Optimal Node Selection Algorithm for Parallel Access in Overlay Networks

Seung Chul Han and Ye Xia
Computer and Information Science and Engineering Department
University of Florida
301 CSE Building, PO Box 116120
Gainesville, FL 32611-6120
Email: {schan, yx1}@cise.ufl.edu

Abstract

In this paper, we investigate the issue of node selection for parallel access in overlay networks, which is a fundamental problem in nearly all recent content distribution systems, grid computing or other peer-to-peer applications. To achieve high performance and resilience to failures, a client can make connections with multiple servers simultaneously and receive different portions of the data from the servers in parallel. However, selecting the best set of servers from the set of all candidate nodes is not a straightforward task, and the obtained performance can vary dramatically depending on the selection result. In this paper, we present a node selection scheme in a hypercube-like overlay network that generates the optimal server set with respect to the worst-case link stress (WLS) criterion. The algorithm allows scaling to very large system because it is very efficient and does not require network measurement or collection of topology or routing information. It has performance advantages in a number of areas, particularly against the random selection scheme. First, it minimizes the level of congestion at the bottleneck link. This is equivalent to maximizing the achievable throughput. Second, it consumes less network resources in terms of the total number of links used and the total bandwidth usage. Third, it leads to low average round-trip time to selected servers, hence, allowing nearby nodes to exchange more data, an objective sought by many content distribution systems.

Index Terms

Node Selection, Server Selection, Overlay Networks, Hypercube, Content Distribution Networks, Peer-to-Peer Networks, Parallel Download, Link Stress

I. Introduction

A number of structured network topologies, including the hypercube, torus, butterfly and Banyan network, have traditionally been extensively studied in application areas such as parallel or distributed processing and switching. Recently, structured networks are becoming attractive platforms for building overlay networks through which network applications and services can be deployed. Examples of the proposed structured overlay networks include Chord [1], Tapestry [2], Pastry [3], CAN [4], ODRI [5], and Ulysses [6]. They have been applied to diverse applications such as application-level multicast, persistent data storage, file/data access or sharing, media streaming or other content distribution, and grid computing. The benefits of taking the structured overlay approach are many: For instance, it allows fast resource location, decentralizes massive computation or data access, enables large-scale resource sharing, simplifies routing, and improves service quality and fault-tolerance by using multiple paths.

In this paper, we present an optimal node selection scheme that minimizes the worst-case link stress in an overlay network with one of the most popular network topologies, the hypercube. One of the important applications of such a scheme is parallel access of data in a communication network [7], [8], for instance, a content distribution, a peer-to-peer (P2P), or a grid-computing network. The conventional way of downloading data is for a client to select one node as the server from one or more candidate servers. The performance, e.g., the throughput or data downloading time, is directly influenced by the load of the server, congestion of the path of the connection and any traffic fluctuation that may affect routing [9]. In order to alleviate these problems, the client can be allowed to connect to multiple servers simultaneously and receive different portions of the data, or file chunks, from each server in parallel. Such parallel access can avoid server overload, achieve higher throughput or faster distribution speed, and be more resilient to link failure and traffic fluctuation [7]. Most of the recent content distribution systems rely on the parallel download approach. In P2P file sharing/distribution such as BitTorrent [10], parallel download takes the more sophisticated form of *swarming* (or collaborative download), where the file chunks are spread out across the peers and the peers subsequently exchange the chunks with each other to speed up the distribution process. In either the simple parallel-download or the swarming case, the node selection problem arises when a node must select a subset of the nodes that contain the wanted file or file chunks and establish actual connections for receiving data. In sum, node selection is a fundamental component in many content distribution or parallel access systems.

The node-selection algorithms in existing systems usually have a user-centric performance objective, such as reducing the round-trip time (RTT) or the completion time of individual download, but tend to ignore the congestion caused by the concurrent connections from different servers or the total network resource usage. (See Section I-A1 for a review.) The algorithms are typically greedy, local-search heuristics. For instance, in several systems, every node gradually but independently samples the throughput of connections from different servers and selects the servers that lead to higher connection throughput. This myopic strategy treats different connections separately but ignores their possible interaction. If many connections pass through the same network links, they may overload the links. A likely consequence is poorly balanced network load, which hurts the overall performance of all users, as well as causes inefficient utilization of the network resources.

This paper is fairly unique in emphasizing both the network-related and user-related performance metrics. (See also [11], [12].) It advocates an optimal, recursive selection algorithm that minimizes the worst-case link stress (WLS). The WLS is directly related to the worst congestion level in the network, an important performance concern for network and service providers. The minimum WLS corresponds to the best-balanced network load. A balanced use of the network allows it to accommodate more traffic, i.e., to improve the overall throughput. Optimizing the WLS also improves the service quality received by a user or a multi-user session. When every user tries to minimize the WLS in its server selection process, the chance is better that other users can also receive good performance in their communication sessions. Applications such as media streaming are sensitive to bandwidth availability and lower WLS implies better performance. In the case of elastic data transfer, the inverse of the WLS provides an alternative performance perspective: It yields the maximum achievable throughput of the parallel download session, which determines the shortest download time without causing network congestion. Finally, our particular recursive WLS-minimizing algorithm is also very competitive in terms of two other performance metrics, the average path length and total network resource usage.

Another distinguishing feature of the paper is that the overlay network is considered as a complete virtual network where the overlay nodes act as virtual routers for the overlay, logical network. As a result, data (as apposed to only queries for data) are routed on the overlay network instead of the underlay network. This is in congruence with the interesting trend of network virtualization through the overlay approach, which is useful not only for introducing new network applications but also for deploying new network architectures [13].

In this paper, the overlay network is organized as a hypercube. The motivation is that the hypercube is one of the simplest and most versatile structured networks. It is well known ([14], [15], [16], [17], [18], [19]) that the structural regularity of the hypercube often makes parallel or distributed algorithms extremely simple and efficient. The same is expected for many networking algorithms on the hypercube. For instance, routing can be performed based on a pre-specified convention without the need of a routing protocol. The chance is high that the hypercube will be the topology of choice for many applications that employ virtual networks. The focus of this paper is to develop a similarly efficient algorithm for minimizing the WLS on the hypercube. It turns out that such an algorithm does exist and it requires no knowledge of the current network condition, not even the topology or routing information, hence, avoiding the expensive overhead of network measurement or passing control messages. The only requirement is that the receiving node (client) can obtain a list of IDs of the server candidates through the overlay network. This is usually done by searching a distributed directory based on the distributed hash table (DHT) [1], [4], [20], possibly augmented by some gossip protocol [21], [10] in a highly dynamic network where nodes frequently arrive and depart. The efficiency and simplicity of the algorithm allow scaling the network to very large size, capable of handling massive content distribution or parallel access. The paper will show that the recursive WLS-minimization algorithm has desirable performance with respect to the three aforementioned metrics for both the overlay hypercube-like network and the underlay Internet-like network.

The combination of the hypercube and the choice of the WLS-minimizing objective makes this paper directly relevant to the infrastructure-based overlay network for massive content distribution, and overlay virtual networks for distributed access or computation, such as those required by E-science or grid computing. But, it should be also relevant to other P2P/overaly file distribution, and parallel or distributed computation on other network topologies, including other structured networks or general, non-structured networks. The node selection algorithm can be extended to those networks, but with some loss of efficiency. Since the hypercube overlay network is a special case of the Plaxton-type networks [22] and a relative to Pastry [3] and Tapestry [20], extension to those networks is expected to require the smallest modification.

This paper is organized as follows. In Section II, we introduce the hypercube overlay network and present required definitions and facts. In Section III, we present the optimal server selection algorithm for minimizing the WLS and analyze its running time. We evaluate the algorithm and compare it with other algorithms through simulation in

Section IV. Finally, the conclusions are drawn in Section V.

A. Literature Review

1) Node Selection Algorithms: The literature on node selection is vast. We will mainly review the most relevant studies in overlay networks. Existing distribution systems handle node selection in a variety of (usually ad-hoc) ways[1]. The infrastructure-based content distribution networks (CDN) (e.g., Akamai [23]) and web caches generally assign the closest server to each client. In SplitStream [24], and FastReplica [25], server selection is essentially done randomly. Other systems employ a server or peer ranking function. A node favors those peers with high ranking. The ranking function may be the nodal load (CoBlitz [26]), the round-trip time (RTT) (ChunkCast [27]), the sending and/or receiving bandwidth to and from each peer (Bullet$'$ [28], Slurpie [29] and BitTorrent [10]), and the degree of content overlap between the receiver and the server candidate (Bullet [21]). One common practice is that a node initially selects some random peers, but gradually probes other peers and dynamically switches to those with better ranking over the course of downloading. Julia [30] assumes a structured but locality-aware P2P network, where each peer exchanges file chunks with direct neighbors in differential amount, more with closer neighbors. This reduces the total *work*, which is defined as the weighted sum of the total network traffic during the entire distribution process, where the weights are the distances travelled by each bit. [31] formulates several continuous optimization problems regarding peer selection in P2P file download or streaming. In more traditional server selection literature, [32] presents a dynamic server selection scheme based on instantaneous measurement of the RTT and available bandwidth. [9] proposes a selection scheme that utilizes the underlying network topology and the performance information of the senders. Similar research works are also reported in [7], [33].

2) Hypercube Networks: The hypercube is among the most popular networks studied by researchers due to its useful structural regularity. Many have studied its topological properties and communication capability for parallel and distributed computing applications. (See [14], [15], [16], [17], [18] for a small sample and [19] for a comprehensive textbook treatment.) In the area of communication networks, [34] investigates how the hypercube affects the spread of worms and viruses. [35] considers a multiple access protocol in wireless ad-hoc networks with the hypercube topology. [36] provides analysis of a hypercube-based distributed hash table (DHT) that achieves asymptotically optimal load balance. [37] proposes a failure recovery protocol for structured P2P networks that use hypercube routing. We are not aware of any prior node-selection algorithms on the hypercube that are similar to ours.

II. Preliminaries

In this section, we introduce the definition of the hypercube network and some of its properties that make server selection based on the WLS-minimizing criterion efficient and easy.

