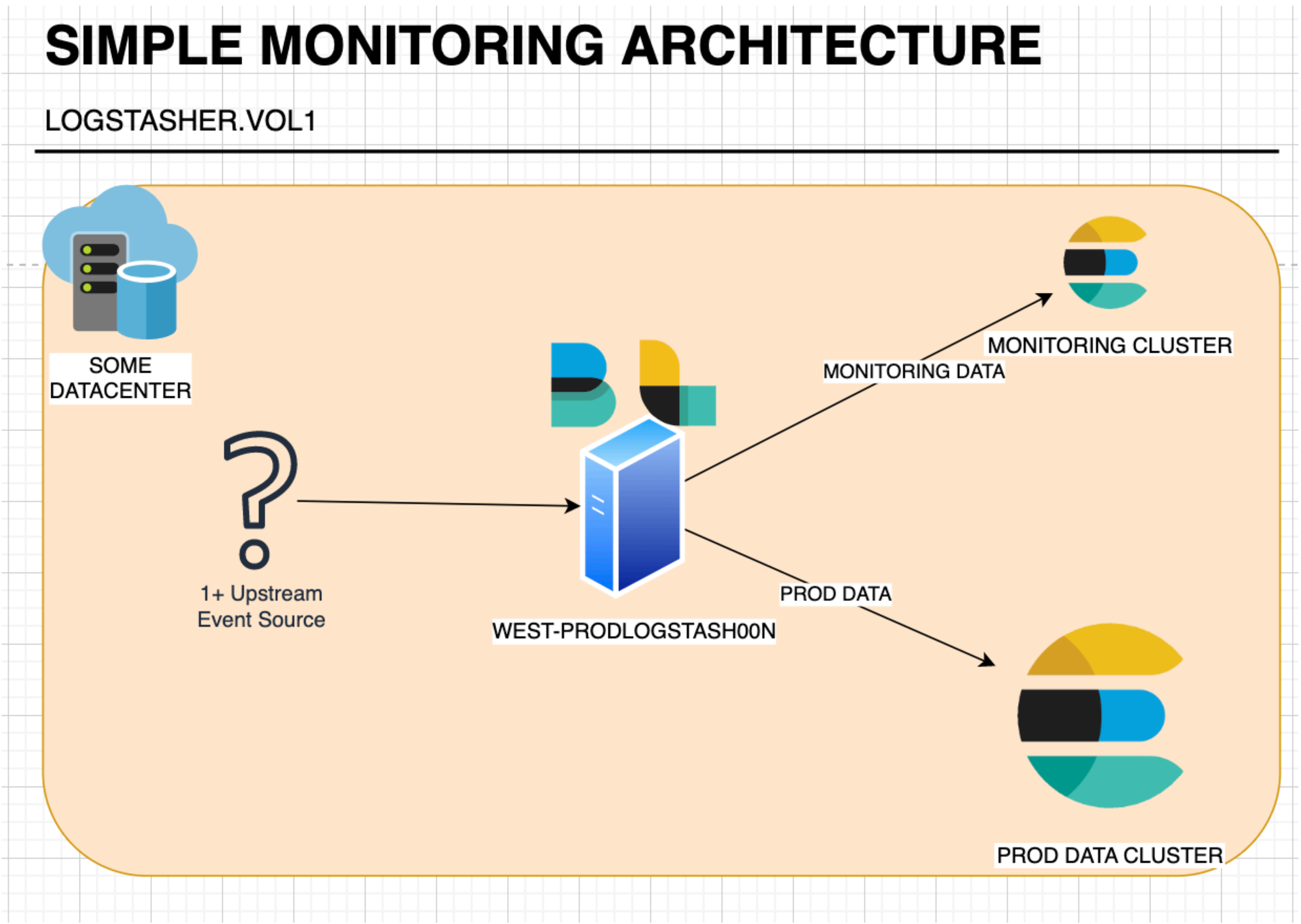
By now we've finagled with the event rate a little bit and seen how we can increase this rate by bumping the workers of our SUT pipeline.But how was the application doing during all this? We don't

```
$ curl localhost:9600/_node/stats 2>/dev/null | jq .jvm
{
  "threads": {
    "count": 169,
    "peak_count": 235
  },
  "mem": {
    "heap_used_percent": 25,
    "heap_committed_in_bytes": 8303607808,
    "heap_max_in_bytes": 8303607808,
    "heap_used_in_bytes": 2110012816,
    "non_heap_used_in_bytes": 245239048,
    "non_heap_committed_in_bytes": 287936512,
    "pools": {
      "young": {
        "committed_in_bytes": 2290614272,
        "max_in_bytes": 2290614272,
        "peak_max_in_bytes": 2290614272,
        "peak_used_in_bytes": 2290614272,
        "used_in_bytes": 1879176360
      },
      "old": {
        "committed_in_bytes": 5726666752,
        "max_in_bytes": 5726666752,
        "peak_max_in_bytes": 5726666752,
        "peak_used_in_bytes": 207727432,
        "used_in_bytes": 207727432
      },
      "survivor": {
        "committed_in_bytes": 286326784,
        "max_in_bytes": 286326784,
        "peak_max_in_bytes": 286326784,
        "peak_used_in_bytes": 103185384,
        "used_in_bytes": 23109024
      }
    }
  },
  "gc": {
    "collectors": {
      "young": {
        "collection_time_in_millis": 75408,
        "collection_count": 10763
      },
      "old": {
        "collection_time_in_millis": 40179,
        "collection_count": 68
      }
    }
  },
  "uptime_in_millis": 17685914
}
```

know because we didn't look. We need some better visibility into the JVM, and we can go about this a number of ways. If we just want to see the metrics at a given point in time, that's easy enough.

I don't know about you, but that doesn't mean shit to me. Save for *jvm.mem.heap_used_percent* which tells me at the time that command ran, only 25% of the heap was used. Great. But what's the trend? If only there were a way to poll this data at specified intervals and then ship it so it can be analyzed and visualized. For this we will use metricbeat's logstash-xpack module to ship the metrics to elasticsearch.

If you've been using logstash for a while you may be familiar with xpack monitoring. TL;DR xpack/X-Pack was originally a "set of closed-source features that extend the Elastic Stack".[10] In this case we're talking specifically about monitoring. X-Pack monitoring data gets dumped into the cluster as a hidden index with a naming convention like *.monitoring-<es|logstash|kibana>-<major_version_number-<?mb>-<YYYY.mm.dd>-NNNNNN*. A typical architecture would look like this.



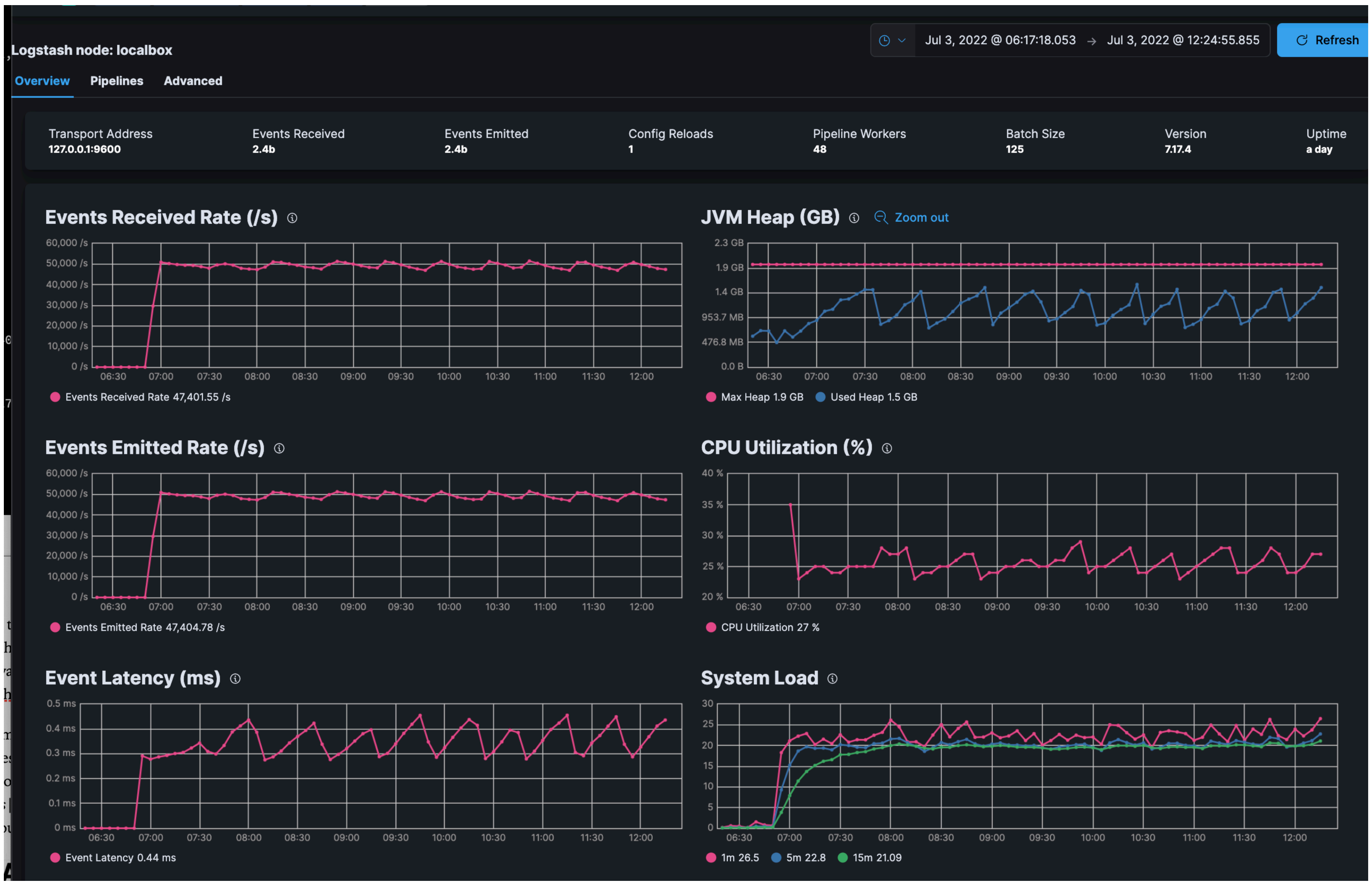
IMG006 - A best practice monitoring architecture will keep stack monitoring data separate from prod data. In the diagram above monitoring cluster is smaller because this is typically the case as monitoring data often requires less resources than your prod data lake.

The above segregates monitoring data from prod data. Why? Well, what happens if our prod data cluster experiences as an outage? RIP, there goes our monitoring data for the rest of the stack. One saying

[10] OPEN X-PACK: https://www.elastic.co/what-is/open-x-pack

that has stuck with me over the years is: “You don’t use the tool to monitor the tool”. Seems sensible enough to me. Duly noted. Moving on.

We won’t cover setup of the metricbeat logstash-xpack module. It’s assumed the reader has some familiarity with beats and how to configure them. If this is not the case, Elastic’s documentation is solid.[11] From the earlier lab you’ll recall that we modified the SUT workers to 48 (1:1 with the number of cpus). Shortly after that I setup the metricbeat module to begin collecting stats.


