Dynamics of Distributed Hash Tables

Wolf-Tilo Balke and Wolf Siberski

21.11.2007

Overview

1. CHORD (again)

2. New node joins: Stabilization

3. Reliability in Distributed Hash Tables

4. Storage Load Balancing in Distributed Hash Tables

5. The P-GRID Approach
Distributed Hash Tables

• Distributed Hash Tables (DHTs)
  ▶ Also known as structured Peer-to-Peer systems
  ▶ Efficient, scalable, and self-organizing algorithms
  ▶ For data retrieval and management

• Chord (Stoica et al., 2001)
  ▶ Scalable Peer-to-peer Lookup Service for Internet Applications
  ▶ Nodes and data are mapped with hash function on a Chord ring
  ▶ Routing
    ▶ Routing ("Finger") - Tables
    ▶ O (log N)

CHORD

• A consistent hash function assigns each node and each key an m-bit identifier using SHA 1 (Secure Hash Standard).
  ▶ m = any number big enough to make collisions improbable
  ▶ Key identifier = SHA-1(key)
  ▶ Node identifier = SHA-1(IP address)

• Both are uniformly distributed
• Both exist in the same ID space
CHORD

- Identifiers are arranged on a closed identifier circle (e.g. modulo $2^6$)

=> Chord Ring

CHORD

- A key $k$ is assigned to the node whose identifier is equal to or greater than the key's identifier
- This node is called successor($k$) and is the first node clockwise from $k$
CHORD

// ask node n to find the successor of id
n.find_successor(id)
  if (id = (n; successor))
    return successor;
  else
    // forward the query around the circle
    return successor.find_successor(id);

=> Number of messages linear in the number of nodes!

CHORD

- Additional routing information to accelerate lookups
- Each node n contains a routing table with up to m entries (m: number of bits of the identifiers)
  => finger table
- i\textsuperscript{th} entry in the table at node n contains the first node s that succeeds n by at least 2
- s = successor (n + 2\textsuperscript{i-1})
- s is called the i\textsuperscript{th} finger of node n
Finger table:  
\[ \text{finger}[i] := \text{successor}(n + 2^i) \]

- Search in finger table for the nodes which most immediately precedes id
- Invoke \text{find_successor} from that node

\[ \Rightarrow \text{Number of messages } O(\log N) \]
CHORD

- Important characteristics of this scheme:
  - Each node stores information about only a small number of nodes (m)
  - Each node knows more about nodes closely following it than about nodes farther away
  - A finger table generally does not contain enough information to directly determine the successor of an arbitrary key k

Overview

1. CHORD (again)
2. New node joins: Stabilization
3. Reliability in Distributed Hash Tables
4. Storage Load Balancing in Distributed Hash Tables
5. The P-GRID Approach
Volatility

- Stable CHORD networks can always rely on their finger tables for routing

- But what happens, if the network is volatile (churn)?

- Lecture 3: Nodes can fail, depart or join
  - Failure is handled by ‘soft-state’ approach with periodical republish by nodes
  - Departure can be handled by clean shutdown (unpublish) or simply as failure
  - Joining needs stabilization of the CHORD ring
**CHORD Stabilization**

- Stabilization protocol for a node x:
  - x.stabilize():
    - ask successor y for its predecessor p
    - if $p \in (x; y]$ then p is x's new successor
  - x.notify():
    - notify x's successor p of x's existence
    - notified node may change predecessor to x

**CHORD Stabilization**

• N26 joins the system
  - N26 aquires N32 as its successor
  - N26 notifies N32
  - N32 aquires N26 as its predecessor
CHORD Stabilization

- N26 copies keys
- N21 runs stabilize() and asks its successor N32 for its predecessor which is N26.

