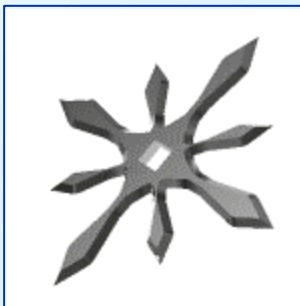


Scalable, Distributed Data Structures for Internet Service Construction

Steve Gribble, Eric Brewer, Joe Hellerstein,
and David Culler

gribble@cs.washington.edu



Ninja Research Group
(<http://ninja.cs.berkeley.edu>)
The University of California at Berkeley
Computer Science Division

Challenges: Simplicity and Scalability

"It's like preparing an aircraft carrier to go to war," said Schwab spokeswoman Tracey Gordon of the daily efforts to keep afloat a site that has already capsized eight times this year.

New York Times, June 20, 1999

In response to MailExcite' outages caused by a surge in traffic, one user wrote in a message, **"If MailExcite were a car we'd all be dead right now."**

c|net news, December 14, 1998

Motivation

- Building and running Internet services is very hard!
 - especially those that need to manage **persistent state**
 - their design involves many tradeoffs...
 - scalability, availability, consistency, simplicity/manageability
 - and there are very few adequate reusable pieces
- Goals of this work:
 - to design/build a reusable storage layer for services
 - to demonstrate properties of this layer quantitatively
 - all of the 'ilities, plus acceptable performance

Outline of Talk

- Motivation
- Introduction: Distributed Data Structures (DDS)
- Distributed hash table prototype
- Performance numbers
- Example services
- Wrapup

Context

- Clusters are natural platforms for Internet services
 - incremental scalability, natural parallelism, redundancy
 - but, state management is hard (must keep nodes consistent)
- But, no appropriate cluster state mgmt. tool exists
 - **(parallel) RDBMS?** expensive, overly powerful semantic guarantees, limited availability under faults
 - **distributed FS?** overly general abstractions, high overhead, often no fault tolerance/availability, ill-defined consistency
 - **roll your own?** \$\$\$, not reusable, complex to get right
 - optimal performance is possible this way...

An alternative storage layer for clusters

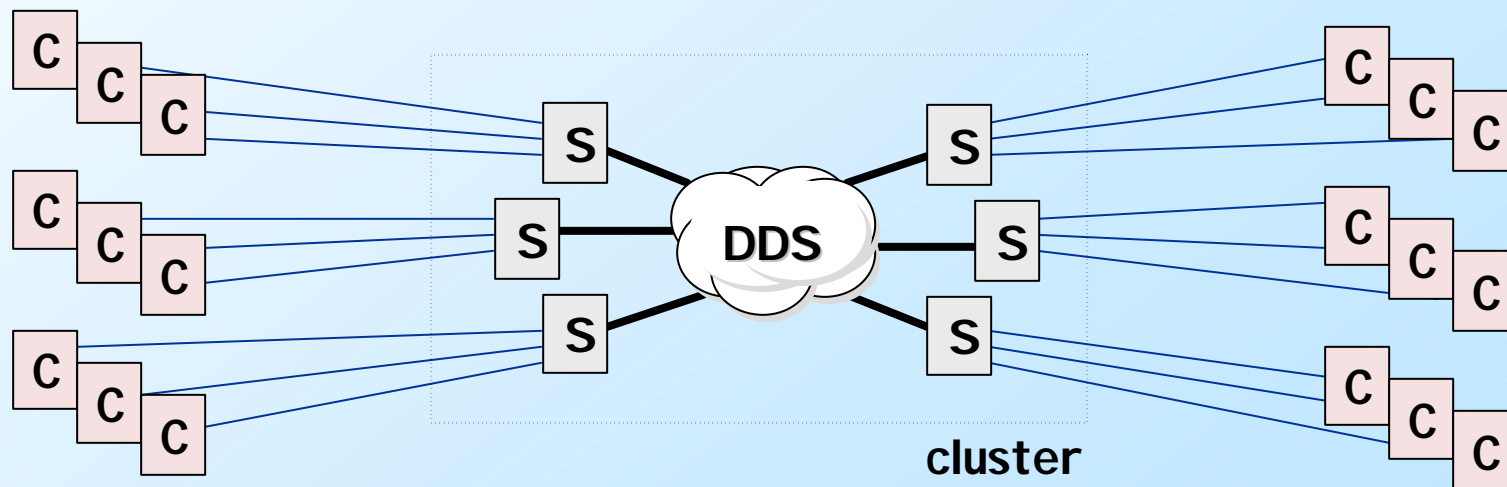
- **Distributed Data Structures (DDS)**

- start w/ hash table, tree, log, etc., and:

- partition it across nodes (parallel access, scalability, ...)
- replicate partitions in replica groups (availability)
- sync replicas to disk (durability)

- DDS maintains a consistent view across cluster

- atomic state changes (but not transactions)
- engenders a simple architectural model: any node can do any task