Logstash node: localbox

Jul 3, 2022 @ 06:17:18.053 → Jul 3, 2022 @ 12:24:55.855

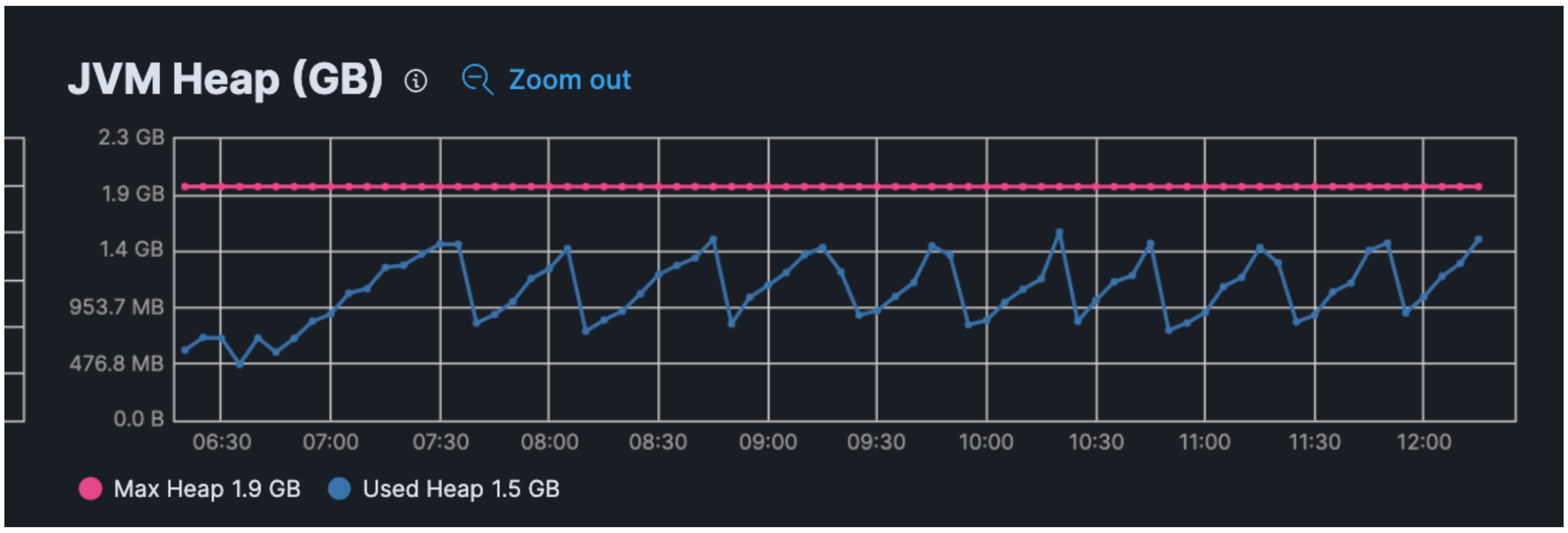
Transport Address	Events Received	Events Emitted	Config Reloads	Pipeline Workers	Batch Size	Version	Uptime
127.0.0.1:9600	2.4b	2.4b	1	48	125	7.17.4	a day

IMG007 - Logstash monitoring stats visualized shortly after increasing the pipeline workers to match the number of CPUs

On the top left graph you’ll note the sudden spike of the event rate after the change to the number of workers was completed. On the top right, you’ll notice the JVM Heap graph. Which we’ll take a closer look at On the next page. The event rate spike hit’s its peak / norm at 07:00 on the graph. Thereafter we see that the *Used Heap* metric (blue line) starts to exhibit a more frequent climb an drop (commonly referred to as a sawtooth pattern; stable, mostly consistent pattern). The drop indicates increased free JVM heap space caused by Garbage Collection. From various engagements with Elastic support Engineers, I can say simply that sawtooth == GOOD/OK. If you want to dive deep on Java Garbage collection, I’m going to copycat Elastic and basically say “RTFM” at Oracle. If you can suffer through that and gain some insight, you’re a better human than I.[12]

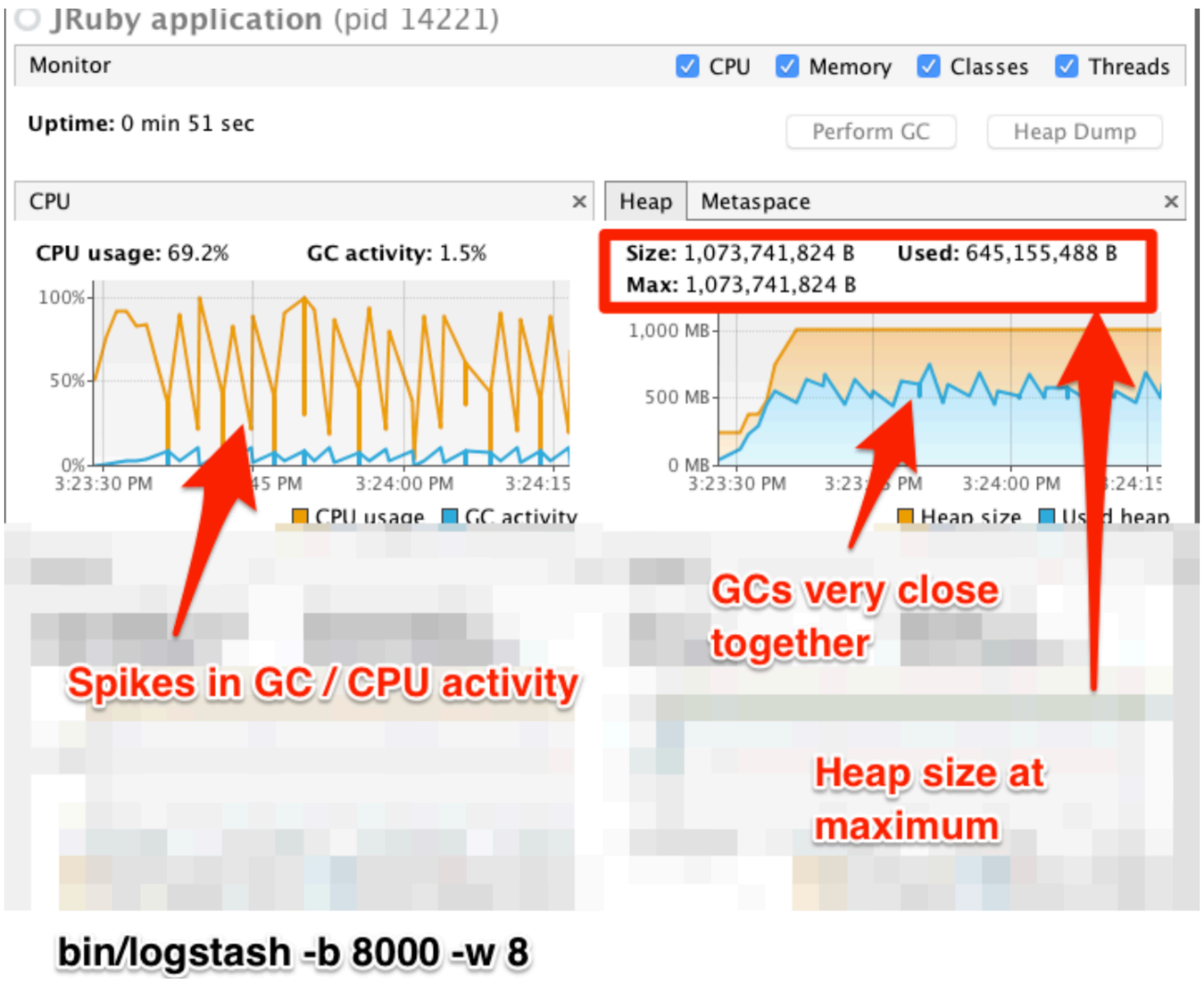
[11] METRICBEAT - LOGSTASH-XPACK MODULE: https://www.elastic.co/guide/en/beats/metricbeat/current/metricbeat-module-logstash.html

[12] ORACLE: https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html



IMG008 - Closer view of the JVM Heap status from IMG007. Note the sawtooth pattern.

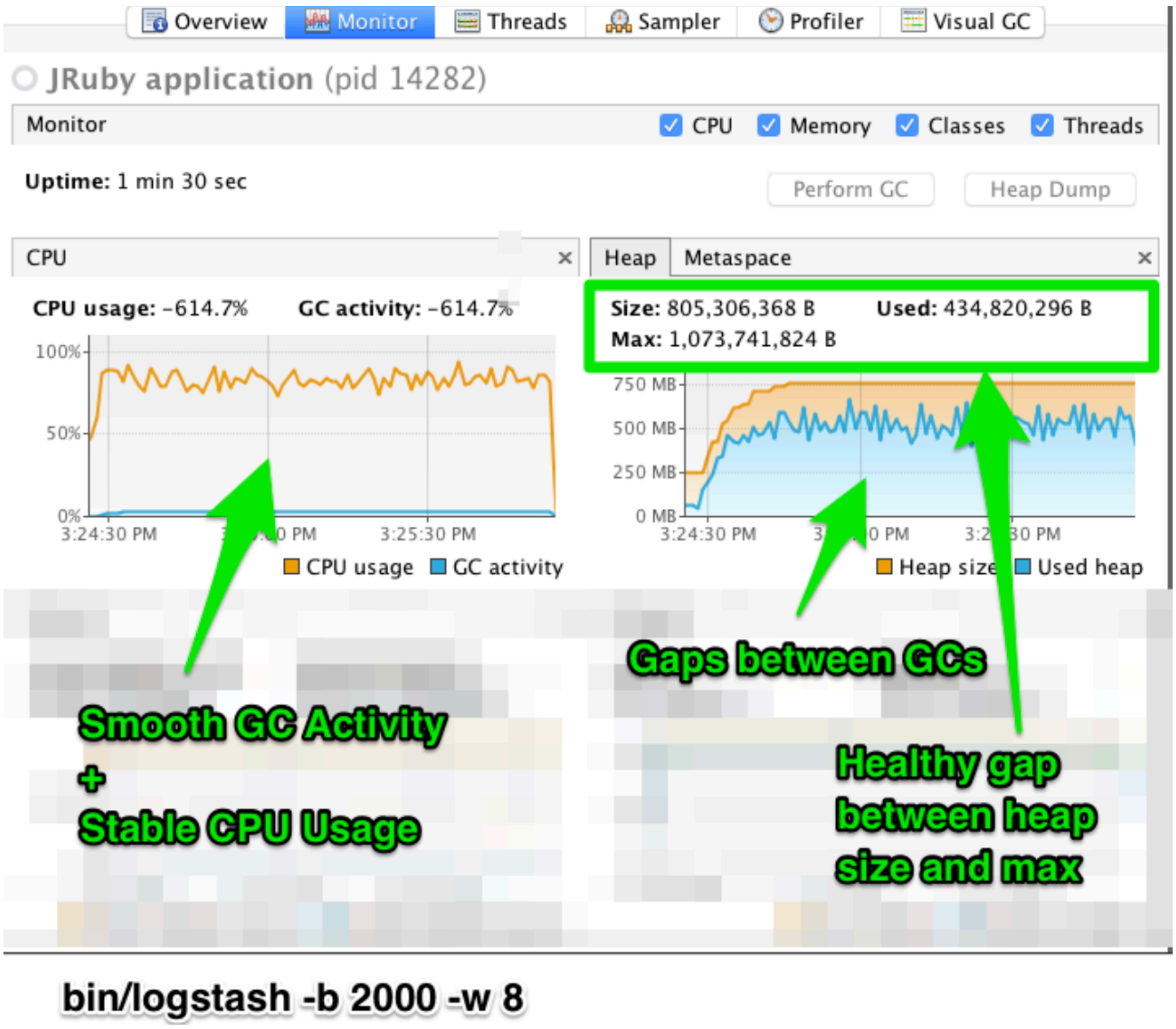
How does this compare with the "Tuning and Profiling Logstash Performance" blurb?[13] Curious what you think of it. To me the two can't really be compared. For starters, the blurb referenced in the footnote uses VisualVM to visualize heap usage. Secondly, the graphs therein are only somewhat different from each other. Refer to the next two samples.



IMG009 - BAD JVM PERFORMANCE (according to the article in [12]).

[13] TUNING AND PROFILING LOGSTASH: https://www.elastic.co/guide/en/logstash/current/tuning-logstash.html

In this example the claim is that “GCs [sic] very close together”, but note per the screenshot they’re only looking at like 3 minutes worth of data? I’m puzzled as to why anyone would want to use such a brief sample window to relay a point about garbage collection. The confusion doesn’t stop there either. Look at what is given for the GOOD case.



