Hortonworks Cybersecurity Package

Tuning Guide

(September 12, 2017)

docs.cloudera.com

Hortonworks Cybersecurity Package: Tuning Guide

Hortonworks Cybersecurity Package (HCP) is a modern data application based on Apache Metron, powered by Apache Hadoop, Apache Storm, and related technologies.

HCP provides a framework and tools to enable greater efficiency in Security Operation Centers (SOCs) along with better and faster threat detection in real-time at massive scale. It provides ingestion, parsing and normalization of fully enriched, contextualized data, threat intelligence feeds, triage and machine learning based detection. It also provides end user near real-time dashboards.

Based on a strong foundation in the Hortonworks Data Platform (HDP) and Hortonworks DataFlow (HDF) stacks, HCP provides an integrated advanced platform for security analytics.

Please visit the Hortonworks Data Platform page for more information on Hortonworks technology. For more information on Hortonworks services, please visit either the Support or Training page. Feel free to Contact Us directly to discuss your specific needs.

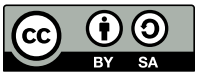

Table of Contents

1. Overview

This document provides guidance from our experiences tuning the Apache Metron Storm topologies for maximum performance. You'll find suggestions for optimum configurations under a 1 Gbps load along with some guidance around the tooling we used to monitor and assess our throughput.

In the simplest terms, Hortonwork Cybersecurity Package powered by Apache Metron is a streaming architecture created on top of Kafka and three main types of Storm topologies: parsers, enrichment, and indexing. Each parser has its own topology and there is also a highly performant, specialized spout-only topology for streaming PCAP data to HDFS. We found that the architecture can be tuned almost exclusively through using a few primary Storm and Kafka parameters along with a few Metron-specific options. You can think of the data flow as being similar to water flowing through a pipe, and the majority of these options assist in tweaking the various pipe widths in the system.

2. General Tuning Suggestions

Note that there is currently no method for specifying the number of tasks from the number of executors in Flux topologies (enrichment, indexing). By default, the number of tasks will equal the number of executors. Logically, setting the number of tasks equal to the number of executors is sensible. Storm enforces num executors <= num tasks. The reason you might set the number of tasks higher than the number of executors is for future performance tuning and rebalancing without the need to bring down your topologies. The number of tasks is fixed at topology startup time whereas the number of executors can be increased up to a maximum value equal to the number of tasks.

When configuring Storm Kafka spouts, we found that the default values for poll.timeout.ms, offset.commit.period.ms, and max.uncommitted.offsets worked well in nearly all cases. As a general rule, it was optimal to set spout parallelism equal to the number of partitions used in your Kafka topic. Any greater parallelism will leave you with idle consumers since Kafka limits the maximum number of consumers to the number of partitions. This is important because Kafka has certain ordering guarantees for message delivery per partition that would not be possible if more than one consumer in a given consumer group were able to read from that partition.

3. Component Tuning Levers

This chapter provides guidelines for tuning levers for components that are key to HCP.

- Kafka
  - Number partitions
- Storm
  - Kafka spout
    - Polling frequency
    - Polling timeouts
    - Offset commit period
    - Max uncommitted offsets
  - Number workers (OS processes)
  - Number executors (threads in a process)
  - Number ackers
  - Max spout pending
  - Spout and bolt parallelism
- HDFS
  - Replication factor

3.1. Kafka Tuning

The main lever you're going to work with when tuning Kafka throughput will be the number of partitions. A handy method for deciding how many partitions to use is to first calculate the throughput for a single producer (p) and a single consumer (c), and then use that with the desired throughput (t) to roughly estimate the number of partitions to use. You would want at least max(t/p, t/c) partitions to attain the desired throughput. For more information, see Confluent - How to choose the number of topics/partitions in a Kafka cluster?.

3.2. Storm Tuning

There are quite a few options you will be confronted with when tuning your Storm topologies and this is largely trial and error. As a general rule of thumb, we recommend starting with the defaults and smaller numbers in terms of parallelism while iteratively working up until the desired performance is achieved. You will find the offset lag tool indispensable while verifying your settings.

