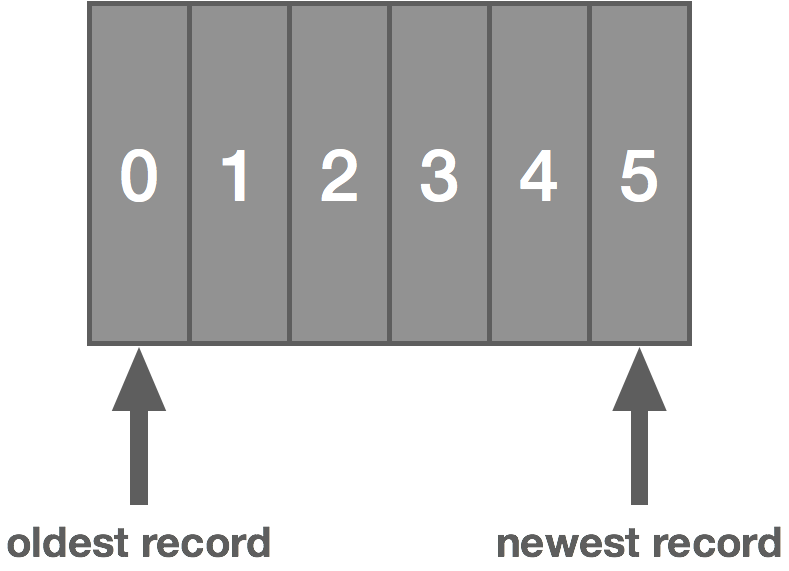
# Building a Distributed Log from Scratch, Part 1: Storage Mechanics

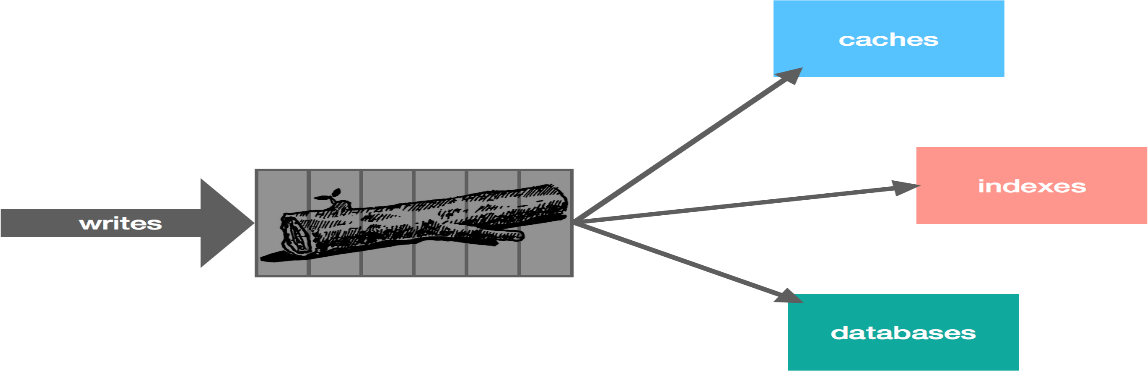
The log is a totally-ordered, append-only data structure. It’s a powerful yet simple abstraction—a sequence of immutable events. It’s something that programmers have been using for a very long time, perhaps without even realizing it because it’s so simple. Whether it’s application logs, system logs, or access logs, logging is something every developer uses on a daily basis. Essentially, it’s a timestamp and an event, a *when* and a *what*, and typically appended to the end of a file. But when we generalize that pattern, we end up with something much more useful for a broad range of problems. It becomes more interesting when we look at the log not just as a system of record but a central piece in managing data and distributing it across the enterprise efficiently.

[](https://bravenewgeek.com/wp-content/uploads/2017/12/log.png)

There are a number of implementations of this idea: [Apache Kafka](https://kafka.apache.org/), [Amazon Kinesis](https://aws.amazon.com/kinesis/data-streams/), [NATS Streaming](https://github.com/nats-io/nats-streaming-server), [Tank](https://github.com/phaistos-networks/TANK), and [Apache Pulsar](https://pulsar.apache.org/) to name a few. We can probably credit Kafka with popularizing the idea.

I think there are at least three key priorities for the effectiveness of one of these types of systems: performance, high availability, and scalability. If it’s not fast enough, the data becomes decreasingly useful. If it’s not highly available, it means we can’t reliably get our data in or out. And if it’s not scalable, it won’t be able to meet the needs of many enterprises.

When we apply the traditional pub/sub semantics to this idea of a log, it becomes a very useful abstraction that applies to a lot of different problems.

[](https://bravenewgeek.com/wp-content/uploads/2017/12/log_use_cases.png)

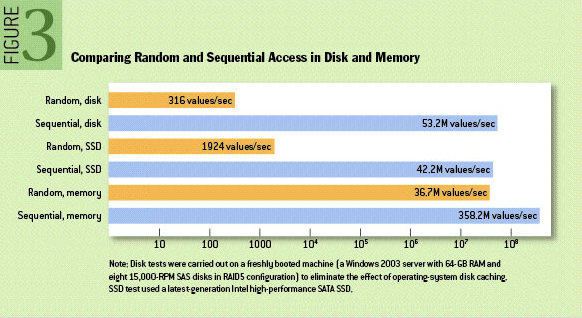
In this series, we’re not going to spend much time discussing *why* the log is useful. Jay Kreps has already done the legwork on that with [*The Log: What every software engineer should know about real-time data’s unifying abstraction*](https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying). There’s even a [book](https://www.amazon.com/Heart-Logs-Stream-Processing-Integration/dp/1491909382) on it. Instead, we will focus on what it takes to *build* something like this using Kafka and NATS Streaming as case studies of sorts—Kafka because of its ubiquity, NATS Streaming because it’s something with which I have personal experience. We’ll look at a few core components like leader election, data replication, log persistence, and message delivery. Part one of this series starts with the storage mechanics. Along the way, we will also discuss some lessons learned while building NATS Streaming, which is a streaming data layer on top of the [NATS](https://nats.io/) messaging system. The intended outcome of this series is threefold: to learn a bit about the internals of a log abstraction, to learn how it can achieve the three goals described above, and to learn some applied distributed systems theory.

With that in mind, you will probably never need to build something like this yourself (nor should you), but it helps to know how it works. I also find that software engineering is all about pattern matching. Many types of problems look radically different but are surprisingly similar. Some of these ideas may apply to other things you come across. If nothing else, it’s just *interesting*.

Let’s start by looking at data storage since this is a critical part of the log and dictates some other aspects of it. Before we dive into that, though, let’s highlight some first principles we’ll use as a starting point for driving our design.

As we know, the log is an ordered, immutable sequence of messages. Messages are *atomic*, meaning they can’t be broken up. A message is either in the log or not, all or nothing. Although we only ever add messages to the log and never remove them (as with a message queue), the log has a notion of *message retention* based on some policies, which allows us to control how the log is truncated. This is a practical requirement since otherwise the log will grow endlessly. These policies might be based on time, number of messages, number of bytes, etc.

The log can be played back from any arbitrary position. With position, we normally refer to a logical message timestamp rather than a physical wall-clock time, such as an offset into the log. The log is stored on disk, and sequential disk access is actually relatively *fast*. The graphic below taken from the ACM Queue article [*The Pathologies of Big Data*](http://queue.acm.org/detail.cfm?id=1563874) helps bear this out (this is helpfully pointed out by Kafka’s [documentation](https://kafka.apache.org/documentation/#design_filesystem)).

[](https://bravenewgeek.com/wp-content/uploads/2017/12/disk_access.png)

That said, modern OS page caches mean that sequential access often avoids going to disk altogether. This is because the kernel keeps cached pages in otherwise unused portions of RAM. This means both reads and writes go to the in-memory page cache instead of disk. With Kafka, for example, we can verify this quite easily by running a simple test that writes some data and reads it back and looking at disk IO using *iostat*. After running such a test, you will likely see something resembling the following, which shows the number of blocks read and written is exactly zero.

avg-cpu: %user %nice %system %iowait %steal %idle

13.53 0.00 11.28 0.00 0.00 75.19

Device: tps Blk\_read/s Blk\_wrtn/s Blk\_read Blk\_wrtn

xvda 0.00 0.00 0.00 0 0

With the above in mind, our log starts to look an awful lot like an actual logging file, but instead of timestamps and log messages, we have offsets and opaque data messages. We simply add new messages to the end of the file with a monotonically increasing offset.

[](https://bravenewgeek.com/wp-content/uploads/2017/12/log_file.png)

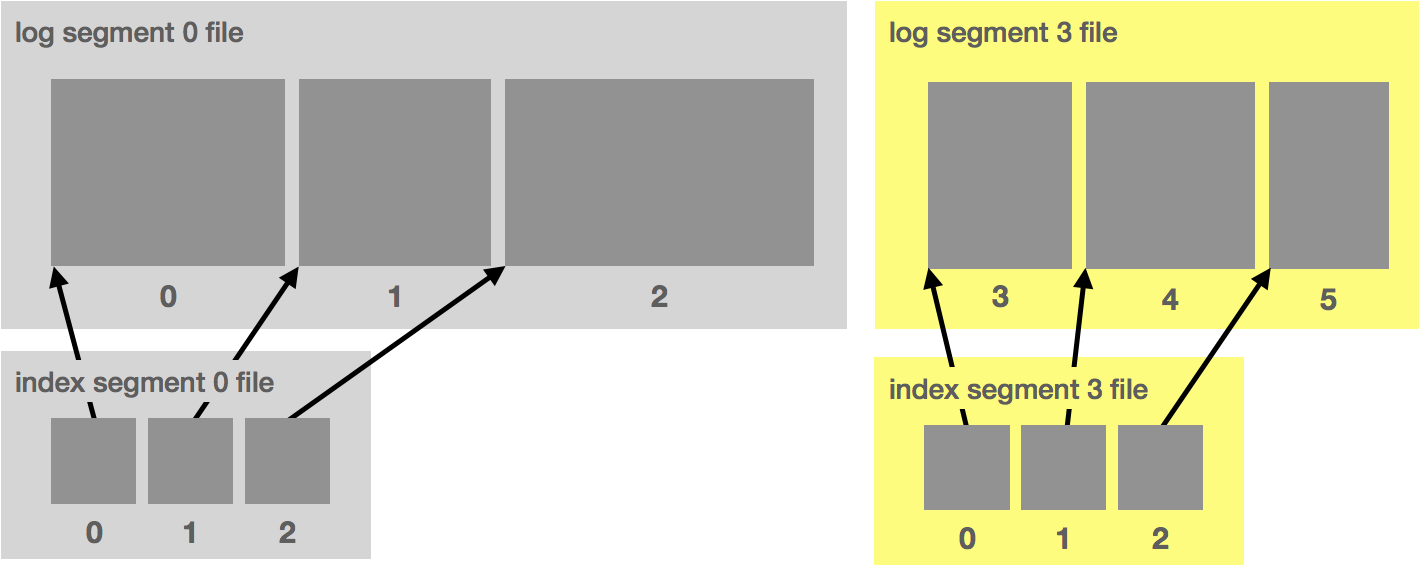
However, there are some problems with this approach. Namely, the file is going to get very, very large. Recall that we need to support a few different access patterns: looking up messages by offset and also truncating the log using a variety of different retention policies. Since the log is ordered, a lookup is simply a binary search for the offset, but this is expensive with a large log file. Similarly, aging out data by retention policy is harder.

To account for this, we break up the log file into chunks. In Kafka, these are called segments. In NATS Streaming, they are called slices. Each segment is a new file. At a given time, there is a single active segment, which is the segment messages are written to. Once the segment is full (based on some configuration), a new one is created and becomes active.

Segments are defined by their base offset, i.e. the offset of the first message stored in the segment. In Kafka, the files are also named with this offset. This allows us to quickly locate the segment in which a given message is contained by doing a binary search.

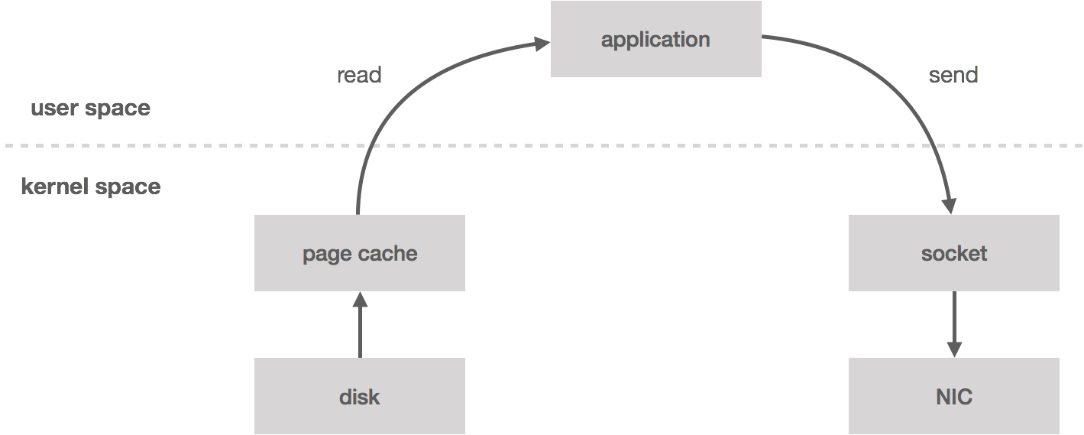
[](https://bravenewgeek.com/wp-content/uploads/2017/12/log_segments.png)

Alongside each segment file is an index file that maps message offsets to their respective positions in the log segment. In Kafka, the index uses 4 bytes for storing an offset relative to the base offset and 4 bytes for storing the log position. Using a relative offset is more efficient because it means we can avoid storing the actual offset as an int64. In NATS Streaming, the timestamp is also stored to do time-based lookups.