CHORD Stabilization

- N21 acquires N26 as its successor
- N21 notifies N26 of its existence
- N26 acquires N21 as predecessor
Overview

1. CHORD (again)

2. New node joins: Stabilization

3. Reliability in Distributed Hash Tables

4. Storage Load Balancing in Distributed Hash Tables

5. The P-GRID Approach
“Stabilize” Function

- Stabilize Function to correct inconsistent connections
- Remember:
  - Periodically done by each node n
  - n asks its successor for its predecessor p
  - n checks if p equals n
  - n also periodically refreshes random finger x
    - by (re)locating successor
- Successor-List to find new successor
  - If successor is not reachable use next node in successor-list
  - Start stabilize function
- But what happens to data in case of node failure?

Reliability of Data in Chord

- Original
  - No Reliability of data
- Recommendation
  - Use of Successor-List
  - The reliability of data is an application task
  - Replicate inserted data to the next f other nodes
  - Chord informs application of arriving or failing nodes
Properties

- **Advantages**
  - After failure of a node its successor has the data already stored

- **Disadvantages**
  - Node stores $f$ intervals
    - More data load
  - After breakdown of a node
    - Find new successor
    - Replicate data to next node
      - More message overhead at breakdown
  - Stabilize-function has to check every Successor-list
    - Find inconsistent links
      - More message overhead

---

Multiple Nodes in One Interval

- **Fixed positive number $f$**
  - Indicates how many nodes have to act within one interval at least

- **Procedure**
  - First node takes a random position
  - A new node is assigned to any existing node
  - Node is announced to all other nodes in same interval
Multiple Nodes in One Interval

- Effects of algorithm
  - Reliability of data
  - Better load balancing
  - Higher security

Reliability of Data

- Insertion
  - Copy of documents
    - Always necessary for replication
  - Less additional expenses
    - Nodes have only to store pointers to nodes from the same interval
  - Nodes store only data of one interval
Reliability of Data

- **Reliability**
  - Failure: no copy of data needed
    - Data are already stored within same interval
  - Use stabilization procedure to correct fingers
    - As in original Chord

Properties

- **Advantages**
  - Failure: no copy of data needed
  - Rebuild intervals with neighbors only if critical
  - Requests can be answered by \( f \) different nodes

- **Disadvantages**
  - Less number of intervals as in original Chord
    - Solution: Virtual Servers
### Fault Tolerance: Replication vs. Redundancy

<table>
<thead>
<tr>
<th><strong>Replication</strong></th>
<th><strong>Redundancy</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Each data item is replicated K times</td>
<td>Each data item is split into M fragments</td>
</tr>
<tr>
<td>K replicas are stored on different nodes</td>
<td>K redundant fragments are computed</td>
</tr>
<tr>
<td></td>
<td>- Use of an &quot;erasure-code&quot; (see e.g. V. Pless: <em>Introduction to the Theory of Error-Correcting Codes</em>. Wiley-Interscience, 1998)</td>
</tr>
<tr>
<td></td>
<td>Any M fragments allow to reconstruct the original data</td>
</tr>
<tr>
<td></td>
<td>For each fragment we compute its key</td>
</tr>
<tr>
<td></td>
<td>M + K different fragments have different keys</td>
</tr>
</tbody>
</table>

### Overview

1. CHORD (again)
2. New node joins: Stabilization
3. Reliability in Distributed Hash Tables
4. Storage Load Balancing in Distributed Hash Tables
5. The P-GRID Approach
Storage Load Balancing in Distributed Hash Tables

- Suitable hash function (easy to compute, few collisions)

- Standard assumption 1: uniform key distribution
  - Every node with equal load
  - No load balancing is needed

- Standard assumption 2: equal distribution
  - Nodes across address space
  - Data across nodes

- But is this assumption justifiable?
  - Analysis of distribution of data using simulation

Storage Load Balancing in Distributed Hash Tables

- Analysis of distribution of data

- Example
  - Parameters
    - 4,096 nodes
    - 500,000 documents
  - Optimum
    - ~122 documents per node

→ No optimal distribution in Chord without load balancing
Storage Load Balancing in Distributed Hash Tables

- Number of nodes without storing any document
  - Parameters
    - 4,096 nodes
    - 100,000 to 1,000,000 documents
  - Some nodes without any load

Why is the load unbalanced?

