Lottery Scheduled Distributed Lock Services for Multi-Tenant Coordination

Ryan Wang, Eliot Yoon, Steve Zang

Background

In modern day distributed systems, distributed lock services provide key synchronization primitives for important tasks such as leader election and partitioning a workload among worker nodes. At its core, distributed lock services must enforce mutual exclusion. However, lock services make no additional guarantees on lock acquisition policy, with the implementation dependent on the specific lock service. Two lines of work in distributed locking are Google's Chubby and Yahoo’s ZooKeeper. Importantly, both systems implement a FIFO acquisition model for locks.

Motivation

Distributed lock services make the assumption that all services that share a distributed lock service are cooperative. For example, Chubby, a distributed lock service that is deployed and widely used at Google, was designed with the assumption that each team would have its own lock service. However, they observed that in reality, a single lock service was shared by many tenants. This is because the underlying reality of distributed systems is that tenants share key pieces of state or coordination points. For example, services across tenants often are coupled, updating the same metadata or requiring coordination because they are part of a cross-service workflow. However, this can create problems as tenants have different SLAs, workloads, and priorities. This means that a mistake in one tenant can lead to starvation for other tenants and aggressive tenants can drastically disrupt more passive tenants. Therefore, in multi-tenant systems, locks must be treated as a shared resource that require isolation and proportional access.

Design

We implement a modular distributed lock service testing environment that allows for the configuration of diverse workloads that can run on either a fully simulated synthetic network or an end-to-end asynchronous TCP network.

Lock Service

The lock service implements three scheduling policies: FIFO, Lottery, and Unweighted Lottery. FIFO, in which incoming lock requests are served in a first come, first serve fashion, is the predominant policy in many real world distributed lock services and serves as the baseline. Lottery is the primary focus of our testing, and implements a policy in which the next tenant to be granted the lock is chosen from a random selection, weighted by each tenant's ticket allocation. The requests for the selected tenant are then served by a FIFO queue. Unweighted Lottery equalizes each tenant's ticket share, serving as another baseline.

During runtime, incoming requests to the lock service are passed to the `handle_acquire` method, which enqueues the request according to the configured policy and grants the lock to the request only if the lock is not already held. When a lock-holding request completes and releases the lock, the request is passed to the `handle_release` method, which frees the lock only if the incoming request matches the lock holder that the lock service has on record. When the lock is freed, the next holder is chosen according to the configured policy and granted the lock. Thus, it is clear that the service enforces the requirement of mutual exclusion.

Workloads

Workloads define the tenant configuration that is being run. A workload can be easily modified or generated to model a specific distributed process. The key fields of a workload configuration are as follows:

- `tenants`: Defines the competing processes/systems for a lock and their ticket shares
- `horizon`: The maximum runtime of a workload
- `warmup_time`: The length of runtime that passes before any metrics begin to be tracked
- `arrival_mode`: Determines whether tenants continuously send requests one after another (`poisson`) or wait until a request is granted and completed before sending another (`closed_loop`)
- `arrival_rate`: The rate at which tenants send requests in `poisson` mode
- `cs_time_mean`: The average runtime of the critical section of a tenant's request
- `wait_slo`: The service level objective wait time of a tenant, only used for metric computation
- `num_clients`: The number of per-tenant client processes that concurrently send requests in `closed_loop` mode

Additionally, the test script introduces another field, `load_scale`, which scales the arrival rates of all tenant requests by a constant amount. This field allows us to test the effects of varying the overall load on a system while keeping everything else constant.

Simulator

The simulator is a synthetic network that implements tenant processes as events that are scheduled for execution. Events are generated according to the workload configuration and stored in a min-heap by event time. There are two types of events: arrivals and releases. Arrivals represent when tenant requests arrive to the server that maintains the lock service. On arrival, a tenant's request is sent to the lock service through the `handle_acquire` method. If configured to the Poisson arrival mode, the tenant's next arrival is then scheduled according to the arrival rate of the tenant. When the lock service grants a tenant request, the simulator schedules a release event for that request after the critical section length completes. On release, the request is sent to the lock service through the `handle_release` method, which then grants the lock to a new request. If configured to the closed loop arrival mode, the tenant's next arrival is scheduled immediately after the release. Additionally, the simulator adds an optional `NETWORK_DELAY` and `LOCK_DELAY` to the scheduled events, which can be configured to better model a real-world network.