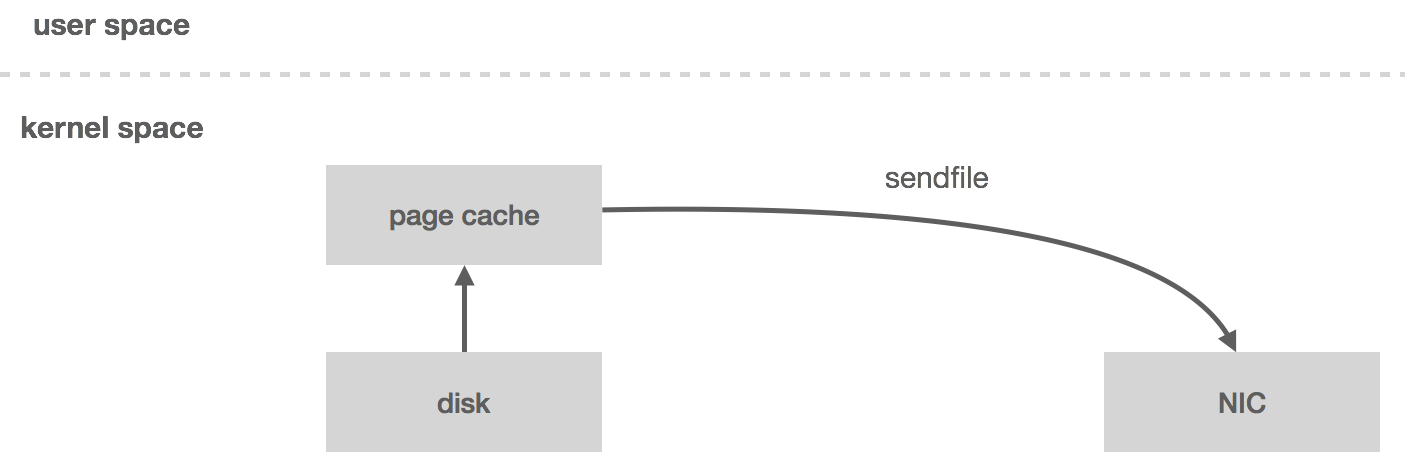
[](https://bravenewgeek.com/wp-content/uploads/2017/12/log_index.png)

Ideally, the data written to the log segment is written in protocol format. That is, what gets written to disk is exactly what gets sent over the wire. This allows for zero-copy reads. Let’s take a look at how this otherwise works.

When you read messages from the log, the kernel will attempt to pull the data from the page cache. If it’s not there, it will be read from disk. The data is copied from disk to page cache, which all happens in kernel space. Next, the data is copied into the application (i.e. user space). This all happens with the *read* system call. Now the application writes the data out to a socket using *send*, which is going to copy it back into kernel space to a socket buffer before it’s copied *one last time* to the NIC. All in all, we have *four* copies (including one from page cache) and *two* system calls.

[](https://bravenewgeek.com/wp-content/uploads/2017/12/read.png)

However, if the data is already in wire format, we can bypass user space entirely using the *sendfile*system call, which will copy the data directly from the page cache to the NIC buffer—*two* copies (including one from page cache) and *one* system call. This turns out to be an important optimization, especially in garbage-collected languages since we’re bringing less data into application memory. Zero-copy also reduces CPU cycles and memory bandwidth.

[](https://bravenewgeek.com/wp-content/uploads/2017/12/sendfile.png)

NATS Streaming does not currently make use of zero-copy for a number of reasons, some of which we will get into later in the series. In fact, the NATS Streaming storage layer is actually *pluggable* in that it can be backed by any number of mediums which implement the storage interface. Out of the box it includes the file-backed storage described above, in-memory, and SQL-backed.

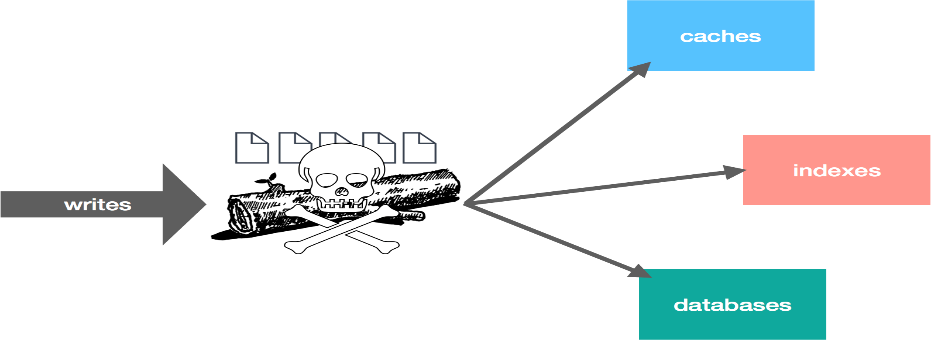
There are a few other optimizations to make here such as message batching and compression, but we’ll leave those as an exercise for the reader.

In [part two](https://bravenewgeek.com/building-a-distributed-log-from-scratch-part-2-data-replication/) of this series, we will discuss how to make this log fault tolerant by diving into data-replication techniques.

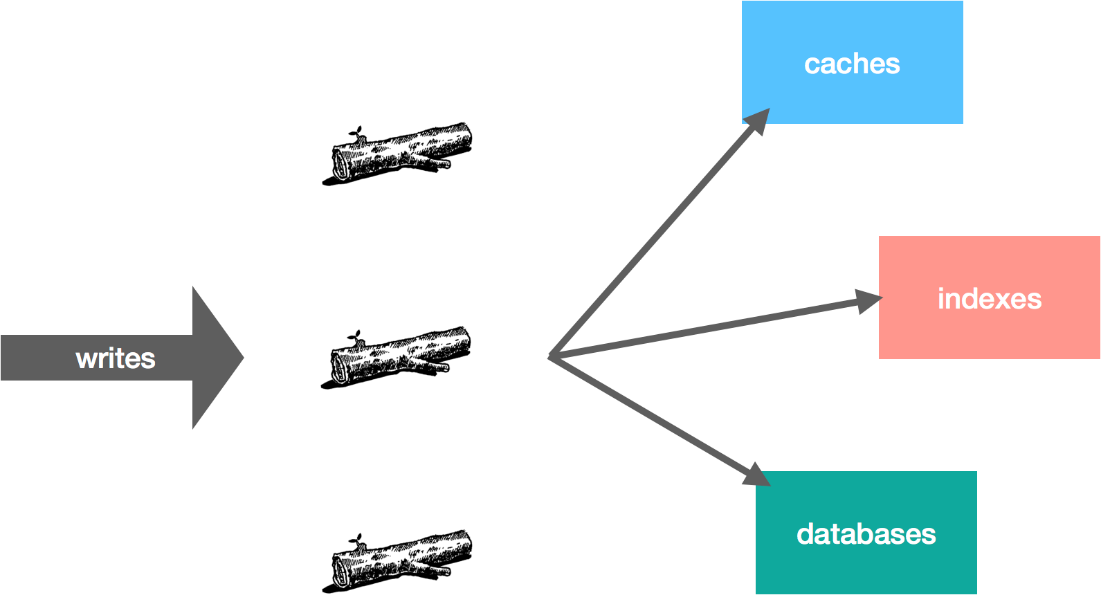
## [**Building a Distributed Log from Scratch, Part 2: Data Replication**](https://bravenewgeek.com/building-a-distributed-log-from-scratch-part-2-data-replication/)

In [part one](https://bravenewgeek.com/building-a-distributed-log-from-scratch-part-1-storage-mechanics/) of this series we introduced the idea of a message log, touched on why it’s useful, and discussed the storage mechanics behind it. In part two, we discuss data replication.

We have our log. We know how to write data to it and read it back as well as how data is persisted. The caveat to this is, although we have a durable log, it’s a single point of failure (SPOF). If the machine where the log data is stored dies, we’re SOL. Recall that one of our three priorities with this system is high availability, so the question is how do we achieve high availability and fault tolerance?

[](https://bravenewgeek.com/wp-content/uploads/2017/12/spof.png)

With high availability, we’re specifically talking about ensuring continuity of reads and writes. A server failing shouldn’t preclude either of these, or at least unavailability should be kept to an absolute minimum and without the need for operator intervention. Ensuring this continuity should be fairly obvious: we eliminate the SPOF. To do that, we replicate the data. Replication can also be a means for increasing scalability, but for now we’re only looking at this through the lens of high availability.

[](https://bravenewgeek.com/wp-content/uploads/2017/12/replicated_log.png)

There are a number of ways we can go about replicating the log data. Broadly speaking, we can group the techniques into two different categories: gossip/multicast protocols and consensus protocols. The former includes things like epidemic broadcast trees, bimodal multicast, SWIM, HyParView, and NeEM. These tend to be eventually consistent and/or stochastic. The latter, which I’ve described in more detail [here](https://bravenewgeek.com/understanding-consensus/), includes 2PC/3PC, Paxos, Raft, Zab, and chain replication. These tend to favor strong consistency over availability.

So there are a lot of ways we can replicate data, but some of these solutions are better suited than others to this particular problem. Since ordering is an important property of a log, consistency becomes important for a replicated log. If we read from one replica and then read from another, it’s important those views of the log don’t conflict with each other. This more or less rules out the stochastic and eventually consistent options, leaving us with consensus-based replication.

There are essentially two components to consensus-based replication schemes: 1) designate a leader who is responsible for sequencing writes and 2) replicate the writes to the rest of the cluster.

Designating a leader can be as simple as a configuration setting, but the purpose of replication is fault tolerance. If our configured leader crashes, we’re no longer able to accept writes. This means we need the leader to be dynamic. It turns out leader election is a well-understood problem, so we’ll get to this in a bit.

Once a leader is established, it needs to replicate the data to followers. In general, this can be done by either waiting for all replicas or waiting for only a quorum (majority) of replicas. There are pros and cons to both approaches.

|  |  |  |
| --- | --- | --- |
|  | **Pros** | **Cons** |
| **All Replicas** | Tolerates f failures with f+1replicas | Latency pegged to slowest replica |
| **Quorum** | Hides delay from a slow replica | Tolerates f failures with 2f+1replicas |

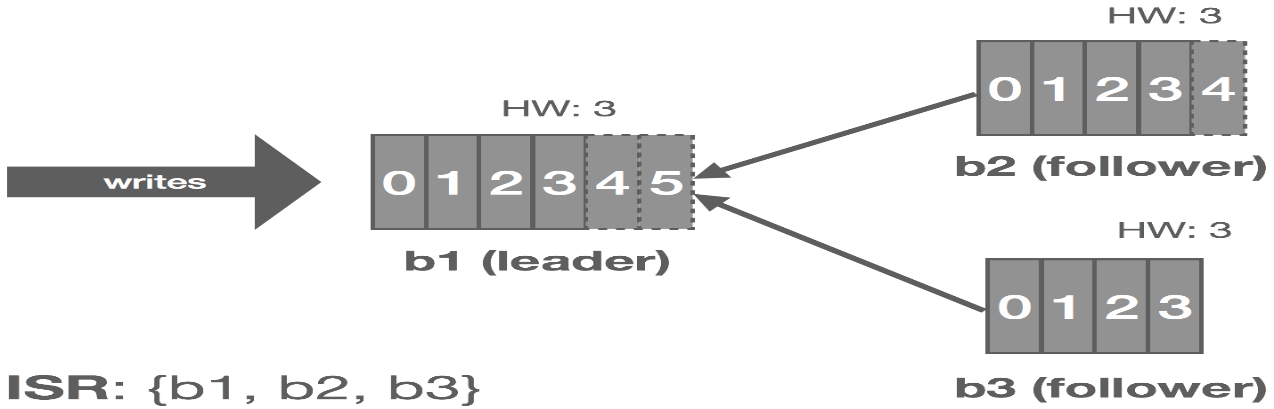
Waiting on all replicas means we can make progress as long as at least one replica is available. With quorum, tolerating the same amount of failures requires more replicas because we need a majority to make progress. The trade-off is that the quorum hides any delays from a slow replica. Kafka is an example of a system which uses all replicas (with some conditions on this which we will see later), and NATS Streaming is one that uses a quorum. Let’s take a look at both in more detail.

### **Replication in Kafka**

In Kafka, a leader is selected (we’ll touch on this in a moment). This leader maintains an in-sync replica set (ISR) consisting of all the replicas which are fully caught up with the leader. This is every replica, by definition, at the beginning. All reads and writes go through the leader. The leader writes messages to a write-ahead log (WAL). Messages written to the WAL are considered uncommitted or “dirty” initially. The leader only commits a message once all replicas in the ISR have written it to their own WAL. The leader also maintains a high-water mark (HW) which is the last committed message in the WAL. This gets piggybacked on the replica fetch responses from which replicas periodically checkpoint to disk for recovery purposes. The piggybacked HW then allows replicas to know when to commit.

Only committed messages are exposed to consumers. However, producers can configure how they want to receive acknowledgements on writes. It can wait until the message is committed on the leader (and thus replicated to the ISR), wait for the message to only be written (but not committed) to the leader’s WAL, or not wait at all. This all depends on what trade-offs the producer wants to make between latency and durability.