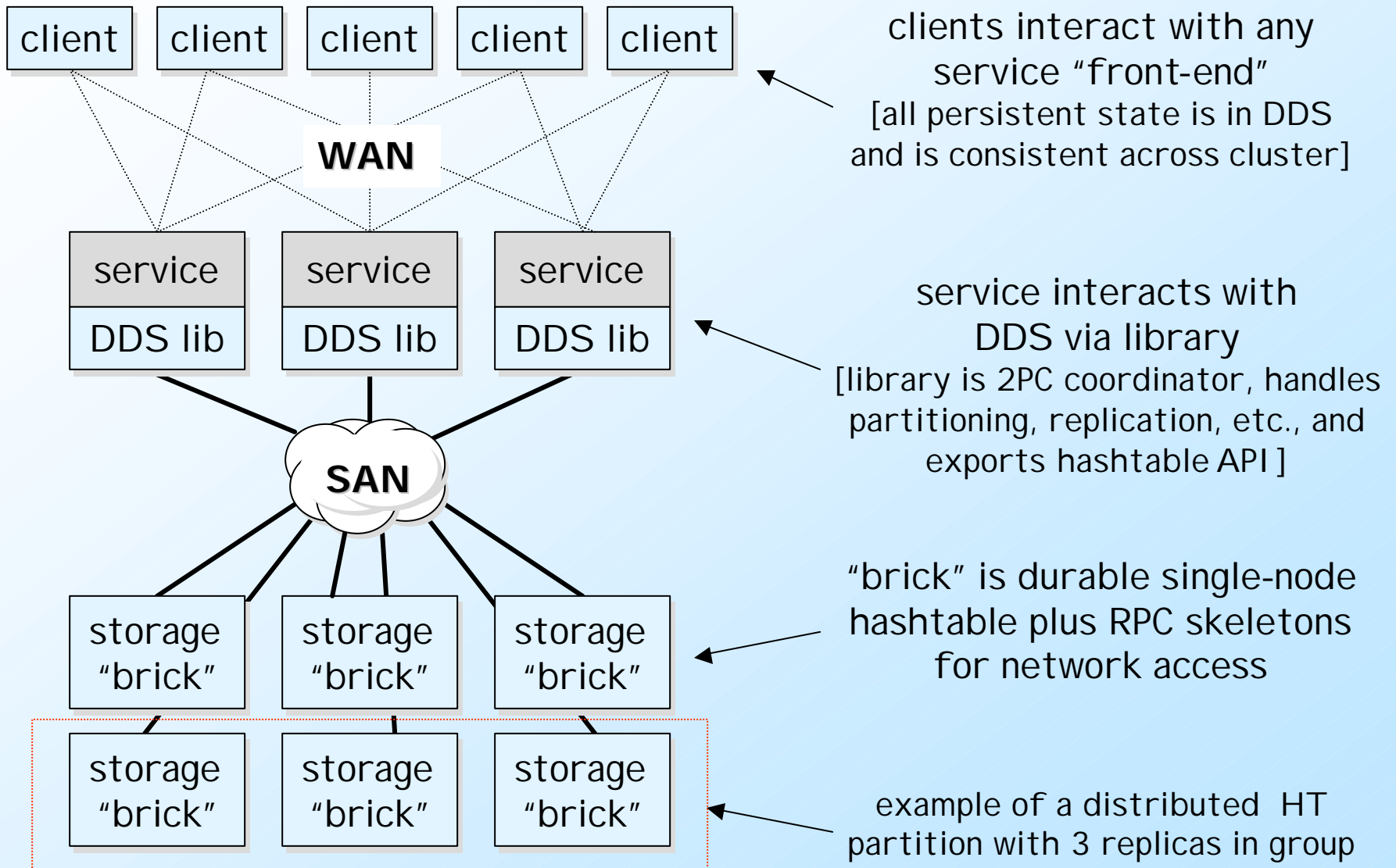
Guiding Principles

1. Simplification through separation of concerns
 - decouple persistence/consistency from rest of service
 - DDS abstraction: programmers understand data structures, so this is a natural extension
2. Appeal to properties of clusters to mitigate the hard distributed systems problems
 - “**cluster** ¹ **wide area**”: physically secure, well administered, redundant SAN, controlled heterogeneity
 - e.g. low-latency network → two-phase commit not prohibitive
 - e.g. redundant SAN → no network partitions → “presumed commit” optimistic two phase commits

Outline of Talk

- Motivation
- Introduction: Distributed Data Structures (DDS)
- Distributed hash table prototype (in Java)
- Performance numbers
- Example services
- Wrapup

Prototype DDS: distributed hash table



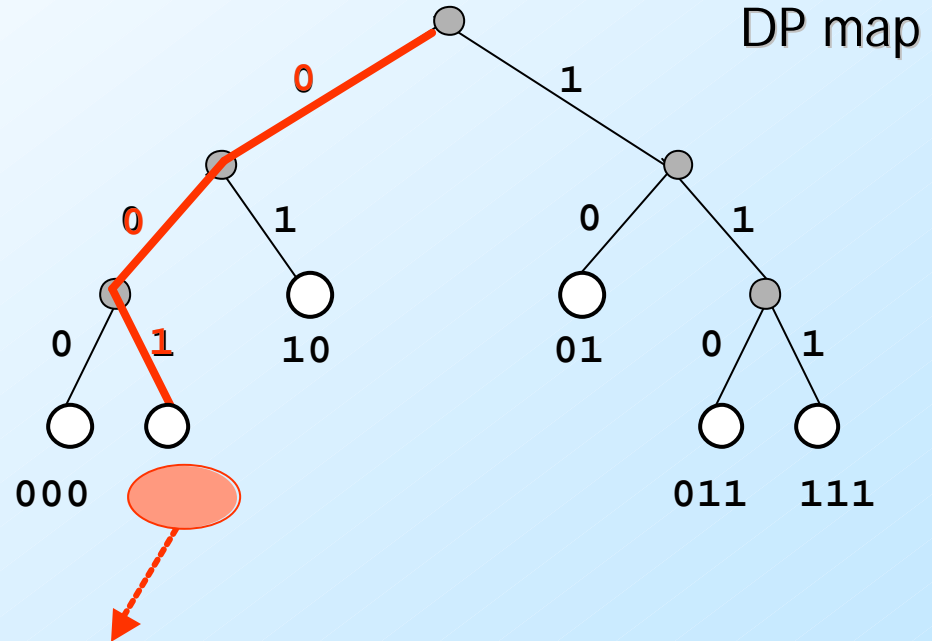
Distribution: cluster-wide metadata structures

- Two data structures are maintained across cluster:
 - ***data partitioning map (DPmap)***
 - given key, returns name of replica group that handles key
 - as the hash table grows in size, map subdivides
 - “subdivision” ensures localized changes (bounds # of groups affected)
 - ***replica group membership maps (RGmap)***
 - given replica group name, returns list of bricks in replica group
 - nodes can be dynamically added/removed from replica groups
 - node failure is subtraction from group
 - node recovery is addition to group
- the consistency of these maps is maintained, but lazily
 - clients piggyback operations w/ hash of their view of maps
 - if view is out of date, bricks send new map to client
 - maps are also broadcast periodically

Metadata maps: hash table put

key: **11010100**

1. lookup RG name in DP map trie
2. lookup RG members in RG map table
3. two-phase commit put to all RG members



RG name	RG membership list
000	dds1.cs, dds2.cs
100	dds3.cs, dds4.cs, dds5.cs
10	dds2.cs, dds3.cs, dds6.cs
011	dds7.cs

