A Lightweight Data Location Service for Nondeterministic Exascale Storage Systems

ZHIWEI SUN and ANTHONY SKJELLUM, The University of Alabama at Birmingham
LEE WARD and MATTHEW L. CURRY, Sandia National Laboratories

In this article, we present LWDLS, a lightweight data location service designed for Exascale storage systems (storage systems with order of 10^{18} bytes) and geo-distributed storage systems (large storage systems with physically distributed locations). LWDLS provides a search-based data location solution, and enables free data placement, movement, and replication. In LWDLS, probe and prune protocols are introduced that reduce topology mismatch, and a heuristic flooding search algorithm (HFS) is presented that achieves higher search efficiency than pure flooding search while having comparable search speed and coverage to the pure flooding search. LWDLS is lightweight and scalable in terms of incorporating low overhead, high search efficiency, no global state, and avoiding periodic messages. LWDLS is fully distributed and can be used in nondeterministic storage systems and in deterministic storage systems to deal with cases where search is needed. Extensive simulations modeling large-scale High Performance Computing (HPC) storage environments provide representative performance outcomes. Performance is evaluated by metrics including search scope, search efficiency, and average neighbor distance. Results show that LWDLS is able to locate data efficiently with low cost of state maintenance in arbitrary network environments. Through these simulations, we demonstrate the effectiveness of protocols and search algorithm of LWDLS.

Categories and Subject Descriptors: C.2.2 [**Computer-Communication Networks**]: Network Protocols

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Efficient search, Exascale, nondeterministic, scalability, storage

ACM Reference Format:
Zhiwei Sun, Anthony Skjellum, Lee Ward, and Matthew L. Curry. 2014. A lightweight data location service for nondeterministic exascale storage systems. ACM Trans. Storage 10, 3, Article 12 (July 2014), 22 pages.
DOI: http://dx.doi.org/10.1145/2629451

1. INTRODUCTION

According to predicted hardware characteristics and input-output (I/O) throughput of Exascale systems by 2018, Exascale storage systems will need to support I/O at the level of more than 60TB/s [Dongarra 2010]. To meet the storage demands including speed, capacity, and cost, both flash storage devices and traditional disks will be used to meet the required I/O throughput.

As storage systems scale to Exascale, storage environments will become more heterogeneous and dynamic than today's High Performance Computing (HPC) storage systems because of system scale, the use of various storage components, and the level of parallelism required by Exascale computing. Reducing limitations on data placement

This work was supported in part by the United States Department of Energy under Contract DE-AC04-94AL85000 and by the National Science Foundation under grant OCI-1064247 and grant CCF-1239962. Sandia National Laboratories is a multiprogram laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.
Author's addresses: Z. Sun and A. Skjellum, Department of Computer and Information Sciences, The University of Alabama at Birmingham; L. Ward and M. L. Curry, Sandia National Laboratories. Correspondence email: zhwsun@uab.edu.

and replication will possibly improve I/O throughput. Some nondeterministic storage systems [Nowoczynski et al. 2008; Curry et al. 2012] have already been designed with the goal of facilitating writes. However, locating data in nondeterministic storage systems is challenging. The traditional centralized approach has limited scalability and has difficulty supporting at the level of parallelism required by Exascale systems. Deterministic strategies for both data placement and data location have been used in some HPC file systems [Yang et al. 2004; Weil et al. 2006b]. Furthermore, global state is commonly used for achieving high performance [Tang and Yang 2003; Weil et al. 2006a]. These methods are efficient but limit data placement.

In this article, we present a lightweight data location service (LWDLS) for Exascale storage systems (storage systems with order of 10^{18} bytes) and geo-distributed storage systems (large storage systems with physically distributed locations). The main contributions of this work are as follows.

(1) We present a new approach for locating data in Exascale storage systems. LWDLS provides a search-based data location method and enables arbitrary data placement.
(2) LWDLS is directly applicable to nondeterministic storage systems [Nowoczynski et al. 2008; Curry et al. 2012] designed for Exascale computing and other storage systems for finding data, and thereby can enable reads in systems like Zest [Nowoczynski et al. 2008].
(3) LWDLS addresses two problems together: topology mismatch and inefficient search performance of the pure flooding search [Jiang et al. 2008]. In LWDLS, probe and prune protocols are designed to reduce topology mismatch, and a new heuristic flooding search algorithm provides higher search efficiency than the pure flooding search algorithm [Jiang et al. 2008] while having comparable search speed and search coverage.
(4) LWDLS is lightweight and scalable in terms of low overhead, high search efficiency, no global state, and avoiding periodic messages.
(5) LWDLS can be applied in geo-distributed systems to improve I/O throughput by taking advantage of locality.

2. THE PROBLEM

Locating data in Exascale storage systems (and other large-scale storage systems) is fundamental but challenging. It affects many aspects of Exascale systems, such as performance, scalability, reliability, and management. To meet the storage demands of Exascale computing by the end of this decade, some nondeterministic storage systems [Nowoczynski et al. 2008; Curry et al. 2012] have already been designed with the goal of facilitating writes. Nondeterministic storage systems can provide certain useful capabilities that are difficult to implement in today's parallel file systems, such as high-performance writes, dynamic load-balancing, enriched flexibility for moving data and creating replicas, and potentially other advantages. However, reading data from nondeterministic storage systems proves challenging, because of problems such as how to find data, how to deal with the consistency among replicas, and others. For instance, in Zest [Nowoczynski et al. 2008], nondeterministic data placement has been used to facilitate peak media speeds for application checkpoints, but Zest [Nowoczynski et al. 2008] does not support reads. The data segments residing in Zest [Nowoczynski et al. 2008] need to be copied into the parallel file system [Bent et al. 2009] before clients can read them back. Solutions that enable reads in nondeterministic storage systems without the need for reorganizing data, or that at least allow clients to find the data before the process of relocating data is finished (like in Zest) will be useful.

In Peer-to-Peer (P2P) systems, most search-based algorithms are insufficient for nondeterministic HPC storage systems. A search-based solution needs to address two problems effectively together. The first problem is topology mismatch between physical network topology and overlay network topology. Since the overlay network does not necessarily reflect the physical network underneath, communication efficiency can be reduced by long-distance communications and by sending the same message on a single physical network path multiple times. The second problem is search performance. Pure flooding search [Jiang et al. 2008] is a simple, reliable, and fast method for finding data, but it is inefficient and unscalable because of its high cost of redundant messages generated during search. Other search methods that are based on random graph theory, such as random walks [Pearson 1905; Lv et al. 2002], and probabilistic flooding search methods [Crisóstomo et al. 2012; Gaeta and Sereno 2011], require maintaining the statistical properties of overlay network graph always in order to find data with the given search success rate. In LWDLS, we work to avoid this strong requirement on Exascale storage systems, and also retain the ability of optimizing the overlay network based on the network latency or others factors. Furthermore, these probabilistic methods trade response time against search efficiency, but HPC applications have a strong requirement for response time minimization.

In this article, we present LWDLS, a new approach for locating data in large-scale HPC storage systems. Features of LWDLS include providing a search-based data location service, enabling arbitrary data placement, avoiding global state or periodic messages, being able to reduce topology mismatch, low-cost of state maintenance, fast search speed, and high search efficiency.

3. RELATED WORK

We discuss several areas of research that are closely related to this work, including file systems and peer-to-peer (P2P) systems.

3.1. File Systems

In many existing HPC storage systems, the location of data is explicitly stored and managed by centralized servers in the form of a metadata directory, such as in GPFS [Schmuck and Haskin 2002], Lustre [Schwan 2003], zFS [Rodeh and Teperman 2003], PVFS [Carns et al. 2000], GoogleFS [Ghemawat et al. 2003], and others. This approach requires the involvement of centralized servers in cases of data placement, movement, or replication. Therefore, it is difficult to support the billion-way parallelism required by Exascale systems.

In some other file systems, including Sorrento [Yang et al. 2004] and Ceph [Weil et al. 2006b], deterministic mapping algorithms [Tang and Yang 2003; Weil et al. 2006a] from data object IDs to storage server IDs are designed for data placement and location. The Sorrento file system [Yang et al. 2004] enables freely moving data and replicas in the system by using a deterministic method to locate the metadata that contains the location information of data. In the event of relocating data, the residence information can be found and updated by any server in the system. In our design of LWDLS, we want a fully nondeterministic solution with less state and lower overhead of state maintenance. In Ceph [Weil et al. 2006b], the CRUSH [Weil et al. 2006a] algorithm is used to find data in a deterministic manner, and to distribute data uniformly among servers in the system for load balance. Generally, these deterministic data location methods provide better scalability than the centralized methods, and are able to find data directly and efficiently without the need for an explicit layout lookup. However, the deterministic data placement mechanism does not provide the ability to load balance dynamically or to use preferentially the storage that is “close” to applications as in nondeterministic storage systems.

Some nondeterministic storage systems, such as Zest [Nowoczynski et al. 2008] and Sirocco [Curry et al. 2012] have been designed to support Exascale computing with the goal of allowing computing nodes to write their buffers into storage systems as fast as possible by eliminating limitations on data placement. Zest [Nowoczynski et al. 2008] has been used to facilitate application checkpoints, but it does not support reads. To read data back, Zest [Nowoczynski et al. 2008] requires copying its data into a full-featured parallel file system [Bent et al. 2009] so that data can be found by using the data location method of the parallel file system [Bent et al. 2009]. In the meantime, clients cannot read the data residing in Zest [Nowoczynski et al. 2008].

LWDLS provides a direct, lightweight solution to enable nondeterministic storage systems such as Zest [Nowoczynski et al. 2008] and Sirocco [Curry et al. 2012] to find data without relocating data or waiting for completion of data relocation to new places. Furthermore, LWDLS imposes zero-cost of state maintenance when the location service is not used.

