Low Latency RPCs

RAMCloud Design Review, April 1, 2010

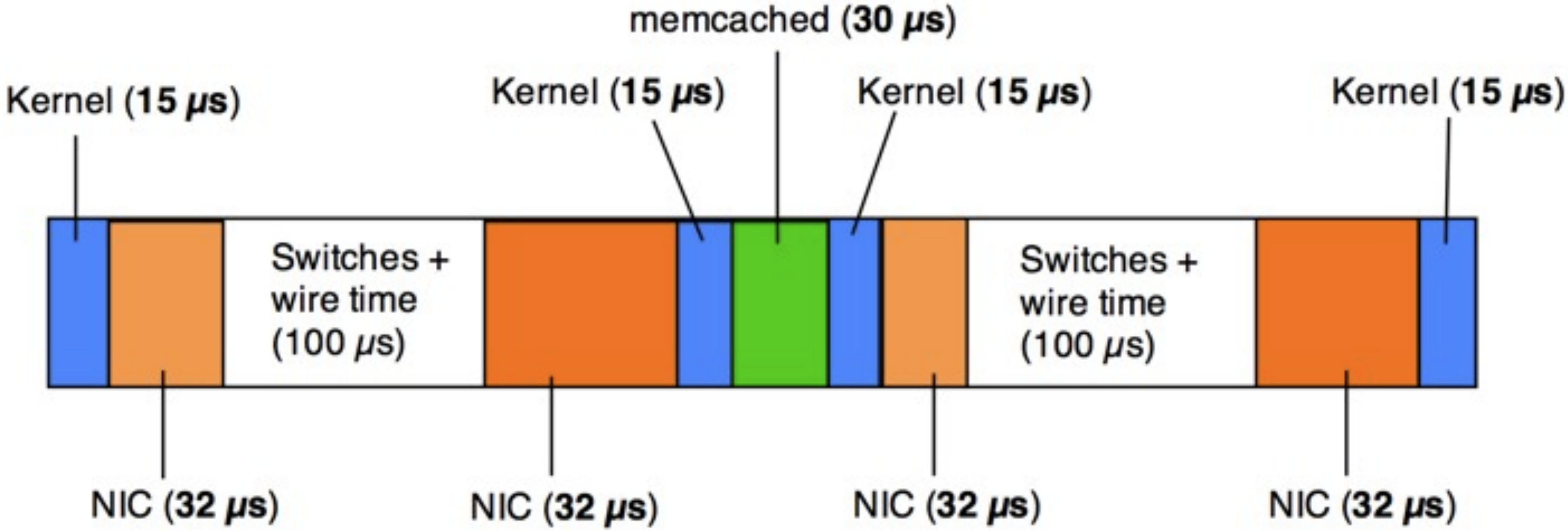
Aravind Narayanan
Stanford University

Overview

	1985	2010	Improvement
CPU Speed	12 Mhz	4 Ghz	333 x
Bandwidth	10 Mbps	10 Gbps	1000 x
Latency	2 ms	500 µs	4 x

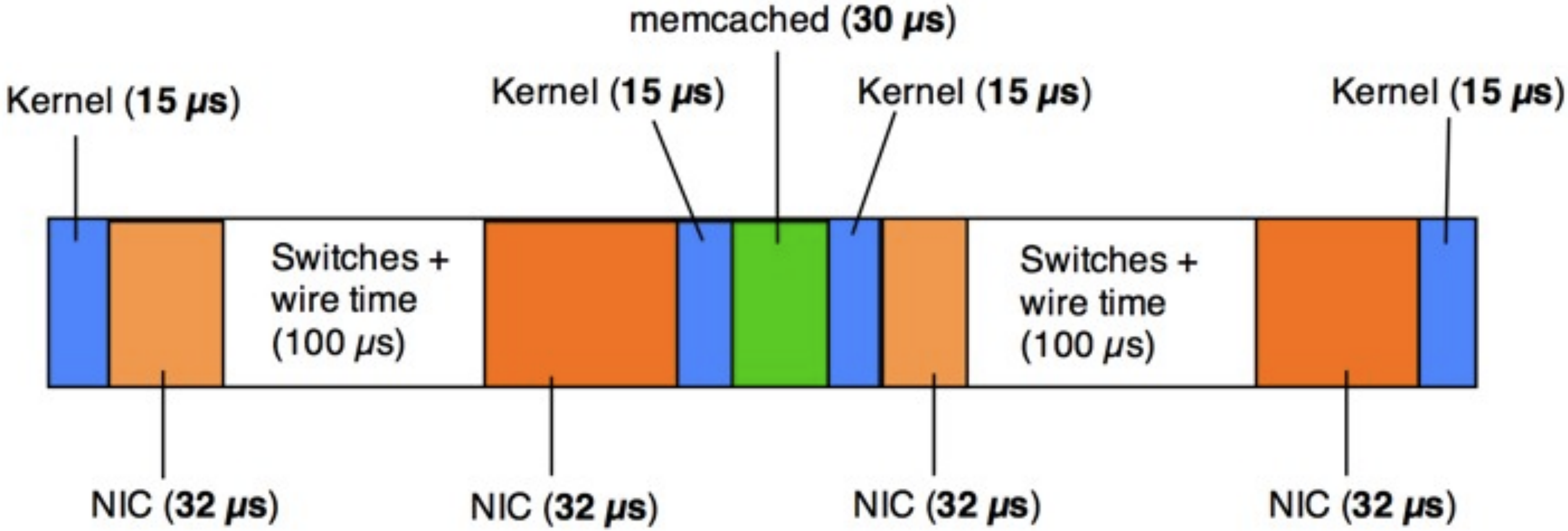
- Goal: 5-10 µs RPCs
- Experimental result: 11 µs RTT
- Three parts: current sources of latency, experimental results, RPC system

