PAST: A large-scale, persistent peer-to-peer storage utility

Peter Druschel
Rice University, Houston, TX 77005, USA*
druschel@cs.rice.edu

Antony Rowstron
Microsoft Research, Cambridge, CB2 3NH, UK
antr@microsoft.com

Abstract

This paper sketches the design of PAST, a large-scale, Internet-based, global storage utility that provides scalability, high availability, persistence and security. PAST is a peer-to-peer Internet application and is entirely self-organizing. PAST nodes serve as access points for clients, participate in the routing of client requests, and contribute storage to the system. Nodes are not trusted, they may join the system at any time and may silently leave the system without warning. Yet, the system is able to provide strong assurances, efficient storage access, load balancing and scalability.

Among the most interesting aspects of PAST's design are (1) the Pastry location and routing scheme, which reliably and efficiently routes client requests among the PAST nodes, has good network locality properties and automatically resolves node failures and node additions; (2) the use of randomization to ensure diversity in the set of nodes that store a file's replicas and to provide load balancing; and (3) the optional use of smartcards, which are held by each PAST user and issued by a third party called a broker. The smartcards support a quota system that balances supply and demand of storage in the system.

1 Introduction

There are currently many projects aimed at constructing peer-to-peer applications and understanding more of the issues and requirements of such applications and systems [1, 2, 4, 5, 6, 7]. Peer-to-peer systems can be characterized as distributed systems in which all nodes have identical capabilities and responsibilities and all communication is symmetric. We are developing PAST, an Internet-based, peer-to-peer global storage utility, which aims to provide strong persistence, high availability, scalability and security.

The PAST system is composed of nodes connected to the Internet, where each node is capable of initiating and routing client requests to insert or retrieve files. Optionally, nodes may also contribute storage to the system. The PAST nodes form a self-organizing overlay network. Inserted files are replicated on multiple nodes to ensure persistence and availability. With high probability, the set of nodes over which a file is replicated is diverse in terms of geographic location, ownership, administration, network connectivity, rule of law, etc. Additional copies of popular files may be cached in any PAST node to balance query load.

A storage utility like PAST is attractive for several reasons. First, it exploits the multitude and diversity (in geography, ownership, administration, jurisdiction, etc.) of nodes in the Internet to achieve strong persistence and high availability. This obviates the need for physical transport of storage media to protect backup and archival data; likewise, it renders unnecessary the explicit mirroring of shared data for high availability and throughput. A global storage utility also facilitates the sharing of storage and bandwidth, thus permitting a group of nodes to jointly store or publish content that exceeds the capacity of any individual node.

While PAST offers persistent storage services, its access semantics differ from that of a conventional filesystem. Files stored in PAST are associated with a quasi-unique *fileId* that is generated at the time of the file's insertion into PAST. Therefore, files stored in PAST are *immutable* since a file cannot be inserted multiple times with the same fileId. Files can be shared at the owner's discretion by distributing the fileId (potentially anonymously) and, if necessary, a decryption key. PAST does not support a *delete* operation. Instead, the owner of a file may *reclaim* the storage associated with a file, which does not guarantee that the file is no longer available. These weaker semantics avoid agreement protocols among the nodes storing the file.

An efficient routing scheme called *Pastry* [11] ensures that client requests are reliably routed to the appropriate nodes. Client requests to *retrieve* a file are routed to a node that is "close in the network"[1] to the client that issued the request, among all live nodes that store the requested file. The number of PAST nodes traversed while routing a client

*Work done in part while visiting Microsoft Research, Cambridge, UK.

[1] Network proximity is based on a scalar metric, such as the number of IP hops, geographic distance, or a combination of these and other factors.

request is at most logarithmic in the total number of PAST nodes in the system under normal operation.

A storage management scheme in PAST ensures that the global storage utilization in the system can approach 100%, despite the lack of centralized control and widely differing file sizes and storage node capacities [12]. In a decentralized storage system where nodes are not trusted, an additional mechanism is required that ensures a balance of storage supply and demand. Towards this end, PAST includes a secure quota system. In simple cases, users are assigned fixed quotas, or they are allowed to use as much storage as they contribute. Optionally, organizations called *brokers* may trade storage and issue smartcards to users, which control how much storage must be contributed and/or may be used. The broker is not directly involved in the operation of the PAST network, and its knowledge about the system is limited to the number of smartcards it has circulated, their quotas and expiration dates.

