Distributed Locks are Dead; Long Live Distributed Locks!

Ensar Basri Kahveci | April 02, 2019

Share

"Distributed locks aren't real", some like to remind us. "Anyone who's trying to sell you a distributed lock is selling you sawdust and lies." This may sound rather bleak, but it doesn't say that locking itself is impossible in a distributed system: it's just that *all* of the system's components must participate in the protocol. This blog post is the story of how we implemented a distributed locking protocol that gives your components a straightforward way of joining in.

In line with Hazelcast's tradition, the coordinating component of the protocol is an object that extends the semantics of **`java.util.concurrent.locks.Lock`**. We call it *FencedLock*, following the naming used in Martin Kleppmann's 2016 post "How to Do Distributed Locking".

You can also check out my follow-up blog post about how we tested the CP Subsystem APIs, including FencedLock, with Jepsen.

For the impatient reader, here are the takeaways of this blog post:

- FencedLock is a linearizable distributed implementation of the **`java.util.concurrent.locks.Lock`** interface with well-defined execution and failure semantics. It can be used for both coarse-grained and fine-grained locking.
- FencedLock replicates its state over a group of Hazelcast members via the Raft consensus algorithm. It is not vulnerable to split-brain problems.
- FencedLock tracks liveness of lock holders via a session mechanism that works in a unified manner for both Hazelcast servers and clients.
- **FencedLock allows 3rd-party systems to participate in the locking protocol and achieve mutual exclusion for the side-effects performed on them.** This is the "fenced" part of the story.
- FencedLock is battle-tested with an extensive Jepsen test suite. We have been testing its non-reentrant and reentrant behavior, as well as the monotonicity of the fencing tokens. **To the best of our knowledge, FencedLock is the first open source distributed lock implementation that is tested with such a comprehensive approach.**

Concurrency and nondeterminism are the main reasons that make multithreaded programming difficult. When there are multiple threads of

Relevant Resources

What's New in IMDG 3.12

Webinar | Video | 60 m

Watch Now +

Hazelcast IMDG 3.12 Product Datasheet

Datasheet | PDF | 4 pages

Download +

Hazelcast IMDG Deployment and Operations Guide

Guide | PDF | 66 pages

Download +

Stream Processing Essentials

execution, even a simple algorithm can produce a different result on every new run. Conceptually, distributed applications are not much different from multithreaded applications, so they need to deal with concurrency and nondeterminism as well. Distributed applications contain another underlying complexity that does not exist in multithreaded applications: partial failures. A distributed application is expected to keep running while some of its components have failed.

We can capitalize on this conceptual similarity to make concurrency primitives useful in distributed applications. Take JDK's **`java.util.concurrent.locks.Lock`** interface for example. It defines an API to coordinate access to a shared resource by multiple threads. Lock implementations can either enforce mutual exclusion to preserve correctness or efficiency, or allow concurrent access to a shared resource in a controlled manner, like **`ReadWriteLock`**. Moreover, the **`Lock`** interface is general enough to enable a distributed implementation where lock-acquiring threads live in different processes and/or the lock state is replicated over multiple processes.

Hazelcast IMDG 3.12 introduces a linearizable distributed implementation of the **`java.util.concurrent.locks.Lock`** interface in its CP Subsystem: **`FencedLock`**. It is efficient for both coarse-grained and fine-grained locking. You can use the monotonic fencing tokens provided by FencedLock to achieve mutual exclusion across multiple threads that live in different processes. In this blog post, you'll discover how we extended the semantics of the **`Lock`** interface for distributed execution and covered several failure modes that we can face in a distributed setting.

If you aren't yet familiar with Hazelcast's CP Subsystem, you can read our CP Subsystem primer. One basic fact about it is this: a Hazelcast cluster may have not one, but several subclusters that we call *CP groups*. Each CP group is a cluster on its own with respect to the CP services it provides.

The code samples in this blog post start one JVM and form a three-member CP group within it. We do this strictly for simplicity because the members still communicate over the network and aren't even aware that they are sharing the JVM. Their behavior is equivalent to each Hazelcast member running concurrently on its own JVM and the events occur in the order shown in the code samples. All code samples shown in this blog post are also placed in our code samples repository.