The overlay network we consider is a hypercube with N nodes, numbered $0, 1, \ldots, N-1$. Suppose $N = 2^m$, where m is a positive integer. The node IDs can be expressed as binary strings, and two nodes are linked with an edge if and only if their binary strings differ in precisely one bit. The routing rule is as follows. At each node s

[1]Not all systems frame or handle the node selection problem explicitly. But all should have at least an implicit selection algorithm.

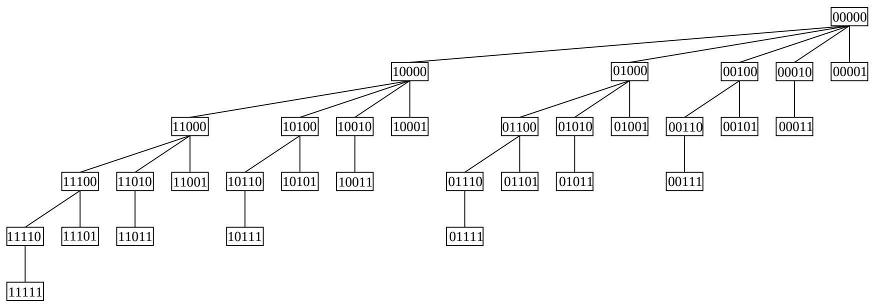

Fig. 1. A labelled 5-level binomial tree B_5.

and for the destination d, let l be the first bit position, counting from the right, that s and d have different values. Then, the next hop on the route to d is the neighbor of s that differs with s in the l^{th} bit. Consider the example where $N = 2^5$, the source node is 10110 and the destination node is 00000. The route consists of the following sequence of nodes: $10110 \rightarrow 10100 \rightarrow 10000 \rightarrow 00000$.

In most earlier works, especially in the parallel computing community, the definition of a hypercube only specifies how nodes are connected. In this paper, as in many structured overlay networks, the routing rule is an important part of the definition. With the above routing rule, the hypercube network is a special instance of the Plaxton-type networks [22], which are broad enough to also include Pastry [3] and Tapestry [20], and are fundamentally related to Chord [1] and CAN [4]. We also note that the hypercube clearly requires that every position of the name space is occupied by a node, which is our current assumption. (In Section IV-F, we consider the situation where the name space of the overlay network is not fully populated by nodes.)

A helpful device for visualizing the hypercube and its routing is the embedded tree, as shown in Fig. 1, consisting of all valid routes allowed by the routing rule from all nodes to node 0. It is known that such a tree is a binomial tree [38], [19]. The binomial tree embedded in the hypercube network is a labelled tree, where the label of each node is the node's ID. By the symmetry of the hypercube, there is an embedded binomial tree rooted at every node where the tree paths are the valid routes to the root node. Given the labelled binomial tree rooted at node 0, an easy way to derive the labels for the binomial tree rooted at node $d \neq 0$ is to XOR the node labels of the former tree with d.

Throughout the paper, let us denote the k-level binomial tree rooted at node 0 by $\mathcal{T} = B_k$. Let us denote the ID of node u by $I(u)$. When there is no confusion, we will use u and $I(u)$ interchangeably to denote node u.

Without loss of generality, the root node 0 is understood as the client. The candidate nodes are those that contain the data and the server nodes are those that are eventually selected to participate in transmitting the data.

The following lemmas and corollary are simple consequences of the hypercube definition, including its routing rule. They will be required later.

Lemma 2.1: Suppose u is the parent of node v in $\mathcal{T}$. Then, u and v differ in exactly one bit, and the value of the bit is 0 for u and 1 for v. All bits to the right of the different bit are 0's.

Lemma 2.2: Suppose u is a node in $\mathcal{T}$, and $I(u) = a_1\ \ldots\ a_i\ 0 \ldots 0$, $0 \leq i < m$. Then,

(i) the subtree rooted at u contains all nodes whose IDs fall between $a_1 \ldots a_i 0 \ldots 0$ and $a_1 \ldots a_i 1 \ldots 1$, inclusive;

(ii) $I(u) \leq I(v)$ for all nodes v in the subtree;

(iii) if the subtree contains nodes v and w, then it contains all nodes whose IDs fall between $I(v)$ and $I(w)$;

(iv) if v is a descendant of u, the longest common prefix of $I(u)$ and $I(v)$ is greater than i.

Corollary 2.3: Suppose that, in $\mathcal{T}$, node v's ID is $a_1 \ldots a_m$. Then, a node u is an ancestor of node v if and only if the ID of u has the form $a_1 \ldots a_i 0 \ldots 0$, where $0 \leq i < m$[2], and $I(u) \neq I(v)$.

Definition 1: A **common ancestor** of a set of nodes $S = \{s_1, \ldots, s_n\}$, where $n \geq 2$, in a rooted tree is a node u satisfying the condition that, for each i, $1 \leq i \leq n$, either u is an ancestor of s_i or u is s_i.

The definition allows the following case. If node v is an ancestor of node w, then v is a common ancestor of v and w.

Definition 2: The **lowest common ancestor** (LCA) of a set of nodes $S = \{s_1, \ldots, s_n\}$, where $n \geq 2$, in a rooted tree is the deepest node in the tree that is a common ancestor of all nodes in S. It is denoted by $LCA(S)$ or $LCA(s_1, \ldots, s_n)$.

For example in Fig. 1, if $I(u) = 01011$ and $I(v) = 01101$, then $LCA(u, v)$ is the node with ID 01000.

Definition 3: Given a subset, S, of two or more nodes in $\mathcal{T}$, let us denote the set of LCAs of all subsets of S with two or more nodes by $SLCA(S)$.

Let us denote the *longest common prefix* (LCP) of a set of m-digit binary labels, $R = \{b^1, \ldots, b^k\}$, by $LCP(b^1, \ldots, b^k)$ or $LCP(R)$, and denote the m-digit label equal to the concatenation of $LCP(R)$ with an appropriate number of 0's by $\lambda(R)$ or $\lambda(b^1,\ \ldots,\ b^k)$. For instance, $LCP(01110,\ 01011) = 01$, and $\lambda(01110,\ 01011) = 01000$. For a set of nodes $S = \{s_1, ..., s_n\}$, $LCP(S)$ or $LCP(s_1, \ldots, s_n)$ are the simplified notations for $LCP(I(s_1),\ \ldots,\ I(s_n))$, and $\lambda(S)$ or $\lambda(s_1, \ldots, s_n)$ are the simplified notations for $\lambda(I(s_1),\ \ldots,\ I(s_n))$.

Definition 4: Suppose $S = \{s_1, \ldots, s_n\}$ is a set of nodes in $\mathcal{T}$ and $E = \{e_1, \ldots, e_l\}$ is the set of edges used by the paths from the root to nodes in S, called the **S-paths**. The **link stress** of an edge e in $\mathcal{T}$, denoted by $LS(e)$, is the number of S-paths via e. Let E be the set of all edges in $\mathcal{T}$. The **worst link stress** (WLS) is defined as $\max_{e \in E} LS(e)$.

The WLS is the largest number of downloading streams on any link and is an indication of how well the load is balanced in the network and of how much connections from different servers interfere with each other at the bottleneck link, assuming the links have roughly identical bandwidth. It is both a measure of the burden placed by the parallel download session on the network resources and a measure of the quality of data delivery. The reciprocal of the WLS tells how many multiples each downloading stream can be increased without causing network congestion. It gives the maximum achievable throughput of the parallel download session, which, in turn, determines the shortest download time. A session that follows the WLS-minimizing rule in its server selection makes a balanced use of the network bandwidth, which tends to cause the least interference to other sessions. If every session follows the same rule, the chance is good that all sessions can benefit.

[2]By convention, when $i = 0$, $a_1 \ldots a_i 0 \ldots 0$ is $0 \ldots 0$.

III. Optimal Server Selection for Minimizing Worst Link Stress

A. Highlight of the Algorithm and Results

The problem addressed in this section can be stated as follows. Given the client and a set of nodes containing the data (*the candidates*), denoted by S, select k (*the degree of parallelism*) servers from the candidate set to participate in transmitting the data to the client so that the worst-case link stress (WLS) is minimized. The key observation is that the link stress of any edge (u, v), where u and v are nodes in $\mathcal{T}$ and u is the parent of v, is no less than that of any edge in the subtree $\mathcal{T}_v$, the subtree rooted at node v. Hence, the edge with the worst link stress must be connected to the root of $\mathcal{T}$. One possible algorithm for minimizing the WLS is as follows. Let W be the set of children of the root. For $w \in W$, let S_w be the set of candidates in $\mathcal{T}_w$, the subtree rooted at w. Suppose each $w \in W$ is labelled with $|S_w|$, i.e., the number of nodes in S_w. Then, we call the *MIN-MAX* algorithm with the arguments $l = |W|$, $(b_1, \ldots, b_l)$ equal to the list of $|S_w|$'s and $q = k$. The returned list of numbers from the MIN-MAX algorithm, which will be discussed in Section III-C, is an optimal allocation, with respect to the WLS-minimization criterion, of the number of servers to be selected in each subtree $\mathcal{T}_w$, for all $w \in W$.

The actual algorithm (Algorithm 1) is different. It minimizes the WLS recursively in the sense that it minimizes the WLS for every subtree rooted at every $v \in SLCA(S)$. The WLS for each such subtree, $\mathcal{T}_v$, is defined over only the edges in $\mathcal{T}_v$. That is, the WLS over $\mathcal{T}_v$ is $\max_{e \in E_v} LS(e)$, where E_v is the set of edges in $\mathcal{T}_v$. The initial motivation for recursively minimizing the WLS is that, in the more general problem involving multiple clients, this leads to a better chance of achieving lower link stress for heuristic algorithms built on top of the optimal single-client algorithm[3]. This will be confirmed experimentally by our evaluations. (See Section IV-C and IV-D.) It further turns out the recursive algorithm leads to excellent performance with respect to two other criteria, the average path length and the total network resource usage.

The theoretical running time of the algorithm is $O(nm^2)$, where n is the number of nodes in the candidate set and 2^m is the size of the hypercube. Hence, m is at least $O(\log n)$. We will see that the practical running time of the algorithm is in fact $O(nm)$. The saving comes from how one thinks about manipulating m-bit numbers. For instance, in theory, adding or comparing two m-bit numbers takes $O(m)$ running time. However, for practical problems, m is almost always small enough, e.g., 32 or 128. Such operations can be completed in constant time in typical microprocessor architectures. We call any operation an *elementary operation* if it can be completed in constant time in typical microprocessor architectures, provided that the numbers involved in the operation are of constant sizes. In practice, the algorithm takes $O(nm)$ elementary operations.