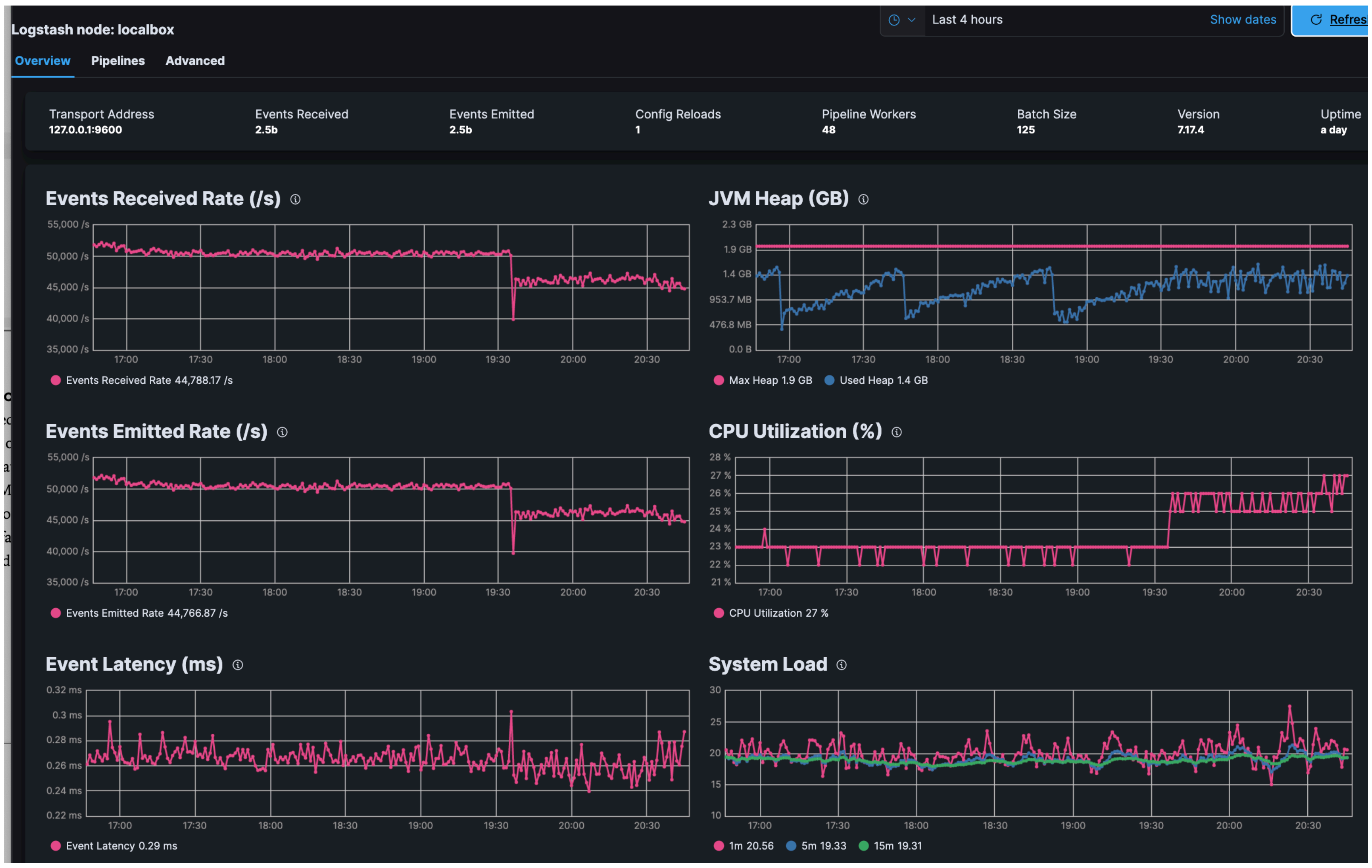
Uptime: 1 min 30 sec

CPU usage: -614.7% GC activity: -614.7%

Size: 805,306,368 B Used: 434,820,296 B
Max: 1,073,741,824 B

IMG010 - GOOD JVM PERFORMANCE (according to the article in [12]).

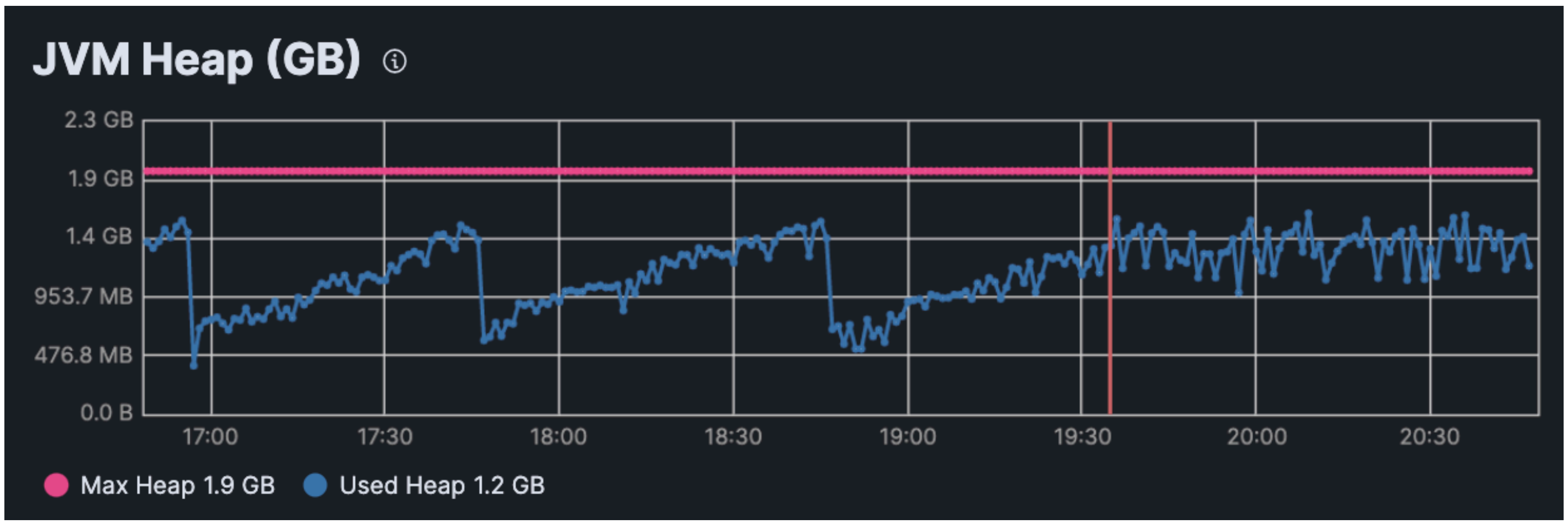
To me, if a Garbage Collection event is denoted by a drop on the blue line in the graph on the right, it seems to me that “GCs [sic] very close together” is more true in what is claimed in [12] to be the good/green case, not the red? Let’s take a look at some better data. Recall that earlier we saw solid event rates from logstash (50K/sec) and healthy (sawtooth) pattern of the JVM metrics in IMG007 and IMG008. In this case the logstash syslog pipeline is basically running in a “default” state (though the workers were explicitly set to 48 to match cpu, but if we didn’t do this, the workers would default to number of cpus) with a batch size of 125. Shortly after I took those screenshots and continued writing, I increased the batch size to 500 and look what happened.


Logstash node: localbox

Transport Address	Events Received	Events Emitted	Config Reloads	Pipeline Workers	Batch Size	Version	Uptime
127.0.0.1:9600	2.5b	2.5b	1	48	125	7.17.4	a day

IMG011 - Logstash metrics with SUT pipeline batch size set at 500 (over previous 125, which is the default settings)

We have a clear deviation from the sawtooth pattern shortly after this change was made. CPU utilization increased a few percent, event latency remained mostly the same, and our event rate dropped to a norm of about 5000 less events per second. Below is a closer look at the JVM Heap graph.



IMG012 - JVM Heap of SUT Logstash after batch size increase to 500. The used heap no longer exhibits a steady climb but instead has become chaotic. No sawtooth == BAD.

I suppose the gist is the same as the (likely old / stale / forgotten) example in [12]; batch size, workers and other factors can effect Logstash JVM performance. But UMMMOOOOONNNNN ELASTIC! Why leave that vague example up when we can use other Elastic products to greater effect? What the heck even is VisualVM?[14] Anyways. Now that we have logstash monitoring enabled and have gone over effects of pipeline settings on the JVM, we can tinker a bit more in the next lab.

SYSLOG LAB - 003

Before we get too deep, a word of caution. Learn from our mistakes. If you go googling around for "LOGSTASH UDP PERFORMANCE" you'll come across a number of blog posts or forum discussion discussing modifications to linux sysctl settings.[15] While it's ultimately the technology owner's decision as to how they want to fiddle with their logstash systems, we recommend not doing so without solid evidence to back up your case (like the metrics above). Those event rates we saw above are nowhere near what we see in a production environment with the following characteristics.[16]

- 4 x logstash nodes at 16 cpu 32G ram
- Syslog data being pushed from numerous systems through load balancers that load balance across those 4 logstash nodes
- 24hr event counts > 1 Billion across multiple syslog sources.
- Event rate hovers between 1-2000/sec, some spikes up to 3-4000/sec for a given pipeline.

Your network may be different, but unless you're pretty positive that your pipeline's event rate is being pushed to it's limit (the only way we see how to really test this is using a similar setup, except with an external stressor node that can generate fake events), it's best not to get too into the weeds tinkering with some settings you found on a blog. That being said, lets get into the weeds tinkering with some settings we found on a blog.

Recall earlier that we created a little wrapper for our stressor node… this is good and all but it's a little tedious / annoying to run the stressor process in the foreground. What's better is running it as a service so we can start/stop/restart it at will. To do this we'll piggyback off of the existing logstash service and make directory /opt/stress for our configs to live in.[17] The high level steps are listed below

- make directory /opt/stress
- copy the existing logstash.service to /opt/stress/stressd.service
- edit the service file, changing the description, user, group. Delete the EnvironmentFile entries (runs fine without them) and modify the ExecStart entry to point to the logstash bin in our tar folder. Set the batch (-b) size to 125 (not technically necessary, but a good reminder of default) and workers to 2, data path to /tmp/data and log directory to /tmp/stressd.log.[18] Set the config path to /opt/stress/stress.conf (we'll create it later).
- fire in the hole

The stress.service and sample commands are on the next page.

---

[14] No offense to the Visual VM project owners also no offense to the original author, it is often the case that the only documentation that exists for some weird rabbit hole is the version 1 document lol (CTRL-F project owners, very fine print, bottom of page, in gray on gray…): https://visualvm.github.io/download.html

[15] INCREASE UDP INPUT PLUGIN PERFORMANCE: https://discuss.elastic.co/t/increase-udp-input-plugin-performance/130798/3

[16] Granted, things aren't exactly equal, for example the production syslog config has 1000s of lines.

[17] LINUX FILESYSTEM HIERARCHY: https://tldp.org/LDP/Linux-Filesystem-Hierarchy/html/opt.html

[18] Technically this was a fat finger on my part, and I left it in (i.e. i usually name actual log files somenamehere.log). Yes I know /var/log/ is probably more appropriate than temp, but 16TB disk YOLOOOOOOOOOOOOOOOOO

```
$ mkdir /opt/stress && cd /opt/stress
$ locate logstash.service
/etc/systemd/system/logstash.service
/etc/systemd/system/multi-user.target.wants/logstash.service
$ cat /etc/systemd/system/logstash.service
[Unit]
Description=logstash

[Service]
Type=simple
User=logstash
Group=logstash
# Load env vars from /etc/default/ and /etc/sysconfig/ if they exist.
# Prefixing the path with '-' makes it try to load, but if the file doesn't
# exist, it continues onward.
EnvironmentFile=-/etc/default/logstash
EnvironmentFile=-/etc/sysconfig/logstash
ExecStart=/usr/share/logstash/bin/logstash "--path.settings" "/etc/logstash"
Restart=always
WorkingDirectory=/
Nice=19
LimitNOFILE=16384

# When stopping, how long to wait before giving up and sending SIGKILL?
# Keep in mind that SIGKILL on a process can cause data loss.
TimeoutStopSec=infinity

[Install]
WantedBy=multi-user.target
$ cp /etc/systemd/system/logstash.service . && mv logstash.service stressd.service
$ ls
stressd.service
$ cat stressd.service
[Unit]
Description=stressd - generates output to logstash node for testing

[Service]
Type=simple
User=root
Group=root
ExecStart=/opt/logstash-files/logstash-8.1.3/bin/logstash "--path.config" /opt/
stress/stress.conf -b 125 -w 2 --path.data=/tmp/data -l /tmp/stressd.log
Restart=always
WorkingDirectory=/
Nice=19
LimitNOFILE=16384

# When stopping, how long to wait before giving up and sending SIGKILL?
# Keep in mind that SIGKILL on a process can cause data loss.
TimeoutStopSec=infinity

[Install]
WantedBy=multi-user.target
```

The only thing left after that is to flesh out the config, which we can copy pasta from our wrapper and breakout into multiple lines like we are sane people. This is pasted on the next page for reference.

```
input {
    generator {
        message => '<161>Jul  2 13:37:00 localbox logstasher[1337]: LOGSTASHER
VOLUME ONE - METRICS,METRICS,METRICS'
        threads => 2
    }
}
output {
    udp {
        host => 'localhost'
        port => '5014'
    }
}
```

The current state of the SUT is as such.