Basic Semantics

Let's start with a basic example and proceed by solving one problem at a time. Here's all the code you need to write to begin using a FencedLock:

```
Config config = new Config();
config.getCPSubsystemConfig().setCPMemberCount(3);

// Create 3 Hazelcast server instances, they will automatically form the CP Su
HazelcastInstance hz1 = Hazelcast.newHazelcastInstance(config);
HazelcastInstance hz2 = Hazelcast.newHazelcastInstance(config);
HazelcastInstance hz3 = Hazelcast.newHazelcastInstance(config);

// Obtain a handle to the same lock from two different HZ instances:
FencedLock hz1Lock = hz1.getCPSubsystem().getLock("my-lock");
FencedLock hz2Lock = hz2.getCPSubsystem().getLock("my-lock");

// Acquire the lock on instance 1:
hz1Lock.lock();
```

```
// Fail to acquire the lock on instance 2:
boolean lockedByHz2 = hz2Lock.tryLock();
assert !lockedByHz2;

// Release the lock on instance 1:
hz1Lock.unlock();

// Now Instance 2 can acquire the lock:
lockedByHz2 = hz2Lock.tryLock();
assert lockedByHz2;
```

In a nutshell,

1. Instance One acquires the lock
2. Instance Two fails to acquire the lock
3. Instance One releases the lock
4. Instance Two acquires the lock

We can conclude that, once a Hazelcast instance has acquired the lock, no other instance can acquire it until the holder explicitly releases it (or the system does it after the holder fails – see below for details). The instance can be either a server or a client instance; it works the same way.

FencedLock is reentrant by default:

```
// Acquire the lock on instance 1:
hz1Lock.lock();

// Acquire the lock again (succeeds):
boolean heldByHz1 = hz1Lock.tryLock();
assert heldByHz1;

// Release the lock once:
hz1Lock.unlock();

// Try to acquire from instance 2 (fails):
boolean heldByHz2 = hz2Lock.tryLock();
assert !heldByHz2;

// Release the lock from instance 1 again:
hz1Lock.unlock();

// Now instance 2 can acquire it:
heldByHz2 = hz2Lock.tryLock();
assert heldByHz2;
```

If you try to re-acquire the lock from the same instance, but on a different thread, you won't be allowed to:

```
// Acquire the lock on instance 1:
hz1Lock.lock();

// Ask for the lock again but on another thread (fails):
new Thread(() -> {
    boolean acquired = hz1Lock.tryLock();
    assert !acquired;
}).start();
```

This behavior is vital to the lock behaving the way you expect: protecting a single critical section at a time.

Java's **Lock** interface does not enforce a particular approach to reentrancy. Given this freedom, FencedLock offers configurable behavior and is reentrant by default, as shown above. You can disable reentrancy and use FencedLock as a non-reentrant mutex or set a custom reentrancy limit. When the configured reentrancy limit is reached, further lock acquire attempts fail with **`LockAcquireLimitReachedException`**.

FencedLock also offers a set of utility methods to query its status. You can check whether the lock is held and how many times it was reentrantly acquired (this fact is also committed to the CP group). You can query the lock-acquire count from anywhere, not necessarily from the lock-holding Hazelcast member and thread. Just keep in mind that you can use these facts only in an informative fashion – you can't rely on them for correctness because you'd be committing the *check-then-act* fallacy. For example, between the time you get the response *"you have the lock"* and the time you take some action on the assumption you have the lock, you may already have lost it.

While we're on this subject, the same logic applies even to the primary **`FencedLock.lock()`** call: at the very next line of code in your program, you may no longer be holding the lock. This is what the *fencing token* is for and we discuss it later on in this post.

FencedLock provides fairness. If a **`lock()`** request from *caller1* is committed before another **`lock()`** request from *caller2*, *caller1* wins.

We pretty much covered the most relevant parts of FencedLock's distributed execution semantics. Let's dive a bit deeper and investigate how FencedLock deals with failures. I have to admit, this is my favorite part!

Server-Side Failures

The good thing about using the Raft consensus algorithm under the hood is not having to worry about the consistency of the lock state. Each operation on a FencedLock gets committed to the majority of the CP group and each change in the lock state is guaranteed to be persisted as long as the CP group majority is alive. This means our FencedLock is invulnerable to split-brain problems. It preserves linearizability in case of network partitions and remains available on the side where the majority of the CP group is present.