Baseline Performance



- **Experiment Setup:**
  - Intel Xeon - 3.4 Ghz
  - Intel 82541GI Gigabit NICs
  - Standard Linux Kernel with UDP
  - Switches + wire time
    - Estimated using a typical data center
    - 10 switches

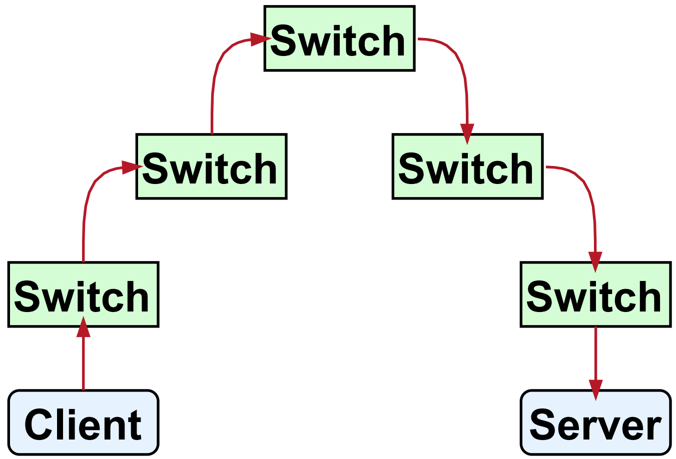
Baseline Performance



- **Total time: ~ 400 μs**
- **Main sources of latency:**
  - Switch + wire time: 200 μs
  - NIC: 128 μs
  - Kernel: 60 μs
  - memcached: 30 μs

Causes: Data center network time

- **Network latency: 150 - 300 µs**
  - 10 - 30 µs per switch, 5 switches each way
- **Latest Arista product:**
  - 0.6 µs per switch
  - Need cut-through routing, congestion management
- **Hoping for help!**
  - Not RAMCloud's goal



Causes: NIC Hardware

- **Most hardware is designed for throughput, not latency**
- **Interrupt coalescing/throttling: ~ 64 µs one way**
  - Design the NIC to avoid live lock, and to lower CPU utilization
  - Optimize for bandwidth
  - Default setting!

