Efficient and Secure Storage Systems Based on Peer-to-Peer Systems

Yongge Wang, Yuliang Zheng, and Beitseng Chu
University of North Carolina at Charlotte
{yonwang, yzheng, billchu}@uncc.edu

Abstract

Two fundamental problems that confront peer-to-peer applications are to efficiently locate the node that stores a data object and to achieve data security (e.g., confidentiality, integrity, authentication, and authorization). This paper presents a model for secure distributed applications based on peer-to-peer systems and presents Chord+, a distributed lookup protocol that address the data object location problem in peer-to-peer systems. The protocol Chord+ is a novel combination of Chord and Tapestry/Plaxton-Rajaraman-Richa with provable better performance. In particular, node insertion in our scheme could be finished in $O(\log_b N)$ steps (all other existing node insertion algorithms for peer-to-peer systems take $O((\log_2 N)^2)$ steps—see the summary in Hildrum et al.).

1 Introduction

The Internet is growing well beyond the intent of its designers. One important future driver for the further growth of Internet is the home media networks [2] and personal digital stores [1]. Another important future driver for further growth of the Internet is the emergence of small, inexpensive and low-powered devices with the capability to support different types of functionalities and to perform regular tasks. These devices can be deployed in large numbers and can be embedded, pervasively and unobtrusively, in the environment. In pursuing transparency, an immediate question is how to provide persistent information to these new applications such as home media networks, personal digital stores, and small embedded devices? As pointed out in [7], persistent information is necessary for transparency since it permits the behavior of devices to be independent of the devices themselves, allowing an embedded component to be rebooted or replaced without losing vital configuration information. Further, the loss or destruction of a device does not lead to lost data.

The contributions of this paper are: a scalable architecture for distributed storage services, efficient security mechanisms for this service, and an efficient and scalable protocol for data object lookup in a dynamic peer-to-peer system.

2 Distributed storage service requirements

Networked storage systems make distributed storage services simple and possible. On the other hand, confidentiality, reliability, availability, and durability of networked storage systems are relatively difficult to implement. Only after these problems are solved for networked storage systems, distributed storage services are possible. For distributed storage services, we have the following distinguishing requirements: (1) The storage systems and customers should be connected to each other through some sorts of media and this kind of connection should be possible for mobility. (2) The information in these storage systems should be extremely durable and survivable, and the customers need absolute trust and confidence in the storage service (the same trust and confidence that customers have in banks). (3) Data availability should always be guaranteed. (4) Customers do not rely on storage service providers (SSP) such as data banks to protect their privacy. (5) When customers store their encrypted data in data bank or SSP sites, the keys should only be held by customers themselves. (6) The systems are survivable against malicious attacks.

In practice, there are several analogies to survivable data storage systems. For example, in the telecommunication industry, there are two kinds of companies. Some companies build the underlying communication infrastructures, and other companies sell communication services such as phone services and data network services. In order to use some telecommunication services, customers could just ask the service company to set up all the services or the customers could first build a local area network (such as LAN) and ask the service company to connect the worldwide network to their local network.

In order to build survivable and distributed storage services, one could certainly incorporate the advantages of the analog models into the storage service model and avoid the disadvantages of them. Indeed, one can build survivable and distributed storage services as follows: There is a data storage infrastructure pool in the world. Each data storage infrastructure builder contributes (sells) their data storage infrastructures to this pool. In addition, there are data storage service companies which sells the storage from

these pool to the customers. There are also separate key banks for data storage services so that customers can store their shares of keys (kinds of key-escrow or secret sharing schemes) in these banks for future key recovery. In the next sections, we will present detailed specifications and mechanisms for each part of the storage services. Distributed applications could be easily built on this kind storage services. For example, this service could provide virtual private storage (VPS) services for enterprises (that is, if an enterprise chooses to outsource their storage service, they will be able to get a virtual private storage system). Distributed web services could be conveniently embedded in this kind of storage infrastructure (data in this kind of service requires write-access control though no read-access control). It is also convenient to provide storage services for small devices from this service. A small device has only a few line of codes (BIOS) stored in its ROM. All other data including operating systems are fetched from the storage infrastructure (this should be globally accessible). Lastly, personal digital store could easily be built on this kind of storage services.

