Red Hat

AMQ Streams sizing guidance

Introduction

This document describes an approach to evaluate the size of an AMQ Streams cluster.

Based on information given by our customers about their use cases, the sizing exercise aimed at providing the size of the target AMQ Streams cluster that will be able to sustain that load through the number of brokers/nodes and the size of each of those brokers/nodes.

Method

A running AMQ Cluster receives and delivers messages at the same time, and it's quite difficult to perform performance computation in such a situation.

We propose to divide this behavior into a discrete set of 2 behaviors. At the instant T0, there is no message in the cluster. The consuming applications are idle, and only a producer can be active. Only once the message has been acknowledged by the cluster, which means replicated to at least the in-sync replicas number of nodes, it can be consumed by the consuming applications.

As a result, the method described in this document will consist of 4 steps:

1) Firstly, we look at the provider side only, and we try to evaluate how the cluster could absorb the quantity of information sent to it
2) Secondly, we look at the consumer side only, and we try to evaluate how the cluster could deliver the absorbed messages to the consumers
3) Then, we'll confront the result to best practices
4) Finally, we'll apply a security margin to take potential peaks or deviation into account

Through this document, we will assume a one-to-one mapping between an AMQ Streams broker and a "node"; here a node can be a physical host, a VM or a Container/Pod.

Throughput

The principal information we need, and that will drive the sizing all along the method, is the production throughput. It's the result of the average number of messages published to the cluster per unit of time by the average size of a message:

Example

Parameter name	Value
Average msg per second	20000
Average message size	40KB

Throughput	800 MB/s

A Kafka message can have a header in addition to the payload. When headers are not in use, we can estimate the overhead to 9 bytes per message, whose max size is always 1MB.
In our above example, the messages are already relatively large, and this overhead represents only 0.0002%. In total, the header adds up 9 bytes x 20000 msg/s = 180KB which is negligible.

1. Message ingestion

Any message received by an AMQ Streams node is written to disk. Thus, the ingestion of the throughput by the AMQ Streams cluster is directly dependent on the performance of the storage. As a best practice, it's preferred to use a storage architecture consisting of several "slower" disks put in parallel rather than one single "faster" disk. Technology such as Raid or Jbod can help achieve such a recommended architecture.

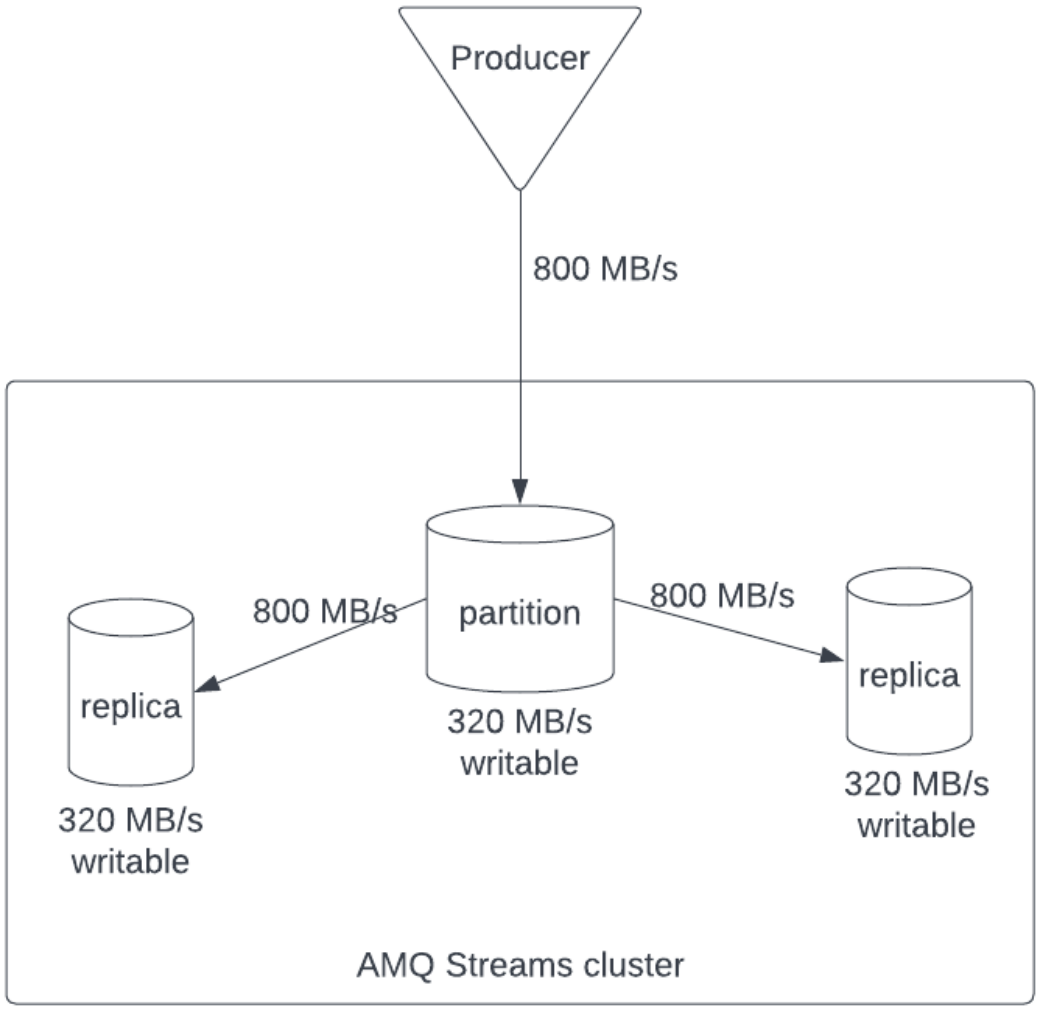
Each received message is stored in a partition. It is written to the local disk of the node hosting the partition leader, but, based on the replication factor, is also written to the disks belonging to the replicas.