3.2. Peer-to-Peer Systems

Peer-to-Peer (P2P) systems are built on overlay networks that are imposed on top of physical networks. According to the types of overlay networks, they can be categorized into unstructured, structured, and hybrid P2P systems [Buford 2013].

In unstructured P2P systems, search methods mainly include flooding search and random walks [Buford 2013]. Pure flooding search [Jiang et al. 2008] is a simple, fast, and reliable method, but is not scalable because of the cost of redundant messages generated during search. To reduce redundant messages, different flooding search schemes [Jiang et al. 2008; Gkantsidis et al. 2005; Lv et al. 2002; Lin et al. 2009], are designed to reduce the cost of search. In some methods [Chawathe et al. 2003], replication and the properties of network topologies are utilized to improve search performance. However, these methods do not address the topology mismatch problem. LightFlooding [Jiang et al. 2008] requires maintaining additional information, such as 2-hop neighborhoods and tree-like structures on every node. LWDLS avoids these strong requirements and the overhead of state maintenance.

Probabilistic flooding search algorithms [Crisóstomo et al. 2012; Gaeta and Sereno 2011; Oikonomou et al. 2010] reduce the number of messages needed by only sending messages to a subset of neighbors on each step that are selected based on probabilistic functions [Stauffer and Barbosa 2004]. Probabilistic flooding searches trade search response time against search efficiency. Furthermore, these algorithms are based on random graph models where the lengths of edges are assumed to be equal, and they require the networks satisfy particular statistical properties [Newman et al. 2001; Stauffer and Barbosa 2004] in order to achieve the desired rate of search hits. By way of contrast, LWDLS is designed to retain the helpful properties of flooding search while providing higher search efficiency. LWDLS is also able to reduce topology mismatch.

Random Walk [Pearson 1905; Lv et al. 2002] chooses one neighbor to send a message at each hop along the search path. In K-random walks [Lv et al. 2002], K walkers are used to search in parallel. Like probabilistic flooding search, the search speed of random walks is much slower than pure flooding search [Jiang et al. 2008], although they have better search efficiency.

For the topology mismatch problem, LTM [Liu et al. 2005] and THANKS [Liu 2008] provided simple and efficient probe and cut mechanisms that are similar to the probe and prune protocols of LWDLS, but there are important differences. One distinct difference is that LTM [Liu et al. 2005] and THANKS [Liu 2008] try to break all of cycles formed within the two-hop neighborhood by removing the largest edge of a triangle to improve search efficiency. However, LWDLS tries to remove redundant long-distance communications while allowing the physically closer servers to form neighbors. The triangles formed by the physically close servers are kept and used to improve both search response time and search efficiency. Furthermore, LTM [Liu et al. 2005] requires the clocks of servers be accurately synchronized, and requires periodically sending messages among the 2-hop neighbors by each server in the system. However, LWDLS tries to avoid these. THANKS [Liu 2008] is similar to LTM [Liu et al. 2005], except it kept the distances of 2-hop neighbors updated by piggybacking neighbor distances. LTM [Liu et al. 2005] and THANKS [Liu 2008] used pure flooding search [Jiang et al. 2008], but in LWDLS, a heuristic flooding search algorithm is designed for running on the optimized overlay network.

In structured P2P systems, Distributed Hash Tables (DHTs) are designed to improve search performance by using structured overlay networks. Examples of DHTs are Chord [Stoica et al. 2001], Pastry [Rowstron and Druschel 2001], Kademlia [Maymounkov and Mazières 2002], CAN [Ratnasamy et al. 2001], Tapestry [Zhao et al. 2001], Cycloid [Shen et al. 2006], Ketama [Ketama 2013], Memcached [Memcached 2013], Dynamo [DeCandia et al. 2007], Cassandra [Lakshman and Malik 2010], ZHT [Brandstatter et al. 2013], and others. Generally, DHTs provide bounded worst-case search performance. Most DHTs are multi-hop DHTs, such as Chord [Stoica et al. 2001], Pastry [Rowstron and Druschel 2001], which take more than one-hop to locate data. Dynamo [DeCandia et al. 2007], Cassandra [Lakshman and Malik 2010], One hop DHT [Gupta et al. 2003], Memcached [Memcached 2013], and ZHT [Brandstatter et al. 2013], are 1-hop or also called zero-hop DHTs. By taking advantage of global state, given a request, each server in the system is able to determine the location of data directly.

DHTs require maintaining structured overlay networks using periodic messages. As discussed previously, Sorrento [Yang et al. 2004] has provided an example of enabling nondeterministic data placement by using the DHT technique to find the residency information of data efficiently. However, we are more interested in a completely nondeterministic solution, so that both the need for maintaining the structured overlay network and the need for updating the residency information of data can be eliminated.

In hybrid P2P systems [Yang and Yang 2010; Loo et al. 2004], some functionality is still centralized, although such systems require fewer state than structured overlay networks. Generally, we want a fully distributed solution for system scalability.

4. APPROACH

LWDLS is a fully distributed, P2P-technology-inspired solution. It provides a search-based service for locating data in large-scale HPC storage systems such as Exascale storage systems. In a target system, given the membership information each server possesses, an overlay network can be constructed upon which the location service of LWDLS is performed. However, searching overlay networks reveals two immediate problems. The first problem is topology mismatch between the physical network and the overlay network, which reduces communication efficiency. The second problem is the overall search performance including search speed, efficiency, and scalability, must meet the requirements of HPC applications.

To address these two problems, in LWDLS, we designed probe and prune protocols that reduce topology mismatch with the support of network distance measurements; in addition, we designed a heuristic flooding search (HFS) algorithm that has comparable search speed and coverage to the pure flooding search algorithm [Jiang et al. 2008] but is more efficient overall. Furthermore, HFS can be used independently without adapting or altering the overlay network, or can be combined with probe and prune protocols to optimize the overlay network during search process.

In this article, we primarily describe the functional properties of LWDLS and performance implications, but we do not cover operational aspects such as recovery from

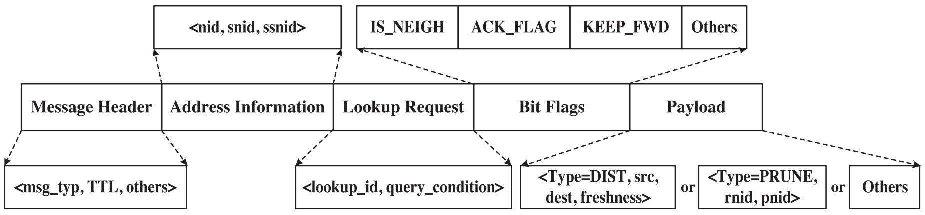


Fig. 1. Definition of a lookup message in LWDLS.

hardware and network faults. The remainder of this article is organized as follows: Section 4.1 describes the assumptions and terminology used; Section 4.2 offers the definition of lookup messages used by the protocols and the LWDLS algorithm to implement its services; Section 4.3 introduces the types of server lists used for building virtual overlay networks needed for search; Section 4.4 describes the probe and prune protocols; Section 4.5 describes a heuristic flooding search algorithm; Section 5 presents the experiments and results; finally, Section 6 summarizes this work and suggests additional future work that builds on our results presented here.

4.1. Assumptions and Definition of Terminology

Before introducing the protocols and search algorithm of LWDLS, we describe the assumptions we made in our designs and the terminology that has been used throughout this article.

4.1.1. The Overlay Network. Following are the assumptions made in our design. First, in the target systems, we assume that each server has some membership information, but the membership information may not be updated or be globally consistent (i.e., not global state). Second, the overlay network graph constructed based on the membership information must be a connected graph. LWDLS does not control the forming or the connectivity of the initial overlay network, while it does maintain the connectivity of the overlay network during search process in a fault-free environment. Third, the overlay network graph may not necessarily have certain structure [Stoica et al. 2001; Jiang et al. 2008], have particular statistical properties [Newman et al. 2001; Pearson 1905] such as on random graphs, or perfectly match the underlying physical network topology.

In P2P systems, the servers that are connected on an overlay network are called "neighbors," and the number of neighbors each server has is usually called the "degree." We use these terms throughout this article.

4.1.2. Nondeterministic Storage. In a nondeterministic storage system, given a lookup request, the target data can be stored anywhere in the system. Furthermore, data can be placed on or be moved to any server in the system with few limitations. Zest [Nowoczynski et al. 2008] and Sirocco [Curry et al. 2012] are two examples of nondeterministic storage systems. In both these systems, search is useful but might only be needed occasionally. To be applicable to these systems, LWDLS was designed to be as lightweight as possible. It only uses lookup messages to search for data and to optimize overlay network. When search is not called, LWDLS imposes zero-cost for state maintenance.