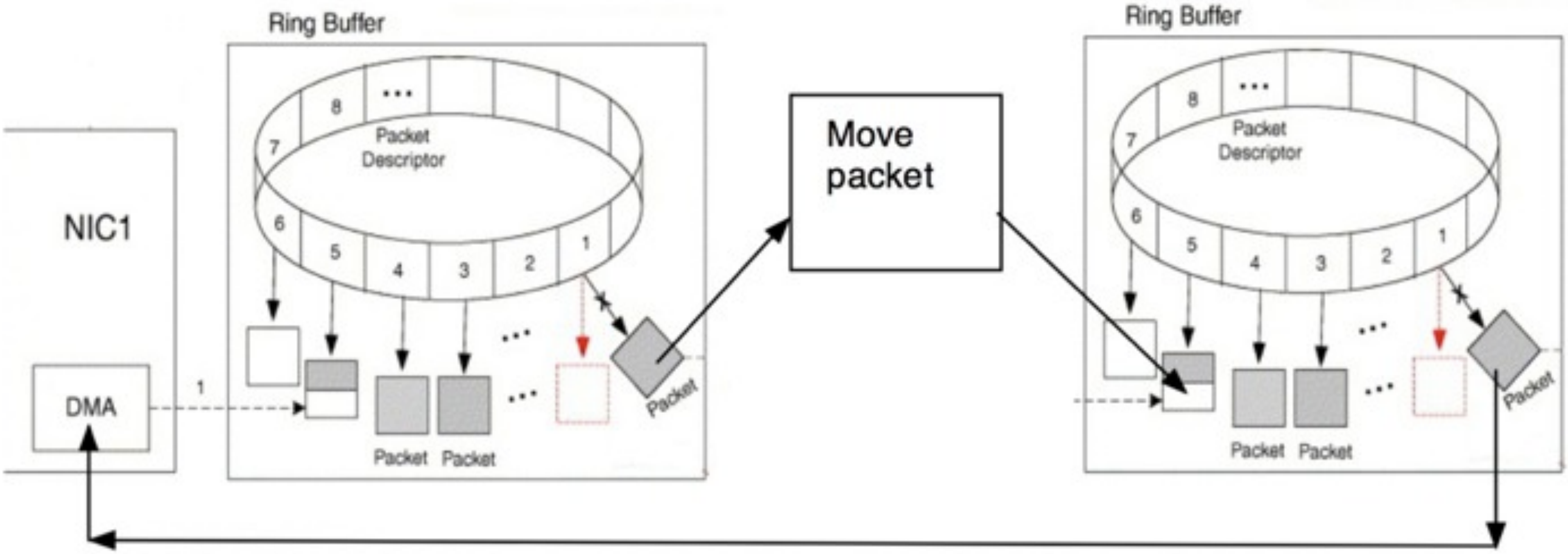
Causes: Software

- **Kernel network stack**
  - Packet takes **15 µs** to bubble through the kernel (each way)
  - **60 µs** of overhead per RTT!
- **Protocol overhead**
  - TCP is inherently slow
    - requires a lot of processing and state
  - IP: options may add processing time
- **Unnecessary intermediate copies**
  - From user-space to kernel
- **CPU scheduling/preemption**
- **Context switches**