Another issue in peer-to-peer systems, and particularly in storage and file-sharing systems, is privacy and anonymity. A provider of storage space used by others may not want to risk prosecution for content it stores, and clients inserting or retrieving files may not wish to reveal their identity. PAST clients and storage providers need not trust each other, and place only limited trust in brokers. In particular, all nodes trust the brokers to facilitate the operation of a secure PAST network by balancing storage supply and demand via responsible use of the quota system. On the other hand, users need not reveal to brokers (or anyone else) their identity, the files they are retrieving, inserting or storing. Each user holds an *initially unlinkable pseudonym* [8] in the form of a public key. The pseudonym is not easily linkable to the user's identity, unless the user voluntarily reveals the binding. If desired, a user may use multiple pseudonyms to obscure that certain operations were initiated by the same user. To provide stronger levels of anonymity and other properties such as anti-censorship, additional mechanisms may be layered on top of PAST [14, 15, 16].

2 PAST design

Some of the key aspects of PAST's architecture are (1) the Pastry routing scheme, which routes client requests in less than $\lceil log_{16}N \rceil$ steps on average within a self-organizing, fault tolerant overlay network; (2) the use of randomization to ensure probabilistic storage load balancing and diversity of nodes that store replicas of a file, without the need for centralized control or expensive distributed agreement protocols; (3) a decentralized storage management and caching scheme that balances the storage utilization among the nodes as the total utilization of the system approaches 100%, and balances query load by caching copies of popular files close to interested clients; and, (4) the optional use of smartcards, which support a quota system to control storage supply and demand.

PAST is composed of nodes connected to the Internet. Each node can act as a storage node and a client access point and is assigned a 128-bit node identifier (nodeId), derived from a cryptographic hash of the node's public key. Each file that is inserted into PAST is assigned a 160-bit fileId, corresponding to the cryptographic hash of the file's textual name, the owner's public key and a random salt. Before a file is inserted, a file certificate is generated, which contains the fileId, its replication factor k, the salt, the insertion date and a cryptographic hash of the file's content. The file certificate is signed by the file's owner.

When a file is inserted in PAST, Pastry routes the file to the k nodes whose node identifiers are numerically closest to the 128 most significant bits of the file identifier (fileId). Each of these nodes then stores a copy of the file. The replication factor k depends on the availability and persistence requirements of the file and may vary between files. A lookup request for a file is routed towards the live node with a nodeId that is numerically closest to the requested fileId.

This procedure ensures that (1) a file remains available as long as one of the k nodes that store the file is alive and reachable via the Internet; (2) with high probability, the set of nodes that store the file is diverse in geographic location, administration, ownership, network connectivity, rule of law, etc.; and, (3) the number of files assigned to each node is roughly balanced. (1) follows from the properties of the PAST routing algorithm described in Section 2.2. (2) and (3) follow from the uniformly distributed, quasi-random identifiers assigned to each node and file.

In the following, we discuss some of the key aspects of PAST's design, namely security, routing and content location, self-organization, storage management and caching.

2.1 Security

PAST's security model is based on the following assumptions: (1) It is computationally infeasible to break the public-key cryptosystem and the cryptographic hash function used in PAST; (2) while clients, node operators and node software are not trusted and attackers may control the behavior of individual PAST nodes, it is assumed that most nodes in the overlay network are well behaved; and, (3) an attacker cannot control the behavior of the smartcards.

In the following discussion, we assume the use of smartcards. As discussed later in this section, it is possible to operate a PAST network without smartcards. Each PAST node and each user of the system hold a smartcard. A private/public key pair is associated with each card. Each smartcard's public key is signed with the smartcard issuer's private key for certification purposes. The smartcards generate and verify various certificates used during insert and

reclaim operations and they maintain storage quotas. Next, we sketch the main security related functions.

Generation of nodeIds A smartcard provides the nodeId for an associated PAST node. The nodeId is based on a cryptographic hash of the smartcard's public key. This assignment of nodeIds probabilistically ensures uniform coverage of the space of nodeIds and diversity of nodes with adjacent nodeIds, in terms of geographic location, network attachment, ownership, rule of law, etc. Furthermore, nodes can verify the authenticity of each other's nodeIds.