RG map

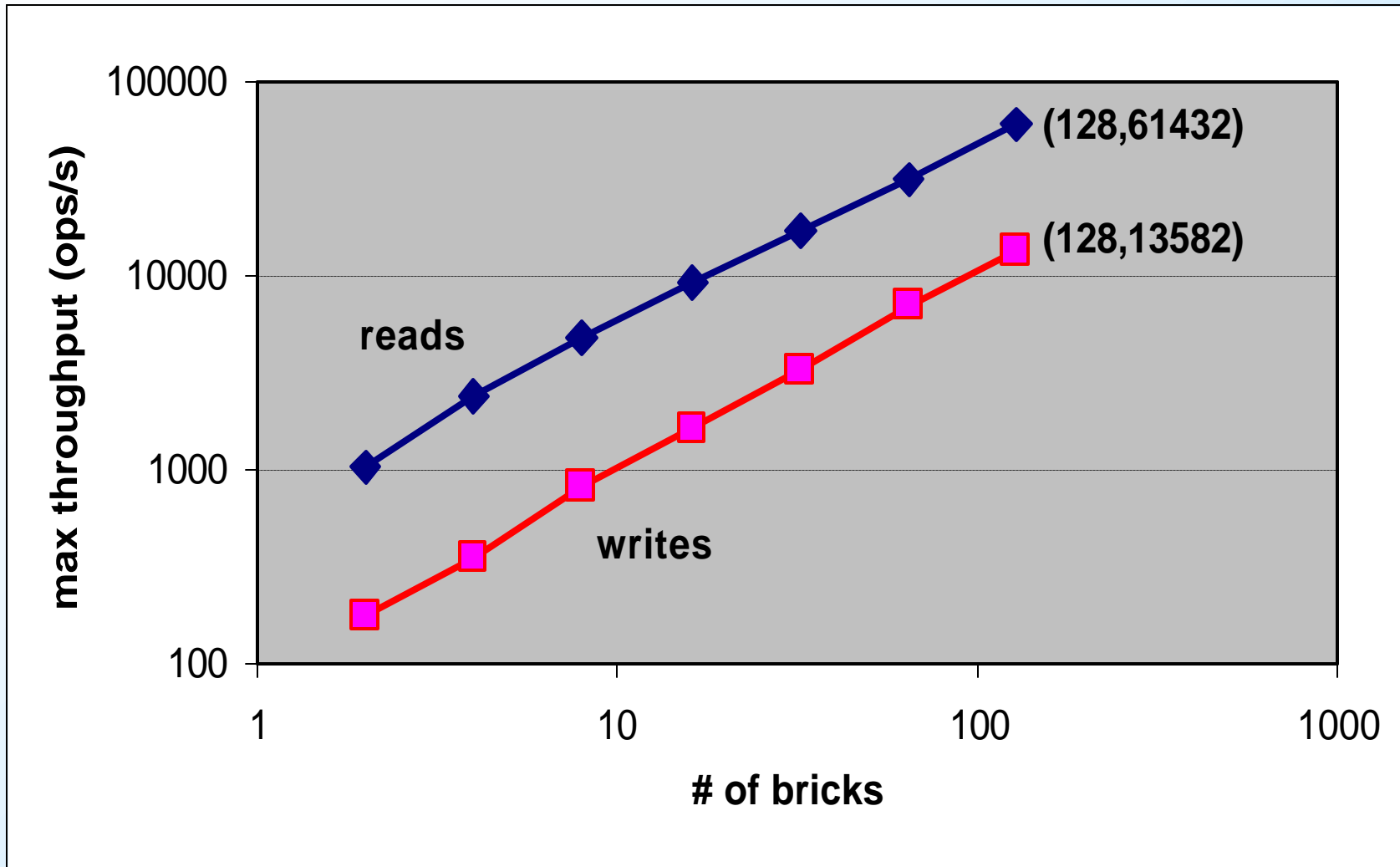
Recovery

- Insights:
 1. make hash table “best effort”- it’s ok to say no
 - e.g., if can’t get lock, replica group membership changes during operation., etc
 2. enforce invariants to simplify
 - no state changes unless client + all replicas agree on current maps
 3. make partitions small (10-100 MB), but have many
 - given fast SAN, copying an entire partition is fast (1-10 seconds)
 4. brick failures don’t happen often (once per week)
- Given these insights, brick failure recovery is easy:
 - grab write lock over one replica in a partition
 - copy the entire replica to the recovering node
 - propagate new RGmap to other nodes in replica group
 - release lock

