Peer-to-Peer Systems

Structured P2P file sharing systems

Michael Welzl michael.welzl@uibk.ac.at

DPS NSG Team http://dps.uibk.ac.at/nsg
Institute of Computer Science
University of Innsbruck, Austria

Part 1: Introducing DHTs

Searching and Addressing

- Two basic ways to find objects:
  1. Search for them
  2. Address them using their unique name
- Both have pros and cons (see below)
- Most existing P2P networks built on searching, but some networks are based on addressing objects
- Difference between searching and addressing is fundamental
  - Determines how network is constructed
  - Determines how objects are placed
  - Determines efficiency of object location
- Let's compare searching and addressing

Addressing vs. Searching

- "Addressing" networks find objects by addressing them with their unique name (cf. URLs in Web)
- "Searching" networks find objects by searching with keywords that match objects's description (cf. Google)

Addressing

- Pros:
  - Each object uniquely identifiable
  - Object location can be made efficient
- Cons:
  - Need to know unique name
  - Need to maintain structure required by addresses

Searching

- Pros:
  - No need to know unique names
    - More user friendly
- Cons:
  - Hard to make efficient
    - Can solve with money, see Google
  - Need to compare actual objects to know if they are same

Addressing vs. Searching: Examples

	Searching	Addressing
Physical name of object	Searching in P2P networks, Searching in filesystem (Desktop searches) (Search components of URL with Google?)	URLs in Web
Logical name of object	? (Search components of URNs)	Object names in DHT, URNs
Content or metadata of object	Searching in P2P networks, Standard Google search Desktop searches	N/A

Searching, Addressing, and P2P

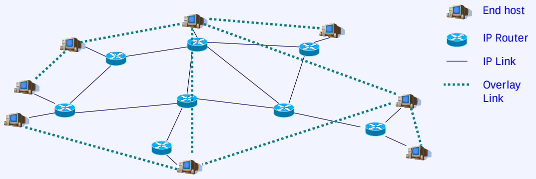
- We can distinguish between two main P2P network types

Unstructured networks/systems
- Based on searching
- Unstructured does NOT mean complete lack of structure
  - Network has graph structure, e.g., scale-free
- Network has structure, but peers are free to join anywhere and objects can be stored anywhere
- So far we have seen unstructured networks

Structured networks/systems
- Based on addressing
- Network structure determines where peers belong in the network and where objects are stored
- How to build structured networks?

Overlay Network

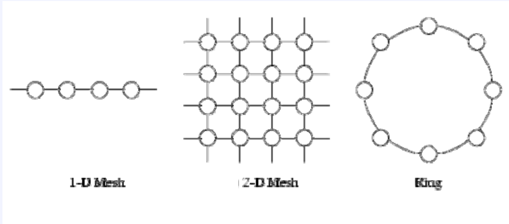
- Overlay network means a virtual network on top of the underlying IP network



- Can have various forms, including regular ones (rings, stars, ..)
  - Remember LANs? Logical token ring over physical token bus...

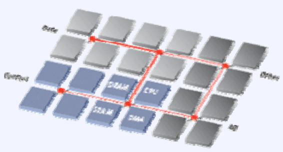
Interconnection networks

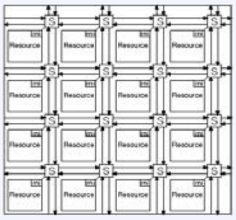
- Mainly used for parallel computers: regular network structures with certain advantageous properties
  - But also in other areas (e.g. for Networks On Chips, NoCs)



Source: http://www.gup.jku.at/thesis/diploma/christian_schaubschlaeger/

NoCs

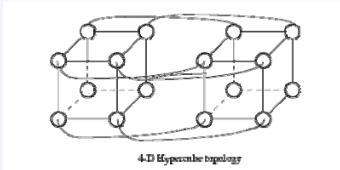




- Yes, people put (packet) networks on chips these days...
  - Reason: "design productivity gap" (space grows faster than our ability of using it)
- 2-dimensional grid topology common
  - Easy to build
  - Simple XY routing: from (3,3) to (2,4) means "one left, one down"
    - Note: if (X,Y) can be mapped to an object, finding it in a Grid is easy!
  - Strategies like hot potato routing can be applied (route around congestion instead of congestion control if traffic is small)

Hypercube

- Most prominent interconnection network for parallel systems
- K-dimensional hypercube: 2^k nodes, k connections each
- Scales well: maximum latency in k-dimensional hypercube is $\log_2 N$ ($N = 2^k$)
- Addressing: Gray code
  - K-bit addresses
  - Distance in hops = "bit-distance" (no. of unequal bits)
- Significant efforts were made for mapping parallel programs onto hypercubes



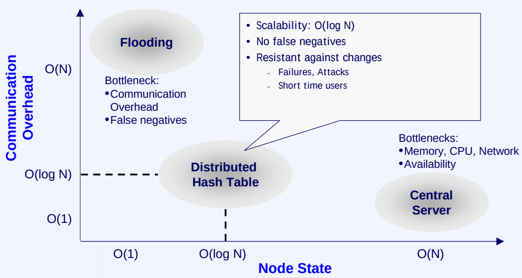
Source: http://www.gup.jku.at/thesis/diploma/christian_schaubschlaeger/