Generation of file certificates and store receipts The smartcard of a user wishing to insert a file into PAST issues a *file certificate*. The certificate contains a cryptographic hash of the file's contents (computed by the client node), the fileId (computed by the smartcard), the replication factor, the salt, and is signed by the smartcard. During an insert operation, the file certificate allows each storing node to verify (1) that the user is authorized to insert the file into the system, which prevents clients from exceeding their storage quotas; (2) that the contents of the file arriving at the storing node have not been corrupted en route from the client by faulty or malicious intermediate nodes; and, (3) that the fileId is authentic, thus defeating denial-of-service attacks where malicious clients try to exhaust storage at a subset of PAST nodes by choosing fileIds with nearby values. Each storage node that has successfully stored a copy of the file then issues and returns a *store receipt* to the client, which allows the client to verify that k copies of the file have been created on nodes with adjacent nodeIds, which prevents a malicious node from suppressing the creation of k diverse replicas. During a retrieve operation, the file certificate is returned along with the file, and allows the client to verify that the contents are authentic.

Generation of reclaim certificates and receipts Prior to issuing a reclaim operation, the user's smartcard generates a *reclaim certificate*. The certificate contains the fileId, is signed by the smartcard and is included with the reclaim request that is routed to the nodes that store the file. When processing a reclaim request, the smartcard of a storage node first verifies that the signature in the reclaim certificate matches that in the file certificate stored with the file. This prevents users other than the owner of the file from reclaiming the file's storage. If the reclaim operation is accepted, the smartcard of the storage node generates a *reclaim receipt*. The receipt contains the reclaim certificate and the amount of storage reclaimed; it is signed by the smartcard and returned to the client.

Storage quotas The smartcard maintains storage quotas. Each user's smartcard is issued with a usage quota, depending on how much storage the client is allowed to use. When a file certificate is issued, an amount corresponding to the file size times the replication factor is debited against the quota. When the client presents an appropriate reclaim receipt issued by a storage node, the amount reclaimed is credited against the client's quota. This prevents clients from exceeding the storage quota they have paid for. A node's smartcard specifies the amount of storage contributed by the node (possibly zero). Nodes are randomly audited to see if they can produce files they are supposed to store, thus exposing nodes that cheat by offering less storage than indicated by their smartcard.

In the following, we briefly discuss how some of the system's key properties are maintained.

System integrity Several conditions ensure the basic integrity of a PAST system. Firstly, to maintain approximate load balancing among storage nodes, the nodeIds and fileIds should each be uniformly distributed. The procedure for generating and verifying nodeIds and fileIds ensures this. Secondly, there must be a balance between the sum of all client quotas (potential demand) and the total available storage in the system (supply). The broker ensures that balance, potentially using the monetary price of storage to regulate supply and demand. Thirdly, individual malicious nodes must be incapable of persistently denying service to a client. A randomized routing protocol, described in Section 2.2, ensures that a retried operation will eventually be routed around the malicious node.

Persistence File persistence in PAST depends primarily on three conditions. (1) Unauthorized users are prevented from reclaiming a file's storage, (2) the file is stored on k storage nodes, and (3) there is sufficient diversity in the set of storage nodes that store a file. By issuing and requiring reclaim certificates, the smartcards ensure condition (1). (2) is enforced through the use of store receipts and (3) is ensured by the quasi-random distribution of nodeIds, which can't be biased by an attacker. The choice of a replication factor k must take into account the expected rate of transient storage node failures to ensure sufficient availability. In the event of storage node failures that involve loss of the stored files, the system automatically restores k copies of a file as part of a failure recovery procedure [12].

Data privacy and integrity Users may use encryption to protect the privacy of their data, using a cryptosystem of their choice. Data encryption does not involve the smartcards. Data integrity is ensured by means of the file certificates issued by the smartcards.

Pseudonymity A user's smartcard signature is the only information associating a stored file or a request with the responsible user. The association between a smartcard and the user's identity is only known to the user, unless the user voluntarily releases this information. Pseudonymity of storage nodes is similarly ensured because the node's smartcard signature is not linkable to the identity of the node operator. Moreover, the Pastry routing scheme avoids the widespread

dissemination of information about the mapping between nodeIds and IP addresses.

Smartcards Next, we briefly reflect on the role of smartcards and brokers in PAST. The use of smartcards and even the presence of brokers as trusted third parties are not fundamental to PAST's design. First, smartcards could be replaced by secure on-line quota services run by the brokers. Second, it is possible to run PAST without a third party. However, given today's technology, the smartcards/brokers solve several issues efficiently:

(1) The smartcards/brokers ensure the integrity of nodeId and fileId assignment. Without a third party, it more difficult and expensive to prevent attackers from choosing, by trial and error, fileIds or nodeIds that fall between two adjacent existing PAST nodeIds.