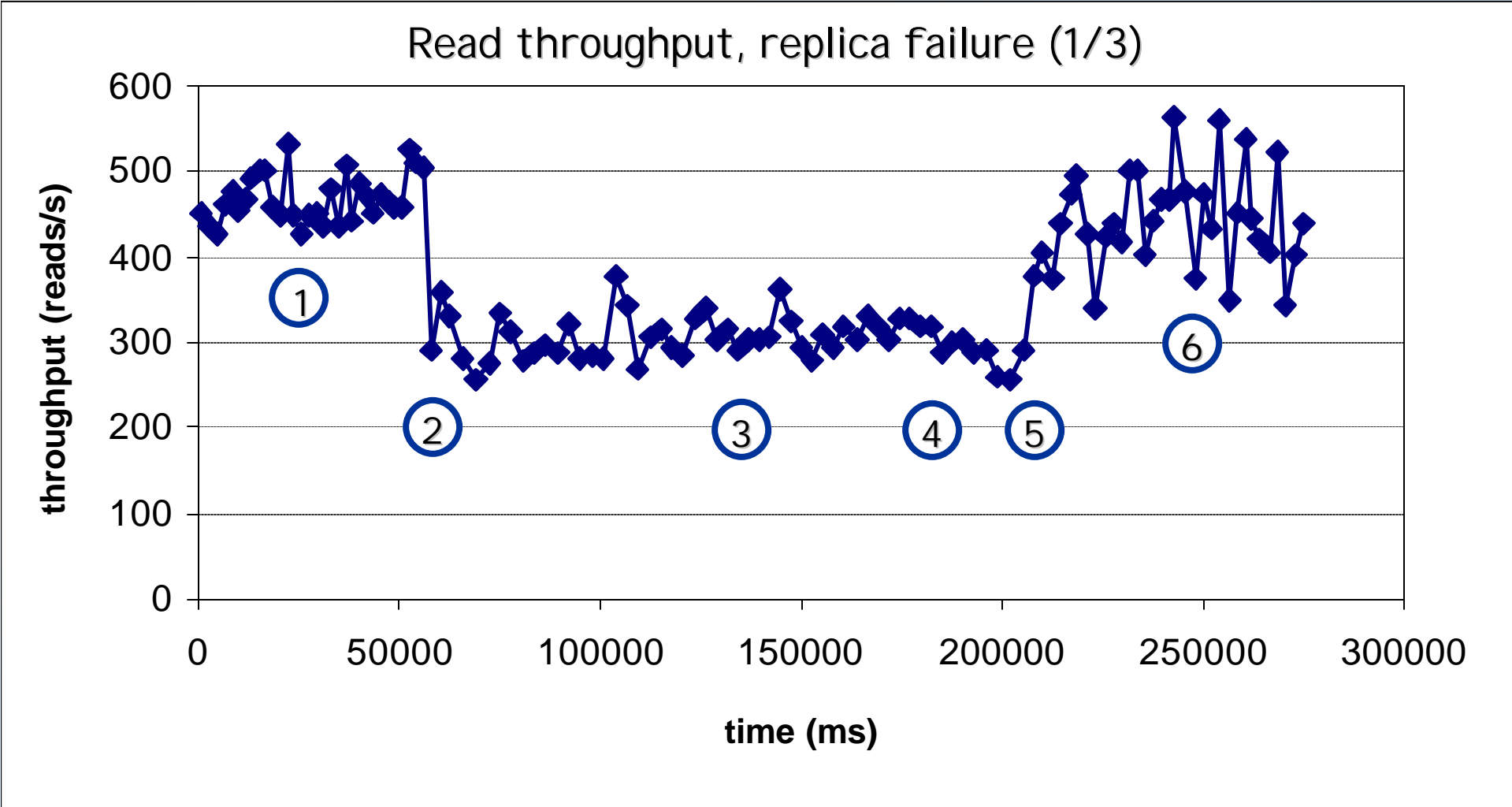
Outline of Talk

- Motivation
- Introduction: Distributed Data Structures (DDS)
- Distributed hash table prototype
- Performance numbers
- Example services
- Wrapup

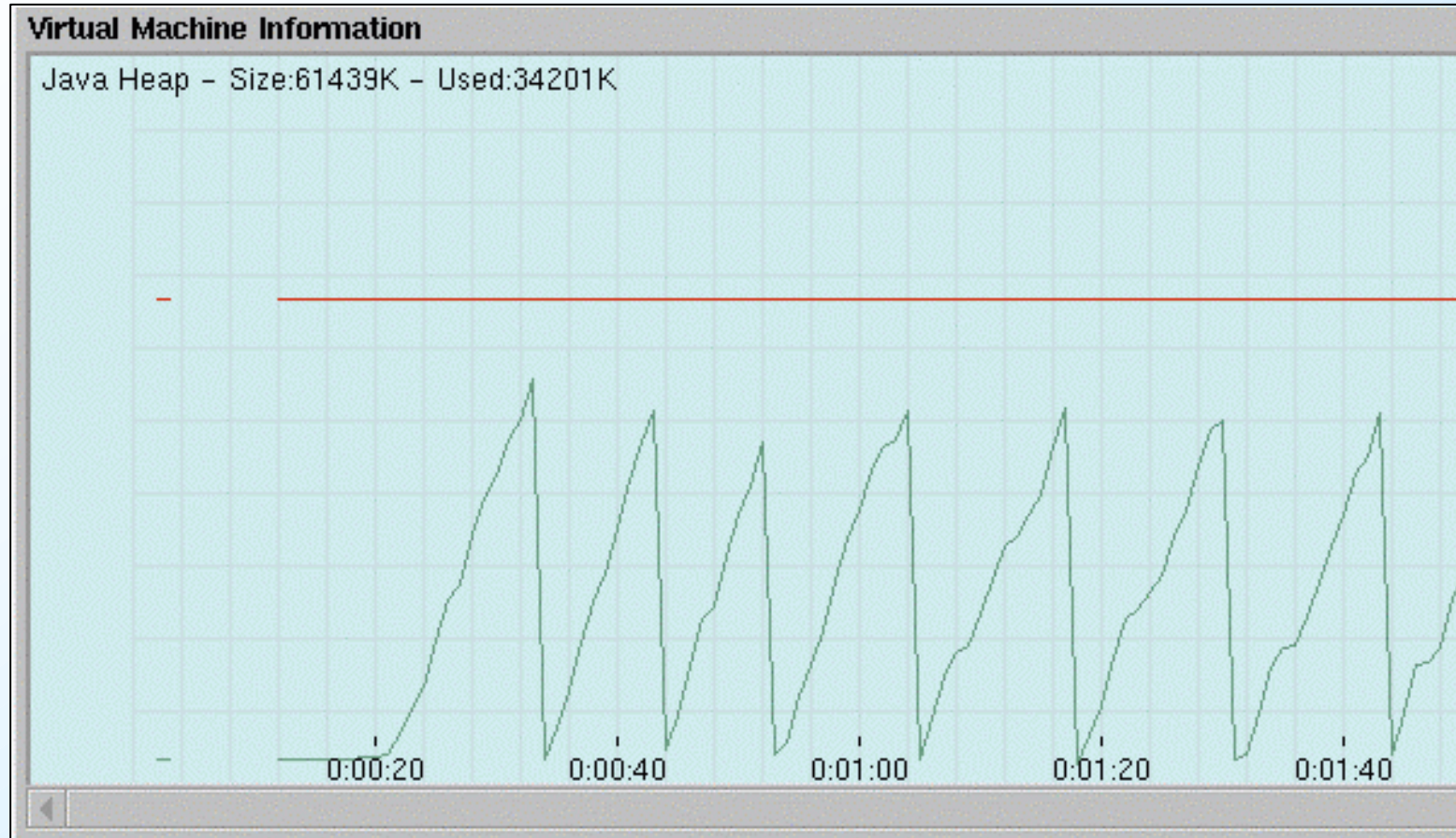
Scalability (reads and writes)



