What We Have Learned From RAMCloud (So Far)

John Ousterhout

Stanford University

(with Asaf Cidon, Ankita Kejriwal, Diego Ongaro, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang)

Introduction

A collection of broad conclusions we have reached during the RAMCloud project:

- Randomization plays a fundamental role in large-scale systems
- Need new paradigms for distributed, concurrent, fault-tolerant software
- Exciting opportunities in low-latency datacenter networking
- Layering conflicts with latency
- Don’t count on locality
- Scale can be your friend

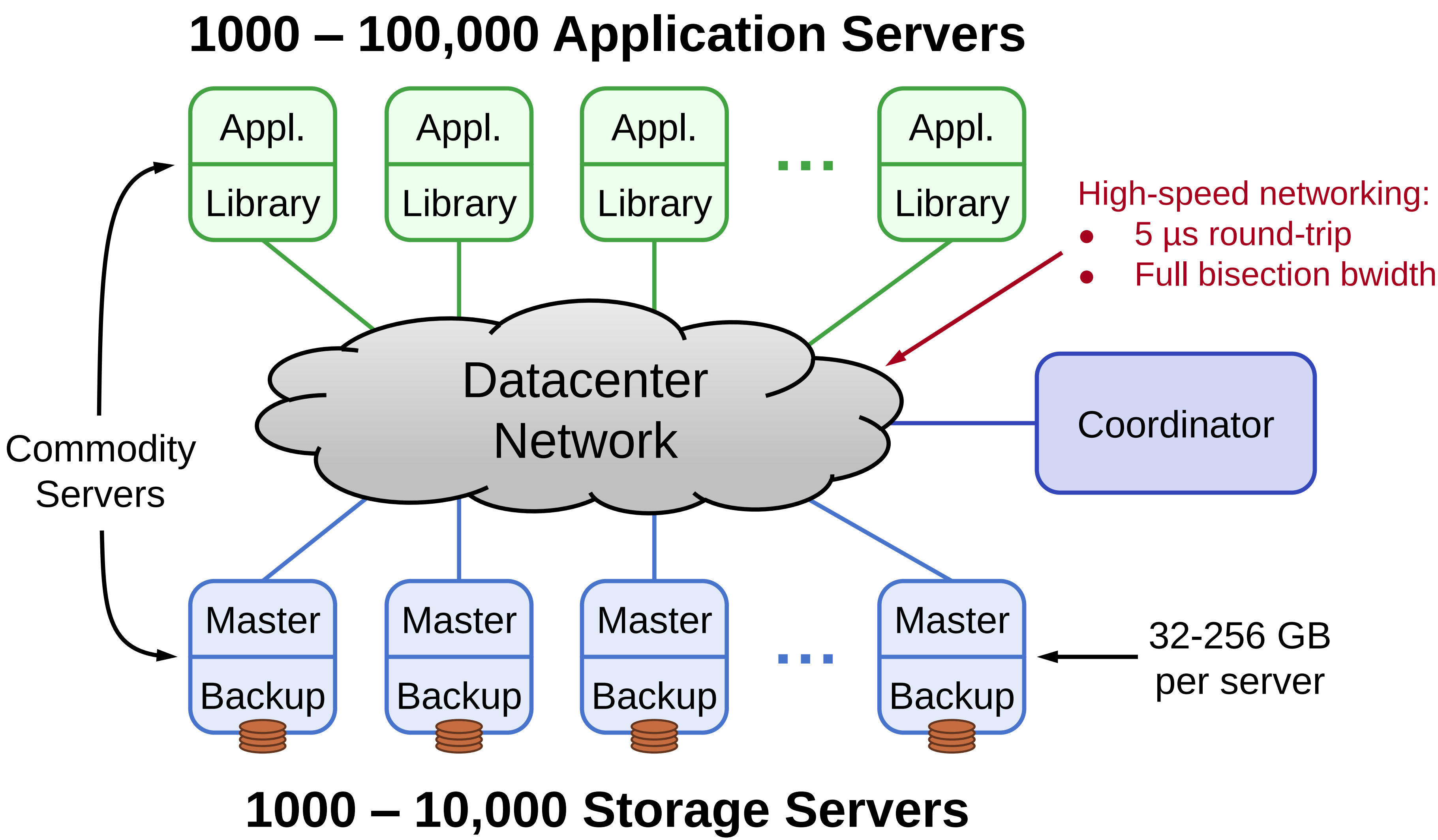
RAMCloud Overview

Harness full performance potential of large-scale DRAM storage:

- **General-purpose key-value storage system**
- **All data always in DRAM (no cache misses)**
- **Durable and available**
- **Scale: 1000+ servers, 100+ TB**
- **Low latency: 5-10µs remote access**

Potential impact: enable new class of applications

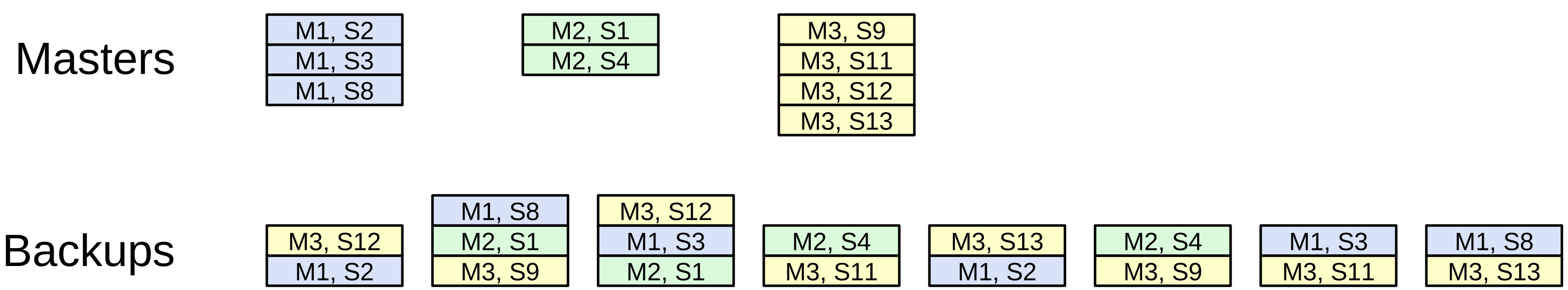
RAMCloud Architecture



Randomization

Randomization plays a fundamental role in large-scale systems

- **Enables decentralized decision-making**
- **Example: load balancing of segment replicas. Goals:**
  - Each master decides where to replicate its own segments: no central authority
  - Distribute each master's replicas uniformly across cluster
  - Uniform usage of secondary storage on backups

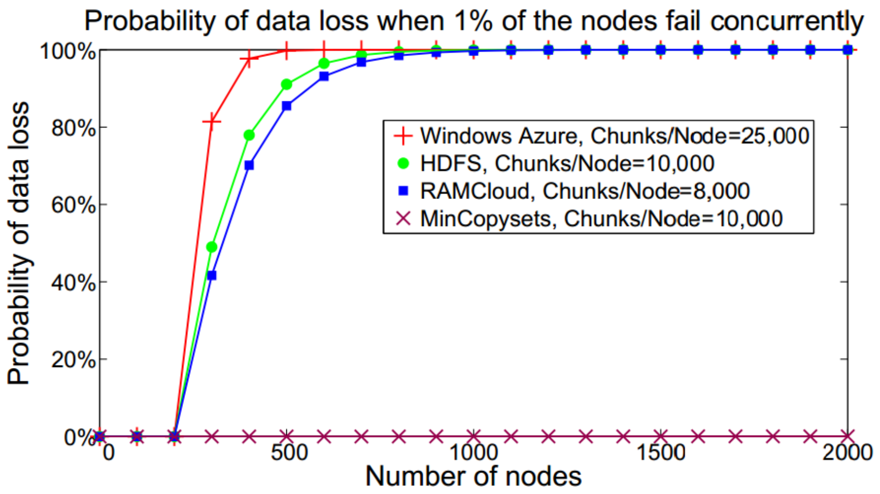


Randomization, cont'd

- **Choose backup for each replica at random?**
  - Uneven distribution: worst-case = 3-5x average
- **Use Mitzenmacher's approach:**
  - Probe several randomly selected backups
  - Choose most attractive
  - Result: distribution is nearly uniform