FencedLock additionally achieves *exactly-once* semantics thanks to the idempotent implementations of its internal operations. For instance, even if the implementation retries a **`lock()`** operation because of a failed Raft leader, the lock is still acquired just once. The same rule applies to the other methods in the API.

This is a very useful guarantee because CP member failures can create a bit of trouble for in-flight operations. Say the Raft leader commits a lock-acquire request but then fails before sending the response to the caller. If we present the failure directly to the caller, how can it know whether the leader committed the request or not? In the reentrant mode, a naive retry performed by the client-side proxy would increment the acquisition count again and then a subsequent **`unlock()`** wouldn't release it. In the non-reentrant mode, if the first **`lock()`** request acquired the lock, the retried one would fail with **`LockAcquireLimitReachedException`**.

To avoid these pitfalls, we built the client side of the FencedLock to retry operations internally in an idempotent manner, providing the user with an end-to-end *exactly-once* execution guarantee.

Client-side Failures

What Happens if a Lock Holder Dies?

We rely on the mechanics of the Raft consensus algorithm and the idempotence technique to tolerate failures of CP members, i.e., replicas of the lock state. However, there is another critical participant: our client, the lock holder! What if a caller acquires a FencedLock and then crashes (or experiences a long-lasting hiccup) while still holding it? If we don't perform some cleanup, the entire application will suffer because the lock remains forever unavailable. This situation is equivalent to a Java program that calls **`Thread.suspend()`** or **`Thread.destroy()`** on a thread that is holding a lock. The Java designers got the easy way out: they simply banned thread suspension and destruction. Unfortunately, in the distributed world this is an unavoidable fact of life.

In order to deal with client failures, we introduced a mechanism to track the liveness of Hazelcast servers and clients in a unified manner: *CP Sessions*. We use this mechanism in the CP data structures that manage resource ownership; currently, there are two: **`FencedLock`** and **`ISemaphore`**. We create a new CP session when a Hazelcast server or client makes its very first lock or semaphore acquire request. After that, all lock and semaphore requests of the caller are associated with this CP session. Note that a Hazelcast server or a client maintains only a single CP session for a given CP group. Even if it is interacting with dozens of FencedLock instances in the same CP group, they will be all associated with a single CP session.

We keep the CP session alive for some time after we committed the last operation associated with it. If a client doesn't perform any operations on its own but is otherwise live, its proxy implementation will automatically send periodic *heartbeat* operations. If the client dies or has a very long hiccup, the session times out. We close it, release all the associated locks and semaphore permits, and cancel all pending requests.

Let's see the CP session auto-cleanup mechanism in action:

```java
Config config = new Config();
CPSubsystemConfig cpSubsystemConfig = config.getCPSubsystemConfig();
cpSubsystemConfig.setCPMemberCount(3);
cpSubsystemConfig.setSessionHeartbeatIntervalSeconds(1);
cpSubsystemConfig.setSessionTimeToLiveSeconds(10);
HazelcastInstance hz1 = Hazelcast.newHazelcastInstance(config);
HazelcastInstance hz2 = Hazelcast.newHazelcastInstance(config);
HazelcastInstance hz3 = Hazelcast.newHazelcastInstance(config);

// Hazelcast member One acquires the lock
hz1.getCPSubsystem().getLock("my-lock").lock();

// Member One crashes. After some time, the lock
// will be auto-released due to missing CP session heartbeats:
hz1.getLifecycleService().terminate();
FencedLock lock = hz2.getCPSubsystem().getLock("my-lock");
while (lock.isLocked()) {
    Thread.sleep(TimeUnit.SECONDS.toMillis(1));
    System.out.println("Waiting for auto-release of the lock...");
}

System.out.println("The lock was automatically released");
```

In this code sample, a Hazelcast member acquires the lock and then crashes. The rest of the CP group realizes the configured timeout has passed without any operations from that member and closes its session. This causes **`my_lock`** to be released.

