Large Message Handling with Kafka: Chunking vs. External Store

By Adam Kotwasinski, Senior Software Development Engineer, Workday

Background

Multiple services in Workday use Kafka as a messaging bus (https://kafka.apache.org/uses), streaming various types of data. Unfortunately, Kafka imposes a limit on the size of the payload that can be sent to the broker (compared to RabbitMQ, that does not have such a limit). Removing the limit would allow the business layer code to solve the business problem, without Kafka-specific behaviours leaking into application code.

If the message is larger than the value accepted by the broker, the Kafka producer returns this exception:

```
The message is 58105770 bytes when serialized which is larger than the
maximum request size you have configured with the max.request.size
configuration.
```

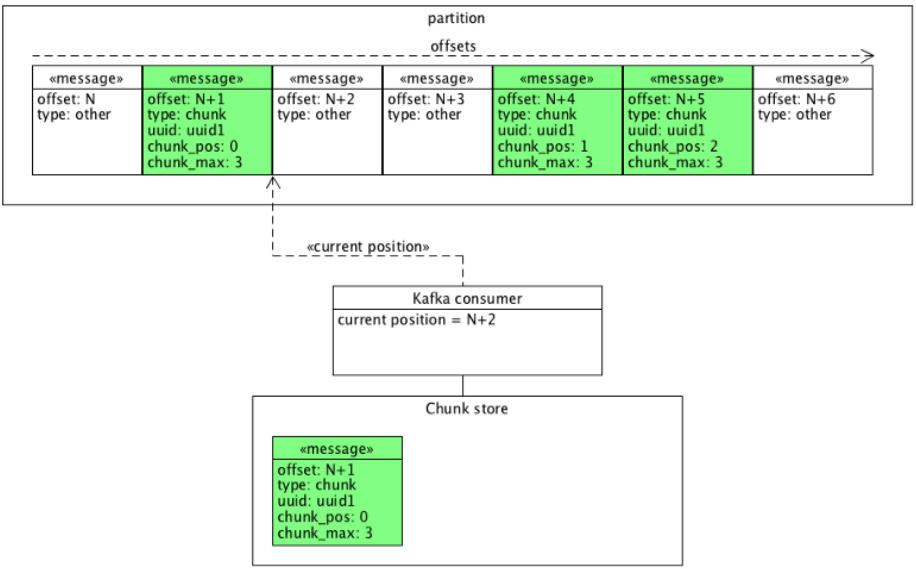
Implementation 1—Chunking

Large payloads can be split into multiple smaller chunks that can be accepted by brokers.

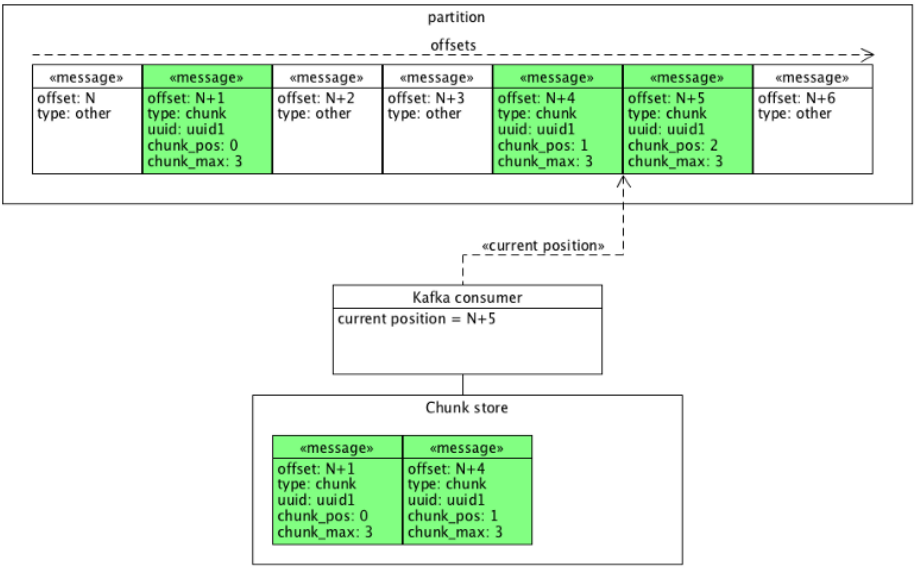
The chunks can be stored in Kafka in the same way as ordinary (not-chunked) messages. The only difference is that the consumer would need to keep the chunks and combine them into the real message when all chunks have been collected.