Regular network structures and P2P

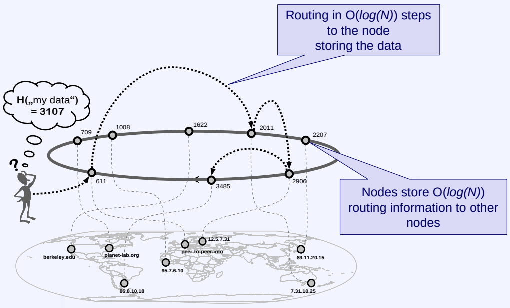
- Question: could structured interconnection networks be used to gain advantages in P2P systems?
  - E.g. create a Hypercube overlay
- Answer:
  - In principle yes, if we have a way to map node addresses to objects for finding them (key must identifiy objects)
  - But, no: interconnection networks typically designed for static topology
    - E.g. Hypercube based multiprocessor machines: hardware that can be bought
  - They must be adapted to the needs of P2P systems
- P2P file sharing systems need efficient functions for
  - Joining the overlay
  - Leaving the overlay
  - Storing objects and finding objects (related to addressing)
- Typical solution: Distributed Hash Table (DHT)

Distributed Hash Tables (DHTs)

- Communication overhead vs. node state



1.4 Distributed Indexing



Distributed Indexing

- Approach of distributed indexing schemes
  - Data and nodes are mapped into same address space
  - Intermediate nodes maintain routing information to target nodes
    - Efficient forwarding to „destination" (content – not location)
    - Definitive statement of existence of content
- Problems
  - Maintenance of routing information required
  - Fuzzy queries not primarily supported (e.g, wildcard searches)

System	Per Node State	Communi-cation Overhead	Fuzzy Queries	No false negatives	Robustness
Central Server	O(N)	O(1)	✓	✓	✗
Flooding Search	O(1)	O(N²)	✓	✗	✓
Distributed Hash Tables	O(log N)	O(log N)	✗	✓	✓

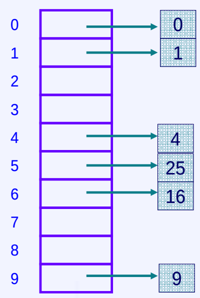
DHT: Motivation

- Why do we need DHTs?
- Searching in P2P networks is not efficient
  - Either centralized system with all its problems
  - Or distributed system with all its problems
  - Hybrid systems cannot guarantee discovery either
- Actual file transfer process in P2P network is scalable
  - File transfers directly between peers
- Searching does not scale in same way
- Original motivation for DHTs:
  More efficient searching and object location in P2P networks
- Put another way: Use addressing instead of searching

Recall: Hash Tables

- Hash tables are a well-known data structure
- Hash tables allow insertions, deletions, and finds in constant (average) time
- Hash table is a fixed-size array
  - Elements of array also called *hash buckets*
- *Hash function* maps keys to elements in the array
  - Note: mapping normally requires one precise key, no complex queries
- Properties of good hash functions:
  - Fast to compute
  - Good distribution of keys into hash table
  - Example: SHA-1 algorithm

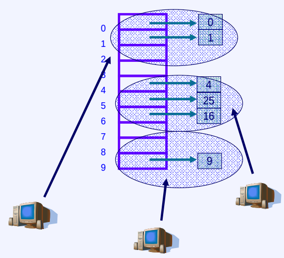
Hash Tables: Example



- Hash function:
  hash(x) = x mod 10
- Insert numbers 0, 1, 4, 9, 16, and 25
- Easy to find if a given key is present in the table

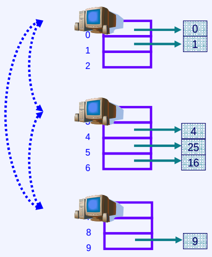
Distributed Hash Table: Idea

- Hash tables are fast for searching
- Idea: Distribute hash buckets (value ranges) to peers
- Result is Distributed Hash Table (DHT)
- Need efficient mechanism for finding which peer is responsible for which bucket and routing between them



DHT: Principle

- In a DHT, each node is responsible for one or more hash buckets
  - As nodes join and leave, the responsibilities change
- Nodes communicate among themselves to find the responsible node
  - Scalable communications make DHTs efficient
- DHTs support all the normal hash table operations

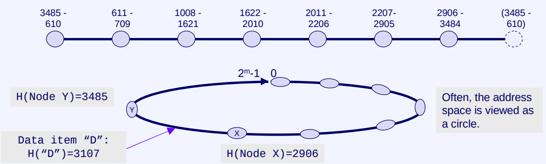


Summary of DHT Principles

- Hash buckets distributed over nodes
- Nodes form an overlay network
  - Route messages in overlay to find responsible node
- Routing scheme in the overlay network is the difference between different DHTs
- DHT behavior and usage:
  - Node knows "object" name and wants to find it
    - Unique and known object names assumed
  - Node routes a message in overlay to the responsible node
  - Responsible node replies with "object"
    - Semantics of "object" are application defined