- pipeline.workers: 48 (default based on system cpu)
- pipeline.batch.size: 125 (default)
- UDP input with default settings
  - receive buffer bytes: 106496 (operating system default)
  - queue size: 2000

With these settings in place we have the following observed behavior (from previous)

- healthy JVM sawtooth pattern
- event rate hovering around 40K/sec
- CPU utilization exhibiting a sawtooth (matching the JVM perf) with low of 20% and high of < 30%

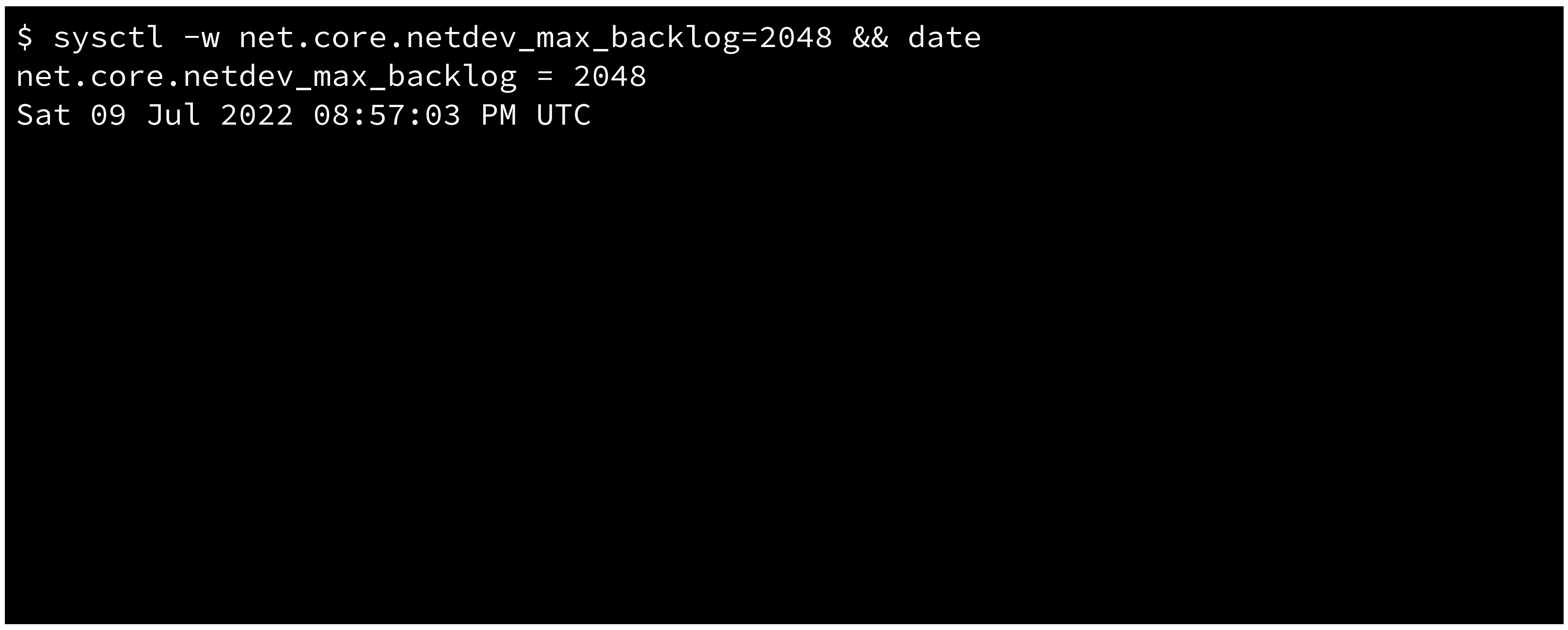
Recall from [14] there are some linux OS settings that people commonly jack with to try and increase performance of the UDP input plugin. The commands for editing these are pasted below as well as the current settings

```
sudo sysctl -w net.core.somaxconn=2048
sudo sysctl -w net.core.netdev_max_backlog=2048
sudo sysctl -w net.core.rmem_max=33554432
sudo sysctl -w net.core.rmem_default=262144
sudo sysctl -w net.ipv4.udp_rmem_min=16384
sudo sysctl -w net.ipv4.udp_mem="2097152 4194304 8388608”

$ sysctl -a  | grep -E 'net.core.(somaxconn|netdev_max_backlog|rmem_max|
rmem_default)|net.ipv4.udp_(rmem_min|mem)' | sort
net.core.netdev_max_backlog = 1000
net.core.rmem_default = 212992
net.core.rmem_max = 212992
net.core.somaxconn = 4096
net.ipv4.udp_mem = 3081390    4108522   6162780
net.ipv4.udp_rmem_min = 4096
```

So how do they compare?

- *net.core.somaxconn* on our system actually 2x that specified in [14].
- *net.core.netdev_max_backlog* on our system is < 1/2 of [14]
- *net.core.rmem_max* on our system is 1/157 of that in [14].
- *net.core.rmem_default* on our system only 49152 units less than [14]
- *net.ipv4.udp_rmem_min* on our system is 1/4 of [14]
- *net.ipv4.udp_rmem_min* value 0 on our system is > than [14], value 1 is < [14] by about 90K, and value 2 is < [14] by about 2 Million.

Hmm what do? Which setting should we change first ? The *somaxconn* setting can probably be ignored entirely, since it's larger on our system.[19] Let's start with *netdev_max_backlog* and let things run in their current state and see if we see any clearly visible changes in the logstash performance. Remember to note the previous settings on your system so you can restore them. Also note the time of our change as you



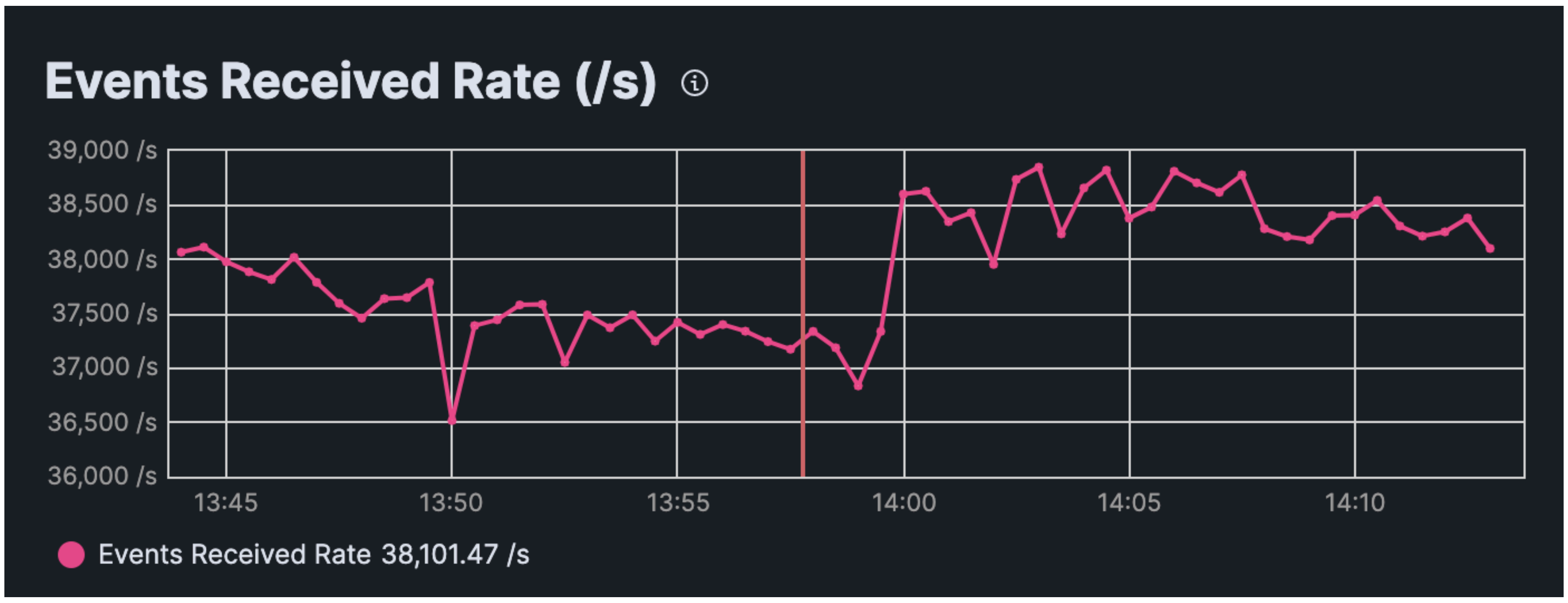
```
$ sysctl -w net.core.netdev_max_backlog=2048 && date
net.core.netdev_max_backlog = 2048
Sat 09 Jul 2022 08:57:03 PM UTC
```

should only be concerned with metrics after this time if you're monitoring what impact if any this change had to performance.[20] We'll also want many metrics, the more the better (requires time, resources), but this may not always be possible to do. Looking at a 30 minute window of metrics (recall change was at 57m pass the hour) before and after the change, we can say that after the change, event rate did climb a little bit. Looking at the larger window of the JVM heap (sawtooth pattern takes about 50m to repeat itself), we see no drastic change to the pattern. It may have begun smoothing out a little bit more.

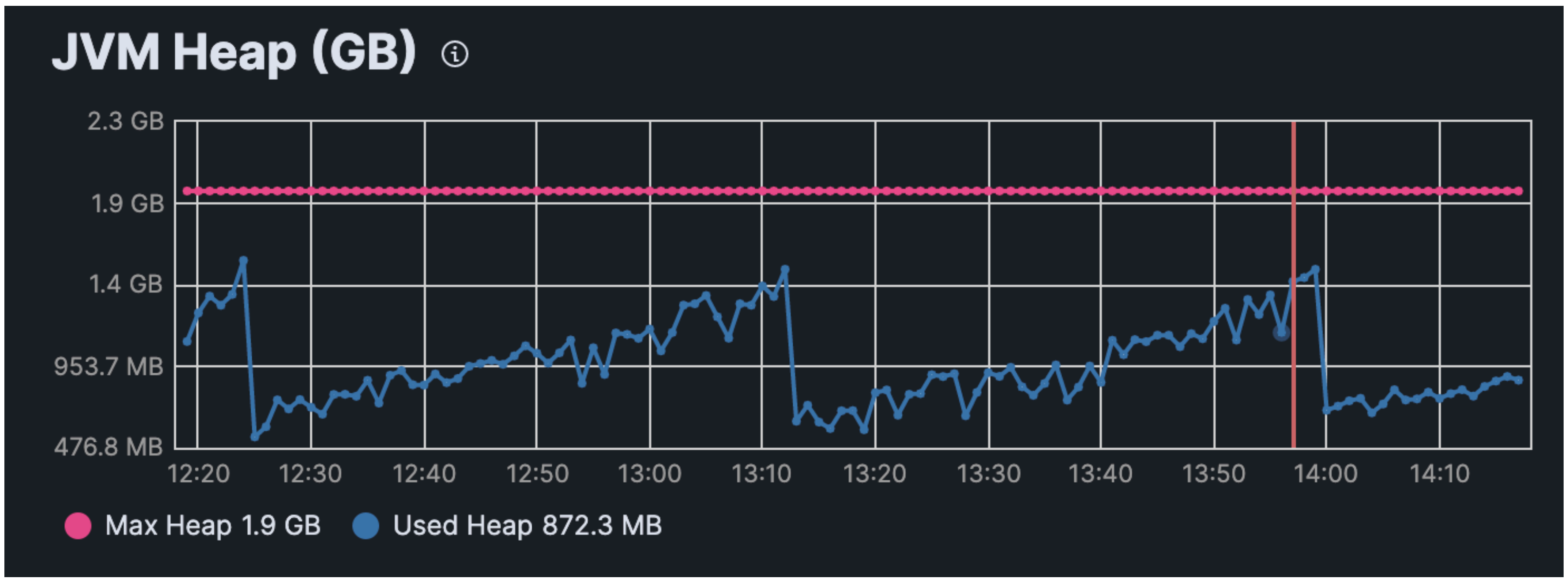
An event rate increase of about 1K is not that impressive when we're already seeing 38-40K/sec (1000 / 38000 = 0.0263 = 2.63 %). If you look back in the metrics at 2hrs ago or greater, you can hardly notice it at all.

[19] RTFM: $ man listen: If the backlog argument is greater than the value in /proc/sys/net/core/somaxconn, then it i silently truncated to that value. Since Linux 5.4, the default in this file is 4096; in earlier kernels, the default value is 128. In kernels before 2.4.25, this limit was a hard coded value, SOMAXCONN, with the value 128.

[20] From contextual experience, sysctl settings take effect immediately on the host system and do not require a restart of affected services (logstash for example).



IMG013 - Events Received rate climbing slightly after the change to net.core.netdev_max_backlog



IMG014 - JVM heap graph may have gotten smoother after the change but we'll have to wait an hour or more to see if this is one off or not.