Although the high-level ideas about Algorithm 1 are mostly straight-forward, an efficient implementation is not. The analysis on its running time is not trivial either. In Section III-B, we describe a crucial characterization of

[3]One may notice that a more complete problem formulation is to consider a set of clients, each with a set of known server candidates. The problem is to assign a subset of the servers to each client so as to minimize the WLS. Here, the link stress is naturally defined as the total number of connections on the link, regardless of which client the connection belongs to. The solution to this problem, although very interesting to think about, will be complicated and will require that every client has the complete knowledge of all other clients, as well as their server candidate sets. Such information will be hard to obtain cheaply in a large network. Because the single-client WLS-minimization algorithm aims at balancing the network load, the simple heuristic of applying the single-client algorithm separately and independently by every client is a viable approach.

$SLCA(S)$ that makes Algorithm 1 efficient. Then, we describe two key subroutines in Section III-C and III-D, and discuss their running time. In Section III-E, we give a full description of the algorithm, discuss the remaining key steps, and give the total running time.

B. A Characterization of the Set $SLCA(S)$

We will need the following two lemmas for a characterization of $SLCA(S)$.

Lemma 3.1 (Ancestor Algebra): Suppose S is a set of nodes in an arbitrary tree. Let $W_1, \ldots, W_n$ be a covering of S. That is, $W_i \subseteq S$ for $1 \le i \le n$, and $\cup_{i=1}^{n} W_i = S$. Then,

$$LCA(S) = LCA(LCA(W_1), \ldots, LCA(W_n))$$

The proof of Lemma 3.1 is rather easy. We omit it for brevity.

Lemma 3.2: Suppose $(s_1, \ldots, s_n)$ is a sorted list of distinct nodes in $\mathcal{T}$ by their IDs, for some $1 < n \le 2^m$. Then,

$$LCA(s_1, \ldots, s_n) = LCA(s_1, s_n)$$

The proof of Lemma 3.2 is given in [39].

Lemma 3.3: Suppose $(s_1, \ldots, s_n)$ is a sorted list of distinct nodes in $\mathcal{T}$ by their IDs, for some $1 < n \le 2^m$. Then, $SLCA(s_1, \ldots, s_n) = \bigcup_{i=1}^{n-1} LCA(s_i, s_{i+1})$.

Proof: Let $S = \{s_1, \ldots, s_n\}$. By the Ancestor Algebra Lemma (Lemma 3.1), we only need to focus on the LCAs of node pairs, because they form a covering for every subset of S. Thus, by definition, $SLCA(S) = \bigcup_{1 \le i < j \le 2^m} LCA(s_i, s_j)$.

The proof is based on induction. Let us define the sets $S^i = \{s_1, \ldots, s_i\}$, for $2 \le i \le n$. As the base case, the lemma is trivially true for S^2. We make the induction hypothesis that the lemma is true for S^l, where $2 \le l < n$. Then,

$$\begin{aligned} & SLCA(S^{l+1}) \\ = & \bigcup_{1 \le i < j \le l+1} LCA(s_i, s_j) \\ = & \Big(\bigcup_{1 \le i < j \le l} LCA(s_i, s_j) \Big) \cup \Big(\bigcup_{i=1}^{l} LCA(s_i, s_{l+1}) \Big) \\ = & SLCA(S^l) \cup \Big(\bigcup_{i=1}^{l} LCA(s_i, s_{l+1}) \Big) \\ = & \Big(\bigcup_{i=1}^{l-1} LCA(s_i, s_{i+1}) \Big) \cup \Big(\bigcup_{i=1}^{l} LCA(s_i, s_{l+1}) \Big) \end{aligned} \tag{1}$$

Consider $LCA(s_i, s_{l+1})$ for some $1 \le i \le l-1$.

$$\begin{aligned} LCA(s_i, s_{l+1}) &= LCA(s_i, s_l, s_{l+1}) \\ &= LCA(LCA(s_i, s_l), LCA(s_l, s_{l+1})) \\ &= LCA(s_i, s_l) \text{ or } LCA(s_l, s_{l+1}) \end{aligned}$$

The first equality above is by Lemma 3.2, the second one is by the Ancestor Algebra Lemma (Lemma 3.1). The last equality holds because $LCA(s_i, s_l)$ and $LCA(s_l, s_{l+1})$ are both ancestors to s_l, or one or both are identical to s_l, and in a rooted tree, one must be the ancestor to the other unless they are the same node (Both are on the path from s_l to the root.).

We have shown that $LCA(s_i,\, s_{l+1})$ is either the same as $LCA(s_l,\, s_{l+1})$, or the same as $LCA(s_i,\, s_l)$, which is already in $SLCA(S^l)$. Hence, by (1)

$$\begin{aligned} SLCA(S^{l+1}) &= SLCA(S^l) \cup LCA(s_l, s_{l+1}) \\ &= \bigcup_{i=1}^{l} LCA(s_i, s_{i+1}) \end{aligned}$$

■

C. The MIN-MAX Algorithm

In this section, we describe the MIN-MAX algorithm, which is called by Algorithm 1. For brevity, we will not list the full algorithm and will omit some proofs. We will only give the basic idea. The algorithm is invoked by calling MIN-MAX(l, $(b_1, \ldots, b_l)$, q), where l and q are positive integers, and $(b_1, \ldots, b_l)$ is a vector of l positive integers. It returns a vector of l positive integers.

Consider l sets, 1 through l. Set j contains b_j items, for $j = 1, \ldots, l$. Suppose we must choose q items from these l sets. The objective is to decide the number of items chosen from set j, denoted by c_j, for $1 \leq j \leq l$, so that $\sum_{j=1}^{l} c_j = q$ and that the largest c_j is minimized. That is, $\max_{1 \leq j \leq l} c_j$ is minimized. This is called a *min-max allocation* of the items to the sets. In Algorithm 1, we frequently face the situation that, at a fixed node $u \in SLCA(S)$, we need to select $q(u)$ servers from the subtree $\mathcal{T}_u$. The strategy taken is to consider the immediate descendants of u that are in $SLCA(S)$, say $v_1, \ldots, v_l$. Suppose, for $i = 1, \ldots, l$, the subtree $\mathcal{T}_{v_i}$ contains b_i candidates. We then call MIN-MAX(l, $(b_1, \ldots, b_l)$, $q(u)$) to decide how many servers are to be selected from each subtree $\mathcal{T}_{v_i}$.

We will describe a *water-filling* idea on which the MIN-MAX algorithm is based. Imagine filling l jars of volumes b_1, …, b_l with q (volume) units of water, which is infinitely divisible. An idealized strategy is to fill all l jars simultaneously at the same rate. When a jar becomes full, the filling for it stops, while the filling for the rest continues, until the q units of water are all used up. The resulting allocation of water is the min-max allocation. It is easy to see that a literal implementation of the water-filling algorithm takes $O(l^2)$ elementary operations on numbers. The actual MIN-MAX algorithm is a more efficient implementation of the idea, which first finds the maximum of the min-max allocation. It takes $O(l \log l)$ elementary operations on numbers.

One desirable property of the min-max allocation is that it is the most “balanced” partition of the items into different sets. This is captured in the following lemma.

Lemma 3.4: Suppose each item is infinitely divisible. In the min-max allocation, the difference between the largest allocation and the j^{th} smallest allocation is no greater than in any other allocation, for all j.

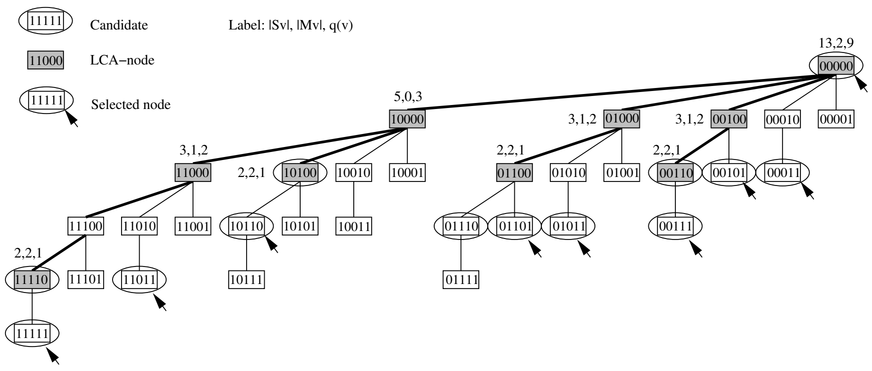

Fig. 2. An example of a virtual tree $\hat{\mathcal{T}}$ in the binomial tree B_5. The candidate set S consists of all circled nodes. The set of nodes in $\hat{\mathcal{T}}$ are all the nodes in $SLCA(S)$, which are shaded, and the highlighted paths are edges in $\hat{\mathcal{T}}$. Nine nodes are to be selected from thirteen candidates in S. The selected nodes by Algorithm 1 are indicated by arrows. Each node in $\hat{\mathcal{T}}$ is also labelled with $|S_v|$, $|M_v|$ and $q(v)$, where $q(v)$ is the number of servers to be selected from the subtree of $\mathcal{T}$ rooted at v, $\mathcal{T}_v$.

D. *Constructing the Virtual Tree, $\hat{\mathcal{T}}$*

The SERVER-SELECTION-WLS algorithm (Algorithm 1) in Section III-E involves visiting all nodes in a virtual tree formed by the nodes in $SLCA(S)$. In this section, we describe an efficient algorithm for constructing this tree. We will show later that its running time dominates the running time of Algorithm 1.

Definition 5: Given a set S with two or more nodes in $\mathcal{T}$, the virtual tree $\hat{\mathcal{T}} = (\hat{V}, \hat{E})$ is formed by the nodes $\hat{V} = SLCA(S)$. For any two nodes $u, v \in \hat{V}$, $(u, v) \in \hat{E}$ if u is the immediate ancestor of v in $\hat{V}$. That is, u is on the path from v to the root of $\mathcal{T}$, and there are no other nodes in $\hat{V}$ on the path segment from v to u. The root of $\hat{\mathcal{T}}$ is $LCA(S)$.