We won't go into a full discussion about Storm's architecture - see references section for more info - but there are some general rules of thumb that should be followed. First, it's important to understand the ways you can impact parallelism in a Storm topology.

- num tasks
- num executors (parallelism hint)
- num workers

Tasks are instances of a given spout or bolt, executors are threads in a process, and workers are jvm processes. You'll want the number of tasks as a multiple of the number of executors, the number of executors as multiple of the number of workers, and the number of workers as a multiple of the number of machines. The main reason for this approach is that it will give a uniform distribution of work to each machine and jvm process. More often than not, your number of tasks will be equal to the number of executors, which is the default in Storm. Flux does not actually provide a way to independently set number of tasks, so for enrichments and indexing which use Flux, num tasks will always equal num executors.

You can change the number of workers via the property `topology.workers`.

Other Storm Settings

```
```
topology.max.spout.pending
```
```

This is the maximum number of tuples that can be in a field (for example, not yet acked) at any given time within your topology. You set this as a form of back pressure to ensure you don't flood your topology.

```
topology.ackers.executors
```

This specifies how many threads should be dedicated to tuple acking. We found that setting this equal to the number of partitions in your inbound Kafka topic worked well.

spout-config.json

```
{
    ...
    "spout.pollTimeoutMs" : 200,
    "spout.maxUncommittedOffsets" : 10000000,
    "spout.offsetCommitPeriodMs" : 30000
}
```

These are the spout recommended defaults from Storm and are currently the defaults provided in the Kafka spout itself. In fact, if you find the recommended defaults work fine for you, then you can omit these settings altogether.

4. Use Case Specific Tuning Suggestions

The following discussion outlines a specific tuning exercise we went through for driving 1 Gbps of traffic through a Metron cluster running with 4 Kafka brokers and 4 Storm Supervisors.

General machine specs:

- 10 GB network cards
- 256 GB memory
- 12 disks
- 32 cores

4.1. Performance Monitoring Tools

Before we get to tuning our cluster, it helps to describe what we might actually want to monitor as well as any potential pain points. Prior to switching over to the new Storm Kafka client, which leverages the new Kafka consumer API under the hood, offsets were stored in ZooKeeper. While the broker hosts are still stored in ZooKeeper, this is no longer true for the offsets which are now stored in Kafka itself. This is a configurable option, and you may switch back to ZooKeeper if you choose, but Metron is currently using the new defaults. With this in mind, there are some useful tools that come with Storm and Kafka that we can use to monitor our topologies.

4.1.1. Tooling

Kafka

- Consumer group offset lag viewer
- There is a GUI tool to make creating, modifying, and generally managing your Kafka topics a bit easier - see kafka-manager
- Console consumer - useful for quickly verifying topic contents

Storm

For more information on the Storm user interface, see Reading and Understanding the Storm UI.

Example: Viewing Kafka Offset Lags

First we need to set up some environment variables.

```
export BROKERLIST  your broker comma-delimated list of host:ports>
export ZOOKEEPER  your zookeeper comma-delimated list of host:ports>
export KAFKA_HOME  kafka home dir>
export METRON_HOME  your metron home>
```

```
export HDP_HOME  your HDP home>
```

If you have Kerberos enabled, set up the security protocol

```
$ cat /tmp/consumergroup.config
security.protocol=SASL_PLAINTEXT
```

Now run the following command for a running topology's consumer group. In this example we are using enrichments.

```
${KAFKA_HOME}/bin/kafka-consumer-groups.sh \
    --command-config=/tmp/consumergroup.config \
    --describe \
    --group enrichments \
    --bootstrap-server $BROKERLIST \
    --new-consumer
```

This will return a table with the following output depicting offsets for all partitions and consumers associated with the specified consumer group.

```
GROUP                          TOPIC              PARTITION  CURRENT-OFFSET
 LOG-END-OFFSET  LAG            OWNER
enrichments                    enrichments        9          29746066
 29746067        1              consumer-2_/xxx.xxx.xxx.xxx
enrichments                    enrichments        3          29754325
 29754326        1              consumer-1_/xxx.xxx.xxx.xxx
enrichments                    enrichments        43         29754331
 29754332        1              consumer-6_/xxx.xxx.xxx.xxx
...
```

Note

You won't see any output until a topology is actually running because the consumer groups only exist while consumers in the spouts are up and running.