The replication factor and the disk throughput characteristic of one node are therefore the 2 elements of information we need to perform the evaluation.

Notice that we should take a small percentage of the disk throughput out of the equation to leave some bandwidth for the operating system activities. Usually, assuming we can dedicate 80% of the disk capacity should be fine.

Example

Parameter name	Value
Replication factor	3
Disk throughput –technical characteristics	400 MB/s
Disk capacity = 80% disk throughput (d)	320 MB/s
Total throughput = throughput x replicas (t)	3* 800 MB/s = 2400 MB/s
Number of nodes	7.5 => 8 nodes



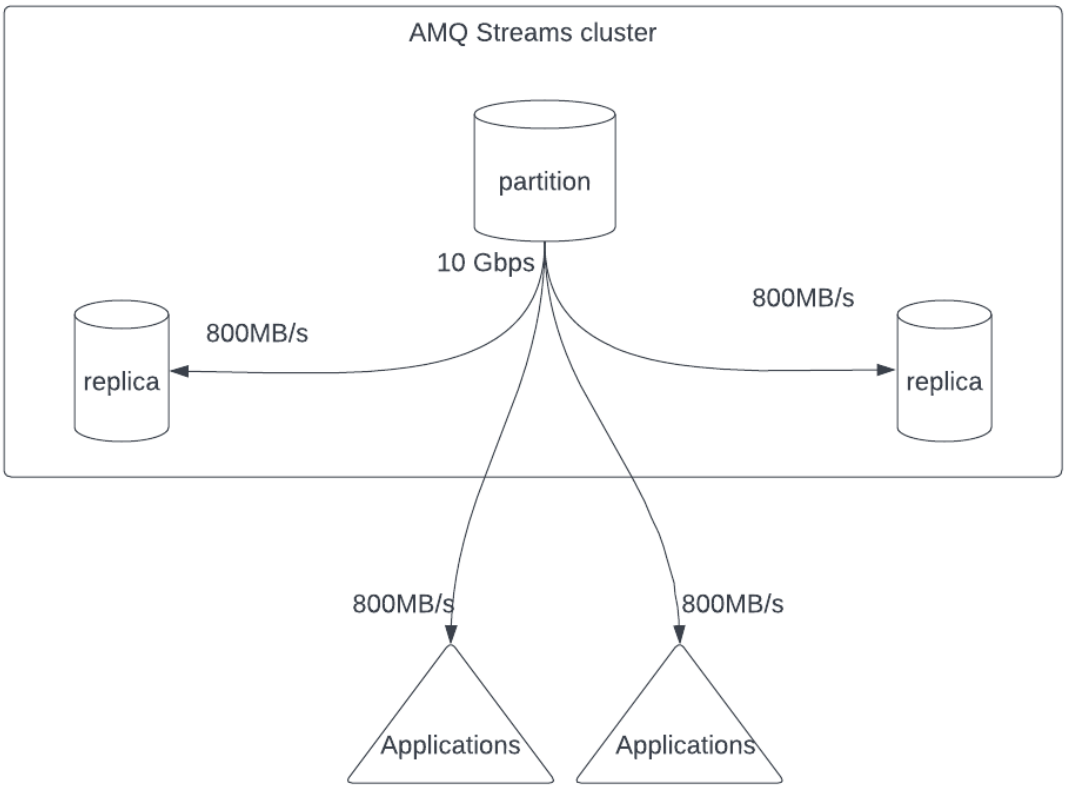
2. Message delivery

Each message stored in a partition is going to be transferred over the network to potentially multiple consumers. Among the consumers, we can identify the nodes hosting the replicas, potentially the nodes of MirrorMaker (we'll leave it aside in the example), as well as consuming applications. To be able to perform the global computation, the 2 critical elements of information that we need are the network adapter bandwidth, the replication factor, potentially the number of MirrorMaker nodes, and the average number of consuming applications (which will become AMQ Streams consumer groups) per topic. We could evaluate this number by dividing the total number of consuming applications by the total number of topics.

Again, we should take a small percentage of the network bandwidth out of the equation to leave it to other operative or administrative tasks.

Example

Parameter name	Value
Number of replicas (r)	2
Adapter speed (technical characteristics)	10 Gb/s
Adapter speed in bytes/s	1250 MB/s
Usable bandwidth (80%)	1 000 MB/s
Total number of topics	1362
Total number of consuming apps	1985
Consuming apps/topic (c)	2
Total number of consumers (r+c)	4