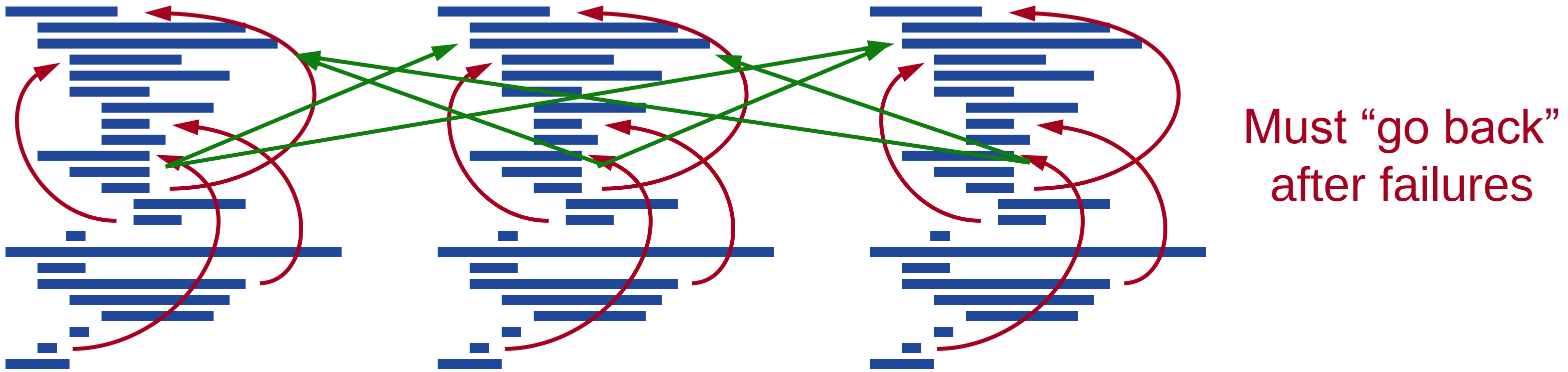
Sometimes Randomization is Bad!

- **Select 3 backups for segment at random?**
- **Problem:**
  - In large-scale system, any 3 machine failures results in data loss
  - After power outage, ~1% of servers don't restart
  - Every power outage loses data!
- **Solution: derandomize backup selection**
  - Pick first backup at random (for load balancing)
  - Other backups deterministic (replication groups)
  - Result: data safe for hundreds of years
  - (but, lose more data in each loss)



DCFT Code is Hard

- **RAMCloud often requires code that is distributed, concurrent, and fault tolerant:**
  - Replicate segment to 3 backups
  - Coordinate 100 masters working together to recover failed server
  - Concurrently read segments from ~1000 backups, replay log entries, re-replicate to other backups
- **Traditional imperative programming doesn't work**



- **No common patterns, each system built from scratch**

DCFT Code: Need Pattern(s)

- **Emerging pattern in RAMCloud subsystems: rules**

```
if no server assigned for replica
then select backup
```

Predicate on state

Action

```
if header committed, unreplicated data,
   no RPC outstanding
then start replication RPC
```

```
if ...
then ...
```

- **Rule = predicate + action**
  - Actions short, nonblocking, predictable: no faults within an action
  - Rule execution order unpredictable: reflects faults, etc.
- **Rules organized into higher-level structures: tasks, pools**
- **These ideas are still evolving**

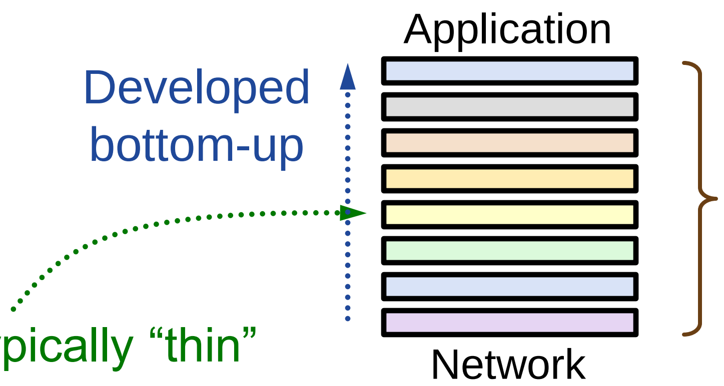
Low-Latency Networking

- **Datacenter evolution, phase #1: scale**
- **Datacenter evolution, phase #2: latency**
  - Typical round-trip in 2010: 300µs
  - Feasible today: 5-10µs
  - Ultimate limit: < 2µs
- **No fundamental technological obstacles, but need new architectures:**
  - Must bypass OS kernel
  - New integration of NIC into CPU
  - New datacenter network architectures (no buffers!)
  - New network/RPC protocols: user-level, scale, latency (1M clients/server?)