4.2. Definition of Lookup Messages in LWDLS

Figure 1 shows the content of a lookup message. Three server IDs, IDs of the requesting server (*nid*), sender (*snid*), and the previous sender of sender (*ssnid*), are encoded in every lookup message. Server IDs are unique in the system. In the current

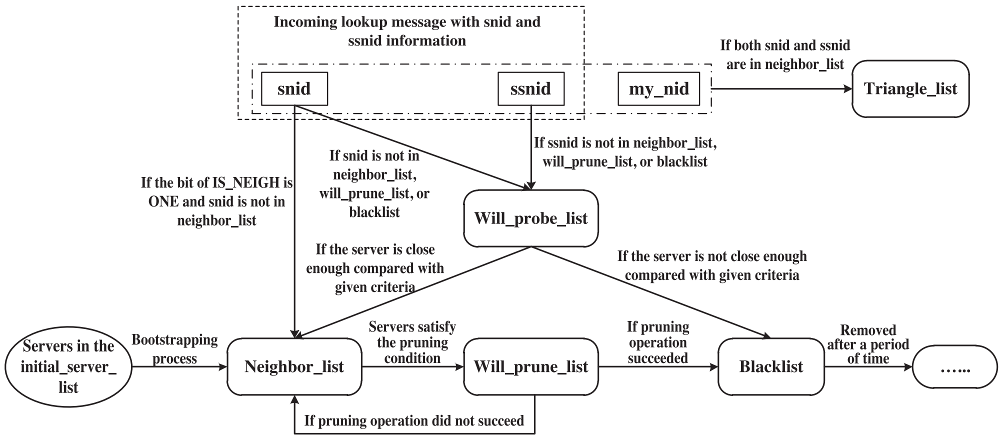

Incoming lookup message with snid and ssnid information
snid
ssnid
my_nid
If both snid and ssnid are in neighbor_list
Triangle_list
If the bit of IS_NEIGH is ONE and snid is not in neighbor_list
If snid is not in neighbor_list, will_prune_list, or blacklist
If ssnid is not in neighbor_list, will_prune_list, or blacklist
Will_probe_list
If the server is close enough compared with given criteria
If the server is not close enough compared with given criteria
Servers in the initial_server_ list
Bootstrapping process
Neighbor_list
Servers satisfy the pruning condition
Will_prune_list
If pruning operation succeeded
Blacklist
Removed after a period of time
……
If pruning operation did not succeed

Fig. 2. Diagram of state transitions for servers among the types of server lists used in LWDLS.

implementation, the server IDs are convertible to the IP addresses and port numbers of the servers. This simple implementation allows us to differentiate servers uniquely in our experiments, and enables the receiver of a lookup message to communicate with the servers whose IDs are attached on the message. Other methods that can provide the same functionality and can provide improved adaptability to different network environments, such as in NAT, are also applicable to LWDLS.

The field of a lookup request of a lookup message shown in Figure 1 includes a unique lookup ID (*lk_id*) and a query condition. The receiver of a lookup message searches its local storage and responds accordingly.

A set of bit flags is defined to express operations of protocols of LWDLS. The *IS_NEIGH* bit indicates if the receiver is one of the neighbors of the sender. The *ACK_FLAG* bit indicates if the receiver needs to send an ACK message to the sender. The *KEEP_FWD* bit indicates if the receiver needs to keep forwarding the lookup request even a local search hit is found.

In addition to searching, lookup messages are also used to convey operation requests of protocols. For instance, a pruning operation request, Prune[*rnid*, *pnid*]⟩, where *rnid* is the ID of the server requesting this pruning operation and *pnid* is the ID of the server under pruning, is attached on a message in the form of $\langle type = PRUNE, rnid, pnid\rangle$, as shown in Figure 1.

4.3. Types of Server Lists

In LWDLS, different types of server lists are used to construct the overlay network and to support the operations of reducing topology mismatch, which include *initial_server_list*, *neighbor_list*, *will_prune_list*, *will_probe_list*, *blacklist*, and *triangle_list*. The state transitions for servers among these lists are shown in Figure 2.

The *initial_server_list* is given to each server when the server is added into the system. It contains a list of servers, all or some of which should exist. The use of the *initial_server_list* helps ensure the connectivity of the overlay network formed by servers in the system. Other methods for adding new servers that ensure the connectivity of overlay network are also applicable to LWDLS.

The *neighbor_list* is used to record the servers that are physically close to this particular server. Furthermore, this list allows the servers that send messages with the bit of IS_NEIGH=1 to be added into the list directly without measuring their distances. In LWDLS, when a server sends a message to servers in its `neighbor_list`, it is required to

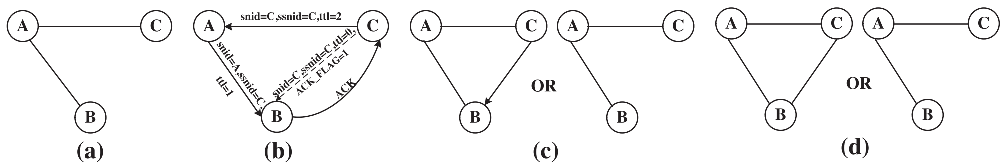


Fig. 3. Example of probing operation. (a) `B` and `C` are not direct neighbors. (b) `C` starts to probing `B`, and `B` sends `ACK` message back to `C`. (c) After probing, `C` may or may not add `B` into its `neighbor_list` according to their network latency. (d) If `C` added `B` into its `neighbor_list`, eventually `B` will also add `C` into its `neighbor_list`, because `C` will send to `B` messages with the bit of `IS_NEIGH=1`.

set the IS_NEIGH bit to one. By doing so, the use of the IS_NEIGH bit not only facilitates adding new servers, but also makes the network graph undirected for simplicity. When a server is joining the system, servers in the `initial_server_list` are directly added into the `neighbor_list`.

The *will_probe_list* lists the servers whose distances are unknown and need to be measured. For every lookup message, the receiver checks whether it needs to add `snid` and `ssnid` attached on the message into its `will_probe_list` based on if their distances are known.

The *will_prune_list* records the servers that will be removed from the `neighbor_list`, because of their longer distances than others and the existence of at least one alternative path to them.

The *blacklist* is used to exclude the servers in it from measuring their distances for a given period of time, because their distances are not close enough according to the previous results.

The *triangle_list* records the triangles formed by three servers on an overlay network. When receiving a message, the receiver (with server ID *my_nid*) detects if itself has formed a triangle with server `snid` and server `ssnid` by checking if both `snid` and `ssnid` are in its `neighbor_list`. If so, a triangle defined as $\langle$my_nid, snid, ssnid$\rangle$ is added into the `triangle_list`. If a server is removed from the `neighbor_list`, all related triangles should also be removed from the `triangle_list` accordingly.

4.4. Probe and Prune Protocols

In order to reduce topology mismatch, we designed *probe* and *prune* protocols that enable a server to explore other physically close servers and to reduce redundant long-distance communications. Requests of probing and pruning operations are only carried by lookup messages with ACK_FLAG=1, as shown in Figure 1.

With the reasonable (weak) assumption that the clocks on the servers in the system will not be accurately synchronized, in probe and prune protocols the network distance between two servers is approximated using the half of Round Trip Time (*RTT*), although this is not completely accurate in real systems. Other more accurate methods are also compatible with LWDLS, but are beyond the scope of this article.

4.4.1. Probing Operation. By encoding `snid` and `ssnid` on lookup messages, the receiver of a lookup message is able to probe its 2-hop neighbors on a dynamic overlay network. As introduced in Section 4.3, `snid` and/or `ssnid` are added into the `will_probe_list`, if their distances to the receiver are unknown. The servers in the `will_probe_list` are probed by sending lookup messages with information of $\langle$`TTL=0,ACK_FLAG=1,IS_NEIGH=0`$\rangle$, as shown in Figure 3(b), so that the servers being probed will send ACK messages back to the server that initiates the probing operation. After receiving an ACK message, a server is able to compute the distance to the probed server. If their distance is smaller than a threshold (the average neighbor distance is

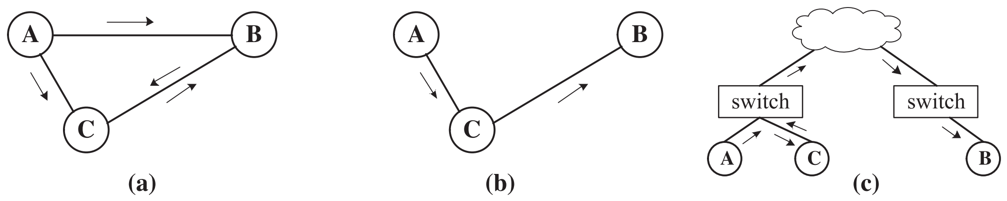


Fig. 4. Pruning operation based on the ratio of the lengths of two adjacent edges of a triangle. (a) When A initiates the flooding search, two redundant messages are generated on the edge BC. (b) After pruning the edge AB, two redundant messages are eliminated. (c) One example of an ideal case where A and C are physically closer than with B, and topology mismatch is reduced by pruning operations of LWDLS.

used), the probed server is moved from the `will_probe_list` into the `neighbor_list`. Otherwise, the probed server is moved into the `blacklist`. Figure 3 shows an example of the probing operation. Initially, server C and B are not direct neighbors. Over communications, B and C learn each other from messages among them, since they are within 2-hop neighborhood. Let C start probing B first. C sends message to B and measures their distance from RTT. If the distance |BC| is smaller than the average neighbor distance of C, then B is moved from C's `will_probe_list` into C's `neighbor_list`. Otherwise, B is moved into the `blacklist` of C.

4.4.2. Pruning Operation. Pruning operations remove redundant long-distance overlay links among servers, which are detected from triangles in the `triangle_list`. Figure 4 shows an example of pruning operation. In Figure 4(a), A, B, and C form a triangle on the overlay network, and there are two paths to reach B from A ($A \rightarrow B$ and $A \rightarrow C \rightarrow B$). In Figure 4(a), if A initiates a flooding search, two of the four messages that are generated in the search process are redundant (on the edge BC). Furthermore, if the distance |AC| is much shorter than |AB|, after removing the edge AB, only two messages are needed to achieve the same search coverage without redundant messages, but with certain acceptable delay introduced between A and B, as shown in Figure 4(b).

A ratio of the lengths of two adjacent edges of a triangle, such as the ratio of |AC| over |AB| in Figure 4(a), is used to determine if one of the two edges should be removed, although some delay is possibly introduced. This is a trade-off between response time and search efficiency. Figure 4(c) shows an ideal case where both redundant messages and topology mismatch are reduced by pruning operations. The advantages of pruning based on the ratio of the lengths of two adjacent edges of a triangle are as follows: first, it is simple and lightweight, particularly in distributed environments, this is because each server in the system is able to initiate a pruning operation only based on its local information, and thereby avoids the distributed consistency problem; second, this mechanism enables a server to differentiate the servers that are physically closer than others to reduce topology mismatch. As a result, the triangles formed by the physically close servers are reserved.