Recovery Behavior



Recovery Behavior



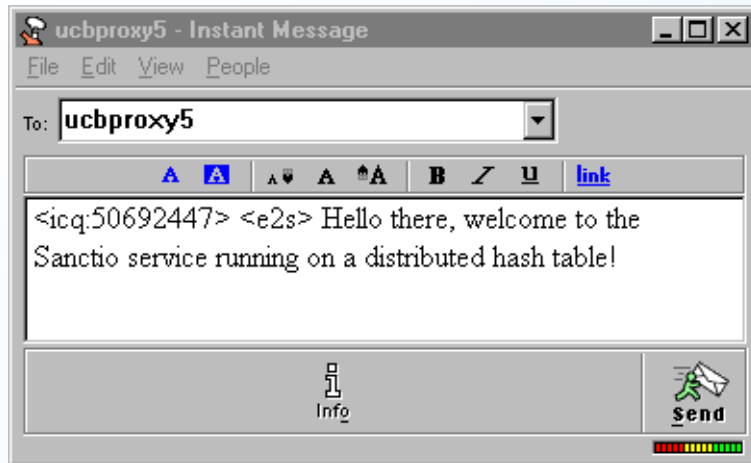
Capacity

- Scaled a single hash table to 1.2 TB
 - 128 brick nodes
 - 128 disks (10 GB per disk)
 - 512 partitions (1 replica per partition, i.e. no replication)
- Performance:
 - 200 8KB inserts per second per brick node
 - 1.5 hours to load the full terabyte table
 - is about 2 MB/s per disks
 - DP map + single-node hash function = seek-dominated traffic

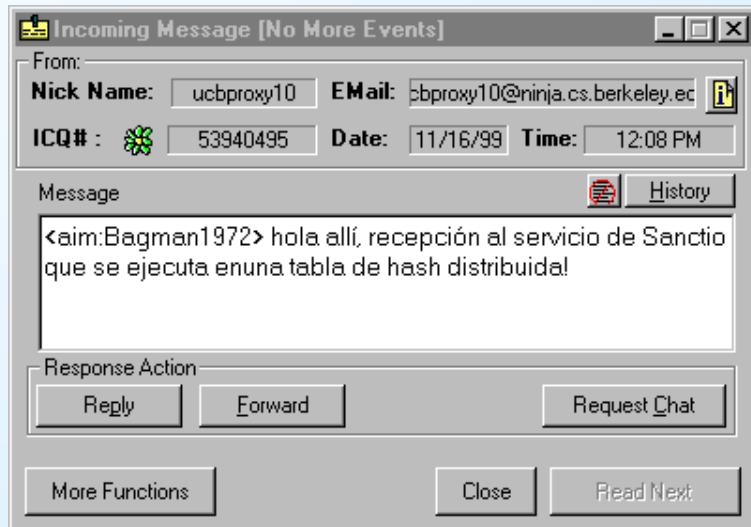
Outline of Talk

- Motivation
- Introduction: Distributed Data Structures (DDS)
- I/O layer design
- Distributed hash table prototype
- Performance numbers
- Example services
- Wrapup

Example service: "Sanctio"



AOL client



ICQ client

- instant messaging gateway
 - ICQ <-> AIM <-> email <-> voice
 - Babelfish language translation
- large routing and user pref. state maintained in service
 - each task needs two HT lookups
 - one for user pref, one to find correct "proxy" to send through
 - strong consistency required, write traffic is common (change routes)
- very rapid development
 - 1 person-month, most effort on IM protocols. State management: 1 day

Outline of Talk

- Motivation
- Introduction: Distributed Data Structures (DDS)
- I/O layer design
- Distributed hash table prototype
- Performance numbers
- Example services
- **Wrapup**

Wrapup

- Distributed data structures are a viable mechanism to simplify Internet service construction
 - they possess the 'ilities: scalability, availability, durability
 - they engender a simple and familiar programming model
- Some guiding principles of DDS design
 - exploit properties of clusters to simplify
 - two-phase commit optimizations, fault recovery design
 - make hash table "best effort"
 - saying 'no' simplifies recovery, implementation, etc.

For all the gory details, PhD thesis online at:

<http://www.cs.washington.edu/homes/gribble>

Backup Slides

I/O layer design decisions

- It turns out the interesting design choices are:
 - i. **APIs**: subtle changes in API lead to radical changes in usage
 - e.g.: always allow user to pass in a token to an async. enqueue that will be returned with the corresponding completion
 - e.g.: allow user to specify destination of completions on every enqueue
 - it took me 6 versions of library to get all this right...!
 - ii. mechanisms for passing **completions and chaining** queues/sinks
 - **polling** (polls fan down chains) vs. **upcalls** (completions run up queues)
 - polling seemed correct, but:
 - when do you poll? (always, maybe with some timing delay loops)
 - what do you poll? (everything, as can't know what is ready)
 - who does the polling? (everybody waiting for completions)
 - upcalls much more efficient: events generated exactly when data ready
 - "dream OS": async. everything, no app contexts but upcall handlers

Layering on top of basic HT

- Lightweight layering through FSMs is heavily exploited
 - “basic” distributed hash table layer
 - operations may suffer transient failures (locks, timeouts, etc)
 - maximum value size 8KB
 - “sugar” distributed hash table layer
 - bust up large HT values (>8KB), stripe across many smaller values
 - “reliable” distributed hash table layer
 - on transient failures, retry operation a few times
- Additional data structures can reuse layers
 - planned: tree, log, skiplist?
 - layer on top of existing 2PC, brick, I/O substrate
 - replace data partitioning map
 - less efficient: layer on “basic” or “sugar” distributed hash table
 - may negatively impact performance (e.g. could specialize lower layers for that particular data structure)