The CP Subsystem also offers the ability to destroy a CP session manually. You can use this when you have the means to find out directly that a CP session owner crashed.

```
Config config = new Config();
CPSubsystemConfig cpSubsystemConfig = config.getCPSubsystemConfig();
cpSubsystemConfig.setCPMemberCount(3);
HazelcastInstance hz1 = Hazelcast.newHazelcastInstance(config);
HazelcastInstance hz2 = Hazelcast.newHazelcastInstance(config);
HazelcastInstance hz3 = Hazelcast.newHazelcastInstance(config);

hz1.getCPSubsystem().getLock("my-lock").lock();
// The lock holding Hazelcast instance crashes..
hz1.getLifecycleService().terminate();

CPSessionManagementService sessionManagementService = hz2.getCPSubsystem().get
Collection sessions = sessionManagementService.getAllSessions(CPGroup.DEFAULT_
// There is only one active session and it belongs to the first instance
assert sessions.size() == 1;
CPSession session = sessions.iterator().next();
// We know that the lock holding instance is crashed.
// We are closing its session forcefully, hence releasing the lock...
sessionManagementService.forceCloseSession(CPGroup.DEFAULT_GROUP_NAME, session

FencedLock lock = hz2.getCPSubsystem().getLock("my-lock");
assert !lock.isLocked();
```

Finally, a Hazelcast instance (both server and client) automatically closes its CP session on graceful shutdown:

```
Config config = new Config();
CPSubsystemConfig cpSubsystemConfig = config.getCPSubsystemConfig();
cpSubsystemConfig.setCPMemberCount(3);
HazelcastInstance hz1 = Hazelcast.newHazelcastInstance(config);
HazelcastInstance hz2 = Hazelcast.newHazelcastInstance(config);
HazelcastInstance hz3 = Hazelcast.newHazelcastInstance(config);

hz1.getCPSubsystem().getLock("my-lock").lock();
hz1.shutdown();

FencedLock lock = hz2.getCPSubsystem().getLock("my-lock");
assert !lock.isLocked();
```

To be precise, the CP Subsystem resets the CP session timeout in 3 circumstances:

- When the session owner performs a FencedLock or ISemaphore operation.
- When the session owner sends a periodic heartbeat.
- When the current Raft leader of the CP group crashes and a new leader is elected. This must be done in order to give the session owner time to discover the new Raft leader and start sending operations/heartbeats to it.

As veteran distributed systems folks might have already noticed, we borrowed several ideas from Google's paper "The Chubby lock service for loosely-coupled distributed system". We believe that the semantics of CP sessions are easy to grasp.

What if a Lock Holder Plays Dead, But Isn't?

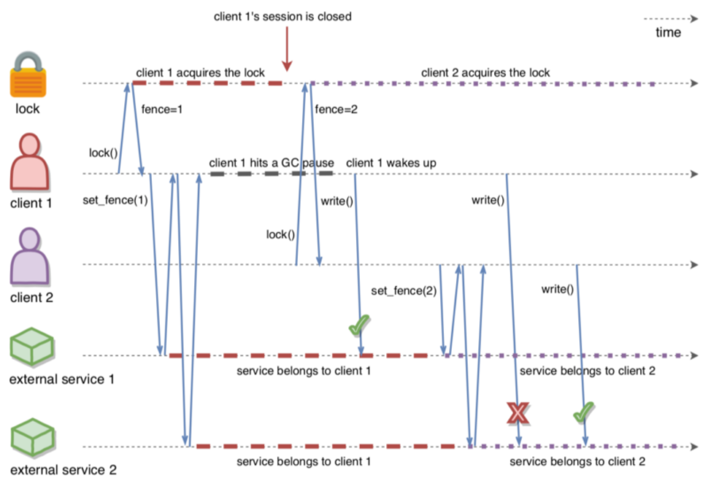
In an asynchronous system, the locking service cannot ensure that only one process thinks it holds the lock because there is no way to distinguish between a slow and a crashed process. Consider a scenario where a Hazelcast client acquires a FencedLock, then hits a long GC pause or becomes partitioned from the cluster. Since it will not be able to commit session heartbeats in the meantime, its CP session will be eventually closed and another Hazelcast client can acquire this lock. If the first client wakes up again or reconnects to the cluster, it may not immediately notice that it has lost ownership of the lock. In this case, multiple clients think they hold the lock. If they attempt to operate on a shared resource, they can break the system. Even if the first client actually crashes in this scenario, requests sent by 2 clients can be reordered in the network and hit the external resource in reverse order.