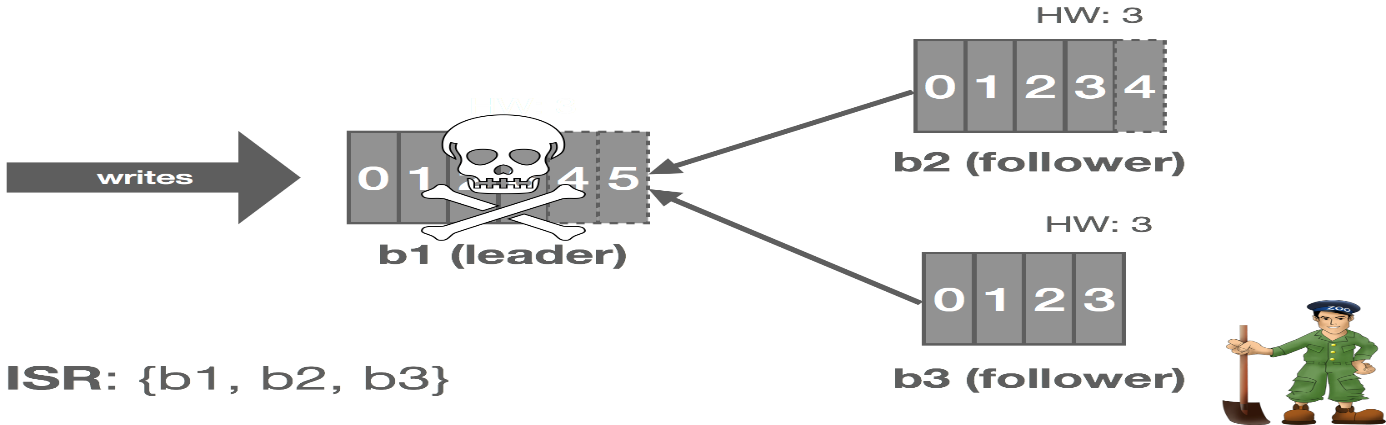
The graphic below shows how this replication process works for a cluster of three brokers: b1, b2, and b3. Followers are effectively special consumers of the leader’s log.

[](https://bravenewgeek.com/wp-content/uploads/2017/12/kafka_replication.png)

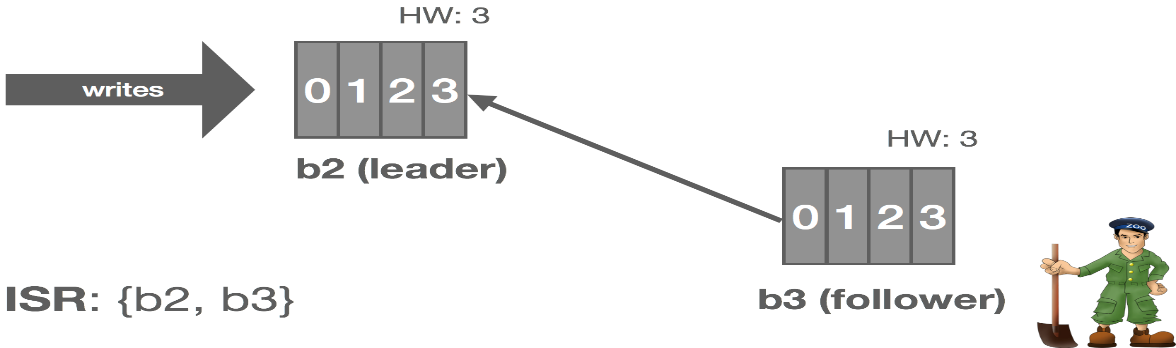
Now let’s look at a few failure modes and how Kafka handles them.

#### Leader Fails

Kafka relies on [Apache ZooKeeper](https://zookeeper.apache.org/) for certain cluster coordination tasks, such as leader election, though this is not actually how the log leader is elected. A Kafka cluster has a single controller broker whose election is handled by ZooKeeper. This controller is responsible for performing administrative tasks on the cluster. One of these tasks is selecting a new log leader (actually partition leader, but this will be described later in the series) from the ISR when the current leader dies. ZooKeeper is also used to detect these broker failures and signal them to the controller.

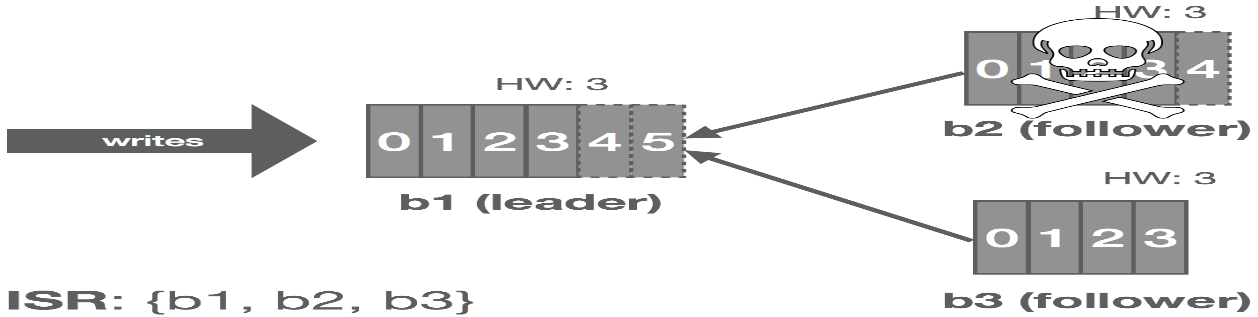
[](https://bravenewgeek.com/wp-content/uploads/2017/12/kafka_leader_failure.png)

Thus, when the leader crashes, the cluster controller is notified by ZooKeeper and it selects a new leader from the ISR and announces this to the followers. This gives us automatic failover of the leader. All committed messages up to the HW are preserved and uncommitted messages may be lost during the failover. In this case, b1 fails and b2steps up as leader.

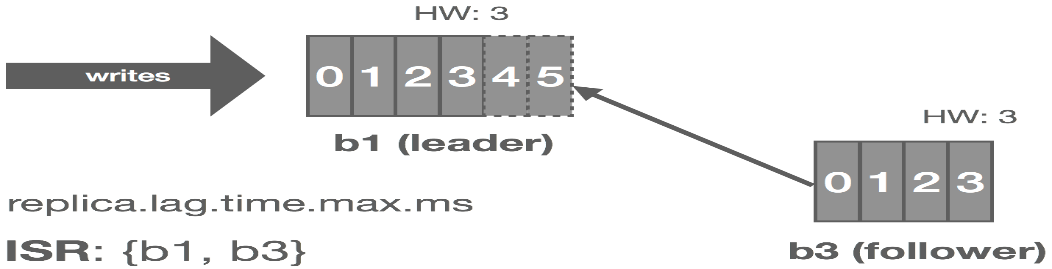
[](https://bravenewgeek.com/wp-content/uploads/2017/12/kafka_leader_failover.png)

#### Follower Fails

The leader tracks information on how “caught up” each replica is. Before Kafka 0.9, this included both how many messages a replica was behind, replica.lag.max.messages, and the amount of time since the replica last fetched messages from the leader, replica.lag.time.max.ms. Since 0.9, replica.lag.max.messages was removed and replica.lag.time.max.ms now refers to both the time since the last fetch request and the amount of time since the replica last caught up.

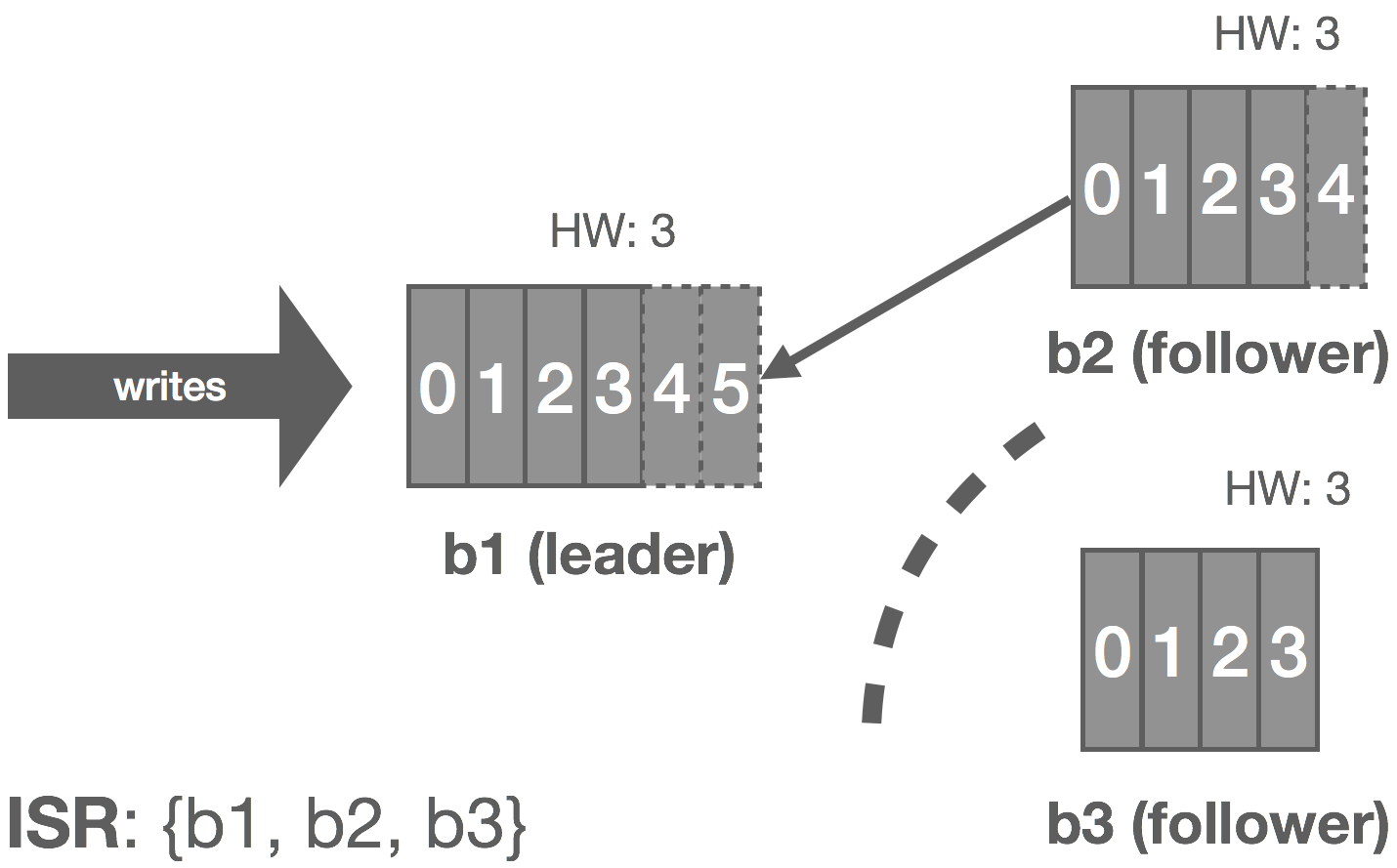
[](https://bravenewgeek.com/wp-content/uploads/2017/12/kafka_follower_failure.png)

Thus, when a follower fails (or stops fetching messages for whatever reason), the leader will detect this based on replica.lag.time.max.ms. After that time expires, the leader will consider the replica out of sync and remove it from the ISR. In this scenario, the cluster enters an “under-replicated” state since the ISR has shrunk. Specifically, b2 fails and is removed from the ISR.

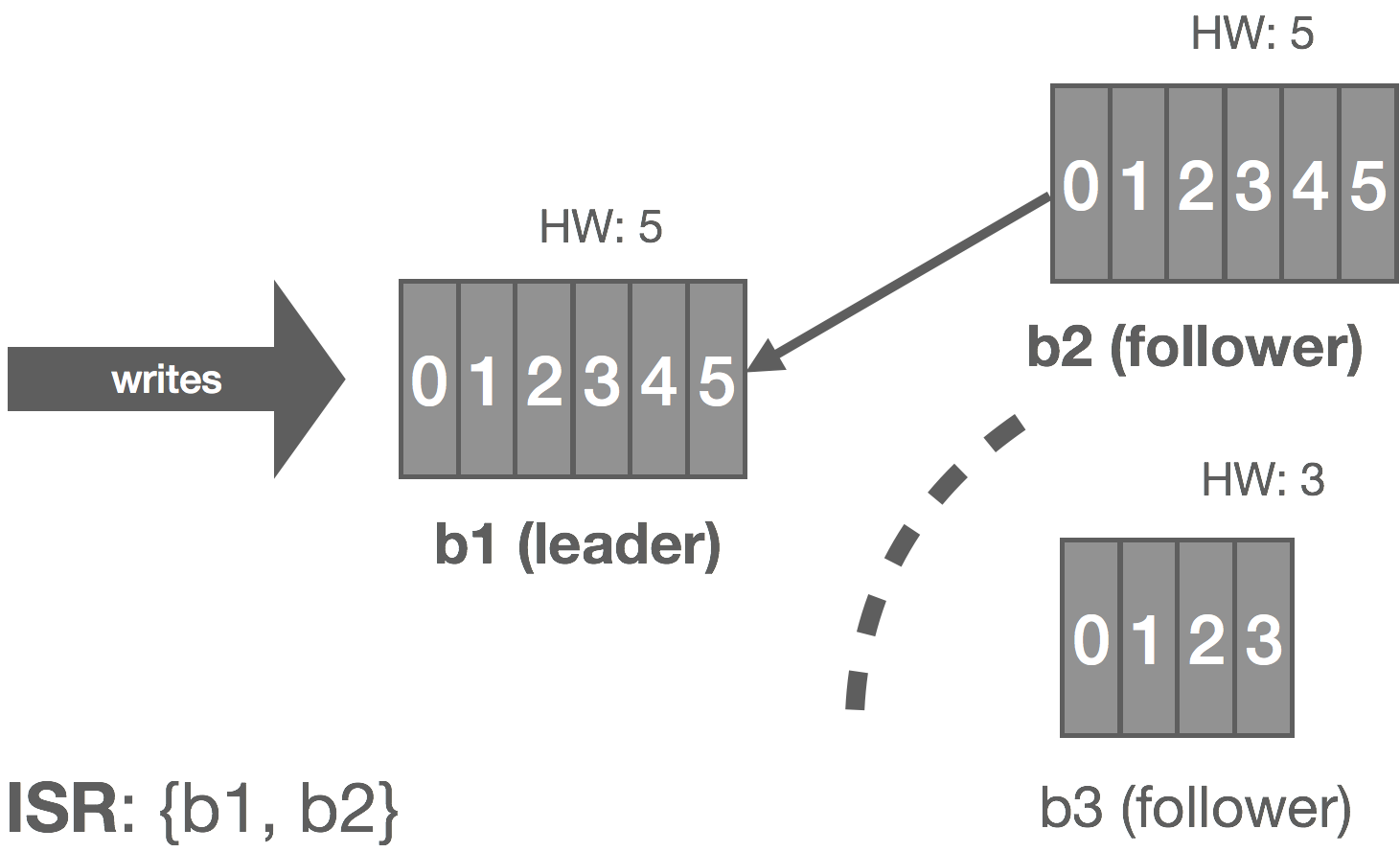
[](https://bravenewgeek.com/wp-content/uploads/2017/12/kafka_follower_failure_removed.png)