An example of the virtual tree $\hat{\mathcal{T}}$ in the binomial tree B_5 is shown in Fig. 2. An efficient algorithm for constructing $\hat{\mathcal{T}}$ is based on Lemma 3.3. First, we sort the nodes in S in increasing order of the node ID. This takes $O(n \log n)$ comparisons of m-bit numbers. Now, assume $S = (s_1, \ldots, s_n)$ is a sorted list. The set of nodes in $\hat{\mathcal{T}}$ is $\hat{V} = \{LCA(s_1, s_2),\ LCA(s_2, s_3),\ \ldots,\ (s_{n-1}, s_n)\}$. The root is $r = LCA(s_1, s_n)$.

The edges in $\hat{E}$ can be identified by traversing the paths in $\mathcal{T}$ from each node, say v, in $\hat{V}$ toward the root of $\mathcal{T}$. In each step along the way, we inspect if the current node, say u, is in $\hat{V}$. If so, the edge (u, v) is added to $\hat{E}$, and the path traversal originating from node v is stopped. This scheme will work on any general network provided the client has collected the topology information of the network. In the hypercube network, of course, the topology information is known without data collection. We will not dwell on this scheme but will use a more efficient algorithm for identifying the edges.

1) Edge identification algorithm for $\hat{\mathcal{T}}$: Suppose $SLCA(S) = \{v_1, \ldots, v_l\}$, for some $1 \le l \le n-1$. We first sort the nodes in $SLCA(S)$ by increasing order of their IDs, which takes $O(n \log n)$ comparisons of m-bit numbers. Suppose the resulting sorted list is $L = (v_1, \ldots, v_l)$. By Lemma 2.2, the root of $\hat{\mathcal{T}}$ must be v_1. The children of v_1 in $\hat{\mathcal{T}}$, denoted by $\Pi(v_1)$, can be identified by scanning the list. Let node w be any node in $\Pi(v_1)$. Also by Lemma 2.2, all the LCA-nodes (i.e., the nodes in $SLCA(S)$) in the subtree of $\mathcal{T}$ rooted at w, i.e., $\mathcal{T}_w$, must be consecutive in the list L. Hence, *Fact 3.5:* Starting from v_2, the nodes in L are partitioned into $|\Pi(v_1)|$ groups. Each group corresponds to one $w \in \Pi(v_1)$ and is a contiguous sub-list of L. The first node in each sub-list is a child of v_1.

In addition to $v_2 \in \Pi(v_1)$, the other children of v_1 can be identified with the help of the following additional facts.

Fact 3.6: For any two nodes $w_1, w_2 \in \Pi(v_1)$, $\lambda(w_1, w_2) = I(v_1)$.

Fact 3.7: For any descendant x_1 of w_1, we have $|LCP(w_1, w_2)| < |LCP(w_1, x_1)|$, where $|LCP(w_1, w_2)|$ denotes the length of the longest common prefix between $I(w_1)$ and $I(w_2)$.

We will call a descendant of v_1 of depth i in $\hat{\mathcal{T}}$ a *level-i descendant*, for $i \geq 1$. Since v_2 is the first identified level-1 descendant of the root v_1, we denote it by $v_{1,1}$. Let us also denote v_1 by $v_{0,1}$. Next, we find the first node after $v_{1,1}$ in L, denoted by $v_{1,2}$, such that $\lambda(v_{1,1}, v_{1,2}) = I(v_{0,1})$. This is another node in $\Pi(v_{0,1})$. Suppose $v_{1,j}$ has been identified, for some $j \geq 1$. We proceed by finding the first node after $v_{1,j}$ in the list L, denoted by $v_{1,j+1}$, such that $\lambda(v_{1,j}, v_{1,j+1}) = I(v_{0,1})$. When this procedure finishes, we have identified all the nodes in $\Pi(v_{0,1})$, i.e., the level-1 descendants. We then repeat the procedure for each sub-list of L headed by one level-1 descendant and terminated just before the next level-1 descendant in L.

In the general step, suppose we have identified all level-j descendants of $v_{0,1}$, for some $j \geq 1$, denoted by $v_{j,1}, \ldots, v_{j,n_j}$, where $n_j \geq 1$. We repeat the procedure for each sub-list headed by one level-j descendant and terminated just before the next level-j descendant in L.

The total number of levels is no more than m. For each level, the procedure steps through the list L. Hence, the procedure takes $O(nm)$ elementary manipulation of m-bit numbers. The running time is $O(nm^2)$ in theory, and $O(nm)$ in practice. The construction of the virtual tree, $\hat{\mathcal{T}}$, is called in line 6 of Algorithm 1. Hence,

Lemma 3.8: The running time for line 6 in Algorithm 1 is $O(nm^2)$ in theory, and $O(nm)$ in practice.

2) Edge identification example: Consider the example shown in Fig. 2. The sorted list of nodes in $SLCA(S)$ (shaded nodes) is (00000, 00100, 00110, 01000, 01100,10000, 10100, 11000, 11110). From this list, we first identify 00000 as the root of $\hat{\mathcal{T}}$, and 00100 as one of the children of 00000. As we continue to scan the list, we see that λ(00100, 00110) $=$ 00100 and λ(00100, 01000) $=$ 00000. We conclude that node 00110 is a descendant, in fact, the only child, of 00100 and 01000 is another children of 00000. Now, starting from node 01000, we have λ(01000, 01100) = 01000 and λ(01000, 10000) = 00000. Hence, 10000 is another children of 00000 and 01100 is the child of 01000. Continue with node 10000, we have λ(10000, 10100) = 10000, λ(10000, 11000) = 10000 and λ(10000, 11110) $=$ 10000. Hence, nodes 10100, 11000 and 11110 are all descendants of 10000.

The procedure repeats for each children of node 00000. For instance, consider node 10000 and the associated list (10000, 10100, 11000, 11110). We can immediately conclude that node 10100 is one of its children. From λ(10100, 11000) $=$ 10000, we know that 11000 is another children of 10000. Finally, from λ(11000, 11110) $=$ 11000, we conclude that 11110 is the child of node 11000.

E. Server Selection for Minimizing the WLS

This section describes the remaining key steps of Algorithm 1 and its running time. In Algorithm 1, the nodes in the tree $\hat{\mathcal{T}}$ are visited in the breadth-first fashion . The data structure Q is a first-in, first-out queue of nodes yet to be visited. Let M_v be the set that contains all nodes in S for which v is the immediate ancestor in $\hat{\mathcal{T}}$ (i.e., in

$SLCA(S)$), and that have no descendants in S. More formally, let v be a node in $\hat{\mathcal{T}}$ and let S_v be the set of nodes in S in the subtree of $\mathcal{T}$ rooted at node v, denoted by $\mathcal{T}_v$. Then, M_v is a subset of S_v, and for each node $w \in M_v$ not the same as v, v is the only node in $\hat{\mathcal{T}}$ on the path segment from w up to v, and w has no descendants in S (I.e., w is not in $\hat{\mathcal{T}}$.). In addition, if $v \in S$, then v is also included in M_v. Consider the example in Fig. 2. For $v = 00000$, M_v contains two nodes, v itself and node 00011. For $v = 11000$, M_v contains only node 11011, but not node 11110 since it is the parent of node 11111, which is in S.

1) An server selection example: Before discussing the formal algorithm, let us consider the example shown in Fig. 2, where $\mathcal{T}$ is a 5-level binomial tree and S consists of 13 candidates, indicated as circled nodes. The objective is to select $k = 9$ servers from the set S. The set of nodes in $\hat{\mathcal{T}}$ are shaded, and the highlighted paths are edges in $\hat{\mathcal{T}}$. Each node in $\hat{\mathcal{T}}$ is also labelled with $|S_v|$, $|M_v|$ and $q(v)$, where $q(v)$ is the number of servers to be selected from the subtree of $\mathcal{T}$ rooted at v, $\mathcal{T}_v$. The selected servers by Algorithm 1 are indicated with arrows. The first node to be visited is the root of $\hat{\mathcal{T}}$, node 00000, denoted by u. It is understood that 9 servers is to be selected from the tree rooted at u. Since two nodes are in M_u, node u itself and node 00011, those two are selected first. The motivation is that the nodes in M_u generates no more than 1 unit of link stress on the edges from themselves up to u. The remaining 7 servers to be selected are allocated to the subtrees of $\mathcal{T}$ rooted at the children of u in $\hat{\mathcal{T}}$, according to the MIN-MAX algorithm. The resulting allocation is that 2, 2, and 3 servers are to be selected from the subtrees rooted at node 00100, 01000, and 10000, respectively. Then, the node selection algorithm runs recursively for each subtree.

2) Description of line 7: Line 7 can be accomplished in the process of building an extended virtual tree, denoted by $\hat{\mathcal{T}}^e$, whose nodes are $SLCA(S) \cup S$. An edge (u, v) belongs to $\hat{\mathcal{T}}^e$ if u is on the path from v to the root in $\mathcal{T}$, and there is no other node in $\hat{\mathcal{T}}^e$ on the path segment from v to u. In other words, $\hat{\mathcal{T}}^e$ is derived by extending $\hat{\mathcal{T}}$ to include all nodes in S and the corresponding new edges. The algorithm for establishing the edges of $\hat{\mathcal{T}}$ can be applied to $\hat{\mathcal{T}}^e$. For every node $v \in SLAC(S)$, the value $|S_v|$, which is the number of nodes in S in the subtree $\mathcal{T}_v$, is obtained by counting the number of nodes that are in S in the sublist started with v. After $\hat{\mathcal{T}}^e$ is established, the set M_v and its size $|M_v|$ can be obtained and recorded at node $v \in SLAC(S)$ by scanning the list of nodes in S. Suppose that, in $\hat{\mathcal{T}}^e$, $v \in SLAC(S)$ is the parent of node $w \in S$, and w is a leaf node. We add w to M_v and increment a counter for the size of M_v. The running time for line 7 is the same as that for constructing $\hat{\mathcal{T}}$ (line 6).