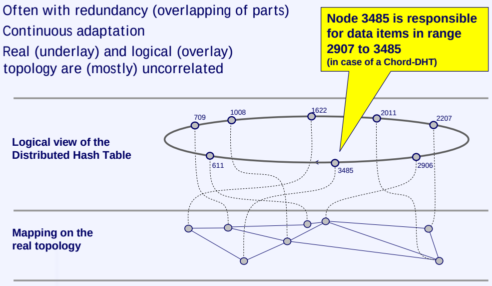
Addressing in Distributed Hash Tables

- Step 1: Mapping of content/nodes into linear space
  - Usually: 0, …, 2^m-1 >> number of objects to be stored
  - Mapping of data and nodes into an address space (with hash function)
    - E.g., Hash(*String*) mod 2^m: H(„*my data*") → 2313
  - Association of parts of address space to DHT nodes



Association of Address Space with Nodes

- Each node is responsible for a part of the value range
  - Often with redundancy (overlapping of parts)
  - Continuous adaptation
  - Real (underlay) and logical (overlay) topology are (mostly) uncorrelated

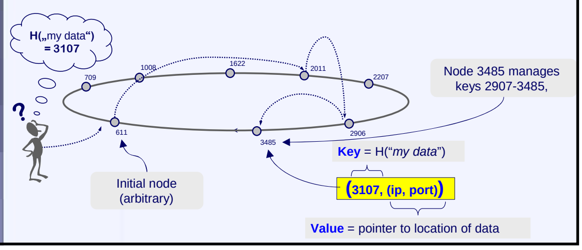


Step 2: Routing to a Data Item

- Step 2:
  Locating the data (content-based routing)
- Goal: Small and scalable effort
  - O(1) with centralized hash table
    - But:
      Management of a centralized hash table is very costly (server!)
  - Minimum overhead with distributed hash tables
    - O(log N): DHT hops to locate object
    - O(log N): number of keys and routing information per node (N = # nodes)

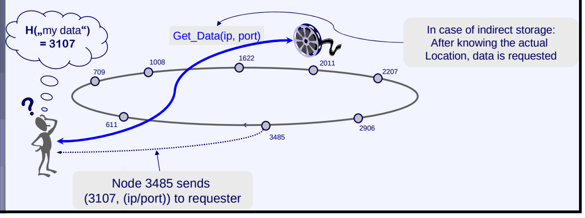
Step 2: Routing to a Data Item /2

- Routing to a K/V-pair
  - Start lookup at arbitrary node of DHT
  - Routing to requested data item (key)



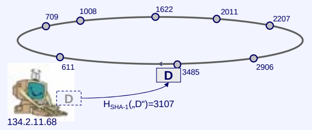
Step 2: Routing to a Data Item /3

- Getting the content
  - K/V-pair is delivered to requester
  - Requester analyzes K/V-tuple
    (and downloads data from actual location – in case of indirect storage)



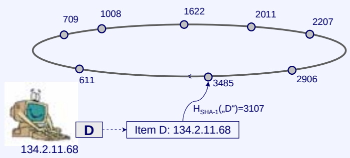
Data-ID Association - Direct Storage

- How is content stored on the nodes?
  - Example: H(“*my data*”) = 3107 is mapped into DHT address space
- Direct storage
  - Content is stored in responsible node for H(“*my data*”)
    → Inflexible for large content – o.k., if small amount data (<1KB)



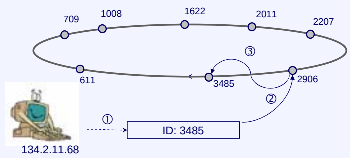
Data-ID Association - Indirect Storage

- Indirect storage
  - Nodes in a DHT store tuples like (key,value)
    - Key = Hash(„*my data*”) → 2313
    - Value is often real storage address of content:
      (IP, Port) = (134.2.11.140, 4711)
  - More flexible, but one step more to reach content



Node Arrival

- Joining of a new node
  1. Calculation of node ID
  2. New node contacts DHT via arbitrary node
  3. Assignment of a particular hash range
  4. Copying of K/V-pairs of hash range (usually with redundancy)
  5. Binding into routing environment

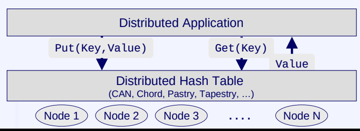


Node Failure / Departure

- Failure of a node
  - Use of redundant K/V pairs (if a node fails)
  - Use of redundant / alternative routing paths
  - Key-value usually still retrievable if at least one copy remains
- Departure of a node
  - Partitioning of hash range to neighbor nodes
  - Copying of K/V pairs to corresponding nodes
  - Unbinding from routing environment

DHT Interfaces

- Generic interface of distributed hash tables
  - Provisioning of information
    - Publish(key,value)
  - Requesting of information (search for content)
    - Lookup(key)
  - Reply
    - Value
- DHT approaches are interchangeable (with respect to interface)
  - Generic DHTs can be built – cf. OpenDHT (http://ww.opendht.org)