Without the global knowledge of a network graph, removing edges of a network graph in distributed environments has the risk of partitioning the network. To overcome this problem, a pruning operation requires the agreement from all neighbors of a server. Furthermore, to reduce the cost of sending messages to ask for the agreement explicitly, a pruning operation is performed only when a server has an opportunity to broadcast lookup messages with information of ⟨ACK_FLAG=1 and Prune[rnid, pnid]⟩ to its neighbors, as introduced in Section 4.2. Figure 5 shows a diagram of broadcasting lookup messages with pruning requests and asking for ACK messages on which the decisions from the neighbors are attached.

In order to ensure the connectivity of the network graph, after receiving a pruning request, generally, the receiver checks if there is at least one alternative path to

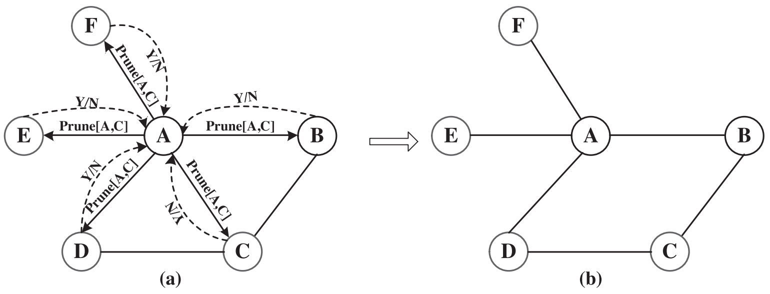


Fig. 5. Diagram of a pruning operation. (a) Server `A` broadcasts lookup messages with pruning requests. (b) After pruning, `C` is removed from `A`'s `neighbor_list`.

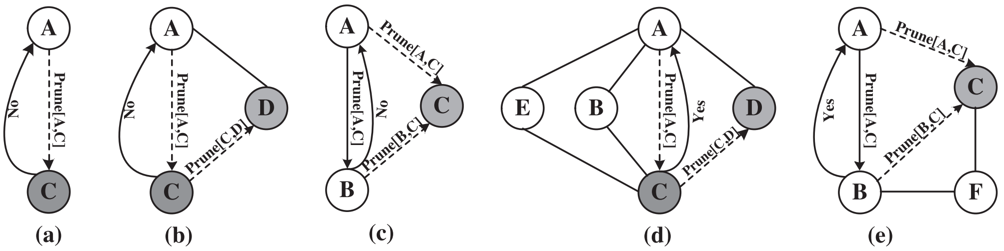


Fig. 6. Examples of pruning operations, where server `A` tries to prune server `C` from its `neighbor_list` in distributed environments. In cases (a) and (b), server `C` will disagree with `A`'s pruning request, since there is no alternative path to `A`; in case (d), server `C` will agree on server `A`'s pruning request. In case (c), one of server `A`'s neighbors, say server `B`, will disagree on pruning `C` by `A`. And in case (e), `B` will agree on `A`'s pruning operation.

server `rnid` and `pnid` separately after the pruning, based on its `neighbor_list` and `triangle_list` without conflict with its undergoing pruning operations. There are two cases to consider. First, if the receiver is the one under pruning, it checks if there is alternative path (2-hop indirect paths) found from its `triangle_list` to server `rnid` without conflict with its undergoing pruning operations. Its decision is sent through the `ACK` message. In Figures 6(a) and 6(b), server `C` is the one under pruning by server`A`, and `C` disagrees with `A`'s pruning request because of the lack of other path to `A` after pruning. In Figure 6(d), server `C` agrees with `A`'s pruning request, since there are other paths (`C` → `B` → `A` and `C` → `E` → `A`) to `A`.

In the second case, the receiver is not the one under pruning. There are two further cases to consider. First, if the server `pnid` is not in the `neighbor_list` of the receiver, the receiver will agrees with the pruning request directly. This is because the receiver does not need to have a path to server `pnid` in order to maintain the connectivity of the overlay network. Second, if server `pnid` is in the `neighbor_list` of the receiver, the receiver needs to check if there are alternative paths (both direct and 2-hop indirect) to server `rnid` and to `pnid` respectively without conflict with its undergoing pruning operation(s). The decision is sent with the `ACK` message.

In Figure 6(c), server `B` disagrees with `A`'s pruning request because of the conflict with `B`'s pruning operation on `C`. Figure 6(d) shows a case where `B` agrees with `A`'s pruning request, since `B` is able to reach `A` (through `B` → `A`) and reach `C` (through `B` → `F` → `C`) after the pruning.

In summary, a pruning request succeeds only when all receivers agreed on it. Furthermore, once a server agrees with a pruning request, all the related triangles are removed from its `triangle_list`.

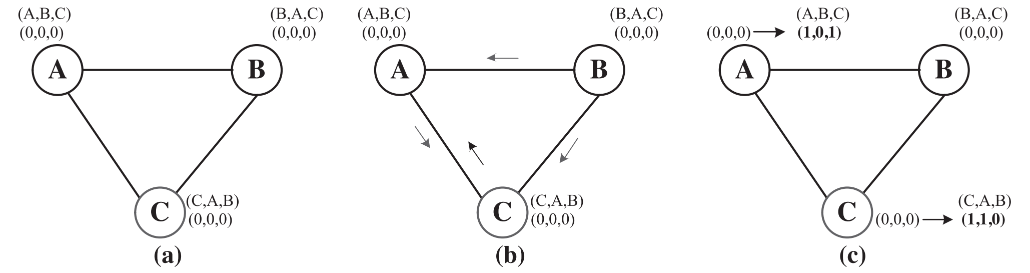


Fig. 7. Example of triangles with knowledge bits updated over communications. (a) Initially, A, B, and C do not know the triangle that they have formed. (b) B broadcast lookup messages to its neighbors, and A and C will forward messages. (c) A and C received redundant messages, and updated their knowledge bits.

4.5. Heuristic Flooding Search

In this section, we introduce a new heuristic flooding search (HFS), which can be ran independently without the probe or prune protocols, or be combined with the probe and prune protocols to optimize the overlay network during the search process. HFS is a type of pure flooding search, but HFS is able to take advantage of a cache of the most recent lookup histories and the `triangle_list` that each server possesses to broadcast lookup messages while reducing redundant messages. Generally, on each hop along the search path, each server will broadcast lookup messages to all its neighbors unless it knows a neighbor should have seen this lookup request before according to the server's cache of lookup history and the `triangle_list`. HFS is heuristic in the sense that each server is able to make local decisions on forwarding lookup messages independently, by observing the events of sending and receiving messages and using its lookup history cached.

With probing and pruning operations, the servers that are physically close are more likely to be neighbors of each other, and the triangles formed by them cannot be broken by pruning operations because of the criteria defined and discussed above. For each triangle detected during the search process, three bits, denoted here as "knowledge bits," are used to represent which of the three servers on the triangle of the overlay network graph have known the existence of this triangle, as shown in Figure 7(a). For example, in Figure 7(c), after server A received the same lookup messages from B and C respectively, A detected the triangle (A,B,C) and updated its knowledge bits to (1,0,0). Furthermore, since server A has forwarded the same lookup message to server C, C should have also detected this triangle. Therefore, server A updated the knowledge bits to (1,0,1).

The `triangle_list` with knowledge bits are used by HFS to reduce redundant messages. HFS is a type of flooding search algorithm. At the beginning, when there are few triangles that have been detected, it works like the pure flooding search [Jiang et al. 2008]. However, as more triangles with knowledge bits are formed, HFS is able to start saving cost by avoiding redundant messages. This is based on the fact that when a server on a triangle of the overlay network, say server A, broadcasts messages to the other servers on the triangle, say server B and C; if B or C knows they have formed a triangle with server A, then B or C does not need to send the received message to others on this triangle, because they should have received the same message from server A.

Figure 8 shows the mechanism for accumulating states, and then use them to reduce redundant messages. In Phase-I, redundant messages are generated on the edges of triangles. However, over a series of communications, more triangles are detected, and servers keep updating the knowledge bits. Some redundant messages are sent purposely by a server to help its neighbors with detecting triangles and to update their

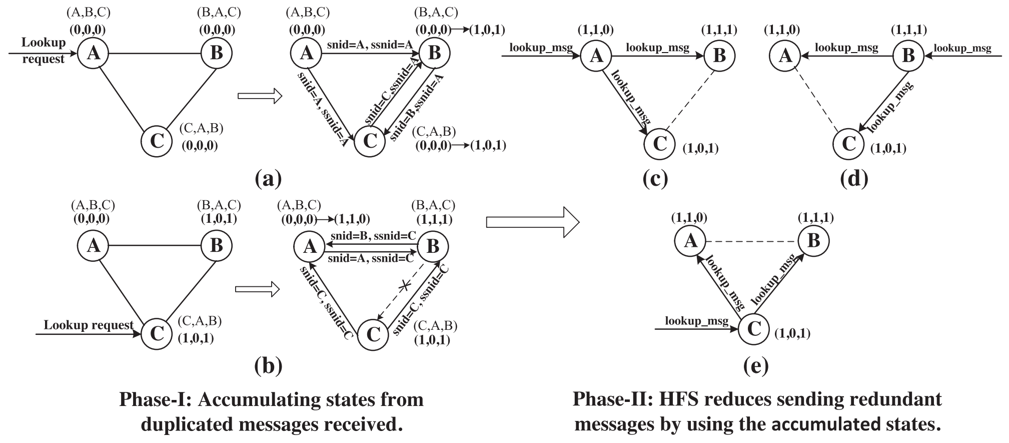


Fig. 8. Accumulating states for reducing redundant messages with the support of triangles detected and knowledge bits encoded. (a) After B and C received redundant messages from each other, both of them detected the triangle ABC, and updated the knowledge bits. (b) When B received a redundant message from A, B sent a duplicated message with to A on purpose, so that A detected this triangle and updated the knowledge bits. In (c), (d), and (e), only two messages are needed between the three servers without sending redundant message by HFS.

states. For instance, on receiving a redundant message, a server checks if the sender is on a triangle on which the corresponding knowledge bit for the sender is zero. If so, this implies that the sender might have not detected this triangle. In this case, a redundant message is sent to the sender with appropriate snid and ssnid information. The cost of sending redundant messages to help update states of neighbors can be amortized by future communications, as long as the triangle exists. Figure 8(b) shows, after serverB received redundant message from A, B sent an extra message to A with snid=B and ssnid=C, so that server A detected the triangle and updated its knowledge bits.