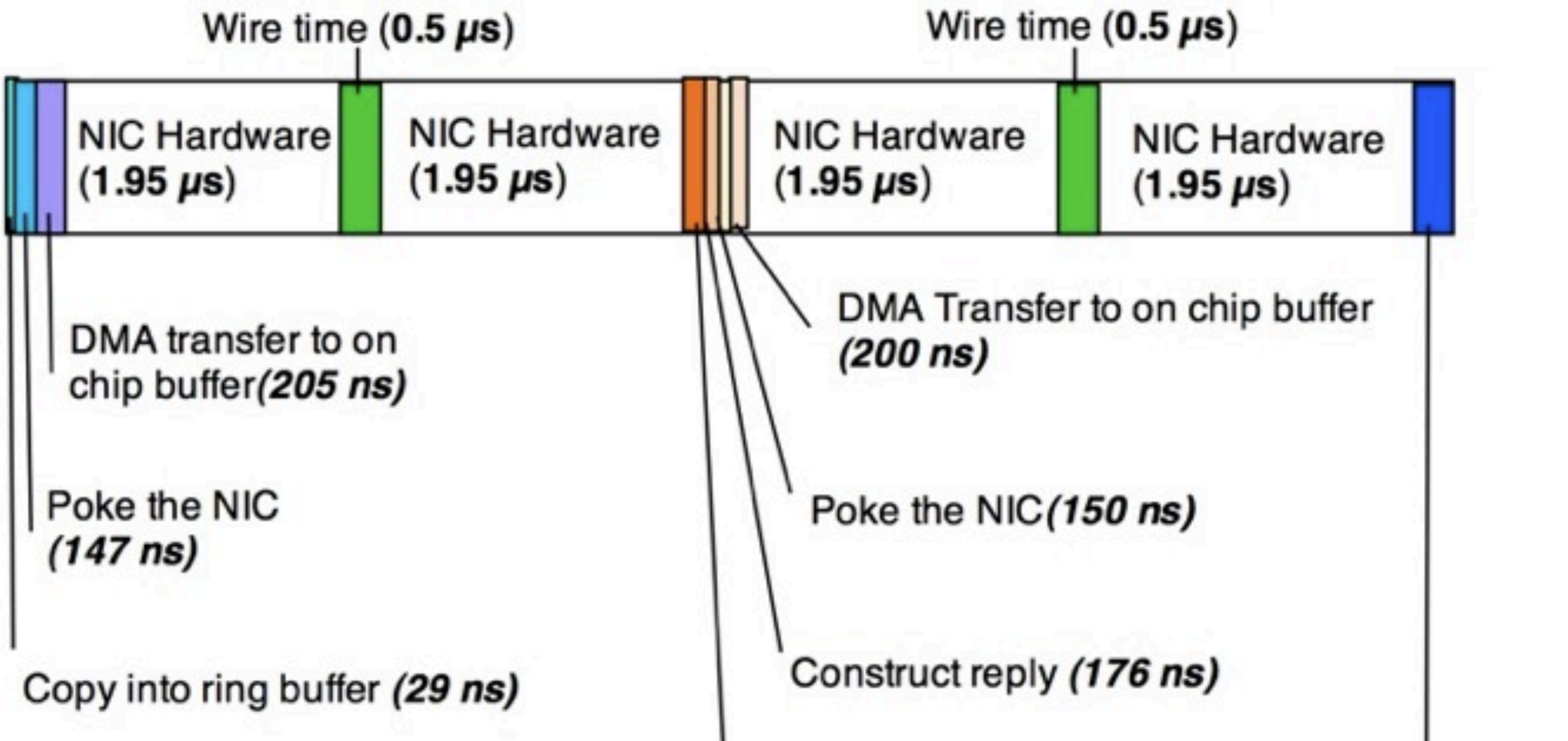
Radical Experiment

- **Part 1: Tune the NIC**
  - Turn off interrupt coalescing, saving ~ 128 μs
  - Poll the NIC with a dedicated core, no interrupts!
- **Part 2: Rip out unnecessary layers of software**
  - Map the NIC directly into user space
    - User software can access NIC's registers and ring buffers
  - Eliminate networking layer
  - Avoids unnecessary copying
  - No kernel/context switching overhead
- **Part 3: Eliminate protocol overhead**

NIC Ring Buffers



Experimental Result



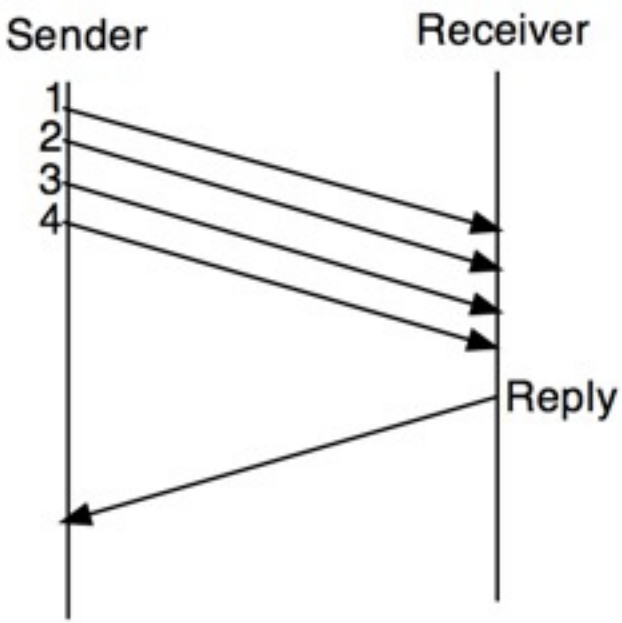
- **11 μs RTT**
- **< 1 μs of software overhead**
- **NIC hardware latency: 1.95 μs x 4**
- **Future experiment: Test over 10 Gig NICs**

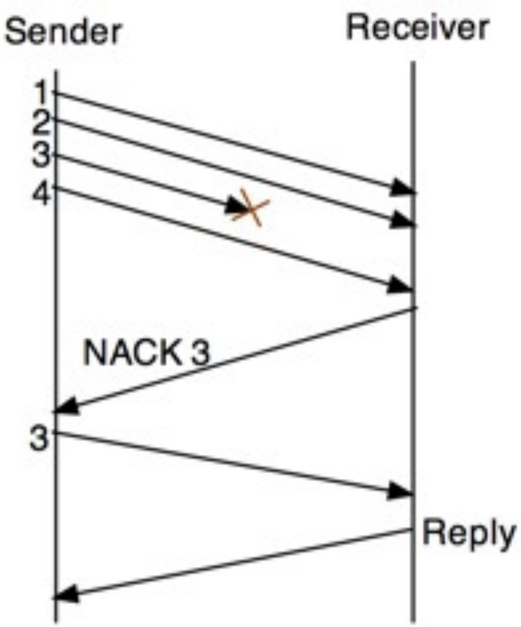
RPC System

- **Build a real system**
  - As fast as weird experimental version?
- **Requirements:**
  - Reliability
  - Handles messages larger than 1 frame
  - Retain single copy
    - From ring buffer to server's log (on receive)