(2) The smartcards maintain storage quotas securely and efficiently. Achieving the same scalability and efficiency with an on-line quota service is difficult. Enforcing quotas in the absence of a trusted third party would likely require complex agreement protocols.

(3) The smartcards are a convenient medium through which a user can obtain necessary credentials to join the system in an anonymous fashion. A user can obtain a smartcard with the desired quota from a retail outlet anonymously in exchange for cash. Obtaining the credentials on-line carries the risk of revealing the user's identity or leaking sensitive information to third parties.

There are disadvantages to the use of smartcards. First, clients need to obtain a card and periodically replace it (e.g., every year) to ensure key freshness. Second, sophisticated, resource-rich attackers could compromise a smartcard, permitting them to cheat against the storage quota and mount certain limited denial-of-service attacks until the card is revoked or expires.

Finally, there are performance costs due to the limited processing speed and I/O performance of smartcards. Fortunately, read operations involve no smartcard operations. (In fact, read-only users do not need a smartcard). Write operations require a file certificate verification and a store receipt generation, and we expect that a smartcard keeps up with the speed of a single disk. Larger storage nodes use multiple smartcards, and very large storage nodes may require more powerful tamperproof hardware. Professionally managed storage sites also have the option of contracting with a broker, thus obviating the need for trusted hardware.

Future Internet technologies like an anonymous transactions and micropayment infrastructure could obviate the need for smartcards in PAST. For instance, micro-payments could be used to balance the supply and demand of storage without quotas, and anonymous transactions could make it possible for a user to securely and anonymously obtain necessary credentials, including nodeIds and fileIds. We plan to re-evaluate the use of smartcards as alternatives become available.

It is to be noted that multiple PAST systems can co-exist in the Internet. In fact, we envision PAST networks run by many competing brokers, where a client can access files in the entire system. Furthermore, it is possible to operate isolated PAST systems that serve a mutually trusting community without a broker or smartcards. In these cases, a virtual private network (VPN) can be used to interconnect the system's nodes.

2.2 Pastry

We now briefly describe Pastry, the location and routing scheme used by PAST. Given a fileId, Pastry routes the associated message towards the node whose nodeId is numerically closest to the 128 most significant bits (msb) of the fileId, among all live nodes. Given the invariant that a file is stored on the k nodes whose nodeIds are numerically closest to the 128 msbs of the fileId, it follows that a file can be located unless all k nodes have failed simultaneously (i.e., within a recovery period).

Pastry is highly efficient, scalable, fault resilient and self-organizing. Assuming a PAST network consisting of N nodes, Pastry can route to the numerically closest node to a given fileId in less than $\lceil log_{2^b} N \rceil$ steps on average (b is a configuration parameter with typical value 4). With concurrent node failures, eventual delivery is guaranteed unless $\lfloor l/2 \rfloor$ nodes with *adjacent* nodeIds fail simultaneously (l is a configuration parameter with typical value 32).

The tables required in each PAST node have only $(2^b - 1) * \lceil log_{2^b} N \rceil + 2l$ entries, where each entry maps a nodeId to the associated node's IP address. Moreover, after a node failure or the arrival of a new node, the invariants in all affected routing tables can be restored by exchanging $O(log_{2^b} N)$ messages among the affected nodes. In the following, we briefly sketch the Pastry routing algorithm.

For the purpose of routing, nodeIds and fileIds are thought of as a sequence of digits with base 2^b. A node's routing table is organized into $\lceil log_{2^b} N \rceil$ levels with $2^b - 1$ entries each. The $2^b - 1$ entries at level n of the routing table each refer to a node whose nodeId matches the present node's nodeId in the first n digits, but whose $n + 1$th digit has one of the $2^b - 1$ possible values other than the $n + 1$th digit in the present node's id. The uniform distribution of nodeIds ensures an even population of the nodeId space; thus, only $\lceil log_{2^b} N \rceil$ levels are populated in the routing table. Each entry in the routing table points to one of potentially many nodes whose nodeIds have the appropriate prefix. Among such nodes, the one closest to the present node, according to the proximity metric, is chosen in practice.

In addition to the routing table, each node maintains IP addresses for the nodes in its *leaf set*, i.e., the set of nodes

with the $l/2$ numerically closest larger nodeIds, and the $l/2$ nodes with numerically closest smaller nodeIds, relative to the present node's nodeId.