The chunks in the Kafka log can be interwoven with ordinary messages.

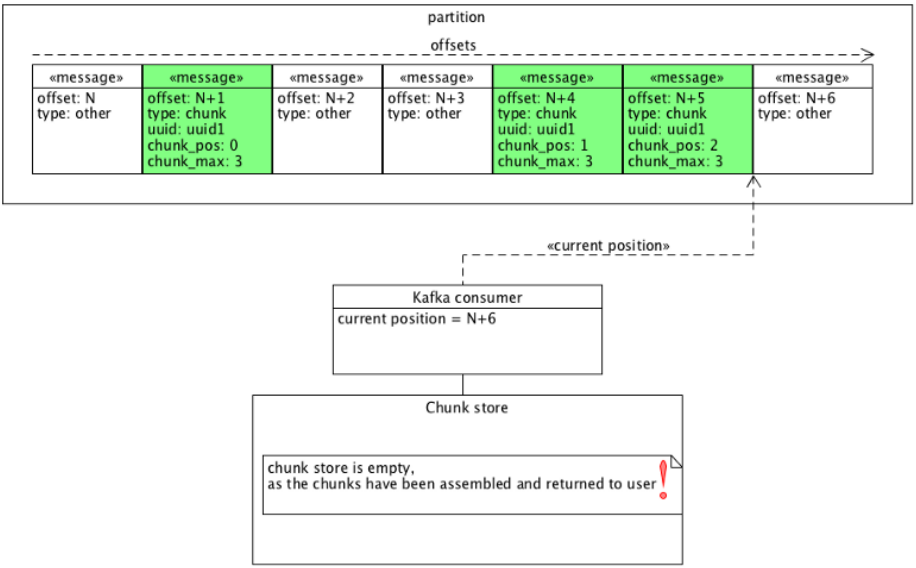
For example, for a message chunked into three chunks, the consumer could have already consumed one of the chunks:



After the consumer receives more messages, we can see that the second chunk has been received:



After the third and final chunk is received, the message can finally be combined and returned to the end user, and the cache can be cleaned up:



The chunk size is derived from the Kafka producer configuration (max.request.size).

Each of the chunks get sent together with this metadata:

- Message uuid
- The chunk's position in the payload
- The total number of chunks

Chunk payload format

```
[HEADER][type:1b=1][chunk_uuid:36b][chunk_pos:4b][chunk_max:4b][payload:*]
where
    HEADER := "PRF_wdkafcli_header_".getBytes(UTF-8)
    type := 1
    chunk_uuid := chunk's uuid
    chunk_pos := position of chunk in original payload, values 0..chunk_max - 1
    chunk_max := number of chunks payload := chunk payload
```

Sending messages

The main classes for this functionality are message producer and message chunker.

