A Distributed Hash Table for Computational Grids

Chris Riley, Christian Scheideler
Department of Computer Science
Johns Hopkins University
{chrisr,scheideler}@cs.jhu.edu

Abstract

*In this paper, we present and analyze a distributed hash table-based supervised peer-to-peer system that allows an even distribution of and efficient lookup for objects (*e.g. *data or tasks) stored in the system. A* supervised peer-to-peer system *is a system that is formed by a supervisor but in which all other activities can be performed on a peer-to-peer basis without involving the supervisor. Our system has average constant degree and can distribute objects evenly among the peers up to a constant factor in expectation. The supervised peer-to-peer approach makes the system particularly useful for computational grids. As an example, we discuss the use of our structure for recursively defined algorithms such as dynamic programming and distributed tree searches, and practical problems such as web crawling; our structure distributes tasks randomly and prevents repeated computations to optimize parallel efficiency.*

1 Introduction

The ultimate challenge of computational grids is to use loosely connected unreliable computers as a source of computational power for arbitrary sequential or concurrent computations, and to allow that power to be sold and used on demand like modern day electrical systems. Networks of computers suitable for more restricted problem domains are in active use. Some of these networks address problems based on brute force checking of a large input range, or problems involving management of extremely large data sets.

As a structure to support these and future computational grid applications, we present a supervised distributed hash table supporting efficient object insertion and lookup along with balanced allocation; our objects are treated in the same way as data items in other distributed hash tables. Because of our supervised approach, the network can achieve a constant factor expected load balance with average constant node degree, something not achieved yet by decentralized distributed hash tables. The supervised approach makes this structure particularly suitable for computational grid applications by providing a reliable and trusted location for node authentication and task assignment and combining results. Since our system can be used to spawn and evenly distribute further subtasks by the peers without involving the supervisor, and since the supervisor only needs to know a constant-sized portion of the network, the supervisor can be a low-powered node. The supervisor also serves as a reliable anchor for code execution rollback, which is important for failure recovery mechanisms such as those used in the Time Warp system[10, 18]. We discuss the use of our structure to support distributed divide-and-conquer algorithms, including dynamic programming, tree searches, and web crawling.

1.1 Related Work

Computational grids. Existing networks for distributed computation include SETI@home[21], Folding@home[5], and distributed.net[4]. These are systems solving problems which run relatively small pieces of code on very many inputs; such problems are easily divided into subtasks corresponding to subranges of the input, and the entire code can be sent to each client. They use a star topology, with a central server connecting directly to each client for all communications. No communication is needed between the clients, so code can be executed in parallel with very little overhead (there is only the cost of sending the code to the participants). While these systems are adequate for their tasks, they represent only a fraction of the potential of general purpose computational grids. Furthermore, a star topology places a high demand on the central server since it must be involved in all computation assignments and must keep track of all nodes participating in the system, limiting scalability and increasing expense. The OptimalGrid project at IBM[15] extends the reach of these systems by allowing communication between the subtasks, but as it still uses a star topology it suffers from the same scalability concerns.

Recently the CERN project [3] has attracted significant media attention. CERN's network, known as the Grid, combines computing power from research centers in 12 countries. This power is used to examine terabytes of data generated by a new particle accelerator at CERN called the Large Hadron Collider (LHC). Unlike SETI@home and similar programs, their focus is on managing large amounts of data rather than using highly dynamic networks. To the best of our knowledge their work does not address topological considerations in peer-to-peer computational grids.

Extensive research on computational grids is being conducted by the Globus Alliance [8]. Their work includes services such as security tools and a data management structure known as the Data Grid, designed for "coordinated resource sharing and problem solving"[6]. The Data Grid offers services for abstracting out the storage system layers at each network node and services for maintaining and organizing metadata about the stored data, as well as services for replica management, along with applications for data exchange within the Globus framework; they also do not appear to consider topological designs in their research. Their system is best suited for data-intensive applications such as the CERN project.

Supervised distributed networks. The authors have some ongoing work on using supervised peer-to-peer networks for different problems, including reliable broadcasting [19] and distributed monitoring systems [1]. The design in these works is tree-based and consequently very different from the design presented in this paper.

Pandurangan, Raghavan, and Upfal [16] present a supervised peer-to-peer network where connections are made by a low powered *host server*; their network has constant degree and low diameter, and remains connected with high probability under a stochastic model of node arrivals and departures, though insertion work is $\Omega(\log n)$ in some cases. Their network does not seem to be capable of supporting object lookup or message passing, though it could be used for broadcasting.

Peer-to-peer systems. Research in peer-to-peer systems has been active for several years now. Though the origins of interest in peer-to-peer systems are in popular file sharing networks like Gnutella [9] and Napster [14], the academic community quickly adopted the model for research purposes. Distributed hash tables (DHT) are frequently used for peer-to-peer token location, and have been implemented with different structures by multiple researchers; for some classic examples, see [17, 20, 22].

We are aware of three papers [7, 11, 13] proposing peer-to-peer systems based on the DeBruijn graph; our approach uses a DeBruijn topology as well, though our construction and maintenance are quite different. In particular, these structures are probabilistic in their organization, while ours is deterministic in all but the placement of objects.

