

## CS290i - Lecture 16 Cluster Data Struct

Scalable Internet Services and Systems, Spring 2001

Thorsten von Eicken  
Department of Computer Science  
University of California at Santa Barbara

## Distributed Data Struct.

### † Scalable, Distributed Data Structures for Internet Service Construction

† S. Gribble, E. Brewer, J. Hellerstein, D. Culler, @UCB

### † Background

- † Internet services need clusters
  - † To handle the load
- † Service throughput is high, but service latency is high too
  - † Many requests handled simultaneously
  - † All largely independent of one another

## DDS

### † Idea

- † Self-managing storage layer
- † Designed to run on a cluster
- † DDS holds persistent application state
- † Application code (*service instances*) are stateless, or has soft-state that can be reconstructed from DDS

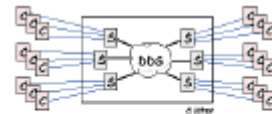


Figure 1: High-level view of a DDS: a DDS is a self-managing, cluster-based data repository. All service instances (SI) in the cluster use the same consistent language of the DDS as a model. Any "REAL" client (C) can communicate with any service instance.

### † Overall structure

- † Internet/web clients send requests to cluster
- † Use favorite load distribution scheme to pick service instance
- † Service instances use DDS to access consistent state independent of location

## So, what is it?

### † It's not a database (RDBMS)

- † RDBMSs focus on durability & consistency (ACID properties)
  - † Always choose consistency over availability
- † RDBMSs decouple logical structure of data from physical
  - † Powerful SQL query language
  - † Hard to parallelize

### † It's not a distributed file system

- † DFSs have weak consistency guarantees
  - † Or coarse-grained locking (at file level)
- † DFSs impose structure of directory, file, and bytes

### † DDS: Fits in between

- † Strict consistency model: all operations are atomic
- † Operations are on entire data struct elements, not byte ranges
- † Small fixed set of operations, no general "queries"

## Design principles

- † **Separation of concerns**
  - † Separate service code from storage management: simplifies system architecture
  - † Encapsulate failure handling, restart, etc. in storage layer
    - ‡ Can interesting services really be stateless?
- † **Appeal to properties of clusters**
  - † Low-latency (<1ms), high bandwidth network (>100Mbps)
    - ‡ Can use two-phase commit
  - † No partitions
    - ‡ Highly interconnected, respect. single external connection
- † **High throughput, high concurrency**
  - † Use asynchronous, event-driven control flow
    - ‡ Not threads
    - ‡ Graceful degradation by accumulating work in even queues

## Distributed Hash Table

- † **Client**
  - † Unaware of DDS
  - † Connects to service
- † **Service**
  - † Set of cooperating service instances
  - † Rely on DDS to manage persistent state
- † **Hash Table API**
  - † Put(), get(), remove(), create(), destroy()
  - † Keys are 64-bit integers
  - † Values are opaque byte arrays
- † **DDS library**
  - † Interface to storage bricks
  - † 2-phase commit coordinator
- † **Brick**
  - † Implements persistent chained hash table
  - † Buffer cache, lock manager, etc.

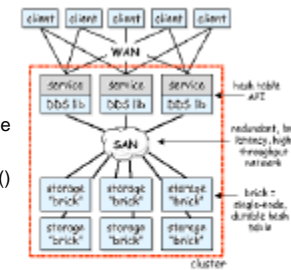


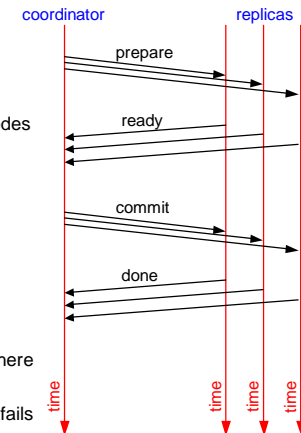
Figure 2: Distributed hash table architectures: each box in the diagram represents a software process. In the simplest case, each process runs on its own physical machine, however there is nothing preventing processes from sharing machines.

## Assumptions

- † **No partitions**
- † **Fail-stop software**
  - † Terminate if any unexpected condition occurs
- † **Independent software failures**
  - † Not true in the case of bugs
- † **Hash table workload**
  - † Population density of 64-bit keys is even
  - † Working set of "hot keys" larger than number of nodes in cluster
  - † Tables are large and long-lived

## Partitioning & replication

- † **Strategy**
  - † Slice hash table into partitions
  - † Distribute partitions among nodes
    - ‡ Parallelism
  - † Replicate each partition to several nodes
    - ‡ Replica groups
    - ‡ Failure tolerance
- † **Operations**
  - † Read any replica
  - † Write (put/remove) all replicas
    - ‡ Use optimistic 2-phase commit
- † **2-phase commit**
  - † Phase 1: reach common decision
  - † Phase 2: implement decision everywhere
  - † Timeouts to abort in case of failure
  - † Replicas communicate if coordinator fails
    - ‡ All commit if one receives "commit"