Network

The network implementation models an end-to-end asynchronous TCP system in which a central server maintains the lock service and communicates with multiple clients representing tenants and, in closed-loop mode, their individual client processes. Clients asynchronously and independently generate requests according to the configuration, sleeping between arrivals and during critical sections, and send releases when the critical section of a granted request is complete. In Poisson arrival mode, clients continuously send requests according to the arrival rate, while in closed loop mode each client waits for a grant and release before sending another request. The server passes all requests and releases to the lock service through the `handle_acquire` and `handle_release` methods, and sends granted requests back to the client it came from. The `TIME_SCALE` parameter scales the network execution times by a configurable amount in order to expedite testing, at the expense of exact timing fidelity due to scheduling overhead and network jitter in the asynchronous runtime.

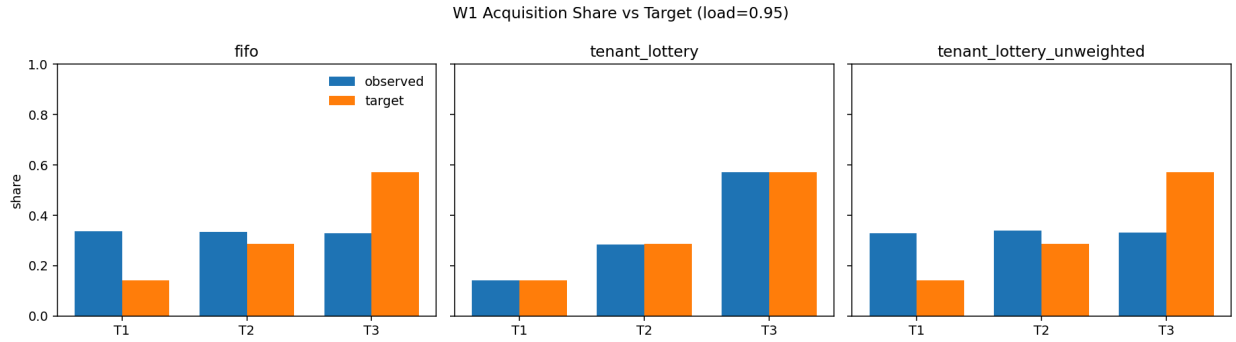
Evaluation

Experimental Setup

We evaluated our lottery-scheduled lock service against FIFO and unweighted lottery baselines across nine workloads spanning diverse multi-tenant scenarios. Each workload varies ticket weights, arrival rates, and critical section times to compare different aspects of lock scheduling. We focus on these key metrics: acquisition share relative to target weights, SLO Attainment, and SLO throughput.

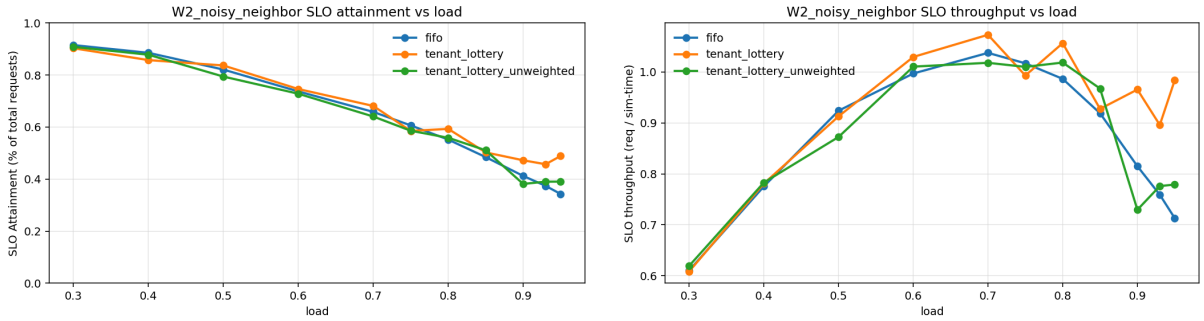
Workload Analysis

The following highlighted results were obtained from a simulation run of the benchmark suite from the project repository. Results from the network implementation are not included in the analysis due to the excessive runtime of the benchmark suite when tested in a real-time network. However, when tested on a less comprehensive quick suite the network results are mostly consistent with the simulation results, and minor discrepancies can largely be attributed to differences in latency and scheduling overhead.