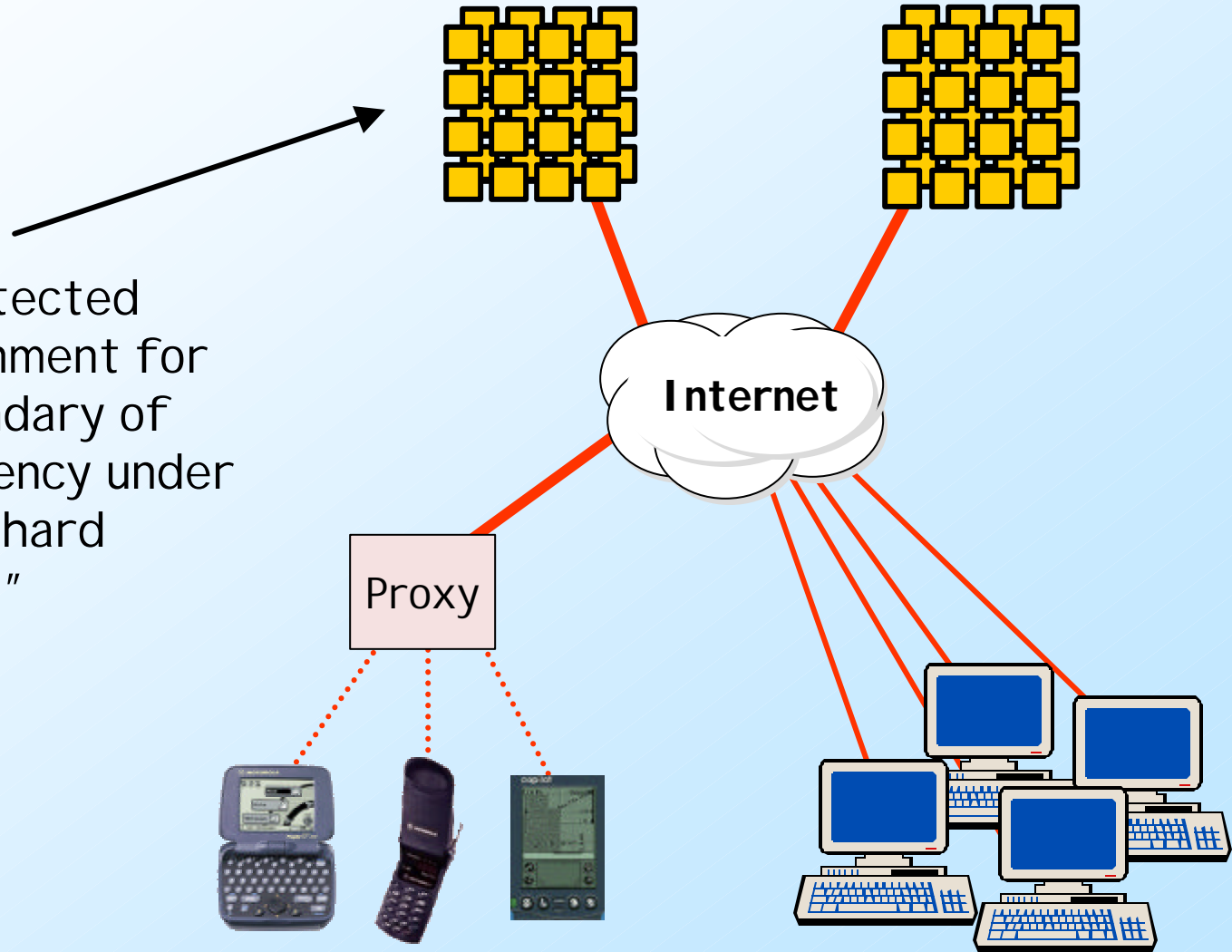
DDS vs RDBMS

- DDS uses RDBMS techniques...
 - buffer cache, lock manager, HT access method, two-phase commit, recovery path
- but with different goals, abstractions, and semantics
 - high availability **and** consistency
 - HT API is a simple declarative language
 - does give both data independence and implementation freedom
 - but is at lower semantic level: exposes intention of operations
 - current semantics: atomic single operations
 - but, "Telegraph" project at Berkeley: transactional system on top of same I/O layer API and implementation

DDS vs. distributed, persistent objects

- Current DDS's don't provide:
 - pointers between objects
 - especially those that exist outside of object infrastructure
 - (distributed objects: anonymous references are possible)
 - the need to GC is especially hard in this case
 - extensibility
 - intention of access is not as readily apparent as DDS
 - with objects, ability to create any DS out of them
 - type enforcement
 - extra metadata and constraints to enforce at access time

The Big Picture



Base: well-protected cluster environment for services. Boundary of strong consistency under faults. "Solve hard problems here."

Java- what worked

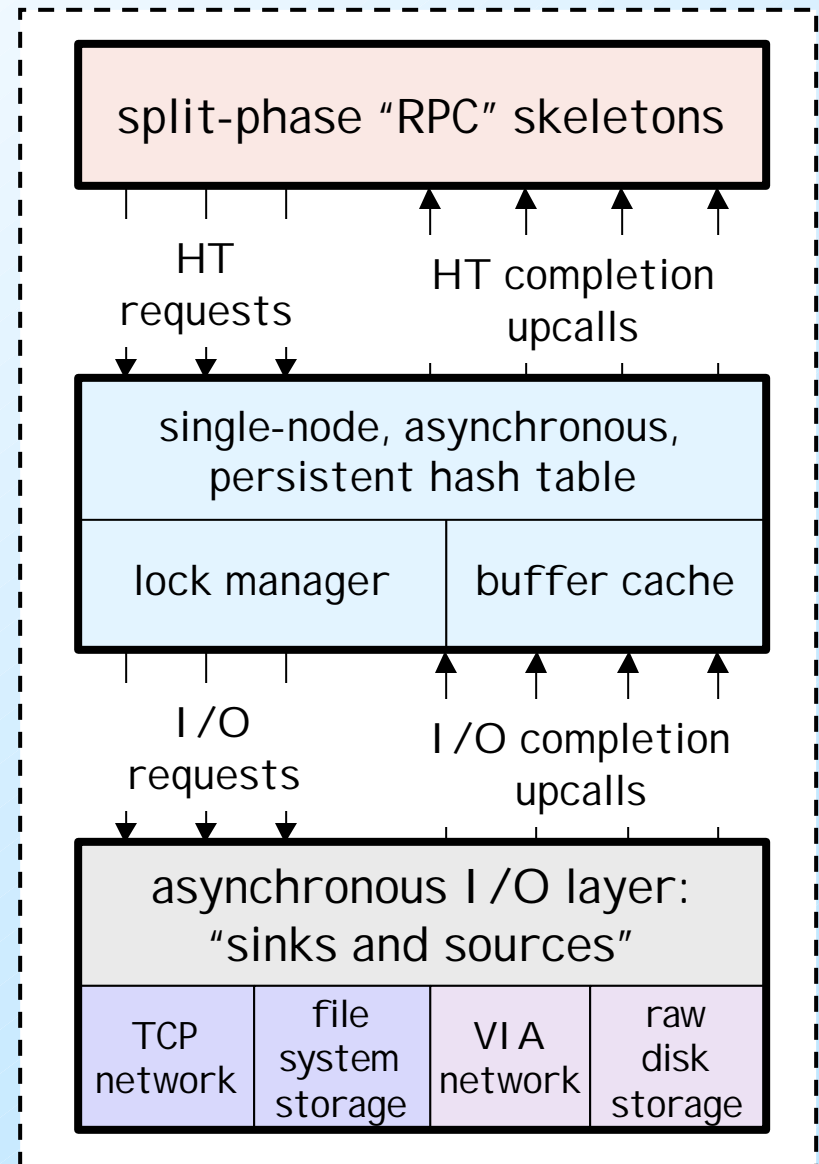
- strong typing, rich class library, no pointers
 - made software engineering much, much simpler
 - conservatively estimate 3x speedup in implementation time
- subclassing, declared interfaces
 - much, much cleaner I/O core API as a result
- portability
 - it was possible to pick up DDS and run it on:
 - NT, linux, solaris
 - but, of course, each JDK had its own peculiarities