Comparison: DHT vs. DNS

- Comparison DHT vs. DNS
  - Traditional name services follow fixed mapping
    - DNS maps a logical node name to an IP address
  - DHTs offer flat / generic mapping of addresses
    - Not bound to particular applications or services
    - „*value*“ in *(key, value)* may be an address, a document or other data

Domain Name System
- Mapping: Symbolic name → IP address
- Is built on a hierarchical structure with root servers
- Names refer to administrative domains
- Specialized to search for computer names and services

Distributed Hash Table
- Mapping: key → value can easily realize DNS
- Does not need a special server
- Does not require special name space
- Can find data that are independently located of computers

Summary: Properties of DHTs

- Use of routing information for efficient search for content
- Keys are evenly distributed across nodes of DHT
  - No bottlenecks
  - A continuous increase in number of stored keys is admissible
  - Failure of nodes can be tolerated
  - Survival of attacks possible
- Self-organizing system
- Simple and efficient realization
- Supporting a wide spectrum of applications
  - Flat (hash) key without semantic meaning
  - Value depends on application
- Fuzzy queries inherently not supported

DHT Examples, Part 1: CAN and Chord

Reminder: all DHT systems provide storage/retrieval of K/V pairs – they differ in the overlay routing scheme.

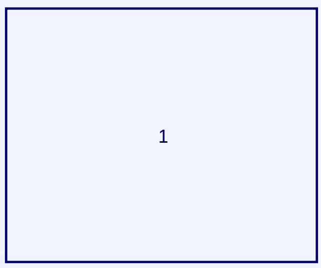
CAN: Content Addressable Network

- Developed at UC Berkeley
  - Originally published in 2001 at Sigcomm conference
- CAN's overlay routing easy to understand
  - Paper concentrates more on performance evaluation
  - Also discussion on how to improve performance by tweaking
- Project did not have much of a follow-up
  - Only overlay was developed, no bigger ensuing efforts

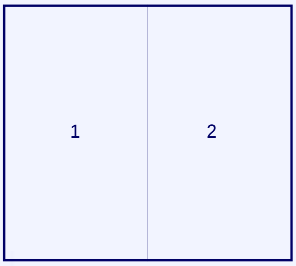
CAN: Basics

- CAN based on D-dimensional Cartesian coordinate space
  - Our examples: D = 2
  - One hash function for each dimension
- Entire space is partitioned amongst all the nodes
  - Each node owns a zone in the overall space
- Abstractions provided by CAN:
  - Can store data at points in the space
  - Can route from one point to another
- Point = Node that owns the zone in which the point (coordinates) is located
- Order in which nodes join is important

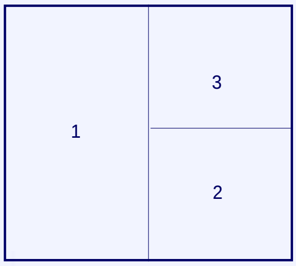
CAN: Partitioning



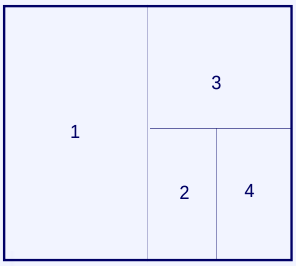
CAN: Partitioning



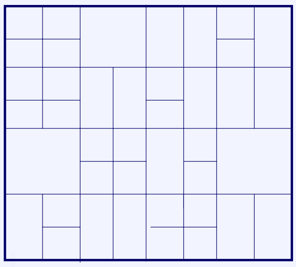
CAN: Partitioning



CAN: Partitioning

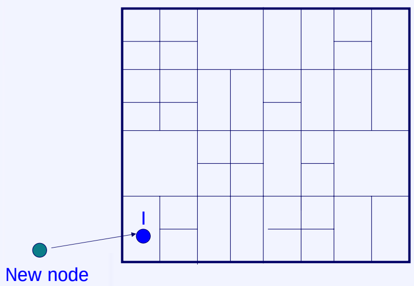


CAN: Partitioning



- CAN forms a d-dimensional torus

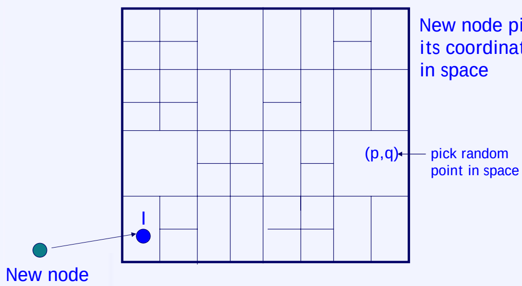
CAN: Node Insertion



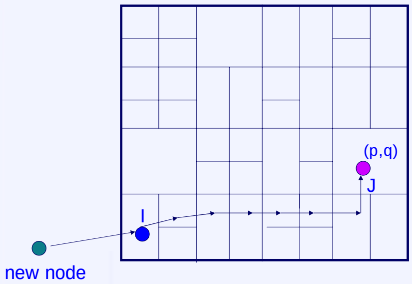
Discover some node “I” already in CAN

CAN: Node Insertion

New node picks its coordinates in space

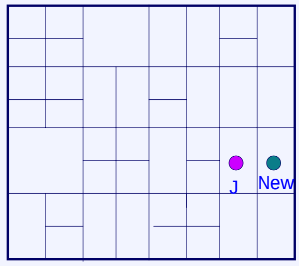


CAN: Node Insertion



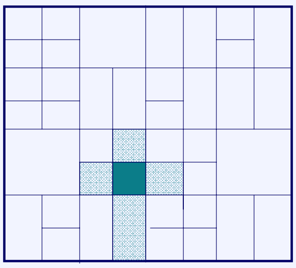
I routes to (p,q), and discovers that node J owns (p,q)

