Routing, Storage Management and Caching, and Security of Peer-to-Peer Storage Systems

Yingwu Zhu and Yiming Hu
Department of Electrical & Computer Engineering and Computer Science
University of Cincinnati
e-mail: {zhuy, yhu}@ececs.uc.edu

Abstract

The popularity of peer-to-peer (P2P) file sharing systems such as Napster, Gnutella and Freenet, has inspired a whole new breed of P2P storage systems, which aims to harness the rapid growth of network bandwidth and storage capacity to provide inexpensive, scalable, fault-tolerant, and highly-available storage without centralized servers. Many P2P storage systems have been proposed and described individually. However, there is still no systematic way to compare and contrast them. In this paper we present a framework for the evaluation of the P2P storage system. We review a set of P2P storage systems including Napster, Gnutella, Freenet, CFS, PAST and Oceanstore, and give a comparative summary of these systems. Finally, we propose some potential questions for future research.

1 Introduction

The popularity of peer-to-peer (P2P) file sharing systems such as Napster [1], Gnutella [2] and Freenet [3] has inspired a whole new breed of distributed, peer-to-peer storage systems. Unlike traditional distributed systems, systems such as CFS [4], PAST [5], and OceanStore [6] aim to harness the rapid growth of network bandwidth and storage capacity to provide inexpensive, scalable, fault-tolerant, and highly-available storage without centralized servers. In such systems, all nodes have identical capabilities and responsibilities (as clients, severs and routers), all communication is symmetric, nodes are able to join and leave the system frequently without affecting its robustness or efficiency, and load should be balanced across the available nodes.

Many P2P storage systems have been proposed and described individually. However, as far as we know, there is still no systematic way to compare and contrast them. A review of the features provided by these P2P storage systems yields a long list. These include load balancing, scalability, persistence, availability, security, and self-maintenance. A variety of design choices have been proposed to achieve these goals. These choices, however, may affect the end goals they achieve. Therefore, it is very important and meaningful for us to present a framework to describe and compare design choices of these systems and the level of end goals they offer. The evaluation framework comprises of a set of qualitative and quantitative evaluation criteria, which can be applied to any P2P storage system. We also use this model as a tool to identify the potential problems in existing systems. Our evaluation framework splits the evaluation process into three main domains, namely location & routing system, storage management and caching, and security.

As an essential building block in P2P storage systems, any location & routing system should implement a core set of functions, which include naming mechanism, structuring mechanism, location & routing algorithm, routing fault-tolerance, network locality and self-organization. After examining these functions, we come to compare a set of location & routing systems in terms of performance, scalability, reliability and maintenance.

Storage management includes load balancing, scalability, availability, persistence, and self-maintenance.

The security aspect of P2P storage systems covers issues related to data integrity, data privacy, anonymity, and prevention of potential attacks.

Even though performance is an important criterion to evaluate storage systems, our evaluation framework doesn't intend to apply such a criterion to P2P storage systems. This is because the performance of P2P storage systems is affected by a number of factors, including the efficiency of location & routing system, the efficiency of storage management and caching, as well as the overhead of security. It is not easy for us to compare the performance among these P2P storage systems.

The rest of this paper is organized as follows. Section 2 describes P2P storage systems we will address in this paper. Section 3 defines our framework for P2P storage systems. Section 4 describes and evaluates the location & routing systems.

Section 5 evaluates storage management and caching on P2P storage systems. Section 6 evaluates the security aspect of P2P storage systems. Finally, we outline items for future work in Section 7 and conclude in Section 8.

2 Current P2P Storage Systems

There are currently several P2P systems in use, like Napster [1], Gnutella [2] and Freenet [3], and some are under development, such as CFS [4], PAST [5], and OceanStore [6].

Napster and Gnutella are the most widely used P2P file sharing systems built from a large group of independent nodes through Internet. They are primarily intended for large-scale sharing of data files (e.g, MP3 files, video files, etc.). In both systems, files are stored on the computers of users or peers, and exchanged through a direct connection between the downloading and uploading peers over an HTTP-style protocol. Napster performs search at a large cluster of central servers, which maintains an index of the files that are currently being shared by active peers. However, the existence of central servers limits scalability and poses a single point of failure. Gnutella, as a P2P Napster clone, instead broadcasts search queries to many peers by a controlled flood.

Freenet is a file sharing system that permits the publication, replication, and retrieval of data while protecting anonymity of both authors and readers. It operates as a network of peers that collectively pool their storage to store data files and cooperate to route queries to the most likely physical location of data. No broadcast or centralized location index is employed.

