Distributed Hashing and Rehashing

BRAC UNIVERSITY
Inspiring Excellence

What is Hashing?

Hashing is the process of mapping one piece of data — typically an arbitrary size object to another piece of data of fixed size, typically an integer, known as hash code or simply hash. A function is usually used for mapping objects to hash code known as a hash function.

For example, a hash function can be used to map random size strings to some fixed number between 0 … N. Given any string it will always try to map it to any integer between 0 to N.

Suppose N is 100. Then for example, for any string hash function will always return a value between 0 to 100.

```
Hello          --->   60
Hello World    --->   40
```

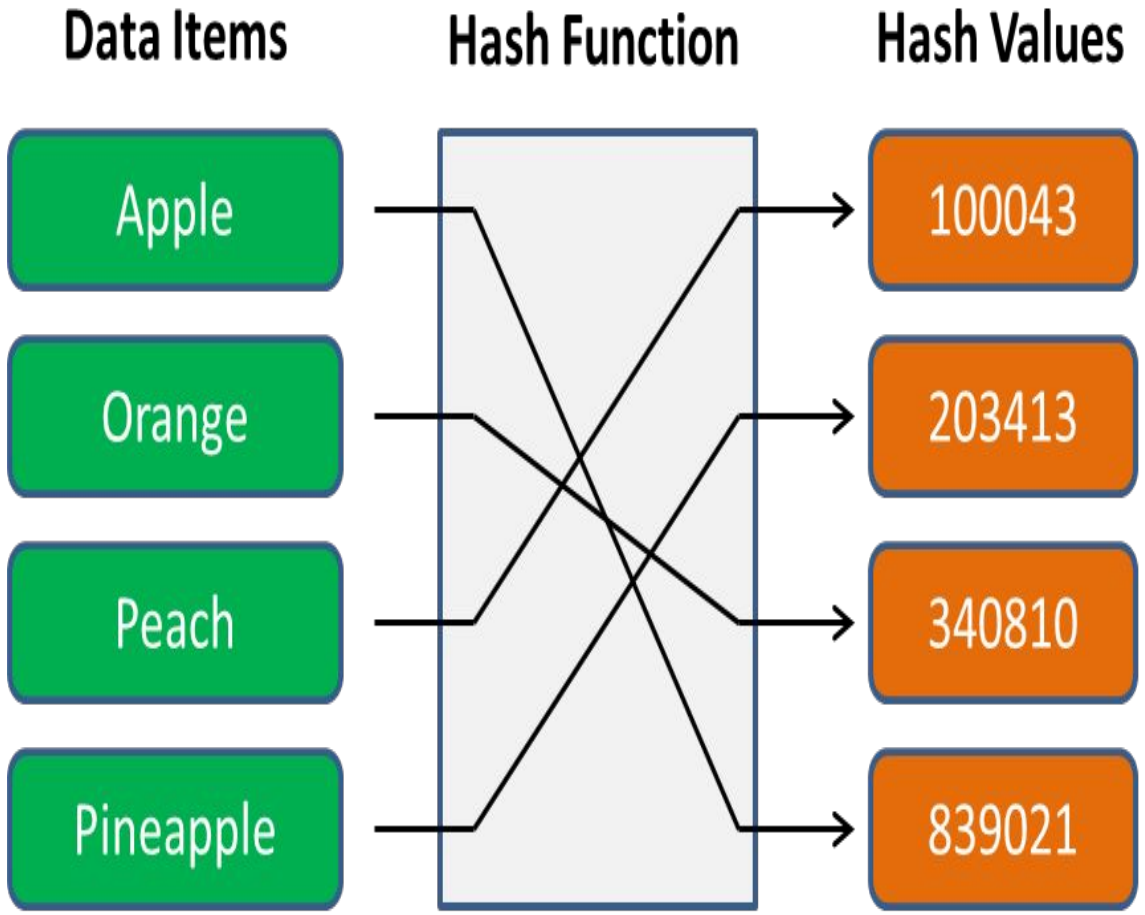
BRAC UNIVERSITY
Inspiring Excellence

Hashing: Why?

- Hashing is the process of applying an algorithm to convert a data item to a value. The data item can be as simple as an integer, a string, or as complex as an object with multiple properties.

- The algorithm is termed the hash function or hasher. The converted value is termed the hash value, hash code, or simply hash

- The diagram shows you a simplified example of a hashing process. Essentially, we start with four kinds of fruit (i.e., data items) on the left side.

