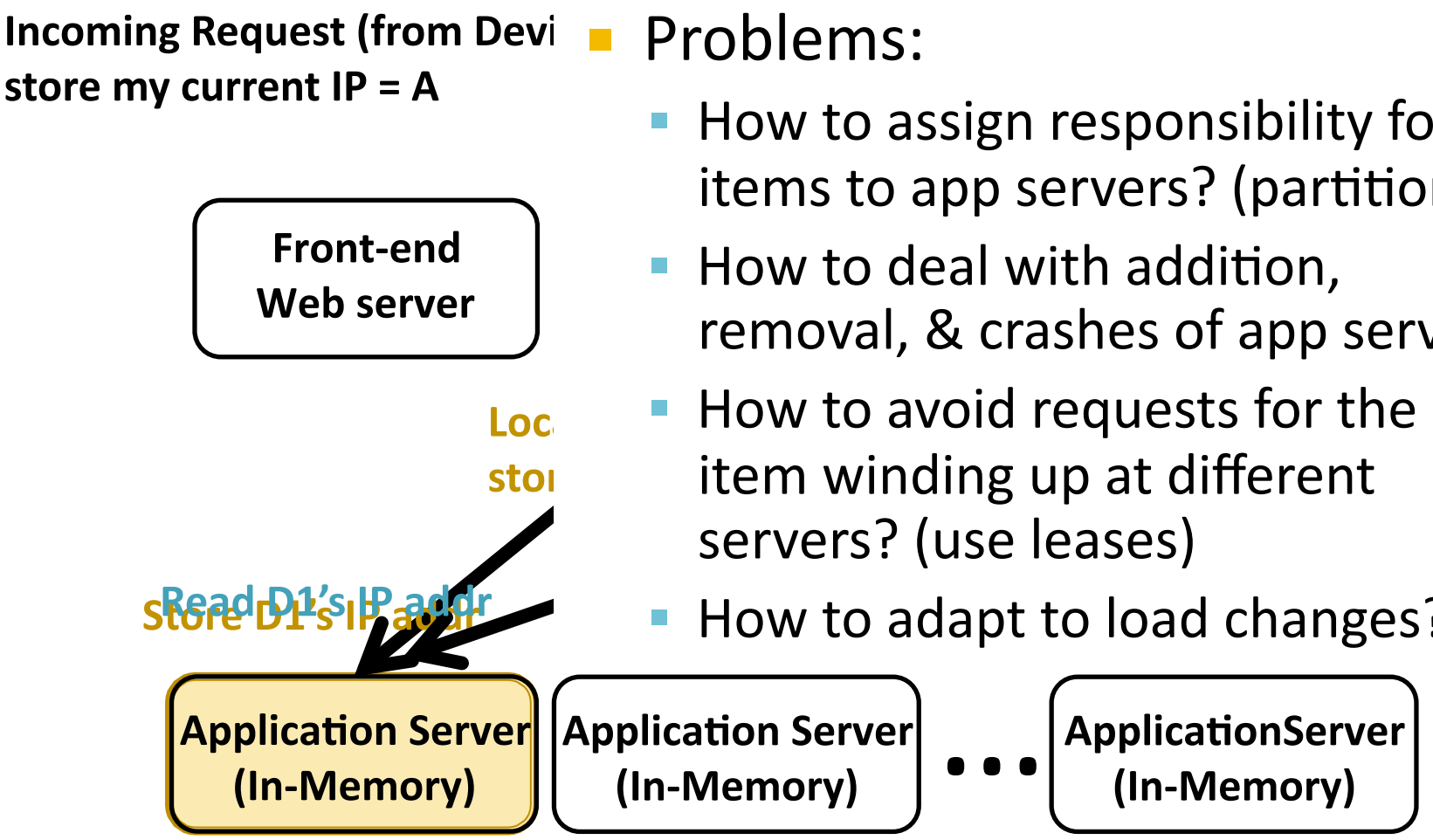
Centrifuge: Integrated Lease Management and Partitioning for Cloud Services

Atul Adya – Google
John Dunagan – Microsoft
Alec Wolman – Microsoft Research

Enabling a Cloud-Based Rendezvous Service



- Problems:
  - How to assign responsibility for items to app servers? (partitioning)
  - How to deal with addition, removal, & crashes of app servers?
  - How to avoid requests for the same item winding up at different servers? (use leases)
  - How to adapt to load changes?

Centrifuge: Reusable Component for Interactive Cloud Services

Targets class of services with these characteristics:

- Interactive (needs low latency)
  - App servers operate on in-memory state
- Application tier operates on cached data: the truth is hosted on clients or back-end storage
- Services use many small objects
- Even the most popular object can be handled by one server
  - Replication not needed to handle load

