Lease-Based Serialization

Solutions developed for single systems often prove inadequate to embrace the added complexities of distributed systems. This is certainly the case with critical-section synchronization in systems with distributed actors. Leases represent a very different, and much more robust approach to serialization.

Challenges of Distributed Locking

Among Deutsch's Seven Fallacies of distributed computing were:

- zero latency
- reliable delivery
- stable topology
- consistent management

Locking operations in distributed systems run afoul of all of these:

- In a single node, a compare-and-swap mutex operation might take many tens of nanoseconds. Obtaining a lock through message exchange will likely take at least tens of milliseconds. That is a minimum one-million-X difference in performance.
- In a single node, a mutex operation (whether implemented with atomic instructions or system calls) is guaranteed to complete (tho perhaps unsuccessfully). In a distributed system the request or response could be lost.
- When a single node crashes, it takes all of its applications down with it, and when they restart, all locks will be re-acquired. In a distributed system the node holding the lock can crash without releasing it, and all the other actors will hang indefinitely waiting for a release that will never happen.
- If a process dies, the OS knows it, and has the possibility of automatically releasing all locks held by that process. If the OS dies, all lock-holding processes will also die, and nobody will have to cope with the fact that the OS no longer knows who holds what locks. When a node dies in a distributed system, there is no meta-OS to observe the failure and perform the cleanup. If the failed node happens to be the lock-manager, the remaining clients may find that their locks have been "forgotten".

Other issues include:

- In a single node it was possible to use some combination of atomic instructions and interrupt disables to prevent parallelism within critical sections. There are no WAN-scale atomic instructions or interrupt disables.
- In a single system, we might understand the resources well enough to be able to assign a total ordering to all resources, and so prevent circular dependencies (and thereby deadlocks). In a distributed system the set of possible resources may not be orderable, or even known, which eliminates ordering as a practical means of deadlock prevention.

Addressing these Challenges

Distributed consensus and multi-phase commits are extremely complex processes, probably far to expensive to be used for every locking operation. It is much easier and more efficient to simply send all locking requests (as messages) to a central server, who will implement them with (simple, efficient, reliable) local locks.

The more complex failure cases and greater deadlock risks can be dealt with by replacing *locks* with *leases*. A lock grants the owner exclusive access to a resource until the owner releases it. A lease grants the owner exclusive access to a resource until the owner releases it or the lease duration expires.

In principle, for normal operation a lease works the same as a lock. Someone who wants exclusive access to a resource requests the lease. As soon as the resource becomes available, the lease is granted, and the requestor can use the resource. When the requestor is done, the lease is released and available for a new owner. But in practice, there is one other very important difference: Locks work on an *honor system*. An actor who does not yet have a lock will not enter the critical section that the lock protects. Leases are often enforced. When a request is sent to a remote server to perform some operation (e.g. update a record in a database), that request includes a copy of the requestor's lease. If the lease has expired, the responding resource manager will refuse to accept the request. It does not matter why a lease may have expired:

- the release message was lost in transit
- the owning process has crashed
- the node on which the owner was running has crashed
- the owning process is running slowly
- the network connection to the owning node has failed

Whatever the reason for the expiration, the lease is no longer valid, operations from the previous owner will no longer be accepted, and the lease can be granted to a new requestor. This means that the system can automatically recover from any failure of the lease-holder (including deadlock). But we do have (at least) two issues:

1. An expired lease prevents the previous owner from performing any further operations on the protected resource. But if the tardy owner was part-way through a multi-update transaction, the object may be left in an inconsistent state. If an object is subject to such inconsistencies, updates should be made in all-or-none transactions. If a lease expires before the operation is committed, the resource should fall back to its last consistent state.
2. The choice of the lease period may involve some careful tuning. If the lease period is short, an owner may have to renew it many times to complete a session. If the lease period is long, it may take the system a long time to recover from a failure.

Sending a network message for every entry into a short critical section would be disastrous. On the other hand, using leases for long lived resources (like DHCP allocated IP addresses) can be extremely economical. It comes down to:

- the number of operations that can be performed under a single lease-grant
- the ratio of the costs of obtaining the lease to the costs of the operations that will be performed under that lease

A three second delay to obtain an IP address is trivial if the IP address can be used for 24 hours thereafter. A one second delay to get a lock on a file might amortize down to nothing if we could then use that lock to do one million writes.

Evaluating Leases

Mutual Exclusion
: Leases are at least as good as locks, with the additional benefit of potential enforcement.

Fairness
: This depends on the policies implemented by the remote lock manager, who could easily implement either queued or prioritized requests.

Performance
: Remote operations are, by their very nature expensive ... but we probably aren't going to network leases for local objects. If lease requests are rare and cover large numbers of operations, this can be a very efficient mechanism.

Progress

The good news is that automatic preemption makes leases immune to deadlocks! But, if a lease-holder dies, other would-be lessees must wait for the lease period to expire.

Robustness
: This was never mentioned as a criterion when we were evaluating single system synchronization mechanisms, but leases are clearly more robust than those single-system mechanisms.

Leases are not without their problems:

- while they easily recover from client failures, correct (highly stateful) recovery from lock-server failures is extremely complex.
- automatic lease expiration is a very powerful feature, but it raises the issue of how to decide "what time it is" in a distributed system without a universal time standard.

These are interesting problems, with interesting solutions, but well beyond the scope of this introductory course.

Opportunistic Locks

For most resources, contention is rare, and the locking code is there only to ensure correct behavior in unlikely situations. It seems unfair to have to pay the high cost of remote lock requests to deal with an unlikely problem. The CIFS protocol supports *opportunistic locks*. A requestor can ask for a long term lease, enabling that node to handle all future locking as a purely local operation. If another node requests access to the resource, the lock manager will notify the op-lock owner that the lease has been revoked, and subsequent locking operations will have to be dealt with through the centralized lock manager.

Summary

Locks are a fundamental concept, but one from a simpler time. They are ignorant of Deutsch's Seven Fallacies, and do not work well in distributed systems. Leases are a richer and more expensive mechanism that more robustly embraces these more complex situations. These capabilities make them interesting even in single-node applications where locking must be robust in the face of an ever-changing set of clients.