A Kafka-based Ordering Service for Fabric

Author: Kostas Christidis

Thanks to: Jason Yellick, Bishop Brock, Gari Singh, Christian Cachin, and Binh Nguyen for their suggestions and feedback when compiling this document.

Introduction

We use Kafka to provide ordering and support for multiple chains in a crash fault tolerant manner. The ordering service consists of a Kafka cluster with its corresponding ZooKeeper ensemble, and a set of ordering service nodes that stand between the clients of the ordering service and the Kafka cluster (**Fig. 1**).

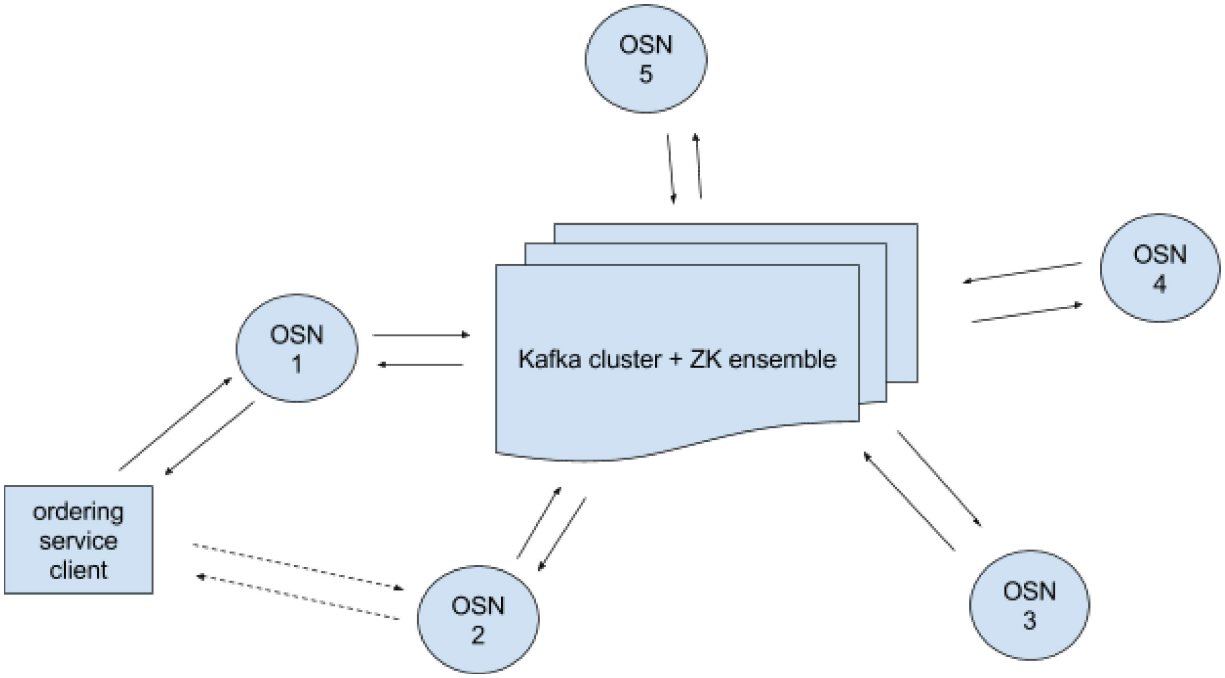

Fig. 1. An ordering service, consisting of 5 ordering service nodes (OSN-n), and a Kafka cluster. The ordering service client can be connected to multiple OSNs. Note that the OSNs do not communicate with each other directly.

These ordering service nodes (**OSNs**) (1) do client authentication, (2) allow clients to write to a chain[1] or read from it using a simple interface, and (3) they also do transaction filtering and validation for configuration transactions that either reconfigure an existing chain or create a new one.

[1] Where chain refers to the log that a group of clients (a “channel”) has access to. For more about channels, see this document.

Where does Kafka come in?

