Hazelcast 3.8.3

2017-10-06

Hazelcast is a distributed in-memory data grid, providing shared data structures for distributed systems. We show that many of Hazelcast's distributed data structures are unsafe in the presence of network partitions: updates to maps can be lost, unique IDs may not be unique, atomic objects are not atomic, locks aren't exclusive, and queues can forget about enqueued elements. Stale and dirty reads are also possible in most types. We do identify a way to build CRDTs on top of Hazelcast, which prevents the loss of acknowledged updates so long as operations do not depend on order. Despite documentation alluding to these risks, Hazelcast users rely on Hazelcast in risky ways. This work was performed independently, without compensation, and conducted in accordance with the Jepsen ethics policy.

1 Errata

2017-10-07: IMap is not the only datatype supporting merge: ICache is mergable as well.

2017-10-17: Java's AtomicReferences may only be sequential (and linearizable in certain cases, like CaS updates), rather than linearizable. The literature is somewhat unclear on this point.

2 Background

Hazelcast provides easy-to-use distributed data structures with rich, intuitive APIs and optional persistence. It is often deployed as a synchronization service for databases, caches, session stores, messaging buses, or for service discovery. Users may embed a full Hazelcast node directly into their JVM application, or use a lightweight network client (available in many languages) to talk to a dedicated Hazelcast cluster. Either way, Hazelcast offers transparent distribution for rich datatypes, with familiar Java APIs for sets, lists, maps, locks, semaphores, queues, atomic objects, id generators, and more.

Hazelcast's high-level documentation, and the names of these objects themselves, imply that operations on these objects provide certain safety guarantees. For instance, the features page claims that Hazelcast's AtomicReferences offers guaranteed atomic compare-and-set across a cluster, and the AtomicReference documentation confirms this claim. What the documentation for this datatype does *not* mention is that compare-and-set on AtomicReferences is not, in point of fact, atomic.

In section 26 of the manual, Network Partitioning - Split-Brain Syndrome, Hazelcast explains that in the event of a network partition, each component of the network continues to run independently. They go on to discuss a lost-update scenario when the MapStore persistence layer is enabled for Maps: both clusters could write conflicting entries to their backing database. The documentation does not mention that this problem also arises *without* a backing database—users could be forgiven for assuming that they won't experience lost updates if they use Hazelcast as a purely in-memory store.

The documentation goes on to discuss split-brain merging: for Maps (and only Maps; a footnote explains that other datatypes are not merged), conflicting values for the same key are merged using a configurable merge policy. The built-in policies are a pair of non-commutative heuristics (larger or smaller cluster wins), last-write-wins, or higher-hits-wins. As we have seen in prior Jepsen analyses, all of these techniques can result in the loss of committed updates. For all datatypes other than Maps, updates to the smaller cluster (or an arbitrary cluster if the split is even) are thrown away.

Hazelcast goes on to explain:

> Hazelcast's Split-Brain Protection enables you to specify the minimum cluster size

> required for operations to occur. This is achieved by defining and configuring a split-brain protection cluster quorum. If the cluster size is below the defined quorum, the operations are rejected and the rejected operations return a QuorumException to their callers.
>
> Your application continues its operations on the remaining operating cluster. Any application instances connected to the cluster with sizes below the defined quorum will be receiving exceptions which, depending on the programming and monitoring setup, should generate alerts. The key point is that rather than applications continuing in error with stale data, they are prevented from doing so.

This implies that we should be able to update Maps, Transactional Maps, Caches, Locks, and Queues safely, so long as we choose a majority quorum policy for those data structures. This is, unfortunately, not the case:

> It is normally seconds or tens of seconds before the cluster is adjusted to exclude unreachable members.... For this reason, there will be a time window between the network partitioning and the application of Split-Brain Protection.

That implies that operations on most Hazelcast datatype may be lost when the network is unreliable. In this analysis, we experimentally confirm this hypothesis, by testing the behavior of several Hazelcast datatypes when the network is allowed to fail.