#### Follower Temporarily Partitioned

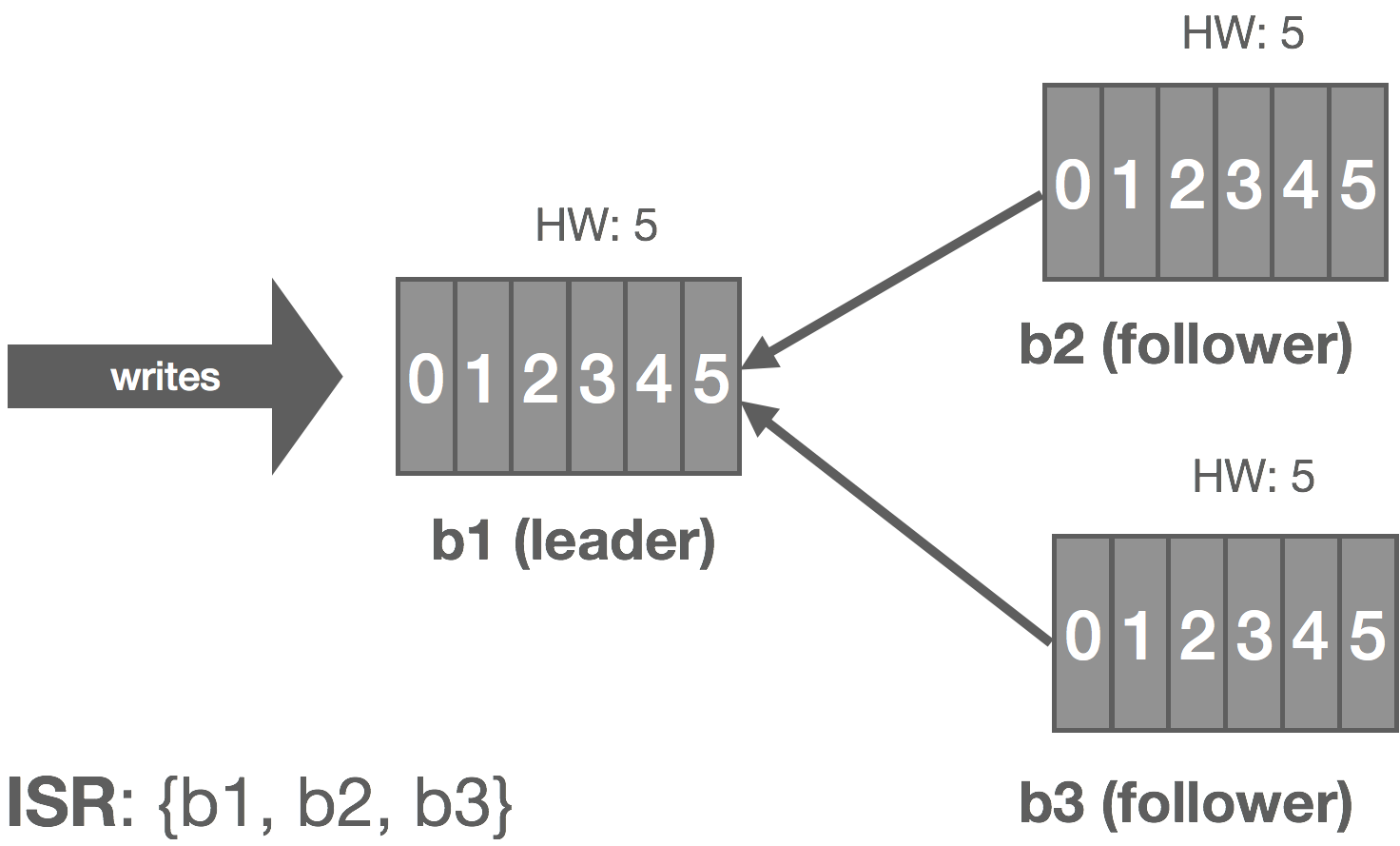
The case of a follower being temporarily partitioned, e.g. due to a transient network failure, is handled in a similar fashion to the follower itself failing. These two failure modes can really be combined since the latter is just the former with an arbitrarily long partition, i.e. it’s the difference between crash-stop and crash-recovery models.

[](https://bravenewgeek.com/wp-content/uploads/2017/12/kafka_follower_partition.png)

In this case, b3 is partitioned from the leader. As before, replica.lag.time.max.ms acts as our failure detector and causes b3 to be removed from the ISR. We enter an under-replicated state and the remaining two brokers continue committing messages 4 and 5. Accordingly, the HW is updated to 5 on these brokers.

[](https://bravenewgeek.com/wp-content/uploads/2017/12/kafka_follower_partition_removed.png)

When the partition heals, b3 continues reading from the leader and catching up. Once it is fully caught up with the leader, it’s added back into the ISR and the cluster resumes its fully replicated state.

[](https://bravenewgeek.com/wp-content/uploads/2017/12/kafka_follower_partition_healed.png)

We can generalize this to the crash-recovery model. For example, instead of a network partition, the follower could crash and be restarted later. When the failed replica is restarted, it recovers the HW from disk and truncates its log up to the HW. This preserves the invariant that messages after the HW are not guaranteed to be committed. At this point, it can begin catching up from the leader and will end up with a log consistent with the leader’s once fully caught up.

### **Replication in NATS Streaming**

NATS Streaming relies on the [Raft consensus algorithm](https://raft.github.io/) for leader election and data replication. This sometimes comes as a surprise to some as Raft is largely seen as a protocol for replicated state machines. We’ll try to understand why Raft was chosen for this particular problem in the following sections. We won’t dive deep into Raft itself beyond what is needed for the purposes of this discussion.

While a log is a state machine, it’s a very simple one: a series of appends. Raft is frequently used as the replication mechanism for key-value stores which have a clearer notion of “state machine.” For example, with a key-value store, we have set and delete operations. If we set foo = bar and then later set foo = baz, the state gets rolled up. That is, we don’t necessarily care about the provenance of the key, only its current state.

However, NATS Streaming differs from Kafka in a number of key ways. One of these differences is that NATS Streaming attempts to provide a sort of unified API for streaming and queueing semantics not too dissimilar from [Apache Pulsar](https://pulsar.apache.org/). This means, while it has a notion of a log, it also has subscriptions on that log. Unlike Kafka, NATS Streaming tracks these subscriptions and metadata associated with them, such as where a client is in the log. These have definite “state machines” affiliated with them, like creating and deleting subscriptions, positions in the log, clients joining or leaving queue groups, and message-redelivery information.

Currently, NATS Streaming uses multiple Raft groups for replication. There is a single metadata Raft group used for replicating client state and there is a separate Raft group per topic which replicates messages and subscriptions.

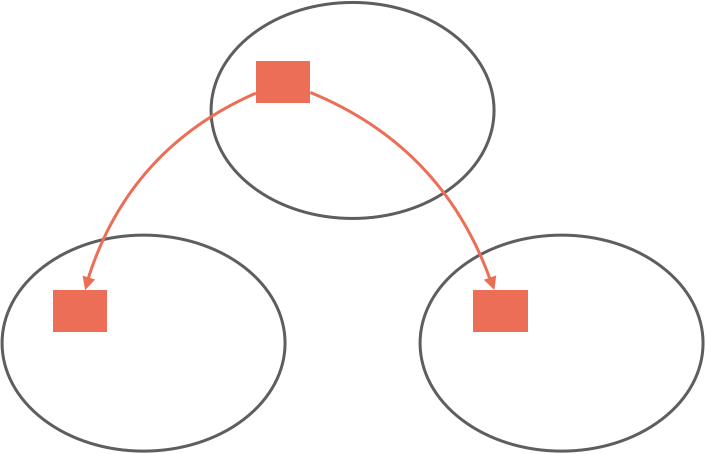
Raft solves both the problems of leader election and data replication in a single protocol. The [Secret Lives of Data](http://thesecretlivesofdata.com/raft/) provides an excellent interactive illustration of how this works. As you step through that illustration, you’ll notice that the algorithm is actually quite similar to the Kafka replication protocol we walked through earlier. This is because although Raft is used to implement replicated state machines, it actually is a replicated WAL, which is exactly what Kafka is. One benefit of using Raft is we no longer have the need for ZooKeeper or some other coordination service.

Raft handles electing a leader. Heartbeats are used to maintain leadership. Writes flow through the leader to the followers. The leader appends writes to its WAL and they are subsequently piggybacked onto the heartbeats which get sent to the followers using AppendEntries messages. At this point, the followers append the write to their own WALs, assuming they don’t detect a gap, and send a response back to the leader. The leader commits the write once it receives a successful response from a quorum of followers.

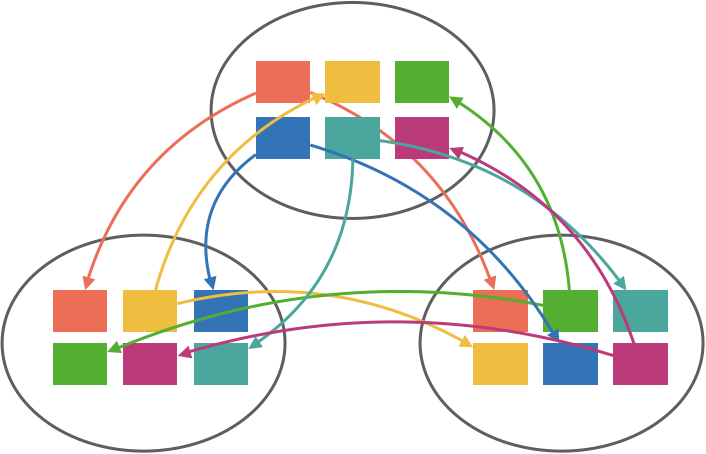
Similar to Kafka, each replica in Raft maintains a high-water mark of sorts called the commit index, which is the index of the highest log entry known to be committed. This is piggybacked on the AppendEntries messages which the followers use to know when to commit entries in their WALs. If a follower detects that it missed an entry (i.e. there was a gap in the log), it rejects the AppendEntries and informs the leader to rewind the replication. The [Raft paper](https://raft.github.io/raft.pdf) details how it ensures correctness, even in the face of many failure modes such as the ones described earlier.

Conceptually, there are two logs: the Raft log and the NATS Streaming message log. The Raft log handles replicating messages and, once committed, they are appended to the NATS Streaming log. If it seems like there’s some redundancy here, that’s because there is, which we’ll get to soon. However, keep in mind we’re not just replicating the message log, but also the state machines associated with the log and any clients.

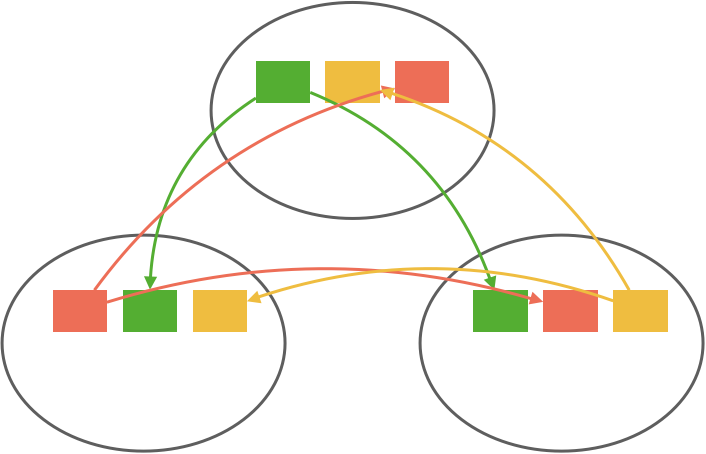
There are a few challenges with this replication technique, two of which we will talk about. The first is scaling Raft. With a single topic, there is one Raft group, which means one node is elected leader and it heartbeats messages to followers.

[](https://bravenewgeek.com/wp-content/uploads/2017/12/raft_single_topic.png)

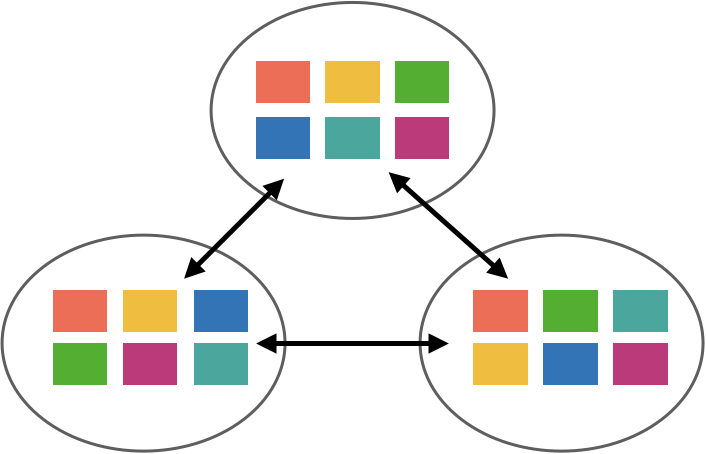
As the number of topics increases, so do the number of Raft groups, each with their own leaders and heartbeats. Unless we constrain the Raft group participants or the number of topics, this creates an explosion of network traffic between nodes.