Lemma 3.9: The running time for line 7 in Algorithm 1 is $O(nm^2)$ in theory, and $O(nm)$ in practice.

3) Description of the **while** *loop:* The variable $q(u)$ is the number of nodes in S yet to be selected from the subtree $\mathcal{T}_u$. When possible, the algorithm first selects the nodes in M_u, which are either leaf nodes in $\hat{\mathcal{T}}^e$ connected to node u, or u itself if $u \in S$ (lines 12 through 18). After that, if more nodes still need to be selected, they will be from the subtrees of $\mathcal{T}$ rooted at each of the children of u in $\hat{\mathcal{T}}$. The algorithm determines an optimal allocation of the number of nodes yet to be selected from each of these subtrees according to the min-max criterion (line 21). For each $v \in \Pi(u)$, if the allocation to the subtree rooted at v is non-zero, then v is added to the end of queue, Q (lines 23 through 25).

4) Running time analysis for the **while** *loop:* Let us now analyze the running time of the *while* loop (lines 10 through 29) in Algorithm 1.

Algorithm 1 SERVER-SELECTION-WLS(S, k)

```
1:  input: $S = \{s_1, \ldots, s_n\}$, a candidate set; an integer $1 \leq k \leq n$
2:  output: $G$, an optimal set with $k$ servers w.r.t. the WLS cost
3:  $G \leftarrow \emptyset$
4:  Sort $S$ and re-index the nodes so that $I(s_1) \leq I(s_2) \leq \ldots \leq I(s_n)$
5:  $SLCA(S) \leftarrow \{LCA(s_1, s_2), LCA(s_2, s_3), \ldots, (s_{n-1}, s_n)\}$
6:  Construct $\hat{\mathcal{T}}$, $r \leftarrow$ root of $\hat{\mathcal{T}}$
7:  At each $v \in SLCA(S)$, record $|S_v|$, $M_v$, $\kappa(v) \triangleq |M_v|$
8:  $Q \leftarrow \{r\}$
9:  $q(r) \leftarrow k$
10: while $Q \neq \emptyset$ do
11:     $u \leftarrow$ head[$Q$]
12:     if $q(u) > \kappa(u)$ then
13:         $G \leftarrow G \cup M_u$
14:         $q(u) \leftarrow q(u) - \kappa(u)$
15:     else if $0 < q(u) \leq \kappa(u)$ then
16:         Add an arbitrary subset of $M_v$ with $|q(u)|$ elements to $G$
17:         $q(u) \leftarrow 0$
18:     end if
19:     if $q(u) \neq 0$ then
20:         // Optimally allocate $q(u)$ to $v \in \Pi(u)$
21:         $(q(v))_{v\in\Pi(u)} \leftarrow$ MIN-MAX($|\Pi(u)|$, $(|S_v|)_{v\in\Pi(v)}$, $q(u)$)
22:         for $v \in \Pi(u)$ do
23:             if $q(v) > 0$ then
24:                 Enqueue($Q, v$)
25:             end if
26:         end for
27:     end if
28:     Dequeue(Q)
29: end while
30: return $G$
```

Lemma 3.10: The *while* loop (lines 10 through 29) in Algorithm 1 takes $O(n \log n)$ elementary operations.

Proof: We will show that the worst-case is achieved when the root of $\hat{\mathcal{T}}$ has $O(n)$ children, in which case, the total number of elementary operations is $O(n \log n)$. We first argue that, in each entry into the *while* loop, we only need to consider the running time of the MIN-MAX algorithm called in line 21. Line 16 may consist of more than one elementary operations. However, since each operation adds a node to the set G, the total number of elementary operations by the completion of the *while* loop cannot be more than the size of G, which is at most n, multiplied by a constant factor. Similarly, the *for* loop in lines 22 through 26 cannot take more than $O(n)$ elementary operations over all entries into the *while* loop, because this part of the code simply visits all nodes in $SLCA(S)$ one at a time.

Let $T(n, l)$ be the number of elementary operations in the worst case, when the candidate set S has n nodes, and the tree $\hat{\mathcal{T}}$ has l interior nodes (i.e., non-leaf nodes), denoted by v_1, …, v_l. The total number of nodes in $\hat{\mathcal{T}}$ is no greater than $n-1$, and $l \leq n-2$. Suppose v_i has n_i children in the tree $\hat{\mathcal{T}}$, for $i = 1$, …, l. Then, $\sum_{i=1}^{l} n_i$ is equal to the number of nodes in $\hat{\mathcal{T}}$ minus 1, which is no greater than $n-2$. The MIN-MAX algorithm will be called

l times, one for each v_i, and the number of elementary operations taken is $n_i \log n_i$ for v_i. Consider the problem of $\max \sum_{i=1}^{l} n_i \log n_i$ subject to the constraint $\sum_{i=1}^{l} n_i \leq n-2$. The maximum is achieved at $n_i = (n-2)/l$, assuming $(n-2)/l$ is an integer, and the maximum is $(n-2)\log((n-2)/l)$. We do not care whether such a tree $\hat{\mathcal{T}}$ with $n-1$ nodes, l interior nodes, each interior node with $(n-2)/l$ children exists. The useful result is

$$T(n,l) \leq (n-2)\log((n-2)/l). \tag{2}$$

The maximum of the right hand side is achieved at $l = 1$. We get $T(n,l) \leq (n-2)\log(n-2)$. Let $T(n)$ be the number of elementary operations in the worst case, when the candidate set S has n nodes. We have $T(n) \leq \max_l T(n,l) \leq (n-2)\log(n-2)$. The total number of elementary operations for a tree that can exist is of $O(n \log n)$. ■

We state without proof that the upper bound $O(n \log n)$ can be achieved.

Finally,

Theorem 3.11: The running time for Algorithm 1 is $O(nm^2)$ in theory, and $O(nm)$ in practice.

Proof: By Lemma 3.8, 3.9, and 3.10, the running time for Algorithm 1 is determined by line 6 or 7, which is $O(nm^2)$ in theory, and $O(nm)$ in practice. ■

IV. Evaluation

In this section, we present simulation results demonstrating the benefits of the WLS-minimizing server selection scheme. We compare four selection algorithms, the *random* scheme where the client chooses k servers uniformly at random from the list of candidate nodes, the *closest* scheme where the client chooses k servers with the shortest RTTs from the candidate pool, the DOI-minimizing scheme that generates an optimal server set with respect to the degree of interference (DOI) criterion, and the WLS-minimizing scheme (Algorithm 1). The random scheme and the closest scheme are the most typical strategies in the related works ([26], [40], [27], [31], [9], [41], [7], [33], [42]). We use three performance metrics to evaluate the server selection schemes.

- **Worst case link stress**: We measure the stress of each link by counting the number of downloading streams on the link and report the worst case link stress.
- **Degree of Interference (DOI)**: the DOI of the selected servers.
- **Average path length**: The average is taken over the paths from the selected servers to the client.

We now elaborate on the DOI performance measure.

Definition 6: Suppose $S = \{s_1, \ldots, s_n\}$ is a set of nodes in $\mathcal{T}$ and $E = \{e_1, \ldots, e_l\}$ is the set of edges used by the S-paths. The **degree of interference** (DOI) of nodes $s_1, \ldots, s_n$, denoted by $d(S)$ or $d(s_1, \ldots, s_n)$, is

$$\begin{aligned} d(s_1, \ldots, s_n) &= \sum_{i=1}^{l} (\text{number of } S\text{-paths via } e_i - 1) \\ &= n(\text{average path length}) - l \end{aligned}$$

For example in Fig. 1, if $u = 01100$, $v = 01101$ and $w = 01010$, then $d(u,v,w) = 3$ because the number of S-paths on edge (01100, 01000) is 2 and on edge (01000, 00000) is 3.

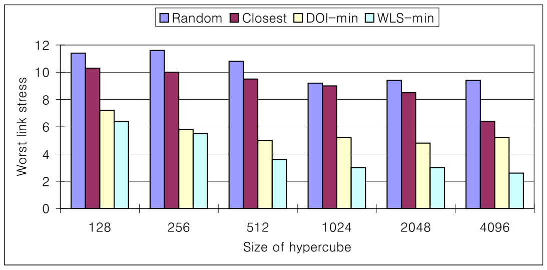

Fig. 3. The worst-case link stress at the overlay network

The following gives a justification for this measure. Suppose the servers in S each transmit a data stream to the client (the root). Let the base case for comparison be that every edge involved in the parallel download session sees exactly one downloading stream, which would be the case if the paths from the servers to the root are all disjoint. The DOI measures the difference between the total number of streams seen by all edges and the base case. From a slightly different viewpoint, suppose there is one unit of cost associated with a stream traversing an edge. The DOI is the difference between the actual total cost and the cost of the base case. By its definition, the DOI is equal to the sum of the path lengths from all nodes in S, minus the total number of edges used. It tends to be small when the average path length, hence the average path delay, is small. Since the aggregate bandwidth used by all connections in a downloading session is proportional to the average path length, the DOI also measures the total bandwidth usage by the session. In this respect, it is similar to the performance measure, *work*, in [30]. The algorithm for minimizing the DOI is presented in our previous work [39], and will not be a focus of the paper. Note that minimizing the DOI is related to but not the same as minimizing the average path length.

A. Single Client Case - Performance on Overlay Network

In this experiment, we compare the goodness of the server selection schemes for a single client. We vary the size of the hypercube from 128 to 4096 nodes and, from each hypercube, a reasonable size (25%) subset of all nodes is chosen as the candidate set. The client selects $k = 20$ servers from the candidate set.

Fig. 3 shows the number of streams on the most stressed edge, i.e., the WLS, for different sizes of the hypercubes when the random scheme, the closest scheme, the DOI-minimizing scheme, and the WLS-minimizing scheme are used, respectively. In all cases, the WLS-minimizing scheme clearly generates the best results, and the random scheme performs the worst. The WLS of the random scheme can be four times as much as that of the WLS-minimizing scheme. The WLS-minimizing scheme and the DOI-minimizing scheme are effective in reducing the bottleneck stress. We observe that the closest scheme isn't significantly better than the random scheme. The WLS shows no clear trend as the hypercube size increases under the random scheme, the closest scheme, and the DOI-minimizing scheme. It decreases under the WLS-minimizing scheme due to the decrease in the density of the servers in the candidate set.

