# KIP-72: Allow putting a bound on memory consumed by Incoming requests



## Status

**Current State**: *Draft*

**Discussion Thread**: link

**JIRA**: KAFKA-4602

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

## Motivation

Kafka currently supports setting an upper bound on the number of requests allowed into the (incoming) request queue. This is an indirect way of controlling memory consumption and has a few drawbacks:

1. An administrator needs to estimate the average request size in order to provide a meaningful size limit.
2. This size limit may need to be periodically updated as the workload changes.
3. The server is still susceptible to a simultaneous batch of large requests exhausting the JVM memory (causing an OOM exception).

The third scenario actually occurred a few times at LinkedIn - a sudden spike of very large request batches (1000s requests each) from a Hadoop job caused OOM exceptions on a production cluster.

This KIP proposes allowing an administrator to specify a memory limit in bytes, which resolves the above problems.

Also, the code/facilities introduced by this KIP might prove useful for similar problems on the client side - like producer/consumer memory bounds.

## Public Interfaces

This KIP introduces a new server configuration parameter, **`queued.max.request.bytes`**, that would specify a limit on the volume of requests that can be held in memory. This configuration parameter will co-exist with the existing **`queued.max.requests`** (the code will respect both bounds and will not pick up new requests when either is hit).

Beyond the proposed new configuration key this KIP makes no changes to client or server public APIs.

## New Sensors

1. MemoryPoolAvgDepletedPercent - percent of the time request were not being read out of socket due to lack of memory
2. MemoryPoolAvailable - number of bytes available in the pool
3. MemoryPoolUsed - number of bytes currently allocated out of the pool and still not returned

## Proposed Changes

### Memory Pools

a new MemoryPool interface would be introduced into the kafka codebase:

**MemoryPool.java**

```
/**
 * A common memory pool interface for non-blocking pools.
 * Every buffer returned from tryAllocate() must always be release()ed.
 */
public interface MemoryPool {
    /**
     * Tries to acquire a ByteBuffer of the specified size
     * @param sizeBytes size required
     * @return a ByteBuffer (which later needs to be release()ed), or null
if no memory available.
     *         the buffer will be of the exact size requested, even if
backed by a larger chunk of memory
     */
    ByteBuffer tryAllocate(int sizeBytes);
    /**
     * Returns a previously allocated buffer to the pool.
     * @param previouslyAllocated a buffer previously returned from
tryAllocate()
     */
    void release(ByteBuffer previouslyAllocated);
    /**
     * Returns the total size of this pool
     * @return total size, in bytes
     */
    long getSize();
    /**
     * Returns the amount of memory available for allocation by this pool.
     * NOTE: result mey be negative (pools may over allocate to avoid
starvation issues)
     * @return
     */
    long getAvailableMemory();
    /**
     * Returns true if the pool cannot currently allocate any more buffers
     * - meaning total outstanding buffers meets or exceeds pool size and
     * some would need to be released before further allocations are
possible.
     *
     * This is equivalent to getAvailableMemory() <= 0
     * @return true if out of memory
     */
    boolean isOutOfMemory();
}
```

1. the pool is non-blocking, so network threads would not be blocked waiting for memory and could make progress elsewhere.
2. SocketServer would instantiate and hold a memory pool, which Processor threads would try to allocate memory from when reading requests out of sockets (by passing the pool to instances of NetworkReceive that they create).
3. NetworkReceive.readFromReadableChannel() would be modified to try allocating memory (it is already written in a way that reading may involve multiple repeated calls to readFromReadableChannel(), so not a big change to behavior)
4. memory would be released at the end of request processing (in KafkaRequestHandler.run()), and also in case of disconnection mid request-building in KafkaChannel.close()

5. As the pool would allow any size request if it has *any* capacity available, the actual memory bound is **`queued.max.request.bytes` + `socket.request.max.bytes`**. The up-side is no issues with large requests getting starved out

### Request throttling by way of channel muting

when memory is unavailable, Selector would mute incoming channels so as not to get into a tight-loop where read-ready keys are returned by poll() but cannot actually be read from (because no memory). care must be taken when muting/unmuting because:

1. SSL transports cannot be muted mid-handshake (I believe this is more for implementation simplicity in kafka than a fundamental limitation, but not one this KIP should address)
2. muting a channel that has already been allocated memory to read into might result in a deadlock.
3. muting channels is also kafka's way of ensuring only a single request is processed at a time from the same channel. this guarantee must be preserved.
4. SSLTransportLayer has intermediate buffers where data may get "stuck" (due to no memory) and yet the underlying socket may be done and so will not show up on further select() calls.

so, the following changes are proposed in support of channel muting under memory pressure:

1. TransportLayer.isInMutableState() would be introduced to account for transports that cannot currently be muted (currently only SSL during handshake)
2. TransportLayer.hasBytesBuffered() would be introduced to account for transports that have unread data in any intermediate buffers (currently only possible for ssl transport)
3. KafkaChannel.isInMutableState() would be introduced to account for channels with already allocated memory or channels who's underlying transport is not in a mutable state.
4. Selector.poll() would mute all channels that are in a mutable state if the server has no memory to accept any further requests - this would prevent their keys from being returned by the underlying select() call and thus prevent a tight loop in SocketServer.Processor.run()
5. when memory becomes available again (in some subsequent Selector.poll() call) any channels previously muted **except those muted for single-request-procedding reasons** (see #3 above) will be unmuted. this would require maintainign a set of channels explicitely muted (so they would not be unmuted when memory is available) in Selector
6. channels who's underlying transports have yet-unread data buffered must be accounted for to prevent the case of a stale tail of data getting stuck in said buffers.

### Caveats

1. concerns have been raised about the possibility of starvation in Selector.pollSelectionKeys() - in case the order of keys in Set<SelectionKey> selectionKeys is deterministic and memory is tight, sockets consistently at the beginning of the set get better treatment then those at the end of the iteration order. to overcome this code has been put in place to shuffle the selection keys and handle them in random order ONLY IF MEMORY IS TIGHT (so if the previous allocation call failed). this avoids the overhead of the shuffle when memory is not an issue.

## Compatibility, Deprecation, and Migration Plan

There are a few approaches w.r.t migration. **The current preference is to go with the third option.**

1. **`queued.max.requests`** is deprecated/removed in favor of **`queued.max.request.bytes`**. In this case, the conversion of existing configurations could use **`queued.max.request.bytes = queued.max.requests * socket.request.max.bytes`** (which is conservative, but "safe")
2. **`queued.max.requests`** is supported as an alternative to **`queued.max.request.bytes`** (either-or), in which case no migration is required. A default value of 0 could be used to disable the feature (by default) and runtime code would pick a queue implementation depending on which configuration parameter is provided.
3. **`queued.max.requests`** is supported in addition **`queued.max.request.bytes`** (both respected at the same time). In this case a default value of **`queued.max.request.bytes = -1`** would maintain backwards compatible behavior.

The current naming scheme of **`queued.max.requests`** (and the proposed **`queued.max.request.bytes`**) may be a bit opaque. Perhaps using **requestQueue.max.requests** and **requestQueue.max.bytes** would more clearly convey the meaning to users (indicating that these settings deal with the request queue specifically, and not some other). The current `queued.max.requests` configuration can be retained for a few more releases for backwards compatibility.

## Configuration Validation

**queued.max.request.bytes** must be larger than **socket.request.max.bytes** (in other words, memory pool must be large enough to accommodate the largest single request possible), or <=0 (if disabled). the default would be -1.

## Test Plan

- A unit test was written to validate the behavior of the memory pool
- A unit test that validates correct behavior of RequestChannel under capacity bounds would need to be written.
- A micro-benchmark for determining the performance of the pool would need to be written
- Stress testing a broker (heavy producer load of varying request sizes) to verify that the memory limit is honored.
- Benchmarking producer and consumer throughput before/after the change to prove that ingress/egress performance remains acceptable.
- Testing of SSL connections (both inter-broker and client-broker) since implementation bugs may affect ssl

# Rejected Alternatives

Here are some alternatives that we have discussed (at LinkedIn):

1. Reducing producer max batch size: this is harmful to throughput (and is also more complicated to maintain from an administrator's standpoint than simply sizing the broker itself). This is more of a workaround than a fix
2. Reducing producer max request size: same issues as above.
3. Limiting the number of connected clients: same issues as above
4. Reducing `queued.max.requests` in the broker: Although this will conservatively size the queue it can be detrimental to throughput in the average case.
5. controlling the volume of requests enqueued in `RequestChannel.requestQueue` (would not suffice as no bound on memory read from actual sockets)

# Implementation Concerns

1. the order of selection keys returned from a selector.poll call is undefined. in case the actual implementation uses a fixed order (say by increasing handle id?) and under prolonged memory pressure (so never enough memory to service all requests) this may lead to starvation of sockets that are always at the end of the iteration order. to overcome this the code shuffles the selection keys if memory is low.
2. a strict pool (which adheres to its max size completely) will cause starvation of large requests under memory pressure (as they would never be able to allocate if there is a stream of small requests). to avoid this the pool implementation will allocate the requested amount of memory if it has *any* memory available (so if pool has 1 free byte and 1 MB is requested, 1MB will be returned and the number of available bytes in the pool will be negative). this means the actual bound on number of bytes outstanding is **queued.max.request.bytes** + **socket.request.max.bytes** - 1 (socket.request.max.bytes representing the largest single request possible)

# State of Implementation

an implementation is available - https://github.com/radai-rosenblatt/kafka/tree/broker-memory-pool-with-muting