To prevent such situations, you can choose to use a long or maybe infinite CP session time-to-live duration, but then you'll get liveness issues. CP sessions offer a trade-off between liveness and safety. Long story short, no timing assumption can be 100% reliable in asynchronous networks.

We offer a fencing mechanism to deal with these situations and achieve mutual exclusion across the whole system without sacrificing liveliness. Namely, we order lock holders by monotonic fencing tokens. **Each FencedLock instance contains a fencing token which is incremented each time the lock instance moves from the available state to the locked state.** Before attempting any side-effectful actions, the lock holder must pass this fencing token to all external services it will use, thereby fencing off previous lock holders. It must also include the token in every request. The service must persist the largest observed fencing token and reject any incoming request whose fencing token is less than the current one. **This approach ensures that no two lock holders can interact with the service concurrently.**

Figure 1 below illustrates this idea. In the beginning, *Client-1* acquires the lock and receives *1* as its fencing token. It passes this token to the external services. Just after that, *Client-1* hits a long GC pause and eventually loses ownership of the lock because it isn't committing CP session heartbeats. Then, *Client-2* comes in and acquires the lock, receiving *2* as the fencing token.

Now the most interesting thing happens. Before *Client-2* had the chance to propagate its fencing token to the services, *Client-1* wakes up and manages to use its token to execute a request against the service, even though it's no longer the owner of the lock. This shows us that the fact *"Client-N owns the lock"* is not linearizable across the whole system (the combination of all the actors: the FencedLock, the clients, and the external services). The services learn the fact indirectly, through the clients' explicit propagation of the fencing token. **Nevertheless, mutual exclusion on the external services is not jeopardized as long as the clients obey the rules and propagate their fencing tokens before taking any side-effectful actions**.


client 1's session is closed
time
client 1 acquires the lock
client 2 acquires the lock
lock
fence=1
fence=2
lock()
client 1 hits a GC pause
client 1 wakes up
client 1
set_fence(1)
write()
write()
lock()
client 2
set_fence(2)
write()
external service 1
service belongs to client 1
service belongs to client 2
external service 2
service belongs to client 1
service belongs to client 2
Figure 1: Using fencing tokens to fence off stale lock holders

Note the key message here: all external services ***must*** participate in the fencing-token protocol, ***with guaranteed linearizability***, for the whole setup to uphold its invariants. You can check Martin Kleppmann's "How to do Distributed Locking" blog post to learn more about the fencing token idea. Google Chubby also implements a very similar solution, which they call *"sequence numbers"*.

We have a code sample that demonstrates how to utilize fencing tokens when talking to external services in our code samples repository. It's too long to show here.

Lock holders can query the FencedLock API for their fencing tokens. In the following code sample, the first Hazelcast member acquires the lock and then asks for the fencing token by calling **`FencedLock.getFence()`**. Then, the second Hazelcast member acquires the lock. Its fencing token is greater than the fencing token assigned to the first member. Note that fencing tokens are incremented only when the lock moves from the available state to the held state. Hence, the subsequent reentrant lock acquire of the second member returns the same fencing token.

```
Config config = new Config();
CPSubsystemConfig cpSubsystemConfig = config.getCPSubsystemConfig();
cpSubsystemConfig.setCPMemberCount(3);
HazelcastInstance hz1 = Hazelcast.newHazelcastInstance(config);
HazelcastInstance hz2 = Hazelcast.newHazelcastInstance(config);
HazelcastInstance hz3 = Hazelcast.newHazelcastInstance(config);

// The lock switches from the available state to the held state.
FencedLock hz1Lock = hz1.getCPSubsystem().getLock("my-lock");
hz1Lock.lock();
long fence1 = hz1Lock.getFence();
hz1Lock.unlock();

// The lock switches from the available state to the held state.
FencedLock hz2Lock = hz2.getCPSubsystem().getLock("my-lock");
hz2Lock.lock();
long fence2 = hz2Lock.getFence();

assert fence2 > fence1;

// The lock is already held by the second instance.
// Making a reentrant lock acquire.
hz2Lock.lock();
long fence3 = hz2Lock.getFence();

assert fence3 == fence2;

hz2Lock.unlock();
hz2Lock.unlock();

// The lock switches from the available state to the held state.
hz2Lock.lock();
```

```
long fence4 = hz2Lock.getFence();
hz2Lock.unlock();

assert fence4 > fence3;
```

You can use **`FencedLock.lockAndGetFence()`** and **`FencedLock.tryLockAndGetFence()`** methods to avoid repeated patterns of **`lock.lock(); lock.getFence();`** chains.