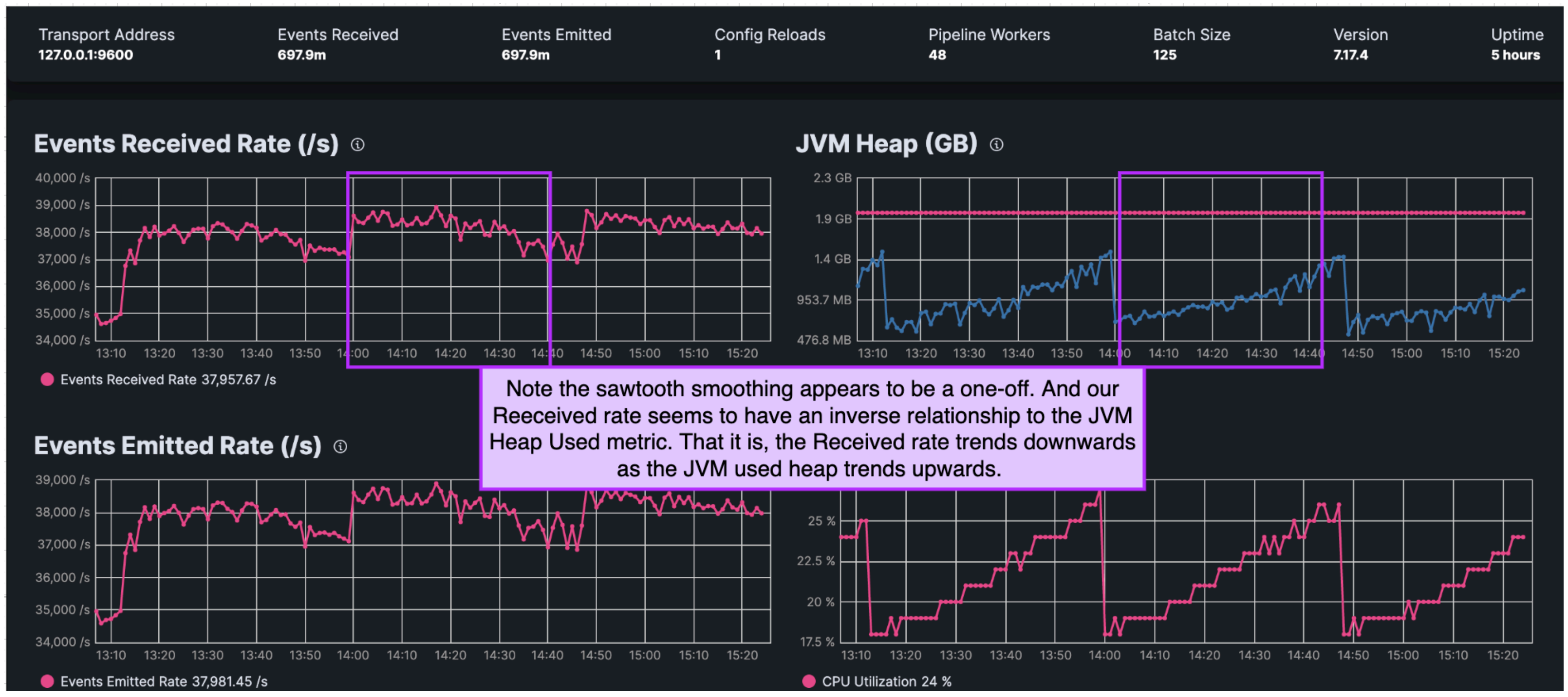
There's no wow effect thus far with this change. But we can let it ride and ignore it for a while to see what gives. Based off first impressions, it seems like our primary impact may have been a cleaner JVM sawtooth metric, but we'll have to observe the stats over a few hours to see whether or not this was a one off occurrence. Another interesting test will be to be to drop this setting to something absurdly low as well as something absurdly high and measure impact. At an absurdly low number (max backlog of 10), I would anticipate that our events received rate drops significantly due to inbound UDP messages filling the backlog and then having no place to go.[21] At the absurdly high end (let's say max backlog 102400) I'm unsure what we could see. Perhaps backup in the Recv-Q of our UDP listening port (5014). Refer to the example on the next page.

[21] This is purely speculation. The relationship is not that simple; we're talking about the interrelationship between the Network Interface Card (NIC) and how the Linux kernel is setting limits on a backlog which is likely set per cpu or something. We'll dig in more based on our findings.

```
$ netstat -anu
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
udp        0      0 0.0.0.0:38193           0.0.0.0:*
udp        0      0 127.0.0.1:46517         127.0.0.1:46517         ESTABLISHED
udp        0      0 0.0.0.0:60907           0.0.0.0:*
udp        0      0 127.0.0.53:53           0.0.0.0:*
udp   213760      0 0.0.0.0:5014            0.0.0.0:*
udp        0      0 0.0.0.0:5353            0.0.0.0:*
udp        0      0 0.0.0.0:8125            0.0.0.0:*
udp6       0      0 :::42733                :::*
udp6       0      0 fe80::266e:96ff:fe3:546 :::*
udp6       0      0 :::5353                 :::*
```

On an actively listening UDP port receiving a lot of traffic, it's normal to see the "*Recv-Q*" increment and drain as traffic gets processed by the application (Logstash). A symptom of trouble at the application side would be a "*Recv-Q*" value that appears pinned in that it increments up to a system defined limit, and hovers at that number, indicating that the application cannot keep up with the load enough to drain the queue. [22]

Let's check back on the metrics after our sysctl change. At this point I'm getting impatient, and the next iteration of the sawtooth indicates that the smoothing was a one off scenario.



Transport Address	Events Received	Events Emitted	Config Reloads	Pipeline Workers	Batch Size	Version	Uptime
127.0.0.1:9600	697.9m	697.9m	1	48	125	7.17.4	5 hours

Note the sawtooth smoothing appears to be a one-off. And our Reeceived rate seems to have an inverse relationship to the JVM Heap Used metric. That it is, the Received rate trends downwards as the JVM used heap trends upwards.

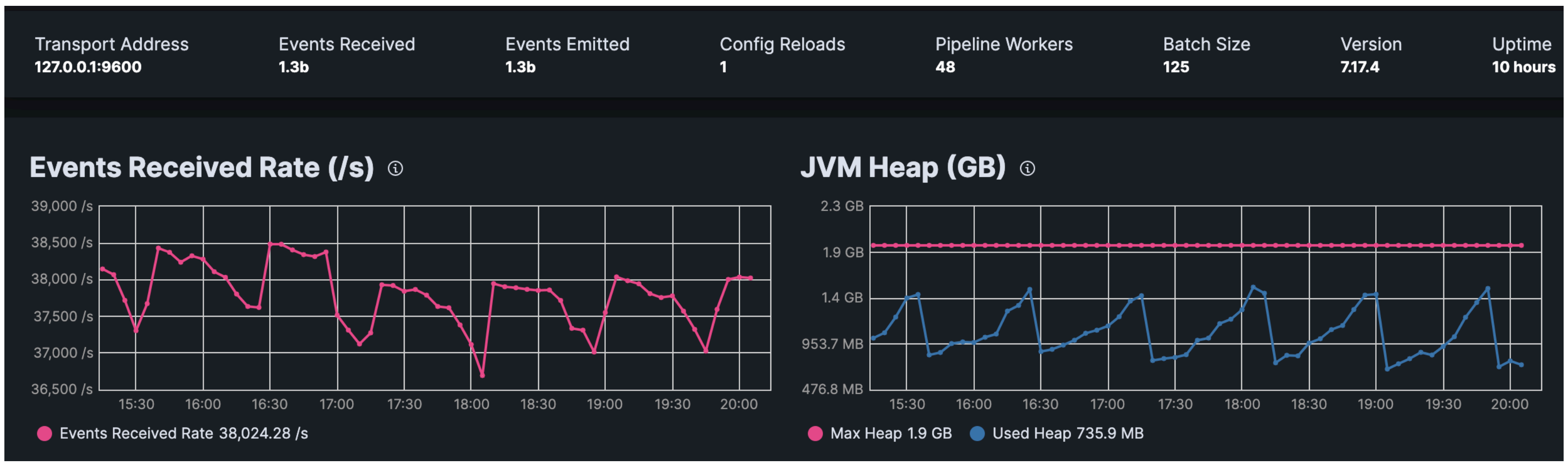
IMG015 - Looking at at ~2.5 hour window of metrics (change executed at 13:57 on these graphs) there are no drastic changes in performance, however we do notice that during a given 50 minute window (time it takes for JVM sawtooth pattern to emerge) that the Events Received Rate and JVM Heap Used metric have an inverse relationship.

[22] If you're curious what you're UDP receive queues are doing on linux, can watch them with: watch -n 1 netstat -anu

While our event rate during this window was mostly boring, we happen to notice that our event rate trends downward during the same time frame that JVM Heap Used is trending upwards. Let's finagle with the net.core.netdev_max_backlog setting again, this time dropping it to a low number (10). We'll be sure to note the time of this change again (10:39PM UTC == 3:39PM LOCAL).

```
$ sysctl -w net.core.netdev_max_backlog=10 && date
net.core.netdev_max_backlog = 10
Sat 09 Jul 2022 10:39:32 PM UTC
```

Almost 5 hours has passed and we see this trend in the metrics.



Transport Address 127.0.0.1:9600 | Events Received 1.3b | Events Emitted 1.3b | Config Reloads 1 | Pipeline Workers 48 | Batch Size 125 | Version 7.17.4 | Uptime 10 hours

IMG016 - Looking at at ~5 hour window from when we dropped the max backlog setting; JVM Used Heap sawtooth is essentially the same, but we see the event rate appears more sporadic, but still having an inverse relationship to the JVM metric. The events received rate stays between 37000 and 38000 for the most part.

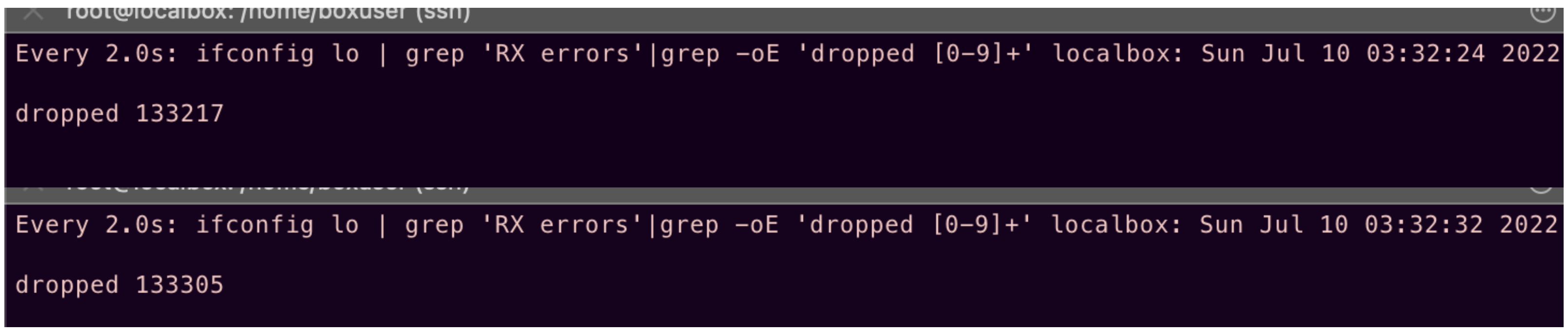
No major effect that we can clearly see. It would be curious to see if packets were dropped though. Recall that our stressor is shipping to localhost:5014/UDP. There are multiple ways to check the number of dropped packets for a given interface but we'll just use ifconfig.

```
$ ifconfig lo
lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 2371013101  bytes 882926794773 (882.9 GB)
        RX errors 0  dropped 127117  overruns 0  frame 0
        TX packets 2371013101  bytes 882926794773 (882.9 GB)
```

Here we see the dropped count is 127117. But unfortunately we didn't think to take a look at this before. Is this a normal number? Is this number increasing? We can monitor this live on one terminal screen with the following one liner

```
$ watch “ifconfig lo | grep ‘RX errors’| grep -oE ‘dropped [0-9]+’”
```

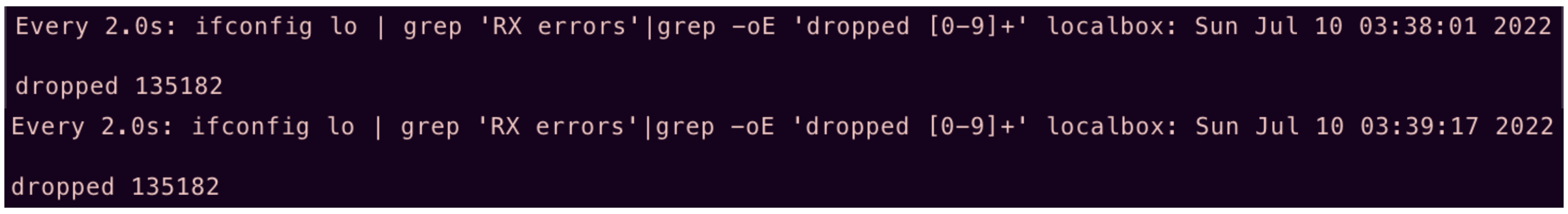
If we watch this for a bit we should see that the number is indeed incrementing by about 20 every 2 seconds. sample below.



```
root@localbox: /home/boxuser (ssh)
Every 2.0s: ifconfig lo | grep 'RX errors'|grep -oE 'dropped [0-9]+' localbox: Sun Jul 10 03:32:24 2022

dropped 133217

Every 2.0s: ifconfig lo | grep 'RX errors'|grep -oE 'dropped [0-9]+' localbox: Sun Jul 10 03:32:32 2022

dropped 133305
```

IMG017 - Using the *watch* command to run 'ifconfig lo' every 2 seconds and process the output with grep shows that our dropped packets on the *lo* interface are incrementing.