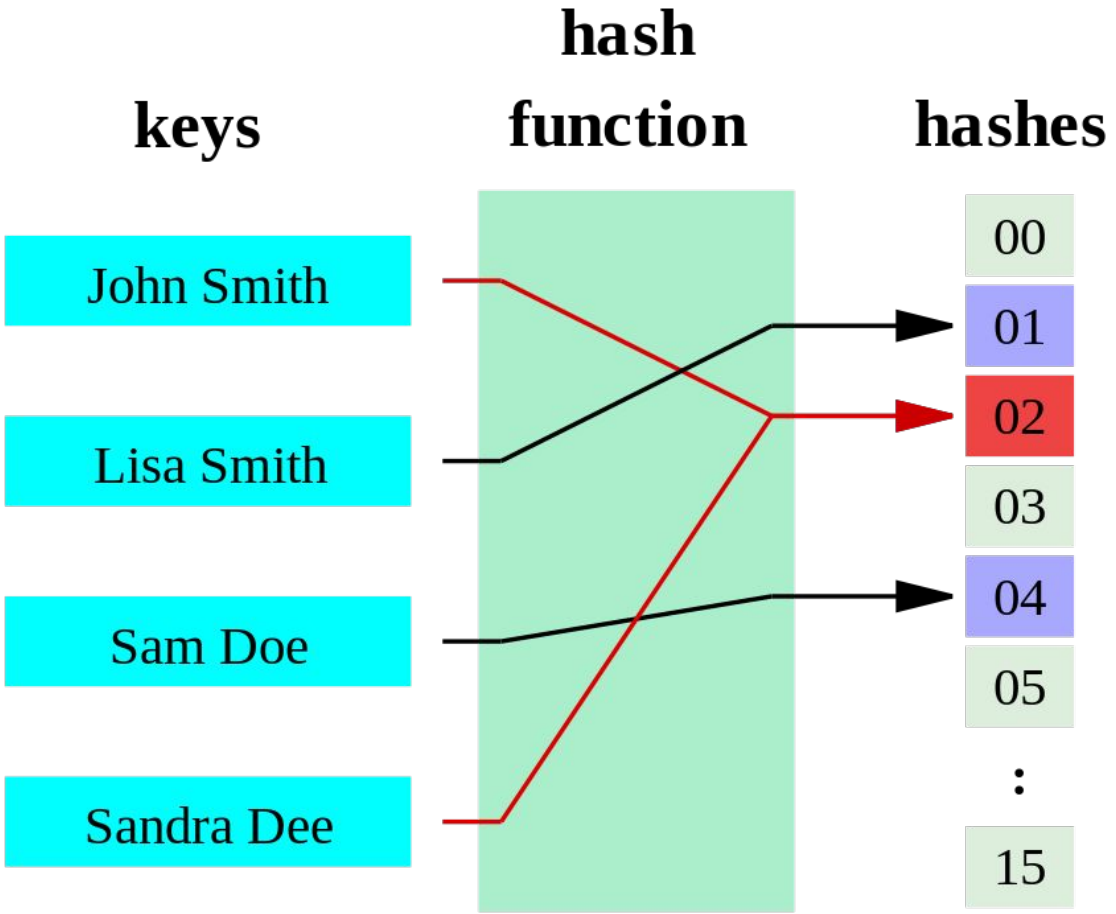


Hash Function

- A hash function is
  any function that can be used to
  map data of arbitrary size to
  fixed-size values.

- The values returned by a hash
  function are
  called hash values, hash codes,
  digests, or simply hashes.

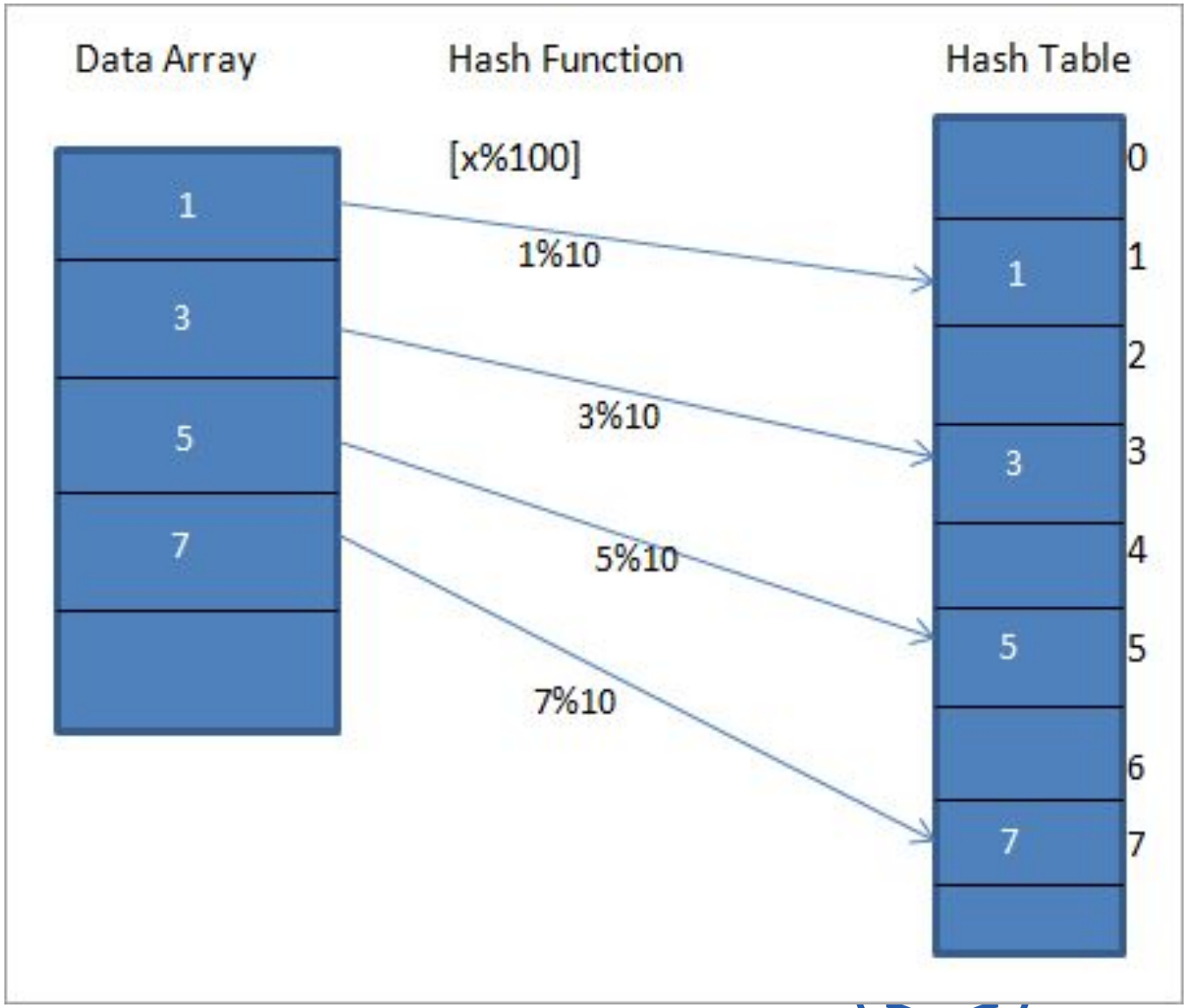
- The values are used to index a
  fixed-size table called
  a hash table.