3 Survivable storage service infrastructure and system overview

The underlying infrastructure pool is basically a collection of inter-connected storage device nodes. A storage infrastructure builder could add (sell) new storage devices to this pool and existing storage devices could leave the pool due to crash or close of business. Storage service providers sell the storage capacity to end users and users pay a monthly subscription fee to one specific storage service provider to access certain amount of storage capacity in the global pool. The storage service should be highly available from anywhere in the world. For example, small diskless embedded devices can access the storage pool wirelessly from airports, offices, and cafeteria. Diskless server computers, desktop personal computers, laptops, and digital home media networks can access the storage pool via high-speed networks. When a device (either small device or big ones such as server computers) is broken, the only thing that one needs to do is to replace it with a new device and install appropriate authentication tokes for accessing the storage pool. The infrastructure will have the following distinguished characteristics: durability, untrusted infrastructure, locality, stability, decentralization, and easy to use key recovery banks.

4 Approaches to the infrastructure design

Similar to several other peer-to-peer systems (e.g., Chord [11], Tapestry [7, 5], Plaxton/Rajaraman/Richa [8], Pastry [4], and CAN [9]), consistent hashing mechanism [11, 6] is used in our scheme to keep even distribution of data objects in the storage infrastructure pool. We will assume that m is a fixed integer (e.g., $m = 160$ if SHA-1 is used as the underlying hash function). Specifically, Chord+ provides efficient mapping of data objects to nodes responsible for them. In previous schemes such as Chord, Tapestry, or Pastry, the consistent hashing function is used to balance the load of each node. That is, with high probability, all nodes receive the same number of data objects. This is not an ideal solution for global storage infrastructure since different nodes may have different volume capacity. Chord+ improves the scalability of volume capacity by taking variable identifiers for storage nodes. A Chord+ node needs only to know a small amount of "routing" information about other nodes. In the storage infrastructure, each node maintains information about at most $b * m/\log b$ other nodes where b is any fixed integer (e.g., $b = 16, 128$, etc.) and m is the bit number of identifiers, and a data object lookup requires at most $m/\log b$ queries. When a node with an m_α-bit identifier string joins or leaves the network, Chord+ must update the routing information for each node. A join or leave requires at most $O(k/N + k2^{-m_\alpha})$ message (including data objects) exchanges.

4.1 Node and data object identifiers

Each data object (this could a data block–sector–on a hard disk or an entire file of data or anything else depending on applications) in our system is assigned an m-bit identifier using a base hash function such as SHA-1. A data object's identifier is produced by hashing data object owner's ID and other auxiliary information. For example, if the data object is the first sector of Alice's computer computer1.alice.org, then the data object identifier could be the hash value of the string "computer1.alice.org||sectorone||Alice||etc", where "etc" is a field that could be used to put other information. For example, in order to keep data availability, each data object should be stored in several geographically different locations (we will further discuss this later) and this is done by using different values of "etc" for each data object and store copies of the data object under these different identifiers. The "etc" field could also contain a private key of Alice so that no one else can trace Alice's data object locations. The length m of the identifier strings should be large enough so that the probability of two different data objects hashing to the same identifier is negligible. We will use the term "data object" to refer to both the data object itself and its identifiers under the hash function.