Fig. 4 shows the DOI of all servers for different sizes of the overlay networks. As we would expect, the DOI-minimizing scheme has produced the best results over all the cases. It is interesting to notice that the WLS-

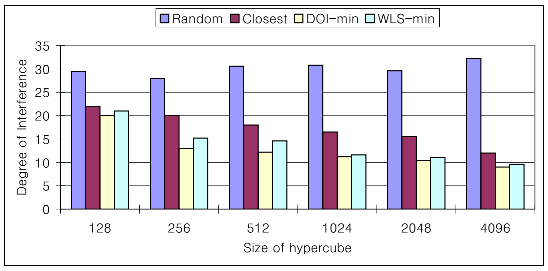

Fig. 4. The degree of interference of the servers at the overlay network

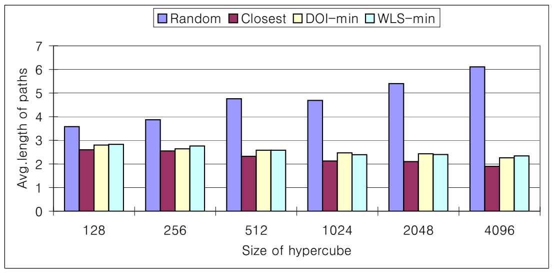

Fig. 5. The average length of the paths from the servers to the client at the overlay network

minimizing scheme generates only slightly higher DOI than the DOI-minimizing scheme. The random scheme has up to three times as much DOI as the DOI-minimizing scheme or the WLS-minimizing scheme.

In Fig. 5, we compare the average length of the paths from the selected servers to the client. The optimal schemes (the closest scheme, the DOI-minimizing scheme, and the WLS-minimizing scheme) offer much shorter average path length than the random scheme. For example, when the size of hypercube is 4096, the average path length can be reduced from 6.11 to 2.3, if we use the WLS-minimizing scheme rather than the random scheme. This means that the optimal schemes can perform better than the random scheme in terms of response time due to shorter round trip time (RTT), particularly true when the overlay hypercube network is locality aware. As noted in our literature review in Section I, downloading from nearby servers is an objective sought by many existing content distribution systems. In addition, since the number of paths is constant, the optimal schemes use much fewer links, hence, less network resource, than the random scheme.

B. Single Client Case - Performance on Physical Network

Similar simulations were also performed on the Transit-Stub model [43] using a 2-level hierarchy of routing domains with transit domains that interconnect lower level stub domains. The simulations run on network topologies consisting of 4200 nodes split into 10 Autonomous Systems (AS). The average diameter of the network is 10.5. We vary the size of the overlay networks from 128 to 4096 nodes and, from each network, a reasonable size (25%) subset of all nodes is chosen as the candidate set. The candidate nodes are uniformly distributed over the 4200 nodes. The client selects $k = 20$ servers from the candidate set. The simulator counts the number of streams on each *physical* link and assigns a constant delay to each link. It does not model either queuing delay or packet losses. The

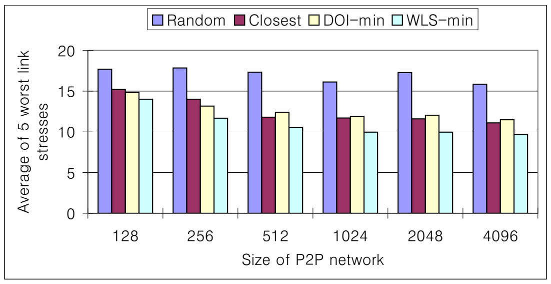

Fig. 6. The average of 5 worst physical link stresses

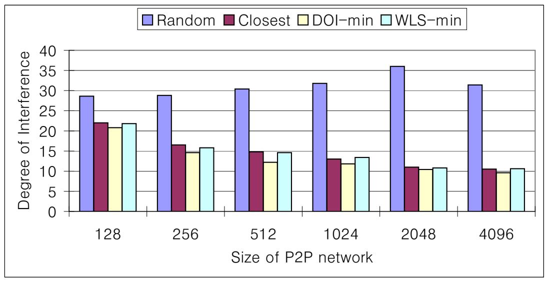

Fig. 7. The degree of interference of the servers

path length is measured in terms of the number of *physical* links. We run each experiment with different random number generator seeds and present the average of the results obtained.

Fig. 6 shows the average number of streams on the five most stressed physical links, for different sizes of the overlay networks when the random scheme, the closest scheme, the DOI-minimizing scheme, and WLS-minimizing scheme are used, respectively. In all cases, the WLS-minimizing scheme generates the best results, and the random scheme performs the worst. Improvement of up to 38% was observed, if we use the WLS-minimizing scheme rather than the random scheme.

Fig. 7 shows the DOI on the physical links for different sizes of the overlay networks. The DOI-minimizing scheme has produced the best results over all the cases. It is noticeable that the WLS-minimizing scheme and the closest scheme generate only slightly higher DOI than the DOI-minimizing scheme. The random scheme has up to three times as much DOI as the other three schemes.

Fig. 8 shows the average length of the paths from the selected servers to the client for different sizes of the overlay networks. The optimal schemes performs significantly better than the random scheme. As the size of overlay networks increases, the average length of paths increases under the random scheme, but, the optimal schemes show no such trend. The optimal schemes use much fewer links, hence, less network resource, than the random scheme.

C. Multiple Client Case - Performance on Overlay Network

In this experiment, the overlay network has 4096 nodes, a quarter of which are candidate nodes. There are 100 clients, each selecting 20 servers from the shared candidate set. The candidate nodes and the clients are both

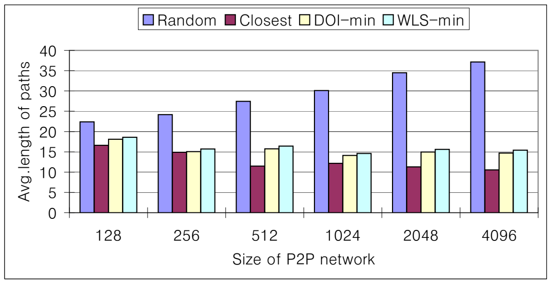

Fig. 8. The average length of the paths from the servers to the client

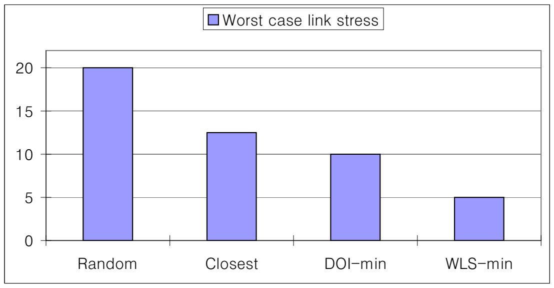

Fig. 9. The worst case link stress for the case of 100 clients at the overlay network

uniformly distributed over the 4096 nodes. We use a heuristic algorithm in which each client independently applies the optimal schemes.

Fig. 9 depicts the number of streams on the most stressed edge for the four selection schemes. The WLS-minimizing scheme still achieves the best result (5), as compared to the DOI-minimizing scheme (10), the closest scheme (12.5) and the random scheme (20). The WLS-minimizing scheme is still very effective at reducing the stress at the bottleneck link for the multiple-client case.

Fig. 10 shows the total DOI of all connections between each and every client and its selected servers. The DOI-minimizing scheme yields the lowest DOI, much lower than the random scheme, and slightly lower than the WLS-minimizing scheme and the closest scheme.

Fig. 11 compares the average path length of the connections between each and every client and its selected servers.

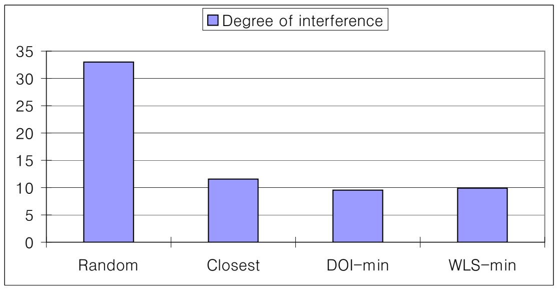

Fig. 10. The degree of interference of all connections for the case of 100 clients

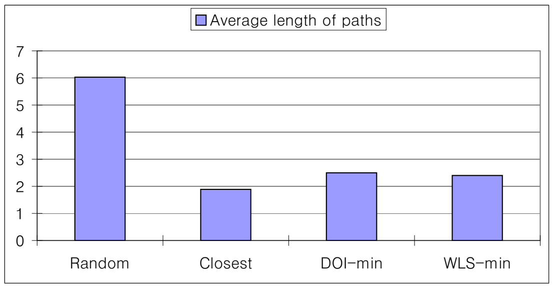

Fig. 11. The average path length of connections for the case of 100 clients at the overlay network

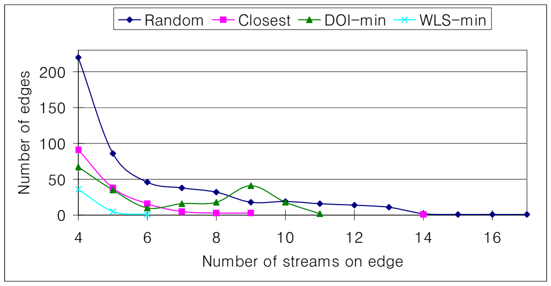

Fig. 12. Distribution of link stress for the case of 100 clients at the overlay network

The average path lengths of the closest scheme, the DOI-minimizing scheme and the WLS-minimizing scheme are 1.9, 2.5 and 2.4, respectively, and that of the random scheme is 6.03. The former three schemes still perform much better than the random scheme, leading to lower response time and less network resource requirement.

Fig. 12 plots the distribution of the link stress in the case of 100 clients. Note that, for every stress level, the WLS-minimizing scheme has the fewest edges at that level than the other schemes, and that the curve also has shorter tail.This suggests that, if we identify “load” with the number of streams on an edge, the WLS-minimizing scheme is the best from the network load-balancing point of view.