Distributed hash tables built on peer-to-peer networks could be considered for distributed computation as well. On top of a DHT one could run a system like Time Warp to ensure successful code execution. The Time Warp operating system [10, 18] was designed to support parallel execution of discrete event simulations, which have been and will continue to be among the most expensive computational tasks. The Time Warp system is based on a distributed process rollback mechanism, and involves significant changes to traditional operating systems designs. This mechanism is used to ensure the successful completion of a distributed program even when some pieces fail; rollback processes shift the state of the running program back to a valid state to resume normal computation. Decentralized systems like pure peer-to-peer systems are not very useful for this, since they lack a single reliable point to ensure that code rollback can proceed successfully in all cases.

1.2 Our results

We present and analyze a DHT-based supervised peer-to-peer system, **SPON-DHT**, that allows an even distribution of and efficient lookup for objects stored in the system. A *supervised peer-to-peer system* is a system that is formed by a supervisor but in which all other activities can be performed on a peer-to-peer level without involving the supervisor. The supervisor only needs to be involved with node join and leave operations. The network supports efficient node-to-node communication and object lookup. Object lookup trivially supports object insertion and deletion on request from the object's lookup destination. While these results assume graceful node departures (nodes which do not depart without advance notice), the network functions as well when this assumption is removed, though some $O(1)$ costs increase to $O(\log n)$. The network performance can be summarized as follows:

1. Join operations require $O(1)$ work and time and leave operations require $O(1)$ average ($O(\log n)$ worst-case) work and time, plus the cost of object movements.
2. For all sets of nodes V the expected relative load $\mathbf{E}[\lambda_v]$ $\forall\, v \in V$ is $\Theta\left(\frac{1}{n}\right)$.
3. Object lookup requires $O(\log n)$ work and time.
4. All nodes have constant out-degree and only need constant storage to manage the object space.
5. The supervisor's network knowledge, measured in out-degree and storage, is constant.

The relative load λ_v for any node v in the current set of nodes in the system V is a random variable corresponding to the fraction of objects in the system mapped to node v. The expectation of λ_v, $\mathbf{E}[\lambda_v]$, in a distributed hash table on the

$[0, 1)$ real line is equal to the size of the subrange assigned to node v. In our system the size of the subrange mapped to v is guaranteed to be $\Theta\left(\frac{1}{n}\right)$ for all nodes v and all sets of current nodes V. In most previous distributed hash tables, for all sets of nodes V, $max_{v \in V}(\mathbf{E}[\lambda_v]) = \Theta\left(\frac{\log n}{n}\right)$ with high probability (based on the random hash function h used to select points on the identifier circle for all $v \in V$). Some papers use virtual nodes to reduce this to a constant, but node degrees and join/leave work increase by a $O(\log n)$ multiplicative factor to achieve this. Our deterministic construction allows us to achieve a low expected relative load without increasing average node degree.

1.3 Network applications

SPON-DHT can be used to support a new class of computational grid applications such as distributed divide-and-conquer programs. Our structure allows these operations to be executed without repeating subtasks. Since a star topology is not used, the effort exerted by the supervisor can be reduced significantly by allowing nodes to spawn subtasks and combine the results themselves; this allows inexpensive supervisors to manage large networks. Combined with a system like Time Warp, applications can be executed correctly in unreliable networks. Furthermore, the supervisor can keep known malicious nodes from entering the system, and can authenticate nodes so that malicious behavior can be detected by other nodes and reported.

Since our network is a distributed hash table, it is also suitable for storage systems or any other application requiring an object management infrastructure.

2 Network overview

2.1 Interface

We provide a network which offers object management in a dynamic network. Our objects could be either units of storage data or computational tasks or some combination of the two; we refer to these units as objects in the spirit of object-oriented programming. The network supports three fundamental operations: *join()* and *leave()* calls for nodes, and *lookup(t)* calls for an object t. This interface can be used by applications to perform distributed computation or storage. The basic functionality of these operations is given below.

1. A join operation requires the network to include the new node so it can communicate with all nodes and preserve all existing communications. Also, some objects must be moved to the new node to preserve a good load balance.
2. A leave operation requires the network to extract the departing node while preserving the ability to communicate between all other node pairs. Also, the node's objects must be moved to other nodes while preserving the load balance.
3. A lookup operation involves locating the physical node currently responsible for the object t, whether or not the object exists.

Insertions and deletions for an object t can be accomplished by an application by performing a lookup and then requesting that the node responsible for t insert or delete t.

For the sake of simplicity we assume for most of this paper that nodes depart *gracefully*. A graceful departure is one in which a node requests permission and waits for a confirmation before departing; this allows the network to easily accomodate the node's departure. We will remove this assumption by extending our system later.

2.2 Supervisor

The network includes a low powered reliable supervisor, which handles node insertions and deletions; the supervisor is not involved in direct node-to-node communication. The use of a supervisor allows the network to achieve a greater level of security and trustworthiness than can be acquired by fully decentralized systems. Since messages are passed without the supervisor's interference or knowledge, the nodes in the network are capable of unrestricted concurrent communication. However, nodes can be identified and authenticated when joining, and through appropriate message signatures can be held responsible for their actions by other nodes, so that improper activity can be detected and reported to the supervisor.