There's another subtlety related to reentrancy and the possibility of losing your lock in a hiccup. At the point your code re-acquires the lock, it may have lost it beforehand, so it doesn't actually re-acquire it, but gets it afresh. Your critical section was broken and other clients did their actions while you were gone, but everything appears to be going on as planned. The external systems agree that you are the lock holder and you proceed on the assumption that no other requests but your own got through. Eventually your code will try to release the lock twice and get an exception on the second release, but at that point the damage has already been done.

FencedLock implements a further mechanism that prevents this from happening: it makes the lock holder aware that it has lost ownership of the lock the next time it interacts with the FencedLock proxy. In Figure 2, *Client-1* acquires the lock and hits a long GC pause. It loses ownership of the lock after some time due to the failure to commit any heartbeat operations. Then, *Client-2* acquires and releases the lock while *Client-1* is still frozen. Eventually, *Client-1* comes back alive and tries to reentrantly acquire the lock for the second time, thinking it still holds it. Now FencedLock throws **`LockOwnershipLostException`** and stops it from proceeding with the happy path.

You can use the same mechanism even without re-acquiring the lock, as an optimization: you can find out at any point in the code that you lost the lock, before trying to access the external system.

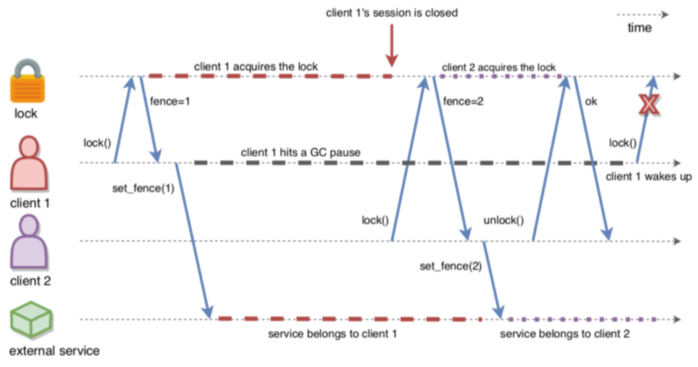


Figure 2: Failing stale lock holders

We'll demonstrate the failure with **`LockOwnershipLostException`** on an even simpler scenario. In the following code sample, we forcefully close the CP session of the lock-holding Hazelcast member as if its session was closed due to missing heartbeats. Then, the next **`lock()`** call of the kicked-out lock holder fails with **`LockOwnershipLostException`**. This scenario could occur when a Hazelcast member comes back alive after it was assumed to be crashed and its CP sessions were forcefully closed. Note that **`LockOwnershipLostException`** is still thrown even if the lock is not acquired by anyone else after it is forcefully released in the CP group.

```
Config config = new Config();
CPSubsystemConfig cpSubsystemConfig = config.getCPSubsystemConfig();
cpSubsystemConfig.setCPMemberCount(3);
HazelcastInstance hz1 = Hazelcast.newHazelcastInstance(config);
HazelcastInstance hz2 = Hazelcast.newHazelcastInstance(config);
HazelcastInstance hz3 = Hazelcast.newHazelcastInstance(config);

FencedLock lock = hz1.getCPSubsystem().getLock("my-lock");
lock.lock();

CPSessionManagementService sessionManagementService = hz2.getCPSubsystem().get
Collection sessions = sessionManagementService.getAllSessions(CPGroup.DEFAULT_

assert sessions.size() == 1;
CPSession session = sessions.iterator().next();
// There is only one active session and it belongs to the first instance.
// We are closing its session forcefully to mimic that
// its session was closed because of missing session heartbeats...
sessionManagementService.forceCloseSession(CPGroup.DEFAULT_GROUP_NAME, session

try {
    // The new lock acquire call of the lock holder
    // fails with LockOwnershipLostException
    lock.lock();
    assert false;
} catch (LockOwnershipLostException expected) {
}
```

Testing FencedLock with Jepsen

If there is anything harder than building distributed systems, it is validating their behavior. Jepsen has proven to be a very successful tool in this endeavor and became the standard tool to test the safety of distributed systems. It subjects the system to many kinds of failures while running a test case and verifies that the system keeps its promises.