BRAC UNIVERSITY
Inspiring Excellence

Hash function and Array

- Hash function can be used to hash object key (which is email) to an integer number of fixed size.
- We can then use array to store the employee details in such a way that, index i has employee details whose key hash value is *i*.
- But ideally the output range of hash functions are very large, and it will be impractical and waste of memory to store objects in array



Distributed Hashing (Hash Table)

- A distributed hash table (DHT) is a distributed system that provides a lookup service like a hash table: key-value pairs are stored in a DHT, and any participating node can efficiently retrieve the value associated with a given key.
- The main advantage of a DHT is that nodes can be added or removed with minimum work around re-distributing keys.
- Keys are unique identifiers which map to values, which in turn can be anything from addresses, to documents, to arbitrary data.
- Responsibility for maintaining the mapping from keys to values is distributed among the nodes, in such a way that a change in the set of participants causes a minimal amount of disruption.
- This allows a DHT to scale to extremely large numbers of nodes and to handle continual node arrivals, departures, and failures.

Distributed Hash Table Properties

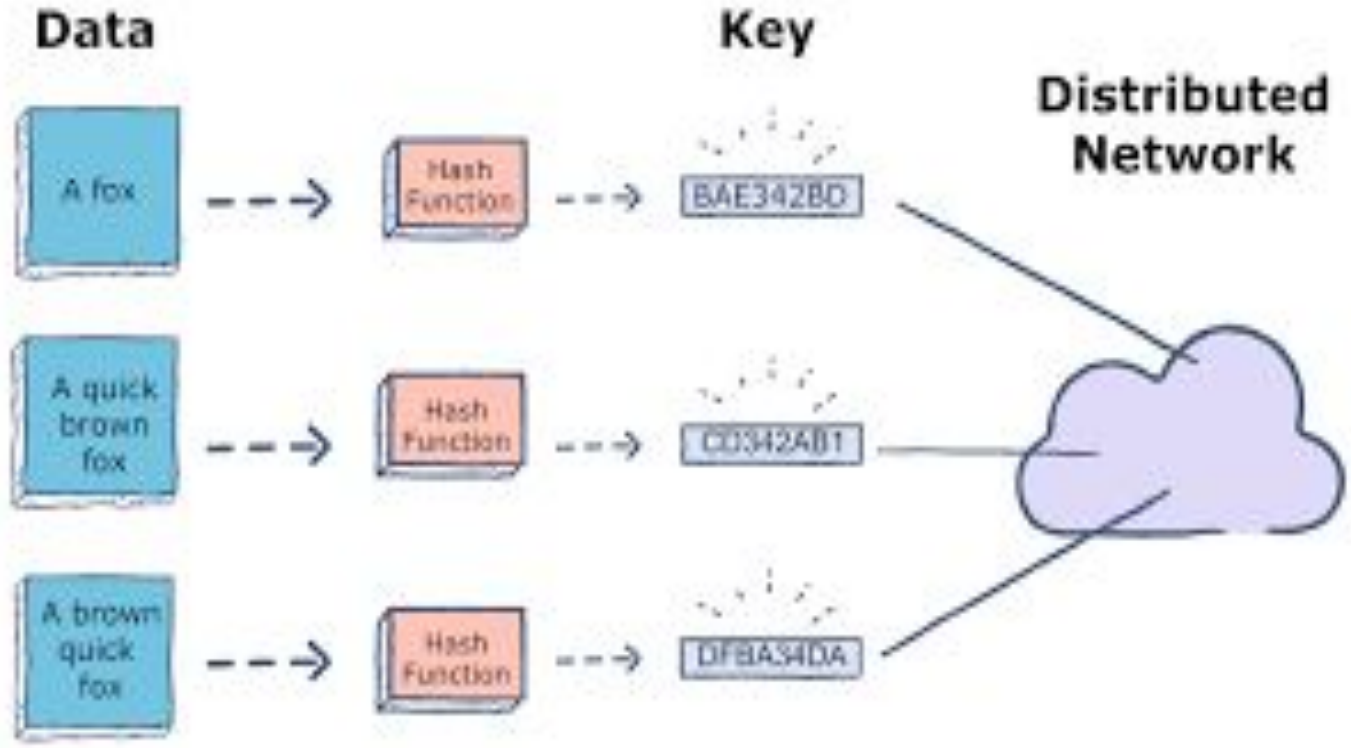
- Autonomy and decentralization: the nodes collectively form the system without any central coordination.
- Fault tolerance: the system should be reliable (in some sense) even with nodes continuously joining, leaving, and failing.
- Scalability: the system should function efficiently even with thousands or millions of nodes.