Layering Conflicts With Latency

Most obvious way to build software: lots of layers

Developed bottom-up



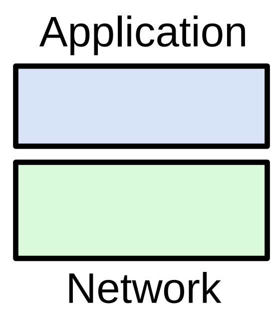
Layers typically “thin”

Problems:

- Complex
- High latency

For low latency, must rearchitect with fewer layers

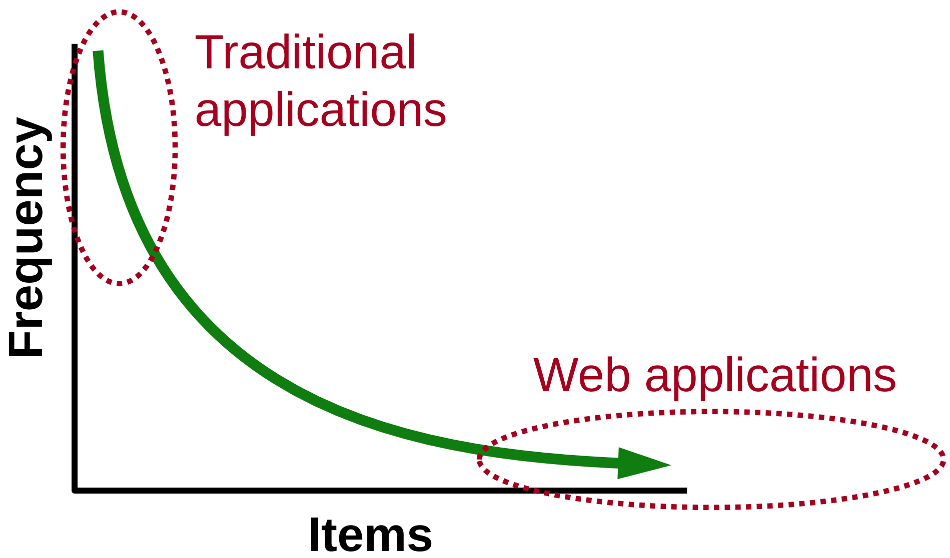
Harder to design (top-down and bottom-up)



But, better architecturally (simpler)

Don’t Count On Locality

- **Greatest drivers for software and hardware systems over last 30 years:**
  - Moore’s Law
  - Locality (caching, de-dup, rack organization, etc. etc.)
- **Large-scale Web applications have huge datasets but less locality**
  - Long tail
  - Highly interconnected (social graphs)



Make Scale Your Friend

- **Large-scale systems create many problems:**
  - Manual management doesn't work
  - Reliability is much harder to achieve
  - "Rare" corner cases happen frequently
- **However, scale can be friend as well as enemy:**
  - RAMCloud fast crash recovery
    - Use 1000's of servers to recover failed masters quickly
    - Since crash recovery is fast, "promote" all errors to server crashes
  - Windows error reporting (Microsoft)
    - Automated bug reporting
    - Statistics identify most important bugs
    - Correlations identify buggy device drivers
    - Automatic installation of fixes

Conclusion

Build big => learn big

- **My pet peeve: too much “summer project research”**
  - 2-3 month projects
  - Driven by conference paper deadlines, not technical goals
  - Superficial, not much deep learning
- **Trying to build a large system that really works is hard, but intellectually rewarding:**
  - Exposes interesting side issues
  - Important problems identify themselves (recurrences)
  - Deeper evaluation (real use cases)
  - Shared goal creates teamwork, intellectual exchange
  - Overall, deep learning