Each node α in the storage infrastructure pool is assigned an m_α-bit ($m_\alpha \leq m$) identifier string according to its volume capacity. The larger the node volume capacity, the smaller the value of m_α. For example, for a node α with a volume capacity of petabytes, the value of m_α could be 140, and for a node β with a volume capacity of

terabytes, the value of m_β could be 150. A node's identifier is chosen by taking the leftmost m_α bits of hashing output of the node's IP address and other auxiliary information such as node owner's ID. This is done generally when the node registers itself to the storage infrastructure pool. The node identifiers should meet the following requirements: 1) no node identifier is a prefix of another node identifier; 2) no node identifier is equal to another node identifier. In order to guarantee that no node identifier is a prefix of another node identifier, when an m_α-bit identifier is assigned to a node, we need to make sure that no other node has been assigned a identifier β which has α as its prefix. This can always be achieved by changing the information in the auxiliary field of the hash inputs at the registration time. In a large network, it could happen that we cannot find a gap for node identifiers with the required property. This problem could be solved by reserving some prefixes for certain capacity nodes. For example, all identifiers beginning with "0101010101", "1010101010", ... could be reserved for nodes of capacity larger than petebyte.

For the reason of convenience, we convert an m_α-bit node identifier ID to an m-bit identifier ID$'$ by repeatedly appending 1's to the end of ID. From now on, we assume that all node identifiers are m bits via the above conversion. We will use the term "node" to refer to both the node and the m-bit identifier.

Both node identifiers and data object identifiers could be ordered on an identifier circle modulo 2^m. A data object with identifier ID_d is stored on the first node whose identifier equals to or follows the identifier ID_d in the identifier circle (see Figure 1). As in Chord [11], this node is called the *successor node* of the data object ID_d, denoted by $successor(\text{ID}_d)$. If identifiers are represented as a circle of numbers from 0 to $2^m - 1$, then $successor(\text{ID}_d)$ is the first node clockwise from ID_d.

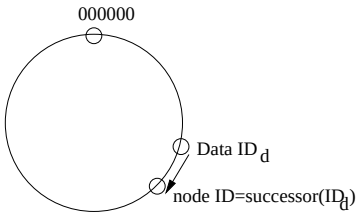

Figure 1: Data objects are stored in their successor nodes

Another way to interpret the identifier circle is to map the identifiers on the identifier circle to the leafs of a tree (if the identifiers are represented by binary strings, then the tree is a binary tree; however, if the identifiers are represented by base b digits, then the tree is a b-branch tree). For example, if identifier strings are represented by base 4 strings, and $m = 6$, then the points on the identifier circle could be interpreted as the leafs of the tree in Figure 2. Note that in the interpretation, we glue the identifier 000 to the right side of the identifier 333 to make it cyclic.

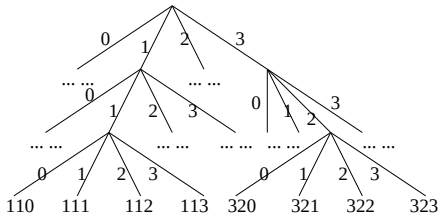

Figure 2: Identifiers are leafs of the tree

As pointed out in [11], one advantage of storing data objects in their successors is that nodes enter and leave the storage infrastructure with minimal disruption. To maintain the consistency of the storage infrastructure, certain data objects need to be relocated when nodes join and leave. In our scheme, if a node α joins the storage infrastructure, some data objects stored in α's successor are relocated to α, and if a node α leaves the storage infrastructure, data objects stored in α are moved to its successor. In a commercial storage infrastructure environment, this can be achieved much efficiently and in a volunteer-participant-based storage environment, this could also be achieved with relative efficiency.

In all other peer-to-peer based systems, each node is assigned an identifier string of the same length (e.g., 160 bits if SHA-1 is used). The advantages of our scheme are: 1) efficient routing for these nodes with short identifiers; 2) balance of the load to nodes in the pool (node with a larger volume capacity receives more data objects proportionally according to its identifier length). Note that in Chord CFS [3] and PAST [4], the volume capacity problem is addressed by letting larger capacity server running multiple "virtual servers" or splitting the node into several nodes. Theoretically, the approach taken by Chord CFS is equivalent to our approach. However, our solution is more natural and much more efficient.