The code generating the chunking payloads (together with the above headers) is relatively simple. Given the payload and maximum allowed size, we split it into roughly payload.size/chunkSize number of chunks:

```
private List<byte[]> generateChunks(final byte[] payload, final int chunkSize) {
    final byte[] uuid =
UUID.randomUUID().toString().getBytes(EnvelopeConstants.STRING_CHARSET);
    final int payloadSize = payload.length;
    final int chunkCount = (payloadSize / chunkSize) + (0 == payloadSize % chunkSize
? 0 : 1);

    int start = 0;
    int index = 0;
    final List<byte[]> subarrays = new ArrayList<>(chunkCount);
    while (start < payloadSize) {
        final int end = (start + chunkSize) < payloadSize ? start + chunkSize :
payloadSize;
        final int size = end - start;
        final byte[] chunk = new byte[EnvelopeConstants.CHUNK_ENVELOPE_SIZE + size];
        ByteBuffer bb = ByteBuffer.wrap(chunk);
        bb.put(EnvelopeConstants.ENVELOPE_HEADER);
        bb.put(EnvelopeConstants.CHUNK_ENVELOPE_VERSION);
        bb.put(uuid);
        bb.putInt(index);
        bb.putInt(chunkCount);
        bb = null;

        System.arraycopy(payload, start, chunk,
EnvelopeConstants.CHUNK_ENVELOPE_SIZE, size);
        subarrays.add(chunk);

        index++;
        start = end;
    }

    return subarrays;
}
```

The offset returned by send methods must point to the lowest offset assigned to all of the chunks. Any other result would make it impossible to seek to the first chunk:

```
private static final Comparator<RecordMetadata> OFFSET_COMPARATOR = (a, b) ->
Long.compare(a.offset(), b.offset());

----------------
```

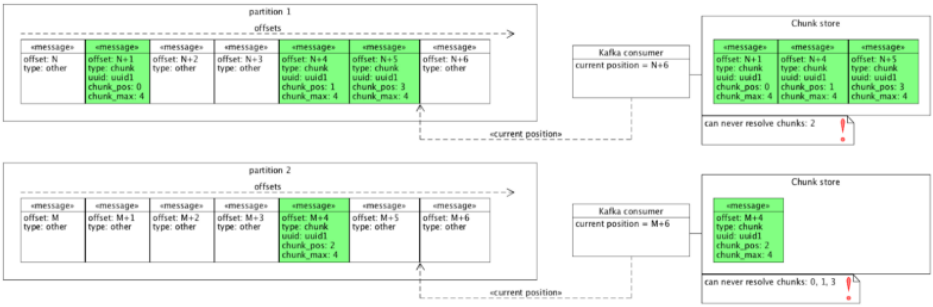
```
final List<Future<RecordMetadata>> futures = new ArrayList<>(records.size());
for (final ProducerAction record : records) {
    final Future<RecordMetadata> future = this.producer.send(record.getRecord());
    futures.add(future);
}

final List<RecordMetadata> sendResults = new ArrayList<>(records.size());
for (final Future<RecordMetadata> future : futures) {
    final RecordMetadata recordMetadata = future.get(timeout, timeUnit);
    sendResults.add(recordMetadata);
}

final RecordMetadata lowestOffsetMetadata =
sendResults.stream().min(OFFSET_COMPARATOR).get();
return new SendStatusImpl(message, destination, lowestOffsetMetadata);
```

The Kafka producer API allows the user to compute the message's partition from the message key. The result is based on the number of partitions currently hosted in the cluster (DefaultPartitioner in Kafka 1.0). Right now, the implementation sends the chunks to partition 0 if the partition was not specified. In future versions, we might want to change it by reading the partitions returned by the first chunk send operation, and sending other chunks to the same partition.

Otherwise, we could end up with a situation where consumers consuming from a single partition would never manage to receive a full message, as chunks would be spread across partitions:



Receiving messages

The main classes for this functionality are abstract consumer and message chunker.

When a Kafka message containing a chunk is received, it is kept locally and not returned to the user (as one would see no benefit in getting just a part of the payload). Only when all chunks have been collected are they assembled together into a single message and returned to the user.

Cleaning up this store might be needed after offset-changing operations, as seeking into a position between chunks could have returned a full message, when the user intended to seek to a later point (after the head of the message). Example: for a six chunk message, we already have received chunks 1, 2, and 3. After seeking to position three again, we'd have consumed chunks: 3 (again), 4, 5 and 6 (the new ones). This means that all chunks have been received, while chunks 1 and 2 were received before the seek operation, and should have not been made available to the user. In the previous diagrams, the offset to seek for would be `N+1`.

Currently the chunk store is implemented as a Java in-memory map.