D. Multiple Client Case - Performance on Physical Network

The simulation results on the Transit-Stub model are reported in Fig. 13, Fig. 14 and 15. Fig. 13 depicts the average number of streams on the five most stressed physical links. It can be reduced by up to 51% if we use the WLS-minimizing scheme rather than the random scheme. Fig. 14 shows the total DOI of all connections between each and every client and its selected servers. Fig. 15 shows the average path length of the connections. It is noticeable that the general trends on the physical links are the same as the overlay cases. We also observed that the distribution of physical link stress shows the same trends as in Fig. 12.

E. Distributed Implementation and Scalability

Low computation complexity is only one of the factors for successful adaptation of the server selection algorithms to large networks. The scalability of the algorithms in a distributed system is often dominated by communication

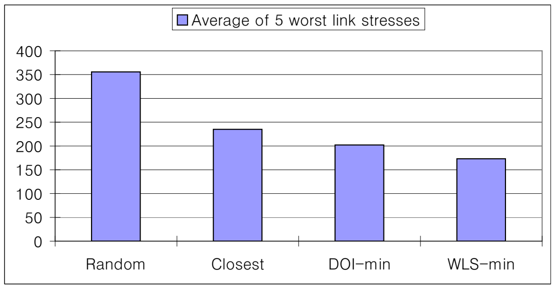

Fig. 13. The average of 5 worst physical link stresses for the case of 100 clients

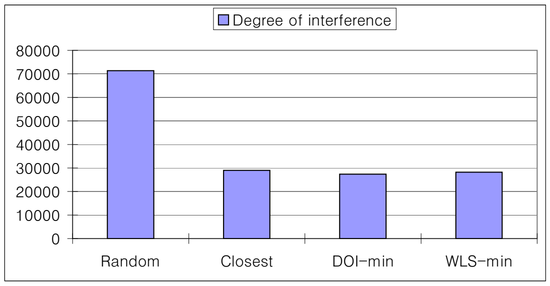

Fig. 14. The degree of interference of all connections for the case of 100 clients

overhead and constraints rather than computation complexity. In this section, we will address such issues in distributed implementation.

First, we would like to point out that the proposed algorithm is friendly to distributed implementation, because only the IDs of the candidate nodes are needed at each client. This important fact is due to the known structure of the hypercube network. The problem of distributing the candidate list is similar to many relatively easy, classical information dissemination problems in networking. For instance, we may rely on well-known servers that maintain the candidate list, on limited flooding of the candidate list or the queries for the list, on random walk type of information dissemination, or on distributed hash table based information retrieval.

In a large network with a large candidate server set, it may be impractical for each client to receive the entire

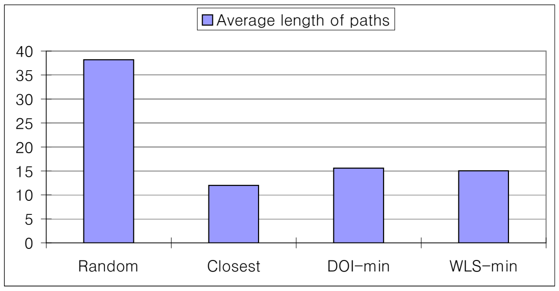

Fig. 15. The average path length of connections for the case of 100 clients

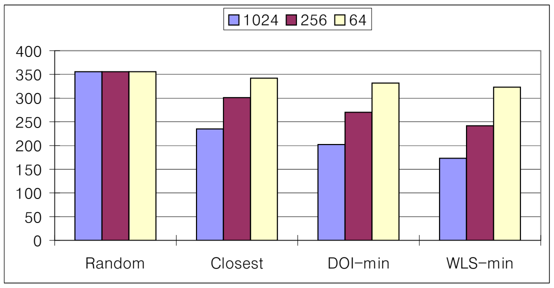

Fig. 16. The average of 5 worst physical link stresses for the case of 100 clients. The entire candidate set has 1024 nodes. The three columns for each server selection scheme correspond to the cases where each client has knowledge of 1024, 256 or 64 candidates.

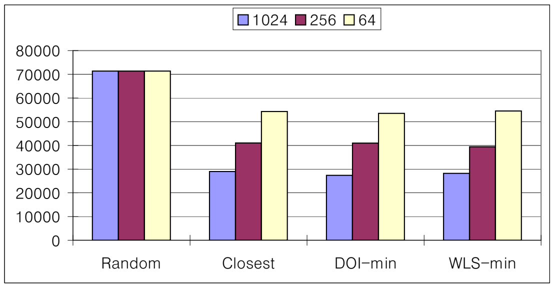

Fig. 17. The degree of interference of all connections for the case of 100 clients. The entire candidate set has 1024 nodes. The three columns for each server selection scheme correspond to the cases where each client has knowledge of 1024, 256 or 64 candidates.

list of candidates in time, particularly when such a set changes quickly overtime. We will show that the proposed server selection algorithm still yields significant performance improvement when only a (different) subset of the candidates is visible to each client.

In the experiment, we assume the same Transit-Stub model as in previous sections, and the overlay network has 4096 nodes, 1024 candidate nodes, 100 clients each selecting 20 servers. All performance metrics are with respect to the underlay physical network. We compare three situations where each client has knowledge of all 1024 candidates, or 256, or 64 candidates. In the case where each client sees only a subset of the candidates, say 256 of them, the partial set is randomly chosen from the entire candidate set.

Fig. 16 depicts the average number of streams on the five most stressed physical links for the four selection schemes. Regardless the size of the candidate set for each client, the random scheme performs equally bad. For the optimal schemes, the link stress decreases as each client sees more and more candidates. Even when the size of each visible candidate set, say 256, is significantly smaller than that of the entire candidate set, the reduction in link stress compared with the random scheme is significant.

The simulation results on the DOI and the average path length shown in Fig. 17 and 18 also confirm the above observation. In fact, for those two performance measures, the optimal schemes perform better even with 64 visible candidates for each client.

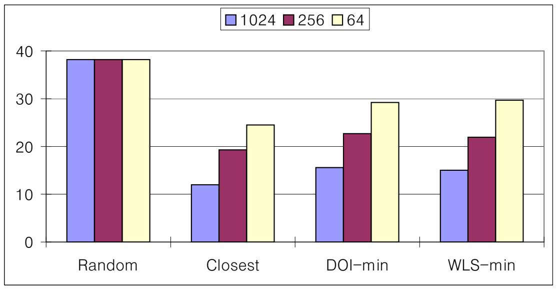

Fig. 18. The average path length of connections for the case of 100 clients. The entire candidate set has 1024 nodes. The three columns for each server selection scheme correspond to the cases where each client has knowledge of 1024, 256 or 64 candidates.

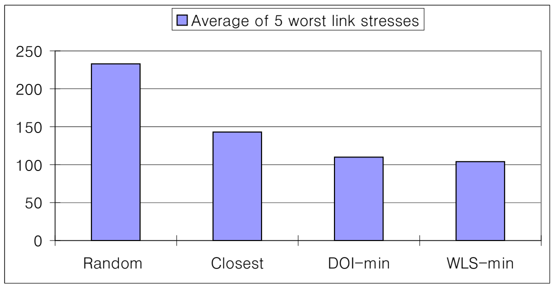

Fig. 19. The average of 5 worst physical link stresses for the case of 100 clients. The overlay network has a name space size 4096 and contains 2048 nodes.

F. Name Space Not Fully Occupied

In this section, we consider the situation where the name space is not fully populated by nodes. This is the most likely scenario for an ad-hoc P2P network with frequent node joins and leaves, but less an issue for a managed content distribution network. We modify the routing as follows. When a message for destination d arrives at a node, the next node will be the first available node on the hypercube path to d. This results in an increase in the routing table size. However, one can substitute routing-table-based routing by on-demand routing. The hypercube path to the destination is probed before the data messages are sent, and the route is either encoded in the messages themselves or cached at the intermediate nodes.

The server selection strategy is, given the list of candidate, each client selects servers as if the network is a hypercube. In our simulation experiments, we again use the same Transit-Stub model as in previous sections. The overlay network has a name space size 4096, but only contains 2048 actual nodes, uniformly distributed over the 4200 nodes. There are 100 clients, each selecting 20 servers from the shared candidate set of 1024 nodes. Fig. 19, 20 and 21 show the resulting worst link stress, DOI and average path length, respectively. Compared with the results for the fully-occupied hypercube in Section IV-C, the performance gain of the optimal schemes has not deteriorated.

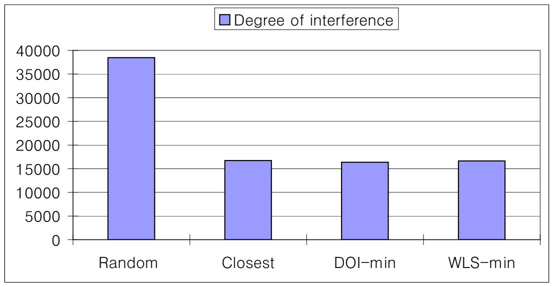

Fig. 20. The degree of interference of all connections for the case of 100 clients. The overlay network has a name space size 4096 and contains 2048 nodes.

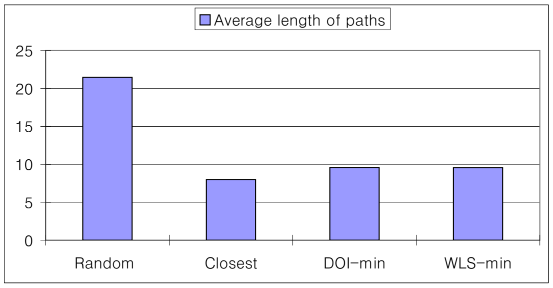

Fig. 21. The average path length of connections for the case of 100 clients. The overlay network has a name space size 4096 and contains 2048 nodes.

V. Conclusions

In this paper, we make an in-depth investigation on the issue of server/peer selection, which is a fundamental problem in parallel data access or collaborative content distribution. We envision a hypercube overlay network, and give a node selection scheme that generates an optimal server set with respect to the worst link stress (WLS) criterion. The optimal scheme does not require the network topology or routing information; nor does it require the measurement of the network performance data. The only assumption is that the client can obtain the list of IDs of the candidate servers through the overlay network. The scheme has a running time of $O(nm^2)$ in theory and $O(nm)$ in practice, where n is the number of nodes in the candidate set and 2^m is the size of the network. The freedom from network measurement and low implementation complexity make the overall scheme scalable. One of the main contributions of the paper is to carefully develop the ideas that make this fast and simple algorithm possible.