CAN: Node Insertion

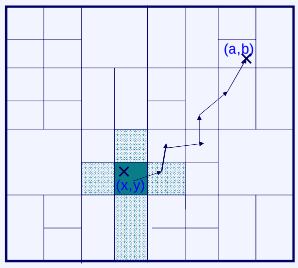


Split J's zone in half. New owns one half

CAN: Routing Table



CAN: Routing

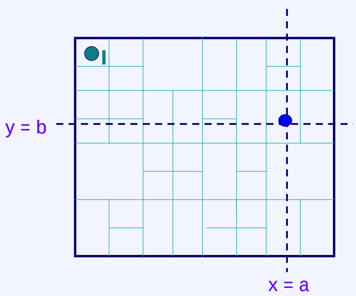


CAN: Storing Values

node I::insert(K,V)

$a = h_x(K)$

$b = h_y(K)$

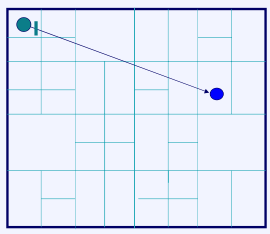


CAN: Storing Values

node I::insert(K,V)

(1) $a = h_x(K)$
$b = h_y(K)$

(2) route(K,V) -> (a,b)



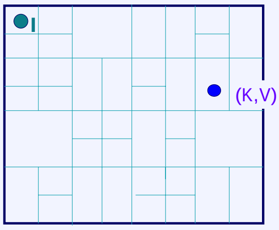
CAN: Storing Values

node I::insert(K,V)

(1) $a = h_x(K)$
$b = h_y(K)$

(2) route(K,V) -> (a,b)

(3) (a,b) stores (K,V)

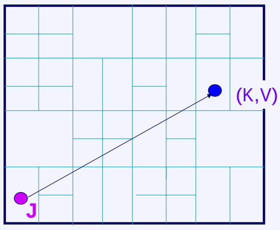


CAN: Retrieving Values

node J::retrieve(K)

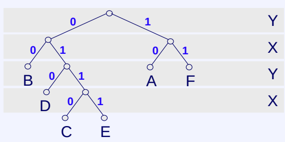
(1) $a = h_x(K)$
$b = h_y(K)$

(2) route "retrieve(K)" to (a,b)



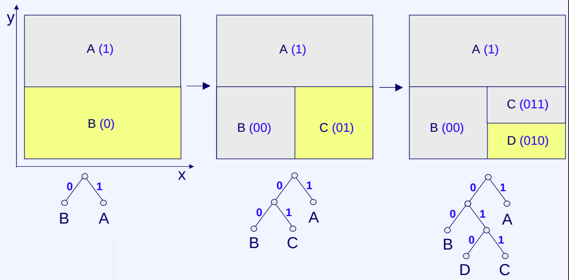
Node Removal

- When a CAN node quits (or disappears), a neighbor must take over
- CAN is based on squares
  - Strange figures could appear… defragmentation needed
  - Tree representation facilitates understanding the process
- Partitioning is performed according to some rules:
  - Strict sequencing of value range partitioning
  - According to order D
    e.g. x, y, z, x, y, z, … if D=3
- Partitioning tree reflects „history" of partitioning process



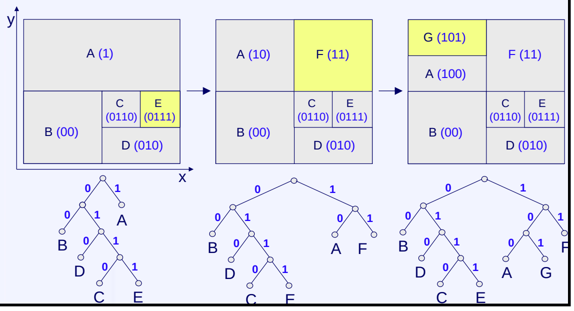
Structure of a CAN - Example

Insertion of nodes A,…, D



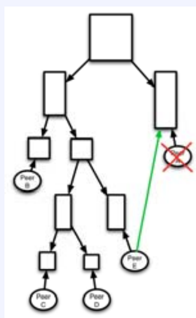
Structure of a CAN - Example /2

Insertion of nodes E, F, G



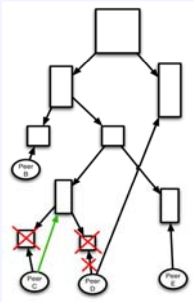
Node Removal

- Region and managed key/value pairs are transferred to neighbor
  - Ideal case: regions can be merged according to prior partitioning → tree
  - Otherwise: neighbor with smallest number of key gets both regions to manage (no merging!)
- Exit of a node: regular transfer procedure
- Failure of a node: TAKEOVER procedure
  - Non-appearance of periodic update information at neighbors
  - Neighbors initiate timer in proportion to size of region
  - (Smallest) neighbor signals TAKEOVER to other neighbors and takes region over