3 Test Design

We use the Jepsen distributed systems testing library to evaluate Hazelcast's safety. We write a trivial server application which starts a Hazelcast instance on each of five nodes. We tune the Hazelcast configuration to speed up test times, making heartbeats more frequent and timeouts shorter. Cluster membership is fixed at startup, and all nodes are connected via direct TCP connections to every other node. We apply majority quorum constraints to every datatype that supports them.

Jepsen then uses the Hazelcast client library to connect to various nodes, and perform operations against Hazelcast datatypes. During these operations, we introduce network partitions lasting 15 seconds, followed by 30 seconds of full connectivity to allow the cluster to recover. Before performing any final reads to confirm safety, we allow Hazelcast 500 seconds of total network connectivity to fully heal[1], and merge any unmerged records. Once the test is complete, we analyze the history of operations to identify whether common-sense invariants on that datatype were preserved.

The operations we perform, and how we check for correctness, depend on the particular datatype being tested.

3.1 Locks

Hazelcast's feature list claims that locks provide guaranteed mutual exclusion across a cluster.

> If you lock using an ILock, the critical section that it guards is guaranteed to be executed by only one thread in the entire cluster. Even though locks are great for synchronization, they can lead to problems if not used properly.

Specifically, those problems include:

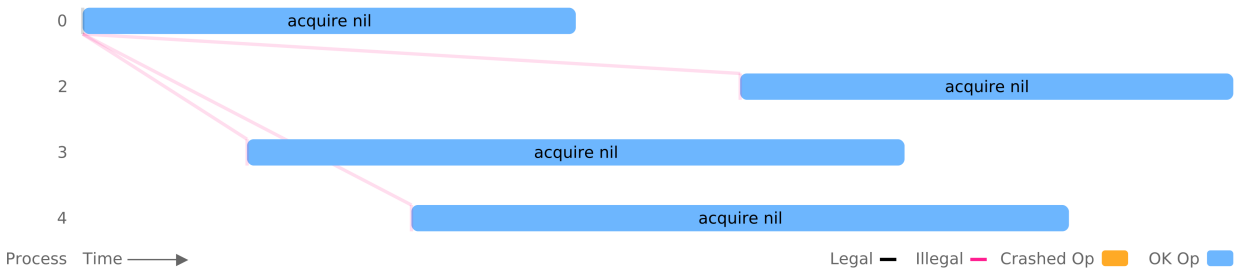
> In the split-brain scenario, the cluster behaves as if it were two different clusters. Since two separate clusters are not aware of each other, two members from different clusters can acquire the same lock.

So locks don't guarantee mutual exclusion—but the docs go on to point to split-brain protection, so perhaps there's a chance of safety. We'll set the lock to use majority quorums, so only a cluster with more than 1/2 of the nodes can acquire a lock safely. In addition, we prevent Hazelcast from releasing a lock before another client explicitly unlocks it by omitting the `leaseTime` arguments to `tryLock`.[2]

Our clients perform a sequence of alternating acquire and release operations, and verify that the resulting history is linearizable; e.g. no two processes can hold the same lock at the same time. This test fails reliably as soon as a partition occurs, as this example shows.

[1] 500 seconds is a fairly long time to wait for a cluster to become available again, but even with heartbeats and timeouts significantly lowered to improve fault detection and recovery time, Hazelcast routinely takes hundreds of seconds to recover from a network fault. Most datastores we've tested with Jepsen recover in tens of seconds; some as quickly as 1 second.

[2] Of course, this is not reasonable in a real distributed system, where clients may crash—all distributed lock services are fundamentally unsafe—but we can pretend for testing purposes.



Just after the start of a partition, several clients succeeded in acquiring a lock which was *already held*. This diagram shows the concurrent structure of lock operations. Time flows left to right, and each concurrent process is shown as a row. Blue bars show successful operations, and pink traces show illegal transitions between operations. Here, process 0 had acquired the lock, and processes 2, 3, and 4 went on to successfully acquire the lock *before* process 0 had released it. This is the opposite of what a lock is supposed to do.

Locks in Hazelcast are not really locks. They may provide mutual exclusion most of the time, but when a network failure occurs, multiple clients may hold the same lock concurrently. This occurs even in the absence of process crashes, without lock timeouts allowing Hazelcast to reclaim stale locks, and with majority quorum protection enabled.