Systems such as CFS, PAST and OceanStore are built on top of P2P distributed hash location & routing systems [7, 8, 9, 10]. Location & routing for keys are performed by routing queries through a set of nodes; each of these nodes uses a local routing table to forward the query toward the node that is ultimately responsible for the key.

CFS is a completely decentralized, P2P read-only storage system developed at MIT. It provides provable guarantees for the efficiency, robustness and load-balance of file storage and retrieval. CFS is layered on top of Chord [7], an efficient distributed lookup system based on consistent hashing.

PAST is a large-scale, P2P archival storage system that provides scalability, availability, security and cooperative resource sharing. Files in PAST are immutable and can be shared at the discretion of their owners. PAST is built on top of Pastry [8], a generic, scalable and efficient lookup system.

OceanStore is a global-scale persistent storage system developed at University of California, Berkeley. It provides data privacy, guarantees durable storage, and supports serializable updates on widely replicated and nomadic data. OceanStore is layered on top of Tapestry [9], a self-organizing, scalable, fault-tolerant location & routing infrastructure.

3 Evaluation Framework

In this section, we distill out the common characteristics among existing P2P storage systems into an evaluation framework. This framework consists of three domains: location & routing system, storage management and caching, and security.

3.1 Location & Routing System

As an essential building block in P2P systems, the location & routing system must be efficient, scalable, fault-resilient, and self-organizing, in the presence of heavy load and network and node failures. Any location & routing system should implement a set of core functions, although they may differ in the detailed design choices.

Naming mechanism It is used to represent both content namespace and node's address namespace. Using hierarchical name spaces increases scalability and eases maintenance. In most P2P systems, however, content namespace and node's address space are pseudorandom, fixed-length bit strings. They are flat, large, and uniformly populated.

Structuring mechanism It includes the overlay network topology and the routing table (lookup structure) maintained at each node. Keeping the size of routing tables small or independent of the number of nodes allows the system to be scalable to a large number of nodes.

Location & routing algorithm It is used to route client queries/requests among the nodes. Unifying location and routing into a single operation can route queries quickly with minimal network overhead. Efficient algorithms improve scalability as well as performance.

Routing fault tolerance It means the ability to route around failures because of the availability of multiple paths between two nodes in the system.

Network locality It means the ability to route a query to a nearby node which holds a copy of the data being requested, in terms of network proximity metric, thus increasing scalability and performance.

Node addition/deletion It includes the ability to handle node addition/ deletion. The key design issue is how to efficiently and dynamically maintain the node state (e.g., the routing table, etc.) in the presence of node additions and node failures. Automatic reorganization of the topology eases maintenance.

3.2 Storage Management and Caching

Storage management aims to exploit and aggregate the multitude and diversity of nodes (in geography, ownership, administration, jurisdiction, etc.) in the system to achieve load balancing, scalability, availability, persistence, and self-maintenance, while caching attempts to minimize fetch distance and balance the query load in the presence of non-uniform popularity of data.

Load balancing It aims to efficiently distribute load among nodes despite the wide variation in the sizes and popularities of files, as well as the diversity of storage capacities of individual nodes.

Scalability It means the ability to remain tractable as the number of nodes and the amount of data increase. The scalability of a P2P storage system depends on the scalability of its location & routing infrastructure as well as the efficiency of its load balancing.

Availability It keeps data available in the presence of network partition and node failures, by the support of data replication, network link redundancy, and node failure detection and recovery. Caching also improves data availability.

Persistence It guarantees the durability of data being stored in the system. It depends primarily upon two factors: preventing unauthorized users from removing data, and maintaining and distributing data replicas over diverse nodes.

Self-maintenance It includes two fundamental properties: fault tolerance and automatic recovery. Fault tolerance, without automatic recovery, is not sufficient for self-maintenance, since further failures might disrupt a system.

Caching It creates and maintains additional data copies to minimize client access latency (in terms of the overlay network routing hops), maximize the query throughput and balance the query load in the system. Caching provides no guarantee for data persistence because it has a bias towards popular data. The primary purpose of caching is for performance consideration.

3.3 Security

Security is of growing importance in an untrusted, dynamically changing environment on P2P systems. It should protect the integrity and privacy of data, provide some kind of anonymity and prevent potential attacks.

Data integrity It guarantees that data modified by a malicious intruder can be detected by using cryptography.

Data privacy It prevents unauthorized users from sharing data by using encryption.

Anonymity It includes publisher-anonymity, reader-anonymity, server-anonymity and document-anonymity. Publisher-anonymity prevents an adversary from linking a publisher to a document because clients inserting files may not wish to reveal their identification. Reader-anonymity protects the privacy of users because clients retrieving files may not wish to expose their identification. Server-anonymity means no server can be linked to a document. Document-anonymity means that a server doesn't know which document it is storing. Server-anonymity and document-anonymity are crucial because a provider of storage space used by others may not want to risk prosecution for documents it is storing.