Messages (records) in Kafka get written to a topic partition. A Kafka cluster can have multiple topics, and each topic can have multiple partitions (**Fig. 2**). *Each partition is an ordered, immutable sequence of records that is continually appended to.*

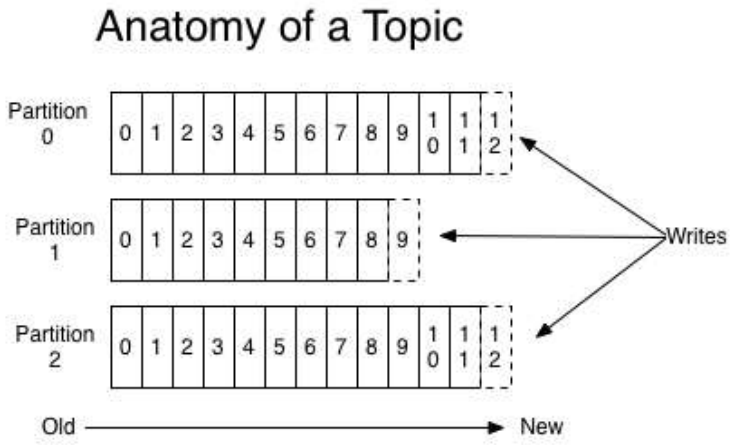

Fig. 2. A topic that consists of three partitions. Each record in a partition is tagged with an offset number. (Diagram taken from the Kafka website.)

Solution 0. Assume then that for every chain we have a separate partition[2]. Once the OSNs have performed client authentication and transaction filtering, they can relay the incoming client transactions belonging to a certain chain to the chain's corresponding partition. They can then consume that partition and get back an ordered list of transactions that is common across all ordering service nodes. This is the high-level view of how we can use Kafka to achieve ordering (Fig. 3).

[2] Whether these partitions belong to the same topic or not, is an implementation detail that is irrelevant at this point.