Number of nodes required = throughput *consumers / adapter speed	(4*800MB/s) / (1000MB/s) = 3.2 => 4 nodes



3. Best practices

From the 2 examples listed above, we would conclude that the target cluster would require a minimum of 8 nodes. We also can deduce that the disk is the constraining factor with regard to the network.

Anyway, no matter what the results of the above computation are, we should verify that they comply to the best practices.

We've already assumed one of them, which is to have a one-to-one mapping between brokers and their underlying nodes. Another one tell about having replicas on separate nodes. Thus, for instance, if the resulting computation above had concluded to a cluster of 2 nodes, but our replication factor was 3, we would therefore have adjusted the result to a final number of 3 nodes.

A replication factor of 3, combined with an in-sync replicas quality of service of 2, have also implicitly been considered as best practices in this document.

Also, another best practice is to keep the number of active partition per node to a reasonable value, and to use Cruise Control to guarantee the continuous uniformity of the cluster. Some benchmarks have set a limit for the total number of partition per broker to 4000. However, bear

in mind that, in case of the crash of a node, all the partition leaders hosted on that node need to be moved to another node, and this operation, unfortunately sequential by design, implies a 10ms delay per leader. For that reason, we like to try to keep the total number of partitions (active and inactive) by node under a maximum of 1500, which translates to 500 active leaders and 2x500 passive replicas.

4. Tolerance

The above figures were computed with some assumptions. Those assumptions now need to be reviewed, and the proper adjustments would need to be made, should those assumptions not be always met.