The primary column we're concerned with paying attention to is the LAG column, which is the current delta calculation between the current and end offset for the partition. This tells us how close we are to keeping up with incoming data. And, as we found through multiple trials, whether there are any problems with specific consumers getting stuck.

Taking this one step further, it's probably more useful if we can watch the offsets and lags change over time. In order to do this we'll add a "watch" command and set the refresh rate to 10 seconds.

```
watch -n 10 -d ${KAFKA_HOME}/bin/kafka-consumer-groups.sh \
    --command-config=/tmp/consumergroup.config \
    --describe \
    --group enrichments \
    --bootstrap-server $BROKERLIST \
    --new-consumer
```

Every 10 seconds the command will re-run and the screen will be refreshed with new information. The most useful bit is that the watch command will highlight the differences from the current output and the last output screens.

4.1.2. Parser Tuning

We'll be using the Bro sensor in this example.

Note

The parsers and PCAP use a builder utility, as opposed to enrichments and indexing, which use Flux.

We started with a single partition for the inbound Kafka topics and eventually worked our way up to 48 partitions. And we're using the following pending value, as shown below. The default is 'null' which would result in no limit.

storm-bro.config

```
{
    ...
    "topology.max.spout.pending" : 2000
    ...
}
```

And the following default spout settings. Again, this can be omitted entirely since we are using the defaults.

spout-bro.config

```
{
    ...

    "spout.pollTimeoutMs" : 200,
    "spout.maxUncommittedOffsets" : 10000000,
    "spout.offsetCommitPeriodMs" : 30000
}
```

And we ran our Bro parser topology with the following options. We did not need to fully match the number of Kafka partitions with our parallelism in this case, though you could certainly do so if necessary. Notice that we only needed 1 worker.

```
 /usr/metron/0.4.0/bin/start_parser_topology.sh -k $BROKERLIST -z $ZOOKEEPER -
s bro -ksp SASL_PLAINTEXT
    -ot enrichments
    -e ~metron/.storm/storm-bro.config \
    -esc ~/.storm/spout-bro.config \
    -sp 24 \
    -snt 24 \
    -nw 1 \
    -pnt 24 \
    -pp 24 \
```

From the usage docs, here are the options we've used. The full reference can be found here Parsers Readme.

```
 -e,--extra_topology_options (JSON_FILE)          Extra options in the form
                                                  of a JSON file with a map
                                                  for content.
   -esc,--extra_kafka_spout_config (JSON_FILE)    Extra spout config options
                                                  in the form of a JSON file
                                                  with a map for content.
                                                  Possible keys are:
                                                  retryDelayMaxMs,retryDelay
                                                  Multiplier,retryInitialDel
                                                  ayMs,stateUpdateIntervalMs
                                                  ,bufferSizeBytes,fetchMaxW
                                                  ait,fetchSizeBytes,maxOffs
                                                  etBehind,metricsTimeBucket
                                                  SizeInSecs,socketTimeoutMs
     -sp,--spout_p (SPOUT_PARALLELISM_HINT)       Spout Parallelism Hint
       -snt,--spout_num_tasks (NUM_TASKS)         Spout Num Tasks
         -nw,--num_workers (NUM_WORKERS)          Number of Workers
           -pnt,--parser_num_tasks (NUM_TASKS)    Parser Num Tasks
             -pp,--parser_p (PARALLELISM_HINT)    Parser Parallelism Hint
```

4.1.3. Enrichment Tuning

We landed on the same number of partitions for enrichment and indexing as we did for bro - 48.

For configuring Storm, there is a flux file and properties file that we modified. Here are the settings we changed for bro in Flux. +Note that the main Metron-specific option we've changed to accommodate the desired rate of data throughput is max cache size in the join bolts.

