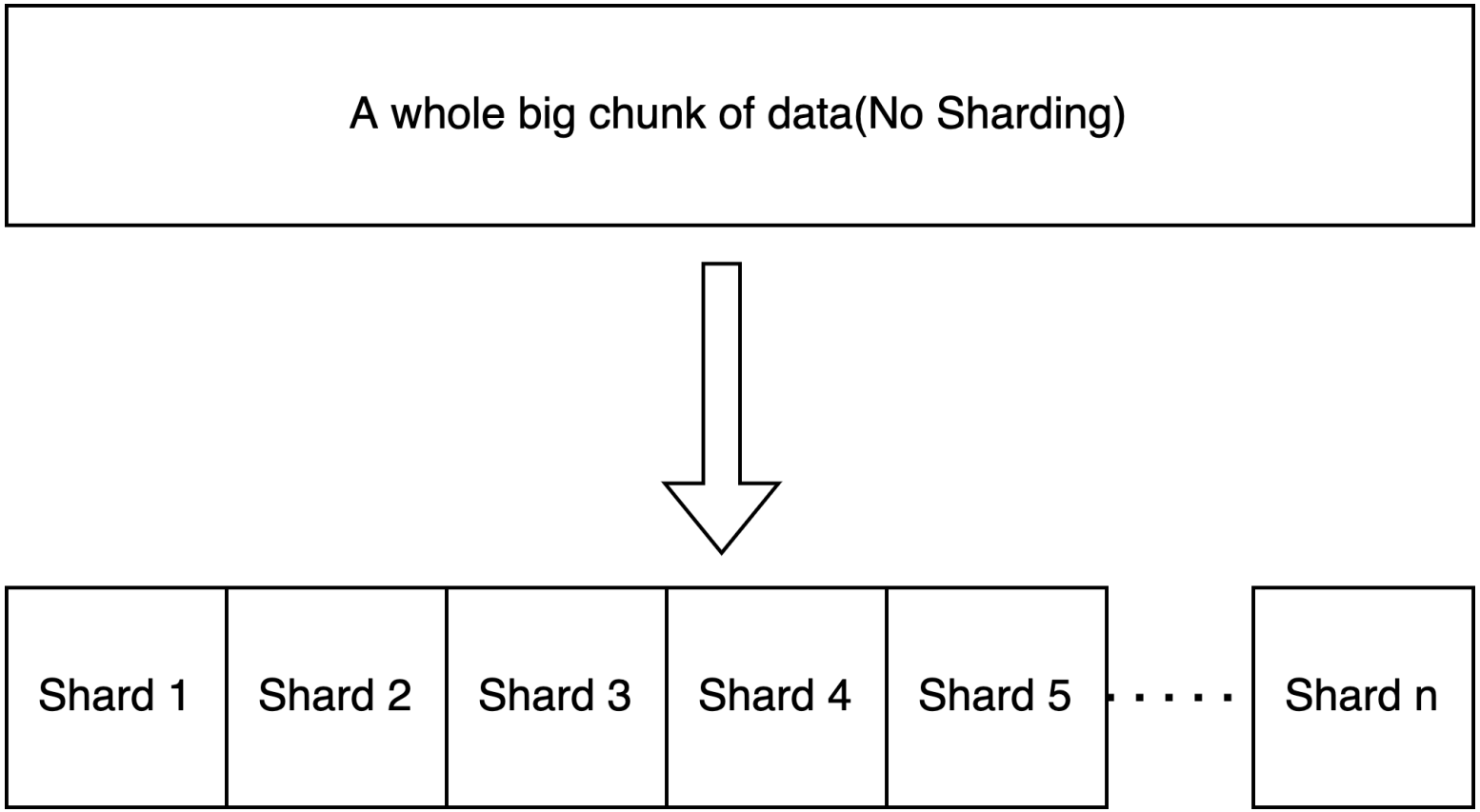
Shard Manager: A Generic Shard Management Framework for Geo-distributed Applications