[](https://bravenewgeek.com/wp-content/uploads/2017/12/raft_many_topics.png)

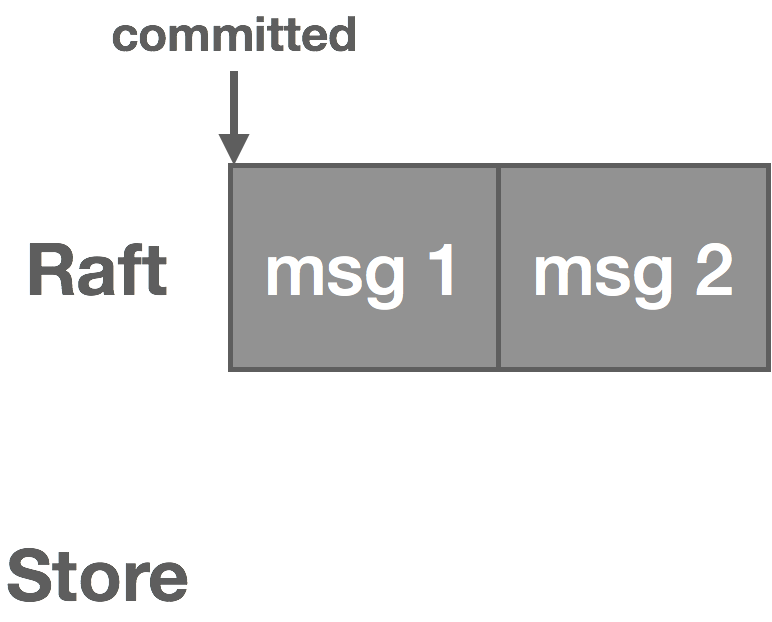
There are a couple ways we can go about addressing this. One option is to run a fixed number of Raft groups and use a consistent hash to map a topic to a group. This can work well if we know roughly the number of topics beforehand since we can size the number of Raft groups accordingly. If you expect only 10 topics, running 10 Raft groups is probably reasonable. But if you expect 10,000 topics, you probably don’t want 10,000 Raft groups. If hashing is consistent, it would be feasible to dynamically add or remove Raft groups at runtime, but it would still require repartitioning a portion of topics which can be complicated.

[](https://bravenewgeek.com/wp-content/uploads/2017/12/raft_fixed_groups.png)

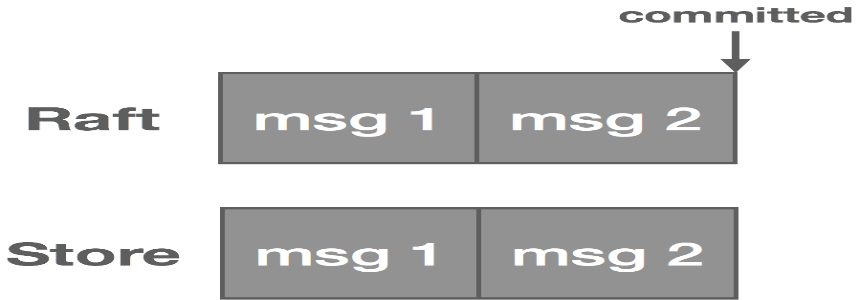
Another option is to run an entire node’s worth of topics as a single group using a layer on top of Raft. This is what CockroachDB does to scale Raft in proportion to the number of key ranges using a layer on top of Raft they call [MultiRaft](https://www.cockroachlabs.com/blog/scaling-raft/). This requires some cooperation from the Raft implementation, so it’s a bit more involved than the partitioning technique but eschews the repartitioning problem and redundant heartbeating.

[](https://bravenewgeek.com/wp-content/uploads/2017/12/multiraft.png)

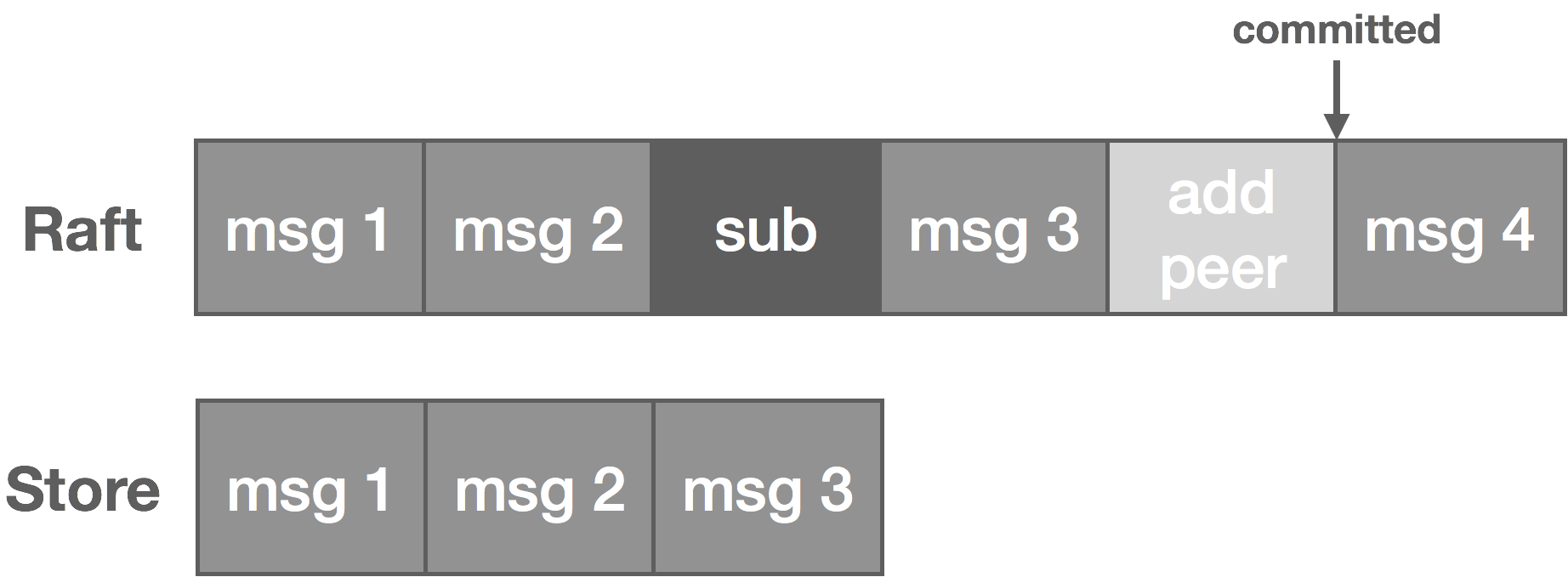
The second challenge with using Raft for this problem is the issue of “dual writes.” As mentioned before, there are really two logs: the Raft log and the NATS Streaming message log, which we’ll call the “store.” When a message is published, the leader writes it to its Raft log and it goes through the Raft replication process.

[](https://bravenewgeek.com/wp-content/uploads/2017/12/wal.png)

Once the message is committed in Raft, it’s written to the NATS Streaming log and the message is now visible to consumers.

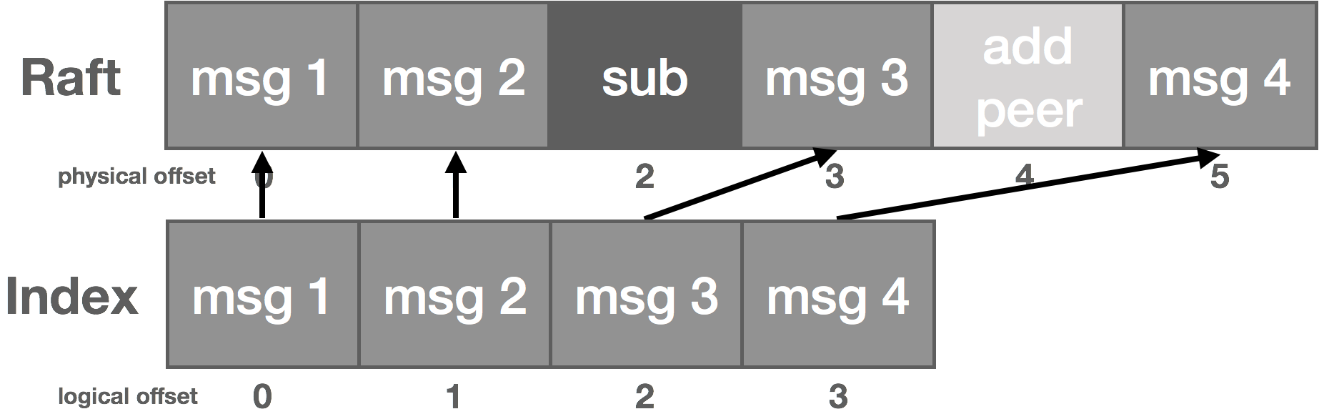
[](https://bravenewgeek.com/wp-content/uploads/2017/12/wal_committed.png)

Note, however, that not only messages are written to the Raft log. We also have subscriptions and cluster topology changes, for instance. These other items are not written to the NATS Streaming log but handled in other ways on commit. That said, messages tend to occur in much greater volume than these other entries.

[](https://bravenewgeek.com/wp-content/uploads/2017/12/dual_writes.png)

Messages end up getting stored redundantly, once in the Raft log and once in the NATS Streaming log. We can address this problem if we think about our logs a bit differently. If you recall from [part one](https://bravenewgeek.com/building-a-distributed-log-from-scratch-part-1-storage-mechanics/), our log storage consists of two parts: the log segment and the log index. The segment stores the actual log data, and the index stores a mapping from log offset to position in the segment.

Along these lines, we can think of the Raft log index as a “physical offset” and the NATS Streaming log index as a “logical offset.” Instead of maintaining two logs, we treat the Raft log as our message write-ahead log and treat the NATS Streaming log as an index into that WAL. Particularly, messages are written to the Raft log as usual. Once committed, we write an index entry for the message offset that points back into the log. As before, we use the index to do lookups into the log and can then read sequentially from the log itself.

[](https://bravenewgeek.com/wp-content/uploads/2017/12/raft_index.png)

### **Remaining Questions**

We’ve answered the questions of how to ensure continuity of reads and writes, how to replicate data, and how to ensure replicas are consistent. The remaining two questions pertaining to replication are how do we keep things fast and how do we ensure data is durable?

There are several things we can do with respect to performance. The first is we can configure publisher acks depending on our application’s requirements. Specifically, we have three options. The first is the broker acks on commit. This is slow but safe as it guarantees the data is replicated. The second is the broker acks on appending to its local log. This is fast but unsafe since it doesn’t wait on any replica roundtrips but, by that very fact, means that the data is not replicated. If the leader crashes, the message could be lost. Lastly, the publisher can just not wait for an ack at all. This is the fastest but least safe option for obvious reasons. Tuning this all depends on what requirements and trade-offs make sense for your application.

The second thing we do is don’t explicitly fsync writes on the broker and instead rely on replication for durability. Both Kafka and NATS Streaming (when clustered) do this. With fsync enabled (in Kafka, this is configured with flush.messages and/or flush.ms and in NATS Streaming, with file\_sync), every message that gets published results in a sync to disk. This ends up being very expensive. The thought here is if we are replicating to enough nodes, the replication itself is sufficient for HA of data since the likelihood of more than a quorum of nodes failing is low, especially if we are using rack-aware clustering. Note that data is still periodically flushed in the background by the kernel.

Batching aggressively is also a key part of ensuring good performance. Kafka supports end-to-end batching from the producer all the way to the consumer. NATS Streaming does not currently support batching at the API level, but it uses aggressive batching when replicating and persisting messages. In my experience, this makes about an order-of-magnitude improvement in throughput.

Finally, as already discussed earlier in the series, keeping disk access sequential and maximizing zero-copy reads makes a big difference as well.

There are a few things worth noting with respect to durability. Quorum is what guarantees durability of data. This comes “for free” with Raft due to the nature of that protocol. In Kafka, we need to do a bit of configuring to ensure this. Namely, we need to configure min.insync.replicas on the broker and acks on the producer. The former controls the minimum number of replicas that must acknowledge a write for it to be considered successful when a producer sets acks to “all.” The latter controls the number of acknowledgments the producer requires the leader to have received before considering a request complete. For example, with a topic that has a replication factor of three, min.insync.replicas needs to be set to two and acks set to “all.” This will, in effect, require a quorum of two replicas to process writes.

Another caveat with Kafka is unclean leader elections. That is, if all replicas become unavailable, there are two options: choose the first replica to come back to life (not necessarily in the ISR) and elect this replica as leader (which could result in data loss) or wait for a replica in the ISR to come back to life and elect it as leader (which could result in prolonged unavailability). Initially, Kafka favored availability by default by choosing the first strategy. If you preferred consistency, you needed to set unclean.leader.election.enable to false. However, as of 0.11, unclean.leader.election.enable now defaults to this.

Fundamentally, durability and consistency are at odds with availability. If there is no quorum, then no reads or writes can be accepted and the cluster is unavailable. This is the crux of the [CAP theorem](https://bravenewgeek.com/cap-and-the-illusion-of-choice/).

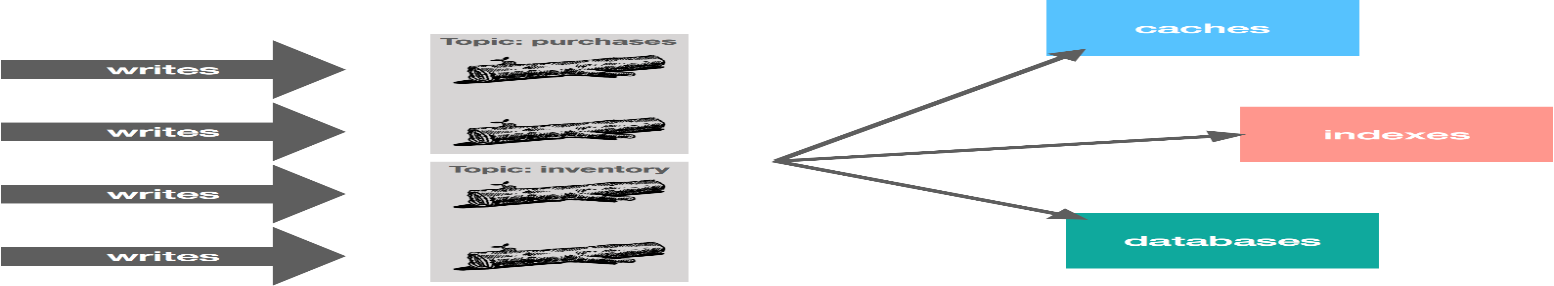
# Building a Distributed Log from Scratch, Part 3: Scaling Message Delivery

In [part two](https://bravenewgeek.com/building-a-distributed-log-from-scratch-part-2-data-replication/) of this series we discussed data replication within the context of a distributed log and how it relates to high availability. Next, we’ll look at what it takes to scale the log such that it can handle non-trivial workloads.