From another terminal we can set the max backlog setting back to 1000 and keep an eye on our watch job. If indeed a low *net.core.netdev_max_backlog* setting is causing dropped packets, we should see that the number of dropped packets remains stable after we change the setting back.



```
Every 2.0s: ifconfig lo | grep 'RX errors'|grep -oE 'dropped [0-9]+' localbox: Sun Jul 10 03:38:01 2022

dropped 135182
Every 2.0s: ifconfig lo | grep 'RX errors'|grep -oE 'dropped [0-9]+' localbox: Sun Jul 10 03:39:17 2022

dropped 135182
```

IMG018 - Here we see that after setting *net.core.netdev_max_backlog back to its default setting on our system (1000), the number of dropped packets stops incrementing and no drop packets are seen over 1 minute window, when previously we were dropping about 10 packets a second.*

Thus far we can conclude the following.

1. Increasing *net.core.netdev_max_backlog* from system default of (1000) to 2048 had no clearly discernible impact on performance.

2. Dropping *net.core.netdev_max_backlog* to a value too low (10) can result in dropped packets at the NIC, but also had no clearly discernible impact on performance (although surely there was a non-zero impact, as dropped packets == missed events).

3. Increasing this setting would probably only be warranted if we were seeing dropped packets on an interface where Logstash has a listening port

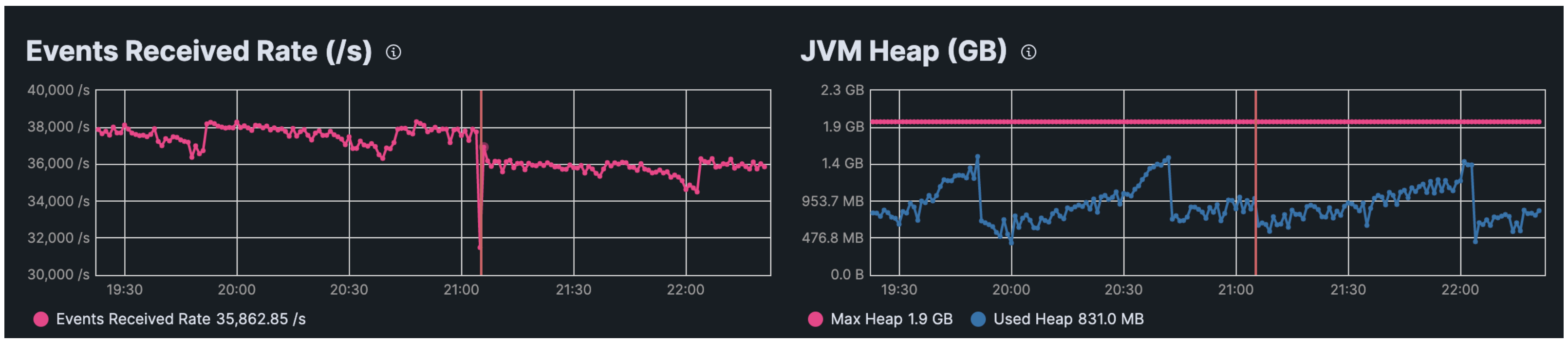
Before attempting to mess with more system settings. Let's tinker with settings of the input plugin itself. [23] There are several settings that stand out as likely candidates that could impact performance.

- **queue_size**: This is the number of unprocessed UDP packets you can hold in memory before packets will start dropping (default 2000). This is an unlikely candidate for increasing performance unless we note packets being dropped under normal conditions (although it's unclear where the drop is occurring… at the NIC? or at the application side, when the NIC considers it a done deal, i.e. would otherwise not increment the dropped packet counter?)

- **receive_buffer_bytes**: The socket receive buffer size in bytes. If option is not set, the operating system default is used. The operating system will use the max allowed value if receive_buffer_bytes is larger than allowed. Consult your operating system documentation if you need to increase this max allowed value (this usually shows up in /var/log/logstash/logstash-plain.log on the "UDP listener started" line, for example on our system logstash reports the value 106496; which is odd because this should technically be the value of *net.core.rmem_default* but as we saw earlier, that value is 212992 which is 2x greater)

- **workers**: Number of threads processing packets (default 2). This seems like the most logical candidate for performance impact, since we've already seen earlier how increasing pipeline workers (handles filter and output) can dramatically increase performance (as measured by event rate).

First we'll take a stab at workers and see if this has any impact. We're interested in major increases, so let's go beyond doubling this (to 4) and bump it to 24 which is 50% of our available cpu. The change was executed at Sun 10 Jul 2022 04:06:07 AM UTC. After an hour or so we have some clearly noticeable differences in the metrics.

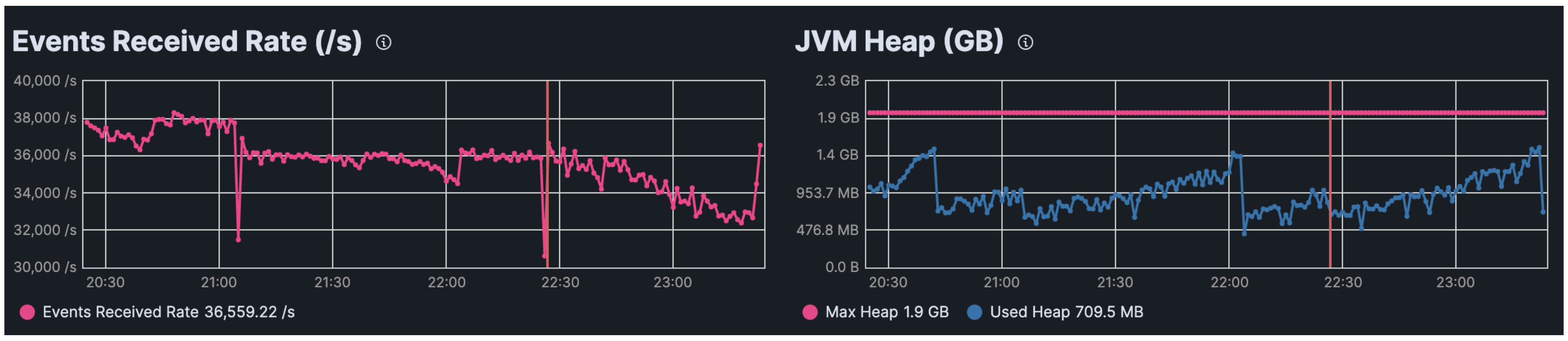


IMG019 - After increasing the UDP input workers count from 2 (default) => 24 (50% of cpu) we have a steady JVM Heap sawtooth but our events received rate has decreased from previous rates, now hovering around 36000 and exhibiting the same inverse relationship to JVM usage in that the event rate trends downward.

So we degraded our event rate slightly by increasing the UDP workers to 24 (50 % of cpu). The new normal at that setting is around 36K/sec. Again, not very drastic but worth noting. We can move on to

[23] LOGSTASH UDP INPUT: https://www.elastic.co/guide/en/logstash/current/plugins-inputs-udp.html

see what the behavior is when dropping the UDP input workers setting to 1 (decrease by 50% from default). The change executed at Sun 10 Jul 2022 05:26:50 AM UTC yields the following.



IMG020 - After decreasing the UDP input workers count to 1 (50% of default) we have a steady JVM Heap sawtooth and our events received rate has become considerably unstable, peaking near 36K then trending downward over about the next 50 minutes.

In this state, the max event rate is still near 36K however the inverse relationship to the JVM Heap Used metric is much more pronounced. At this point, we're seeing diminishing returns with each of our little tweaks, but we are at least making some interesting observations with regard to how these little tweaks impact the event rate and how the event rate of the UDP input is tied to the JVM's used heap under different conditions.

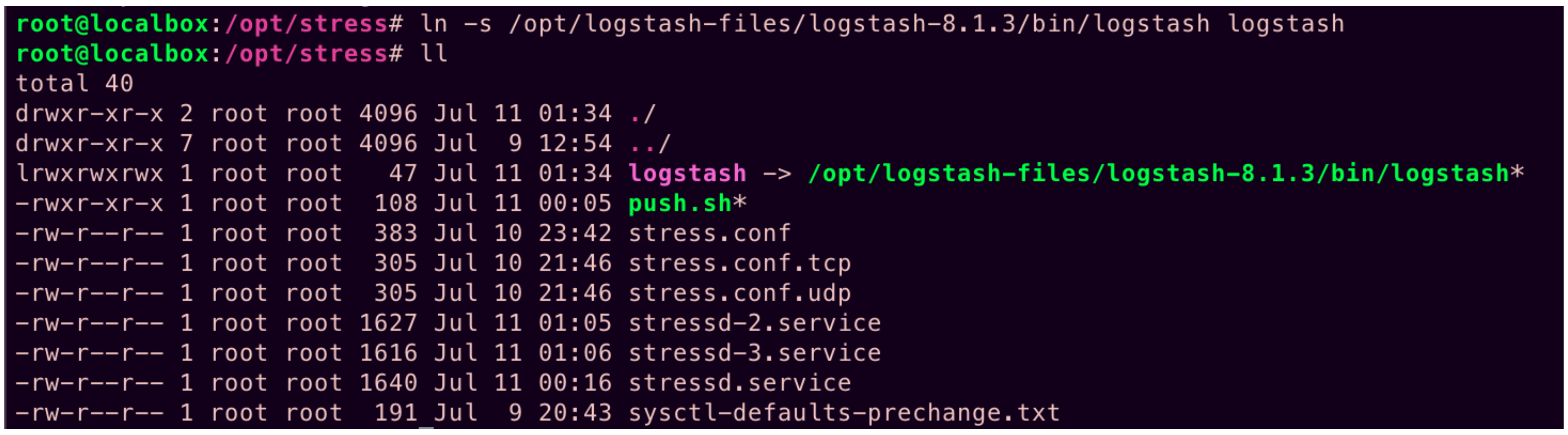
One point we have not yet considered is that we may be hitting the maximum amount of stress we can apply using 1 logstash stressor node process. The good news is that we've already done the leg work of creating *stressd.service,* so it will be easy to make copies of that service file, tweak them to use different data directories and logging directories, then push those to */etc/systemd/system* and start up a second or third stressor at will. This will allow us to further determine where our bottleneck is. If the bottleneck is at the SUT configuration, then we should see no significant change in the event rate with 2x or 3x the stress; perhaps we would even see degraded performance. If the bottleneck is indeed the amount of stress we can get from 1 logstash stressor process, then when we turn up additional processes we should see increased event rates. Let's go over this in the next lab.