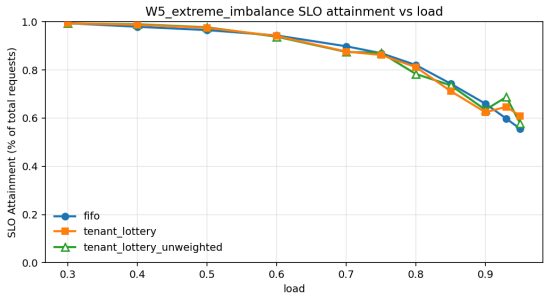
W1_basic_fairness: Serves as our sanity check. With three tenants configured for ticket shares of 1:2:4 in a closed loop system, FIFO and unweighted lottery gives nearly equal acquisition shares to everyone. Weighted lottery moves observed shares to be nearly identical to their targets, confirming the policy actually enforces tenant weights as designed.

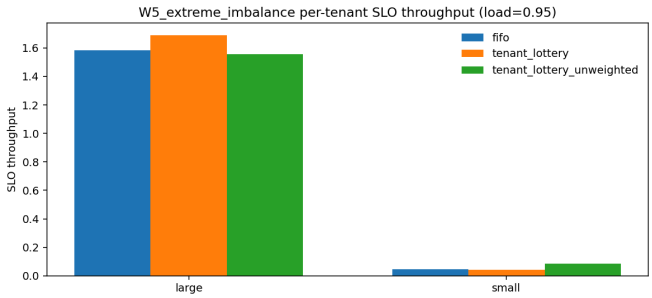
All following workloads are configured to a Poisson arrival mode to simulate systems under continuous contention.



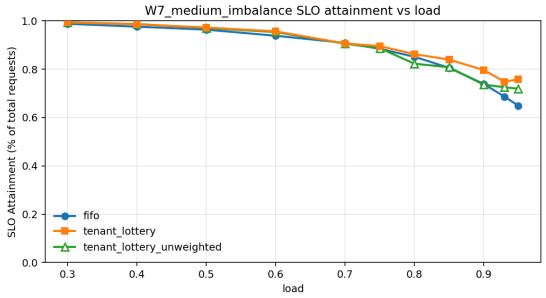
W2_noisy_neighbor: Shows why raw throughput isn't enough. A policy can complete many requests yet miss latency targets, so we track SLO-attaining throughput. Under high load (0.95), FIFO achieves only 0.71 SLO-throughput while weighted lottery reaches 0.98. This shows that lottery-scheduled lock sharing can result in more useful work completed under contention.

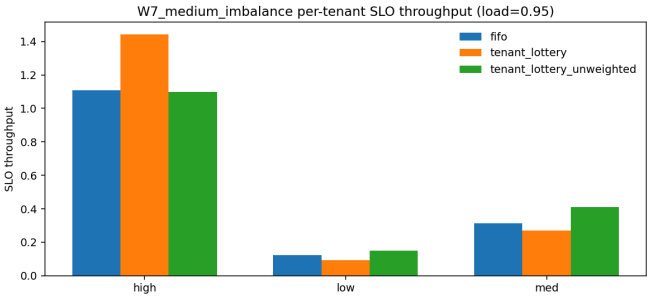
With these baselines established, in the following sections we examine workloads that reveal real contention patterns.