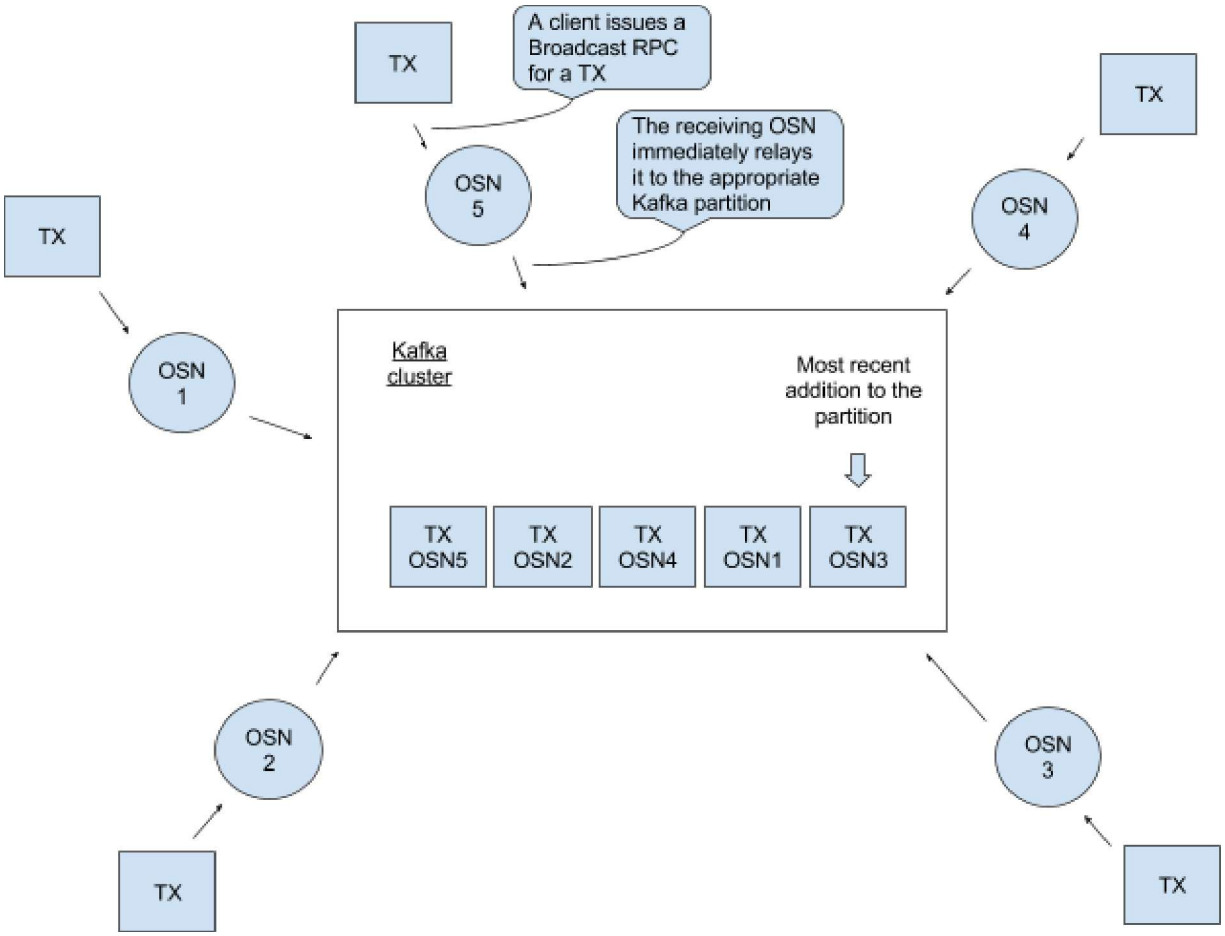

Fig. 3. Assume all client transactions here belong to the same chain. OSNs relay incoming client transactions to the same partition in the Kafka cluster. They can then read from that partition and get all the transactions that were posted to the network in an order that will be the same for all OSNs.

Are we done? Let’s see.

In this case, each client transaction is a separate block; the OSN packages every transaction into a message of type Block, assigns it a number equal to the offset number assigned to it by the Kafka cluster, and signs the block (for auditability). Any Deliver RPCs issued by clients are served by setting up a Kafka consumer on the partition that would seek to specified block/offset number.

This could certainly work.

Problem 1. But consider for a second an incoming transaction rate of, say, 1K transactions per second. The ordering service now has to generate 1K signatures per second. And the clients on the receiving side have to be able to verify 1K signatures per second. As signing and verifying is generally expensive, this could get tricky.

Solution 1. So instead of having each transaction contained in a separate block, let's instead do batching. For the example above, assuming a batch of 1K transactions, the ordering service now has to create only one signature, and the client only has one signature to verify.

That is much better.

Problem 2. But what if the rate with which transactions come in is not uniform? Assume that the ordering service just sent out a batch of 1K transactions, now it has 999 transactions stored in memory and waits for one more transaction to come before it can create a new batch. But no transaction gets posted to the ordering service for an hour. These first 999 transactions of this batch get delivered with an unacceptable delay.

This is not good.

Solution 2. So we realize we need to have a batch timer. When the first transaction for a new batch comes in, we set a timer. Then a block is cut either when we reach the maximum number of messages that we want the block to have (**batchSize**), or when the timer expires (**batchTimeout** seconds elapse) — whichever comes first. This takes care of the 999 transactions in Problem 2, as it makes sure that the block is cut in a timely manner.

Problem 3. However, cutting blocks based on time calls for coordination among the service nodes. Cutting blocks based on number of messages is easy — for every, say, 2 messages (that's your batchSize) you read off a partition, you cut a block. This partition reads the same for all OSNs so as a client you're guaranteed that you'll get the same sequence of blocks no matter which OSN you reach out to. Consider now the case where you have a batchTimeout of 1 second, and two OSNs. A batch has just been cut and a new transaction comes in via, say, OSN1. It gets posted to the partition. OSN2 reads it at time t=5s and sets a timer that will fire at t=6s. OSN1 reads this transaction from the partition at time t=5.6s and sets its own timer accordingly. A second transaction is posted to the partition and is read by OSN2 at t=6.2s, and by OSN1 at t=6.5s. We are now in a situation where the OSN1 has cut a block with both of these transactions, whereas OSN2 cut a block with just the first of them. The two OSNs have now diverged and are outputting different sequences of blocks — this is unacceptable.

Solution 3. So cutting blocks based on time requires an explicit coordination signal. Let's assume that each OSN posts a "time to cut block X" message on the partition (where X is an integer corresponding to the next block in the sequence) before it cuts a block, and that it won't actually cut block X until it reads a "time to cut block X" message from the partition first. Note that it doesn't have to be its own "time to cut block X" (hereinafter referred to as **TTC-X** for brevity); if every OSN waited for their own TTC-X message we would end up once again with diverging sequences. Instead, each OSN cuts block X either when batchSize messages have been collected, or when *the first* TTC-X message is received — whichever happens first.

This implies that all subsequent TTC-X messages (for the same X) will be ignored by the OSNs. **Fig 4.** shows an example of how this could work.