In Phase-II, the accumulated states are used by HFS to save on the cost of redundant messages, as shown in Figures 8(c), 8(d), and 8(e), two redundant messages can be saved with the help of triangles and knowledge bits on a triangle.

In HFS, when a server receives a lookup request it first updates its cache of lookup history by recording the lookup message ID (lk_id) and sending and receiving information for this lookup request. Second, it searches the cached lookup history, adds the servers from which this lookup request was received into a set called In_{neigh}, and add the servers that were forwarded this lookup request into a set called Out_{neigh}. Third, it searches its triangle_list, and adds its neighbors that form triangles with the servers in In_{neigh} into a set called $Skip_{trg\_neigh}$. The servers in $Skip_{trg\_neigh}$ should have received this lookup request from the servers in In_{neigh}, since they form triangles. Furthermore, it searches the triangle_list to find the neighbors that form triangles with the servers in $Skip_{trg\_neigh}$ and have larger server ID than this server, and add them into $Skip_{trg\_neigh}$, until there are no more servers that can be found and added into $Skip_{trg\_neigh}$. One rule in use is that when two servers of a triangle have received the same lookup request, only the one with larger ID should send this message to the third server on the triangle. After finishing the calculation as described above, a server forwards lookup messages to its neighbors in neighbor_list except ones in the sets of In_{neigh}, Out_{neigh}, or $Skip_{trg\_neigh}$.

Figure 9 shows examples of forwarding lookup messages based on the triangles and accumulated information. Figure 9(a) shows the simplest case in which only one triangle is involved. Figure 9(b) shows the case of using two triangles and forwarding

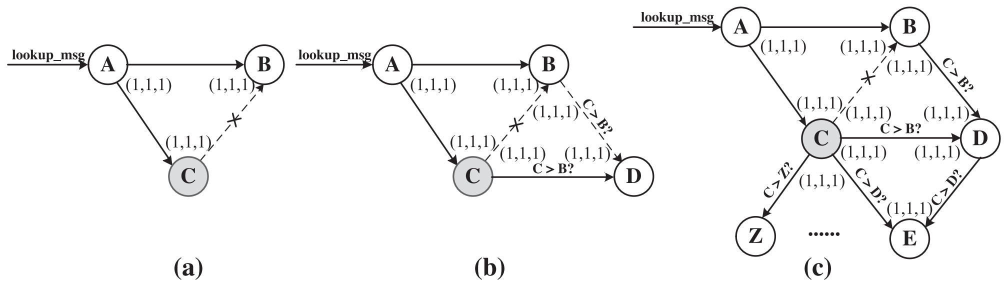


Fig. 9. Examples of selectively forwarding messages by HFS. (a) Only one triangle is involved. C does not need to send message to B, since A has sent. (b) Two triangles are involved. For D, the one having larger server ID among B and C should send to D. (c) A number of triangles are involved.

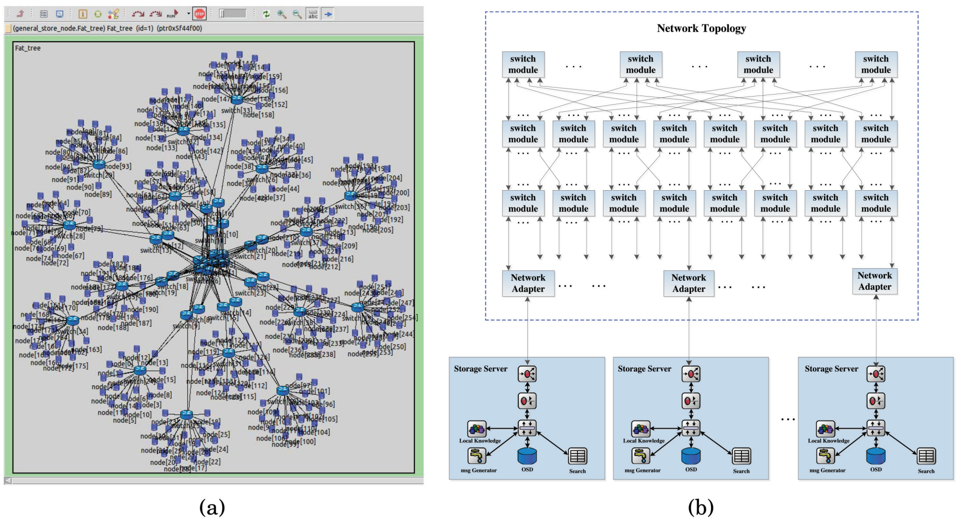


Fig. 10. Simulator. (a) Simulated HPC storage system with three-level fat-tree network topology. (b) Architecture of the simulator.

messages based on the result of the comparison of server IDs. In Figure 9(c), more triangles are involved, and server C makes forwarding decisions based on the triangles it has and the observed events of sending and receiving messages among its neighbors.

5. EXPERIMENTS AND ANALYSIS

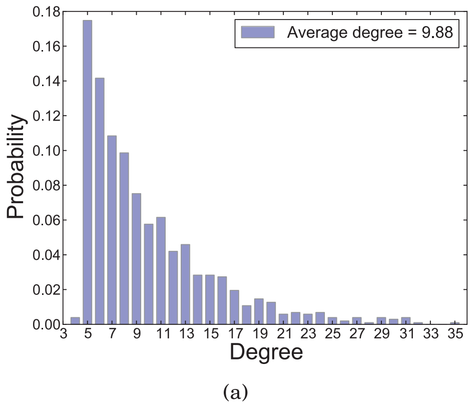
In order to test LWDLS in HPC environment with a large number of storage servers, we developed a simulator using an open-source network simulation framework [Varga and Hornig 2008; Denzel et al. 2008]. We present our simulation results in this section.

5.1. Simulator and Configurations

The simulated HPC storage system, as shown in Figure 10(a), is organized with a three-level fat-tree network topology [Denzel et al. 2008], which connects with all of storage servers in the system. The latency and bandwidth of each physical link used are 5μs and 10Gbps respectively, in order to emulate a common 10 gigabit Ethernet network [Tolley 2011]. In the simulator, a storage server is represented by a compound module containing a set of discrete modules. Figure 10(b) shows the detailed architecture of our simulator. The configuration files are shown in Table I.

Table I. Configuration Files

File Name	Module	Function
Routing tables	Switch	Routing messages on the network
Initial server list	Search	Forming initial overlay network
List of scheduled lookup requests	Search	Test search performance
Indices of data objects	Local object storage	Record data objects stored in each server



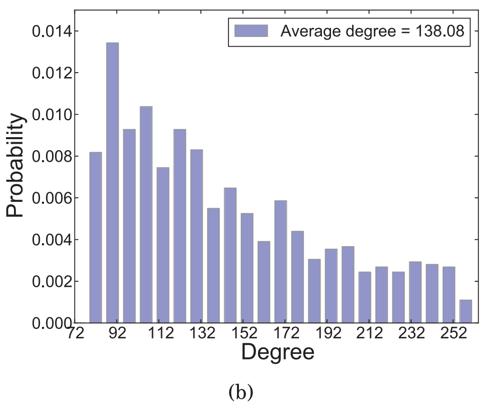


Fig. 11. Two overlay network graphs used for testing LWDLS. (a) Initial degree distribution of network graph with $\mathtt{M} = 5$. (b) Initial degree distribution of overlay network graph with $\mathtt{M} = 80$.

5.1.1. Generation of Overlay Network Graphs. We evaluated the search performance of the HFS algorithm and effectiveness of probing and pruning operations on two overlay network graphs, each incorporating different degree distributions of servers and levels of topology mismatch.

These overlay network graphs were generated at random using the following mechanism. At the beginning, there is only one server in the system. When a new server is added, it randomly selects M existing servers in the system as its neighbors. Furthermore, if a server is selected by other servers that are added later, it also adds them subsequently as its neighbors.

Based on this mechanism, two overlay network graphs both with 1,024 servers were generated. The graphs have $\mathtt{M} = 5$ and $\mathtt{M} = 80$, respectively. Their initial degree distributions are shown in Figure 11. These two overlay network graphs enable us to test LWDLS on both low-degree (the graph with $\mathtt{M} = 5$, average degree $= 9.88$) and high-degree (the graph with $\mathtt{M} = 80$, average degree $= 138.08$) network graphs, and to complete a number of simulations within a manageable amount of time.

Before running a simulation, each server has an `initial_server_list`, as introduced in section 4.3, which contains its neighbors on the overlay network graph to be used during the simulation.

5.1.2. Generation of Lookup Requests. Given a lookup request, search performance was evaluated by letting every server in the system process this request separately, with `TTL` values ranging from 1 to 7. This range of `TTL` values is large enough to achieve search coverage close to 100% [Ripeanu et al. 2002], and this coverage is confirmed in our simulations. Search performance is computed from the average performance of all servers using the same `TTL` value.