1. Projection

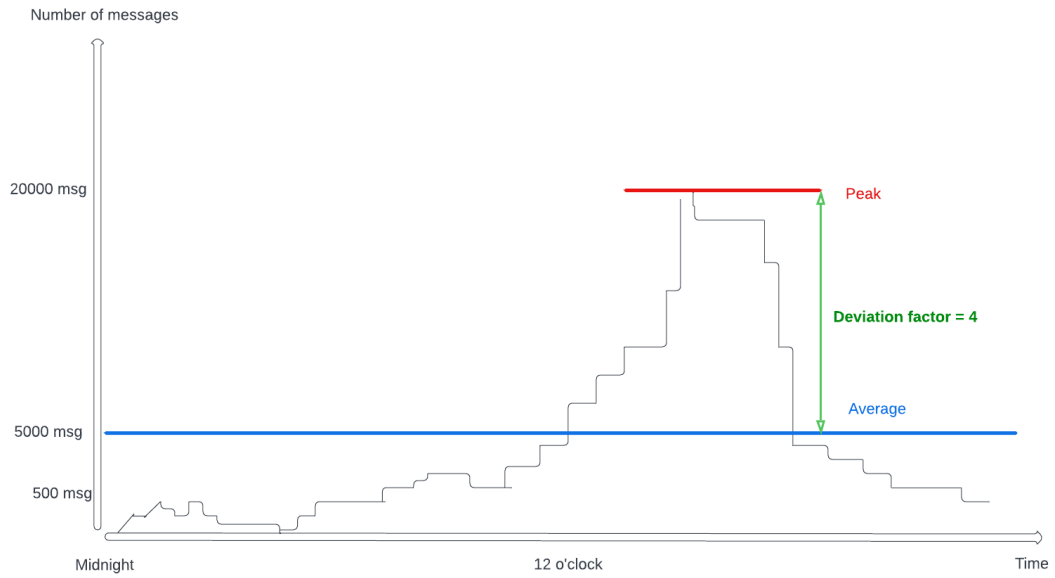
First, we need to recall that the computation leads to the MINIMUM size of the cluster to sustain the load. Therefore, we might need to add a bit of resources to support a slight increasing of the load, so that we won't have to immediately add new nodes as soon as some more messages arrive.

2. Fault tolerance

We might also provide us with the ability to sustain the full workload in the case of an incident. For example, if we want to have a service continuity capability in case of the crash of a couple of nodes, then we simply need to add 2 nodes to the total

3. Deviation

The input we used for the throughput was based on a number of messages per second. The way this number was computed is of importance. Indeed, if it results from the total number of messages per day divided by 24*3600 seconds, it would lead to some performance issue at peak time. The easiest way to cope with a deviation is to try to evaluate it a multiplication factor, like shown on the below diagram.



4. Uniformity

The computation made above assume an AMQ Streams cluster perfectly uniformly distributed. Cruise Control can help you achieve that, but from time to time, upon evolution or after incidents, there are situations where the cluster is unbalanced. For instance, a node might have more active partitions than the others, consumers might be reading from a node than from the others...

This also need to be taken into account in the application of a security margin to the above figures.

Physical characteristics of the nodes

Memory

An AMQ Streams broker should behave correctly with a memory size of 6 GB but a much higher value is recommended.

The reason why is that the data is actually written to the PageCache, whose management (disk operations) is delegated to the Operating System. Retrieving data from the PageCache is much faster than retrieving it directly from the disk.

Why is a large page cache required?

Imagine a situation where 2 consumer groups, 'A' and 'B', both read some data from a single partition. Suddenly, consumer 'A' crashes. Consumer 'B' continue to read data from the partition. When consumer 'A' is back online, its index is far behind the index of consumer 'B'. It's now

actually trying to keep up and recover from "historical data". Forcing the broker to query recent data from disk for this consumer would greatly affect the general performance for all consumers; however, the performance could be maintained as long as the data is still in the PageCache. This is the reason why we don't recommend setting a production AMQ Streams with nodes smaller than nodes of 32GB RAM (sometimes even going up to 64GB).

We generally recommend AMQStreams nodes of 32 GB RAM when possible. This can go up to 64 GB in particular cases.

Maximum lagging time

The maximum lagging time is the time limit behind which an issue on a consumer would force the cluster to gt data back from the disk, slowing down the whole process.

It's computed by:

```
<Maximum lag> = <Usable cache memory> / <inbound throughput>
```

For example, with our example total throughput of 800 MB/s, and evaluating the available cache to 24G (32G - 6G for the broker -2G for the OS):

lag = 24000 MB / 800 MB/s = 30 seconds

With an Openshift-based deployment, the above result would mean that the maximum affordable recovery time for the pod hosting a consuming application is 30 seconds, which might be too close to the pod's startup time, and justify upgrading the nodes to 48 or 64 GB RAM.

CPU

AMQ Streams does not make an intensive use of the CPU. The only 2 activities that required higher CPU consumption is encryption (when the producers/consumers connect to the cluster through an SSL/TLS connection) and file compaction.

AMQ Streams is rather a system designed with parallelization in mind, and thus a physical architecture relying on a larger number of "smaller" CPU will perform much better tahn with a smaller number of "faster" CPU. Also, bear in mind that the number of partitions per topic is directly linked to the number of vCPU. With 10vCPU, it's usually discouraged to go further than 20 partitions due to the increasing "competition" for resources (and of course something from 1 to 10 would give the best efficiency).