By Sangmin Lee, Zhenhua Guo, Omer Sunercan, Jun Ying, Thawan Kooburat, Suryadeep Biswal, Jun Chen, Kun Huang, Yatpang Cheung, Yiding Zhou, Kaushik Veeraraghavan, Biren Damani, Pol Mauri Ruiz, Vikas Mehta, Chunqiang Tang

https://dl.acm.org/doi/10.1145/3477132.3483546

Presented by Bocheng Cui
PhD Student in UNH
noahcui.github.io/info

Sharding


A whole big chunk of data(No Sharding)
Shard 1
Shard 2
Shard 3
Shard 4
Shard 5
Shard n

Regional vs Geo-distributed

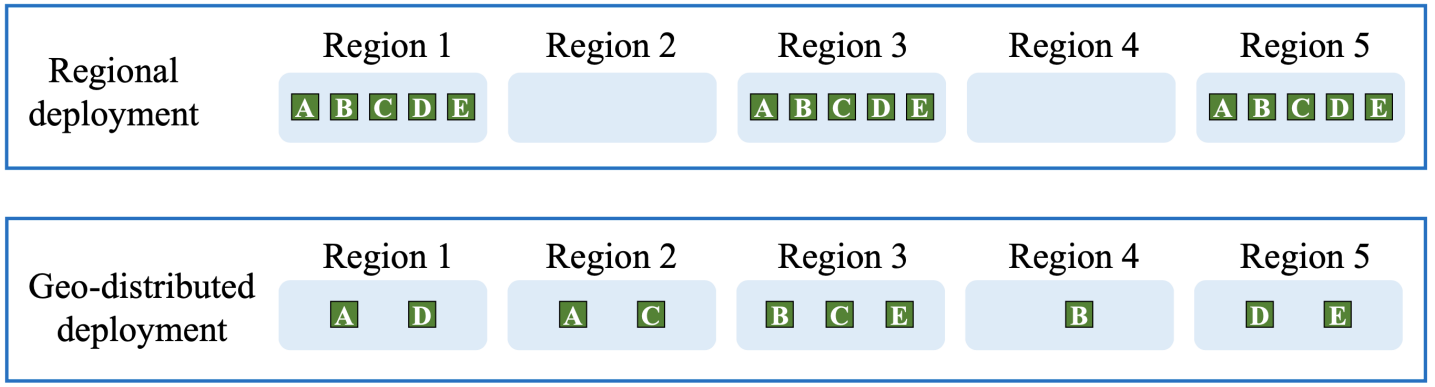


Figure 3. Regional vs. geo-distributed deployment. An application with 5 shards (A-E) is deployed to 5 regions (1-5).