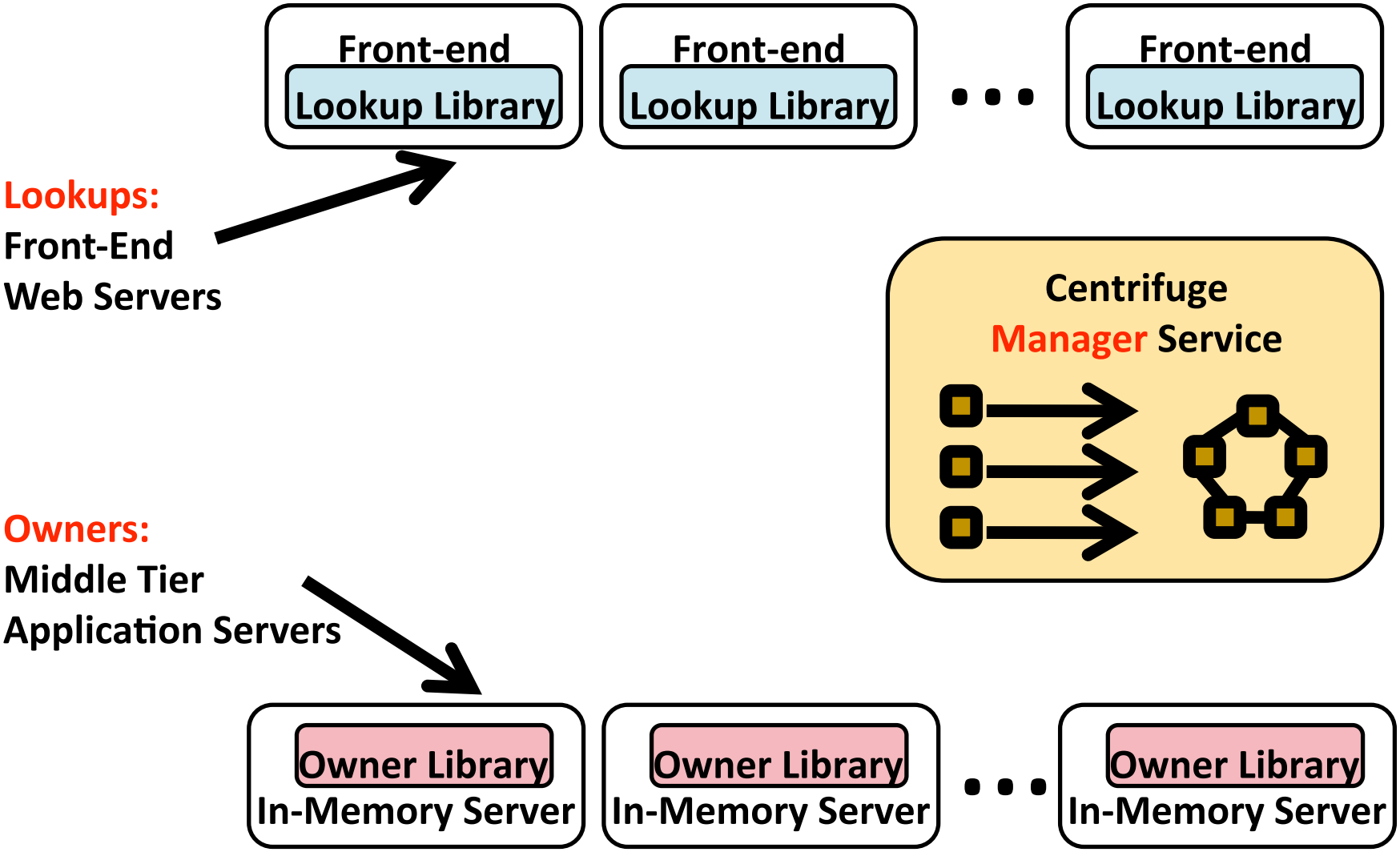
Centrifuge's Contributions

- Prior systems implement leasing and partitioning separately
- We show that integrating leasing and partitioning allows scaling to massive numbers of objects
- This integration requires us to rethink the mechanisms and API for leasing
  - Manager-directed leasing
  - Non-traditional API where clients cannot request leases

Outline

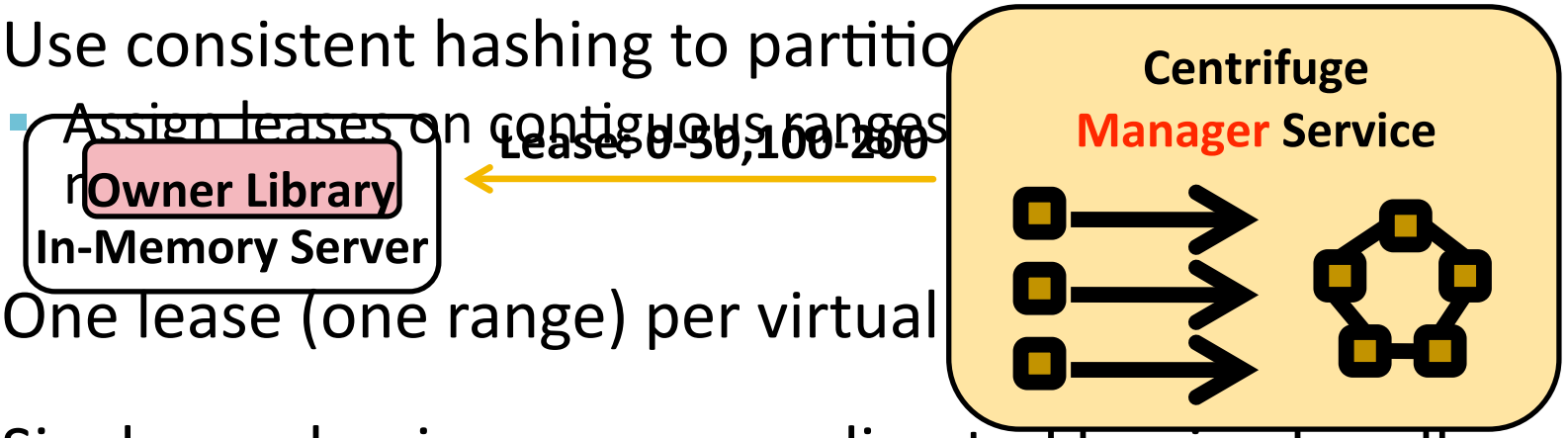
- Centrifuge design
- Centrifuge internals
- Results from live deployment