Each server has a list of lookup requests that are processed at a given time and are randomly selected. The start time of processing a lookup request has no correlation with the search performance of LWDLS, if probing and pruning operations are disabled.

After a simulation, all of the histories of sending and receiving messages are recorded for performance analysis. We analyze our results as a function of TTL value.

5.1.3. Data Indices in the Local Object-based Storage Device. To demonstrate the applicability and to test the search performance of LWDLS, we made the following restriction in our experiments: for each lookup request there is only one copy of the target data object existing and it is randomly distributed in the system. Note that the choice of using different replication mechanisms is an orthogonal concern to using LWDLS and is not specifically addressed further in this article.

Object-based storage devices (OSDs) have recently been used in parallel file systems [Schwan 2003; Carns et al. 2000; Weil et al. 2006b]. In our simulations, local OSDs are only used to store data indices, rather than simulating the input-output (I/O) behavior of storage devices. On receiving a lookup request, a server searches its local OSD; a lookup hit message is sent if there is a local match.

5.2. Performance Metrics

Search scope is defined as the ratio of the number of servers that queries have reached in a given search process divided by the total number of servers in the system. In (pure) flooding searches, search scope is only determined by the TTL values chosen, and thereby achieving 100% search scope is guaranteed by choosing a sufficiently large TTL value. However, TTL values have to be tuned for the size of the network being searched to reduce the cost of sending redundant messages. In probabilistic flooding searches, using large TTL values still cannot guarantee achieving 100% search scope. This is an important difference between (pure) flooding searches and probabilistic flooding searches. In HFS of LWDLS, achieving 100% search scope is also guaranteed if the TTL values chosen are large enough. By caching the lookup histories on every server in the system, one can choose large TTL values safely with reduced network traffic. In this study, with fixed traffic cost, we aim to maximize search scope, while, with fixed search scope, we aim to minimize traffic cost.

Search efficiency is defined as the ratio of the number of servers searched over the total number of messages sent during a search process. This metric is used to evaluate the cost of processing a lookup request on average. It is combined with search scope in order to evaluate the scalability of a search algorithm.

Average neighbor distance is used to evaluate the effectiveness of reducing topology mismatch. Minimizing average neighbor distance implies a better match with the underlying physical network. This quantity is measured both before and after calling probing and pruning operations.

5.3. Effectiveness of the HFS Search Algorithm

First, we compare the performance of probabilistic flooding search algorithms with the pure flooding search algorithm [Jiang et al. 2008]. In the pure flooding search algorithm, upon receiving a lookup request, the receiver forwards lookup messages to all its neighbors, except for the sender of this request. However, in probabilistic flooding search algorithms, a server only forwards lookup messages to a subset of its neighbors, which are selected based on a probabilistic value (called *P*) on each hop along the search path. We used $P = 0.1$, 0.5, and 0.9 values respectively for choosing neighbors to demonstrate the basic mechanism of probabilistic searches.

In Figures 12 and 13, we showed the results of running probabilistic flooding search algorithms on the two example overlay networks. This was shown although probabilistic flooding algorithms are less interesting to us for reasons including the lack of 100% search coverage even assuming large TTL values, and slower search speed than the pure flooding search algorithm [Jiang et al. 2008]. The results show that with the

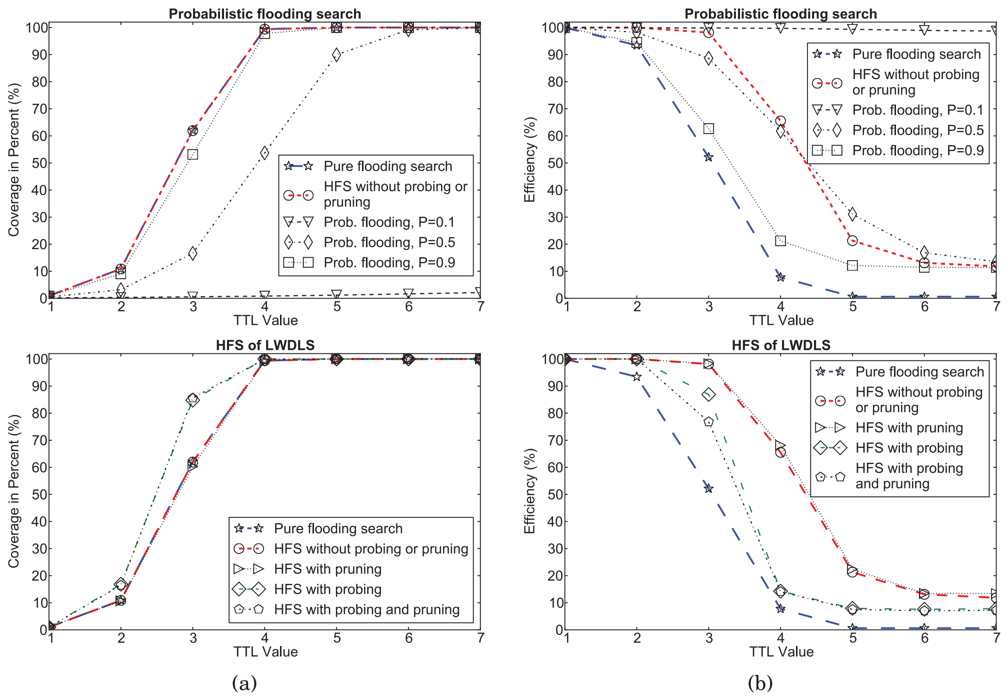


Fig. 12. Comparison of search performance on the overlay network with $\mathtt{M} = 5$. (a) Comparison of search scope. (b) Comparison of search efficiency.

same TTL value, the probabilistic flooding search algorithms have smaller search scope than pure flooding search, but they achieved higher search efficiency. Smaller search scope implies slower search speed, and thus these algorithms are less suitable for HPC systems.

Second, we compare HFS of LWDLS with pure flooding search [Jiang et al. 2008]. The HFS algorithm was tested on the two overlay network graphs, and is primarily compared with the pure flooding search algorithm [Jiang et al. 2008], because they are more comparable than other methods (such as the probabilistic flooding search) in terms of having similar search speed, search scope, and few/minimal requirements for state maintenance.

We ran HFS under four different modes including HFS with pruning, with probing, without probing or pruning, and with both probing and pruning. The results from simulations using the network graph with $\mathtt{M} = 5$ are shown in Figure 12. In this section, we analyzed the performance of HFS without probing or pruning. The analysis of running HFS with probing and/or pruning is described later in Section 5.4.

Figure 12(a) shows that, given a lookup request with certain TTL value, the HFS algorithm without probing or pruning had similar search scope on average to pure flooding search [Jiang et al. 2008], but the HFS algorithm achieved higher search efficiency than pure flooding search on every TTL value used, as shown in Figure 12(b). The savings are more distinct compared to pure flooding search when using TTL values ranging from 2 to 4. As the search coverage nears 100%, the search efficiency of the HFS algorithm drops quickly, especially for TTL values of 5, 6, and 7. This is because on the low-degree network graph HFS has fewer opportunities for taking advantage of states accumulated on the triangles formed by servers in the local neighborhood. In

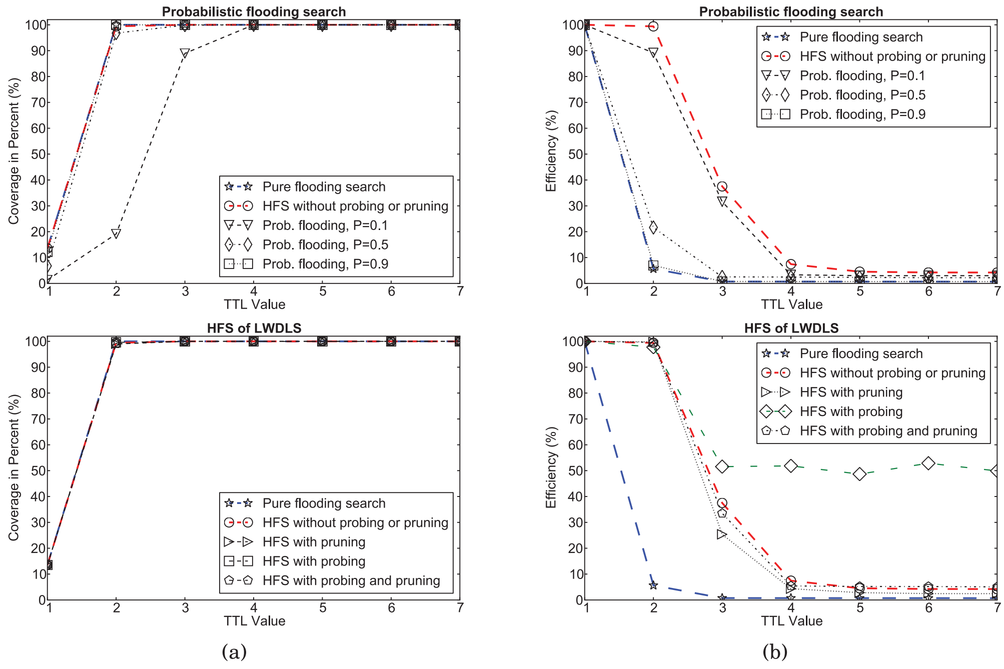


Fig. 13. Comparison of search performance on the network graph with M = 80. (a) Comparison of search scope. (b) Comparison of search efficiency.