Generally, we start with a production setup involving 8 vCPU, and sometimes go up to 12 depending on the circumstances number of disk in parallel, use of TLS…

Physical storage

The physical storage capacity directly depends on the retention. For example, if the retention is 5 days (432000 seconds) for the throughput taken as an example and the replication factor of 3, the total storage capacity required will be:

```
<Storage> = <Retention> * <Inbound rate> * <Replication factor>
```

432000 * 800 * 3 = 1036800000 MB = 989 Terabytes

Message overhead

The above number is the quantity of the message for the pure data. We already took into account the overhead induced by the message header. However, we still need to take into account 2 more factors:

1) Partitions are indexed and the index information are also written to disk
   This can be somehow hard to evaluate. The default configuration divides the storage file system into segment of 1GB, to which are associated 2 indexes of 10MB each
   This leads to a minimum overhead of 2%
2) The retention policy doesn't apply to the last segment of the commit log which is still active. A configuration with large log segments on partitions having a small amount of data can lead to the data being retained much longer than the retention time for this last segment.

For the final evaluation, we suggest adding up a 3 to 5% overhead over the total.

Storage distribution

The total quantity is the total quantity of storage for the entire cluster, across all nodes. On a cluster of 10 nodes, we can't however simply allocate 1/10 of that quantity to each node, and this is for at least 2 reasons:

1. The distribution might not be uniform all the time across the cluster.
2. We need to take fault tolerance into consideration, which means the ability to operate on a full workload with a reduced number of nodes, that might therefore have to hold the complete retention (if the retention is shorter than the longest expected node recovery time).

Additional information

KafkaConnect

DRAFT - Section to be reviewed and not complete

- How to size KafkaConnect Sources and Sink ???

- Size KafkaConnect in the context of MirrorMaker 2 ???

KafkaConnect is the technology used to get data into the AMQ Streams cluster (or possibly to even deliver data to external systems from the cluster). The technology is based on connectors (that are jar files representing the implementation of the data importing logic).

KafkaConnect behave like a small AMQ Streams cluster, in such that a KafkaConnect component can be spread over multiple nodes for high availability. Best practices therefore speak about a minimum number of 2 nodes per KafkaConnect cluster.

Evaluating the resources required for KafkaConnect component correctly is crucial because the technology serves as the basis for cross data center replication, possibly enabling a dual, cross data center active/active AMQ Streams cluster.

A KafkaConnect component can host more than one connector.

In terms of memory, we usually evaluate it to be 2GB for the component itself + 1GB per connector. However, each connector can fork multiple tasks (parallel threads), and this ability can greatly affect the sizing of a KafkaConnect component.

In theory, Sink connectors (connectors reading from topics) should totalize a number of parallel tasks equal to the total number of partition they read from. Let's imagine a cluster of 1000 topics, each topics having 3 partitions (in average). The near real time replication of data would require a total number of threads of 3000. Though the technical limit in the number of threads a Linux system can run is very high (like 16000 to 64000), the competition for the CPU and the memory usage might not render the expected performance.

First, the real virtualization depends on the hardware. A 2 sockets, 6 cores per socket, bi-threaded server would be able to run 24 real parallel processes that the tasks will share. For best performance, we need to keep the number of task per CPU core under **a certain limit???**

Then we can tune up the memory. The lagging/PageCache problematic doesn't come into play in the context of KafkaConnect as there are no disk I/O operations. However, a high number of threads can require adding some memory.

A Linux process requires 8MB in the Operating System thread stack, and each thread within an JVM requires 1MB from the Heap stack by default. 1 GB of Heap will therefore allow 1000 java threads ???

Configuration parameters that influence performance

Num io threads = vCPU x num disks in parallel

Num network threads = replication factor + 1

Zookeeper and cross-datacenter architecture

An AMQ Streams cluster has a technical requirement of a maximum 5ms latency between the nodes of its control plane. For that reason, the deployment of an AMQ Streams cluster stretched across 2 distant data centers is usually not applicable. For a cross data centers scenario, it's usually required to deploy 2 distinct equivalent AMQ Streams clusters, and to replicate the data between them to achieve either an active/passive or an active/active topology.

→ Sizing MirrorMaker based on the total number of topics to replicate ???

Helpers and support

To help you make the computation, you can use the below tool:

https://kafkasizing.azurewebsites.net/size