For an l-digit identifier α and an integer $i \leq l$, we will use $\alpha[i]$ to denote the i-th digit of α (that is, $\alpha = \alpha[1] \ldots \alpha[l]$), and $\alpha[1..i]$ to denote $\alpha[1] \ldots \alpha[i]$.

4.2 Routing

The routing protocol in our scheme is a novel combination of the "Scalable Key Location" algorithm in Chord [11] and the longest prefix routing algorithm in Plaxton-Rajamaran-Richa [8] and Tapestry [7, 5] (note that the longest prefix routing algorithm is similar to the algorithm for CIDR IP address allocation architecture in IETF RFC-1518). In Chord, the location of a data object is claimed to be determined with $O(\log N)$ queries (with high probability, this claim is correct. However, this is not true for the

worst case). The advantage of Chord+ is that the location of a data object could be determined in $m/\log b$ queries (with high probability, the number of queries is bounded by $\log_b N$), which could significantly improve the performance if b is chosen appropriately. Note that $b \cdot \log_b N$ entries for the location of other nodes need to be stored at each node for routing purpose in Chord+, this is similar to Plaxton-Rajamaran-Richa and Tapestry. One advantage of Chord+ over Plaxton-Rajamaran-Richa and Tapestry is the simplicity of implementation (similar to Chord).

The identifiers of nodes and data objects are fixed bit sequences (remember that for node identifiers, this is achieved by appending 1's to the end of their m_α-bit identifiers) represented by a common base b (e.g., 40 hexadecimal digits representing 160 bits). In the following, we assume that the base b is fixed and identifiers have l digits under this common base b (that is, $l = m/\log b$). Each node in the storage infrastructure pool acts both as a router and as a storage server.

Each node α has a multiple level finger map which is a novel combination of the neighbor map in Plaxton-Rajamaran-Richa [8] and Tapestry [7, 5] and the finger table in Chord [11]. A finger map consists of l levels and each level consists of b entries. In another word, a finger map for a node α is an $b \times l$ matrix $\mathcal{F}_\alpha$. Each level of the finger map represents a matching "prefix" up to a digit position in the identifier. A given level of the finger map contains a number of entries "close" to the base of the identifier, where the i-th entry ($0 \le i < b$) in the j-th level ($j \le l$) is the identifier and location of the first node $\alpha_{i,j}$ that succeeds the identifier $\alpha[1..j-1]i0\ldots0$ on the identifier circle. That is, $\alpha_{i,j} = successor(\alpha[1..j-1]i0\ldots0)$. In another word, $\mathcal{F}_\alpha(i,j) = \langle \alpha_{i,j},$ location of $\alpha_{i,j}\rangle$. It is easy to see that the entry nodes in α's j-th level ($j \le l$) finger map distributes evenly in an interval of size b^{l-j+1} containing the node α. For example, the 8th entry in the 5th level for node 234AC478CB is the successor node of 234A800000. In addition to the finger map table, each node maintains pointers to its successor and predecessor. An example of finger map is shown in Figure 3.

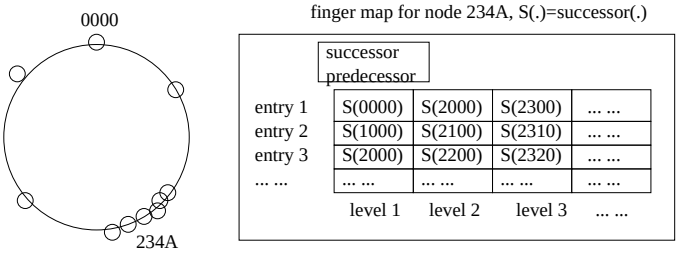

Figure 3: The finger map of node 234A

When a node α needs to know the location of the node which has stored or will store a data object with identifier ID_d, the node α searches its entire finger map (all levels) for a node α' whose identifier most immediately succeeds ID_d, and asks α' for the location of the data object ID_d. By repeating this process, the location of the node with the identifier $successor(\mathrm{ID}_d)$ will be returned to α. For a node α and a data object identifier ID_d, let $\alpha' = \alpha.closest\_succeeding(\mathrm{ID}_d)$ be the unique node in α's finger map (including predecessor) such that there is no other node α'' in α's finger map with the property that $\alpha'' \in (\mathrm{ID}_d, \alpha')$.