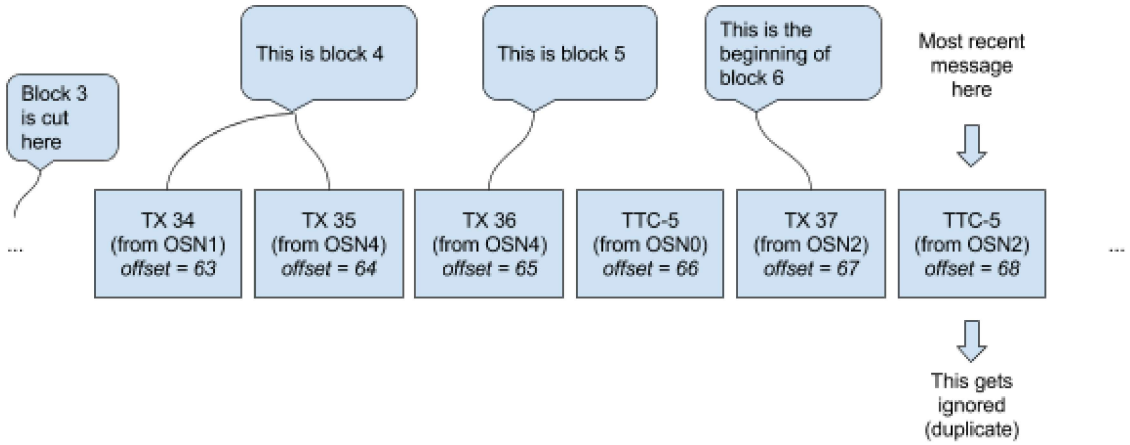

Fig. 4. OSNs post incoming transactions and TTC-X messages to the partition.

We have now figured out a way to cut blocks based on either size, or time and keep the block sequence consistent across all OSNs - that's progress.

Can we stop here? Let's see.

Problem 4. Unlike the case where each transaction was placed in a separate block, the block number now does *not* translate to Kafka's offset number for that partition. So if the ordering service receives a Deliver request starting from block, say, 5, it literally has no idea which offset its consumer should seek to.

That's not good.

Solution 4a. But maybe we could use the Metadata field of the Block message type, and have the OSN note down which offset range this block carries (e.g. Metadata of Block 4 would read "offsets: 63-64"). Then if a client with access to block 9 wanted to get a stream starting from block 5, it would issue a Deliver RPC by setting the starting number to 65, and the OSN could replay the partition log starting from offset 65, cutting blocks every batchSize messages found, or whenever the first TTC-X (where X is bigger than the last compiled block number is) is found – whichever comes first[3]. There are two problems with this approach. One, we're violating the Deliver API contract which calls for the *block* number as a parameter. Two, what happens if a client has missed a bunch of blocks, and just wishes to get a random block X (without owning a copy of X-1), or the most recent block (via the "NEWEST" parameter in the

[3] Note that a client could theoretically send an offset number that doesn't correspond to the beginning of a block, and then they would get back the wrong sequence from the OSN, but they would only be shooting themselves in the foot if they were to do that.

SeekInfo message)? It doesn't know what the right offset number to pass along with the Deliver call is, and the OSN cannot figure that out either.

Solution 4b. So each OSN needs to maintain a table per chain, mapping block numbers, to the first offset that they contain - **Table 1** shows an example of this. *Note that this means that an OSN cannot accommodate a Deliver request unless it has built a lookup table that includes the requested block number.*

Block Number	Offset Number
...	...
4	63
5	65
...	...

Table 1. Example lookup table corresponding to Fig. 4. A block number is mapped to the offset number of the first *transaction* that it should contain.

This lookup table removes the need for block metadata, and for the client to figure out the right offset for the Deliver request. The OSN translates the requested block number to the right offset and sets up a consumer to seek to it. So we have now overcome this problem as well, by having the OSN maintain some state.

This could certainly work.

But there are two problems that we'd like to take care of. **Problem 5.** One, notice that whenever an OSN receives a Deliver request, it has to retrieve all the messages from the partition starting from the requested block number, package them into blocks, and sign them. This packaging and signing is repeated on every replay request and could get expensive. **Problem 6.** Two, since we now have redundant messages in the partition that we need to skip, serving a Deliver request is not as simple as setting up a consumer, seeking to the requested offset, and replaying all the records forward; the lookup table needs to be consulted at all times – the deliver logic is becoming more complex, and these checks on the lookup table result in added latency. Let's set that second problem aside for a second, and focus on the first problem for now. In order to overcome it, we have to persist these blocks as we create them. Then, when replaying we would just have to transmit the persisted blocks.