Preventing attacks It aims to prevent potential attacks from both malicious users and misbehaving nodes. For example, it must prevent a malicious user from overloading targeted nodes by injecting large quantities of useless data, and prevent a malicious node from denying the existence of data it is responsible for.

	Napster	Gnutella	Freenet	Chord	Pastry	Tapestry	CAN
decentralized	no	yes	yes	yes	yes	yes	yes
routing table[1]	$O(N)$	–	–	$O(\log N)$	$O(\log N)$	$O(\log N)$	$O(d)$[2]
lookup cost[3]	$O(1)$	unbounded	unbounded	$O(\log N)$	$O(\log N)$	$O(\log N)$	$O(dN^{1/d})$[2]
node addition[4]	$O(1)$	$O(N)$	–	$O(\log^2 N)$	$O(\log N)$	$O(\log N)$	$O(d)$[2]
node deletion[4]	$O(1)$	$O(N)$	–	$O(\log^2 N)$	$O(\log N)$	$O(\log N)$	$O(d)$[2]
network locality	no	–	no	no	yes	yes	no

Table 1: *Summary of features provided by different location & routing systems*. A "yes" means that the system provides that particular feature, while a "no" means that the system doesn't provide that particular feature. A dash means the corresponding information for that particular feature is not accessible to us. "N" means the number of nodes in the system. (1) It means the amount of routing information must be maintained on each node. (2) d is the number of dimensions in CAN's node coordinate space. If $d = (\log N)/2$, CAN could achieve the same scaling property as $O(\log N)$. (3) Lookup cost means the number of routing hops between the nodes on the overlay network. (4) It means the number of messages exchanged as part of a node join or leave operation.

4 Location & Routing System

In this section, we apply our evaluation framework to a set of location & routing systems, including Napster, Gnutella, Freenet, Chord, Pastry, Tapestry, and CAN [10].

4.1 Napster and Gnutella

Naming The file namespace doesn't require a specific structure in both systems. Users can freely name their files to be shared. Nodes' addresses are globally unique, hierarchically organized IP addresses of Internet hosts.

Structuring In Napster, a large cluster of dedicated central severs maintain an index of files currently being shared, and each node maintains a connection to one of the central servers, through which the file location queries are sent. In Gnutella, the nodes form an overlay network by forging point-to-point connections with a set of neighbors, and each node maintains the state of its neighbors.

Location & routing algorithm In Napster, each node sends a location query to one of the central servers for the requested files. The servers then cooperate to process the query and return a list of matching files and locations. Upon receiving the results, the node may choose to initiate a file exchange directly from another node over HTTP-style protocol. In Gnutella, to locate a file, a node initiates a controlled flood of the network by sending a query to all of its neighbors. Upon receiving a query, a node checks if any locally stored files match the query. If so, the peer sends a query response back towards the query originator. Whether or not a file match is found, the peer continues to flood the query until the TTL (time to live) reaches 0. Upon receiving a query response, the query originator may initiate a file download directly from the node who gives the query response.

Node addition/deletion In Napster, the central servers use "keepalive" to monitor the state of each node in the system and keep the index of shared files up-to-date. However, maintaining such an index in the central severs is very expensive. Moreover, it limits scalability and poses a single point of failure. Gnutella uses *ping* and *pong* messages to maintain the overlay network as the nodes join and leave. Gnutella is completely decentralized and has a better scalability than Napster. However, the flooding of messages makes it difficult to scale to large number of nodes.

4.2 Freenet

Naming Files are identified by a 160-bit key. Each node has a key associated with itself by hashing its IP address.

Structuring Freenet's nodes form an overlay network by forging point-to-point connections with a set of neighbors. Each node maintains a routing table containing addresses of other nodes and the keys that they are thought to hold.

Location & routing algorithm A request operates as a steepest-ascent hill-climbing search with backtracking. Upon receiving a request, a node first checks its datastore for the key specified in the request and if not found, it forwards the

request to another node, whose keys are closer to the requested key. Results for both successful and failed searches backtrack along the path the request traveled. If a node fails to locate the desired key, it returns a failure message back to its upstream node which will then try alternate downstream node that is its next best choice.

Routing fault tolerance During routing, if a node can't forward a request to its preferred downstream node because the target is down, the node having the second-nearest key will be tried, then the third-nearest, and so on. Using a steepest-ascent hill-climbing search with backtracking makes multiple routing paths available.

Node addition/deletion Freenet supports dynamic node additions and deletions.

4.3 Chord/CFS

As an essential building block of CFS, Chord is an efficient distributed lookup system based on consistent hashing.