We outline the following guidelines for a scalable supervised peer-to-peer system:

1. The network should possess a single supervisor node which can be assumed to be reliable.
2. The supervisor should not be involved in any node-to-node communication.
3. The supervisor node should at no point in time need more storage or degree than $O(\log n)$.
4. Non-supervisor nodes must have low degree and storage, at most $O(\log n)$.

Our system exceeds these specifications by requiring only constant degree. The extended version of our system designed to support ungraceful node departures matches these requirements.

While the supervisor is a single point of failure, this is easily corrected: Replace the supervisor with a multicast group of $O(\log n)$ (or $O(1)$) nodes, and let all messages from other network nodes be sent to the multicast address so that all group nodes receive all messages. At any time, a single group node is the active supervisor, and responds to messages; the other nodes process the messages internally but do not respond. If the active supervisor fails a new active supervisor is elected through any process. If a single group node survives, the network will continue to function normally. Using standard models, if each node fails with constant probability, then using a group of size $O(\log n)$ implies that with high probability at least one group member survives. For clarity, in the remainder of this paper we will assume that only a single supervisor is used.

2.3 Dynamic DeBruijn graph

Non-supervisor network nodes are organized into a dynamic version of the DeBruijn graph. The static DeBruijn graph includes two directed edges out from each node, connecting node i to nodes $2i$ and $2i+1$ modulo n when there are $n = 2^b$ nodes for some integer b. This is not easily adapted to dynamic network conditions. When the graph grows, the modulus will change, causing some edges to become invalid because their endpoints had been beyond the previous modulus 2^b but are below the new modulus 2^{b+1}. Similarly, if a static DeBruijn graph were to shrink by removing nodes and their adjacent edges, the graph would not remain properly connected since some edges based on a lower modulus would not exist.

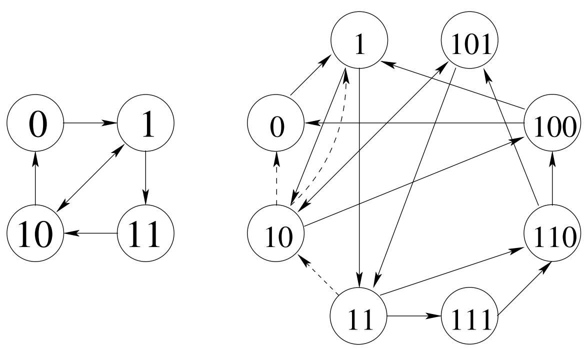

Figure 1. The extension of a 2D DeBruijn graph to 3D, with older edges indicated

We suggest a dynamic extension of the DeBruijn graph: leave all old edges in the graph during expansion. This cannot raise a node's outdegree beyond 4 (2 based on the insertion time modulus and 2 without modulus), though some nodes will have an indegree of $\lceil \log n \rceil$. The node with a binary address of all 0's, for example, will have an incoming edge from each node with a binary address containing a single 1. Should it be desired, it is possible to extend the structure to allow removal and reinsertion of the old edges, though some applications of the network would need modification.

In a graph with n nodes, nodes are labelled with distinct binary strings of at most $\lceil \log n \rceil$ bits, where only the strings 0 through $n-1$ are in use. The structure supports the following:

Invariant 2.1 *Every node with address β whose highest-order 1-bit is bit j has an outgoing edge to every existing node with address $\in \{2\beta, 2\beta+1, 2\beta \textit{ mod } 2^{j+1}, 2\beta+1 \textit{ mod } 2^{j+1}\}$.*

In addition, we maintain:

Invariant 2.2 *Every node with address β has an edge to the nodes with addresses $\beta - 1$ and $\beta + 1$, if they exist. These are the* predecessor *and* successor *nodes.*

Since this is an overlay network, having an edge to a node is equivalent to possessing its real network address (such as an IP address), and sometimes also involves periodic link checking to make sure the node is available; this requires $O(1)$ space per edge. This applies to both incoming and outgoing edges, because a node's incoming neighbors need to be determined before it can be replaced.

2.4 Permanent addressing

SPON-DHT is an overlay network, so addresses assigned to nodes are virtual addresses only, and keeping an edge to a node involves storing the actual IP address or some other contact information to allow direct communication; knowing the virtual address is not sufficient for communication in the underlying network. As a consequence, virtual addresses can be arbitrarily assigned and readily exchanged as long as the functionality of the address is also exchanged.

We will perform deletion by replacement, so that at any given time the only addresses in use in a network of n nodes are 0 through $n-1$. It is worth noting that, because of this, once a node is assigned an address, its address does not need to change, unless it leaves and rejoins or is used as a replacement. This is because shorter addresses can be implicitly padded with zeros, so that an assigned 2D address of 10, for example, is the same as the 3D address 010. In other words, nodes do not need to be contacted to extend their addresses when the graph grows, saving significant periodic update costs.

2.5 Node responsibilities

The supervisor. To maintain the topology during node joins and leaves, the supervisor must keep an edge to the following nodes:

- the 0-node (*i.e.* the node with address of all 0's),
- the end node (*i.e.* the node with the largest virtual address), and
- any nodes with edges to the end node.