Barriers

Availability

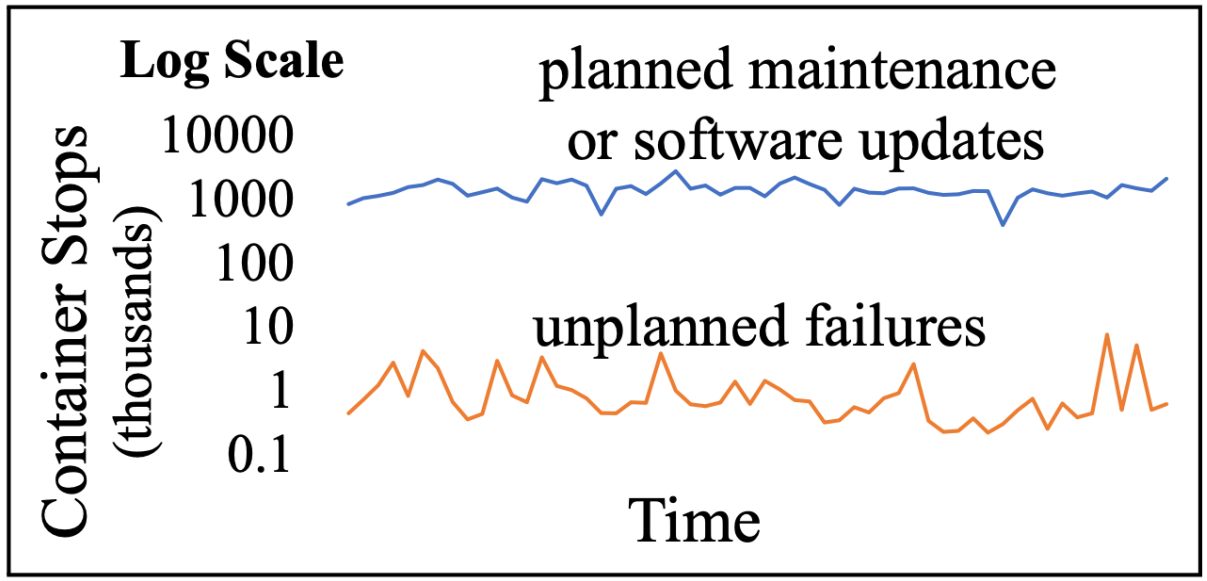


Figure 1. Planned vs. unplanned container stops.

Placement & load-balancing

- Many application require advanced PLB features.
- Existing sharding frameworks don’t account for the bimodal nature of their workloads
- Hardware Constrains:
  - System stability
  - Server capacity
- Soft goals:
  - Region preference
  - Spread of replicas
  - Planned maintenance
  - Utilization threshold
  - Global load balancing
  - Regional load balancing
  - Parallel shard failover