Java- what didn't work

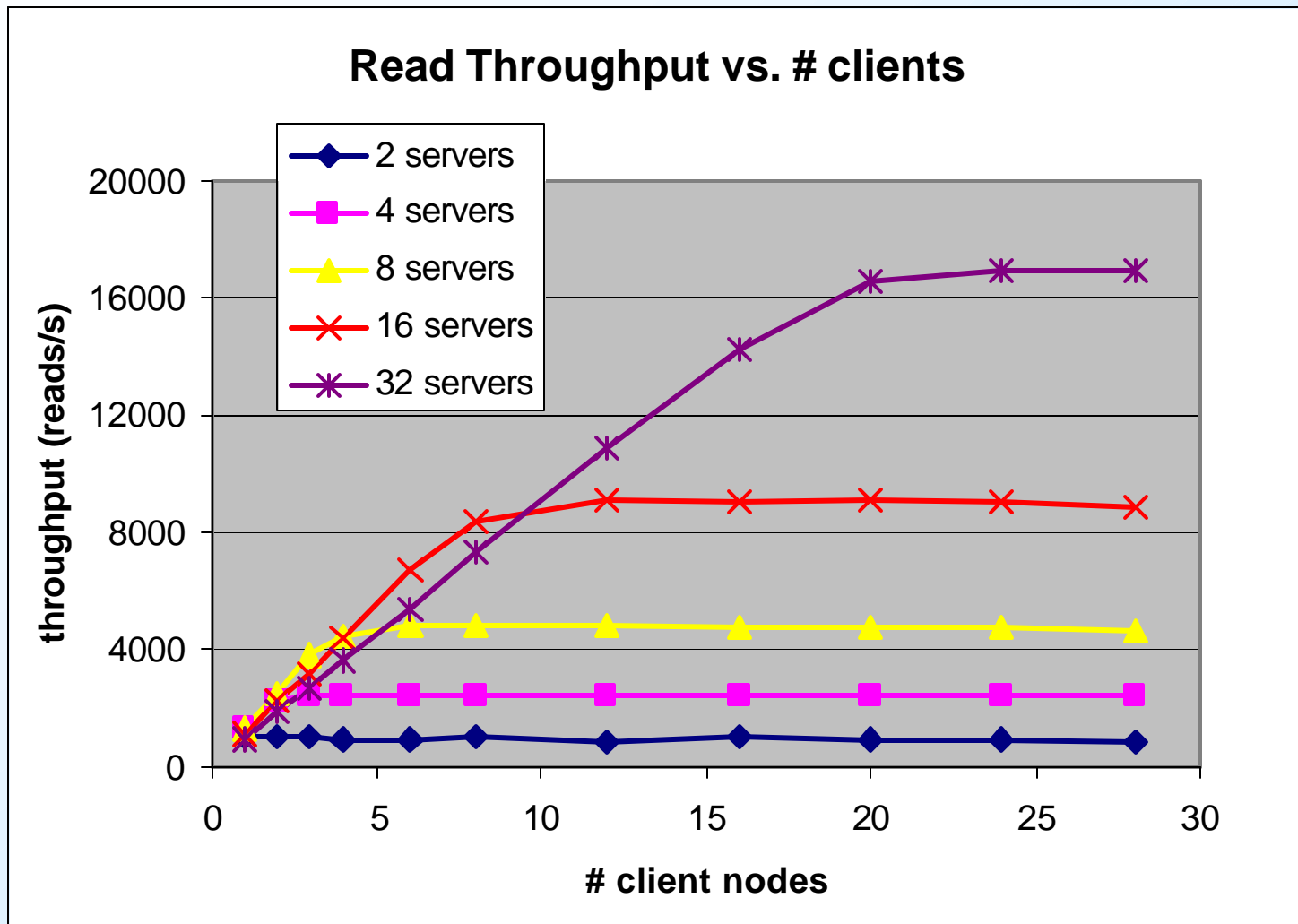
- garbage collection
 - performance bottleneck if uncontrolled
 - Jaguar- bottleneck factor over 100 Mb/s network
 - induced metastable equilibrium
- strong typing and no pointers
 - forced many byte array copies
- lack of appropriate I/O abstractions
 - everything is thread-centric
 - no non-blocking APIs
- Java + linux = pain
 - linux kernel threads: very heavyweight contended locks
 - linux JDK's are behind the curve

"Brick" implementation

- single node hash table
 - RPC skeletons slapped on for remote hash table access
- composed of many layers
 - each layer consists of state machines + chained upcalls
 - layers themselves are asynchronous
 - e.g. buffer cache, lock mgr
- implementation:
 - chained hash table, static # of buckets specified at creation
 - key = 64 bit number
 - value = array of bytes