Theorem 2.3 *The supervisor has constant-sized network knowledge, since it has $O(1)$ out-degree and requires no storage for object space management.*

Proof. As stated previously, we assume that each outgoing edge requires $O(1)$ storage. Since the end node has the largest address in the system, it cannot have more than one incoming edge, the non-modular edge from node $\lfloor \frac{n}{2} \rfloor$, since modular edges end at a node with at most the same value as the source (since the most significant zero in the address becomes more significant and the length is the same); therefore the degree of the supervisor is constant. The supervisor does not need to keep any information pertaining to the object lookup and load balancing, as this process is fully unsupervised. □

Non-supervisor nodes. Each non-supervisor node must keep pointers to the following nodes:

- its predecessor and successor, and
- its outgoing neighbors, and
- its incoming neighbors.

Non-supervisor nodes also keep knowledge of the ranges assigned to them for object allocation purposes; this will be discussed in more detail later. Each node also must store an address to contact the supervisor to be able to announce its departure.

Theorem 2.4 *Non-supervisor nodes have $O(1)$ out-degree and average in-degree $O(1)$ (worst-case $\log n + 3$).*

Proof. The constant out-degree follows from the specifications. Average in-degree is clearly $O(1)$ since all out-degrees are $O(1)$. The worst-case in-degree occurs at the 0-node, which has an incoming edge from every node with only one 1-bit. The number of bits is $\log N$ where $N = 2^{\lceil \log n \rceil}$, which is at most $\log n + 1$; including the predecessor and successor edges produces worst-case degree $\log n + 3$. □

Later we also bound the storage for object management.

3 Node operations

We assume that node join and leave requests are sent to the supervisor; join requests could be sent initially to any node and forwarded from there. Node join and leave requests are processed by the supervisor as node insertions and deletions. Each node insertion and deletion has two major components, the structural modifications and the object movements. We address the structural changes in this section and the object movements later.

3.1 Structural changes

Node deletion occurs through replacement, in the sense that the node in the network with the highest virtual address replaces any other internal node wishing to leave the system, so that the highest virtual address is removed, along with its adjacent edges, regardless of which physical node leaves. Replacement involves creating outgoing edges from the replacement node to match those of the departing node, changing incoming edges to the departing node to point to the replacement node, and moving tokens and other object management data to the replacement node. This approach has been used in other research, *e.g.* [2]. This process implies that an inserted node receives the largest virtual address in the network. We can thus give only the process for inserting or removing the last node in the network (the node with largest virtual address).

An inserted node is given two outgoing edges and one incoming edge. A node inserted into a n-node network will be given (virtual) address n. Node n is given outgoing edges to nodes $2n - 2^l$ and $2n + 1 - 2^l$ in order to satisfy Invariant 2.1 for node n, where $l = \lceil \log n \rceil$ is the number of bits in address n. Also, the node with address $p = \lfloor \frac{n}{2} \rfloor$ is given an outgoing edge to n; this is needed to satisfy Invariant 2.1 for node p.

Since the supervisor keeps an outgoing edge to the current end node x, it can connect n and x as successor and predecessor, satisfying Invariant 2.2. Furthermore, if x's outgoing neighbors are the nodes with addresses k and $k+1$, then n's outgoing neighbors have addresses $k+2$ and $k+3$, and can be located in a few steps by contacting node $k+1$ (which the supervisor can reach through x). This is true for all addresses except for $n = 2^l$ for any integer l, whose outgoing neighbors are 0 and 1. If n is odd, then its incoming neighbor is the same as x's (and is known by the supervisor). If not, its incoming neighbor is the successor of x's and can be easily reached. These properties follow from the DeBruijn topology.

Removing the end node requires the supervisor to add an edge to the new end node's incoming neighbor. Reversing the insertion, this is the predecessor of the removed node's incoming neighbor, except when the removed node's address is odd, when the incoming neighbor is the same.

Theorem 3.1 *Node insertion cost is $O(1)$ in time, number of messages, and local computations, plus any cost of object movements. Node removal cost is $O(1)$ for most nodes, and $O(\log n)$ in the worst case. Invariants 2.1 and 2.2 are preserved by the operations.*

Proof. Follows directly from our algorithms. □

3.2 Node-to-node communication

The algorithm for object lookup requires a primitive for one node to contact another node in the network by knowing its virtual address. The DeBruijn graph is readily equipped to handle such an operation; the static version supports $O(\log n)$-time message passing by shifting in the bits of the address of the destination one bit at a time. Since our version is dynamic, it needs to be shown that the graph can route between any two nodes even though some nodes may be missing. Consider sending a packet from node $x = (x_1 x_2 \dots x_d)$ to $y = (y_1 y_2 \dots y_d)$, where either x_1 or y_1 = 1 (if not the nodes cannot be expected to know d). If all nodes with d-bit addresses exist, then the path given by the default DeBruijn routing algorithm is $x \to x_{2,d} y_1 \to x_{3,d} y_{1,2} \to \dots \to x_d y_{1,d-1} \to y$, where $x_{i,j}$ is bits i through j of x; this path is of length d. If at any step some node $x_{i,d} y_{1,i-1}$ does not exist, then the node with address $0x_{i+1,d} y_{1,i-1}$ is used instead. This node has the same connections as the missing node, so that its use does not change the rest of the path; and because deletion by replacement is used, it must exist, since the only missing network nodes have highest bit 1.