More information on Flux can be found here - `http://storm.apache.org/releases/1.0.1/flux.html`

General storm settings

```
topology.workers: 8
topology.acker.executors: 48
topology.max.spout.pending: 2000
```

Spout and Bolt Settings

```
kafkaSpout
    parallelism=48
    session.timeout.ms=29999
    enable.auto.commit=false
    setPollTimeoutMs=200
    setMaxUncommittedOffsets=10000000
    setOffsetCommitPeriodMs=30000
enrichmentSplitBolt
    parallelism=4
enrichmentJoinBolt
    parallelism=8
    withMaxCacheSize=200000
    withMaxTimeRetain=10
threatIntelSplitBolt
```

```
    parallelism=4
 threatIntelJoinBolt
    parallelism=4
    withMaxCacheSize=200000
    withMaxTimeRetain=10
 outputBolt
    parallelism=48
```

4.1.4. Indexing (HDFS) Tuning

There are 48 partitions set for the indexing partition, per the enrichment exercise above.

These are the batch size settings for the Bro index.

```
cat ${METRON_HOME}/config/zookeeper/indexing/bro.json
{
"hdfs" : {
"index": "bro",
    "batchSize": 50,
    "enabled" : true
  }...
}
```

And here are the settings we used for the indexing topology

General Storm Settings

```
topology.workers: 4
topology.acker.executors: 24
topology.max.spout.pending: 2000
```

Spout and Bolt Settings

```
hdfsSyncPolicy
    org.apache.storm.hdfs.bolt.sync.CountSyncPolicy
    constructor arg=100000
hdfsRotationPolicy
    bolt.hdfs.rotation.policy.units=DAYS
    bolt.hdfs.rotation.policy.count=1
kafkaSpout
    parallelism: 24
    session.timeout.ms=29999
    enable.auto.commit=false
    setPollTimeoutMs=200
    setMaxUncommittedOffsets=10000000
    setOffsetCommitPeriodMs=30000
hdfsIndexingBolt
    parallelism: 24
```

4.1.4.1. PCAP Tuning

PCAP is a specialized topology that is a Spout-only topology. Both Kafka topic consumption and HDFS writing is done within a spout to avoid the additional network hop required if using an additional bolt.

General Storm topology properties

```
topology.workers=16
topology.ackers.executors: 0
```

Spout and Bolt Properties

```
kafkaSpout
    parallelism: 128
    poll.timeout.ms=100
    offset.commit.period.ms=30000
    session.timeout.ms=39000
    max.uncommitted.offsets=200000000
    max.poll.interval.ms=10
    max.poll.records=200000
    receive.buffer.bytes=431072
    max.partition.fetch.bytes=10000000
    enable.auto.commit=false
    setMaxUncommittedOffsets=20000000
    setOffsetCommitPeriodMs=30000

writerConfig
    withNumPackets=1265625
    withMaxTimeMS=0
    withReplicationFactor=1
    withSyncEvery=80000
    withHDFSConfig
        io.file.buffer.size=1000000
        dfs.blocksize=1073741824
```

4.2. Issues

Error

```
org.apache.kafka.clients.consumer.CommitFailedException: Commit cannot be
 completed since the group has already rebalanced and assigned
the partitions to another member. This means that the time between subsequent
 calls to poll() was longer than the configured session.timeout.ms,
which typically implies that the poll loop is spending too much time message
 processing. You can address this either by increasing the
session timeout or by reducing the maximum size of batches returned in poll()
 with max.poll.records
```

Suggestions

This implies that the spout hasn't been given enough time between polls before committing the offsets. In other words, the amount of time taken to process the messages is greater than the timeout window. In order to fix this, you can improve message throughput by modifying the options outlined above, increasing the poll timeout, or both.

5. References

- Apache Storm Flux
- What is the "task" in Storm parallelism
- Understanding the Parallelism of a Storm Topology
- Reading and Understanding the Storm UI
- How to choose the number of topics/partitions in a Kafka cluster?