### **Data Scalability**

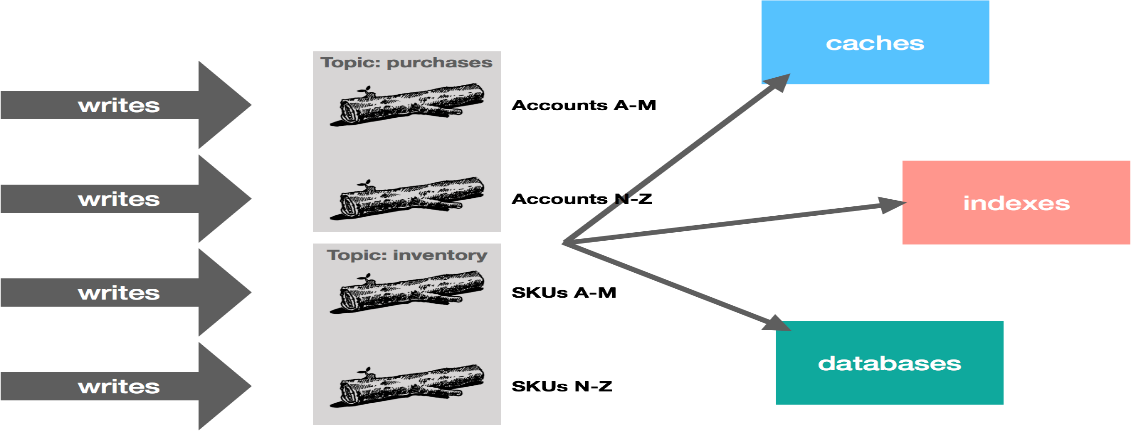
A key part of scaling any kind of data-intensive system is the ability to partition the data. Partitioning is how we can scale a system linearly, that is to say we can handle more load by adding more nodes. We make the system horizontally scalable.

Kafka was designed this way from the beginning. Topics are partitioned and ordering is only guaranteed within a partition. For example, in an e-commerce application, we might have two topics, purchases and inventory, each with two partitions. These partitions allow us to distribute reads and writes across a set of brokers. In Kafka, the log is actually the partition.

[](https://bravenewgeek.com/wp-content/uploads/2018/01/log_partitioning.png)

The challenge with this is how we partition the data. We might distribute data using round robin, in effect randomly distributing it. The problem with this is we lose out on ordering, which is an important characteristic of the log. For example, imagine we have add and remove inventory operations. With random partitioning, we might end up with a remove followed by an add getting processed if they’re placed in different partitions. However, if they’re placed in the same partition, we know they will be ordered correctly from the perspective of the publisher.

We could also distribute by hashing a key and sending all writes with the same keys to the same partitions or some custom partitioning strategy along these lines. Continuing with our example, we might partition purchases by account name and inventory by SKU. This way, all purchase operations by the same account are ordered, as are all inventory operations pertaining to the same SKU. The diagram below shows a (naive) custom strategy that partitions topics by ranges based on the account and SKU.

[](https://bravenewgeek.com/wp-content/uploads/2018/01/log_partitioning_strategy.png)

The important point here is that how you partition your data is largely dependent on your application and its usage patterns, but partitioning is a critical part of scalability. It allows you to scale your workload processing by dividing up responsibilities, which in turn, allows you to throw more resources at the problem in a tractable way.

One of NATS Streaming’s shortcomings, in my opinion, is that it doesn’t currently offer a good story around partitioning. Channels are totally ordered, essentially making them the equivalent of a Kafka partition. The workaround is to partition among multiple channels at the application level. To some, this is a benefit because it’s conceptually simpler than Kafka, but Kafka was designed as such because scalability was a key design goal from day one.

### **Consumer Scalability**

One challenge with the log is the problem of high fan-out. Specifically, how do we scale to a large number of consumers? In Kafka and NATS Streaming, reads (and writes) are only served by the leader. Similarly, Amazon Kinesis supports up to only five reads per second per shard (a shard is Kinesis’ equivalent of a partition). Thus, if we have five consumers reading from the same shard, we’ve already hit our fan-out limit. The thought is to partition your workload to increase parallelism and/or daisy chain streams to increase fan-out. But if we are trying to do very high fan-out, e.g. to thousands of IoT devices, neither of these are ideal solutions. Not all use cases may lend themselves to partitioning (though one can argue this is just a sign of poor architecting), and chaining up streams (or in Kafka nomenclature, topics) tends to be kludgey.

However, we can make the following observation: with an immutablelog, there are no stale or phantom reads. Unlike a database, we can loosen our requirements a bit. Whereas a database is typically mutable, with a log, we’re only appending things. From a consumer’s perspective, a replica is either up-to-date with the leader or in the process of catching up, but in either case, if we read all of the records, we should end up in the same state. Immutability, at least in theory, should make it “easy” to scale to a large number of consumers because we don’t have to read from the leader to get correct results (ignoring log compaction and other “mutable” operations), so long as we’re okay with strong eventual consistency with respect to tailing the log.

In NATS Streaming, with Raft, we could simply allow followers to serve reads and scale reads by increasing the size of the cluster, but this would impact performance because the quorum size would also increase. Instead, we can use “non-voters” to act as read replicas and balance consumers among them. These read replicas do not participate in quorum or leader election, they simply receive committed log entries. In effect, this is the daisy chaining of streams mentioned earlier but done implicitly by the system. This is an otherwise common pattern for increasing consumer fan-out in Kinesis but is usually done in an ad hoc, Rube Goldberg-esque fashion. Note that, in the case of NATS Streaming, this isn’t quite as simple as it sounds due to the delivery mechanism used, which we’ll describe next.

### **Push vs. Pull**

In Kafka, consumers pull data from brokers. In NATS Streaming, brokers push data to consumers. Kafka’s [documentation](http://kafka.apache.org/documentation.html#design_pull) describes this design decision in detail. The key factor largely comes down to flow control. With push, flow control needs to be explicit to deal with diverse consumers. Different consumers will consume at different rates, so the broker needs to be aware of this so as not to overwhelm a consumer.

There are obvious advantages and disadvantages to both approaches. With push, it can be a tricky balance to ensure full utilization of the consumer. We might use a backoff protocol like [additive increase/multiplicative decrease](https://en.wikipedia.org/wiki/Additive_increase/multiplicative_decrease), widely known for its use in TCP congestion control, to optimize utilization. NATS Streaming, like many other messaging systems, implements flow control by using acks. Upon receiving a message, consumers ack back to the server, and the server tracks the in-flight messages for each consumer. If that number goes above a certain threshold, the server will stop delivery until more acks are received. There is a similar flow-control mechanism between the publisher and the server. The trade-off here is the server needs to do some bookkeeping, which we’ll get to in a bit. With a pull-based system, flow control is implicit. Consumers simply go at their own pace, and the server doesn’t need to track anything. There is much less complexity with this.

Pull-based systems lend themselves to aggressive batching. With push, we must decide whether to send a message immediately or wait to accumulate more messages before sending. This is a decision pertaining to latency versus throughput. Push is often viewed as an optimization for latency, but if we’re tuning for low latency, we send messages one at a time only for them to end up being buffered on the consumer anyway. With pull, the consumer fetches all available messages after its current position in the log, which basically removes the guesswork around tuning batching and latency.

There are API implications with this decision too, particularly from an ergonomics and complexity perspective. Kafka clients tend to be “thick” and have a lot of complexity. That is, they do a lot because the broker is designed to be simple. That’s my guess as to why there are so few native client libraries up to par with the Java client. NATS Streaming clients, on the other hand, are relatively “thin” because the server does more. We end up just pushing the complexity around based on our design decisions, but one can argue that the [smart client and dumb server](https://bravenewgeek.com/smart-endpoints-dumb-pipes/) is a more scalable approach. We’ll go into detail on that in the next installment of this series.

Circling back on consumer scalability, the fact that NATS Streaming uses a push-based model means we can’t simply setup read replicas and balance consumers among them. Instead, we would need to partition consumers among the replicas so that each server is responsible for pushing data to a subset of consumers. The increased complexity over pull becomes immediately apparent here.

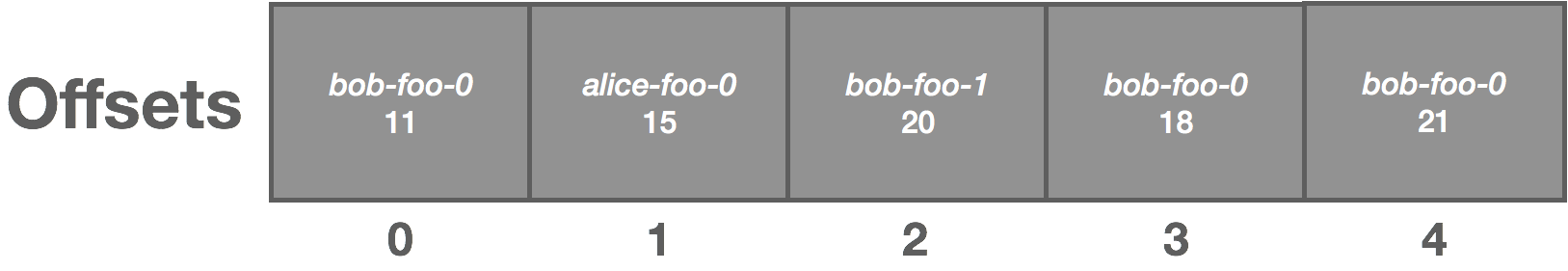
### **Bookkeeping**

There are two ways to track position in the log: have the server track it for consumers or have consumers track it themselves. Again, there are trade-offs with this, namely between API simplicity, server complexity, performance, and scalability. NATS Streaming tracks subscription positions for consumers. This means consumers can come and go as they like and pick back up where they left off easily. Before NATS Streaming supported clustering, this made a lot of sense because the bookkeeping was all in one server. But with clustering, this data must be replicated just like the message data, which poses a performance challenge.

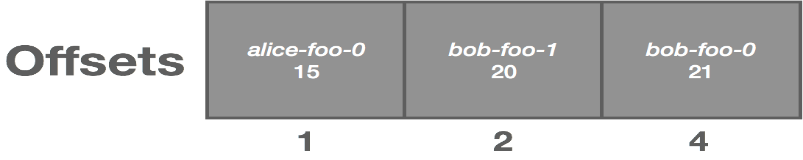
The alternative is to punt the problem to the consumer. But also keep in mind that consumers might not have access to fast stable storage, such as with an IoT device or ephemeral container. Is there a way we can split the difference?

We can store the offsets themselves directly in the log. As of 0.9, this is what Kafka does. Before that, clients had to manage offsets themselves or store them in ZooKeeper. This forced a dependency on ZooKeeper for clients but also posed a major bottleneck since ZooKeeper is relatively low throughput. But by storing offsets in the log, they are treated just like any other write to a Kafka topic, which scales quite well (offsets are stored in an internal Kafka topic called \_\_consumer\_offsets partitioned by consumer group; there is also a special read cache for speeding up the read path).

Clients periodically checkpoint their offset to the log. We then use log compaction to retain only the latest offsets. Log compaction works by rewriting the log to retain only the latest message for a given key. On recovery, clients fetch the latest offset from the log. The important part here is we need to structure our keys such that compaction retains the latest offset for each unique consumer. For example, we might structure it as consumer-topic-partition. We end up with something resembling the following, where the message value is the offset:

[](https://bravenewgeek.com/wp-content/uploads/2018/01/log_offset_storage.png)

The above log is uncompacted. Once compacted, it becomes the following:

[](https://bravenewgeek.com/wp-content/uploads/2018/01/log_offset_storage_compacted.png)

Note that compaction violates some of our previous assumptions around the immutability of the log, but that’s for another discussion.

There are a number of advantages to this approach. We get fault-tolerance and durability due to the fact that our log is already fault-tolerant and durable as designed earlier. We get consistent reads again due to our replication scheme. Unlike ZooKeeper, we get high write throughput. And we reuse existing structures, so there’s less server complexity. We’re just reusing the log, there aren’t really any major new codepaths.

Interestingly, the bookkeeping needed for flow control in push-based systems—such as acks in NATS Streaming—serves much the same purpose as offset tracking inpull-based systems, since it needs to track position. The difference comes when we allow out-of-order processing. If we don’t allow it, then acks are simply a high-water mark that indicate the client is “this far” caught up. The problem with push is we also have to deal with redeliveries, whereas with pull they are implicitly handled by the client.  If we do allow out-of-order processing, then we need to track individual, in-flight messages, which is what per-message acks allow us to do. In this case, the system starts to look less like a log and more like a message queue. This makes push even more complicated.

The nice thing about reusing the log to track offsets is it greatly reduces the amount of code and complexity needed. Since NATS Streaming allows out-of-order processing, it uses a separate acking subsystem which otherwise has the same requirements as an offset-tracking subsystem.

In [part four](https://bravenewgeek.com/building-a-distributed-log-from-scratch-part-4-trade-offs-and-lessons-learned/) of this series, we will discuss some of the key trade-offs involved with implementing a distributed log and some lessons learned while building NATS Streaming.