- We need load balancing to keep the complexity of DHT management low

Definitions

- Definitions
  - System with N nodes
  - The load is optimally balanced,
    - Load of each node is around 1/N of the total load.
  - A node is overloaded (heavy)
    - Node has a significantly higher load compared to the optimal distribution of load.
  - Else the node is light
Load Balancing Algorithms

- Problem
  - Significant difference in the load of nodes

- Several techniques to ensure an equal data distribution
  - Power of Two Choices (Byers et. al, 2003)
  - Virtual Servers (Rao et. al, 2003)
  - Thermal-Dissipation-based Approach (Rieche et. al, 2004)
  - A Simple Address-Space and Item Balancing (Karger et. al, 2004)
  - ...

Outline

- Algorithms
  - Power of Two Choices (Byers et. al, 2003)
    - Virtual Servers (Rao et. al, 2003)

Power of Two Choices

- **Idea**
  - One hash function for all nodes
    - $h_0$
  - Multiple hash functions for data
    - $h_1, h_2, h_3, \ldots h_d$

- **Two options**
  - Data is stored at one node
  - Data is stored at one node & other nodes store a pointer

---

**Inserting Data**

- Results of all hash functions are calculated
  - $h_1(x), h_2(x), h_3(x), \ldots h_d(x)$
- Data is stored on the retrieved node with the lowest load
- Alternative
  - Other nodes stores pointer
- The owner of a data has to insert the document periodically
  - Prevent removal of data after a timeout (soft state)
Power of Two Choices (cont'd)

- **Retrieving**
  - Without pointers
    - Results of all hash functions are calculated
    - Request all of the possible nodes in parallel
    - One node will answer
  - With pointers
    - Request only one of the possible nodes.
    - Node can forward the request directly to the final node

- **Advantages**
  - Simple

- **Disadvantages**
  - Message overhead at inserting data
  - With pointers
    - Additional administration of pointers
      - More load
  - Without pointers
    - Message overhead at every search
Outline

- Algorithms
  - Power of Two Choices (Byers et. al, 2003)
  - Virtual Servers (Rao et. al, 2003)


Virtual Server

- Each node is responsible for several intervals
  - "Virtual server"
- Example
  - Chord

[Image of a Chord network diagram with nodes A, B, and C connected in a ring.]

[Rao 2003]
Rules

- Rules for transferring a virtual server
  - From heavy node to light node

1. The transfer of a virtual server makes the receiving node not heavy
2. The virtual server is the lightest virtual server that makes the heavy node light
3. If there is no virtual server whose transfer can make a node light, the heaviest virtual server from this node would be transferred

Virtual Server

- Each node is responsible for several intervals
  - log (n) virtual servers

- Load balancing
  - Different possibilities to change servers
    - One-to-one
    - One-to-many
    - Many-to-many
  - Copy of an interval is like removing and inserting a node in a DHT
**Scheme 1: One-to-One**

- **One-to-One**
  - Light node picks a random ID
  - Contacts the node $x$ responsible for it
  - Accepts load if $x$ is heavy

**Scheme 2: One-to-Many**

- **One-to-Many**
  - Light nodes report their load information to directories
  - Heavy node $H$ gets this information by contacting a directory
  - $H$ contacts the light node which can accept the excess load

[Rao 2003]
Scheme 3: Many-to-Many

- Many-to-Many
  - Many heavy and light nodes rendezvous at each step
  - Directories periodically compute the transfer schedule and report it back to the nodes, which then do the actual transfer

Virtual Server

- Advantages
  - Easy shifting of load
    - Whole Virtual Servers are shifted

- Disadvantages
  - Increased administrative and messages overhead
    - Maintenance of all Finger-Tables
  - Much load is shifted
Simulation