3.2 Queues

Hazelcast queues provide a distributed equivalent to Java's BlockingQueue. The docs claim: "Using Hazelcast distributed queue, you can add an item in one machine and remove it from another one." The manual goes on to note that Queues support split-brain protection. What they fail to mention is that this is not a guarantee of correctness: you can put an item into a queue on one machine, Hazelcast will dutifully acknowledge that item, and then consign it to the silent void of `/dev/null`.

To test this, we perform a mixture of enqueues and dequeues on a single queue. All enqueues are unique integers, so we can map enqueues to dequeues. At the end of the test, we heal all network partitions, wait 500 seconds to allow the cluster time to recover, and have every client drain all remaining elements from the queue—which should ensure that every added element has been dequeued at least once. We then examine the history to see whether this is the case.[3]

In this 20 minute long run, roughly 1/4 of attempted enqueues appeared to succeed, and of those successfully enqueued messages, 2% were, in fact, silently discarded. The window for data loss is relatively small–only a few seconds at the start of each network partition—because we've aggressively tuned Hazelcast's heartbeats to very short intervals. With the default timeouts, quorum protection takes longer to kick in, and the window for message loss is larger.

```
{:valid? false,
 :lost
 #{1903 1952 1946 1948 1922 1930 1929 1895 1905 1052 1937 1966 1921
   ...
   1915 1935 1959 1974 1914 1951 1960 1967 212},
 :recovered #{1853},
 :recovered-frac 1/2665,
 :unexpected-frac 0,
 :unexpected #{},
 :lost-frac 62/2665,
 :duplicated-frac 42/2665,
 :ok-frac 145/533,
 :duplicated #{186 188 210 ... 1843 199 183}}
```

[3] Unfortunately, as you probably already know, computers.

3.3 Atomic Longs

AtomicLongs (and their cousins, AtomicReferences) provide a number (or arbitrary object) which can be read, written, and updated via atomic get-and-set & compare-and-set operations. Hazelcast's feature list claims these operations are "guaranted atomic across … a cluster." The documentation for IAtomicLong and IAtomicReference fails to mention that these operations are *not* in fact atomic: operations can be lost or improperly interleaved.

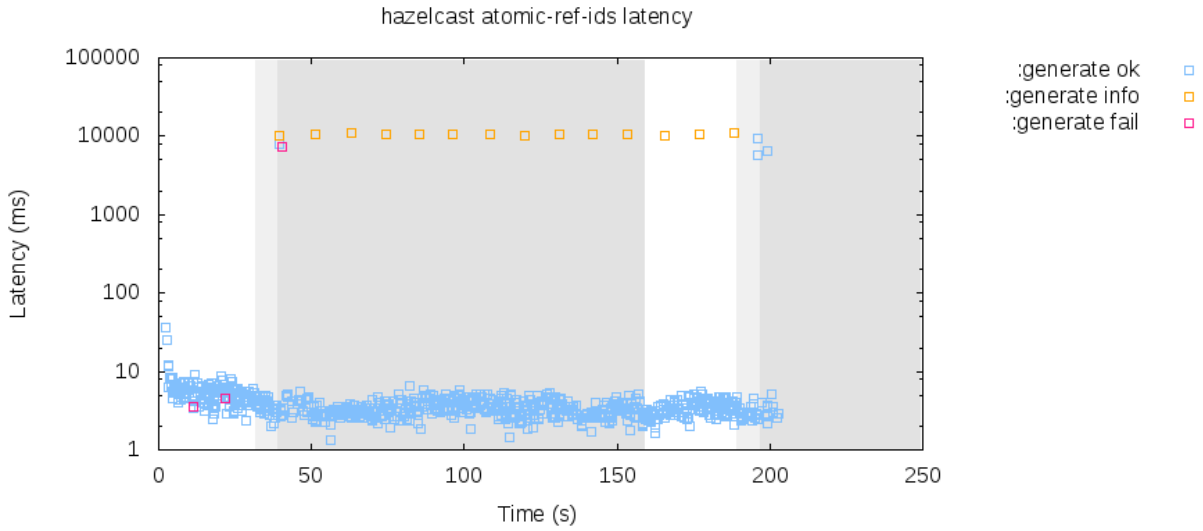
We could check that AtomicReference is linearizable. It's not clear that Hazelcast intends for AtomicReferences to actually *be* linearizable, but the documentation might imply it.[4] However, in this case it is more interesting to measure a *weaker* property: whether AtomicReferences allow for non-atomic or lost updates.