Shard Manager

SM's Architecture

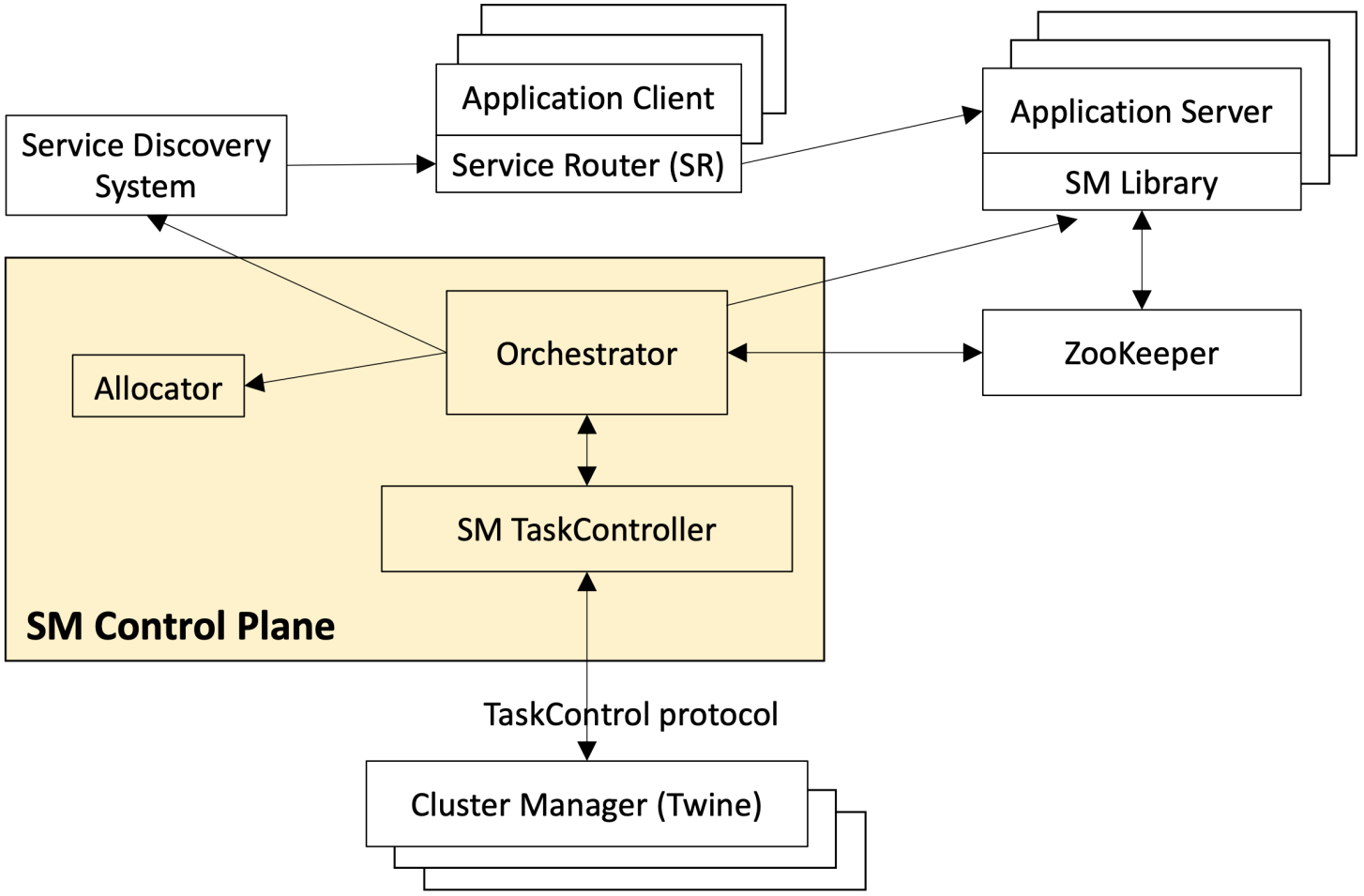


Figure 10. Simplified diagram of the SM ecosystem.

Migration

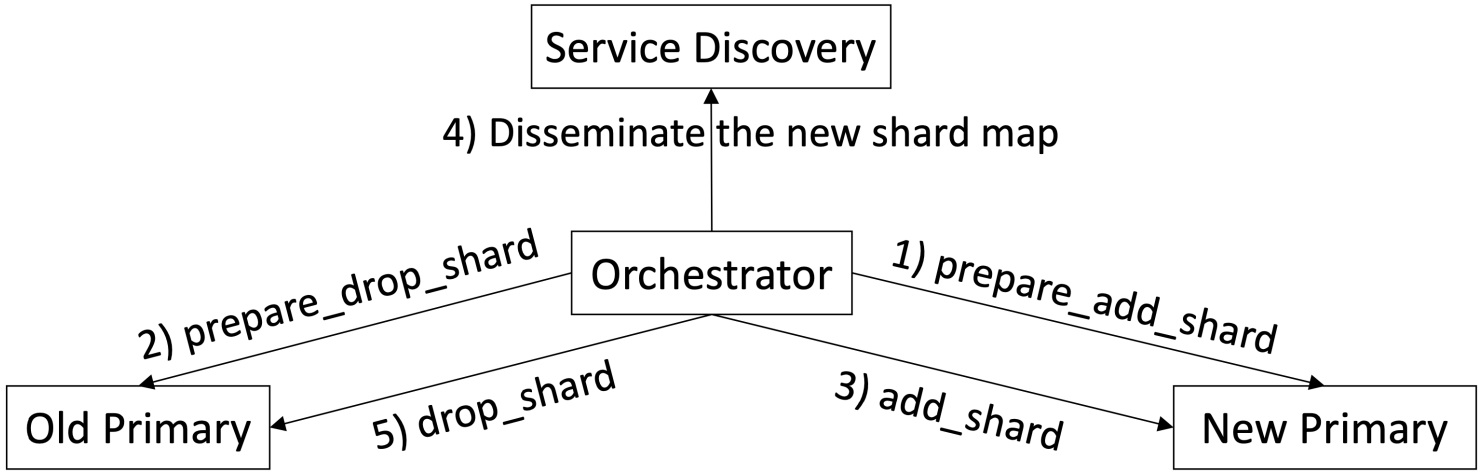


Figure 12. Graceful primary-replica migration.

PLB

```
1: addConstraint(CapacitySpec{.scope="host", .metric="cpu"});
2: addConstraint(CapacitySpec{.scope="rack", .metric="network"});
3: addGoal(BalanceSpec{.scope="host", .metric="cpu''}, 1.0);
4: addGoal(BalanceSpec{.scope="rack", .metric="network''}, 0.5);
5: shardAffinity = {
        {"shard1", "regionA", 1.0},
        {"shard2", "regionB", 2.0},
   }
6: addGoal(AffinitySpec.scope="region", .affinities=shardAffinity)
7: replicaMap = {
        {"shard3_replica1", "shard3"},
        {"shard3_replica2", "shard3"},
   }
8: addGoal(ExclusionSpec.scope="region", .partition=replicaMap);
```

Figure 13. Examples of specifying hard placement constraints and soft goals using the ReBalancer APIs.