# Metadata Maps

- † **Data Partitioning (DP) map**
  - † Maps key to partition
  - † # of partitions related to size of cluster, not size of hash table
  - † Implemented as trie
    - † Minimal unambiguous prefix
- † **Replica Group (RG) map**
  - † Maps partitions to bricks
  - † Keeps track of active/failed bricks
- † **Maps replicated on all nodes**
  - † Library maps lazily updated
    - † Send hashes to verify consistency

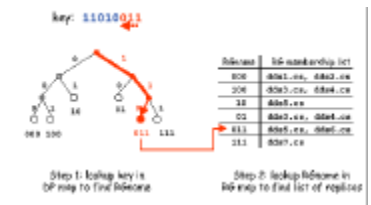


Figure 3: Distributed hash table metadata maps: this illustration highlights the steps taken to discover the set of replica groups which serve as the backing store for a specific hash table key. The key is used to traverse the DP map trie and retrieve the name of the key's replica group. The replica group name is then used looked up in the RG map to find the group's current membership.

# Recovery

- † **Simplifications**
  - † Bricks can "say no" and fail operation
    - † Simplifies handling of odd cases
    - † Library, service, and/or client responsible for retrying
  - † Fail operations with inconsistent metadata maps
- † **Recover by copying partitions**
  - † Keep partitions small (~100MB)
  - † Recover node by:
    - † Locking partition (write lease on all replicas)
    - † Copying all data
    - † Update RG map
    - † Unlocking partitions
  - † Thus can't put/remove while recovering a partition
  - † Use similar algorithm to change DP map

# Performance

- † **Set-up**
  - † 28 2-way SMP nodes
    - † 500Mhz P-II
    - † 512MB
    - † 100Mbps
  - † 38 4-way SMP nodes
    - † 500 Mhz P-II
    - † 1GB
    - † 1Gbps
  - † JDK 1.17, Linux 2.2.5
- † **Cluster roles**
  - † 128 service nodes
  - † 80 load generation nodes
- † **In-Core benchmarks**

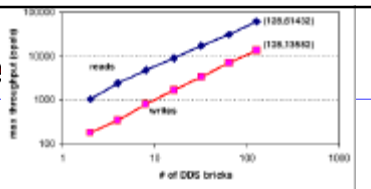


Figure 4: Throughput scalability: this benchmark shows the linear scaling of throughput as a function of the number of bricks serving in a distributed hash table: note that both axes have logarithmic scales. As we added more bricks to the DDS, we increased the number of clients using the DDS until throughput saturated.

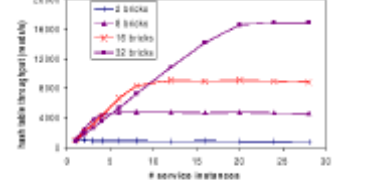


Figure 5: Graceful degradation of reader: this graph demonstrates that the read throughput from a distributed hash table remains constant even if the offered load exceeds the capacity of the hash table.

# Problem #1

- † **Ungraceful degradation of writes**
  - † Due to garbage collection
    - † Random fluctuation causes one node to slow down
    - † Prepare phase causes operations to queue up for at least on network round-trip
    - † More objects pile up in heap in the queues
    - † More GC needed to keep minimum free space
  - † Thrashing unavoidable in systems with finite resources
  - † Maybe admission control or early discard can alleviate problems

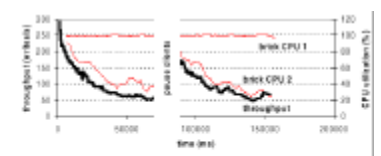


Figure 6: Write imbalance leading to ungraceful degradation: the bottom curve shows the throughput of a two-brick partition under overload, and the top two curves show the CPU utilization of those bricks. One brick is saturated, the other becomes only 30% busy.

## Recovery

### † Out-of-core benchmarks

- † 2 12GB disks per node
- † Reads limited by read disk bandwidth
- † Writes limited by read-write disk bandwidth
- † Random accesses limited by disk seek speed

### † Recovery

1. All bricks ok: 450 reads/sec
2. 1 of 3 bricks failed, 2/3 ops/sec
3. Start recovery
4. End recovery
5. New brick has cold buffer cache
6. Recovered: 450 reads/sec
  - † Oscillations due to GC

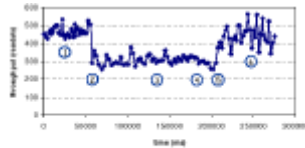


Figure 8: Availability and Recovery: this benchmark shows the read throughput of a 3-brick hash table as a challenge single-node fault is induced, and afterwards as recovery is performed.

13

## Discussion

### † Sample Services

- † Instant messaging gateway
  - † ICQ, AIM, SMS, email, ...
  - † Hash table maps users to addresses
- † Web server
  - † hashes URLs to 64-bit keys
  - † Stores documents in hash table elements

### † Failure problems

- † independence assumption not true in case of bugs

### † Scaling problems

- † 100Mbps switches became bottleneck
- † Network signals dispatch to user-level threads lib

### † Java problems

- † GC becomes bottleneck & source of performance variations
- † Type safety & array bounds checks prevent efficient buffer handling
  - † E.g. prepending headers
- † Lack of asynchronous I/O
  - † Required the use of threads

14