Centrifuge Architecture



How Does Centrifuge's Leasing Scale?

- Need to issue leases for very large # of objects
  - Lease per object will lead to prohibitive overhead
- Centrifuge manager hands out leases on ranges
- Use consistent hashing to partitio
  - Assign leases on contiguous ranges
- One lease (one range) per virtual
- Single mechanism: manager-directed leasing handles both leasing and partitioning



Clients Do Not Request Leases in the Centrifuge API

Lookup API

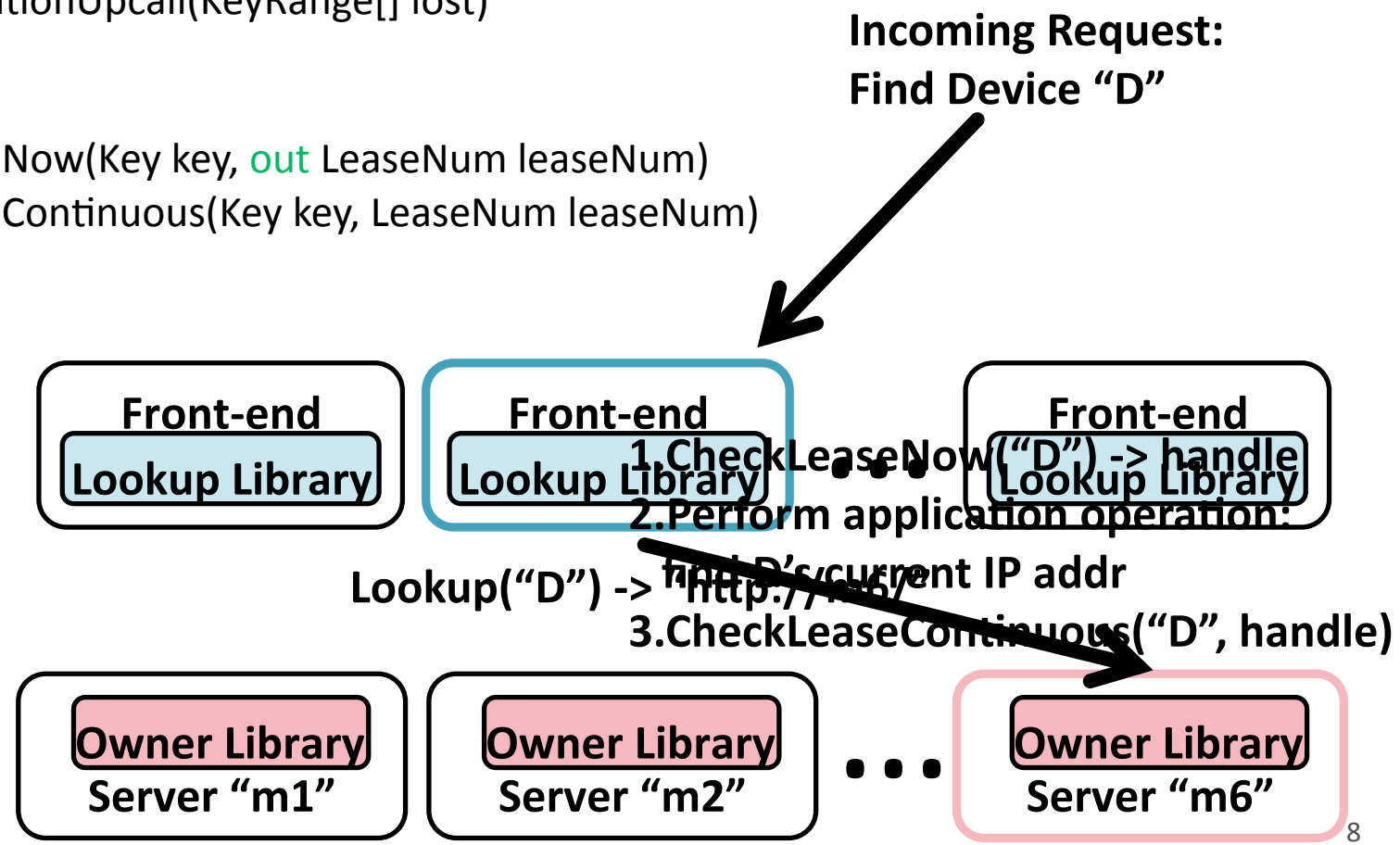
URL Lookup(Key key)

void LossNotificationUpcall(KeyRange[] lost)

Owner API

bool CheckLeaseNow(Key key, out LeaseNum leaseNum)

bool CheckLeaseContinuous(Key key, LeaseNum leaseNum)



Why Recover From Clients (as opposed to Replication)?