We begin with an AtomicLong (or an AtomicReference) initialized to 0 (null), and perform a sequence of atomic increment-and-get (get + compare-and-set) operations against it, across many nodes. If these operations are atomic, successful increment-and-get operations should record a sequence of unique integers, e.g. without duplicates.

In this 30-second test of an AtomicReference, with a single network partition, a little over 10% of all generated numbers were duplicated.

```
{:valid? false,
 :attempted-count 239,
 :acknowledged-count 235,
 :duplicated-count 27,
 :duplicated
 {171 2,
  172 2,
  173 2,
  175 2,
  ...
  195 2,
  196 2,
  197 2,
  198 2},
 :range [1 208]},
```

During a partition, AtomicLongs and AtomicReferences diverge, allowing different nodes to see and mutate different copies of the same "atomic" value. When the partition resolves, all but one of these divergent copies is discarded. Not only do AtomicLongs and AtomicReferences allow stale reads, but they also allow for lost updates and dirty reads. A strictly increment-only counter could go backwards *on a single node*, if a smaller value from another node were to overwrite it.



[4]The docs state "Hazelcast IAtomicLong is the distributed implementation of java.util.concurrent.atomic.AtomicLong". AtomicLong, like all types in java.util.concurrent.atomic, specifies that gets and sets have the memory semantics of volatile reads and writes, and that compare-and-set is like a volatile read and write. The Java Memory Model, in turn, specifies that operations on volatile variables introduce a synchronization barrier, which implies sequential consistency, and linearizability for compare-and-set operations. However, the documentation avoids making claims about real-time ordering, and only states "atomicity". We therefore restrict our testing to a weaker property implied by atomicity.

Note that AtomicReference and AtomicLong do not support quorum protection; unlike locks and queues, they will diverge *as long as the partition persists*, instead of diverging for only a few seconds at the start of a partition. In this plot from a different run with a longer partition, the grey region indicates the duration of a network partition. Note that while one node refuses updates, the other nodes are more than happy to run independent copies of the AtomicReference for the full duration of the partition. In this case, almost half (312 of 824) of generated numbers were duplicates.

3.4 ID Generators

If AtomicLong cannot provide safe atomic updates, a specialized datatype (like a flake ID) might be able to provide unique IDs *without* the need for AtomicLong's coordination properties. Hazelcast provides an IdGenerator type for generating "cluster-wide unique identifiers." What the documentation for IdGenerator does not mention is that IDs may not be unique in the event of a network partition. Indeed, IdGenerator is backed by an IAtomicLong internally, so it is subject to the exact same problems we saw in that type.

We perform the same uniqueness test as for AtomicLongs, but this time using an IdGenerator to generate unique numbers. IdGenerators, too, fail to provide uniqueness. If a pair of IdGenerators request a new block of numbers from their underlying AtomicLong during a partition, they will likely double-allocate that *entire block* of IDs.

In this 30-second test, Hazelcast allocated ~91,000 duplicated IDs out of ~834,000; a little over 10%. Like AtomicReferences, the fraction of duplicated IDs increases the longer a partition lasts; quorum protection does not limit the window for safety violations.

```
{:valid? false,
 :attempted-count 834013,
 :acknowledged-count 834013,
 :duplicated-count 91179,
 :duplicated
 {705516 2,
  705517 2,
  705518 2,
  ...
  705561 2,
  705562 2,
  705563 2},
 :range [0 747591]}
```

3.5 Maps

Hazelcast offers a Map, which is a distributed, sharded implementation of java.util.concurrent.ConcurrentMap. The documentation claims that Maps are "thread-safe to meet your thread safety requirements," implying that operations on maps are safe in the presence of concurrent updates. For optimistic concurrency control, Hazelcast maps offer operations like `putIfAbsent` and `replace` (essentially, compare-and-set), which replace missing or given values for a particular key with some new value, unless the current value in the map differs.