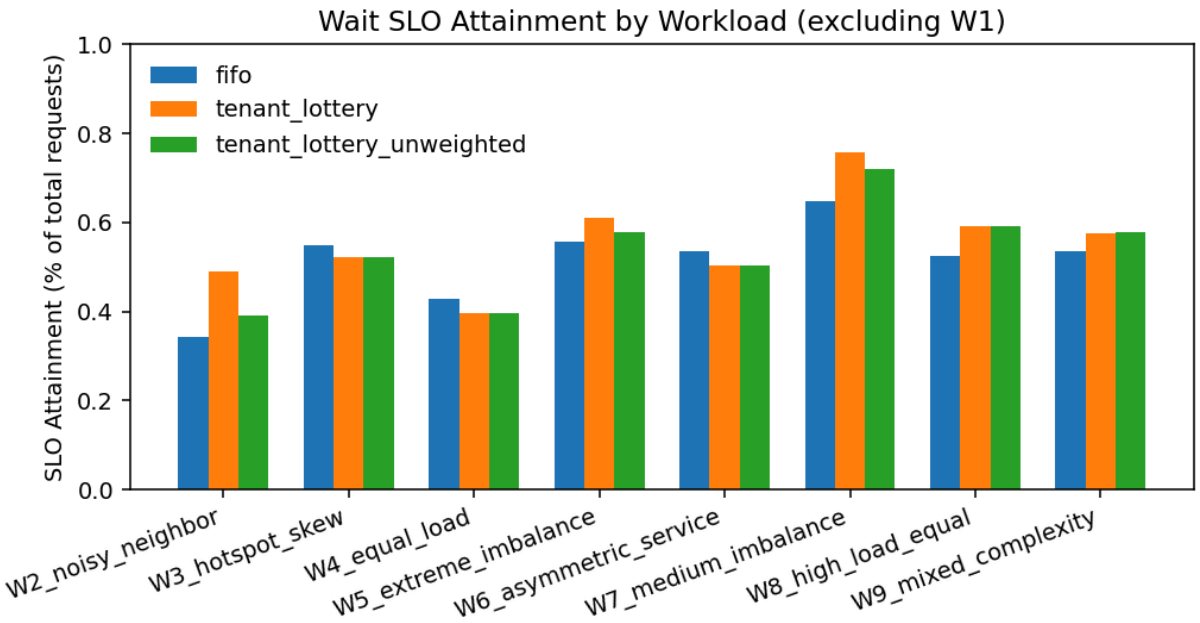


W5_extreme_imbalance: This workload pairs a low-weight tenant (1 ticket, low arrival rate) with a high-weight tenant (1000 tickets, high arrival rate). Under FIFO, the high-arrival tenant dominates lock acquisitions. Weighted lottery amplifies this, with the 1000:1 ticket ratio meaning the high-weight tenant wins nearly all competitions. While this maximizes throughput for the high-weight tenant, the low-weight tenant experiences severe starvation, with SLO attainment near-zero under load. The key insight is that when ticket ratios are extreme and demand is asymmetric, "proportional" access can still mean effective exclusion.





W7_medium_imbalance: This workload features three tenants with progressive ticket weights (1, 10, 100) and increasing arrival rates (0.2, 0.5, 1.8). Weighted lottery achieves the highest overall SLO attainment across most loads. However, this aggregate metric masks the fact that the high-weight tenant thrives (1.44 throughput), while medium and low tenants perform worse under weighted lottery than other policies (0.26 and 0.09 respectively). The ticket advantage compounds with higher demand, concentrating performance gains at the top while sacrificing the lower-weight tenants.



Looking at SLO attainment across workloads, fair scheduling alone isn't enough. In workloads with equal ticket weights but uneven demand, FIFO naturally favors tenants that arrive more often. Lottery gives everyone equal lock chances, which actually caps the busy tenant's throughput and hurts their SLOs. The gap grows with longer critical sections, as lottery selection bypasses earlier arrivals, making them miss deadlines. Thus, we can infer that fair locks help, but we also need to consider demand patterns, job lengths, and system load to achieve real fairness.

Overall, our evaluation shows that lottery scheduling provides proportional lock access and prevents starvation in multi-tenant environments. However, fair locks don’t create fair systems; thus, effective deployment requires not only setting the ticket weights but also understanding workload characteristics.

Contribution Statement

For the implementation of the system, Ryan contributed to the creation of the lock service simulation assumptions, specifications, and interface as well as providing an initial implementation of the lock service simulator. Steve contributed to the creation of a representative and diverse set of evaluation workloads that investigated the strengths and limitations of our design. Eliot contributed to the implementation of introducing real network latency rather than simulating it, validating the applicability and fidelity of our approach. All three members contributed equally to the creation of the midterm and final presentation slides as well as the final report. The code for this project can be found at github.com/RyanWang3/multi-tenant-lock-service-simulator.