# Building a Distributed Log from Scratch, Part 4: Trade-Offs and Lessons Learned

In [part three](https://bravenewgeek.com/building-a-distributed-log-from-scratch-part-3-scaling-message-delivery/) of this series we talked about scaling message delivery in a distributed log. In part four, we’ll look at some key trade-offs involved with such systems and discuss a few lessons learned while building NATS Streaming.

### **Competing Goals**

There are a number of competing goals when building a distributed log (these goals also extend to many other types of systems). Recall from [part one](https://bravenewgeek.com/building-a-distributed-log-from-scratch-part-1-storage-mechanics/) that our key priorities for this type of system are performance, high availability, and scalability. The preceding parts of this series described at various levels how we can accomplish these three goals, but astute readers likely noticed that some of these things conflict with one another.

It’s easy to make something fast if it’s not fault-tolerant or scalable. If our log runs on a single server, our only constraints are how fast we can send data over the network and how fast the disk I/O is. And this is how a lot of systems, including many databases, tend to work—not only because it performs well, but because it’s simple. We can make these types of systems fault-tolerant by introducing a standby server and allowing clients to failover, but there are a couple issues worth mentioning with this.

With data systems, such as a log, high availability does not just pertain to continuity of service, but also availability of data. If I write data to the system and the system acknowledges that, that data should not be lost in the event of a failure. So with a standby server, we need to ensure data is replicated to avoid data loss (otherwise, in the context of a message log, we must relax our requirement of guaranteed delivery).

NATS Streaming initially shipped as a single-node system, which raised immediate concerns about production-readiness due to a single point of failure. The first step at trying to address some of these concerns was to introduce a fault-tolerance mode whereby a group of servers would run and only one would run as the active server. The active server would obtain an exclusive lock and process requests. Upon detecting a failure, standby servers would attempt to obtain the lock and become the active server.

Aside from the [usual issues with distributed locks](https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html), this design requires a shared storage layer. With NATS Streaming, this meant either a shared volume, such as Gluster or EFS, or a shared MySQL database. This poses a performance challenge and isn’t particularly “cloud-native” friendly. Another issue is data is not replicated unless done so out-of-band by the storage layer. When we add in data replication, performance is hamstrung even further. But this was a quick and easy solution that offered some solace with respect to a SPOF (disclosure: I was not involved with NATS or NATS Streaming at this time). The longer term solution was to provide first-class clustering and data-replication support, but sometimes it’s more cost effective to provide fast recovery of a single-node system.

Another challenge with the single-node design is scalability. There is only so much capacity that one node can handle. At a certain point, scaling out becomes a requirement, and so we start partitioning. This is a common technique for relational databases where we basically just run multiple databases and divide up the data by some key. NATS Streaming is no different as it offers a [partitioning story](https://github.com/nats-io/nats-streaming-server#partitioning) for dividing up channels between servers. The trouble with partitioning is it complicates things as it typically requires cooperation from the application. To make matters worse, NATS Streaming does not currently offer partitioning at the channel level, which means if a single topic has a lot of load, the solution is to manually partition it into multiple channels at the application level. This is why Kafka chose to partition its topics by default.

So performance is at odds with fault-tolerance and scalability, but another factor is what I call simplicity of mechanism. That is, the simplicity of the design plays an important role in the performance of a system. This plays out at multiple levels. We saw that, at an architectural level, using a simple, single-node design performs best but falls short as a robust solution. In [part one](https://bravenewgeek.com/building-a-distributed-log-from-scratch-part-1-storage-mechanics/), we saw that using a simple file structure for our log allowed us to take advantage of the hardware and operating system in terms of sequential disk access, page caching, and zero-copy reads. In [part two](https://bravenewgeek.com/building-a-distributed-log-from-scratch-part-2-data-replication/), we made the observation that we can treat the log itself as a replicated WAL to solve the problem of data replication in an efficient way. And in [part three](https://bravenewgeek.com/building-a-distributed-log-from-scratch-part-3-scaling-message-delivery/), we discussed how a simple pull-based model can reduce complexity around flow control and batching.

At the same time, simplicity of “UX” makes performance harder. When I say UX, I mean the ergonomics of the system and how easy it is to use, operate, etc. NATS Streaming initially optimized for UX, which is why it fills an interesting space. Simplicity is a core part of the NATS philosophy, so it caught a small mindshare with developers frustrated or overwhelmed by Kafka. There is appetite for a “Kafka lite,” something which serves a similar purpose to Kafka but without all the bells and whistles and probably not targeted at large enterprises—a classic Innovator’s Dilemma to be sure.

NATS Streaming tracks consumer positions automatically, provides simple APIs, and uses a simple push-based protocol. This also means building a client library is a much less daunting task. The downside is the server needs to do more work. With a single node, as NATS Streaming was initially designed, this isn’t much of a problem. Where it starts to rear its head is when we need to replicate that state across a cluster of nodes. This has important implications with respect to performance and scale. Smart middleware has a natural tendency to become [more complex, more fragile, and slower](https://www.youtube.com/watch?v=JHQlA_tB10c). The [end-to-end principle](https://bravenewgeek.com/from-the-ground-up-reasoning-about-distributed-systems-in-the-real-world/) attests to this. Amusingly, NATS Streaming was originally named STAN because it’s the opposite of NATS, a fast and simple messaging system with minimal guarantees.

Simplicity of mechanism tends to simply push complexity around in the system. For example, NATS Streaming provides an ergonomic API to clients by shifting the complexity to the server. Kafka scales and performs exceptionally well by shifting the complexity to other parts of the system, namely the client and ZooKeeper.

Scalability and fault-tolerance are equally at odds with simplicity for reasons mostly described above. The important point here is that these cannot be an afterthought. As I learned while implementing clustering in NATS Streaming, you can’t cleanly and effectively bolt onfault-tolerance onto an existing complex system. One of the laws of [Systemantics](https://en.wikipedia.org/wiki/Systemantics) comes to mind here: “A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over, beginning with a working simple system.” Scalability and fault-tolerance need to be designed from day one.

Lastly, availability is inherently at odds with consistency. This is simply the [CAP theorem](https://bravenewgeek.com/cap-and-the-illusion-of-choice/). Guaranteeing strong consistency requires a quorum when replicating data, which hinders availability and performance. The key here is minimize what you need to replicate or relax your requirements.

### **Lessons Learned**

The section above already contains several lessons learned in the process of working on NATS Streaming and implementing clustering, but I’ll capture a few important ones here.

First, distributed systems are complex enough. Simple is usually better—and faster. Again, we go back to the laws of systems here: “A complex system that works is invariably found to have evolved from a simple system that works.”

Second, lean on existing work. A critical part to delivering clustering rapidly was sticking with Raft and an existing Go implementation for leader election and data replication. There was considerable time spent designing a proprietary solution before I joined which still had edge cases not fully thought through. Not only is Raft off the shelf, it’s provably correct (implementation bugs notwithstanding). And following from the first lesson learned, start with a solution that works before worrying about optimization. It’s far easier to make a correct solution fast than it is to make a fast solution correct. Don’t roll your own coordination protocol if you don’t need to (and chances are you don’t need to).

There are probably edge cases for which you haven’t written tests. There are many failures modes, and you can only write so many tests. Formal methods and property-based testing can help a lot here. Similarly, chaos and fault-injection testing such as Kyle Kingsbury’s [Jepsen](https://github.com/jepsen-io/jepsen) help too.

Lastly, be honest with your users. Don’t try to be everything to everyone. Instead, be explicit about design decisions, trade-offs, guarantees, defaults, etc. If there’s one takeaway from Kyle’s [Jepsen series](https://jepsen.io/analyses) it’s that many vendors are dishonest in their documentation and marketing. MongoDB became infamous for having [unsafe defaults](https://aphyr.com/posts/322-call-me-maybe-mongodb-stale-reads) and [implementation](https://aphyr.com/posts/284-call-me-maybe-mongodb) [issues](https://jepsen.io/analyses/mongodb-3-4-0-rc3) early on, most likely because they make benchmarks look much more impressive.

In [part five](https://bravenewgeek.com/building-a-distributed-log-from-scratch-part-5-sketching-a-new-system/) of this series, we’ll conclude by outlining the design for a new log-based system that draws from ideas in the previous entries in the series.

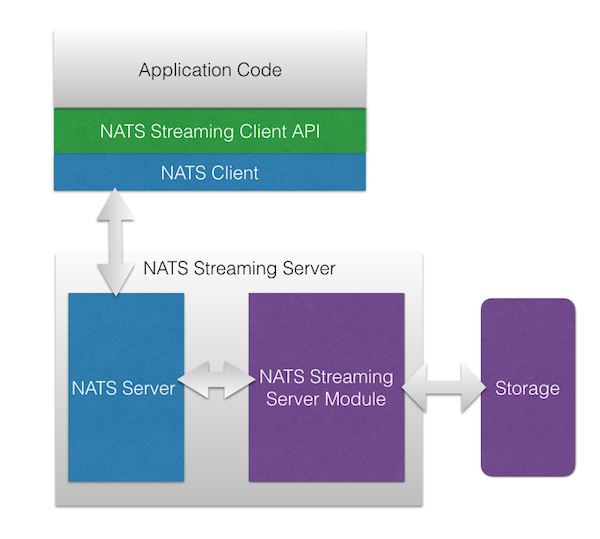
# Building a Distributed Log from Scratch, Part 5: Sketching a New System

In [part four](https://bravenewgeek.com/building-a-distributed-log-from-scratch-part-4-trade-offs-and-lessons-learned/) of this series we looked at some key trade-offs involved with a distributed log implementation and discussed a few lessons learned while building NATS Streaming. In this fifth and final installment, we’ll conclude by outlining the design for a new log-based system that draws from the previous entries in the series.

### **The Context**

For context, [NATS](https://nats.io/) and [NATS Streaming](https://nats.io/documentation/streaming/nats-streaming-intro/) are two different things. NATS Streaming is a log-based streaming system built on top of NATS, and NATS is a lightweight pub/sub messaging system. NATS was originally built (and then open sourced) as the control plane for Cloud Foundry. NATS Streaming was built in response to the community’s ask for higher-level guarantees—durability, at-least-once delivery, and so forth—beyond what NATS provided. It was built as a separate layer on top of NATS. I tend to describe NATS as a dial tone—ubiquitous and always on—perfect for “online” communications. NATS Streaming is the voicemail—leave a message after the beep and someone will get to it later. There are, of course, more nuances than this, but that’s the gist.

The key point here is that NATS and NATS Streaming are distinct systems with distinct protocols, distinct APIs, and distinct client libraries. In fact, NATS Streaming was designed to essentially act as a client to NATS. As such, clients don’t talk to NATS Streaming directly, rather all communication goes through NATS. However, the NATS Streaming binary can be configured to either embed NATS or point to a standalone deployment. The architecture is shown below in a diagram borrowed from the [NATS website](https://nats.io/documentation/streaming/nats-streaming-intro/).

[](https://bravenewgeek.com/wp-content/uploads/2018/01/nats-streaming-arch.png)

Architecturally, this makes a lot of sense. It supports the [end-to-end principle](https://en.wikipedia.org/wiki/End-to-end_principle) in that we layer on additional functionality rather than bake it in to the underlying infrastructure. After all, [we can always build stronger guarantees on top, but we can’t always remove them from below](https://bravenewgeek.com/what-you-want-is-what-you-dont-understanding-trade-offs-in-distributed-messaging/). This particular architecture, however, introduces a few challenges (disclosure: while I’m still a fan, I’m no longer involved with the NATS project and the NATS team is aware of these problems and no doubt working to address many of them).

First, there is no “cross-talk” between NATS and NATS Streaming, meaning messages published to NATS are not visible in NATS Streaming and vice versa. Again, they are two completely separate systems that just share the same infrastructure. This means we’re not really layering on message durability to NATS, we’re just exposing a new system which provides these semantics.

Second, because NATS Streaming runs as a “sidecar” to NATS and all of its communication runs through NATS, there is an inherent bottleneck at the NATS connection. This may only be a theoretical limit, but it precludes certain optimizations like using sendfile to do zero-copy reads of the log. It also means we rely on timeouts even in cases where the server could send a response immediately, such as when there is no leader elected for the cluster.

Third, NATS Streaming currently lacks a compelling story around linear scaling other than running multiple clusters and partitioning channels among them at the application level. With respect to scaling a single channel, the only alternative at the moment is to partition it into multiple channels at the application level. My hope is that as clustering matures, this will too.

Fourth, without extending its protocol, NATS Streaming’s authorization is intrinsically limited to the authorization provided by NATS since all communication goes through it. In and of itself, this isn’t a problem. NATS supports [multi-user authentication and subject-level permissions](https://nats.io/documentation/server/gnatsd-authorization/), but since NATS Streaming uses an opaque protocol atop NATS, it’s difficult to setup proper ACLs at the streaming level. Of course, many layered protocols support authentication, e.g. HTTP atop TCP. For example, the NATS Streaming protocol could carry authentication tokens or session keys, but it currently does not do this.

Fifth, NATS Streaming does not support [wildcard semantics](https://nats.io/documentation/internals/nats-protocol/), which—at least in my opinion—is a large selling point of NATS and, as a result, something users have come to expect. Specifically, NATS supports two wildcards in subject subscriptions: asterisk (\*) which matches any token in the subject (e.g. foo.\* matches foo.bar, foo.baz, etc.) and full wildcard (>) which matches one or more tokens at the tail of the subject (e.g. foo.> matches foo.bar, foo.bar.baz, etc.). Note that this limitation in NATS Streaming is not directly related to the overall architecture but more in how we design the log.