These functions only make sense if operations have some total order, but as we discussed earlier, during partitions, there *is* no single value, or total order, of operations. What does it mean to replace `A` with `B`, when the current values are *both* `A` and `C`? As we shall see, there *is* a good reason for Hazelcast to provide these operations, even when they may be, in general, unsafe.

First things first: let us check whether `replace` actually works. We take a map containing a single key, and store in that key an array of longs, representing a set of numbers. We attempt to add elements to that set by reading the current value, then using `replace` (or `putIfAbsent`) to replace the set we read with a *new* set now containing the added element. If `replace` obeys the contract of a ConcurrentMap, it should perform an atomic compare-and-set. Every successful `replace` should result in that particular element being present in all subsequent versions of the set.

After performing our series of replace operations, we heal all network partitions, allow the cluster 500 seconds to recover, and perform a final read to identify which elements survived.

Unfortunately, not all elements do. Quorum protection is insufficient to prevent the loss of acknowledged updates. In this test, Hazelcast lost four modifications to a single key, all clustered around the start of a network partition.

```
{:valid? false,
 :lost "#{1703..1704 1706..1707}",
 :recovered "#{}",
 :ok
 "#{1..2 4..36 ... 3724 3727..3728}",
 :recovered-frac 0,
 :unexpected-frac 0,
 :unexpected "#{}",
 :lost-frac 2/1865,
 :ok-frac 1221/1865},
```

3.6 Maps with CRDTs

Clearly, Maps are unsafe in the presence of network faults, even if majority quorums are used to prevent concurrent modification in independent clusters. However, Maps offer a feature no other Hazelcast datatype has: they can merge conflicting modifications after split-brain recovery. The default merge policy picks one of the disparate versions and discards the other, losing updates, but we could write a CRDT merge function instead, allowing us to preserve concurrent updates from isolated components of the cluster.

In this case, our data type is a grow-only set (a G-set), and our merge function is set union. We write a merge policy—a Java class—to merge two sets, and instruct the set to use that merge policy.

In this 300-second test, out of 1315 attempts, 1298 elements were successfully added to the set, and every one of those acknowledged elements was present in the final reads. These results are typical of CRDT map tests; they appear, at least in these particular circumstances, to safely preserve all operations.

```
{:valid? true,
 :lost "#{}",
 :recovered "#{}",
 :ok
"#{0..11 13..83 ... 1286..1306 1311..1314}",
 :recovered-frac 0,
 :unexpected-frac 0,
 :unexpected "#{}",
 :lost-frac 0,
 :ok-frac 1298/1315},
:valid? true}
```

Note that the merge function is *only* called during split-brain recovery, which means that we must continue to use our composite read+replace strategy. It may help to understand a Hazelcast cluster as a *cluster of components*, where each component is updated using a sequential compare-and-set, and updates to disparate components are merged with our commutative, associative, and idempotent merge function.

This approach is not limited to G-sets. We can implement any CRDT we like, including observed-removed sets, booleans with or/and, integers with min/max, counters, composite types like maps, and so on. In addition, we can relax the quorum constraint on CRDT maps, allowing updates to proceed safely on all nodes in the cluster, instead of only on a majority component.

4 Discussion

Although Hazelcast's partition-tolerance documentation hints that data loss might be an issue, it fails to make these properties explicit, and many datatypes claim to provide safety properties far stronger than what they actually guarantee.

In addition, the *names* of Hazelcast's datatypes, and the functions provided on those types, imply a certain fitness-for-purpose, e.g. that users can use these types and functions in a meaningful way. What is the point of an ID Generator which emits duplicate IDs? A lock that doesn't lock? Who wants an AtomicReference which is not atomic? Of what possible use is a queue which doesn't, well, *queue*?[5]

By providing familiar interfaces to Java programmers, and explicitly describing Hazelcast types as the analogue of well-known types like AtomicLong or ConcurrentMap, users are encouraged to treat Hazelcast types as if they offered the same concurrency safety properties of their Java counterparts. This would be a wonderful idea if Hazelcast maps *did* offer atomic `replace`, queues which didn't lose inserted elements, and so on.