Naming A node's identifier is produced by hashing the node's IP address. Each piece of content is also named with an identifier, produced by hashing the contents. A key's successor is the node whose identifier most closely follows the key (numerically).

Structuring Chord's nodes form an overlay network that represents an one-dimensional, circular key space. Each node has a successor and predecessor based on the total ordering of node's identifiers. In addition, each node maintains a successor list of nodes which immediately follow it in the circular key space, and a finger table with $O(\log N)$ entries. The finger table accelerates lookups, but doesn't need to be accurate. Only the successor list is required for correctness.

Location & routing algorithm Given a key k, if a node i is not a successor of this key, it searches its finger table for another node j whose identifier most immediately precedes k, and ask j for the node it knows whose identifier is closest to k. Repeat this process until k's successor is found. Routing efficiency is achieved with $O(\log N)$ hops.

Routing fault-tolerance The successor-list mechanism makes it possible to route around a single link or node failure by choosing another node in the successor list.

Network locality Chord makes no explicit effort to achieve network locality.

Node addition/deletion Chord must update its routing information as nodes join and leave the system; a join or leave requires $O(\log^2 N)$ messages. In addition, the node failure can be detected and recovered since each node maintains a successor list of nodes.

4.4 Pastry/PAST

Pastry is the location and routing scheme used by PAST. It is highly efficient, scalable, fault-resilient and self-organizing.

Naming Each node is assigned a 128-bit node identifier (nodeId), derived from a cryptographic hash of the node's public key or IP address. The nodeId indicates a node's position in a circular namespace. Each file is assigned a 160-bit fileId, derived from a cryptographic hash of the file's textual name, the owner's public key and a random salt.

Structuring Pastry's overlay network topology is a deterministic function of participating nodes' public keys or IP addresses. Each node maintains a routing table, a left set and a neighborhood set. A node's routing table is organized into $\log_{2b} N$ levels with $2b - 1$ entries each. The $2b - 1$ entries at level n each refer to a node whose nodeId matches the present node's nodeId in the first n digits but differs in $n + 1$th digit. A node's leaf set is the set of nodes with $l/2$ numerically closest larger nodeIds and the $l/2$ numerically closest smaller nodeIds. A node's neighborhood set is a set of l nodes that are closest to the present node according to the proximity metric.

Location & routing algorithm Pastry is a prefix-based routing protocol. Given a message, the node first checks if the key specified in the message falls within the range of nodeIds covered by its leaf set. If so, the message is forwarded directly to the node in the leaf set whose nodeId is closest to the key. Otherwise, by consulting the routing table, the message is forwarded to a node which shares a longer prefix with the key than the local node. If the appropriate entry in the routing table is empty or the associated node is unreachable, the message is forward to a node which shares a prefix with the key at least as long as the local node, and is numerically closer to the key than the local node's nodeId. Such a node must be in the leaf set unless the message has already arrived at the node with numerically closest nodeId. Routing efficiency is achieved with $O(\log_{2b} N)$ hops.

Routing fault-tolerance During routing, a message is always forwarded to a node which shares a longer prefix with the key, or shares the same prefix length with the local node but is numerically closer in the nodeID space than the local node. However, the choice among multiple nodes which satisfy this criterion can be made randomly, thus avoiding a malicious or failed node along the path.

Network locality Pastry achieves network locality by maintaining locality in the routing table. That is, the entries in the routing table of each node are chosen to be close to the present node, according to the proximity metric, among all live nodes with the desired nodeId prefix. Therefore, each routing step moves a message closer to the destination in the nodeId space, while traveling the least possible distance in the proximity space.

Node addition/deletion Pastry must update the routing table, leaf set and neighborhood set in the presence of node failures, node recoveries, and new node arrivals. A node join or leave requires $O(\log_{2^b} N)$ messages.

4.5 Tapestry/OceanStore

Tapestry is a self-organizing, scalable, fault-tolerant location & routing infrastructure used in OceanStore.

Naming Both objects and nodes have identifiers which are random fixed-length bit strings and are independent of their location (For example, in OceanStore, an object is identified by a GUID, produced by the cryptographic hash of the owner's public key and the object name. The GUID for a node is a cryptographic hash of its public key, and the GUID for an archival fragment is a hash of the data it holds). In addition, the root node of an object is the node whose ID matches the object's ID in the greatest number of trailing bit positions.

Structuring Tapestry functions as an overlay network on top of IP, using a distributed, fault-tolerant data structure to explicitly track the location of objects. Each node maintains a neighbor map. The neighbor map is organized into $O(\log N)$ routing levels. Each level contains entries that points to a set of nodes closest in network distance (the proximity metric) that matches the suffix of that level. Each node also maintains a backpointer list that points to nodes where it is referred to as a neighbor. The backpointer list is used in the node integration. The root node of an object provides a guaranteed or surrogated node where the location mapping for that object can be found.