Since our DeBruijn graph is dynamic, it may be difficult to route at lower levels of the structure, since the nodes may not know the total number of nodes in the network. But if the old edges are left, then the graph contains all nodes and edges of the static DeBruijn graph corresponding to the larger of the dimensions of the source and destination, and this subgraph can be used to pass the message, with cost still $O(\log n)$ in number of messages and rounds of communication.

When sending a message from each node in the network to a randomly chosen destination (the random routing problem), the static DeBruijn graph is well known to have a congestion of at most $O(\log n)$ with high probability on each node in the network. Missing nodes do not significantly affect this, as a node with address $0x$ has to support at most the congestion of $0x$ plus the congestion of $1x$ in a full DeBruijn graph, if node $1x$ is not present; therefore congestion of each node in a dynamic DeBruijn graph is at most twice the congestion of the node in a static DeBruijn graph containing the node. Furthermore, congestion caused by older dimensional edges can be avoided by first sending the message to a random maximal-dimension node and routed from there; the dilation and congestion are asymptotically unchanged. This leads to the following:

Theorem 3.2 *The dynamic DeBruijn graph can perform the random routing problem with congestion and dilation $O(\log n)$ with high probability.*

4 Object mapping

4.1 Virtual locations

Initially, we hash objects uniformly at random to the $[0, 1)$ real line using a repeatable hash function; this output is considered a virtual location for the object. The $[0, 1)$ line is then divided into ranges which are assigned to the nodes in the network, and each node is responsible for the objects with virtual locations within its assigned range(s). The hash function is distributed to all network nodes so that anyone can compute an object's virtual location to perform a lookup operation. In practice, any sufficiently precise and uniform hash function can be used.

When ranges are exchanged between nodes, the objects represented by the ranges must be moved as well. This process may vary from application to application. For example, in a storage system, an object represents data that must be sent across the network. In a computational grid, an object represents a computational task which may be in progress. The task can be restarted at the destination node, or some checkpoint information can be sent, or all current state for the process can be transferred.

4.2 Recursive placement algorithm

We initially give a recursive range placement algorithm. This algorithm specifies range movements during a single node join or leave operation; the iterative distributed lookup algorithm given below is derived from the placement produced by this algorithm. Our recursive algorithm is similar to the cut-and-paste algorithm of [2], designed for dynamic storage systems. Their algorithm is workload optimal for maintaining a perfect expected load across single disk joins and leaves, and supports $O(\log n)$-cost data item searching. A networked version would send load from all existing nodes to a joined node, and would distribute the load from a departing node across all remaining nodes, so that communication costs are $\Omega(n)$ per join or leave operation. We reduce this to a constant while preserving $O(\log n)$-cost lookup, though the load balance is allowed to worsen slightly.

Initially a single node is responsible for the entire $[0, 1)$ range. In our algorithm, each new node takes the higher half of the existing range of each of its two outgoing neighbors (or one if there is only one outgoing edge), and returns the ranges to its neighbors when it departs. As in [2], deletion by replacement ensures that any sequence of mixed node insertions and deletions produces a load distribution equivalent to a sequence of insertions alone. Since only outgoing adjacent nodes are contacted on node join and leaves, and since the dynamic DeBruijn graph has constant outgoing degree, the cost is $O(1)$ per insertion or deletion plus the cost of transferring load.

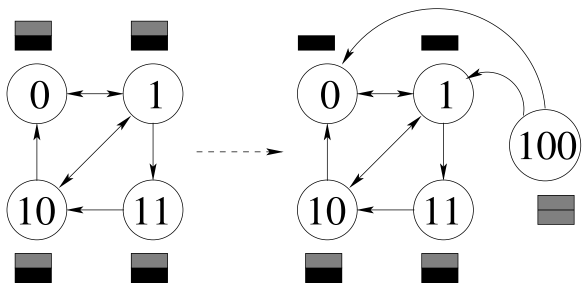

Figure 2. A node insertion with load movement

Since the outgoing neighbors of each sequential node are sequential, half the load is taken from each node before the process restarts (which occurs when the number of nodes passes 2^l for each integer l). At any point in time, if there are n nodes with $l = \lceil \log n \rceil$, then the $[0, 1)$ interval will be divided into subranges of size $\frac{1}{2^l}$ (ranges already split in this dimension) or $\frac{1}{2^{l-1}}$ (unsplit ranges). Each node will have either one split or unsplit subrange or two split subranges assigned to it.

Theorem 4.1 *For all nodes u, $\mathbf{E}[\lambda_u] = \Theta\left(\frac{1}{n}\right)$.*

Proof. When the network changes dimensions (when the number of nodes goes from 2^l to $2^l + 1$), consider the larger subranges to be two subranges of equal size, one of which will be taken away in this dimension while the other will stay. Then we can assume that all subranges are of equal size. Tasks are assigned uniformly at random to the $[0, 1)$ interval, so tasks are assigned uniformly at random to each subrange, so the expected relative load of each subrange is the same, $\frac{1}{2^{\lceil \log n \rceil}}$ if there are n nodes currently in the system. Since each node has either one or two assigned subranges the theorem holds. $\square$