Group management poses a challenge for chunk-based solutions. It is possible that a consumer group could get rebalanced, while some of the chunks have already been received by the old consumer. So, we would end with one (old) consumer storing beginning chunks, and the other (new) consumer receiving the remaining ones. The potential solutions for this are:

- Alternative 1—new consumer—seeking on the rebalance event if the chunk we have received was not the first one.
- Alternative 2—old consumer—starting a parallel non-group assigned consumer that would be responsible for fetching the remaining chunks (kind of "finishing mechanism").
- Alternative 3—sharing chunks between consumers—would be a distributed cache solution, and an adventure in itself.

Implementation 2—External Store

Instead of sending a large payload over Kafka, we could try a different approach—store the real payload in an external data store, and only transfer the pointer to that data. The receiver would then recognize this type of pointer payload, transparently read the data from the external store, and provide it to the end user.

Pointer envelope payload format

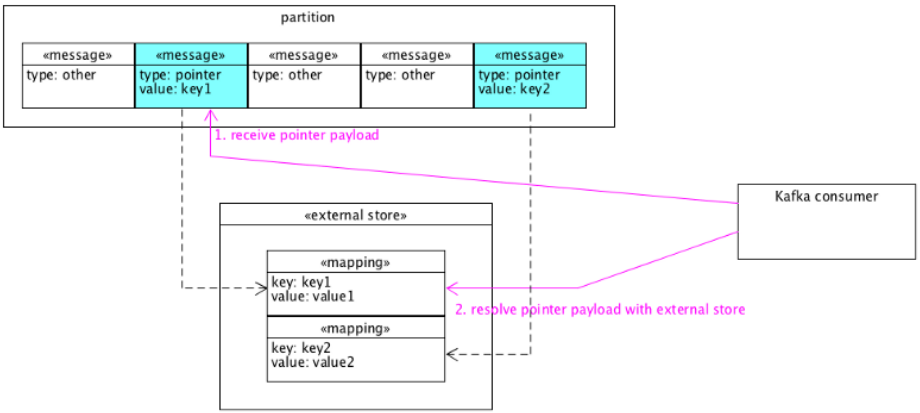
```
[HEADER][type:1b=2][payload:*]
where
    HEADER := "PRF_wdkafcli_header_".getBytes(UTF-8)
    type := 2
    payload := payload to be passed to external store
```

Generic IoC

The sending part would then be responsible for saving the payloads and sending the pointer through Kafka:

- Save payload to external store (via `write` method implemented by provider)
- Generate the envelope that wraps the pointer payload, using the same Kafka metadata
  as the original (topic, partition, key)
- Send it via Kafka

So long story short, the messages in Kafka would refer to the entries in the external store:



The external store would need to implement three methods: write, read, and rollback (in case of failed transactions):

```
/**
 * Save message's payload in external data store
 * Returned value is then sent over Kafka and forwarded to external data source, so
the real payload can be recovered
 * @return pointer pointer payload that uniquely identifies stored payload in
external data source
 */
byte[] write(String uuid, byte[] payload, Destination kafkaDestination);

/**
 * Recovers original Kafka payload
 * @param pointerPayload pointer to data in externdal data store
 * @return original payload stored by external payload sink
 */
byte[] read(byte[] pointerPayload);

/**
 * Rollback saving external payload
 * @param pointerPayload pointer payload generated during write invocation
 */
void rollback(byte[] pointerPayload);
```

The main class for this functionality is message chunker:

```
final byte[] pointerPayload = externalPayloadSink.write(messageUuid, record.value(),
destination); // storing real payload
final ProducerRecord<String, byte[]> envelope = buildPointerRecord(record,
pointerPayload);

----------------

private ProducerRecord<String, byte[]> buildPointerRecord(final
ProducerRecord<String, byte[]> record, final byte[] payload) {
    final byte[] rp = new byte[EnvelopeConstants.POINTER_ENVELOPE_SIZE +
payload.length];
    ByteBuffer bb = ByteBuffer.wrap(rp, 0, EnvelopeConstants.POINTER_ENVELOPE_SIZE);
    bb.put(EnvelopeConstants.ENVELOPE_HEADER);
    bb.put(EnvelopeConstants.POINTER_ENVELOPE_HEADER);
    bb = null;
    System.arraycopy(payload, 0, rp, EnvelopeConstants.POINTER_ENVELOPE_SIZE,
payload.length);
    return new ProducerRecord<String, byte[]>(record.topic(), record.partition(),
record.key(), rp);
}
```