When a node α executes $\alpha.closest\_succeeding(\mathrm{ID}_d)$, it first computes the longest shared prefix $\alpha[1..j-1] = \mathrm{ID}_d[1..j-1]$ between α and ID_d. Then it returns the value $\mathcal{F}_\alpha(\mathrm{ID}_d[j], j)$ in the $\mathrm{ID}_d[j]$-th entry of the j-th level of the α's finger map.

$\alpha.find\_successor(\mathrm{ID}_d)$ works by finding the immediate successor node of the data object identifier in α's finger map (including the predecessor). If after $2l$ recursive calls of the procedure $\alpha.find\_successor(\mathrm{ID}_d)$, α still does not find the successor of ID_d, then α initiates an exhaustive search concurrently: Queries for the identifier $successor(\mathrm{ID}_d)$ is passed around the identifier circle via the successor pointers. Thus, if pointers to successor nodes in the system are correct, then it is guaranteed to find the successor node of ID_d even if other parts of the finger maps are incorrect. In the following, we will show that if the finger maps are consistent, then this routing method guarantees that the location of the node $successor(\mathrm{ID}_d)$ is returned in $l = m/\log b$ recursive calls of the routine $\alpha.find\_successor(\mathrm{ID}_d)$, where m is the bit-length of the data object identifier and b is the digit base (e.g., if hexadecimal digit is used, then $b = 16$).

Theorem 4.1 *Assume that the information contained in all finger maps are correct. Then for any data object with identifier ID_d, the number of nodes that must be contacted to find the location of $successor(ID_d)$ is bounded by l. With high probability, the number of nodes that must be contacted to find the location of $successor(ID_d)$ is bounded by $O(\log_b N)$.*

Recall that in Chord, the number of nodes that must be contacted to find the location of $successor(\mathrm{ID}_d)$ is bounded by $O(\log_2 N)$ with high probability (and is bounded by m with probability 1). In several scenarios, the improvement of contacted nodes to $O(\log_b N)$ with high probability (and to l deterministically) by Chord+ is preferred. In a summary, Chord+ inherits the advantages of Chord [11], Plaxton-Rajaraman-Richa [8], and Pastry [4], and overcomes the disadvantages of these schemes. In particular, Chord+ has the following advantages:

- The location of each data object is determined by making a logarithmic number queries.
- Data objects are well balanced even if the storage nodes do not have equal volume capacity.
- Similar to Chord, Chord+ is substantially less complicated than other schemes.
- The base b could be increased (thus the size of the finger map is increased) to reduce the number of queries in the routing process.

There has been an active research on distributed object location algorithms in the past few years. For example, attenuated Bloom filters, minimality and locality balance techniques have been proposed for OceanStore [7, 5]. Some of these technologies could be used to improve the performance of Chord+ also (we will address these issues in a different paper).

4.3 Node insertion and deletion

4.3.1 Node join

In a dynamic network, node can join and leave at any time. Thus special mechanisms should be designed to preserve the ability to locate each data object in the network. Similar to Chord, this could be achieved by preserving the two invariants:

1. Each node's successor is correctly maintained.
2. For each data object with identifier ID_d, node $successor(\mathrm{ID}_d)$ is responsible for it.

The correct finger map for each node is also necessary for fast data objects lookups.

When a new node α is added to the storage infrastructure pool, three things need to be done to preserve the invariants mentioned above:

1. Initialize the finger map of α.
2. Update the finger maps of existing nodes to reflect the addition of α.
3. Initiate the transfer of data objects for which α is responsible from other nodes to α.