Constraint Solver OPT

1. SM divides a large application into *partitions* (§6.1), builds a smaller optimization problem for each partition separately, and solves them on multiple machines in parallel.
2. We configure ReBalancer to use *Local Search* [1] instead of MIP to solve each partition's optimization problem while meeting our rigid time constraint.
3. ReBalancer has accumulated many domain-independent improvements to accelerate local search.
4. SM's allocator provides domain-specific knowledge to guide ReBalancer to further accelerate local search.

Scalibility

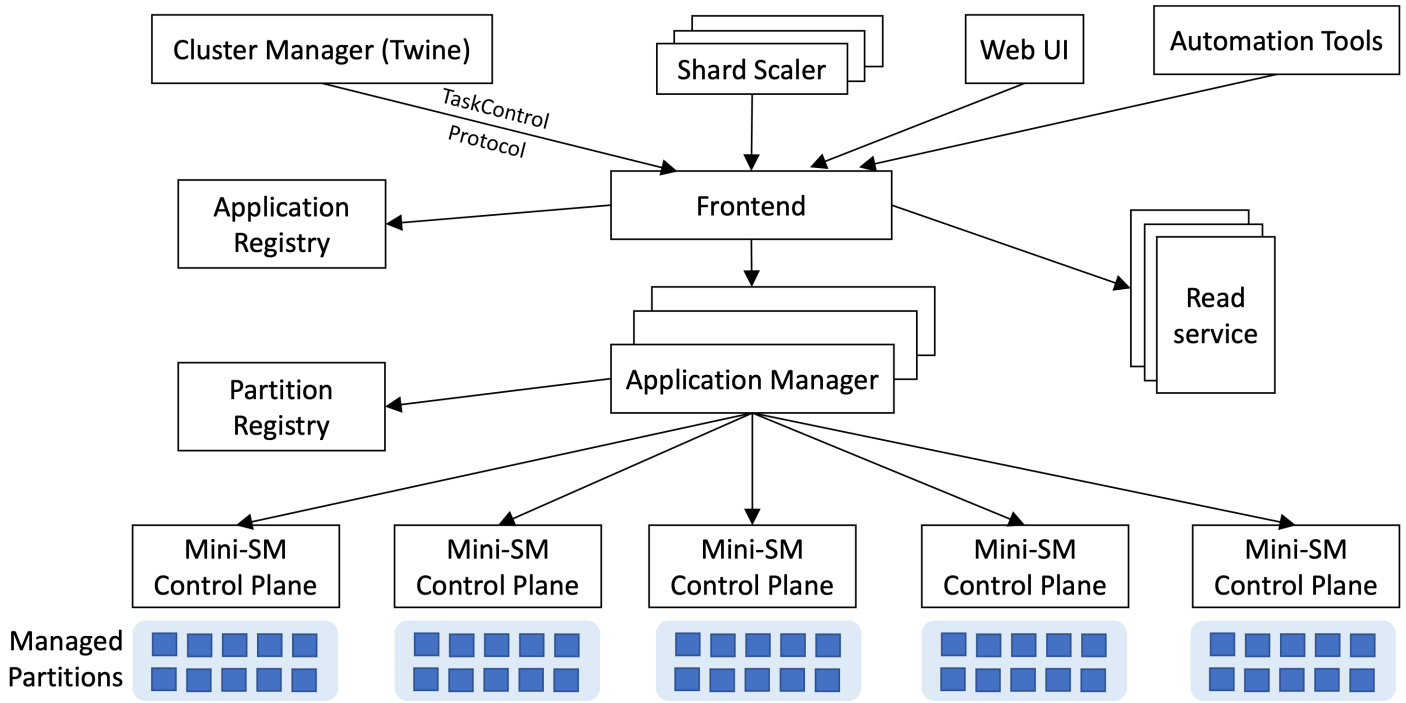


Figure 14. SM's scale-out global control plane.