Theorem 4.2 *The storage required by each node for its ranges is $O(1)$.*

Proof. It has been shown that each node has only one or two ranges, so the proof depends only on the space required to store a single range. This depends on the resolution required to store the range boundaries. For $l = \lceil \log n \rceil$, the resolution needed is l decimal bits, since each range boundary is $\frac{i}{2^l}$ for some integer i between 0 and 2^l. Since this is less information than the amount needed to distinguish nodes (such as an IP address), it is considered a single word. Since each range requires only two boundaries, the theorem holds. $\square$

4.3 Range history

We define a history for each range that gives the set of all nodes which held the range before it was taken away, going back to the node with address 0 which originally holds all ranges. Suppose that the range is held by some node with address v. The range history is a set of nodes $\{v_i\}$ for $i = 0$ to f, where $v_0 = v$ and $v_f = 0$. Note that $v_i < v_j \; \forall \; i > j$, and also that f, the number of steps the range has moved, varies depending on v, though it can be shown to be at most $O(\log^2 v)$. According to our definition of the load balancing algorithm, each v_i must have an outgoing edge to v_{i+1}; so $v_{i+1} = 2v_i$ or $2v_i + 1$ (with appropriate modulus). Intuitively, if v_i is the terminal location in this dimension, then $v_{i+1} = 2v_i$, and otherwise $v_{i+1} = 2v_i + 1$; this is because the a range stolen from $2v_i + 1$ will be taken by some other node before the dimension is full.

More formally, let $d(v_i)$ be the number of bits in the address v_i with all high-order 0's removed. The v_i's are calculated as follows:

- If v_0 holds a single range or if the range in question is the first of two held ranges, then $v_1 = 2v_0 \bmod 2^{d(v_0)}$.
- If the range in question is the second of two ranges held by v_0, then $v_1 = 2v_0 + 1 \bmod 2^{d(v_0)}$.
- If $d(v_i) = d(v_{i-1})$, then $v_{i+1} = 2v_i + 1 \bmod 2^{d(v_i)}$.
- If $d(v_i) \neq d(v_{i-1})$, then $v_{i+1} = 2v_i \bmod 2^{d(v_i)}$.

5 Distributed iterative object lookup

This section gives a distributed iterative algorithm for object lookup derived from the placement produced by the recursive algorithm. We begin by giving the object location in a full DeBruijn graph, where the number of nodes in the network is a power of 2, assuming that all nodes know the network is a full DeBruijn graph; this is a much simpler process, albeit unrealistic, that identifies the heart of the algorithm. Then we extend this to an arbitrary number of nodes.

5.1 In a known full graph

In a full graph, each node v holds a single range r_v. Knowing the graph is full, each node can compute $n = \frac{1}{|r_v|}$ correctly. In order to locate a token each node must be able to compute the node w holding the range r_w containing the token. We give an algorithm to compute the infinite sequence of nodes $\mathbf{v(t)}$ holding the point $t \in [0, 1]$. We also compute the starting points $\mathbf{s(t)}$, where $s_i(t)$ is the point in $[0, 1]$ corresponding to the beginning of the range held by $v_i(t)$.

- $v_0(t) = s_0(t) = 0$.
- If $t \geq s_i(t) + \frac{1}{2^{i+1}}$, $v_{i+1}(t) = v_i(t)/2 + 2^i$ and $s_{i+1}(t) = s_i(t) + \frac{1}{2^{i+1}}$.
- If $t < s_i(t) + \frac{1}{2^{i+1}}$, $v_{i+1}(t) = v_i(t)$ and $s_{i+1}(t) = s_i(t)$.

The node currently holding the point t in a full graph is the largest j such that $v_j(t) < n$ (and thus exists in a graph with addresses 0 through $n - 1$). It is not hard to show that the largest such $j \leq \log n$.

5.2 In an arbitrary graph

We begin with a definition and a lemma:

Definition 5.1 *The **ideal host** of an object is the node which would hold the object if the graph was full at its current dimension (if the number of nodes was extended from the current n to $n' = 2^{\lceil \log n \rceil}$).*

Lemma 5.2 *All node with virtual addresses with fewer bits than the longest existing address must exist. Equivalently, all nodes which are parents of nodes definable within the current dimension must exist.*

Proof. The former holds because of deletion by replacement; the largest address is always removed, so no shorter address could be missing while a larger exists. The latter follows since by definition parent nodes have shorter addresses than their children, and any node definable within the current dimension has address at most the length of the longest existing dimension. □

The process for lookup in a partial graph is more difficult. It can be broken down into the following steps:

1. Guess the *ideal host* for the object based on local information.
2. Route the lookup request to the parent of the ideal host guess; this node must exist.
3. If the guess for the ideal host is incorrect, then the request is forwarded down the tree to the true parent of the ideal host.
4. If the ideal host exists, route the request to it.
5. If the ideal host does not exist, compute the next host to check and route the request to that node's parent. Repeat until found.

Steps 1-3: Finding the parent of the ideal host

The source node of the request estimates the dimensionality of the network using the size of the range it is responsible for. If the source node holds two ranges taken from two different nodes, then its estimate is based on the size of one of the ranges. The node's estimate of the number of nodes in the network is $\frac{1}{\text{range size}}$. If the range is of size $\frac{1}{2^l}$, where $l = \lceil \log n \rceil$, then the node's estimate of the number of nodes is $n' = 2^l = N$, which is considered a correct estimate. If the node has a single unsplit range, in the sense that as the graph grows from its current size to N nodes the range held by the node will be split in half, then $n' = N/2$. In either case, n' is used to guess the ideal host of the object using the computations for a known full graph.

