KIP-81: Bound Fetch memory usage in the consumer

- Status
- Motivation
- Public Interfaces
- Proposed Changes
- Compatibility, Deprecation, and Migration Plan
- Rejected Alternatives
  - Limit sending FetchRequests once a specific number of in-flight requests is reached:
  - Explicit disposal of memory by the user:

Status

Current state: Accepted

Discussion thread: *HERE*

JIRA: Unable to render Jira issues macro, execution error.

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

This work was done in collaboration with Edoardo Comar

Motivation

With KIP-74, we now have a good way to limit the size of Fetch responses, but it may still be difficult for users to control overall memory since the consumer will send fetches in parallel to all the brokers which own partitions that the client is subscribed to. Currently we have:

-`max.fetch.bytes`: This enables to control how much data will be returned by the broker for one fetch

-`max.partition.fetch.bytes`: This enables to control how much data per partition will be returned by the broker

None of these settings take into account that the consumer will be sending requests to multiple brokers in parallel so in practice the memory usage is as stated in KIP-74: `min(num brokers * max.fetch.bytes,  max.partition.fetch.bytes * num_partitions)`

To give users simpler control, it makes sense to add a new setting to properly limit the memory used by Fetch responses in the consumer in a similar fashion than what we already have on the producer.

Public Interfaces

The following option will be added for to the Consumer configuration):

1. `buffer.memory: Type: Long, Priority: High, Default: 100MB`

   The total bytes of memory the consumer can use to buffer records received from the server and waiting to be processed (decompressed and deserialized).

   This setting slightly differs from the total memory the consumer will use because some additional memory will be used for decompression (if compression is enabled), deserialization as well as for maintaining in-flight requests.

   Note that this setting must be at least as big as max.fetch.bytes.

Alongside, we will set the priority of `max.partition.fetch.bytes` to Low.

Proposed Changes

This KIP reuses the MemoryPool implementation from KIP-72 (SimpleMemoryPool).

We propose to change the constructors of Node and KafkaChannel:

```
public Node(int id, String host, int port, String rack, boolean priority) {}

public KafkaChannel(String id, TransportLayer transportLayer, Authenticator authenticator, int
maxReceiveSize, boolean priority) {}
```

The new 'priority' argument will be used to mark connections to the Coordinator. Currently those are only identified by using a large id, marking them explicitly will make their detection simpler.

1) At startup, the consumer will initialize a MemoryPool with the size specified by buffer.memory. This pool enables to track how much memory the consumer is using for received messages. The memory is not pre-allocated but only used as needed.

2) In `Selector.pollSelectionKeys()`, before reading from sockets, the code will check there is still available space left in the MemoryPool.

3) here we have 2 options:

3.a If there is space in the Pool:

`pollSelectionKeys()` will read from sockets and store messages in the MemoryPool. The fairness mechanism introduced in KIP-74 ensures the consumer sends FetchRequests to all partitions it is subscribed to in round robin fashion so we don't need to introduce a new mechanism for fairness when reading from the sockets.

3.b If there is no space in the Pool:

`pollSelectionKeys()` will *only* read messages coming from the Coordinator (identified using the priority flag) and those will be allocated outside of the Pool. Such messages should be small enough to not use too much extra memory. This will prevent any Coordinator starvation (For example, if we send a bunch of pre-fetches right before returning to the user, these fetches might return before the next call to poll(), in which case we might not have enough memory to receive heartbeats, which would block us from sending additional heartbeats until the next call to poll()). When the Pool is not full, messages from the Coordinator are handled normally (3.a).

4) Once messages are decompressed (if compression if enabled), deserialized and queued up to be returned to the user (in `Fetcher.parseFetchedData()`), the memory used in the MemoryPool is released.

The `ConsumerRecords` objects are not stored in the MemoryPool but in the Java heap. The MemoryPool is only used to store bytes received from the network.

The consumer (Fetcher) delays decompression until the records are returned to the user, but because of `max.poll.records`, it may end up holding onto the decompressed data from a single partition for a few iterations. Therefore `buffer.memory` is not a hard bound constraint on the consumer's memory usage as mentioned in the setting's description.

Similarly to KIP-72, metrics about the MemoryPool (usage, free space, etc) will be exposed by the Consumer:

- `memory-pool-free`: The amount of free memory in the MemoryPool
- `memory-pool-used`: The amount of used memory in the MemoryPool
- `memory-pool-avg-depleted-percent`: The percentage of time when the MemoryPool is full
- `memory-pool-depleted-time-total`: The duration when the MemoryPool is full

Compatibility, Deprecation, and Migration Plan

This KIP should be transparent to users not interested in setting this new configuration. Users wanting to take advantage of this new feature will just need to add this new settings to their consumer's properties.

Rejected Alternatives

- ## Limit sending FetchRequests once a specific number of in-flight requests is reached:

  While this was initially considered, this method will result in a loss of performance. Throttling FetchRequests means that when memory is freed, the consumer first has to send a new FetchRequest and wait from the broker response before it can consume new messages.

- ## Explicit disposal of memory by the user:

  It was suggested to have an explicit call to a dispose() method to free up memory in the MemoryPool. In addition of breaking the API, this seems confusing for Java