In each routing step, a node normally forwards the message to a node whose nodeId shares with the fileId a prefix that is at least one digit (or b bits) longer than the prefix that the fileId shares with the present node's id. If no such node exists, the message is forwarded to a node whose nodeId shares a prefix with the fileId as long as the current node, but is numerically closer to the fileId than the present node's id. It follows from the definition of the leaf set that such a node exists unless $\lfloor l/2 \rfloor$ adjacent nodes in the leaf set have failed simultaneously.

Locality Next, we briefly discuss Pastry's locality properties vis-à-vis the proximity metric. Recall that the entries in the node routing tables are chosen to refer to a nearby node, in terms of the proximity metric, with the appropriate nodeId prefix. As a result, in each step a message is routed to a "nearby" node with a longer prefix match (by one digit). This local heuristic clearly can't achieve globally shortest routes, but simulations have shown that the average distance traveled by a message, in terms of the proximity metric, is only 50% higher than the corresponding "distance" of the source and destination in the underlying network [11].

Moreover, since Pastry repeatedly takes a locally "short" routing step, messages have a tendency to first reach a node, among the k nodes that store the requested file, that is near the client, according to the proximity metric. One experiment shows that among 5 replicated copies of a file, Pastry is able to find the "nearest" copy in 76% of all lookups and it finds one of the two "nearest" copies in 92% of all lookups [11].

Node addition and failure A key design issue in Pastry is how to efficiently and dynamically maintain the node state, i.e., the routing table, leaf set and neighborhood sets, in the presence of node failures, node recoveries, and new node arrivals. The protocol is described and evaluated in [11].

Briefly, an arriving node with the new nodeId X can initialize its state by contacting a nearby node A (according to the proximity metric) and asking A to route a special message to the existing node Z with nodeId numerically closest to X. X then obtains the leaf set from Z, the neighborhood set from A, and the ith row of the routing table from the ith node encountered along the route from A to Z. One can show that using this information, X can correctly initialize its state and notify interested nodes that need to know of its arrival, thereby restoring all of Pastry's invariants.

To handle node failures, neighboring nodes in the nodeId space (which are aware of each other by virtue of being in each other's leaf set) periodically exchange keep-alive messages. If a node is unresponsive for a period T, it is presumed failed. All members of the failed node's leaf set are then notified and they update their leaf sets to restore the invariant. Since the leaf sets of nodes with adjacent nodeIds overlap, this update is trivial. A recovering node contacts the nodes in its last known leaf set, obtains their current leafs sets, updates its own leaf set and then notifies the members of its presence. Routing table entries that refer to failed nodes are repaired lazily; the details are described in [11].

Fault-tolerance The routing scheme as described so far is deterministic, and thus vulnerable to malicious or failed nodes along the route that accept messages but do not correctly forward them. Repeated queries could thus fail each time, since they are likely to take the same route.

To overcome this problem, the routing is actually randomized. To avoid routing loops, a message must always be forwarded to a node that shares at least as long a prefix with, but is numerically closer to the destination node in the namespace than the current node. The choice among multiple suitable nodes is random. In practice, the probability distribution is heavily biased towards the best choice to ensure low average route delay. In the event of a malicious or failed node along the path, the query may have to be repeated several times by the client, until a route is chosen that avoids the bad node.

2.3 Storage management and caching

The statistical assignment of files to storage nodes in PAST approximately balances the number of files stored at each node. However, non-uniform storage node capacities and file sizes require more explicit storage load balancing to permit graceful behavior under high global storage utilization; and, non-uniform popularity of files requires caching to minimize fetch distance and to balance the query load.

PAST employs a storage management scheme that achieves high global storage utilization while rejecting few file insert requests. The scheme relies only on local coordination among the nodes in a leaf set, and imposes little overhead. Experimental results show that PAST can achieve global storage utilization in excess of 95%, while the rate of rejected file insertions remains below 5% and failed insertions are heavily biased towards large files [12].

Any PAST node can cache additional copies of a file, which achieves query load balancing, high throughput for popular files, and reduces fetch distance and network traffic. Storage management and caching are described in [12].

3 Related work

There are currently several peer-to-peer systems in use, and many more are under development. Among the most prominent are file sharing facilities, such as Gnutella [2] and Freenet [5]. The Napster [1] music exchange service provided much of the original motivation for peer-to-peer systems, but it is not a pure peer-to-peer system because its

database is centralized. All three systems are primarily intended for the large-scale sharing of data files; persistence and reliable content location are not guaranteed or necessary in this environment.