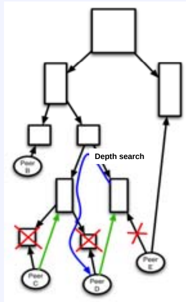
Defragmentation

- Zones are repeatedly reassigned in order to reduce defragmentation
- For every peer which owns at least two zones
  - delete smallest zone
  - find alternate peer which should take over the region
- Simple case:
  Neighbor zone is not split
  - Both peers are leafs in the CAN tree
  - Assign zone to neighbor peer
  - Eliminates the need to split zone above neighbor peer (only one peer left in charge of it)



Defragmentation /2

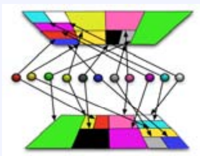
- Difficult case: neighbor zone is split
  - Carry out depth search in neighbor tree until two neighbor leaves are found
  - Assign zones of both leafs to a peer
  - Choose the other peer as replacement
- In the example:
  - Peer E's left assignment removed
  - No other peer directly underneath this region in the tree ⇒ depth first search finds Peer D
  - Reassignment eliminates partitioning of region above C (only one peer left in charge of it)



CAN Performance improvements

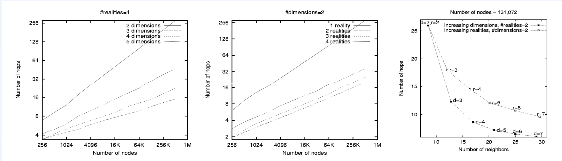
Note: Some of them would also work for other DHTs!

- More dimensions
  - Increases the number of neighbors – decreases the index structure
  - More path selection possibilities
    - Can apply other routing metrics (support for locality):
      - Delay measurement between neighbors
      - Choice of neighbors with the best delay
- More concurrent coordinate systems (realities)
  - More concurrent hash tables – nodes are members of r hash tables
  - (K,V)-tuple is saved within r hash tables
  - Mapping of keys onto r different coordinate systems via different hash functions
  - All "realities" are checked in each routing step



More Dimensions ↔ More Realities

- More dimensions
  - More neighbors
  - More routing possibilities
  - More state information O(2D)
- More coordinate systems (r)
  - r possibilities for routing
  - State information O(r·D)
  - r-time redundancy

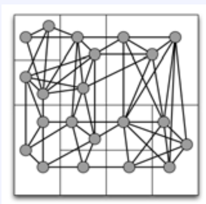


Increase of dimensions | *Increase of realities* | *Comparison*

Conclusion: more dimensions lead to shorter paths within the overlay
(...but more coordinate systems increase the redundancy)

Other improvements for CAN

- Overlap regions
  - k nodes jointly manage one zone (MAXPEERS nodes per zone)
  - Every peer knows neighbors + all peers in its zone
  - Results:
    - more redundancy
    - Faster routing paths because of smaller number of zones (O(MAXPEERS) path length reduction)
    - Multiple possible paths enable locality support



- Equal (uniform) partition of regions
  - Target zone tests during join procedure, whether there are "large" neighbors in proximity being more qualified for partitioning+
- Multiple hashing
  - Objects stored multiple times with different keys
  - Increases robustness, reduces distances (lookup only needed to closest copy, no. of hops is indirectly proportional to no. of copies)

CAN improvements and Complexity

- Overlay-Underlay mapping: Use well-known landmark servers
  - Nodes join CAN in different areas, depending on distance to landmarks
    - Pick points "near" landmark
  - Idea: Geographically close nodes see same landmarks
- *CAN Complexity:*
  (D = dimensions (assumed to be 2 up to now), N = number of peers)
  - **State information per node : O(D)**
    - Need to neighbors per coordinate axis
    - Independent of N
  - **Routing: O(D $N^{1/D}$) hops within overlay**
    - Effort = O(log N), with D = log N
      - Problem: N has to be known before
    - For routing: multiple dimensions are better
    - But: multiple realities improve availability and fault tolerance

Chord

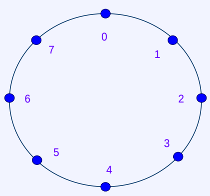
- Chord was developed at MIT
  - Originally published in 2001 at Sigcomm conference (like CAN!)
- Chord's overlay routing principle quite easy to understand
  - Paper has mathematical proofs of correctness and performance
- Many projects at MIT around Chord
  - CFS storage system
  - Ivy storage system
  - Plus many others…

Chord: Basics

- Chord uses SHA-1 hash function
  - Results in a 160-bit object/node identifier
  - Same hash function for objects and nodes
- Node ID hashed from IP address, object ID hashed from object name
  - Object names somehow assumed to be known by everyone
- SHA-1 gives a 160-bit identifier space
- Organized in a ring which wraps around (i.e. modulo arithmetic)
  - Nodes keep track of predecessor and successor
  - Node responsible for objects between its predecessor and itself
  - Overlay is often called "Chord ring" or "Chord circle"