Evaluation

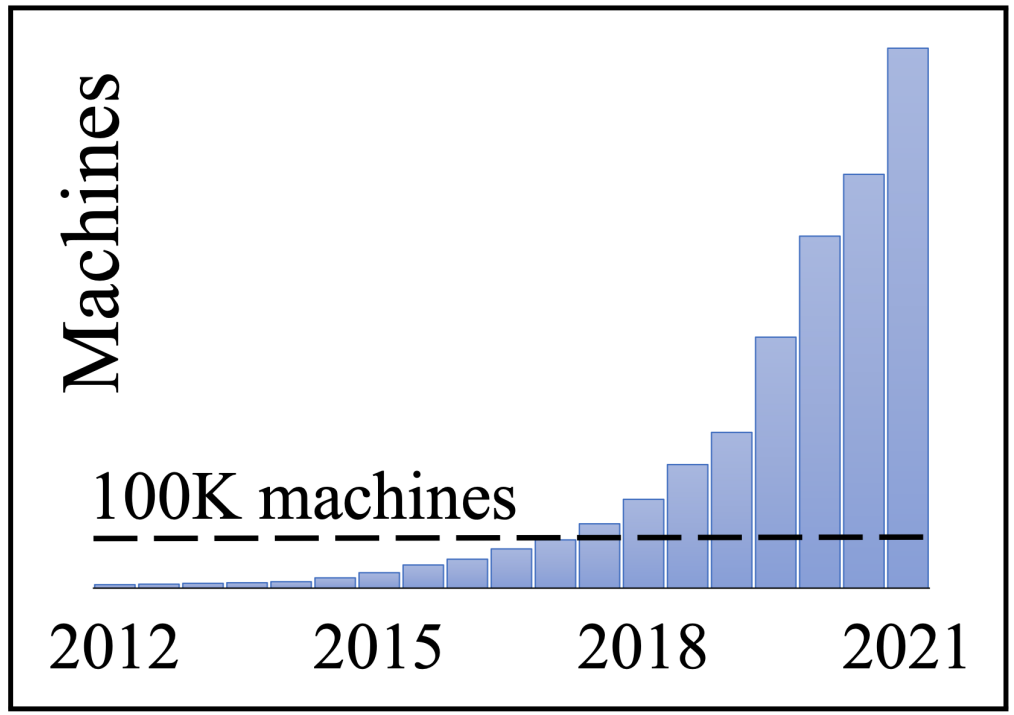


Figure 2. Machines used by SM applications.

Scalability

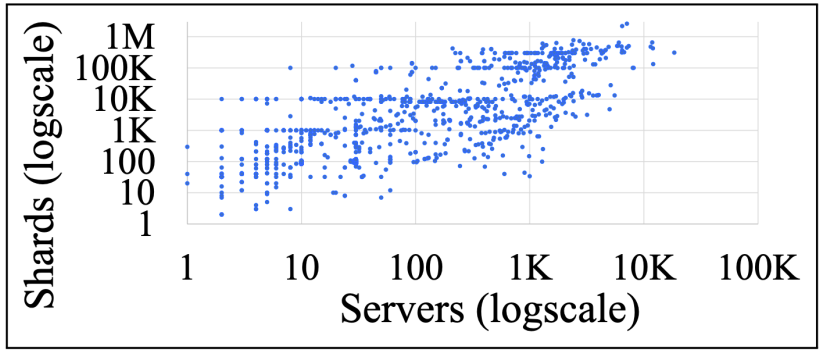


Figure 15. (Production) Scale of SM applications that run in production.

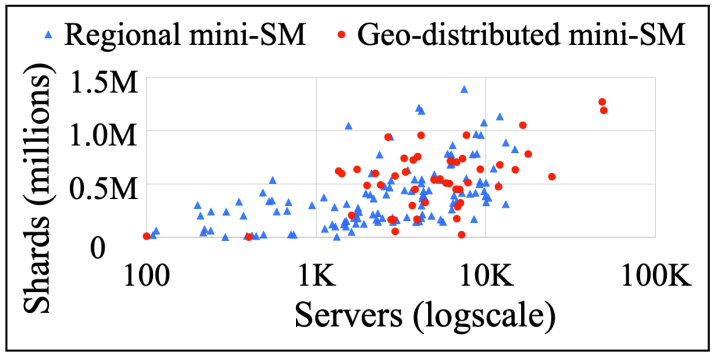


Figure 16. (Production) Scale of mini-SMs that run in production.