Solution 5a. In order to solve this, assume that we want to go all in on Kafka, and we create yet another partition – let's call it **partition 1**, which implies that the partition we kept referencing so far was **partition 0**. Now whenever the OSNs cut a block they post it to partition 1, and all Deliver requests will be served by that partition. Since *each* OSN posts the block they cut in partition 1, we expect a sequence of blocks in there that does not translate directly to the exact chain

sequence; there will be duplicates and, more generally, the block numbers across all OSNs will not be in strictly increasing order — **Fig. 5** shows an example of this.

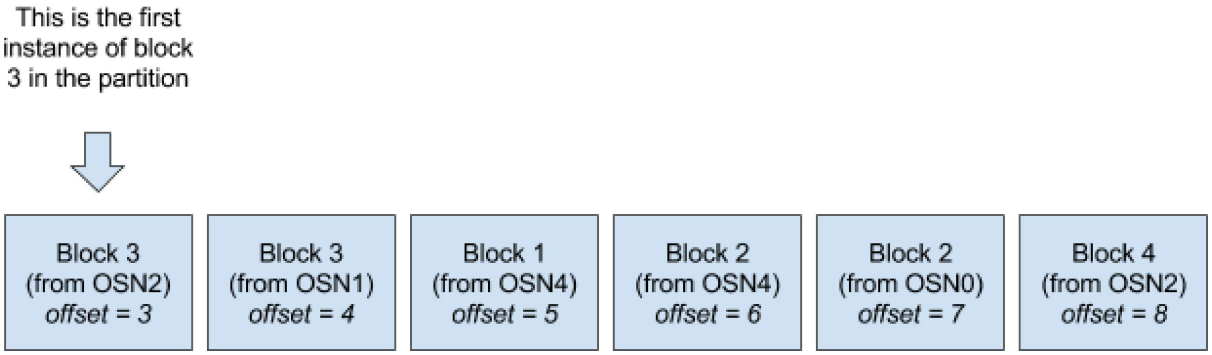

Fig. 5. A possible state of partition 1. Most recent message gets appended to the right. Notice how the block numbers across all OSNs are not in a strictly increasing order (3-3-1-2-2-4). However, the numbers *per OSN* form a strictly increasing sequence as expected (eg. 1-2 for OSN4).

This means that the Kafka-assigned offset numbers cannot map to the OSN-assigned (contiguous) block numbers, so a block-to-offset number lookup table, similar to what we've seen in the previous solution, would also need to be maintained here. **Table 2** shows an example of this.

Block Number	Offset Number
...	...
3	3
4	8
...	...

Table 2. Example lookup table for the partition in Fig. 5, assuming that the first block 3 reference in partition 1 is the one captured in the first block of that figure. The difference with the lookup table in *Table 1* is that the offset number here maps to a block (in partition 1), not a transaction (in partition 0).

This could work. But notice that the second problem that we pointed out earlier, Problem 6, still remains; the Deliver logic on the OSN side is complicated and the lookup table needs to be consulted on every request.

Solution 6. If you think about it, what causes the problem here are the redundant messages that get posted to the partition, whether these are the TTC-X messages of partition 0 (see last message in Fig. 4), or the "Block X" messages that get posted to partition 1 and are smaller or equal to earlier messages (all the in-between

messages in Fig. 5 are smaller or equal to the leftmost Block 3). So how can we get rid of these redundant messages?

Solution 6a. First, let's adopt a rule that says: if you've already received a message from the partition that is identical (minus the signature) to what you are about to post to it, abort your own post operation. Then, going back to the example in Fig. 5, if OSN1 had witnessed the Block 3 message from OSN2 in partition 1 before it had formed its own block 3, it would skip its transmission to partition 1. (Everything we describe here has an equivalent example for partition 0 and the Fig. 4 example.) While this would certainly cut down on the duplicate messages, it would not eliminate them completely. There would certainly be instances where OSNs would post the same blocks around the same time, or while an identical block was in flight to the Kafka broker set. Even if you were to be more clever about this (i.e. add some random delay to each node before they could post on the chain), you would still run into this issue.