The receiving part would then be responsible for recovering the payloads:

- Checking if the message received is really a pointer payload (done by checking the payload header & version).
- Extracting the real pointer payload (generated by the external store).
- Passing the pointer payload to the external store via `read` method to provide the real payload.
- Returning the result to the user, with the same metadata (topic, partitions, offset, key) as the originally received message.

```
final Optional<byte[]> externalStoreKey = extractKeyFromEnvelope(source);
if (externalStoreKey.isPresent()) {
    final byte[] key = externalStoreKey.get();
    final byte[] realPayload = externalPayloadSource.read(key); // fetching real
payload
    final ConsumerRecord<String, byte[]> adapter = new
ConsumerRecord<>(source.topic(), source.partition(), source.offset(), source.key(),
realPayload);
    return Optional.of(adapter);
}

----------------

private Optional<byte[]> extractKeyFromEnvelope(final ConsumerRecord<String, byte[]>
record) {
    final byte[] payload = record.value();
    if (payload.length < EnvelopeConstants.POINTER_ENVELOPE_SIZE) {
        return Optional.empty();
    }
    for (int i = 0; i < EnvelopeConstants.HEADER_LENGTH; ++i) {
        if (EnvelopeConstants.ENVELOPE_HEADER[i] != payload[i]) {
            return Optional.empty();
        }
    }
    final byte envelopeVersion = payload[EnvelopeConstants.ENVELOPE_HEADER.length];
    if (EnvelopeConstants.POINTER_ENVELOPE_HEADER != envelopeVersion) {
        return Optional.empty();
    }
    final byte[] externalKey = new byte[payload.length -
EnvelopeConstants.POINTER_ENVELOPE_SIZE];
    System.arraycopy(payload, EnvelopeConstants.POINTER_ENVELOPE_SIZE, externalKey,
0, payload.length - EnvelopeConstants.POINTER_ENVELOPE_SIZE);
    return Optional.of(externalKey);
}
```

In case of Kafka-send failures, it is necessary to rollback the external store. The payload generated by the store during write operation is provided as an argument to the `rollback` method. It is then the external store's responsibility to do any necessary cleanup:

```
for (final ProducerAction action : records) {
    action.rollback();
}

----------------

final ProducerRecord<String, byte[]> envelope = buildPointerRecord(record,
pointerPayload);
final Runnable rollback = () -> {
    externalPayloadSink.rollback(pointerPayload);
};
return Optional.of(new ProducerAction(envelope, rollback));
```

kafka-over-redis

In this implementation, we have used Redis as an external data store, but other type of persistent storage (like RDBMS) could be used too. The implementation is present as redis-backed external store.

To uniquely recognize payloads, each Redis key is based on the original message UUID and Kafka physical name of destination:

```
final String keyStr = String.format("kafka-over-redis-%s-%s",
destination.getPhysicalName(), uuid);
```

With Redis, the maximum payload size that can be stored as a single entry is 512 MB. Payloads larger than 512 MB need to be saved as multiple entries.