Availability

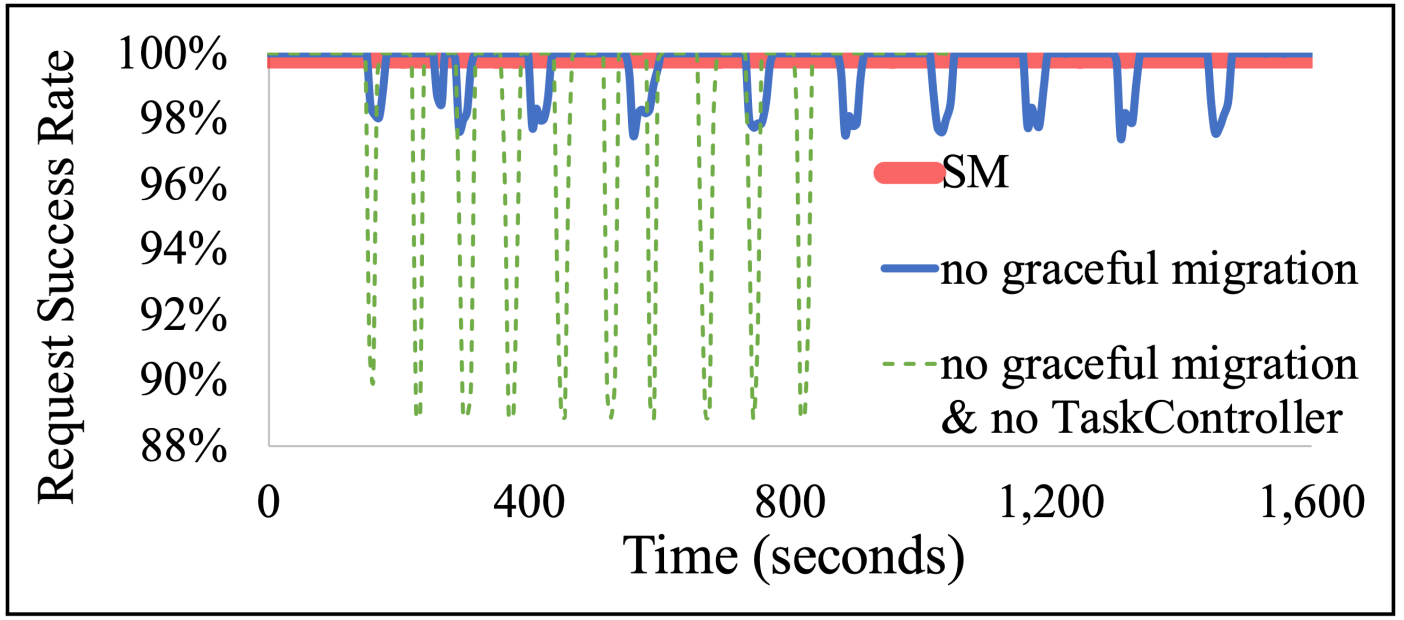


Figure 17. (Experiment) SM upholds availability during software upgrades.

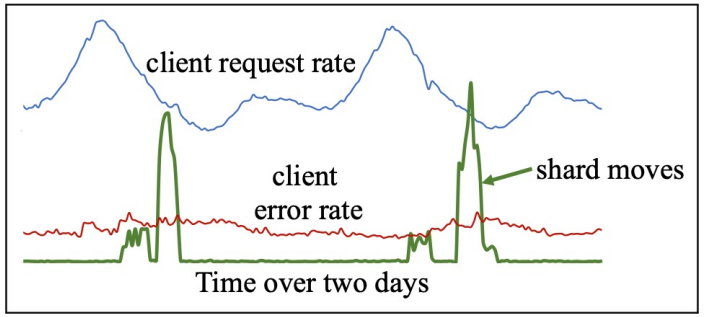


Figure 18. (Production) No increase in client errors during upgrades, thanks to graceful shard migration.