We have presented simulation results to demonstrate the benefits of the optimal node selection schemes. We conclude that the WLS-minimizing scheme performs significantly better than the random node selection scheme in all performance measures we evaluated. Compared to two other optimal schemes, it has a noticeable advantage in reducing the WLS, and is very competitive in terms of the average path length against the closet scheme, and in terms of the DOI against the DOI-minimizing scheme.

We summarize the real-world advantages of the WLS-minimizing scheme as follows. First, it minimizes the level of congestion at the bottleneck link. This balanced use of the network leads to higher chance of accommodating other sessions, and ultimately can benefit all sessions. Second, minimizing the WLS can be turned into maximizing the achievable throughput. This both speeds up the individual downloading session and allows the network to accept more sessions, hence, improving the efficiency of network resource utilization. Third, the outcome of WLS-minimization consumes less network resources in terms of the total number of links used and the total bandwidth usage. Fourth, it leads to low average RTT to selected servers, hence, allowing nearby nodes to exchange more data, a desirable feature for overlay-based content distribution.

Our optimal algorithm can also work well for any general network with known topology, at the expense of higher communication overhead for collecting the topology and routing information and computation complexity for manipulating the unstructured graph. Such a general network is more suitable for an ad-hoc distribution system, whereas the hypercube network is more suitable for a managed system. In this paper, when justifying the WLS as a measure of congestion, we assume the overlay links are of similar bandwidth, a reasonable assumption for a managed infrastructure overlay network. Our formulation and algorithm should handle a small degree of heterogeneity well, possibly with minor modification to treat special cases. If the link bandwidth varies considerably, the optimization objective should take into account this heterogeneity, for instance, by considering the normalized WLS against the link bandwidth. A complete solution to this modified problem is worth further investigation.

References

[1] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, “Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications,” in *Proc. ACM SIGCOMM ’2001*, San Diego, CA, August 2001, pp. 149–160.

[2] B. Y. Zhao, J. Kubiatowicz, and A. D. Joseph, “Tapestry: An infrastructure for fault-tolerant wide-area location and routing,” University of California University, Berkeley, Computer Science Division (EECS), Tech. Rep. UCB/CSD-01-1141, April 2001.

[3] A. Rowstron and P. Druschel, “Pastry: scalable, distributed object location and routing for large-scale peer-to-peer systems,” in *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware ’01)*, Heidelberg, Germany, November 2001.

[4] S. Ratnasamy, P. Francis, M. Hanley, R. Karp, and S. Shenker, “A scalable content-addressable network,” in *Proc. ACM SIGCOMM ’2001*, San Diego, CA, August 2001, pp. 161–172.

[5] D. Loguinov, A. Kumar, V. Rai, and S. Ganesh, “Graph-theoretic analysis of structured peer-to-peer systems: Routing distances and fault resilience,” in *Proceedings of ACM SIGCOMM*, Karlsruhe, Germany, August 2003.

[6] A. Kumar, S. Merugu, J. Xu, and X. Yu, “Ulysses: A robust, low-diameter, low-latency peer-to-peer network,” in *Proceedings of the 11th IEEE International Conference on Network Protocols (ICNP03)*, Atlanta, Georgia, USA, Nov. 2003.

[7] P. Rodriguez, A. Kirpal, and E. Biersack, “Parallel-access for mirror sites in the internet,” in *Proceedings of IEEE Infocom*, Tel Aviv, Israel, March 2000.

[8] P. Rodriguez and E. Biersack, “Dynamic Parallel Access to Replicated Content in the Internet,” *IEEE/ACM Transactions on Networking*, vol. 10, no. 3, August 2002.

[9] M. Hefeeda, A. Habib, B. Boyan, D. Xu, and B. Bhargava, “Promise: peer-to-peer media streaming using collectcast,” in *Proc. of ACM Multimedia*, Berkeley, CA, November 2003.

[10] BitTorrent Website, http://www.bittorrent.com/.

[11] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. Kubiatowicz, “Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination,” in *Proceedings of the 11th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)* , June 2001.

[12] M. C. Castro, M. B. Jones, A.-M. Kermarrec, A. Rowstron, M. Theimer, H. Wang, and A. Wolman, “An evaluation of scalable application-level multicast built using peer-to-peer overlays,” in *Proceedings of IEEE Infocom*, San Francisco, April 2003.

[13] S. Shenker, L. Peterson, and J. Turner, “Overcoming the internet impasse through virtualization,” in *ACM Hot Nets*, 2004.

[14] S. L. Johnsson and C.-T. Ho, “Optimum broadcasting and personalized communications in hypercubes,” *IEEE Transactions on Computers*, vol. 38, no. 9, 1989.

[15] S. S. Gupta, D. Das, and B. P. Sinha, “The generalized hypercube-connected-cycle: An efficient network topology,” in *Third International Conference on High-Performance Computing (HiPC ’96)*, Trivandrum, India, December 1996.

[16] J.-H. Park, H.-C. Kim, and H.-S. Lim, “Fault-hamiltonicity of hypercube-like interconnection networks,” in *19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Denver, Colorado, April 2005.

[17] O. Beaumont, L. Marchal, and Y. Robert, “Broadcast trees for heterogeneous platforms,” in *19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Denver, Colorado, April 2005.

[18] L.-J. Fan, C.-B. Yang, and S.-H. Shiau, “Routing algorithms on the bus-based hypercube network,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, 2005.

[19] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, 1991.

[20] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, “Tapestry: a resilient global-scale overlay for service deployment,” *IEEE Journal on Selected Areas in Communications*, vol. 22, no. 1, pp. 41–53, Jan. 2004.

[21] D. Kostić, A. Rodriguez, J. Albrecht, and A. Vahdat, “Bullet: high bandwidth data dissemination using an overlay mesh,” in *Proceedings of 19th ACM Symposium on Operating Systems Principles (SOSP ’03)*, October 2003.

[22] C. Plaxton, R. Rajaraman, and A. Richa, “Accessing nearby copies of replicated objects in a distributed environment,” in *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA ’97)*, Newport, Rhod Island, June 1997, pp. 311–320.

[23] Akamai Website, http://www.akamai.com.

[24] M. Castro, P. Druschel, A. Kermarrec, A. Nandi, A. Rowstron, and A. Singh, “Splitstream: High-bandwidth multicast in cooperative environments,” in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP ’03)*, October 2003.

[25] L. Cherkasova and J. Lee, “Fastreplica: Efficient large file distribution within content delivery networks,” in *Proceedings of the 4th USITS*, Seattle, WA, March 2003.

[26] K. Park and V. S. Pai, “Scale and performance in the coblitz large-file distribution service,” in *Proceedings of the 3rd USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 06)*, San Jose, CA, May 2006.

[27] B. Chun, P. Wu, H. Weatherspoon, and J. Kubiatowicz, “Chunkcast: An anycast service for large content distribution,” in *Proceedings of the Internaltional Workshop on Peer-to-Peer Systems (IPTPS)*, February 2006.

[28] D. Kostić, R. Braud, C. Killian, E. Vandekieft, J. W. Anderson, A. C. Snoeren, and A. Vahdat, “Maintaining high bandwidth under dynamic network conditions,” in *Proceedings of USENIX Annual Technical Conference*, 2005.

[29] R. Sherwood, R. Braud, and B. Bhattacharjee, “Slurpie: A cooperative bulk data transfer protocol,” in *Proceedings of IEEE Infocom*, Hong Kong, March 2004.

[30] D. Bickson, D. Malkhi, and D. Rabinowitz, “Efficient large scale content distribution,” in *Proceedings of the 6th Workshop on Distributed Data and Structures (WDAS’2004)*, Lausanne, Switzerland, July 2004.

[31] M. Adler, R. Kumar, K. Ross, D. Rubenstein, T. Suel, and D. Yao, “Optimal peer selection for p2p downloading and streaming,” in *Proceedings of IEEE Infocom*, Miami, FL, March 2005.

[32] R. L. Carter and M. Crovella, “Server selection using dynamic path characterization in wide-area networks,” in *Proceedings of IEEE Infocom*, 1997, pp. 1014–1021.

[33] A. Shaikh, R. Tewari, and M. Agrawal, “On the effectiveness of dns-based server selection,” in *Proceedings of IEEE Infocom*, Anchorage, AK 2001.

[34] A. Ganesh, L. Massoulie, and D. Towsley, “The effect of network topology on the spread of epidemics,” in *Proceedings of IEEE Infocom*, Miami, FL, March 2005.

[35] Z. Tang and J. J. Garcia-Luna-Aceves, “Hop-reservation multiple access (HRMA) for ad-hoc networks,” in *Proceedings of IEEE Infocom*, New York, NY, March 1999.

[36] M. Adler, E. Halperin, V. Vazirani, and R. M. Karp, “A stochastic process on the hypercube with applications to peer-to-peer networks,” in *ACM Symposium on Theory of Computing*, San Diego, CA, June 2003.

[37] S. S. Lan and H. Liu, “Failure recovery for structured P2P networks: protocol design and performance evaluation,” in *ACM SIGMETRICS/Performance 2004*, New York, June 2004.

[38] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. The MIT Press, 2001.

[39] S. C. Han and Y. Xia, “Constructing an optimal server set in structured peer-to-peer network,” *IEEE Journal on Selected Areas in Communications*, vol. 25, no. 1, pp. 170–178, Jan. 2007.

[40] S. Annapureddy, M. J. Freeman, and D. Mazières, “Shark: Scaling file servers via cooperative caching,” in *Proceedings of the 2nd USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 05)*, Boston, MA, May 2005.

[41] A. Miu and E. Shih, “Performance analysis of a dynamic parallel downloading scheme from mirror sites throughout the internet,” 1999, http://nms.lcs.mit.edu/~aklmiu/comet/paraload.html.

[42] A. Zeitoun, H. Jomjoom, and M. El-Gendy, “Scalable parallel-access for mirrored servers,” in *Proceedings of IASTED International Conference on Applied Informatics*, February 2002.

[43] K. Calvert, M. Doar, and E. W. Zegura, “Modeling internet topology,” *IEEE Communications Magazine*, June 1997.