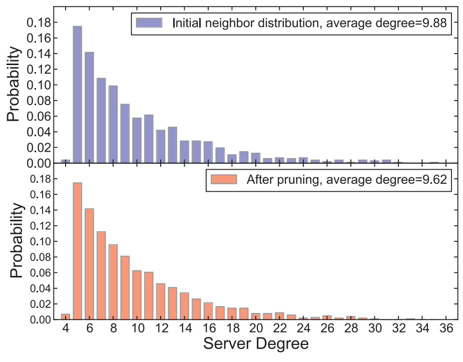
addition, some research [Ripeanu et al. 2002] has shown that 95% of any two servers are less than 7 hops away; in such situations, messages with TTL = 7 are rarely used.

Figure 13 shows the results of running the HFS algorithm and pure flooding search algorithm on the network graph with M = 80. Running the HFS algorithm without probing or pruning had similar search scope to pure flooding search algorithm, but it had much higher search efficiency. In Figure 13(b), the efficiency of pure flooding search dropped quickly from TTL = 2. This shows the high cost of running pure flooding search on high-degree network graphs, and indicates the saving of search cost by the HFS algorithm is significant by comparison.

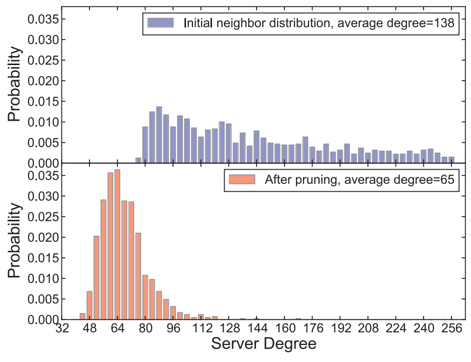
In summary, from the results of using the low-degree and high-degree network graphs, we find that the search coverage that can be achieved depends on the system scale, the degree of the connectivity of overlay networks, and TTL values chosen. In HFS, the only parameter needed is the TTL value, which is simpler to characterize (tune) than the parameters required for probabilistic search methods. As HFS is a (pure) flooding search algorithm, for sufficiently large TTLs, HFS always finds the data in a network. However, practically smaller TTLs are suitable for typical networks. In particular, limiting TTLs is necessary to control network overhead because the TTL value controls the number of messages generated. If a piece of data should not be located with the given TTL, policies for trying larger TTLs on retry are possible, although beyond the scope of the current study. Such adaptive TTL policies on retry could also provide further evidence for ways to adapt the overlay network to manage connectivity.

5.4. Effectiveness of Probe and Prune Protocols

We studied the effectiveness of probe and prune protocols on the two example overlay network graphs. We ran simulations where every server was scheduled with a set of TTL = 2 lookup requests, and then we evaluated the changes of overlay network graphs



(a)



(b)

Fig. 14. Effects of probing and pruning operations on the node degree distribution, tested on the two overlay network topologies. (a) Change of node degree distribution on the overlay network with $\mathtt{M} = 5$. (b) Change of node degree distribution on the overlay network graph with $\mathtt{M} = 80$.

after the simulations. The ratio for pruning used was 0.5. This ratio was chosen because we tried to make the servers connected with the same switch to become neighbors, and once a communication ran across more than one switch, the latency is modeled as doubled, which is reasonable for certain networks. In general, the ratio and the shortest neighbor distance together determine the size of the region from which a server adds other servers into its `neighbor_list`.

Figure 14 shows the changes of server degree distributions on the two example network graphs by pruning operations of LWDLS. Results showed that the degrees of servers that had high-degree were reduced by pruning operations. This effect is more distinct on the network graph with $\mathtt{M} = 80$ as shown in Figure 14(b). This is because high-degree servers have more redundant long-distance overlay links because of the mechanism of randomly selecting neighbors when the network was generated (which introduced topology mismatch). Furthermore, high-degree servers had more opportunities to handle pruning operations than others (whether initiated by themselves or by their neighbors). The effect of reducing server degrees indicates the effectiveness of pruning operations, and implies the concomitant reduction of topology mismatch.

Figure 15 shows the changes of distribution of neighbor distances after pruning operations. These results are from the same simulations shown above. The average neighbor distance was reduced on the two example network graphs by pruning operations during search. In Figure 15(a), the effect of reducing the average neighbor distance is not distinct. This is because the initial overlay network graph is relatively sparse, and there are a limited number of triangles that can be used by pruning. This implies more time and lookup requests are needed for exploring the physically close servers. However, on the high-degree network graph with $\mathtt{M} = 80$, the reduction of average neighbor distance is more pronounced than on the low-degree network. In addition to the effect of reducing server degree as shown in Figure 14(b), these results indicate that reduction of topology mismatch, and thus indicate the effectiveness of the pruning operations.

In Figures 12 and 13, we have also shown the search performance of HFS with probing and/or pruning operations. In Figure 12(a), HFS with probing achieved larger search scope than pure flooding search using TTL values of 2 and 3 on the low-degree network graph, while having higher search efficiency than pure flooding search algorithm [Jiang et al. 2008]. Although the efficiency is lower than running HFS without either probing or pruning operations, the extra cost incurred by probing and pruning was compensated

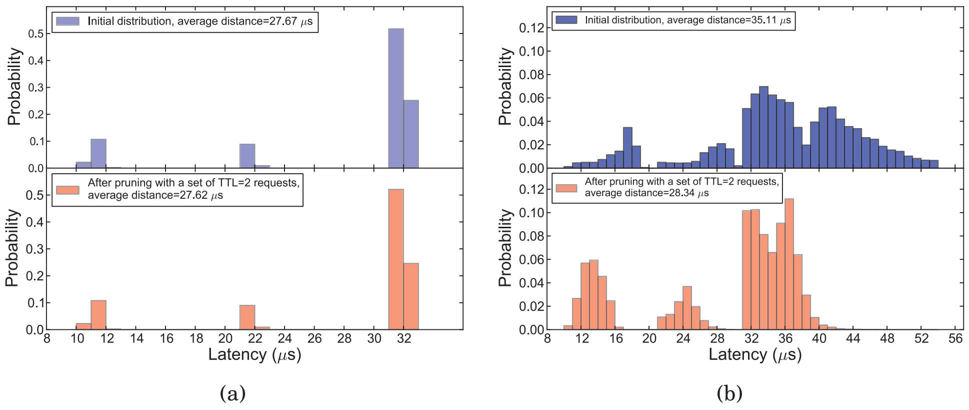


Fig. 15. Effects of pruning operations on the neighbor distance distribution, tested on the two overlay network topologies. (a) Distribution of neighbor distance after pruning with the initial overlay network graph with `M` $= 5$. (b) Distribution of neighbor distances after pruning with the initial overlay network graph with `M` $= 80$.

by the reduction of topology mismatch. In Figure 13, on the high-degree network graph with `M` $= 80$, HFS running under different modes achieved higher search efficiency than pure flooding search. However, in Figure 13(b), running HFS with probing using `TTL` values ranging from 3 to 7 had much higher search efficiency than running HFS under other modes. This is so for two reasons. First, on the high-degree network graph, finding physically close servers is easier than on the low-degree network. Second, as more servers that are physically close are grouped as neighbors by probing operations, more triangles are formed in the local neighborhood. Therefore, HFS is able to use the states accumulated from larger number of triangles with updated knowledge bits to reduce redundant messages.

In summary, HFS has comparable search scope to pure flooding search algorithm [Jiang et al. 2008], but HFS has higher search efficiency. Furthermore, HFS can be used independently without either probing or pruning for cases where reducing topology mismatch is not required. As shown in Figure 13(b), on the high-degree network graph, HFS with probing achieved better search performance than pure flooding search or HFS running under other modes. The results clearly show that probing and pruning operations are able to reduce topology mismatch.

6. CONCLUSIONS AND FUTURE WORK

In this article, we presented LWDLS, a lightweight data location service for Exascale storage systems and geo-distributed storage systems. LWDLS provides a search-based solution for locating data; it enables free data placement, movement, and replication. With the probe and prune protocols and the HFS algorithm, LWDLS is able to address the problems of topology mismatch and inefficient search performance of the pure flooding search algorithm [Jiang et al. 2008].

Compared with existing search methods, LWDLS is comparatively more lightweight and scalable in terms of low overhead, high search efficiency, and for its ability to avoid global state as well as periodic messages. For example, LWDLS can be used as an independent data location method in nondeterministic storage systems [Nowoczynski et al. 2008; Curry et al. 2012] and in deterministic storage systems [Weil et al. 2006b; Yang et al. 2004] to deal with cases where search is needed.

The effectiveness of LWDLS has been tested through extensive simulations modeling large-scale HPC storage environments. The results show that LWDLS is able to locate data quickly and efficiently with low cost of state maintenance.

In this article, we focused mainly on the search algorithm, protocols, and performance, without covering operational aspects such as recovery from hardware or network faults. In the future, we will extend LWDLS with features such as improved fault-tolerance capacity. Furthermore, in this study we focused mainly on the search problem in nondeterministic storage systems. Finding data is a prerequisite to dealing with other related problems of building nondeterministic Exascale storage systems, such as read and write policies, replication mechanisms, and versioning. In the near future, we will test LWDLS at larger scale with different replication strategies, and test it under other physical network topologies and various network environments.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

ACKNOWLEDGMENTS

We wish to thank staff in the High Performance Computing Laboratory (HPCL) of Computer and Information Sciences (CIS) at the University of Alabama at Birmingham, including Amin Hassani, Seok Bae Yun, Shane Farmer, Alex Filby, and Zekai Demirezen, for helping review the draft of this article and providing their valuable feedback and comments. We also wish to thank the UAB CIS IT staff for maintaining the Linux clusters used and for providing timely support. We would like to thank the reviewers for their suggestions and comments that helped improve the quality of this article.