This is particularly vexing because there are existing systems and algorithms which *do* provide these safety properties over similar datatypes. Consensus systems like Zookeeper and etcd, for instance, provide linearizable updates and reads[6] on registers, which is what Hazelcast *claims* to offer with its AtomicReferences. ID generators can be implemented with only a single round of consensus to initialize node IDs, either as local counters modulo `node-count`, or using flake IDs. Likewise: when cluster membership is fixed (e.g. by a round of consensus), counters can be made totally available with extremely low latency by using PN-Counters. Distributed queues cannot guarantee the ordering or exactly-once-delivery of their single-node counterparts, which means they are loosely equivalent to Observed-Removed Sets plus a loose temporal order, which could be, again, provided by flake IDs.

This is not to say that there are no safe uses of Hazelcast. Immutable records are obviously fine, provided users generate their own unique IDs for inserted

[5] Of course, there are use cases which only need safety *some* of the time. A broken lock service, for instance, may still be useful for ensuring a task is *typically* performed by a single node at a time.

[6] By default, reads in Zookeeper are only sequentially consistent, possibly lagging behind recent updates. However, linearizable reads are also possible in Zookeeper, by performing a `SYNC` prior to a read. Likewise, etcd and Consul offer linearizable quorum read options.

records instead of relying on, say, IdGenerator, and so long as read-after-insert consistency is not expected. Engineers typically expect a distributed cache to lose updates and expose stale data, so using a Hazelcast Map as a cache is, in general, okay. So too, for applications where the consequences of data loss are relatively minor (e.g. metrics, service health, sensor data) and outages are infrequent. Pubsub messaging through topics, similarly, is often assumed to be best-effort, and occasional message loss is fine. Hazelcast could be well-suited for service discovery,[7] since we *want* maximal availability, and incorrect or out-of-date information about what nodes run which services should only introduce *liveness* or *performance*, not *safety* problems.

Indeed, most users featured on the Hazelcast web site seem more oriented towards caching and discovery instead of using Hazelcast to ensure data safety. However, there *are* some systems which do appear to rely on Hazelcast's atomic primitives.

Consider, for example, BagriDB, a recently featured partner on the Hazelcast Blog. BagriDB uses Hazelcast to implement its an MVCC transaction manager on top of Hazelcast. While BagriDB claims to provide ACID transactions up to Repeatable Read, the transaction IDs in BagriDB are generated by a Hazelcast AtomicLong, with a small wrapper, incremented at the start of each transaction. This means that BagriDB could pick the same transaction ID for multiple concurrent transactions, which might result in the violation of transactional guarantees. BagriDB explained:

> Yes, the transaction IDs will be duplicated, but they will live in two different clusters after network partitioning and will be committed independently and successfully in most cases.

When asked whether they were aware of AtomicLong's non-atomicity, BagriDB responded:

> No, I was not aware of this behavior of IAtomicLong at network partitioning.

Likewise, OrientDB, one of eight partners featured on Hazelcast's home page, uses Hazelcast Locks to guarantee only a single node at a time can initialize registered nodes and to elect new lock managers. Hazelcast maps are also used to exchange leader and membership information; stale or lost updates to these maps might result in unexpected behavior.

We asked OrientDB about this, and OrientDB's engineers confirmed that they originally made extensive use of Hazelcast Locks for transactions. However, split-brain issues with Lock safety forced them to write their own distributed lock manager, and Hazelcast is now only used to reach consensus on cluster metadata on node startup. OrientDB plans to remove Hazelcast entirely in an upcoming release. They go on to note that they felt the documentation did not describe the potential for inconsistency:

> Documentation doesn't say anything about all of this. If you follow the documentation [it] looks like everything works as it's supposed to do.