A key technique used to achieve these goals is that any one node needs to coordinate with only a few other nodes in the system – most commonly, O(log *n*) of the *n* participants – so that only a limited amount of work needs to be done for each change in membership.

DHTs must deal with more traditional distributed systems issues such as load balancing, data integrity, and performance

Distributed Hash Table: Structure

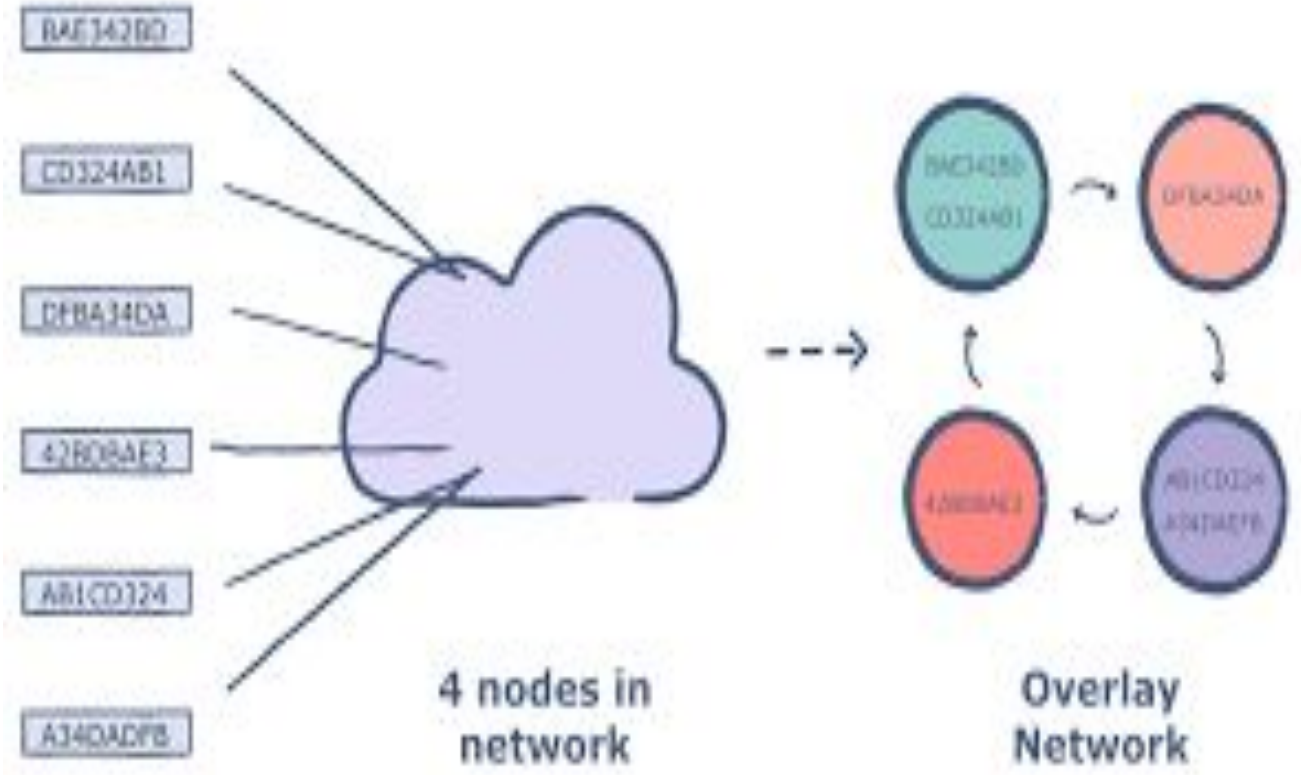
- The structure of a DHT can be decomposed into several main components.
- The foundation is an abstract key space, such as the set of 160-bit strings.
- A key space partitioning scheme splits ownership of this key space among the participating nodes.
- An overlay network then connects the nodes, allowing them to find the owner of any given key in the key space.



BRAC UNIVERSITY
Inspiring Excellence

Distributed Hash Table: Structure

- The structure of a DHT can be decomposed into several main components.
- The foundation is an abstract key space, such as the set of 160-bit strings.
- A key space partitioning scheme splits ownership of this key space among the participating nodes.
- An overlay network then connects the nodes, allowing them to find the owner of any given key in the key space.