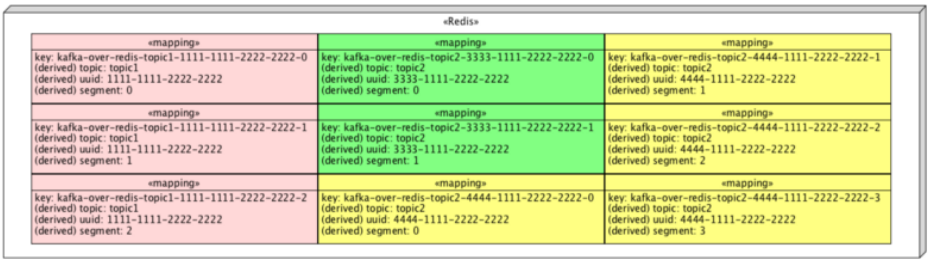
The key is just the (above computed) keyStr with segment number. Right now the expiry policy has to be maintained separately for Redis and Kafka (see `this.dataExpiry` below).

```
final int segments = Long.valueOf((length / DATA_ALLOWED) + (0 == length %
DATA_ALLOWED ? 0 : 1)).intValue();
for (int i = 0; i < segments; ++i) {
    final byte[] segmentKey = String.format("%s-%d", keyStr,
i).getBytes(KEY_CHARSET);
    final int start = DATA_ALLOWED * i;
    final long proposal = (long) DATA_ALLOWED * (i + 1);
    final int end = Long.valueOf(proposal < length ? proposal : length).intValue();
    final int segmentLength = end - start;
    final byte segment[] = new byte[segmentLength];
    System.arraycopy(payload, start, segment, 0, segmentLength);
    // storing the segment
    this.redisClient.setex(segmentKey, this.dataExpiry, segment);
}
```

Since we could have saved the payload as more than one entry in Redis, we need to provide the number of segments in the response payload:

```
final byte[] keyBytes = keyStr.getBytes(KEY_CHARSET);
final byte[] result = new byte[keyBytes.length + RESPONSE_HEADER_SIZE];
ByteBuffer bb = ByteBuffer.wrap(result);
bb.put(RESPONSE_HEADER_VERSION);
bb.putInt(segments);
bb = null;
System.arraycopy(keyBytes, 0, result, RESPONSE_HEADER_SIZE, keyBytes.length);
return result;
```

The diagram below shows how 3 messages could be stored, with 2, 3, or 4 segments each:



Receiving the real payload is just reading N segments with the key extracted from the received argument:

```
public byte[] read(final byte[] arg) {
    final StoreRequest request = parseRequest(arg);
    for (int i = 0; i < request.segmentCount; ++i) {
        final byte[] segmentKey = String.format("%s-%d", request.key,
i).getBytes(KEY_CHARSET);
        final byte[] segment = this.redisClient.get(segmentKey); // reading the
segment
        if (null != segment) {
            segments.add(segment);
        }
        else {
            throw ...
        }
    }
```

The received segments then get merged together and returned to the parent client.

A rollback step is very similar to a read step. Instead of reading payloads, they are deleted.

Comparison

This table summarizes the differences:

Chunking	External Store
Does not use additional services	Depends on external store (e.g. Redis)
Whole payload (chunks) get put into Kafka Requires special care for partition size (default 2gb could be overfilled by one chunked-request)	Pointer payload present only in Kafka
Payload expiry handled by Kafka (retention policy)	Payload expiry has to be maintained in external store (e.g. via Redis' setex)
Internal buffer needs to be modified/cleaned up when consumer's offset is explicitly moved	Pointer payload is a single Kafka message, so no special operations
Increased memory footprint due to keeping not-yet-resolved chunks	Pointer payload is a single Kafka message, so no special operations
Sending to random partition forces sending all chunks to one partition (currently partition 0).	Pointer payload is a single Kafka message, so no special operations
It is possible to lose a chunked message if group consumer switches in middle of chunked message	Pointer payload is a single Kafka message, so no special operations
It is not possible to remove chunks after they have been successfully published, even if the latter ones fail. Might be extended by adding non-user payloads marking certain chunks as failed, forcing their eviction from consumer chunk store.	Rollback needs to be handled by external store.
Compaction cannot be enabled on partitions where chunking is used. Otherwise, compaction would remove all chunks except the last one published.	Compaction would remove kafka pointer only. Explicit cleanup in external store might be required (related to payload expiry)