- Scenario
  - 4,096 nodes (comparison with other measurements)
  - 100,000 to 1,000,000 documents

- Chord
  - m = 22 bits.
  - Consequently, $2^{22} = 4,194,304$ nodes and documents

- Hash function
  - sha-1 (mod $2^m$)
  - random

- Analysis
  - Up to 25 runs per test

Results

- Without load balancing
  - Simple
  - Original
    - Bad load balancing

- Power of Two Choices
  - Simple
  - Lower load
    - Nodes w/o load
Results (cont’d)

- Virtual server

![Graph showing virtual server performance]

+ No nodes w/o load
- Higher max. load than Power of Two Choices

Overview

1. CHORD (again)

2. New node joins: Stabilization

3. Reliability in Distributed Hash Tables

4. Storage Load Balancing in Distributed Hash Tables

5. The P-GRID Approach
• Toy example: Distributing skewed load

• A globally coordinated recursive bisection approach

• Key-space can be divided in two partitions
  ▶ Assign peers proportional to the load in the two sub-partitions
• Recursively repeat the process to repartition the sub-partitions

Partitioning of the key-space so that there is equal load in each partition
  ▶ Uniform replication of the partitions
  ▶ Important for fault-tolerance

• Note: A novel and general load-balancing problem
Lessons from the globally coordinated algorithm

- Achieves an approximate load-balance
- The intermediate partitions may be such that they cannot be perfectly repartitioned
  - There’s a fundamental limitation with any bisection based approach, as well as for any fixed key-space partitioned overlay network
- Nonetheless practical
  - For realistic load-skews and peer populations

Distributed proportional partitioning for overlay construction

- A mechanism to meet other random peers
- A parameter $p$ for partitioning the space
- **Proportional partitioning**: Peers partition proportional to the load distribution
  - In a ratio $p:1-p$
  - Let’s call the sub-partitions as 0 and 1
- **Referential integrity**: Obtain reference to the other partition
  - Needed to enable overlay routing
- **Sorting the load/keys**: Peers exchange locally stored keys in order to store only keys for its own partition
P-Grid

- Scalable Distributed Search Tries (prefix tree)

Peer 1 stores all data with prefixes 000 or 001

A single peer should *not* hold the entire index

Distribute index *disjoint* over peers
P-Grid

- Scalable Distribution of Index

Routing information at each peer is only logarithmic (height of trie)
**P-Grid**

- **Prefix Routing**

  Routing table of peer 4
  
<table>
<thead>
<tr>
<th>prefix</th>
<th>peer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0??</td>
<td>peer1</td>
</tr>
<tr>
<td>10??</td>
<td>peer2</td>
</tr>
<tr>
<td>101</td>
<td>peer3</td>
</tr>
</tbody>
</table>

  peers meet (randomly) and decide whether to extend search tree by splitting the data space

  peers can perform load balancing considering their storage load

  networks with different origins can merge, like Gnutella, FreeNet (loose coupling)

- **Two basic approaches for new nodes to join**

  - **Splitting approach (P-Grid)**
    - peers meet (randomly) and decide whether to extend search tree by splitting the data space
    - peers can perform load balancing considering their storage load
    - networks with different origins can merge, like Gnutella, FreeNet (loose coupling)

  - **Node insertion approach (Plaxton, OceanStore, ...)**
    - peers determine their "leaf position" based on their IP address
    - nodes route from a gateway node to their node-id to populate the routing table
    - network has to start from single origin (strong coupling)
P-Grid

- **Load balance effect**
  - Algorithm converges quickly
  - Peers have similar load
  - E.g. leaf load in case of $2^5 = 32$ possible prefixes of length 5:

![Graph showing load distribution among peers and data objects.]

---

P-Grid

- Replication of data items and routing table entries is used to increase failure resilience.