- Servers in datacenter environment are stable
- Benefits
  - Much cheaper to avoid holding multiple copies in RAM
  - Avoids complexity/performance issues of quorum protocols
  - Doesn't add extra complexity:
    - Need a mechanism to tolerate correlated failures anyway (e.g. security vulnerabilities, patch installation)
- Cost
  - When an application server crashes, items are not available until clients republish

How Does Centrifuge Support Recovery From Clients?

- When application server crashes, Lookups receive Loss Notifications
  - Indicates which ranges are lost
  - Allows the application to determine which clients should republish their state
- Live Mesh services use this model
  - Rely on clients to recover state

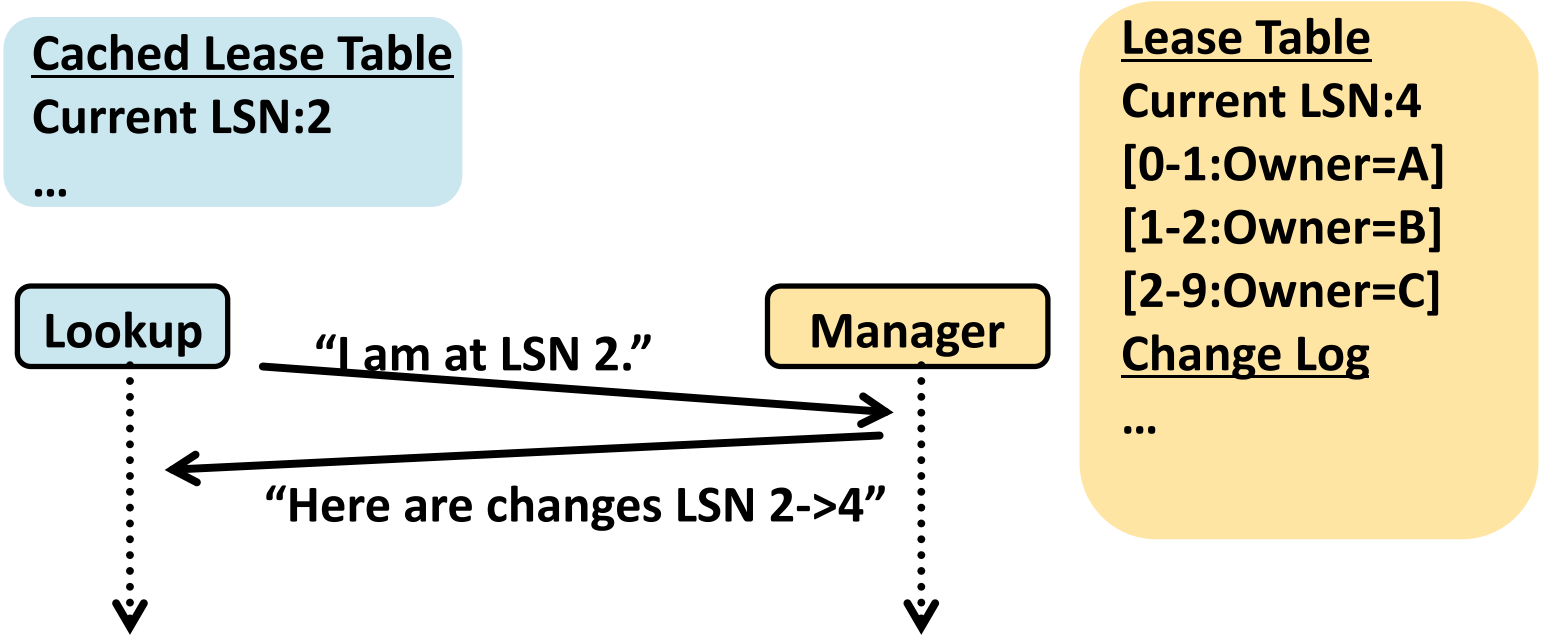
Key Features of Centrifuge

- Partitioning
  - Manager spreads namespace across Owners by assigning leases
- Consistency
  - Leases ensure single-copy guarantee: at any time t, for any key at most one Owner node
- Recovery
  - Loss notifications enable app developer to detect and recover from Owner crashes
- Membership
  - Owners indicate liveness by requesting leases
- Load Balancing
  - Manager rebalances namespace based on reported load