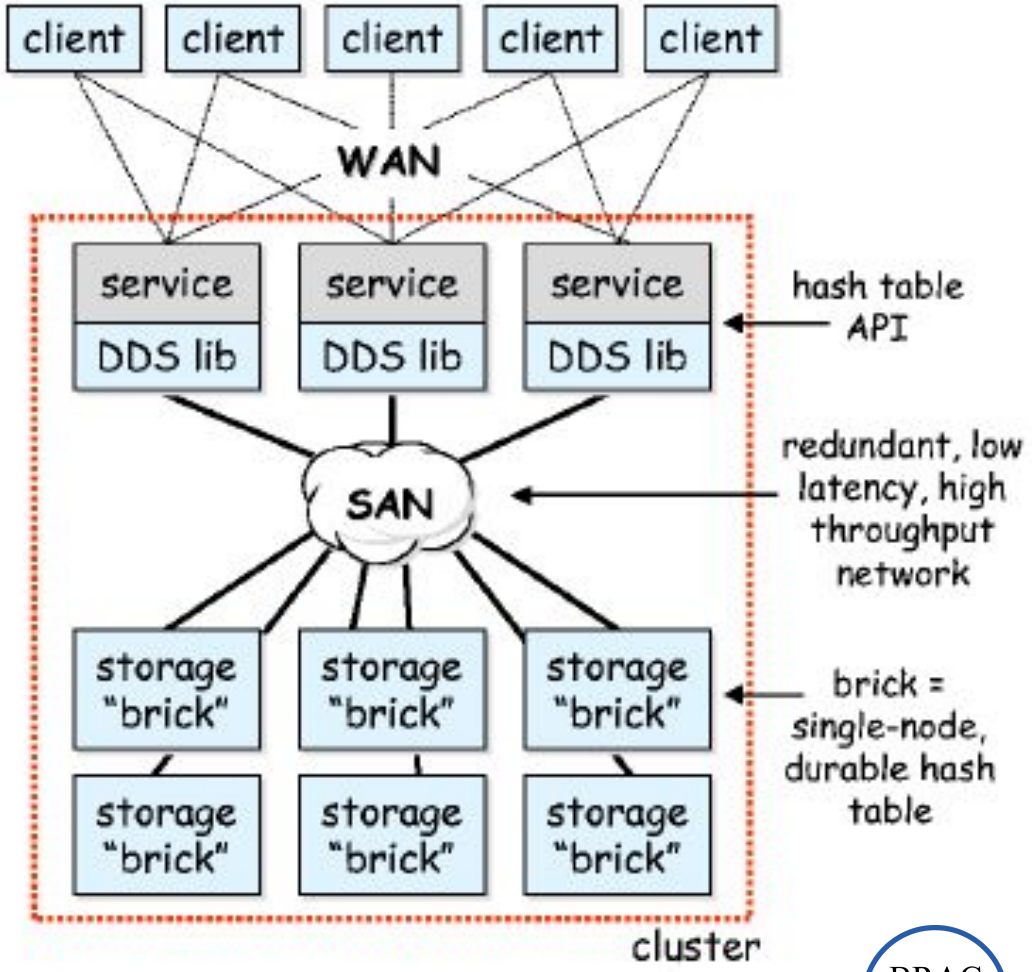


- Once these components are in place, a typical use of the DHT for storage and retrieval might proceed as follows.
- Suppose the key space is the set of 160-bit strings. To index a file with given filename and data in the DHT, the SHA-1 hash of filename is generated, producing a 160-bit key k, and a message put(k, data) is sent to any node participating in the DHT.
- The message is forwarded from node to node through the overlay network until it reaches the single node responsible for key k as specified by the key space partitioning.
- That node then stores the key and the data.
- Any other client can then retrieve the contents of the file by again hashing filename to produce k and asking any DHT node to find the data associated with k with a message get(k).
- The message will again be routed through the overlay to the node responsible for k, which will reply with the stored data.
- The key space partitioning and overlay network components are described below with the goal of capturing the principal ideas common to most DHTs; many designs differ in the details.

BRAC UNIVERSITY
Inspiring Excellence

Distributed Hash Table: Architecture and Implementation

- Client: a client consists of service-specific software running on a client machine that communicates across the wide area with one of many service instances running in the cluster.
- The mechanism by which the client selects a service instance is beyond the scope of this work, but it typically involves DNS round robin, a service-specific protocol, or level 4 or level 7 load-balancing switches on the edge of the cluster.
- An example of a client is a web browser, in which case the service would be a web server. Note that clients are completely unaware of DDS's: no part of the DDS system runs on a client.