Location & routing algorithm Tapestry is a suffix-based routing protocol. When a node inserts an object into the system, Tapestry publishes its location by depositing a location mapping at each hop between the new object and the object's root node. To locate an object, a node routes request messages towards a set of roots (since Pastry assigns multiple roots to each object to avoid a single point of failure). At each hop along the way, if the message encounters a node that contains the location mapping for the object, it is immediately redirected to the node holding this object. Otherwise, the message is forwarded one step closer to the root. When multiple copies of data exist in Tapestry, each node en route to the root node stores location mappings of all such data replicas to increase semantic flexibility, thus allowing the application to define selection operation (to define how objects are chosen, e.g., in terms of the proximity metric). Routing efficiency is achieved with $O(\log N)$ hops.

Routing fault tolerance Using multiple root nodes for an object achieves redundancy and simultaneously makes it difficult to target a single root node with denial of service attack. To locate an object, a node can send request messages to multiple root nodes, thus providing multiple routing paths. Moreover, maintaining two additional backup neighbors in the neighbor map provides alternate routing paths in the presence of the primary neighbor failures.

Network locality Tapestry achieves network locality in the following way: (1) Each level in the neighbor map contains entries that points to a set of nodes closest in network distance. Therefore, each routing step moves a message closer to the root node in the GUID space, while traveling the least possible distance in the proximity space; (2) when multiple copies of data exist in Tapestry, each node en route to the root node stores location mappings of all such data replicas, thus allowing routing to chooses the closest replica in network distance.

Node addition/deletion Tapestry must update the neighbor map and backpointer list as a node joins or leaves; a node join or leave requires $O(\log N)$ messages. In addition, Tapestry provides two *introspective* mechanisms to allow Tapestry to adapt to environmental changes. First, Tapestry nodes tune their neighbor pointers in the presence of changes of network distance and connectivity. Second, it is able to detect query hotspots, and offer suggestion on location where additional copies can significantly improve query response time.

	Napster	Gnutella	Freenet	Chord	Pastry	Tapestry	CAN
performance	bad	bad	moderate	good	good	good	good
scalability	bad	bad	–	good	good	good	good
reliability	bad	good	good	good	good	good	good
maintenance	best	best	good	good	good	good	good

Table 2: Comparison among different location & routing systems.

4.6 CAN

CAN is a distributed infrastructure that provides hash table-like functionality on Internet-like scales. It can be used in large-scale storage systems like OceanStore.

Naming Each node is assigned a distinct zone within a d-dimensional coordinate space. Each piece of data has a unique key K, which is deterministically mapped onto a point P in the d-dimensional coordinate space using a uniform hash function.

Structuring CAN's nodes form an overlay network that represents a d-dimentional coordinate space. The entire coordinate space is dynamically partitioned among all the nodes in the system such that every node owns its individual, distinct zone within the overall space. Each node maintains a coordinate routing table that holds the IP address and virtual coordinate zone of each of its $O(d)$ immediate neighbors in the coordinate space.

Location & routing algorithm Given a key, a node applies the deterministic hash function to map the key onto a point P in the coordinate space. The zone where the point P lies is the destination coordinate. Using its neighbor coordinate set, each node routes the message towards the destination node by simply greedy forwarding to the neighbor with coordinate closest to the destination coordinate. Routing efficiency is achieved with $(d/4)(N^{1/d})$ hops.

Routing fault tolerance Many different paths exist between two points in the CAN d-dimensional space, and each node maintains $O(d)$ neighbors, so even if one or more of a node's neighbors were to crash, a node can automatically route along the next best available path.

Network locality CAN makes no explicit effort to achieve network locality.

Node addition/deletion CAN must update the coordinate routing table as a node joins or leaves; a node join or leave requires $O(d)$ messages. In addition, node failures can be detected and recovered automatically.

4.7 Summary

In this section, we summarize the features of each location & routing system in Table 1, and compare these systems in terms of performance, scalability, reliability and maintenance (as shown in Table 2).

Performance The performance of the location & routing system depends on three factors: the efficiency of the location & routing algorithm (in terms of routing hops), the number of messages exchanged as part of a node join or leave operation, and network locality. Since the central servers in Napster are likely overloaded and pose a bottleneck of performance, Napster has a bad performance. Gnutella goes a step further and is completely decentralized, however, its broadcast based routing prevents it from achieving good performance. Freenet doesn't guarantee a definite answer to a query in a bounded number of network hops, and it trades off between anonymity and performance, so it's performance is also not satisfying. The other four systems, instead, have a good performance.

