Taming the Internet Service Construction Beast

Persistent, Cluster-based Distributed Data Structures

(in Java!)

Steven D. Gribble

gribble@cs.washington.edu

Challenges: Consistency and Availability

Despite a frantic, around-the-clock effort to keep the auction site running after **two embarrassing and costly outages** this week, eBay was again inaccessible to customers this morning. A **corrupted database** was blamed for the disruption.

c|net news, June 11, 1999

Challenges: Manageability 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

MailExcite has been suffering from outages for the past week, as a result of scalability problems 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

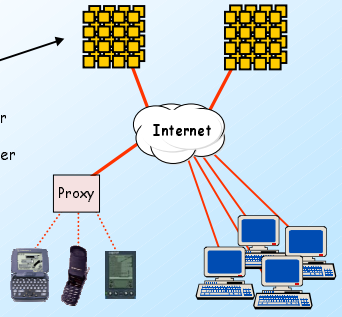
- **Building and running Internet services is very hard!**
  - especially those that need to manage **persistent state**
  - their design involves many tradeoffs…
    - scalability, availability, consistency, simplicity/manageability
  - and there are very few adequate reusable pieces
- **Goals of this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

Outline of Talk

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

The Big Picture

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

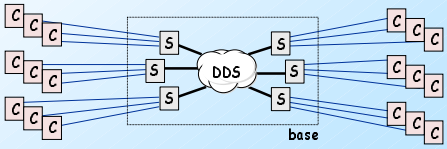


Context

- Clusters are natural platforms for Internet services
  - incremental scalability, natural parallelism, redundancy
  - but, state management is hard (must keep nodes consistent)
- No appropriate persistent cluster state mgmt. tool exists
  - **use (parallel) RDBMS?** expensive, overly powerful semantic guarantees, generality at cost of performance (SQL), limited availability under faults
  - **use distributed FS?** overly general abstractions, high overhead, often no fault tolerance or availability.
    - FS read/write operations hide intent
  - **roll your own?** $$$, not reusable, complex to get right
    - optimal performance is possible this way…

An alternative storage layer for Bases

- 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, persistence)
    - 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



For example…

service	core data structures required
web server	read-mostly hash table or tree *(documents)* log *(hit logging)*
search engine	hash tables *(search $$, word->doc->metadata maps)* write-mostly logs *(hit logs, crawler spool file)* tree *(optional: date index over documents)*
PIM service	hash tables *(users' PIM data)* write-mostly logs *(hit logs)* trees *(date indexes over appointments, emails, etc.)*

Observations and Principles

- Simplification through separation of concerns
  - decouple persistence/consistency from rest of service
  - DDS abstraction: programmers understand data structures, so this is a natural extension
- Appeal to properties of clusters to mitigate the hard distributed systems problems
  - "**cluster ≠ 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
    - e.g. physically secure + firewall → cluster-wide TCB → no authentication for access to DDS inside cluster

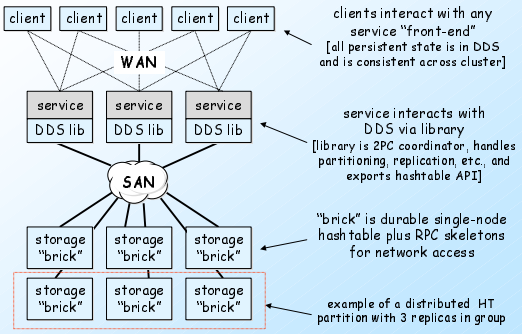
Observations and Principles

- "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

Outline of Talk

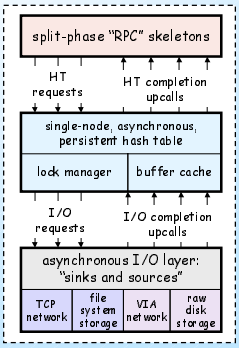
- Motivation
- Introduction: Distributed Data Structures (DDS)
- Distributed hash table prototype
- Performance numbers
- Example services
- Wrapup

Prototype DDS: distributed hash table



"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
    - 8KB maximum value size



Distribution: cluster-wide metadata structures

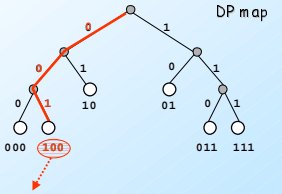
- Two data structures are maintained across cluster:
  - ***data partitioning map*** *(DPmap)*
    - given key, returns name of replica group that handles key
    - as the hash table grows in size, map subdivides
    - "subdivision" ensures localized changes (bounds # of groups affected)
  - ***replica 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

- lookup RG name in DP map trie
2. lookup RG members in RG map table
- two-phase commit put to all RG members, key is remaining bits:

11010



RG map

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

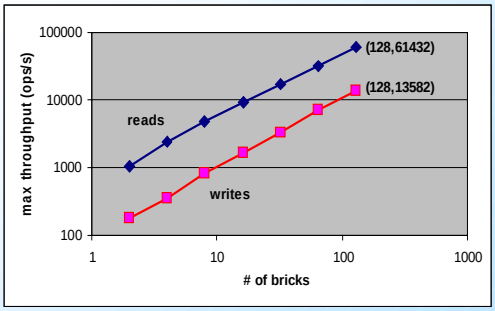
Recovery

- Insights:
  - make hash table "best effort"
    - OK to return failure (if can't get lock, replica group membership changes during op., etc)
    - rely on higher layer or application to retry
  - enforce invariants to simplify
    - no state changes unless client + all replicas agree on current maps
  - make partitions small (10-100 MB), but have many
    - given fast SAN, copying an entire partition is fast (1-10 seconds)
  - 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

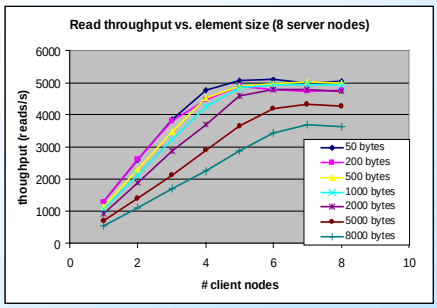
Outline of Talk

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

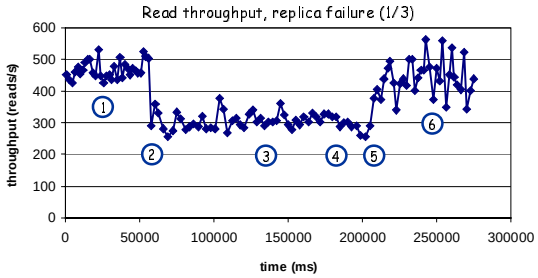
Scalability (reads and writes)



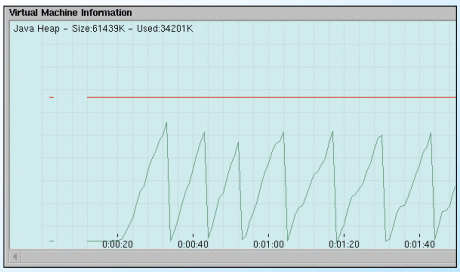
Throughput vs. Read Size



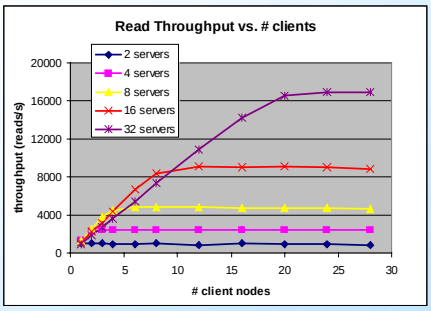
Recovery Behavior



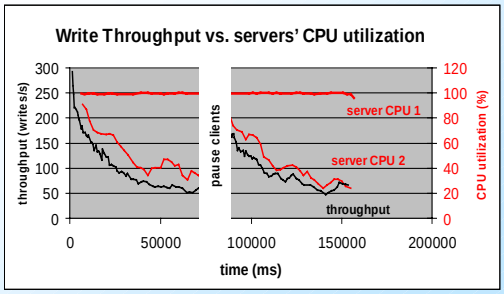
Recovery Behavior



Graceful Degradation on Reads

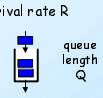


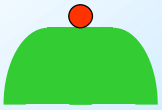
But…an unexpected imbalance on writes



Garbage Collection Considered Harmful…

- What if…
  `service rate S ∝ (queue length Q)^-1`
- then, there is a `Qthresh` where
  `Q > Qthresh ⇒ R > S`
- Unfortunately, garbage collection tickles this case..
  - more objects means more time spent on GC





- 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)

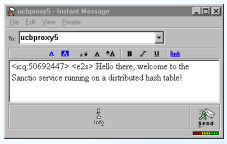
Capacity

- Scaled to 1 TB
  - 128 brick nodes
  - 256 disks (10 GB per disk)
  - 512 partitions
  - 1024 replicas
- Performance:
  - 200 8KB inserts per second per brick node
  - 1.5 hours to do insert
  - (is about 2 MB/s per disk…seeks hurt!)

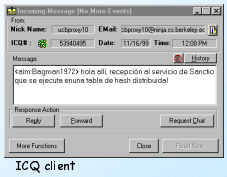
Outline of Talk

- Motivation
- Introduction: Distributed Data Structures (DDS)
- 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

http://sanctio.cs.berkeley.edu
(http://sanctio.cs is running on a DDS too!)

More Example Services

- Scalable web server
  - "service" is HTTPD, fetches content from DDS
  - uses lightweight FSM-layering for CGIs
  - 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, …

Outline of Talk

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

Wrapup

- Distributed data structures are a viable mechanism to simplify Internet service construction
  - they possess all of the 'ilities: scalability, availability, durability
  - they engender a simple and familiar programming model
- Implementing a DDS requires an effective I/O substrate
  - use asynchronous I/O to handle the extreme concurrency
  - FSMs and event-driven programming fall out of this model
    - allows light-weight composition of layers
- Properties of clusters can be exploited in DDS design
  - two-phase commit optimizations, fault recovery design
- Some principles of DDS design:
  - "best effort" hash table simplifies recovery, implementation, etc.
  - additional properties gained by exploiting layering