RPC System

- **The reply is the ACK for most RPCs**
  - RPCs are so fast that it makes no sense to ACK fragments
- **Blast protocol**
  - Send all fragments of an RPC at once, without waiting for ACK
  - Selective NACKs
  - Too slow to retransmit the whole packet

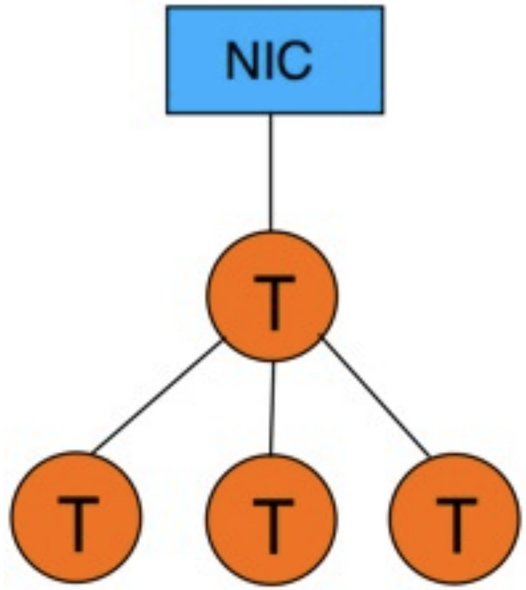




Threading model

- **Increased parallelism:**
  - More cores per chip
  - More threads per core
- **Use multiple threads to increase throughput**
  - Associated dispatching/synchronization overheads
- **On server, how to distribute requests among available worker threads?**
- **Several possible designs**

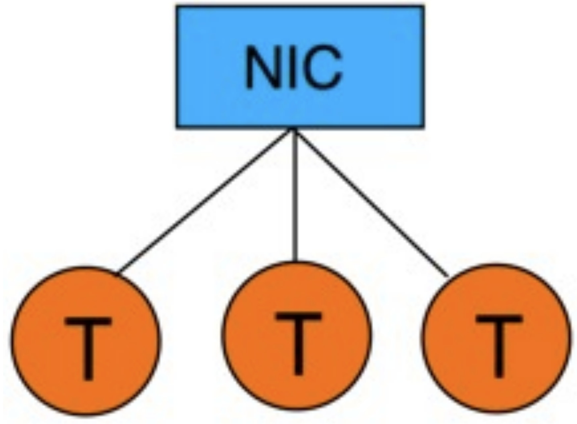
Threading Model



- **Single NIC driver thread**
  - Multiplexes requests among worker threads
- **Intelligent multiplexing**
- **IPC: Shared memory regions**

14

Threading Model



- Faster if we pass around the NIC?
- Needs locking around the NIC

Threading Model

- Single threaded
- Avoid dispatching/synchronization costs
- Lowest latency?

RPC API

- **Asynchronous API:**
  - Can have multiple outstanding RPCs
  - Can be used by master to communicate with backups
  - Can be used by client to perform multiple operations in parallel

```
rpc1.startRPC(backup1, payload);
rpc2.startRPC(backup2, payload);
rpc3.startRPC(backup3, payload);

// do_other_work()

Buffer *reply1 = rpc1.getReply();
Buffer *reply2 = rpc2.getReply();
Buffer *reply3 = rpc3.getReply();
```

- **Broadcast/multicast**
  - Needed for some parts of the system: recovery, etc
  - Support in RPC layer or on top of it?

17

Conclusion

- **Experimental fast RPCs: 11 µs**
  - Rip out unnecessary software layers
  - NIC Hardware: 1.95 µs x 4
- **Software overheads < 1 µs**
  - But in an impractical ways
- **Need help with NIC and switches**
- **Early RPC system design**

18

Discussion

- Is 5-10 µs achievable? Is it worthwhile?
- Threading model: event based vs worker threads
- Should we limit the size of an RPC?
- Is the asynchronous API the right way?
- Other requirements of the RPC system?