Regardless of whether the estimated ideal host w is the true ideal host, its parent node $p(w)$ must exist by Lemma 5.2, so the lookup request can be sent from the initiating node to $p(w)$. If w is not the true ideal host, then the number of nodes was estimated too small ($n' = N/2$), which implies that node w must exist (since it must have highest-order bit 0), and $p(w)$ can route the request to it. Node w can then determine whether or not it is not the true ideal host for the node. If not, it can be shown through the load balancing algorithm that the true ideal host y is a descendant of w. Then node w can compute y (using the true N) and route the request to the true parent of the ideal host $p(y)$. On the other hand, if w does not exist, then it must be the true ideal host, or $w = y$ and $p(w) = p(y)$.

Steps 4-5: Finding the actual host

If the ideal host y exists, then it must hold the object, because as soon as a node joins it gets its full subrange and all associated objects for the current dimension (all nodes but the last in a dimension also get a second subrange which will later be taken away); then $p(y)$ can route the request to it and the request can be processed. If not, the range history $\{v_i\}$ can be examined for y, and the first valid address is the current node containing the point. Consider the set $\{p_i\}$ where p_i is the address of the parent of v_i for all i; all these nodes must exist according to Lemma 5.2. The search request is passed through the set $\{p_i\}$ until some v_i is valid.

Consider the sequence of nodes $\{p_i\}$ where each $p_i = \lfloor v_i/2 \rfloor$, or the parent of v_i in the search tree. Because of the DeBruijn construction each p_i has an edge to its v_i and is capable of determining whether or not v_i is valid. Suppose the search request is at some node p_i, which has determined that v_i does not exist. Node p_i computes v_{i+1} and p_{i+1} and then forwards the request to p_{i+1}.

5.3 Analysis

Lemma 5.3 *For some $t = O(\log n)$, v_t exists.*

Proof. Assume that v_0 does not exist, so that the lemma is nontrivial. Then v_0's address must be of maximum bit length $\log n$, by deletion by replacement. Divide the set of all nodes with $(\log n)$-bit addresses with highest-order bit 1 into subsets $\{S_j\}$, where S_j contains all nodes with all j highest-order bits 1 and the $(j+1)^{st}$ highest-order bit 0. Then j ranges from 1 to $\log n$, where $S_{\log n}$ is the node with address all 1's. Suppose that v_i is in set S_k for $k > 1$. Then v_{i+1} must be in set S_{k-1}, since the highest 0 in v_i's address is shifted up one bit to produce v_{i+1} (and a 0 is introduced if $v_i \in S_{\log n}$). So for some $t = O(\log n)$, v_{t-1} is in S_1. Then v_t must have $d(v_t) < d(v_0)$. All v_t with $d(v_t) < d(v_0)$ must exist by Lemma 5.2. □

Theorem 5.4 *Object lookup requires $O(\log n)$ time steps and messages.*

Proof. The cost in time and messages of finding p_0, the parent of the ideal host, is $O(\log n)$, since it takes $\log n$ steps to find the parent of the first guess of the ideal host, and the request can only be routed down from there. For the previously defined sets $\{v_i\}$ and $\{p_i\}$, let t be such that v_t exists and for all $i < t$, v_i does not exist. Each routing step from p_0 to p_{t-1} requires only a single step and message, since p_i is either equal to or has an outgoing edge to p_{i+1} according to the DeBruijn construction and the observation that for all $i < t$, $d(v_i) = d(v_0)$; this sequence then requires $O(t)$ steps and messages. The last routing step from p_{t-1} to p_t requires at most $\log n$ steps and messages, and p_t has an edge to v_t, the node responsible for the desired object. Since $t = O(\log n)$ by Lemma 5.3, the total cost is $O(\log n)$. □

6 Recursive divide-and-conquer applications

This network is suitable for many types of distributed applications. One class for which it performs well are recursive divide-and-conquer applications, because of its uniform task assignment and its ability to recursively spawn subtasks as well as its ability to avoid repeating subtasks. It is also able to take much of the load of these computations off the central task coordinator. While these applications can in principle be performed with any DHT, they implicitly use a single collection or starting point and are natural candidates for a supervised network.

6.1 Dynamic programming

Our network allows efficient parallel recursive computation of dynamic programming problems. In some dynamic programs, the domain of cases is so large that a table of computed results cannot be handled by a single computer; our network allows this table to be divided among the nodes. At all but the bottom level (the level where tasks complete locally), each task spawns logically separate subtasks, waits for them to return, and then combines the results. Our network maps each subtask to a point in a universal address space (in this paper the $[0, 1)$ real line) which is then mapped to a node in the network. Computation results are cached when completed. When some other node needs the result of a computation, it sends the request for the task to the appropriate node. If this is the first execution of this task, it is executed. If the task has already been completed, the answer is returned immediately. If the task is in progress, the node is added to a set of nodes waiting for the response to the task. No subtask is ever computed twice, as long as results of computations are exchanged along with ranges when nodes join and leave the network.