REFERENCES

John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. 2009. PLFS: A checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. 1–12. DOI: http://dx.doi.org/10.1145/1654059.1654081

Kevin Brandstatter, Dongfang Zhao, Ke Wang, Anupam Rajendran, Zhao Zhang, Ioan Raicu, Tonglin Li, and Xiaobing Zhou. 2013. ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table. In *Proceedings of the IEEE International Parallel & Distributed Processing Symposium (IPDPS'13)*.

John Buford. 2013. Microsoft PowerPoint - JBuford-IETF-P2PSIP-Overlay-Systems-v3.ppt-IETF64_P2PSIP_AdHoc_P2P_Overview_Buford.pdf. (2013). http://www.softarmor.com/sipping/meets/ietf64/slides/IETF64_P2PSIP_AdHoc_P2P_Overview_Buford.pdf.

Philip H. Carns, Walter B. Ligon, III, Robert B. Ross, and Rajeev Thakur. 2000. PVFS: A parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*.

Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. 2003. Making gnutella-like P2P systems scalable. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'03)*. ACM, New York, 407–418. DOI: http://dx.doi.org/10.1145/863955.864000

Sérgio Crisóstomo, Udo Schilcher, Christian Bettstetter, and João Barros. 2012. Probabilistic flooding in stochastic networks: Analysis of global information outreach. *Comput. Netw.* 56, 1, 142–156. DOI: http://dx.doi.org/10.1016/j.comnet.2011.08.014

Matthew L. Curry, Ruth Klundt, and H. Lee Ward. 2012. Using the Sirocco file system for high-bandwidth checkpoints. Sandia National Laboratories, Technical Report SAND2012-1087. http://prod.sandia.gov/techlib/access-control.cgi/2012/121087.pdf.

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.* 41, 205–220. DOI: http://dx.doi.org/10.1145/1323293.1294281

Wolfgang E. Denzel, Jian Li, Peter Walker, and Yuho Jin. 2008. A framework for end-to-end simulation of high-performance computing systems. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops (Simutools'08)*. ICST

(Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), Article 21, http://dl.acm.org/citation.cfm?id=1416222.1416248.

Jack Dongarra. 2010. Impact of architecture and technology for extreme scale on software and algorithm design. In *Proceedings of the Department of Energy Workshop on Cross-Cutting Technologies for Computing at the Exascale*.

Rossano Gaeta and Matteo Sereno. 2011. Generalized probabilistic flooding in unstructured peer-to-peer networks. *IEEE Trans. Parallel Distrib. Syst.* 22, 12, 2055–2062. DOI: http://dx.doi.org/10.1109/TPDS.2011.82

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*. ACM, 96–108. http://www.cs.rochester.edu/sosp2003/papers/p125-ghemawat.pdf.

Christos Gkantsidis, Milena Mihail, and Amin Saberi. 2005. Hybrid search schemes for unstructured peer-to-peer networks. In *Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCom'05)*. Vol. 3, 1526–1537. DOI: http://dx.doi.org/10.1109/INFCOM.2005.1498436

Anjali Gupta, Barbara Liskov, and Rodrigo Rodrigues. 2003. One hop lookups for peer-to-peer overlays. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems (HOTOS'03)*. Vol. 9, USENIX Association, Berkeley, CA, 2–2. http://dl.acm.org/citation.cfm?id=1251054.1251056.

Song Jiang, Lei Guo, Xiaodong Zhang, and Haodong Wang. 2008. LightFlood: Minimizing redundant messages and maximizing scope of peer-to-peer search. *IEEE Trans. Parallel Distrib. Syst.* 19, 5, 601–614. DOI: http://dx.doi.org/10.1109/TPDS.2007.70772

Ketama 2013. Ketama. `http://www.audioscrobbler.net/development/ketama/`.

Avinash Lakshman and Prashant Malik. 2010. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.* 44, 2, 35–40. DOI: http://dx.doi.org/10.1145/1773912.1773922

Tsungnan Lin, Pochiang Lin, Hsinping Wang, and Chiahung Chen. 2009. Dynamic search algorithm in unstructured peer-to-peer networks. *IEEE Trans. Parall. Distrib. Syst.* 20, 5, 654–666. DOI: http://dx.doi.org/10.1109/TPDS.2008.134

Yunhao Liu. 2008. A two-hop solution to solving topology mismatch. *IEEE Trans. Parall. Distrib. Syst.* 19, 11, 1591–1600. DOI: http://dx.doi.org/10.1109/TPDS.2008.24

Yunhao Liu, Li Xiao, Xiaomei Liu, L.M. Ni, and Xiaodong Zhang. 2005. Location awareness in unstructured peer-to-peer systems. *IEEE Trans. Parall. Distrib. Syst.* 16, 2, 163–174. DOI: http://dx.doi.org/10.1109/TPDS.2005.21

Boon Thau Loo, Ryan Huebsch, Ion Stoica, and Joseph M. Hellerstein. 2004. The case for a hybrid p2p search infrastructure. In *Proceedings of the 3rd International Conference on Peer-to-Peer Systems (IPTPS'04)*. Springer-Verlag, Berlin, Heidelberg, 141–150. DOI: http://dx.doi.org/10.1007/978-3-540-30183-7_14

Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. 2002. Search and replication in unstructured peer-to-peer networks. In *Proceedings of the 16th International Conference on Supercomputing (ICS'02)*. ACM, New York, 84–95. DOI: http://dx.doi.org/10.1145/514191.514206

Petar Maymounkov and David Mazières. 2002. Kademlia: A peer-to-peer information system based on the XOR metric. In *Revised Papers from the 1st International Workshop on Peer-to-Peer Systems (IPTPS'01)*. Springer-Verlag, 53–65. http://dl.acm.org/citation.cfm?id=646334.687801.

Memcached 2013. Memcached. `http://www.memcached.org/`.

Mark Newman, Steven Strogatz, and Duncan J. Watts. 2001. Random graphs with arbitrary degree distributions and their applications. *Phys. Rev. E* 64, 2, 026118. DOI: http://dx.doi.org/10.1103/PhysRevE.64.026118

Paul Nowoczynski, Nathan Stone, Jared Yanovich, and Jason Sommerfield. 2008. Zest Checkpoint storage system for large supercomputers. In *Petascale Data Storage Workshop (PDSW'08)*. 1–5. DOI: http://dx.doi.org/10.1109/PDSW.2008.4811883

Konstantinos Oikonomou, Dimitrios Kogias, and Ioannis Stavrakakis. 2010. Probabilistic flooding for efficient information dissemination in random graph topologies. *Comput. Netw.* 54, 10, 1615–1629. DOI: http://dx.doi.org/10.1016/j.comnet.2010.01.007

Karl Pearson. 1905. The problem of the random walk. *Nature* 72, 1865, 294–294. DOI: http://dx.doi.org/10.1038/072294b0

Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. 2001. A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev.* 31, 4, 161–172. DOI: http://dx.doi.org/10.1145/964723.383072

Matei Ripeanu, Adriana Iamnitchi, and Ian Foster. 2002. Mapping the gnutella network. *IEEE Internet Comput.* 6, 1, 50–57. DOI: http://dx.doi.org/10.1109/4236.978369

Ohad Rodeh and Avi Teperman. 2003. zFS - A scalable distributed file system using object disks. In *Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies, 2003 (MSST'03)*. 207–218. DOI: http://dx.doi.org/10.1109/MASS.2003.1194858

Antony Rowstron and Peter Druschel. 2001. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP / ACM International Conference on Distributed Systems Platforms (Middleware)*, Heidelberg, Germany, 329–350.

F. Schmuck and R. Haskin. 2002. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 1st Conference on File and Storage Technologies (FAST'02)*, Monterey, CA.

Philip Schwan. 2003. Lustre: Building a file system for 1,000-node clusters. In *Proceedings of the Linux Symposium*. 9.

Haiying Shen, Cheng-Zhong Xu, and Guihai Chen. 2006. Cycloid: A constant-degree and lookup-efficient P2P overlay network. *Perform. Eval.* 63, 3, 195–216. DOI: http://dx.doi.org/10.1016/j.peva.2005.01.004

Alexandre O. Stauffer and Valmir C. Barbosa. 2004. Probabilistic heuristics for disseminating information in networks. *CoRR* cs.NI/0409001.

Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.* 31, 4, 149–160. DOI: http://dx.doi.org/10.1145/964723.383071

Hong Tang and Tao Yang. 2003. An efficient data location protocol for self-organizing storage clusters. In *Proceedings of the International Conference for High Performance Computing and Communications*.

Bruce Tolley. 2011. Solarflare Fujitsu low latency test report - Solarflare_low-latency_TestReport.pdf. http://www.fujitsu.com/downloads/COMP/ffna/ethernet/Solarflare_Low-Latency_TestReport.pdf.

András Varga and Rudolf Hornig. 2008. An overview of the OMNeT++ simulation environment. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops (Simutools'08)*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). Article 60, http://dl.acm.org/citation.cfm?id=1416222.1416290.

Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. 2006a. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the ACM / IEEE Conference on Supercomputing (SC'06)*. ACM.

Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006b. CEPH: A scalable, high-performance distributed file system. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*. 307–320.

Tao Yang, Hong Tang, Aziz Gulbeden, Jingyu Zhou, and Lingkun Chu. 2004. Sorrento: A self-organizing storage cluster for parallel data-intensive applications. In *Proceedings of the High Performance Computing, Networking and Storage Conference (SC'04)*.

Min Yang and Yuanyuan Yang. 2010. An efficient hybrid peer-to-peer system for distributed data sharing. *IEEE Trans.* 59, 9, 1158–1171. DOI: http://dx.doi.org/10.1109/TC.2009.175

Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. 2001. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Tech. rep. UCB/CSD-01-1141. EECS Department, University of California, Berkeley. http://www.eecs.berkeley.edu/Pubs/TechRpts/2001/5213.html.

Received June 2013; revised September 2013; accepted February 2014