Outline

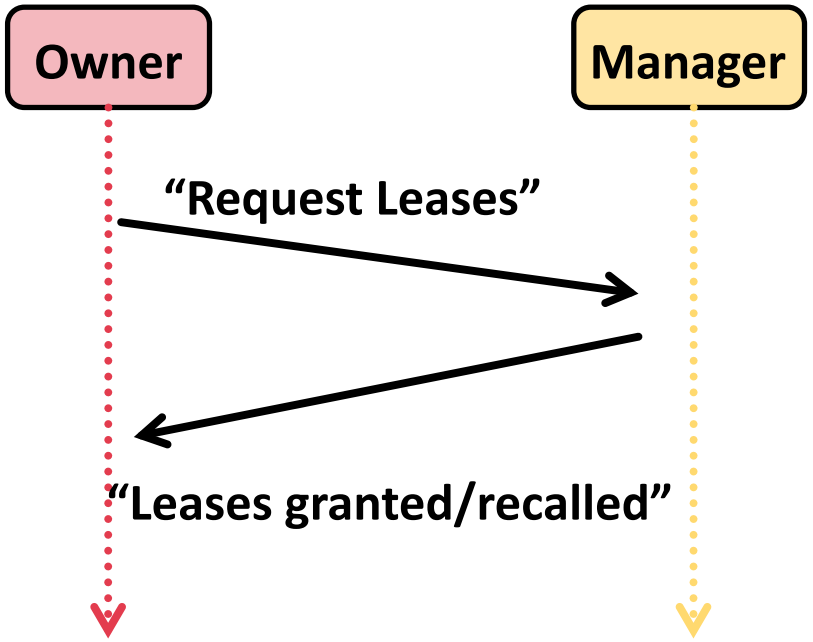
- Centrifuge design
- Centrifuge internals
- Results from live deployment

Lookups Prefetch the Manager's Lease Table



- Incremental protocol to synchronize Lookup and Manager lease tables
- Lookups are fast: no need to contact Manager and incur delay
- Manager load not dependent on incoming request load to Lookups

Lease Protocol is Robust and Safe



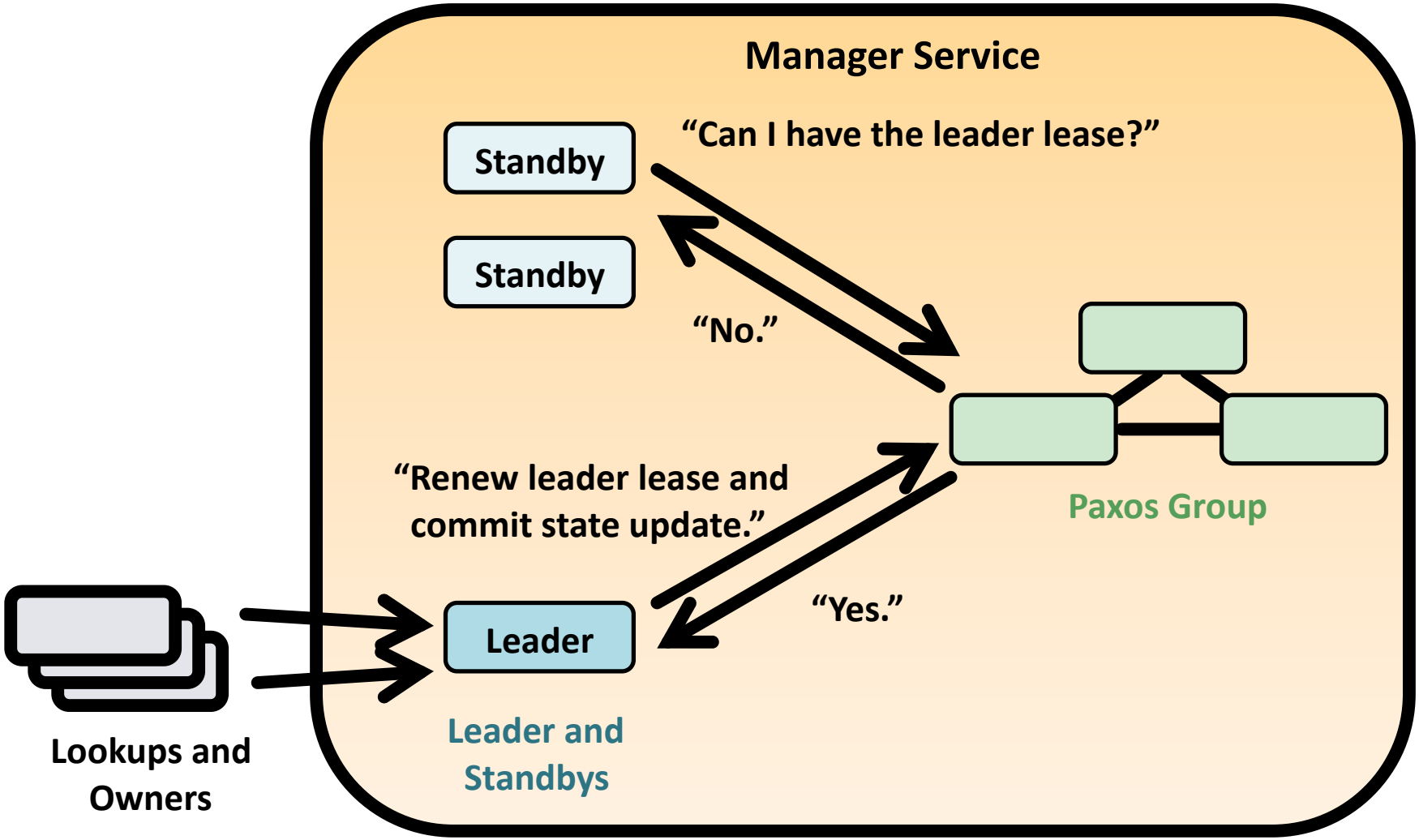
Robustness: Owners have multiple opportunities to retain their leases:

- Leases requested every 15 seconds
- Leases last 60 seconds
- Takes 3 consecutive lost/delayed requests to lose the lease

Safety: owner never thinks it has the lease when the manager disagrees

- Similar to previous lease servers, rely on clock *rate* synchronization

Centrifuge Manager Is Highly Available and Supports Non-Deterministic Code



Scalability of Implementation

- Centrifuge designed to run in a single datacenter
- Scalability target: ~1000 machines in 1 cluster
- Beyond there, scale by deploying multiple clusters

Outline

- Centrifuge design
- Centrifuge internals
- Results from live deployment

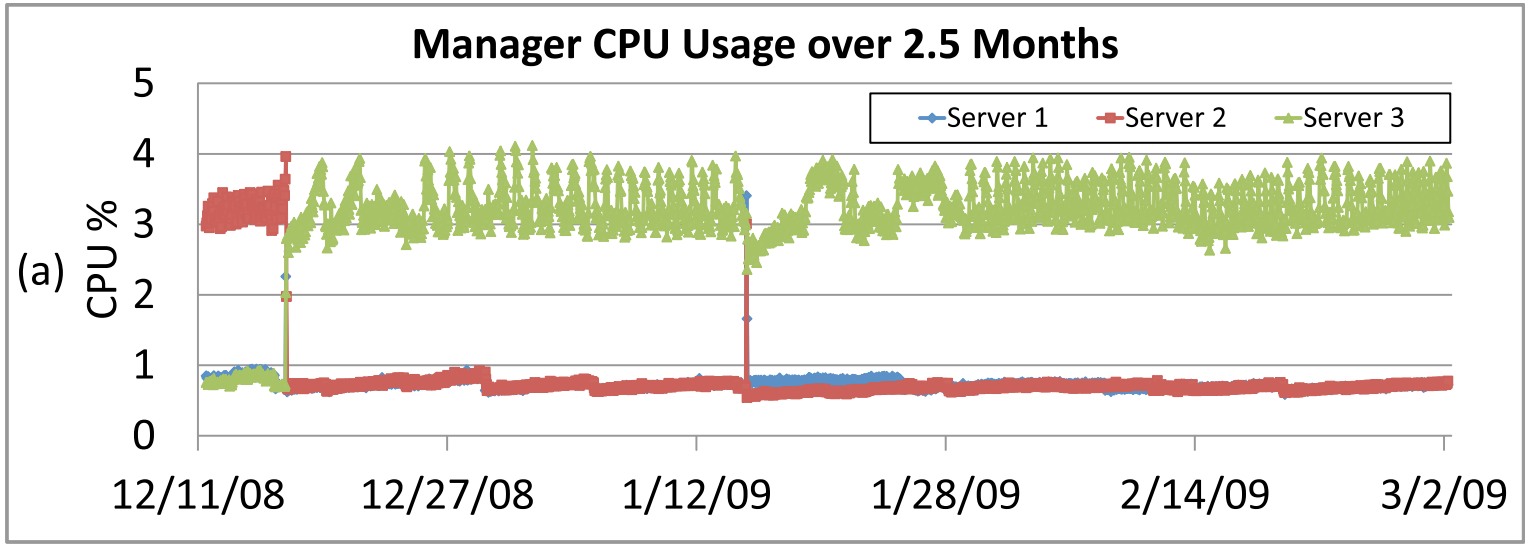
Live Mesh Deployment

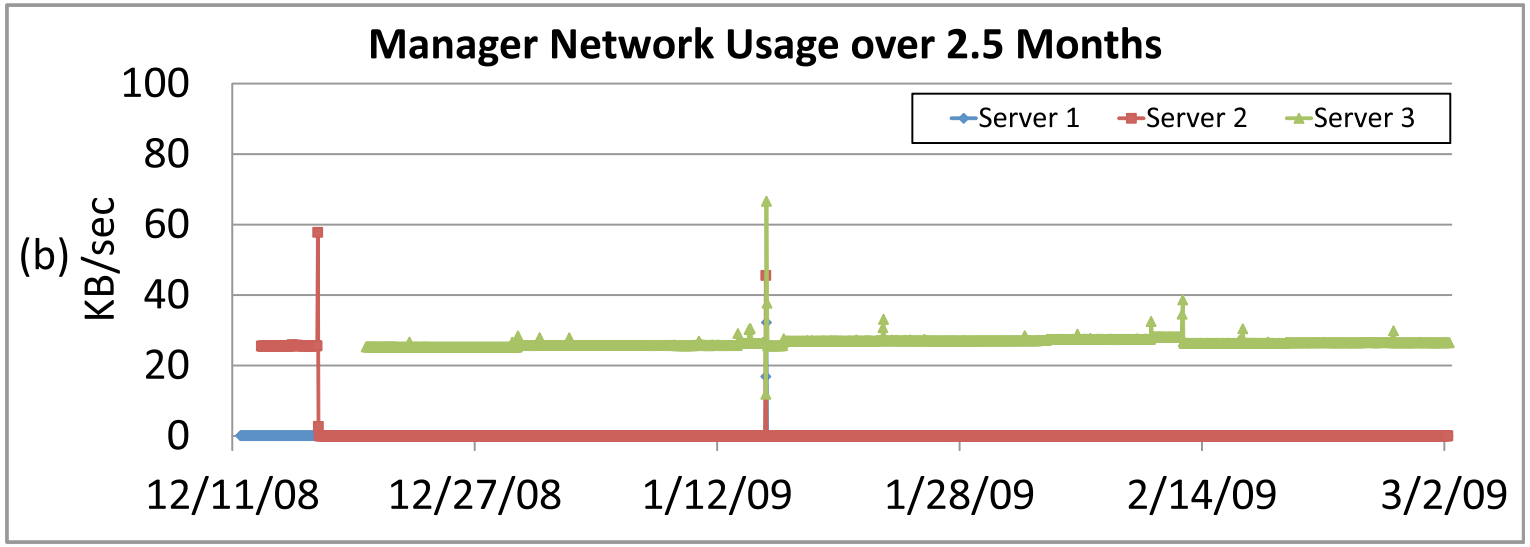
- First deployed in April 2008
- Results cover 2.5 months: Dec ’08 – Mar ‘09
- 1000 Lookups, 130 Owners
- Manager = 8 servers