Solution 6b. How about if we had a leader OSN that would be responsible for posting the next block to partition 1? There are several ways to elect a leader; for example you could make all OSNs compete for a znode in ZooKeeper, or have the leader be the first node who had its TTC-X message posted in partition 0. Another interesting approach would be to have all OSNs belong to the same consumer group, which means that each OSN gets to own (exclusively) a partition in a topic. (Assume a topic that carries only all partition-0's for all chains, and another topic that carries all partition-1's.) Then the OSN responsible for sending, say, the TTC-X message to partition-0, or posting the block to partition-1 for a chain, would be that chain's partition 0 owner — and this is a state that is managed by Kafka and can be queriedt natively via the Kafka Metadata API. (Deliver requests to an OSN for a chain/partition it does not own would have to be redirected to the appropriate OSN behind the scenes.)

That could work, but what happens if the leader sends the Block X message to partition 1, then crashes right away. *The message is still en route to partition 1 but not posted yet*. The OSNs realized the leader has crashed because it no longer holds the leader znode in ZooKeeper and a new leader is elected. The new leader realizes that Block X is in its buffer (it should be the oldest block there) but still not posted in partition 1, so it posts it to partition 1. Now block X from the old leader finally makes its way to partition 1 as well — we got duplicates again! This sequence of events is captured in **Fig. 6**.

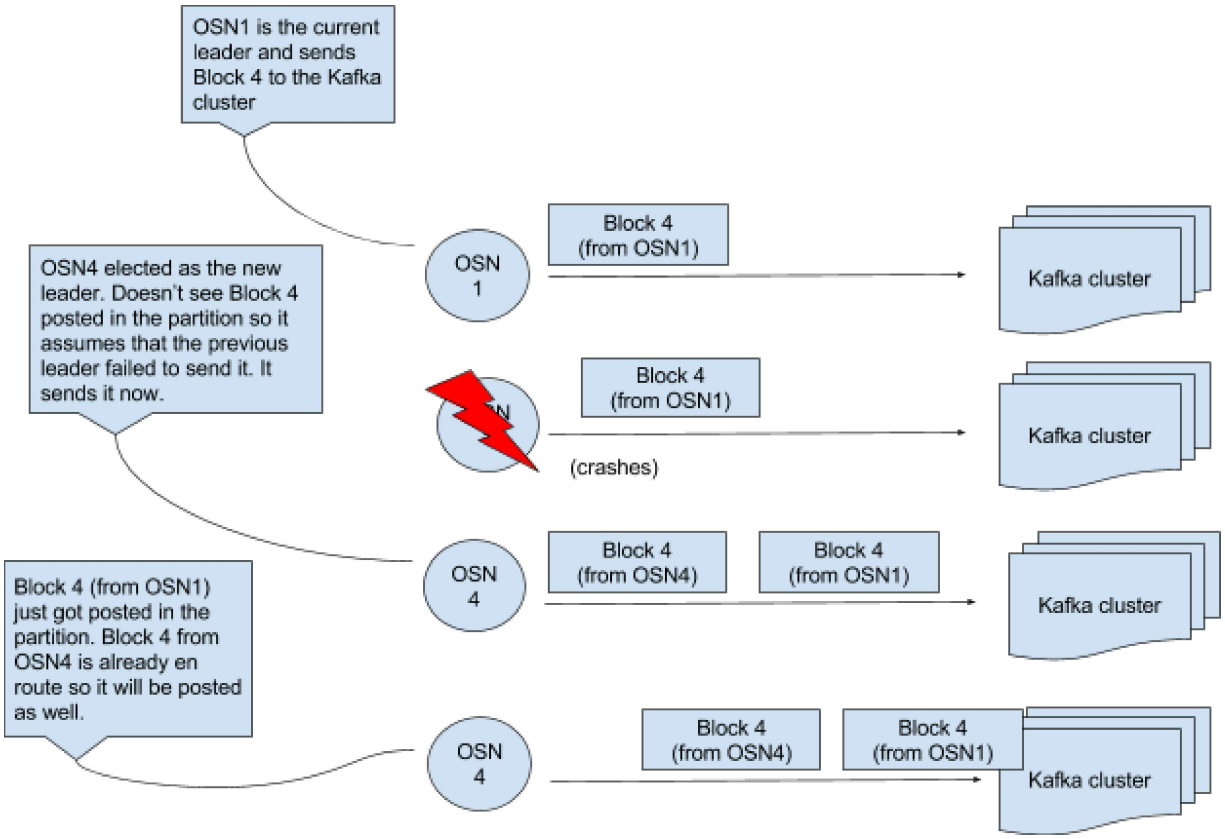

Fig. 6. Having a leader does not eliminate duplicate messages.