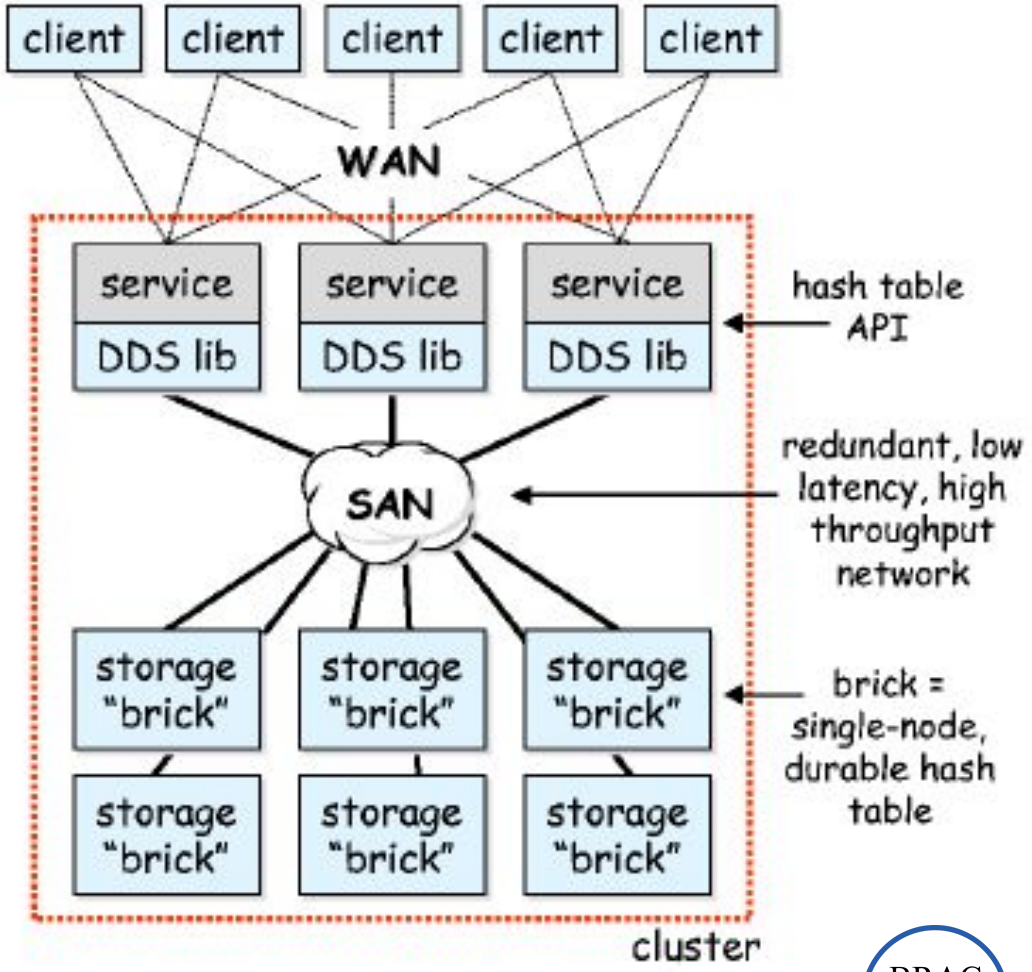


BRAC UNIVERSITY
Inspiring Excellence

Distributed Hash Table: Architecture and Implementation

Service: a service is a set of cooperating software processes, each of which we call a service instance. Service instances communicate with wide-area clients and perform some application-level function. Services may have soft state (state which may be lost and recomputed if necessary), but they rely on the hash table to manage all persistent state.

Hash table API: the hash table API is the boundary between a service instance and its ``DDS library''. The API provides services with put(), get(), remove(), create(), and destroy() operations on hash tables. Each operation is atomic, and all services see the same coherent image of all existing hash tables through this API. Hash table names are strings, hash table keys are 64-bit integers, and hash table values are opaque byte arrays; operations affect hash table values in their entirety.



BRAC UNIVERSITY
Inspiring Excellence

Partitioning, Replication, and Replica Consistency

A distributed hash table provides incremental scalability of throughput and data capacity as more nodes are added to the cluster. To achieve this, we horizontally partition tables to spread operations and data across bricks. Each brick thus stores some number of *partitions* of each table in the system, and when new nodes are added to the cluster, this partitioning is altered so that data is spread onto the new node. Because of our workload assumptions, this horizontal partitioning evenly spreads both load and data across the cluster.

Each partition in the hash table is replicated on more than one cluster node. The set of replicas for a partition form a replica group; all replicas in the group are kept strictly coherent with each other. Any replica can be used to service a get(), but all replicas must be updated during a put() or remove(). If a node fails, the data from its partitions is available on the surviving members of the partitions' replica groups. Replica group membership is thus dynamic; when a node fails, all its replicas are removed from their replica groups. When a node joins the cluster, it may be added to the replica groups of some partitions (such as in the case of recovery, described later).

Partitioning, Replication, and Replica Consistency

To maintain consistency when state changing operations (put() and remove()) are issued against a partition, all replicas of that partition must be synchronously updated. We use an optimistic two-phase commit protocol to achieve consistency, with the DDS library serving as the commit coordinator and the replicas serving as the participants. If the DDS library crashes after *prepare* messages are sent, but before any *commit* messages are sent, the replicas will time out and abort the operation.

However, if the DDS library crashes after sending out any *commits*, then all replicas must commit. For the sake of availability, we do not rely on the DDS library to recover after a crash and issuing pending *commits*. Instead, replicas store short in-memory logs of recent state changing operations and their outcomes. If a replica times out while waiting for a *commit*, that replica communicates with all of its peers to find out if any have received a *commit* for that operation, and if so, the replica commits as well; if not, the replica aborts. Because all peers in the replica group that time out while waiting for a commit communicate with all other peers, if any receives a commit, then all will commit.