Scalability The scalability means that a system is able to remain tractable as the number of nodes increases. It mainly depends on two factors: the efficiency of location & routing algorithm (in terms of routing hops), and the amount of routing information maintained on each node. Obviously, Napster is not scalable because of its centralized structure. Flooding on every request also makes Guntella difficult to scale to large number of nodes, in spite of its completely decentralized structure. The scalability of Freenet has not been determined yet. However, the other four systems have good scalability.

Reliability Generally, reliability includes both data reliability (e.g., achieved by data replication) and network reliability. We here decouple data reliability (it will be addressed in storage management) from the location & routing system, and focus on the network reliability, which includes routing fault tolerance, and node failure detection and recovery. Except Napster, the other six systems have good reliability.

Maintenance It includes the amount of human effort required to maintain the location & routing system. Fault tolerance and self-organization ease maintenance. So systems such as Freenet, Chord, Pastry, Tapestry and CAN are able to achieves good maintenance. For Napster and Gnutella, the nodes are loosely organized, no human effort is required to do about maintenance, thus giving the best maintenance.

5 Storage Management and Caching

In this section, we intend to evaluate the storage management against a common set of criteria: load balancing, scalability, availability, persistence, and self-maintenance. Caching management is also addressed. The comparison in Table 3 summarizes the characteristics of each of the systems presented in this section.

5.1 Napster and Gnutella

Load balancing Both systems don't provide load balancing or have difficulty achieving load balancing since the nodes in both systems are loosely organized.

Scalability Napster is not scalable because of its centralized structure. Gnutella is also not scalable due to its broadcast based routing protocol.

Availability, persistence, self-maintenance, and caching Neither Napster nor Gnutella provides high availability, persistence, self-maintenance, or caching.

5.2 Freenet

Load balancing Freenet makes no explicit effort to achieve load balancing.

Scalability Although the scalability of Freenet has not been fully studied, the protection of anonymity (e.g., publisher, reader and sever anonymity) by using probabilistic routing would hurt the scalability.

Availability Freenet provides network fault-tolerance and uses caching to duplicate files. However, it doesn't explicitly maintain file replicas, and unpopular files may simply disappear from the system, resulting in unpredictable availability.

Persistence Freenet bases data lifetime on the popularity of the data. So, persistence is not guaranteed in Freenet.

Self-maintenance Freenet supports node arrivals and failures, and provides network fault tolerance. However, it doesn't support node failure recovery, so its self-maintenance is not good.

Caching Freenet caches data in a set of nodes using LRU (least recently used) algorithm. Infrequently requested files will fade away naturally after being superseded by newly inserted files.

5.3 CFS/Chord

Load balancing The uniform distribution of both node IDs and data block IDs in their respective spaces, ensures that the number of data blocks stored by each node is roughly balanced. To accommodate diverse file sizes, CFS splits files into data blocks and distributes the blocks across the nodes. To accommodate heterogeneous node capacities, CFS uses the notion of a node acting as multiple *virtual servers*. The number of virtual servers is assigned to a node in rough proportion to the node's storage and network capacity, and can vary based on current load.

Scalability CFS has good scalability due to the scalability of Chord (lookup operations use space and messages at most logarithmic in the number of nodes) and the efficiency of its load balancing.

Availability CFS places a block's replicas at a number of nodes (k) immediately after the block's successor on the Chord ring. A block's successor manages replication of that block by making sure that all k of its successors have copy of the block at all times.

Persistence CFS stores data for an agreed-upon, finite interval. Publishers that want indefinite storage periods can periodically ask CFS for an extension; otherwise, a CFS node may discard data whose guaranteed period has expired.

Self-maintenance CFS supports both fault tolerance and failure recovery, which largely depends on the fault tolerance and failure recovery of Chord in the presence of node failure/removal.

Caching CFS caches data blocks on each of the nodes the clients contacted along the lookup path.

5.4 PAST/Pastry

Load balancing The uniform distribution of both nodeIds and fileIds in their respective domains, ensures that the number of files stored by each node is roughly balanced. To accommodate differences in the storage capacity and utilization of nodes, PAST further takes two steps to achieve load balancing: (1) it uses *replica diversion* to achieve local storage space balancing among the nodes in a leaf set; (2) it uses *file diversion* to achieve global storage space balancing among different portion of the nodeId space (a new fileId is generated by using a different salt).

Scalability PAST has good scalability due to the scalability of Pastry (lookup operations use space and messages at most logarithmic in the number of nodes) and the efficiency of its load balancing.

Availability PAST maintains an invariant that k copies of inserted files are maintained on different nodes (whose nodeIds are numerically closest to the fileId) within a leaf set, even in the presence of node arrivals and node failures.