We assume that the new node α learns the identity of an existing Chord+ node α' by bootstrap mechanisms such as out of band communication. Node α uses α' to initialize its finger map and to add it to the storage infrastructure.

Populating the finger map and updating finger maps of existing nodes: Node α learns its finger map at each level by asking α' to look up $successor(\alpha)$ and updating and copying finger maps along each hop from α'. Starting with node α', node α attempts to route to node α, and updates the finger map for the j-th hop α_j (note that $\alpha_0 = \alpha'$) if necessary. That is, if α lands between $\beta_{i_0,j_0} = \alpha_j[1..j_0]i_0 0 \ldots 0$ and $\alpha_{i_0,j_0} = successor(\beta_{i_0,j_0})$, and $\mathcal{F}_{\alpha_j}(i_0, j_0+1) = \langle \alpha_{i_0,j_0}, \cdot \rangle$, then α_j updates this entry by letting $\mathcal{F}_{\alpha_j}(i_0, j_0+1) = \langle \alpha, \text{ location of } \alpha \rangle$. Assume that the last hop is α_s and α and α_s shares a j_0-digit prefix, where $j_0 < l$. Then α copies entries in the updated level 1, ..., and level $j_0 + 1$ of α_s's finger map. α fills level $j_0 + 2, \ldots,$ and level l of its finger map by computation on the information contained in $successor(\alpha)$'s finger map.

By Theorem 4.1, the subroutine takes at most $O(\log_b N)$ steps in an N-node network. Thus with high probability, the above algorithm for populating the new node finger map and updating existing node finger maps takes at most $O(\log_b N)$ steps in an N-node network.

Further updating finger maps of existing nodes: In the process of "Populating the finger map and updating finger maps of existing nodes", the finger maps of these nodes that were contacted by the subroutine $\alpha'.find\_successor(\alpha)$ have already been updated. Node α may need to be entered into the finger maps of other existing nodes also. Let $\alpha_p := predecessor(\alpha)$ and $j_0 := \max\{i : \alpha[1..i] = \alpha_p[1..i]\}$. Assume that there are more than two existing nodes already in the network. For any existing node $\beta \neq \alpha_p$, $i < b$, and $j \leq l$, let $\beta_{i,j} := \beta[1..j-1]i0 \ldots 0$. Assume that $\mathcal{F}_\beta(i,j) = \langle \beta', \cdot \rangle$. Then β's finger map needs to be updated if and only if $\alpha \in [\beta_{i,j}, \beta')$ for some $i < b$ and $j \leq l$. In this case, β can update its finger map by letting $\mathcal{F}_{\alpha_j}(i,j) := \langle \alpha, \text{ location of } \alpha \rangle$. The challenging problem is how to find all these nodes β efficiently. If $\alpha \in [\beta_{i,j}, \beta')$, then $\beta_{i,j} \in (\alpha_p, \alpha]$ (otherwise, since $\alpha_p \in (\beta', \alpha)$ we have $\alpha_p \in [\beta_{i,j}, \alpha)$ whence $\alpha_p \in [\beta_{i,j}, \beta')$, which is a contradiction with the fact that $\mathcal{F}_\beta(i,j) = \langle \beta', \cdot \rangle$). Thus if β's finger map needs to be updated only if one of the following two conditions hold:

1. $\beta[1..j_0] = \alpha[1..j_0]$;
2. $\beta[1..j_1] = \alpha_p[1..j_1]$ for some $j_1 < j_0$ and $\alpha_p = \alpha_p[1..j_1+1]0...0$.

We can use this fact to search for all nodes whose finger maps need to be updated.