Questions

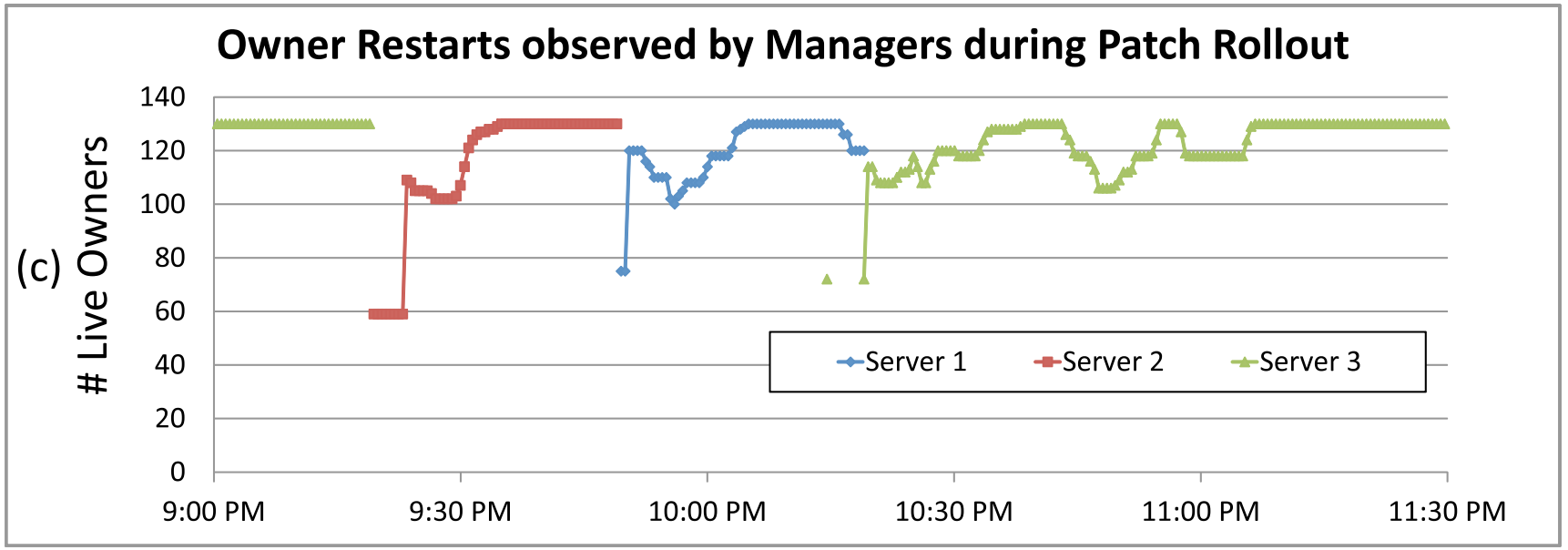
- Is the Centrifuge manager a scalability bottleneck in steady-state?
- How well does Centrifuge handle high-churn events?
- How stable are production servers?