Persistence PAST provides strong persistence by maintaining k replicas for each object across k different nodes.

Self-maintenance PAST supports both fault tolerance and failure recovery, which largely depends on the fault tolerance and failure recovery of Pastry in the presence of node failure/removal.

Caching PAST's nodes use the "unused" portion of their advertised disk space to cache files. A file that is routed through a node as part of lookup or insert operation is inserted into the local disk cache. Cached files can be evicted, based on the GreedyDual-Size policy.

5.5 OceanStore

Load balancing OceanStore uses *introspective replica management* to alleviate the query load on hotspot files by dynamically adapting the number and location of floating file replicas to access requests.

Scalability OceanStore has good scalability due to the scalability of Tapestry (lookup operations use space and messages at most logarithmic in the number of nodes).

Availability OceanStore uses replica management by dynamically adjusting the number and location of floating replicas to achieve high availability.

Persistence OceanStore provides strong persistence by using archival storage.

Self-maintenance OceanStore support both fault tolerance and failure recovery, which largely depends on the fault tolerance and failure recovery of Tapestry in the presence of node failure/removal.

Caching OceanStore caches both objects and locations of objects. When a replica is placed at a node in the system, its location is cached along the way to its root node. Furthermore, OceanStore uses introspective replica management to adjust the number and location of floating file replicas in order to service access requests more efficiently.

6 Security

In this section, we evaluate the security of P2P storage systems against a common set of criteria: data integrity, data privacy, anonymity and prevention of potential attacks (see Table 3).

6.1 Data Integrity

Both Napster and Gnutella as well as OceanStore don't provide data integrity, while Freenet uses public-key cryptography to provide data integrity.

CFS uses the combination of the root block and content hash to provide data integrity. The root block is identified by a public key and signed by the corresponding private key, which allows clients to verify the integrity of the root block. The content hash allows clients to check the integrity of blocks lower in the tree led by the root block.

PAST employs *file certificates* to allow storage nodes and clients to verify the integrity and authenticity of stored content. A file certificate contains a cryptographic hash of the file's contents, the fileId (computed by the smartcard), the replication factors, the salt, and is signed by the smartcard. A file certificate is issued by a user during a file insertion operation.

6.2 Data Privacy

Systems such as Napster, Gnutella and CFS don't provide data privacy, while Freenet encrypts files using the file's descriptive string as a key or a randomly-generated encryption key. PAST allows users to employ encryption to protect the privacy of their data using a cryptosystem of their choice. OceanStore encrypts all data in the system and distributes the encryption key to those users with read permission.

6.3 Anonymity

In both Napster and Gnutella, files are exchanged through direct point-to-point connections between downloading and uploading peers, meaning that the IP addresses of servers and readers are both revealed to each other. So both systems provide no anonymity.

Systems such as CFS, PAST and OceanStore don't provide any kind of anonymity. This is probably because the anonymity guarantee often leads to design compromises that limit reliability and performance.

Freenet explicitly sets out to provide anonymity. It uses probabilistic routing to provide publisher, reader and server anonymity. There is no deterministic way to map a file onto a specific node. Furthermore, files in Freenet are stored in encrypted form, so a Freenet node can't know the contents it stores without knowing the key, thus providing document anonymity.

6.4 Preventing Attacks

Both Napster and Gnutella make no explicit effort to prevent potential attacks from malicious users and misbehaving nodes.

Freenet proposes a way to prevent a malicious user from inserting a large number of junk files by requiring the user to perform a lengthy computation before an insert is accepted, thus slowing down an attack. This method will incur a performance penalty for normal insert operations.

CFS prevents malicious injection of large quantities of data by limiting the amount of data any particular IP address can insert into the system, e.g., each CFS sever limits any one IP address to using 0.1% of its storage.

PAST uses smartcards (which holds user's private/public key pair) to ensure the integrity of nodeId and fileId assignment, thus preventing an attacker from controlling adjacent nodes in the nodeId space, or directing file insertions to a specific portion of the fileId space. Smartcards also maintain storage quotas to limit the amount of data any user can insert into the system. Store receipts prevent a malicious node from causing the system to create fewer than k diverse replicas of a file without the client noticing it. Furthermore, The Pastry routing scheme can be randomized, thus preventing a malicious node along the path from repeatedly intercepting a request and causing a denial of service. It also prevents attackers from forging routing table entries (i.e. nodeId to IP address mappings) by requiring associated nodes to sign them.

Systems such as CFS, PAST, and OceanStore all implement data replication, thus avoiding single point of responsibility and preventing a malicious node from denying the existence of data for which it is responsible for.