.....

BRAC UNIVERSITY
Inspiring Excellence

Partitioning, Replication, and Replica Consistency

If a replica crashes during a two-phase commit, the DDS library simply removes it from its replica group and continues onward. Thus, all replica groups shrink over time; we rely on a recovery mechanism (described later) for crashed replicas to rejoin the replica group. We made the significant optimization that the image of each replica must only be consistent through its brick's cache, rather than having a consistent on-disk image. This allows us to have a purely conflict-driven cache eviction policy, rather than having to force cache elements out to ensure on-disk consistency. An implication of this is that if all members of a replica group crash, that partition is lost. We assume nodes are independent failure boundaries; there must be no systematic software failure across nodes, and the cluster's power supply must be uninterruptible.

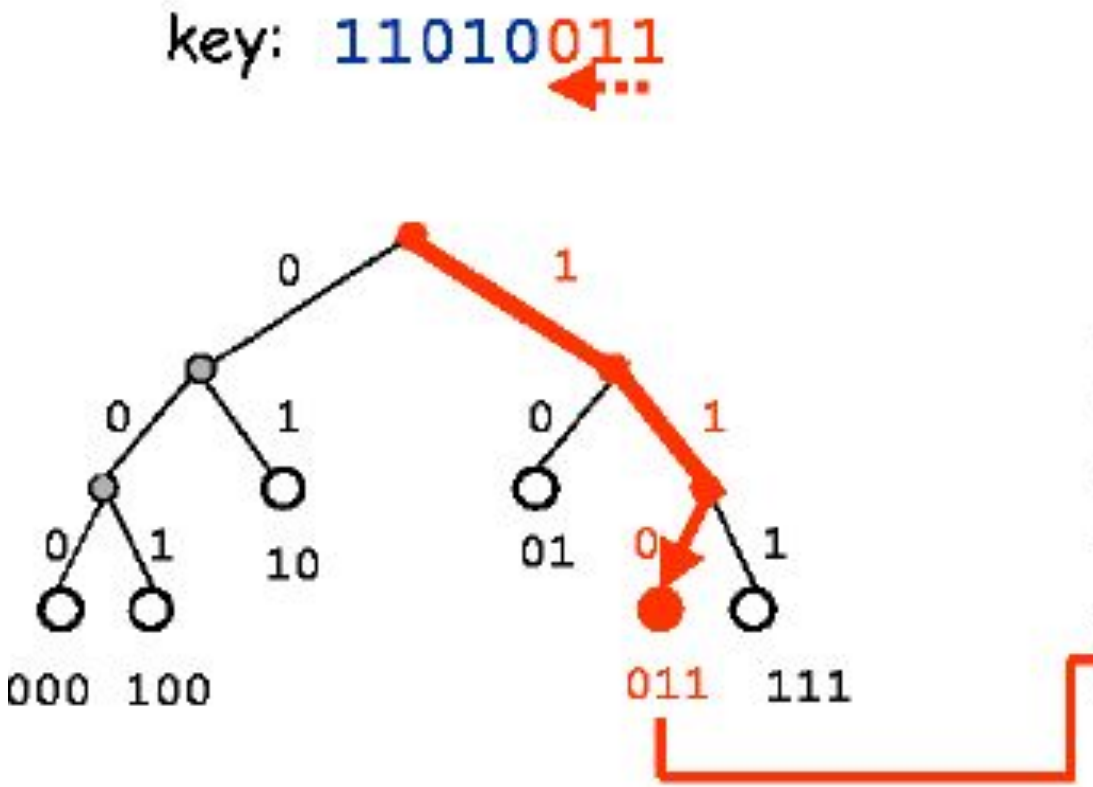
BRAC UNIVERSITY
Inspiring Excellence

Metadata maps

The first map is called the data partitioning (DP) map. Given a hash table key, the DP map returns the name of the key's partition. The DP map thus controls the horizontal partitioning of data across the bricks. As shown in figure, the DP map is a tire over hash table keys; to find a key's partition, key bits are used to walk down the tire, starting from the least significant key bit until a leaf node is found. As the cluster grows, the DP tire subdivides in a ``split'' operation. For example, partition 10 in the DP could split into partitions 010 and 110; when this happens, the keys in the old partition are shuffled across the two new partitions. The opposite of a split is a ``merge''; if the cluster is shrunk, two partitions with a common parent in the tire can be merged into their parent. For example, partitions 000 and 100 could be merged into a single partition 00.

The second map is called the replica group (RG) membership map. Given a partition name, the RG map returns a list of bricks that are currently serving as replicas in the partition's replica group. The RG maps are dynamic: if a brick fails, it is removed from all RG maps that contain it. A brick joins a replica group after finishing recovery. An invariant that must be preserved is that the replica group membership maps for all partitions in the hash table must have at least one member.