SYSLOG LAB - 004

In this lab we will attempt to achieve a near 3x event rate (150K/sec). To do this we'll set up 2 additional *stressd* services called *stressd-2.service* and *stressd-3.service*. We'll start off with our SUT pipeline settings as such:

- UDP input will have default settings (no queue tweaks, no buffer tweaks, no worker tweaks)
- *pipeline.batch.size* will be 125 (default)
- *pipeline.workers* will be 48 (# of system cpu)

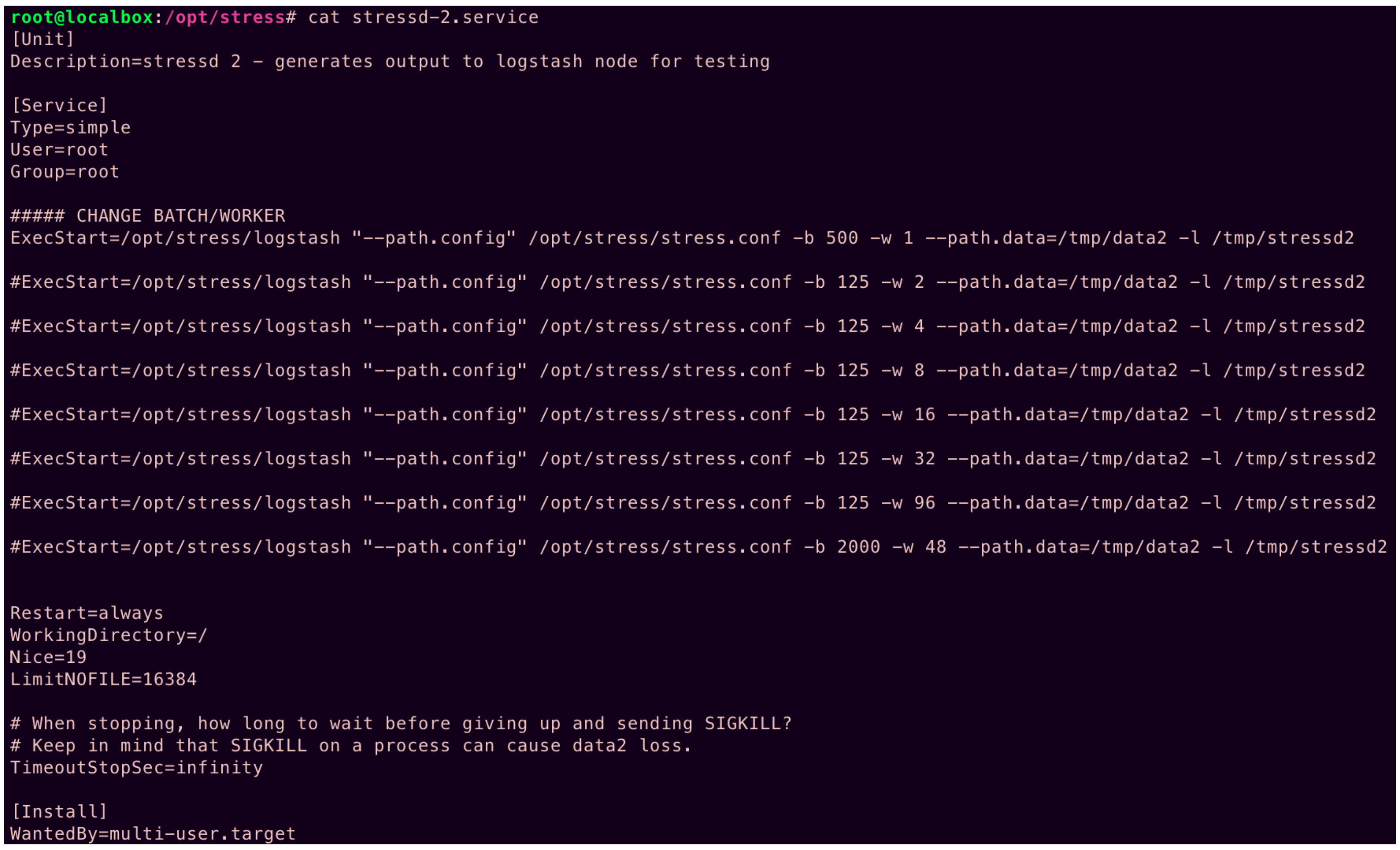
We'll make a symlink to the /opt/logstash-files/logstash-8.1.3/bin/logstash file. This is something we should have done earlier because it helps us keep the service file a little more readable by shortening the path.



```
root@localbox:/opt/stress# ln -s /opt/logstash-files/logstash-8.1.3/bin/logstash logstash
root@localbox:/opt/stress# ll
total 40
drwxr-xr-x 2 root root 4096 Jul 11 01:34 ./
drwxr-xr-x 7 root root 4096 Jul  9 12:54 ../
lrwxrwxrwx 1 root root   47 Jul 11 01:34 logstash -> /opt/logstash-files/logstash-8.1.3/bin/logstash*
-rwxr-xr-x 1 root root  108 Jul 11 00:05 push.sh*
-rw-r--r-- 1 root root  383 Jul 10 23:42 stress.conf
-rw-r--r-- 1 root root  305 Jul 10 21:46 stress.conf.tcp
-rw-r--r-- 1 root root  305 Jul 10 21:46 stress.conf.udp
-rw-r--r-- 1 root root 1627 Jul 11 01:05 stressd-2.service
-rw-r--r-- 1 root root 1616 Jul 11 01:06 stressd-3.service
-rw-r--r-- 1 root root 1640 Jul 11 00:16 stressd.service
-rw-r--r-- 1 root root  191 Jul  9 20:43 sysctl-defaults-prechange.txt
```

IMG021 - Here we make a symlink in /opt/stress that points to the logstash script to keep our service files more readable.

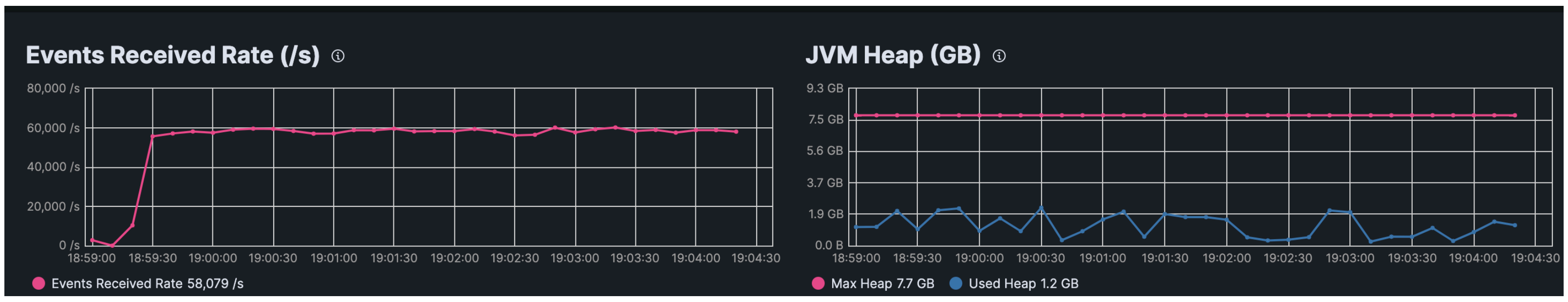
Now that we have our symlink in place we can copy the *stressd.service* file and make edits to it. Below is what we have for *stressd-2.service*. You can see we have multiple commented lines inserted to make modifying the service to increase worker threads or batch size simple to do.



```
root@localbox:/opt/stress# cat stressd-2.service
[Unit]
Description=stressd 2 - generates output to logstash node for testing

[Service]
Type=simple
User=root
Group=root

##### CHANGE BATCH/WORKER
ExecStart=/opt/stress/logstash "--path.config" /opt/stress/stress.conf -b 500 -w 1 --path.data=/tmp/data2 -l /tmp/stressd2

#ExecStart=/opt/stress/logstash "--path.config" /opt/stress/stress.conf -b 125 -w 2 --path.data=/tmp/data2 -l /tmp/stressd2

#ExecStart=/opt/stress/logstash "--path.config" /opt/stress/stress.conf -b 125 -w 4 --path.data=/tmp/data2 -l /tmp/stressd2

#ExecStart=/opt/stress/logstash "--path.config" /opt/stress/stress.conf -b 125 -w 8 --path.data=/tmp/data2 -l /tmp/stressd2

#ExecStart=/opt/stress/logstash "--path.config" /opt/stress/stress.conf -b 125 -w 16 --path.data=/tmp/data2 -l /tmp/stressd2

#ExecStart=/opt/stress/logstash "--path.config" /opt/stress/stress.conf -b 125 -w 32 --path.data=/tmp/data2 -l /tmp/stressd2

#ExecStart=/opt/stress/logstash "--path.config" /opt/stress/stress.conf -b 125 -w 96 --path.data=/tmp/data2 -l /tmp/stressd2

#ExecStart=/opt/stress/logstash "--path.config" /opt/stress/stress.conf -b 2000 -w 48 --path.data=/tmp/data2 -l /tmp/stressd2


Restart=always
WorkingDirectory=/
Nice=19
LimitNOFILE=16384

# When stopping, how long to wait before giving up and sending SIGKILL?
# Keep in mind that SIGKILL on a process can cause data2 loss.
TimeoutStopSec=infinity

[Install]
WantedBy=multi-user.target
```

IMG022 - This is a sample of *stressd-2.service* which we'll use to increase the input load on the SUT. Using a symlink in /opt/stress helped keep the config more readable, and we include multiple commented out ExecStart settings so we can change the batch size and worker threads as needed.

You maybe also noticed bash script in IMG021 called *push.sh ;* this is just a wrapper to a one liner that copies the relevant service file to /etc/systemd/system, runs *systemctl daemon-reload* and then restarts the *stressd* (or *stressd-2*, *stressd-3*) service. We can then start up each service and monitor the behavior of our event rate.



IMG023 - Shortly after starting up our additional stressor services (all using the same config, with batch size 125 and 2 workers) we see that we indeed pushed our event rate to near 60K

So not quite the result we anticipated. We increased our event rate about 15-20K over previous levels, but we've leveled out at about 60K. The first thing we check is whether or not packet drops are incrementing on the *lo* interface (where SUT logstash is listening and where our stressors are shipping) but we see that drops have not incremented beyond the last count we saw (135182). So where is the bottleneck?

1. It's unlikely that our bottleneck is the stressors, we've already proven that one stressor using generator input plugin and UDP output is capable of 40-50K EPS

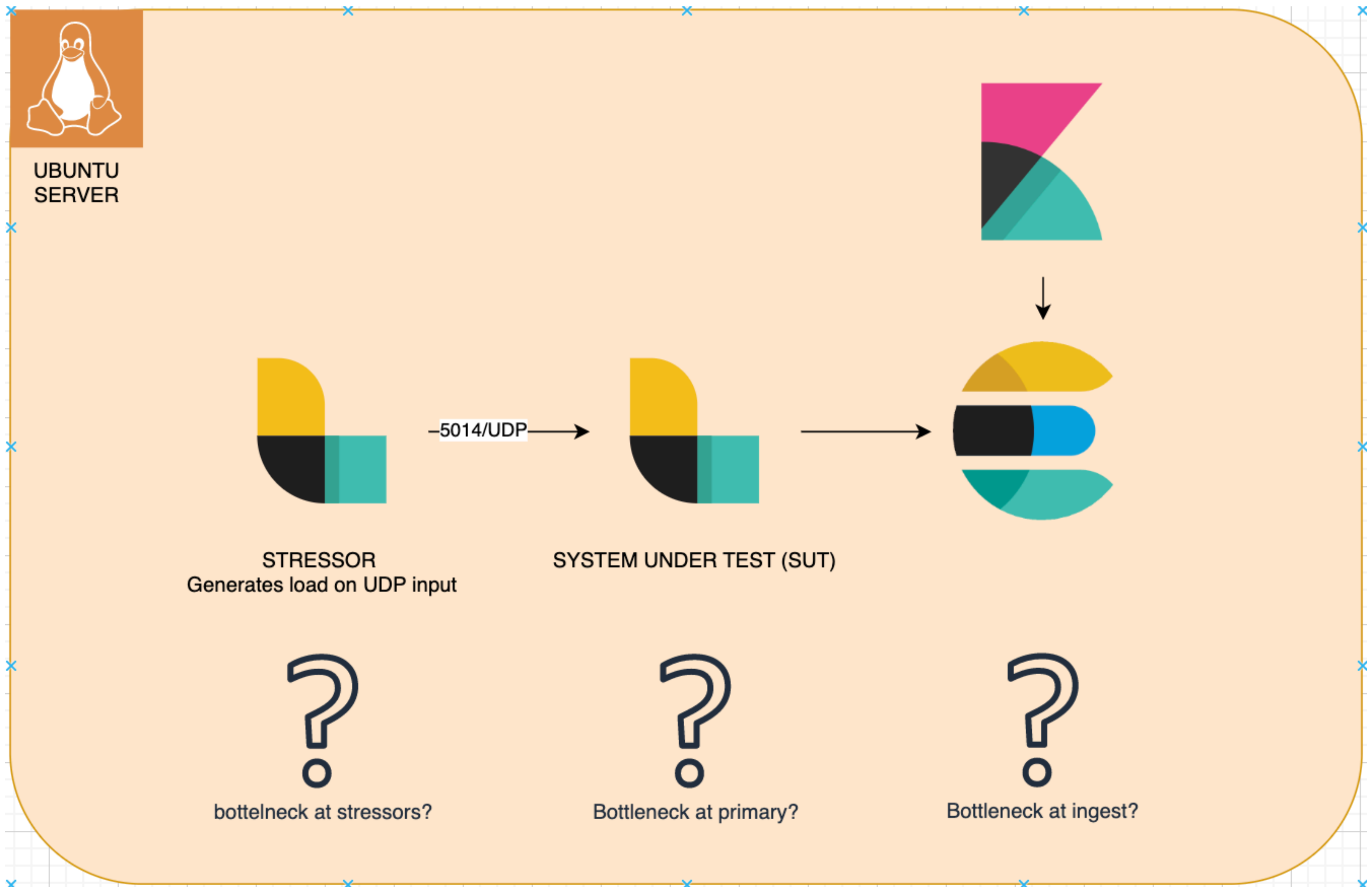
2. It's possible that we're hitting a cap on the throughput of SUT logstash with UDP input and our current configuration (i.e. maybe a TCP input could take things higher, or maybe there's some tuning we could do at the input or JVM side). [24]

3. It's possible that the bottleneck is at the Elasticsearch end. Recall our lab is only running 1 local elasticsearch index, which we have yet to touch. We could try tuning elasticsearch for indexing speed.[25]

4. We could minimize disk ops in our pipeline (we're using the metrics filter to meter events still, and based on whether or not an event is from the stressor or is a metric event, we're writing to disk and outputting to a different index).