For an N-node network, the expected distance between every two nodes is $O(2^m/N)$. Thus with high probability, α and α_p share a prefix of $O(\log_b N)$-digits. Hence, with high probability, the value of j_2 is around $O(\log_b N)$. Similar arguments show that there are constant number nodes whose identifiers begin with the prefix $\alpha_p[1..j_2]$. It takes $O(\log_b N)$ steps to find all such kinds of nodes. Thus with high probability, updating all existing nodes' finger maps takes around $O(\log_b N)$ steps.

It should be noted that in our scheme, a node insertion could be finished in $O(\log_b N)$ steps. All other existing node insertion algorithms take $O((\log_2 N)^2)$ steps (see the summary in [5]).

Transferring data objects: Another operation that has to be performed when a node α joins the network is to move responsibility for all data objects for which node α is now the successor. Node α can become the successor only for data objects that were previously the responsibility of the node immediately following α. Thus this operation can be finished by communications between α and $successor(\alpha)$.

Concurrent joins and stabilization will be discussed in the full version of this paper.

4.3.2 Node departure

When a node α leaves the storage infrastructure pool, two things need to be done to preserve the invariants mentioned at the beginning of this section.

1. Update the finger maps of remaining nodes to reflect the deletion of α.

2. Initiate the transfer of data objects for which α is responsible to $successor(\alpha)$.

A node can actively inform the relevant parties of its departure using its pointers and transferring its data objects to its successor, or just leave the pool and rely on the system to remove it over time. In practice, the first choice should always be taken whenever possible since it will remove the burden for the system to remove it. If a node leaves the infrastructure pool without transferring its data objects to its successor node, then these data objects should be recovered from other nodes and copied to its successor during the stabilization process. In order to deal with node departure without notification, (similar to Chord) each entry of the node finger map maintains "successor-list" of r consecutive successors instead of one successor.

In our storage infrastructure (generally a commercial environment), we assume that the node insertion and departure are generally predictable (different from the assumption for peer-to-peer systems). Thus system consistency could be achieved without significant impact on the system performance by: 1) running the dynamic stabilization algorithms at the node insertion and departure time manually; 2) running the dynamic stabilization algorithms automatically at a fixed period (though less frequently compared to peer-to-peer systems) to remove non-notified node departure or to find out network configuration changes..

Approaches to survivability will be discussed in the full version of this paper.

References

[1] G. Bell. A personal digital store. *CACM*, **44**(1):86–91, 2001.

[2] G. Bell and J. Gemmell. A call for the home media network. *CACM*, **45**(7):71–75, 2002.

[3] F. Dabek, M. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage CFS. In *Proc. of the 18th ACM SOSP*, pages 202–215, Banff, Canada, 2001.

[4] P. Druschel and A. Rowstron. PAST: a large-scale persistent peer-to-peer storage utility. *Proc. HotOS Conf.*, IEEE Computer Soc. Press, 2001.

[5] K. Hildrum, J. Kubiatowicz, S. Rao, and B. Zhao. Distributed object location in a dynamic network. In *Proc. SPAA'02*, August, 2002.

[6] D. Karger, E. Lehman, F. Leighton, M. Levine,D. Lewin, and R. Panigrahy. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In: *Proc. of the 29th ACM STOC*, pages 654–663, 1997.

[7] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaten, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, B. Zhao. OceanStore: An architecture for global-scale persistent storage. ASPLOS 2000.

[8] C.Plaxton, R. Rajaraman, and A. Richa. Accessing nearby copies of replicated objects in a distributed environment. In: *Proceedings of ACM SPAA*, ACM Press, June 2001.

[9] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. *Proc. SIGCOMM Conf.*, pages 161–172, ACM Press, New York, 2001.

[10] A. Rowstron and P. Druschel. Pastry: scalable, distributed object location and routing for large-scale peer-to-peer systems. In: *Proc. of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, Nov. 2001.

[11] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. Kaashoek, and F. Dabek. Chord, a scalable peer-to-peer lookup protocol for Internet applications. *Proc. SIGCOMM Conf.*, ACM Press, New York, 2001.