The maps are replicated on each cluster node, in both the DDS libraries and the bricks. The maps must be kept consistent, otherwise operations may be applied to the wrong bricks. Instead of enforcing consistency



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

Step 1: lookup key in DP map to find RGname

Step 2: lookup RGname in RG map to find list of replicas

BRAC UNIVERSITY
Inspiring Excellence

Recovery

If a brick fails, all replicas on it become unavailable. Rather than making these partitions unavailable, we remove the failed brick from all replica groups and allow operations to continue the surviving replicas. When the failed brick recovers (or an alternative brick is selected to replace it), it must ``catch up'' to all the operations it missed. In many RDBMS's and file systems, recovery is a complex process that involves replaying logs, but in our system, we use properties of clusters and our DDS design for vast simplifications.

There is an interesting choice of the rate at which partitions are transferred over the network during recovery. If this rate is fast, then the involved bricks will suffer a loss in read throughput during the recovery. If this rate is slow, then the bricks won't lose throughput, but the partition's mean time to recovery will increase. We chose to recover as quickly as possible, since in a large cluster only a small fraction of the total throughput of the cluster will be affected by the recovery.

BRAC UNIVERSITY
Inspiring Excellence

Convergence of Recovery

A challenge for fault-tolerant systems is to remain consistent in the face of repeated failures; our recovery scheme described above has this property. In steady state operation, all replicas in a group are kept perfectly consistent. During recovery, state changing operations fail (but only on the recovering partition), implying that surviving replicas remain consistent and recovering nodes have a stable image from which to recover. We also ensure that a recovering node only joins the replica group after it has successfully copied over the entire partition's data but before it release its write lease. A remaining window of vulnerability in the system is if recovery takes longer than the write lease; if this seems imminent, the recovering node could aggressively renew its write lease, but we have not currently implemented this behavior.

BRAC UNIVERSITY
Inspiring Excellence

Rehashing and Rehashing Problems

BRAC UNIVERSITY
Inspiring Excellence

Outline

- Rehashing and Rehashing Example
- How pertions are placed in swift
- Active Replication of Swift
- Ring Builder

Why we need rehashing?

- When one of the server crashes or becomes unavailable !

-> Keys need to be redistributed to account for the missing server.

- What if one or more new servers are added to the pool

-> keys need to be redistributed to include the new servers

BRAC UNIVERSITY
Inspiring Excellence

Example 1

Removing a server

Remving a server

key	Hash	Node
1	7739	2
2	8986	1
3	8338	1

---------->

key	Hash	Node
1	7739	1
2	8986	0
3	8338	0

For example, If we consider 3 servers. In the left side of the picture if a client wants to retrieve 1. Its hash value = 7739. It will go to Node number (7739%3). So, it will contact node 2.

If we remove server C, the allocated nodes for each key will change.

It will go to Node number (7739%2). So, it will contact node 1.

Hash function = (data hash%number of servers)

BRAC UNIVERSITY
Inspiring Excellence

put(DATA)

Hash function = (data hash%number of servers)

THEN PUT IN THE NODE // ?

get(DATA)

Hash function = (data hash%number of servers) //??

THEN RETRIEVE From THE NODE

DRIVER CODE:

PUT("HELLO WORLD")

GET("HELLO WORLD")

Example 2

Adding a Server

Adding a server

key	Hash	Node
1	7739	2
2	8986	1
3	8338	1

---------->

key	Hash	Node
1	7739	3
2	8986	2
3	8338	2

If we add server D, the allocated nodes for each key will also change.

It will go to Node number (7739%4). So, it will contact node 3.

BRAC UNIVERSITY
Inspiring Excellence

PROBLEMS WITH REHASHING

- The distribution changes whenever we change the number of servers.
- Sometimes, the keys won't be found because they won't yet be present at their new location.
- With the more number of redistributions, the more number of misses, the more number of memory fetches, placing an additional load on the node and thus decreasing the performance.

BRAC UNIVERSITY
Inspiring Excellence

What next?

- ➔ **How to solve rehashing problem**
- ➔ **Consistent Hashing**

BRAC UNIVERSITY
Inspiring Excellence

References

OpenStack Swift

A Guide to Consistent Hashing

Hashing in Distributed Systems

Distributed Database Things to Know: Consistent Hashing

BRAC UNIVERSITY
Inspiring Excellence

References

https://www.toptal.com/big-data/consistent-hashing

http://courses.cse.tamu.edu/caverlee/csce438/readings/consistent-hashing.pdf

https://www.researchgate.net/figure/Virtual-Rehashing-of-RQALSH-for-c-2_fig1_319867705

https://www.ably.io/blog/implementing-efficient-consistent-hashing

BRAC UNIVERSITY
Inspiring Excellence

Reference:

https://medium.com/system-design-blog/consistent-hashing-b9134c8a9062

https://en.wikipedia.org/wiki/Distributed_hash_table

https://www.educative.io/edpresso/what-is-a-distributed-hash-table

https://hazelcast.com/glossary/distributed-hash-table/

https://medium.com/system-design-blog/consistent-hashing-b9134c8a9062

https://link.springer.com/referenceworkentry/10.1007%2F978-0-387-39940-9_1232

https://www.toptal.com/big-data/consistent-hashing

BRAC UNIVERSITY

Inspiring Excellence

Acknowledge to

Md Shamiul Islam, Md Nazmur Sakib, Md Sadiqul Islam Sakif

BRAC UNIVERSITY

Inspiring Excellence