6.2 Backtrack distributed tree search

A similar approach can be used to solve the backtrack search problem, discussed in [12] and many other papers. The backtrack search problem involves a parallel traversal of a search tree to find the minimum cost leaf, where each internal node represents a partial solution to a problem and each leaf represents a full solution and its associated cost. Backtrack searches are easy to implement in this setting, since a search beginning at each child can be considered a new object. When a leaf is discovered its cost is returned, and when all the children of a task corresponding to an internal node computation have returned the minimum of their values is then returned.

The paper by Liu *et al.* [12] proves that distributed backtrack search can be performed under a very conservative network model in time $O(n/p + h)$ with high probability, for an n-item tree of maximum depth h using p processors, assuming that task assignment is random. This is asymptotically optimal since at least h time must be used to explore the full depth of the tree, and at least $\frac{n}{p}$ nodes need to be explored by one of the p processors. Using the same analysis, when static our system is asymptotically work-optimal since node workloads within a factor of 2 can at most increase the total computation time by a factor of 2. Under dynamic network conditions, our system maintains this optimality with respect to any online algorithm; the only additional cost comes from task movement away from departing nodes and to joining nodes, which at each step is at most twice what a best-possible online algorithm would move.

6.3 Web crawling

An immediate practical application for the use of this network is distributed web crawling. A web crawler traverses readable directories and links from HTML files to try to explore the entire World Wide Web. This is similar to the process of tree exploration, except that the underlying graph contains cycles. A central database of already explored links or sites can be used to prevent a web crawler from repeating itself, but on the scale of the entire web this is very expensive. An alternate solution is to consider each web page to be an object, and to assign the responsibility for exploring it to the node responsible for the hash of the object. In this way, every link to that web page would be sent to the same node for exploration, which could locally keep track of whether or not it had already explored those sites which it is responsible for, thus avoiding repeated explorations while requiring each participant to keep only its share of the crawl history. If this is too fine a discretization, directories or domains could be considered objects instead of individual pages.

7 Extensions

7.1 Increased fault tolerance

If we remove the graceful departure assumption, we must make the structure more fault-tolerant. To accomplish this, each node can be replaced by a complete graph containing $O(\log n)$ nodes, and each edge can be replaced by a full bipartite graph; then with high probability the system can tolerate a constant fraction of random failures in the network. The supervisor can be contacted to refill heavily depleted clusters with new nodes or nodes acquired by removing whole clusters from the topology. This increases some costs in the network by a logarithmic factor. It also introduces inefficiency into the object management, since the same objects must be stored in $O(\log n)$ places (causing either redundant data storage or redundant computation).

7.2 Broadcasting

In addition to the aforementioned uses, the network also supports broadcasting in logarithmic time with linear message complexity. Some applications could benefit from this capacity, for example to transfer universal update information through the network. Details are omitted due to space considerations.

Acknowledgements

The authors would like to thank Jonathan Shapiro and Brian Wingenroth for their suggestions and comments.

References

[1] G. Ateniese, C. Riley, and C. Scheideler. Survivable monitoring in dynamic networks. In *Proceedings of the International Information Assurance Workshop*, 2004 (to appear).

[2] A. Brinkmann, K. Salzwedel, and C. Scheideler. Efficient, distributed data placement strategies for storage area networks. In *Proceedings of the 12th Annual ACM Symposium on Parallel Algorithms and Architectures* , pages 119–128, 2000.

[3] CERN. http://public.web.cern.ch/public/.

[4] distributed.net. http://www.distributed.net/.

[5] Folding@home. http://folding.stanford.edu/.

[6] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International J. Supercomputer Applications*, 15(3), 2001.

[7] P. Fraigniaud and P. Gauron. The content-addressable network D2B. Technical Report LRI 1349, Univ. Paris-Sud, 2003.

[8] The Globus Alliance. http://www.globus.org.

[9] Gnutella. http://gnutella.wego.com.

[10] D. Jefferson et al. Distributed simulation and the Time Warp operating system. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, 1987.

[11] F. Kaashoek and D. Karger. Koorde: A simple degree-optimal distributed hash table. In *Second International Workshop on Peer-to-Peer Systems*, 2003.

[12] P. Liu, W. Aiello, and S. Bhatt. Tree search on an atomic model for message passing. *SIAM J. Comput.*, 31(1):67–85, 2001.

[13] M. Naor and U. Wieder. Novel architectures for p2p applications: the continuous-discrete app roach. In *Proceedings of the 15th Annual ACM Symposium on Parallelism in Al gorithms and Architectures*, 2003.

[14] Napster. http://www.napster.com.

[15] OptimalGrid. http://www.alphaworks.ibm.com/tech/optimalgrid.

[16] G. Pandurangan, P. Raghavan, and E. Upfal. Building low-diameter p2p networks. In *IEEE Symposium on Foundations of Computer Science*, pages 492–499, 2001.

[17] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM*, pages 161–172. ACM Press, 2001.

[18] P. L. Reiher. Parallel Simulation Using the Time Warp Operating System. In *Proceedings of the Winter Simulation Conference*, pages 38–45, 1990.

[19] C. Riley and C. Scheideler. Guaranteed broadcasting using SPON: Supervised P2P Overlay Network. In *International Zurich Seminar on Communications*, 2004 (to appear).

[20] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM Int'l Conf. on Distributed Systems Platforms*, 2001.

[21] SETI@home. http://setiathome.ssl.berkeley.edu/.

[22] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.