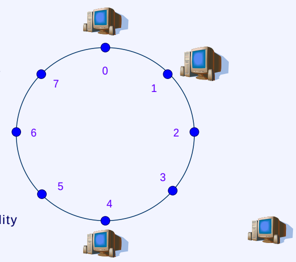
Joining: Step-By-Step Example

- Setup: Existing network with nodes on 0, 1 and 4
- Note: Protocol messages simply examples
- Many different ways to implement Chord
  - Here only conceptual example
  - Covers all important aspects



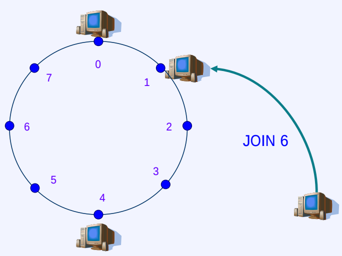
Joining: Step-By-Step Example: Start

- New node wants to join
- Hash of the new node: 6
- Known node in network: Node1
- Contact Node1
  - Include own hash
- Remember, status of the ring:
  - Each node knows predecessor
  - Each node knows successor
  - Each node knows its responsibility
    - Node 0: data for]4..0]
    - Node 1: data for]0..1]
    - Node 4: data for]1..4]

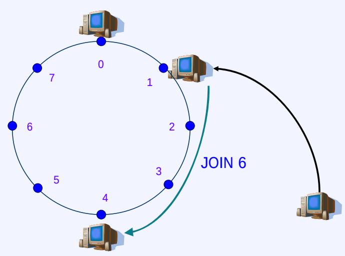


Joining: Step-By-Step Example: Contact known node

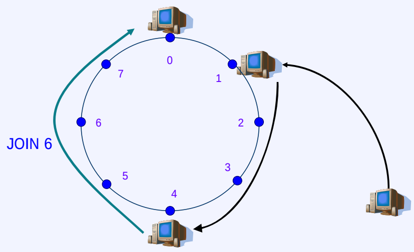
- Arrows indicate open connections
- Example assumes connections are kept open, i.e., messages processed recursively
- Iterative processing is also possible



Joining: Step-By-Step Example: Join gets routed along the network

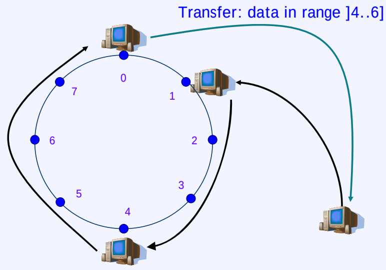


Joining: Step-By-Step Example: Successor of a new node found



Joining: Step-By-Step Example: Join successful + transfer

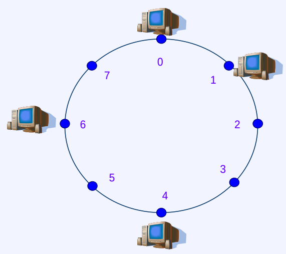
- Joining is successful
- Old responsible node transfers data that should be in new node
- New node informs node 4 about new successor (not shown)



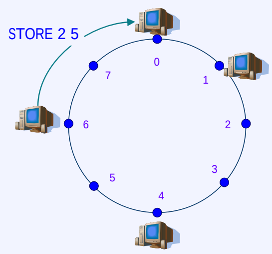
Note: transferring can also happen later

Storing a Value

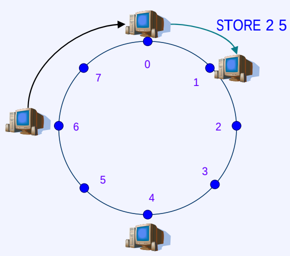
- Node 6 wants to store object with name “Foo” and value 5
- hash(Foo) = 2



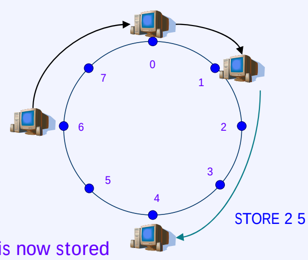
Storing a Value



Storing a Value



Storing a Value

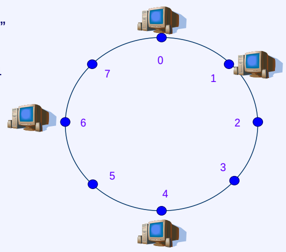


Value is now stored in node 4.