Solution 6c. Could log compaction help? Log compaction ensures that Kafka retains the last known value for each message key in a partition (records in Kafka are key/value pairs) — the diagram in **Fig. 7** depicts this better:

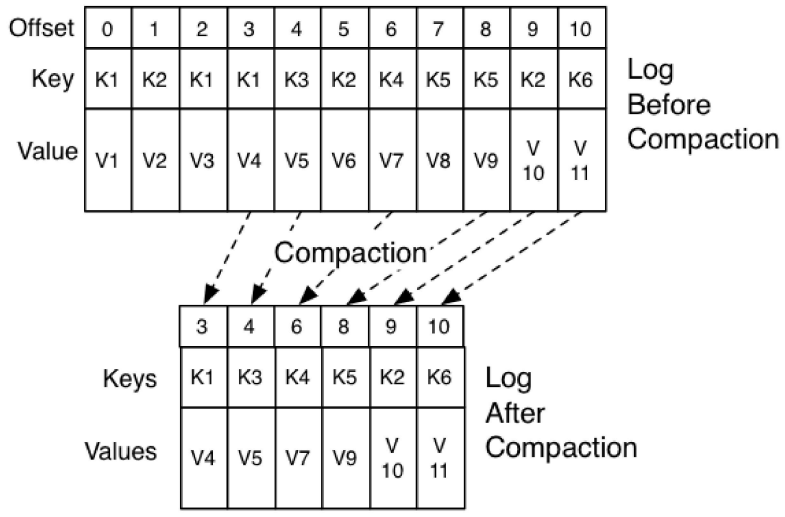

Fig. 7. Log compaction in Kafka. (Diagram taken from the Kafka website.)

So if we were to turn this on, we could certainly eliminate all duplicates from the partition, assuming, of course, that all Block X messages carry the same key, and that key is different for different values of X. (The key being X is the way to go here.) But because log compaction keeps around the latest version of a key, the OSNs could end up with stale lookup tables. Working with the example of Figure 7, and assuming the keys listed there map to blocks, an OSN that has received just the first two messages in that partition has a lookup table that maps block 1 to offset 0 and block 2 to offset 1. Meanwhile, the partition has been compacted to what the bottom half of the diagram shows so seeking to offset 0 (or 1) would result in an error message. Equally importantly though a problem with log compactions, is that the blocks in partition 1 would not be stored in ascending order (again, this is clearly shown in the bottom half of the diagram in Fig. 7 where the sequence is 1-3-4-5-2-6) so the Deliver logic would still remain complicated (remember that this is the problem that we set out solve here – Problem 7). In fact, given the risk of lookup tables going stale, the log compaction route is clearly not the way to go.

So none of the solutions offered here solve Problem 6. Let's see if there's a way around this, and for that, let's go back for a second to problem 5 – persisting the blocks so that replays are faster.

Solution 5b. Instead of going for yet another Kafka partition as shown in Solution 6a, we could stick with partition 0 where transactions and TTC-X messages are posted, and instead have each OSN maintain a local log for every chain (**Fig. 8**).