We have been using Jepsen to test the correctness of our new linearizable `java.util.concurrent.*` implementations under network partition failures. In fact, we followed a process we call *"Jepsen-test-driven development"*. We wrote our Jepsen tests first. Then, we ran those tests against the implementations to see how they behave and where they fail. After each iteration, we improved both the behavior and reliability of the implementations and then made another test run. This approach has been very fruitful to enable our implementations to preserve safety and remain useful for several edge cases. To the best of our knowledge, FencedLock is the first open source distributed lock implementation that is tested with such a comprehensive approach.

We test FencedLock in 4 different ways:

- *Non-reentrancy:* we test if FencedLock behaves as a non-reentrant mutex, i.e., it can be held by a single endpoint at a time and only that endpoint can release it. Moreover, the lock cannot be acquired by the same endpoint reentrantly.
- *Reentrancy:* We test if FencedLock behaves as a reentrant mutex. A single endpoint can hold the lock at a time and that endpoint can acquire the lock reentrantly. The reentrant lock acquire limit is 2 for this test.
- *Monotonic fencing tokens for non-reentrant FencedLock:* We validate the monotonicity of fencing tokens assigned to lock holders.
- *Monotonic fencing tokens for reentrant FencedLock:* Fencing tokens are incremented each time the lock switches from the available state to the

held state. However, if the current lock holder acquires the lock reentrantly, it will get the same fencing token. The reentrant lock acquire limit is 2 for this test.

We fixed all the bugs that these tests revealed so far. You can see our tests in the official Jepsen repo. We are planning to extend our suite with more test scenarios and failure cases in addition to network partition failures. We are also integrating our Jepsen test suite into our CI system.

Closing Words

Locking in a distributed system is possible, but only if all of its components participate in the protocol. Hazelcast provides you with the implementation of such a distributed locking protocol and gives your components a straightforward way to participate in it.

We are looking forward to hearing about how developers use FencedLock for their needs. You can check our documentation and code samples to learn more about FencedLock, and use our Google groups to ask for help.

Until next time…

Acknowledgments

I would like to thank Martin Kleppmann for his valuable feedback and insightful comments, Marko Topolnik for his remarkable effort to help me improve this blog post, and Hazelcast folks for their reviews.

Edit Log

(April 4, 2019): We improved Figure 1 and the explanation of how to use the monotonic fencing tokens to achieve mutual exclusion across external services. In short, although no locking service can ensure only one process thinks it holds the lock, we can use monotonic fencing tokens to ensure that only one process can perform side effects on multiple external services at a time.

(June 6, 2019): The follow-up blog post is linked in the intro.

About the Author

Ensar Basri Kahveci
Distinguished Engineer

Ensar's primary areas of interest are distributed data, replication, consistency, and storage. He has more than seven years of hands-on expertise in designing, developing, and testing distributed

Latest Blogs

Testing the CP Subsystem with Jepsen

algorithms, with solid experience in concurrency. He has authored a number of articles on distributed data and stream processing, and is a frequent speaker at industry conferences on topics such as replication and distributed systems. Several of his talks can be found on YouTube, including Replication Distilled, Distributed Systems for Mere Mortals, and Replication in the Wild. Ensar is a Ph.D. candidate in computer science at Bilkent University in Ankara, Turkey.

Follow me on

Riding the CP Subsystem

Hazelcast IMDG 3.12 Introduces CP Subsystem

View all blogs by the author +

Free Hazelcast Online Training Center

Whether you're interested in learning the basics of in-memory systems, or you're looking for advanced, real-world production examples and best practices, we've got you covered.

Enroll Now

Learn More

Silicon Valley (HQ)
2 West 5th Ave., Suite 300
San Mateo, CA 94402 USA

View All Locations +

Quick Links

- Company
- Products and Services
- Pricing
- Support
- Glossary
- In Memory Data Grid

Newsletter Sign Up

Email Address *

Enter your email

Subscribe

Follow Us

Privacy Policy | Terms of Use