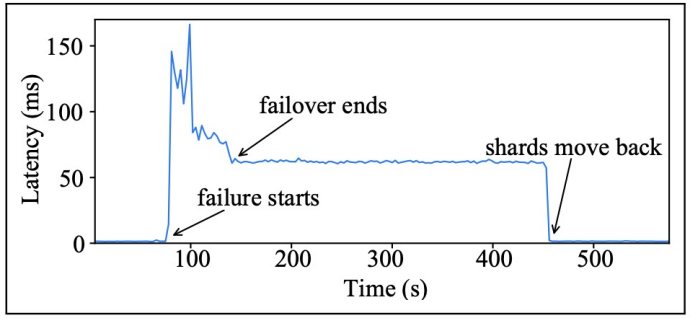


Figure 19. (Experiment) SM migrates a geo-distributed application's shards across regions to handle failures.

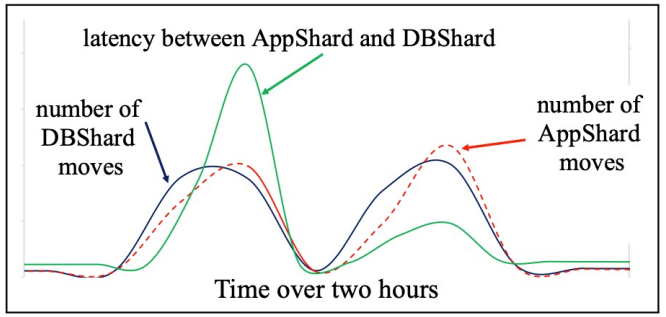


Figure 20. (Production) SM migrates AppShards across regions to follow DBShards to reduce network latency.

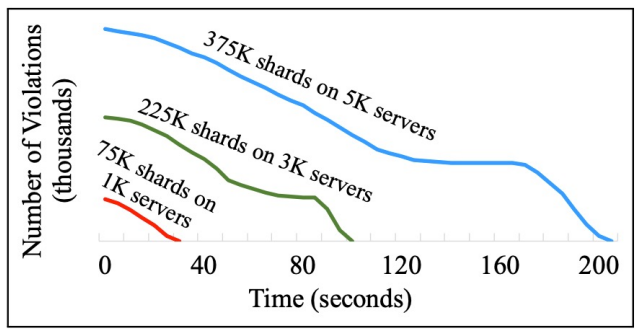


Figure 21. (Experiment) SM allocator scalability w.r.t. the problem size.

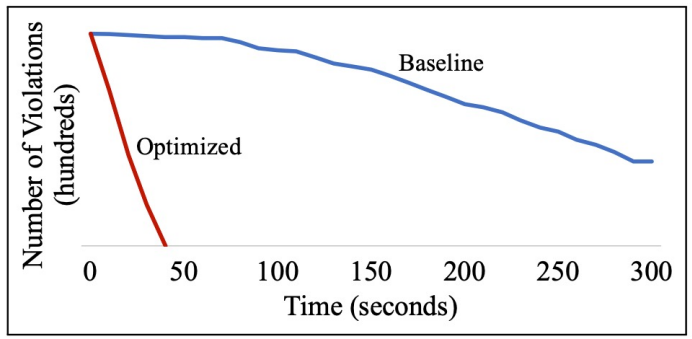


Figure 22. (Experiment) Optimizations help scale the constraint solver.

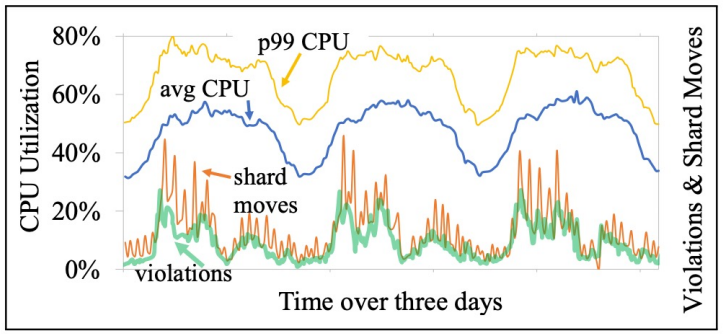


Figure 23. (Production) SM balances load in an ever-changing environment.

Thank you

Presented by Bocheng Cui
PhD student in UNH
noahcui.github.io/info