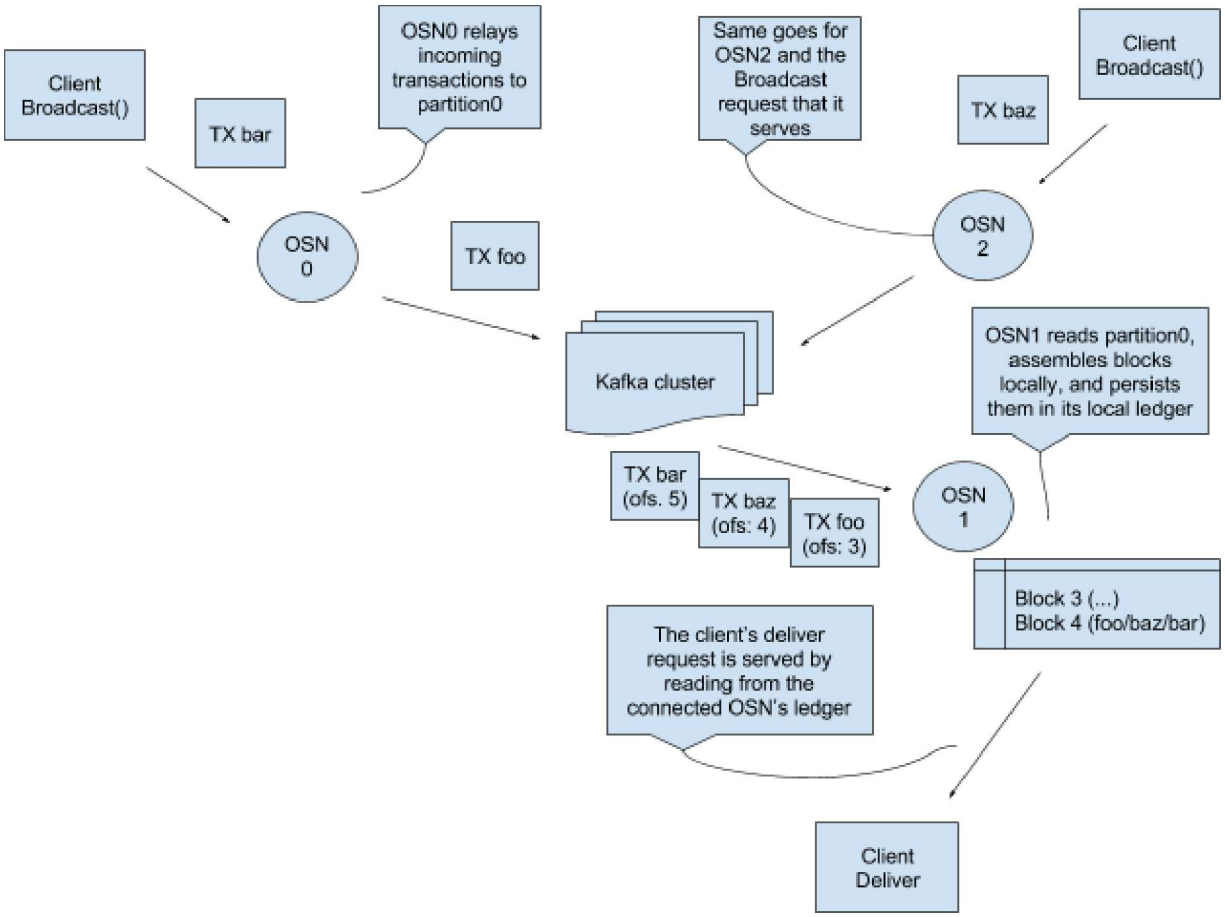

Fig. 8. The proposed design. For simplicity we assume all transactions here belong to the same chain, but the design works for multiple chains.

This has the following *additional* benefits.

One, it solves Problem 6 — **Solution 6d**: Accommodating a Deliver request is now a matter of just reading sequentially from the local ledger. (No duplicates as the OSN would filter them when writing to the local log[4], no lookup tables! The OSN will need to keep track of the last offset number it read though, just so that it knows where to seek to when consuming from Kafka upon reconnection.)

Two, it maximizes the use of common components across the orderer codebase — the Deliver path essentially becomes identical across all existing implementations.

A downside of serving Deliver requests from the local ledger could be that it would be slower than serving them straight from Kafka. But we never serve straight from Kafka; there's always some processing happening on the OSNs.

Specifically, if we are talking about replay requests, the alternatives (solutions 4b and 5a) still require processing on the OSNs (whether this is packaging and checking

[4] An operation that would only have to happen once.

of the lookup table in 4b, or just the latter in 5a), so a penalty is already being paid in the ledger-less solutions.

If we are talking about Deliver requests to keep current (the “tail -f” equivalent), the additional penalty of Solution 6d compared to 4b is that of storing the block to the ledger and serving it from there. In Solution 5a, the block needs to make a second round-trip via partition 1, so depending on the environment this may even be worse than 6d.

Overall an ordering service that uses a single partition (per chain) for incoming client transactions and TTC-X messages (as shown in Solution 3), and which stores the resulting blocks in a local ledger (again, per chain) strikes a nice balance between performance and complexity.