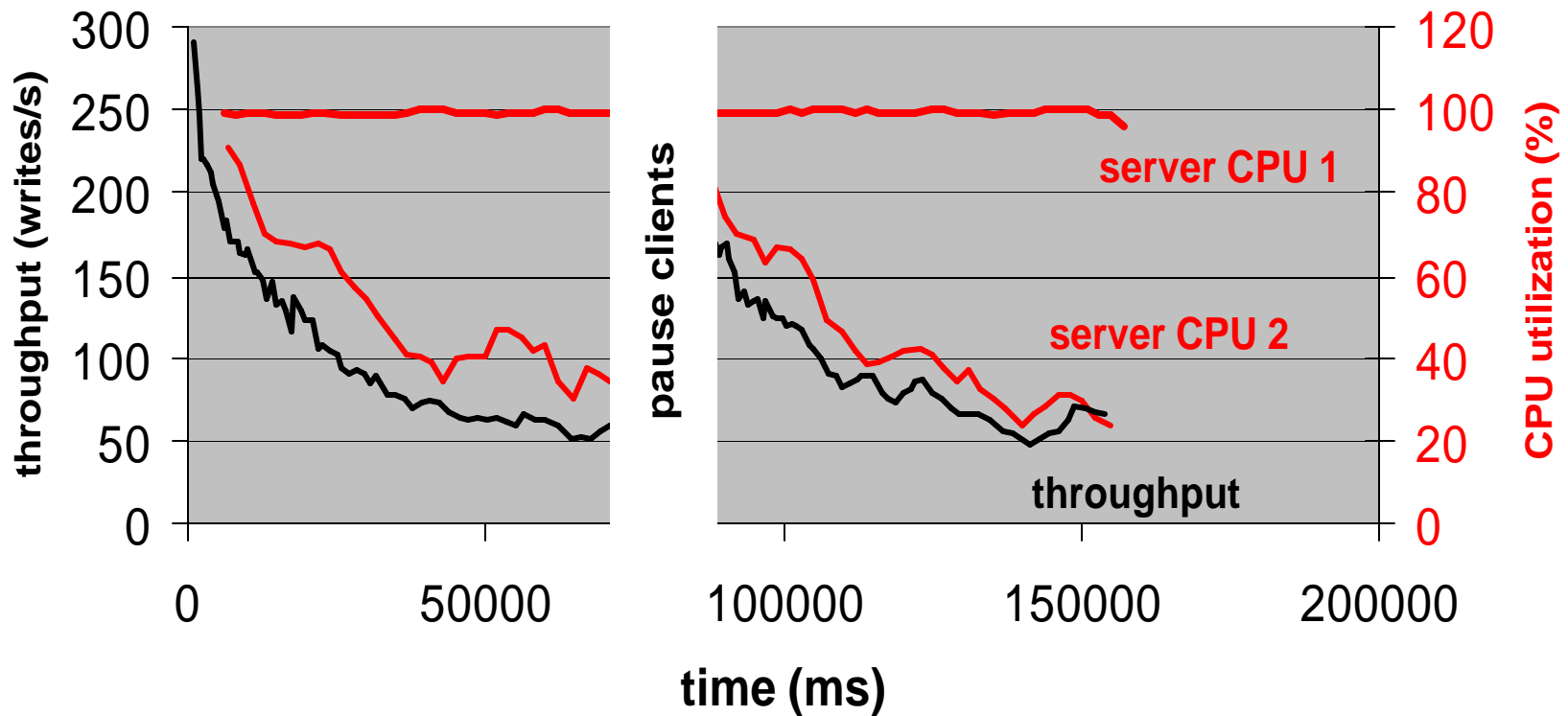


Graceful Degradation (Reads)



But...an unexpected imbalance on writes

Write Throughput vs. servers' CPU utilization



Garbage Collection Considered Harmful...

- What if...

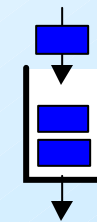
service rate $S \propto (\text{queue length } Q)^{-1}$

- then, there is a Q_{thresh} where

$$Q > Q_{\text{thresh}} \Rightarrow R > S$$

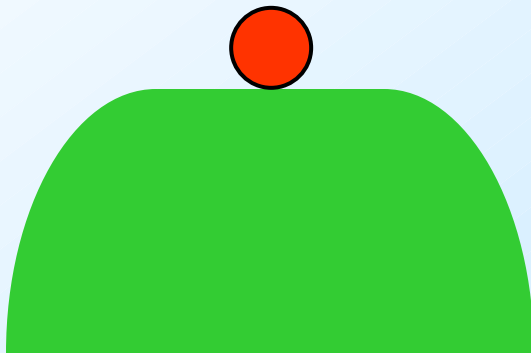
- Unfortunately, garbage collection tickles this case..
 - more objects means more time spent on GC

arrival rate R



queue length Q

service rate S



- Physical analogy: ball on a windy flat-topped hill
 - classic unstable equilibrium
 - need "anti-gravity" force, or need windshield
 - admission control, flow control, discard, ...
- Feedback effect: replica group runs at speed of slowest node (for inserts)

More Example Services

- Scalable web server
 - “service” is HTTPD, fetches content from DDS
 - uses lightweight FSM-layering for CGI s
 - 900 lines of Java, 750 for HTTP parsing etc., <50 for DDS
- “Parallelisms” what’s related server
 - inversion of Yahoo!
 - given a URL, identifies what Yahoo categories it is in
 - returns other URLs in those categories
 - 400 lines, 130 for app-specific logic (rest is HTTP junk)
- Many services in the “Ninja” platform
 - user preference repository, user key repository, collaborative filtering engine for a communal jukebox, ...

Guiding Principles

3. “Internet service” means huge # of parallel tasks
 - optimize system to **maximize task throughput**
 - minimizing task latency is secondary, if needed at all
 - thread per task breaks!
 - focus changes from “pushing a task” to maintaining flows
 - need asynchronous I/O and event-driven model

4. A **layered implementation** with much reuse is possible
 - I/O subsystem and an event framework
 - RPC-accessible storage “bricks”
 - + two-phase commit code, recovery code, locking code, etc.
 - data structures are built on top of these reusable pieces

Throughput vs. Read Size