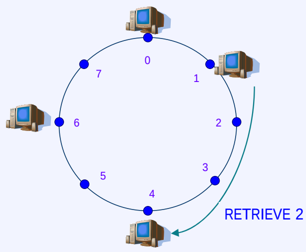
Retrieving a Value

- Node 1 wants to get object with name "Foo"
- hash(Foo) = 2

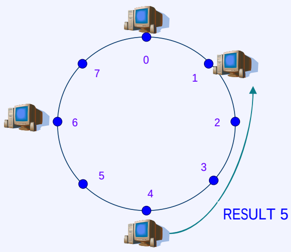
→ Foo is stored on node 4



Retrieving a Value



Retrieving a Value



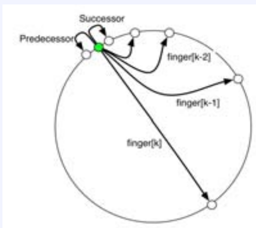
Chord: Scalable Routing

- Routing happens by passing message to successor
- What happens when there are 1 million nodes?
  - On average, need to route 1/2-way across the ring
  - In other words, 0.5 million hops! Complexity *O(n)*
- How to make routing scalable?
- Answer: Finger tables
  - Keep track of more nodes than just successor and predecessor
  - Allow for faster routing by jumping long way across the ring
  - Routing scales well, but needs more state information
- Finger tables not needed for correctness, only performance improvement

Chord: Finger Tables

- In *m*-bit identifier space, node has up to *m* fingers
  - Fingers are stored in the finger table
- Row *i* in finger table at node *n* contains first node *s* that succeeds *n* by at least 2^{i-1} on the ring
  - In other words:

    $finger[i] = successor(n + 2^{i-1})$
  - First finger is immediate successor
- Distance to *finger[i]* is at least 2^{i-1}
- Finger intervals increase with distance from node n
  - If close, short hops; If far, long hops

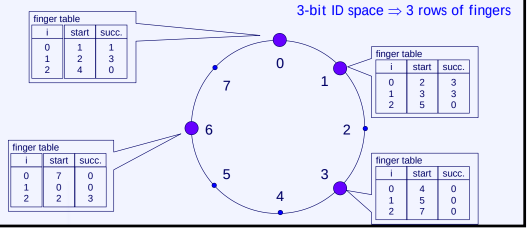


Chord: Scalable Routing

Two key properties:

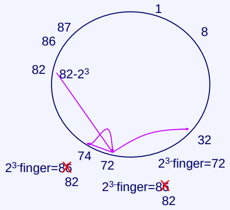
- Each node only stores information about a small number of nodes
- Cannot in general determine the successor of an arbitrary ID

3-bit ID space ⇒ 3 rows of fingers



Joining: update of finger pointers

- Node 82 joins
- Finger entries to node 86 may point now to new node 82
- Candidates for updates:
  - Nodes (counter-clockwise), whose 2^i-th finger entry have to point to N
- Check predecessors t_i of keys $(s - 2^i)$
  - Route to $s - 2^i$
- If t's 2^i-finger points to a node beyond N:
  - change t's 2^i-finger to N
  - Set t to predecessor of t and repeat
- ELSE continue with 2^{i+1}
- $O(\log^2 N)$ for looking up and updating finger entries

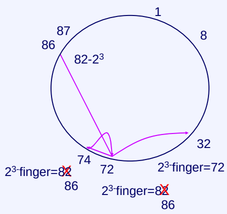


Node Departure

- Deliberate node departure
  - clean shutdown instead of failure
- For simplicity: treat as failure
  - system already failure tolerant
  - soft state: automatic state restoration
  - state is lost briefly
  - invalid finger table entries: reduced routing efficiency
- For efficiency: handle explicitly
  - notification by departing node to
    - successor, predecessor, nodes at finger distances
  - copy (key, value) pairs before shutdown

Node Departure /2

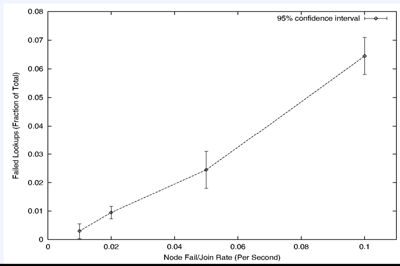
- Similar procedure as with node join
  - Update of fingers pointing to departing node similar to node join procedure



Example: i=3

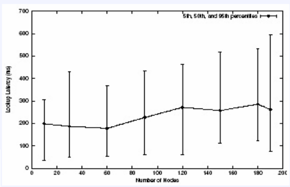
Chord: Performance

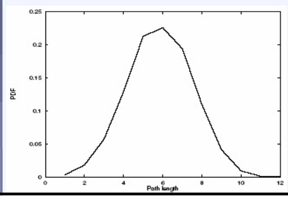
- Impact of node failures on lookup failure rate
  - lookup failure rate roughly equivalent to node failure rate



Chord: Performance /2

Moderate impact of number of nodes on lookup latency





Consistent average path length

Chord: Performance

- Search performance of "pure" Chord *O(n)*
  - Number of nodes is *n*
- With finger tables, need *O(log n)* hops to find the correct node
  - Fingers separated by at least 2^{i-1}
  - With high probability, distance to target halves at each step
  - In beginning, distance is at most 2^m
  - Hence, we need at most *m* hops
- For state information, "pure" Chord has only successor and predecessor, *O(1)* state
- For finger tables, need *m* entries
  - Actually, only *O(log n)* are distinct
  - Proof is in the paper
- Management actions (join/leave/fail): $O(\log^2 n)$

References / acknowledgments

- Slides from:
  - Jussi Kangasharju
  - Christian Schindelhauer
  - Klaus Wehrle