Respondents on StackOverflow also indicate production use of Hazelcast for locking and atomic map updates. For instance:

> We use extensive use of distributed locking to make sure SKU Items of inventory are modified in atomic way because there are hundred of nodes in our web application cluster that operates concurrently on these items.

and

> We are using Hazelcast from last 3 years in our e-commerce application to make sure availability (supply & demand) is consistent, atomic, available & scalable. We are using IMap (distributed map) to cache the data and Entry Processor for read & write operations to do fast in-memory operations on IMap without you having to worry about locks.

We recommend that Hazelcast users carefully review their use of Hazelcast primitives. In particular, we encourage you to ask:

- For Locks, would the system be safe if the lock were omitted?
- For IdGenerators, what would be the impact of giving multiple requestors the same IDs?
- For Queues, what would happen if some enqueued messages were lost?
- Are there any uses of AtomicLongs or AtomicReferences in your code? Why?

[7] As another example, consider the problem of electing leaders in a protocol like multipaxos, where the protocol is safe regardless of which leaders are chosen, but having fewer leaders at any given point in time improves performance.

- For Maps, would the system be safe if any `put` were lost? What about a double-applied `putIfAbsent`? Be especially wary of calls to `replace`.
- Are your Map values structured as a CRDT with an appropriate merge function, or is order of operations important?

And in general, ask:

- Would it be safe to read some *prior* state of Hazelcast, instead of the current state?
- Does the system use Hazelcast as "fuzzy" state—e.g. as a cache, a discovery mechanism, or lossy store?

As for the Hazelcast team: we suggest that Hazelcast add appropriate warnings to the documentation for each datatype. Explicitly tell users that AtomicReferences are not atomic, that Locks are not exclusive, that Queues can lose messages, and so on. While the existing documentation on split-brain hints at these behaviors, more prominent warnings might prevent users from, say, building transaction management systems on top of non-unique ID generators.

We believe that Hazelcast's primitives *are* useful, but objects like AtomicReferences and Maps with `replace` *imply* consensus, and therefore deserve a real consensus system. Hazelcast should adopt or implement a proven consensus algorithm, like ZAB or Raft, and use it to back these datatypes.

Having a consensus system also enables atomic management of cluster membership and shard allocation, which unlocks efficient, eventually consistent implementations of counters, ID generators, queues, and generalized CRDTs. We recommend that Hazelcast adopt coordination-free algorithms (e.g. PN-Counters, flake IDs, etc) for these data structures to simplify their implementation, improve performance, and prevent lost updates.

Finally, almost all uses of lock services for safety in distributed systems are fundamentally flawed: users continue to interpret distributed locks as if they were equivalent to single-node mutexes. Lock services cannot guarantee exclusion in asynchronous networks, because there is no way to distinguish between a crashed node and a slow one: releasing any stale lock runs the risk of handing it to two processes concurrently. Even if we *could* provide a true distributed mutex, there is nothing which guarantees the network messages *emitted* by lock holders only take effect while the lock is held. This is why mature "lock services" like Chubby use a sequence number, which lock holders must use in their operations and downstream services must respect, to enforce exclusive and sequential execution of lockholder side effects. Hazelcast should remove or rename[8] Locks to avoid this problem, or couple them to a sequentially consistent sequence number.

These problems are not bugs; they are fundamental design decisions. Hazelcast has placed sequential or linearizable datatypes atop an eventually-consistent replication system which makes unjustifiably optimistic assumptions about node and network reliability. Jepsen has not filed specific bug reports for these issues with Hazelcast; instead, we feel that a comprehensive re-evaluation of Hazelcast's documentation, datatypes, and replication algorithms is in order.

We wish to thank Jordan Halterman for his discussion of Hazelcast use cases. Luigi Dell'Aquila & Luca Garulli from OrientDB, and Denis Sukhoroslov from BagriDB, were instrumental in understanding those systems' use of Hazelcast. Thanks also to Julia Evans, Sarah Huffman, Camille Fournier, Moishe Lettvin, Tim Kordas, André Arko, Allison Kaptur, Coda Hale, and Peter Alvaro for reading and offering comments on initial drafts. Finally, thanks to Greg Luck, from Hazelcast, for his comments and corrections. This research was performed independently by Jepsen, without compensation, and conducted in accordance with the Jepsen ethics policy.

[8] Perhaps an "ApproxiLock", or a "MostlyMutex".