More generally, clustering and data replication was more of an afterthought in NATS Streaming. As we discussed in [part four](https://bravenewgeek.com/building-a-distributed-log-from-scratch-part-4-trade-offs-and-lessons-learned/), it’s hard to add this after the fact. Combined with the APIs NATS Streaming provides (which do flow control and track consumer state), this creates a lot of complexity in the server.

### **A New System**

I wasn’t involved much with NATS Streaming beyond the clustering implementation. However, from that work—and through my own use of NATS and from discussions I’ve had with the community—I’ve thought about how I would build something like it if I were to start over. It would look a bit different from NATS Streaming and Kafka, yet also share some similarities. I’ve dubbed this theoretical system Jetstream (update: this is now [Liftbridge](https://github.com/liftbridge-io/liftbridge)), though I’ve yet to actually build anything beyond small prototypes. It’s a side project of mine I hope to get to at some point.

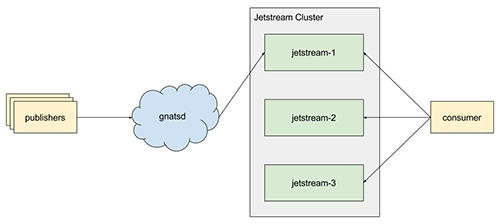
Core NATS has a strong community with solid mindshare, but NATS Streaming doesn’t fully leverage this since it’s a new silo. Jetstream aims to address the above problems starting from a simple proposition: many people are already using NATS today and simply want streaming semantics for what they already have. However, we must also acknowledge that other users are happy with NATS as it currently is and have no need for additional features that might compromise simplicity or performance. This was a deciding factor in choosing not to build NATS Streaming’s functionality directly into NATS.

Like NATS Streaming, Jetstream is a separate component which acts as a NATS client. Unlike NATS Streaming, it augments NATS as opposed to implementing a wholly new protocol. More succinctly, Jetstream is a durable stream augmentation for NATS. Next, we’ll talk about how it accomplishes this by sketching out a design.

#### Cross-Talk

In NATS Streaming, the log is modeled as a channel. Clients create channels implicitly by publishing or subscribing to a topic (called a subject in NATS). A channel might be foo but internally this is translated to a NATS pub/sub subject such as \_STAN.pub.foo. Therefore, while NATS Streaming is technically a client of NATS, it’s done so just to dispatch communication between the client and server. The log is implemented on top of plain pub/sub messaging.

Jetstream is merely a consumer of NATS. In it, the log is modeled as a stream. Clients create streams explicitly, which are subscriptions to NATS subjects that are sequenced, replicated, and durably stored. Thus, there is no “cross-talk” or internal subjects needed because Jetstream messages are NATS messages. Clients just publish their messages to NATS as usual and, behind the scenes, Jetstream will handle storing them in a log. In some sense, it’s just an audit log of messages flowing through NATS.

[](https://bravenewgeek.com/wp-content/uploads/2018/01/jetstream.png)

With this, we get wildcards for free since streams are bound to NATS subjects. There are some trade-offs to this, however, which we will discuss in a bit.

#### Performance

Jetstream does not track subscription positions. It is up to consumers to track their position in a stream or, optionally, store their position in a stream (more on this later). This means we treat a stream as a simple log, allowing us to do fast, sequential disk I/O and minimize replication and protocol chatter as well as code complexity.

Consumers connect directly to Jetstream using a pull-based socket API. The log is stored in the manner described in [part one](https://bravenewgeek.com/building-a-distributed-log-from-scratch-part-1-storage-mechanics/). This enables us to do zero-copy reads from a stream and other important optimizations which NATS Streaming is precluded from doing. It also simplifies things around flow control and batching as we discussed in [part three](https://bravenewgeek.com/building-a-distributed-log-from-scratch-part-3-scaling-message-delivery/).

#### Scalability

Jetstream is designed to be clustered and horizontally scalable from the start. We make the observation that NATS is already efficient at routing messages, particularly with high consumer fan-out, and provides clustering of the interest graph. Streams provide the unit of storage and scalability in Jetstream.

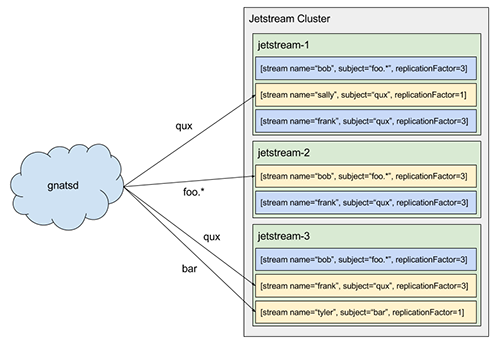
A stream is a named log attached to a NATS subject. Akin to a partition in Kafka, each stream has a replicationFactor, which controls the number of nodes in the Jetstream cluster that participate in replicating the stream, and each stream has a leader. The leader is responsible for receiving messages from NATS, sequencing them, and performing replication (NATS provides per-publisher message ordering).

Like Kafka’s controller, there is a single metadata leader for a Jetstream cluster which is responsible for processing requests to create or delete streams. If a request is sent to a follower, it’s automatically forwarded to the leader. When a stream is created, the metadata leader selects replicationFactor nodes to participate in the stream (initially, this selection is random but could be made more intelligent, e.g. selecting based on current load) and replicates the stream to all nodes in the cluster. Once this replication completes, the stream has been created and its leader begins processing messages. This means NATS messages are not stored unless there is a stream matching their subject (this is the trade-off to support wildcards, but it also means we don’t waste resources storing messages we might not care about). This can be mitigated by having publishers ensure a stream exists before publishing, e.g. at startup.

There can exist multiple streams attached to the same NATS subject or even subjects that are semantically equivalent, e.g. foo.bar and foo.\*. Each of these streams will receive a copy of the message as NATS handles this fan-out. However, the stream name is unique within a given subject. For example, creating two streams for the subject foo.bar named foo and bar, respectively, will create two streams which will independently sequence all of the messages on the NATS subject foo.bar, but attempting to create two streams for the same subject both named foo will result in creating just a single stream (creation is idempotent).

With this in mind, we can scale linearly with respect to consumers—covered in [part three](https://bravenewgeek.com/building-a-distributed-log-from-scratch-part-3-scaling-message-delivery/)—by adding more nodes to the Jetstream cluster and creating more streams which will be distributed among the cluster. This has the advantage that we don’t need to worry about partitioning so long as NATS is able to withstand the load (there is also an assumption that we can ensure reasonable balance of stream leaders across the cluster). We’ve basically split out message routing from storage and consumption, which allows us to scale independently.

Additionally, streams can join a named consumer group. This, in effect, partitions a NATS subject among the streams in the group, again covered in [part three](https://bravenewgeek.com/building-a-distributed-log-from-scratch-part-3-scaling-message-delivery/), allowing us to create competing consumers for load-balancing purposes. This works by using [NATS queue subscriptions](https://nats.io/documentation/concepts/nats-queueing/), so the downside is partitioning is effectively random. The upside is consumer groups don’t affect normal streams.

[](https://bravenewgeek.com/wp-content/uploads/2018/01/jetstream_cluster.png)

#### Compaction and Offset Tracking

Streams support multiple log-compaction rules: time-based, message-based, and size-based. As in Kafka, we also support a fourth kind: key compaction. This is how offset storage will work, which was described in [part three](https://bravenewgeek.com/building-a-distributed-log-from-scratch-part-3-scaling-message-delivery/), but it also enables some other interesting use cases like [KTables](https://docs.confluent.io/current/streams/concepts.html" \l "ktable) in Kafka Streams.

As discussed above, messages in Jetstream are simply NATS messages. There is no special protocol needed for Jetstream to process messages. However, publishers can choose to optionally “enhance” their messages by providing additional metadata and serializing their messages into envelopes. The envelope includes a special cookie Jetstream uses to detect if a message is an envelope or a simple NATS message (if the cookie is present by coincidence and envelope deserialization fails, we fall back to treating it as a normal message).

One of the metadata fields on the envelope is an optional message key. A stream can be configured to compact by key. In this case, it retains only the last message for each key (if no key is present, the message is always retained).

Consumers can optionally store their offsets in Jetstream (this can also be transparently managed by a client library similar to Kafka’s high-level consumer). This works by storing offsets in a stream keyed by consumer. A consumer (or consumer library) publishes their latest offset. This allows them to later retrieve their offset from the stream, and key compaction means Jetstream will only retain the latest offset for each consumer. For improved performance, the client library should only periodically checkpoint this offset.

#### Authorization

Because Jetstream is a separate server which is merely a consumer of NATS, it can provide ACLs or other authorization mechanisms on streams. A simple configuration might be to restrict NATS access to Jetstream and configure Jetstream to only allow access to certain subjects. There is more work involved because there is a separate access-control system, but this gives greater flexibility by separating out the systems.

#### At-Least Once Delivery

To ensure at-least-once delivery of messages, Jetstream relies on replication and publisher acks. When a message is received on a stream, it’s assigned an offset by the leader and then replicated. Upon a successful replication, the stream publishes an ack to NATS on the reply subject of the message, if present (the reply subject is a part of the [NATS message protocol](https://nats.io/documentation/internals/nats-protocol/)).

There are two implications with this. First, if the publisher doesn’t care about ensuring its message is stored, it need not set a reply subject. Second, because there are potentially multiple (or no) streams attached to a subject (and creation/deletion of streams is dynamic), it’s not possible for the publisher to know how many acks to expect. This is a trade-off we make for enabling subject fan-out and wildcards while remaining scalable and fast. We make the assertion that if guaranteed delivery is important, the publisher should be responsible for determining the destination streams a priori. This allows attaching streams to a subject for use cases that do not require strong guarantees without the publisher having to be aware. Note that this might be an area for future improvement to increase usability, such as storing streams in a registry. However, this is akin to other similar systems, like Kafka, where you must first create a topic and then you publish to that topic.

One caveat to this is if there are existing application-level uses of the reply subject on NATS messages. That is, if other systems are already publishing replies, then Jetstream will overload this. The alternative would be to require the envelope, which would include a canonical reply subject for acks, for at-least-once delivery. Otherwise we would need a change to the NATS protocol itself.

#### Replication Protocol

For metadata replication and leadership election, we rely on Raft. However, for replication of streams, rather than using Raft or other quorum-based techniques, we use a technique similar to Kafka as described in [part two](https://bravenewgeek.com/building-a-distributed-log-from-scratch-part-2-data-replication/).

For each stream, we maintain an in-sync replica set (ISR), which is all of the replicas currently up to date (at stream creation time, this is *all* of the replicas). During replication, the leader writes messages to a WAL, and we only wait on replicas in the ISR before committing. If a replica falls behind or fails, it’s removed from the ISR. If the leader fails, any replica in the ISR can take its place. If a failed replica catches back up, it rejoins the ISR. The general stream replication process is as follows:

1. Client creates a stream with a replicationFactor of n.
2. Metadata leader selects n replicas to participate and one leader at random (this comprises the initial ISR).
3. Metadata leader replicates the stream via Raft to the entire cluster.
4. The nodes participating in the stream initialize it, and the leader subscribes to the NATS subject.
5. The leader initializes the high-water mark (HW) to 0. This is the offset of the last committed message in the stream.
6. The leader begins sequencing messages from NATS and writes them to the log uncommitted.
7. Replicas consume from the leader’s log to replicate messages to their own log. We piggyback the leader’s HW on these responses, and replicas periodically checkpoint the HW to stable storage.
8. Replicas acknowledge they’ve replicated the message.
9. Once the leader has heard from the ISR, the message is committed and the HW is updated.

Note that clients only see committed messages in the log. There are a variety of failures that can occur in the replication process. A few of them are described below along with how they are mitigated.

If a follower suspects that the leader has failed, it will notify the metadata leader. If the metadata leader receives a notification from the majority of the ISR within a bounded period, it will select a new leader for the stream, apply this update to the Raft group, and notify the replica set. These notifications need to go through Raft as well in the event of a metadata leader failover occurring at the same time as a stream leader failure. Committed messages are always preserved during a leadership change, but uncommitted messages could be lost.

If the stream leader detects that a replica has failed or fallen too far behind, it removes the replica from the ISR by notifying the metadata leader. The metadata leader replicates this fact via Raft. The stream leader continues to commit messages with fewer replicas in the ISR, entering an under-replicated state.

When a failed replica is restarted, it recovers the latest HW from stable storage and truncates its log up to the HW. This removes any potentially uncommitted messages in the log. The replica then begins fetching messages from the leader starting at the HW. Once the replica has caught up, it’s added back into the ISR and the system resumes its fully replicated state.

If the metadata leader fails, Raft will handle electing a new leader. The metadata Raft group stores the leader and ISR for every stream, so failover of the metadata leader is not a problem.

There are a few other corner cases and nuances to handle, but this covers replication in broad strokes. We also haven’t discussed how to implement failure detection (Kafka uses ZooKeeper for this), but we won’t prescribe that here.

### **Wrapping Up**

This concludes our series on building a distributed log that is fast, highly available, and scalable. In [part one](https://bravenewgeek.com/building-a-distributed-log-from-scratch-part-1-storage-mechanics/), we introduced the log abstraction and talked about the storage mechanics behind it. In [part two](https://bravenewgeek.com/building-a-distributed-log-from-scratch-part-2-data-replication/), we covered high availability and data replication. In [part three](https://bravenewgeek.com/building-a-distributed-log-from-scratch-part-3-scaling-message-delivery/), we we discussed scaling message delivery. In [part four](https://bravenewgeek.com/building-a-distributed-log-from-scratch-part-4-trade-offs-and-lessons-learned/), we looked at some trade-offs and lessons learned. Lastly, in part five, we outlined the design for a new log-based system that draws from the previous entries in the series.

The goal of this series was to learn a bit about the internals of a log abstraction, to learn how it can achieve the three priorities described earlier, and to learn some applied distributed systems theory. Hopefully you found it useful or, at the very least, interesting.

If you or your company are looking for help with system architecture, performance, or scalability, contact [Real Kinetic](https://realkinetic.com/).