Result: Steady-State Load is Low

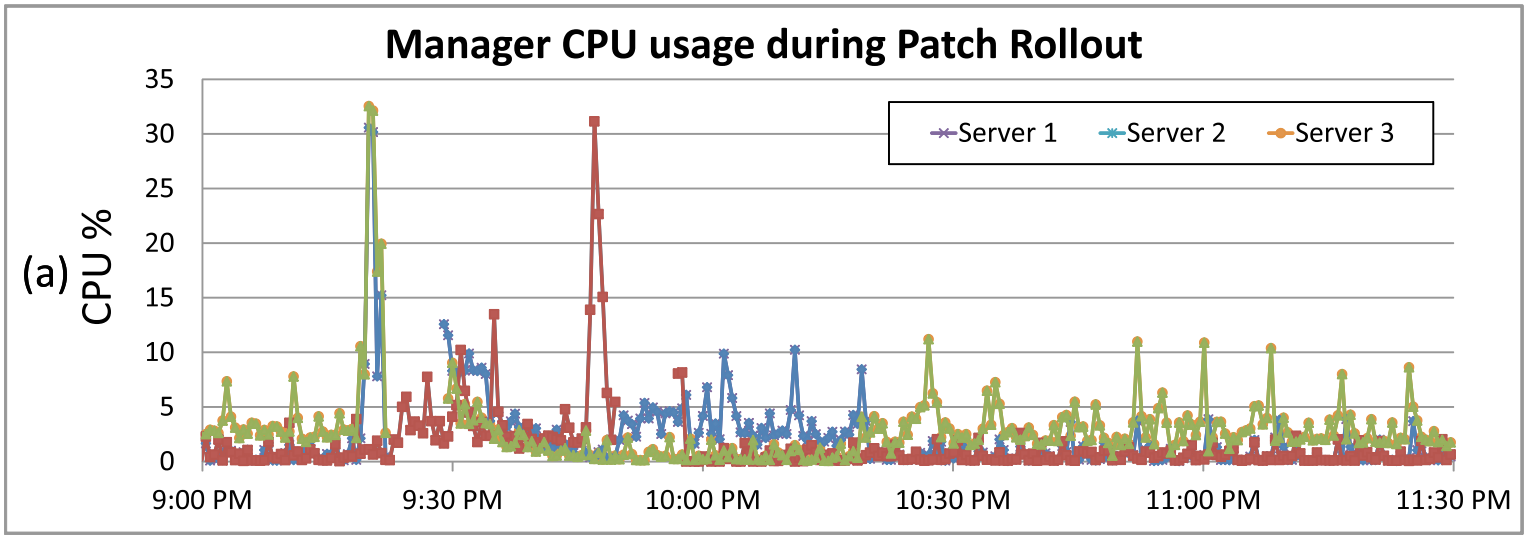


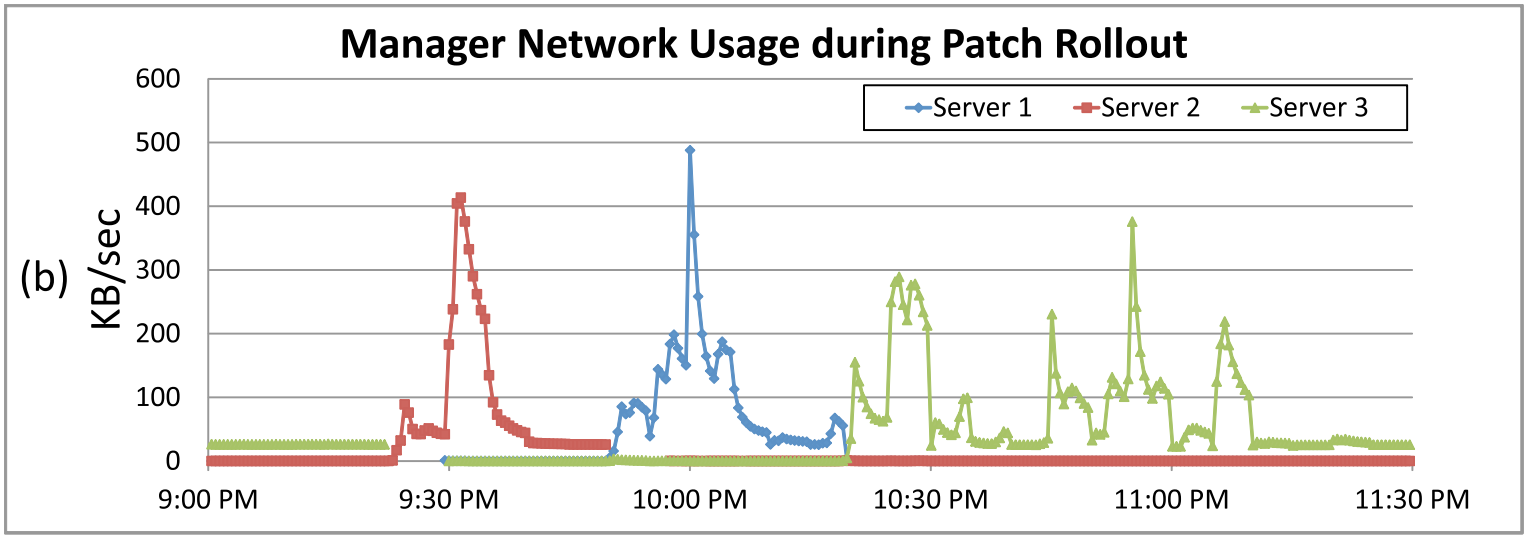


Correlated Failures Do Occur



Result: Even for High Churn, Load is Moderate





Lost-Lease Statistics for 1.5 Months

- From 1/15/09 through 3/2/09, no patch installations
- How stable were the owners during this period?

- Servers are very stable: only 10 lease-loss events
  - 7 cases, servers recovered < 10 minutes
  - 3 cases, servers recovered < 1 hour

Conclusions

- Centrifuge simplifies building scalable application tiers with in-memory state
- Combining leasing and partitioning leads to a simple and powerful protocol
- Deployed within Live Mesh since April 2008, in use by 5 different Live Mesh Services
- Data center server stability enables the single copy in RAM w/loss notifications