[24] There are also other things to consider; for example, using persistent queues for a pipeline is likely to slow event rate down (have to write queue to disk instead of in memory queue). We're also still dumping some events to file (/var/log/syslog.metrics.json) which could impact performance.

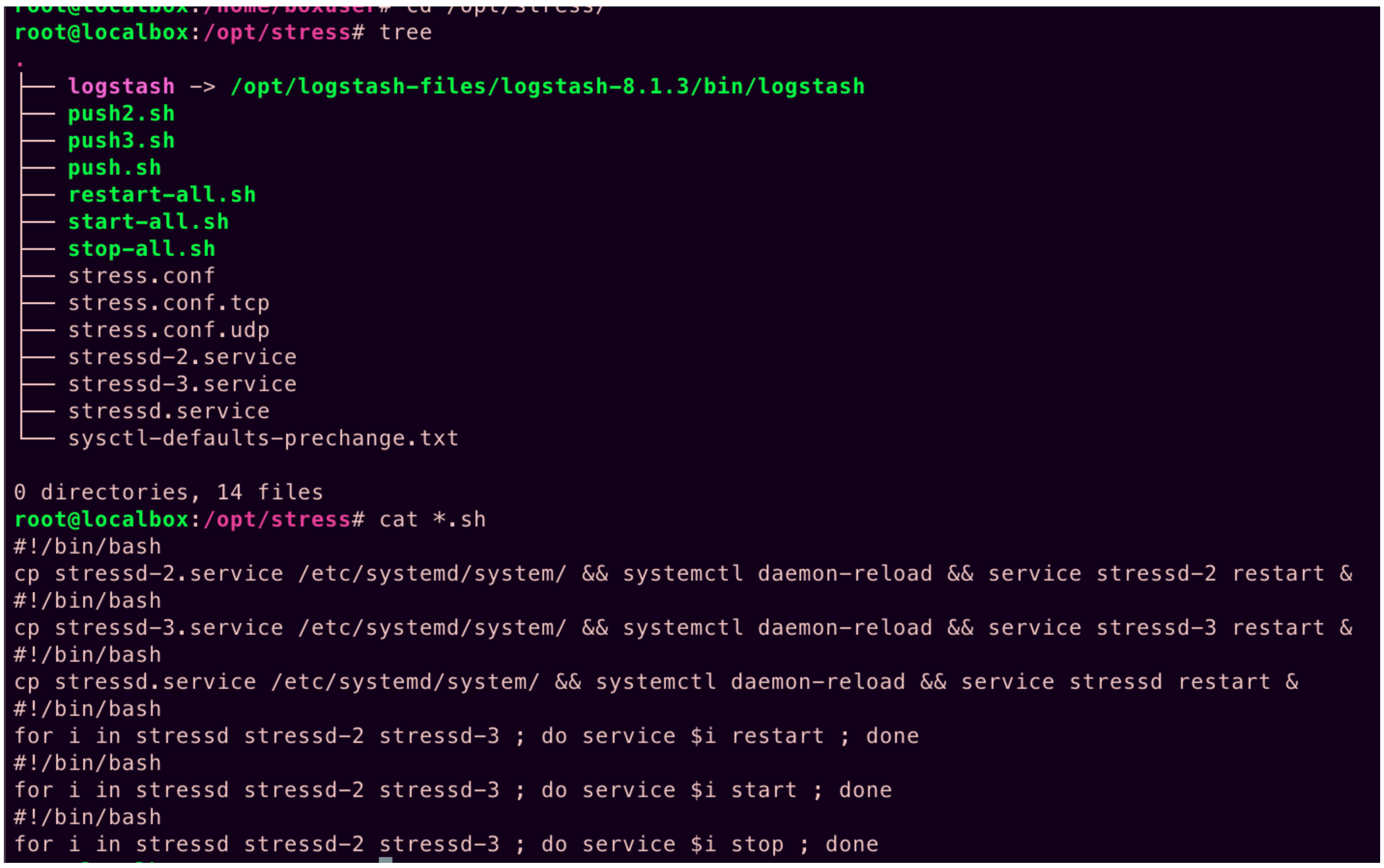
[25] ELASTIC; Tune for indexing speed: https://www.elastic.co/guide/en/elasticsearch/reference/current/tune-for-indexing-speed.html



IMG024 - Where's our bottleneck? It could be at the SUT node, but it could also be at the Elasticsearch side. It seems unlikely the that the stressor/s would be the bottleneck since we've already seen high event rates with 1 stressor.

So which should we tackle first ? It's up to you… we decided to switch up the TCP input. Accepting syslog over TCP isn't uncommon. It's not something we see a lot of but nonetheless, it can and will be done eventually. We'll gloss over setting up the stressors to ship to TCP instead of UDP. We've already laid the groundwork for making modification of the stressors pretty straightforward. On the next page is a screenshot of our /opt/stress layout for reference as well as our little helper scripts. A few points before you modify the stressor config

- **Be sure that the tcp output on the stressors uses either the *plain* or *json_lines* codec;** the default *(json)* will just write a continuous non-delimited stream of events (i.e. '{"dude":1}{"dude":2}' to the SUT which Logstash will try to read as 1 massive event or events, which will continue to eat up the heap until the heap is exhausted resulting in an Out Of Memory (OOM) error. But try it if you want to see a node crash.



```
root@localbox:/opt/stress# tree
.
├── logstash -> /opt/logstash-files/logstash-8.1.3/bin/logstash
├── push2.sh
├── push3.sh
├── push.sh
├── restart-all.sh
├── start-all.sh
├── stop-all.sh
├── stress.conf
├── stress.conf.tcp
├── stress.conf.udp
├── stressd-2.service
├── stressd-3.service
├── stressd.service
└── sysctl-defaults-prechange.txt

0 directories, 14 files
root@localbox:/opt/stress# cat *.sh
#!/bin/bash
cp stressd-2.service /etc/systemd/system/ && systemctl daemon-reload && service stressd-2 restart &
#!/bin/bash
cp stressd-3.service /etc/systemd/system/ && systemctl daemon-reload && service stressd-3 restart &
#!/bin/bash
cp stressd.service /etc/systemd/system/ && systemctl daemon-reload && service stressd restart &
#!/bin/bash
for i in stressd stressd-2 stressd-3 ; do service $i restart ; done
#!/bin/bash
for i in stressd stressd-2 stressd-3 ; do service $i start ; done
#!/bin/bash
for i in stressd stressd-2 stressd-3 ; do service $i stop ; done
```

IMG025 - Tree structure of */opt/stress* as well as our one-liner helper scripts to copy edited service files to */etc/systemd/system* and start / stop / restart all 3 stressor services.

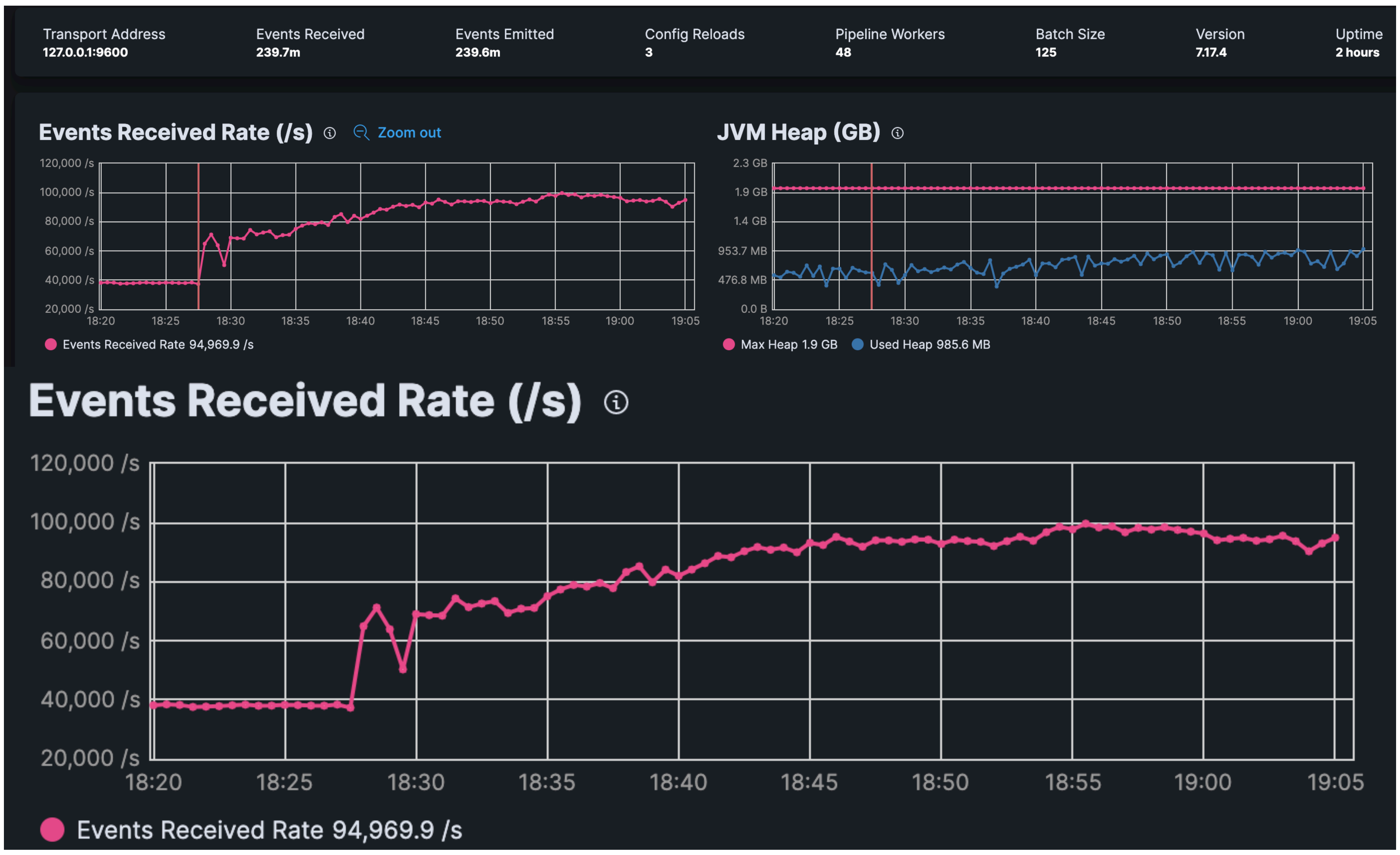
Fortunately, switching to TCP input at the SUT did result in increased EPS (almost 100K!!!…. almost). Not quite the 3x increase that we had hoped for but pretty impressive nonetheless. Each of the stressors was running a default batch size and 2 workers. The SUT was using default batch size and 48 workers (default to # of system CPU). The event rate graphs are listed on the next page.

Where do we go from here? At this point it's unclear where the bottleneck is. The fact that we saw an event rate increase to near 100K with multiple stressors added to the fact that we're still not seeing drops at the NIC seems to indicate that the bottleneck is somewhere downstream from the stressors. Consider the following.

1. If the bottleneck is at the SUT (i.e. we've reached the max event rate for 1 logstash instance) then using the same config, on a second SUT process should push push beyond 100K events per second (we could try this on our lab node, but we may run into resource contention, recall that SUT logstash is using 48 workers, standing up a second would have 96 total workers or 2 * CPU, in addition to the threads used by stressors, elasticsearch, and other system process)

2. If the bottleneck is at Elasticsearch, then we should see an event rate beyond 100K with addition of a second node.

The easiest test is the addition of second logstash node which is left as an exercise for the reader .



IMG026 - Switching the SUT to use TCP input yielded event rates near 100K/sec.

2. PARTING WORDS

As is often the case, our time together has come to a close much to soon. Hopefully it wasn't a waste of the readers time. If so, feel free to flame the comments section of the site. Alternatively, a strongly worded letter is accepted, but only if you use ChatGPT to have it written in Shakespearean style so we're entertained. All jokes aside, happy logging and good wishes.

FIN