7 Future Work

Topology-aware location & routing system The scalability of a P2P storage system is ultimately determined by how efficiently its location & routing system uses the underlying network. Systems such as CFS, PAST, and OceanStore have an overlay network that maps down to a physical IP-level network. The most commonly used measure of the efficiency of the location & routing algorithm is the resulting overlay network hops. Most of the algorithms achieve $O(\log N)$ hops.

	Napster	Gnutella	Freenet	CFS	PAST	OceanStore
load balancing	no	no	no	yes	yes	yes
scalability	bad	bad	–	good	good	good
availability	low	low	moderate	high	high	high
persistence	weak	weak	weak	depends	strong	strong
self-maintenance	bad	bad	bad	good	good	good
caching	no	no	yes	yes	yes	yes
data integrity	no	no	yes	yes	yes	no
data privacy	no	no	yes	no	depends	yes
anonymity	no	no	yes	no	no	no

Table 3: Comparison among P2P storage systems in storage management and caching, and security.

However, the true efficiency measure should be the end-to-end latency of the path rather than the overlay network hops. Because the nodes in P2P systems are geographically distributed, some of these hops could involve transcontinental links. So a small number of hops don't necessarily suggest low latency. Both Pastry/PAST and Tapestry/OceanStore have addressed the network locality in terms of the proximity metric during routing and neighbor selection, resulting in better performance. In addition, many physical networks are *power-law networks*, where most nodes have few links while a tiny number of nodes, called hub, have a large number of links. Current P2P systems, however, don't take the network heterogeneity (connectivity, bandwidth) into account. This might lead to inefficient routing. So designing a topology-aware location & routing system is an important research topic.

Evaluation of P2P storage systems The framework we proposed in this paper can serve as a criterion for evaluation of P2P storage systems. However, in order to compare two different P2P systems, a new set of benchmarks for evaluation is needed. Individual P2P storage systems such as CFS and PAST have provided us with data about storage management and caching through simulations and experiments. But the topology and scale of the network models they used are different, rendering such data not very useful for evaluation and comparison. So the benchmarks for P2P systems should make the topology and scale of the network standard. Furthermore, P2P storage systems may differ in some aspects, such as the level of security they provide. Since different levels of security guarantee (data integrity, data privacy, anonymity, etc.) incur different performance penalties. So, when we use a benchmark to compare the performance of two different systems, the benchmark should specify other requirements they must meet, such as the same level of security. All these concerns make benchmarking of P2P storage systems very necessary.

Security Security is of growing importance in an untrusted, dynamically changing environment on P2P storage systems. Some of P2P systems use cryptographic hash and digital signature to protect data integrity. However, digital signature is very expensive in terms of processing, imposing much performance overhead. Encryption is used to protect data confidentiality, but at the same time it poses another problem – key distribution for file sharing. Anonymity is a desirable feature for P2P systems, but it comes at a cost, compromising performance. In addition, recent P2P research has been focusing on the efficiency of location & routing systems but hasn't considered too much to handle all kinds of potential attacks from malicious users and misbehaving nodes. All these concerns render security as an open research question.

8 Conclusions

This paper has presented an evaluation framework of P2P storage systems. We have reviewed the following P2P storage systems including Napster, Gnutella, Freenet, CFS, PAST, and OceanStore, and mapped them into this framework. We further give a comparative summary for all these systems in terms of the location & routing system, storage management and caching, and security. We hope that our evaluation framework will provide aids to other researchers, either by improving the existing or in developing new ones. Finally, we have suggested some future work in P2P systems.

References

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

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

[3] 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*, (Berkeley, CA), pp. 331–320, July 2000.

[4] F. DABEK, M. KAASHOEK, D. KARGER, R. MORRIS, and I. STOICA, "Wide-area cooperative storage with cfs," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, (Banff, Canada), pp. 202–215, Oct. 2001.

[5] A. Rowstron and P. Druschel, "Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, (Banff, Canada), pp. 188–201, Oct. 2001.

[6] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "Oceanstore: An architecture for global-scale persistent storage," in *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, (Cambridge, MA), pp. 190–201, Nov. 2000.

[7] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of ACM SIGCOMM*, (San Diego, CA), pp. 149–160, Aug. 2001.

[8] A. I. T. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Proceedings of the 18th IFIP/ACM International Conference on Distributed System Platforms (Middleware 2001)*, (Heidelberg, Germany), Nov. 2001.

[9] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph, "Tapestry: An infrastructure for fault-tolerance wide-area location and routing," Tech. Rep. UCB/CSD-01-1141, Computer Science Division, University of California, Berkeley, Apr. 2001.

[10] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and Shenker, "A scalable content-addressable network," in *Proceedings of ACM SIGCOMM*, (San Diego, CA), pp. 161–172, Aug. 2001.