In comparison, PAST aims at combining the scalability and self-organization of systems like FreeNet with the strong persistence and reliability expected of an archival storage system. In this regard, it is more closely related with projects like OceanStore [7], FarSite [4], FreeHaven [6], and Eternity [3]. FreeNet, FreeHaven and Eternity are more focused on providing strong anonymity and anti-censorship.

OceanStore provides a global, transactional, persistent storage service that supports serializable updates on widely replicated and nomadic data. In contrast, PAST provides a simple, lean storage abstraction for persistent, immutable files with the intention that more sophisticated storage semantics be built on top of PAST if needed.

FarSite has more traditional filesystem semantics, while PAST is more targeted towards global, archival storage. Farsite uses a distributed directory service to locate content; this is different from PAST's Pastry scheme, which integrates content location and routing.

Pastry, along with Tapestry [17], Chord [13] and CAN [10], represent a second generation of peer-to-peer routing and location schemes that were inspired by the pioneering work of systems like FreeNet and Gnutella. Unlike that earlier work, they guarantee a definite answer to a query in a bounded number of network hops, while retaining the scalability of FreeNet and the self-organizing properties of both FreeNet and Gnutella.

Pastry and Tapestry bear some similarity to the work by Plaxton et al [9]. The approach of routing based on address prefixes, which can be viewed as a generalization of hypercube routing, is common to all three schemes. However, the Plaxton scheme is not self-organizing and it associates a "root" node with each file, which forms a single point of failure. Pastry and Tapestry differ in their approach to achieving network locality and to replicating objects, and Pastry appears to be less complex.

The Chord protocol is closely related to Pastry, but instead of routing based on address prefixes, Chord forwards messages based on numerical difference with the destination address. Unlike Pastry, Chord makes no explicit effort to achieve good network locality.

CAN routes messages in a d-dimensional space, where each node maintains a routing table with $O(d)$ entries and any node can be reached in $O(dN^{1/d})$ routing hops. Unlike Pastry, the routing table does not grow with the network size, but the number of routing hops grows faster than $logN$.

References

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

[2] The Gnutella protocol specification, 2000. http://dss.clip2.com/GnutellaProtocol04.pdf.

[3] R. Anderson. The Eternity service. In *Proc. PRAGOCRYPT'96*, pages 242–252. CTU Publishing House, 1996. Prague, Czech Republic.

[4] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *Proc. ACM SIGMETRICS'2000*, pages 34–43, 2000.

[5] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Workshop on Design Issues in Anonymity and Unobservability*, pages 311–320, July 2000. ICSI, Berkeley, CA, USA.

[6] R. Dingledine, M. J. Freedman, and D. Molnar. The Free Haven project: Distributed anonymous storage service. In *Proc. Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, CA, July 2000.

[7] J. Kubiatowicz, D. Bindel, P. Eaton, Y. Chen, D. Geels, R. Gummadi, S. Rhea, W. Weimer, C. Wells, H. Weatherspoon, and B. Zhao. Oceanstore: An architecture for global-scale persistent store. In *Proc. ACM ASPLOS'2000*, Cambridge, MA, November 2000.

[8] A. Pfitzmann and M. Köhntopp. Anonymity, unobservability, and pseudonymity: A proposal for terminology, Apr. 2001. http://www.koehntopp.de/marit/pub/anon/Anon_Terminology_IHW.pdf.

[9] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. *Theory of Computing Systems*, 32:241–280, 1999.

[10] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM'01*, San Diego, CA, Aug. 2001.

[11] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware 2001*, Heidelberg, Germany, Nov. 2001.

[12] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. ACM SOSP'01*, Banff, Canada, Oct. 2001.

[13] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. ACM SIGCOMM'01*, San Diego, CA, Aug. 2001.

[14] A. Stubblefield and D. S. Wallach. Dagster: Censorship-resistant publishing without replication. Technical Report TR01-380, Rice University, Houston, Texas, July 2001.

[15] M. Waldman and D. Mazières. Tangler - a censorship resistant publishing system based on document entanglements. In *Eighth ACM Conference on Computer and Communications Security*, Nov. 2001.

[16] M. Waldman, A. D. Rubin, and L. F. Cranor. Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In *Proc. 9th USENIX Security Symposium*, pages 59–72, Denver, CO, August 2000.

[17] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, April 2001.