III

Peer-to-Peer Networks

Peer-to-peer (P2P) computing refers to technology that enables two or more peers to collaborate spontaneously in a network of equals (peers) by using appropriate information and communication systems without the necessity for central coordination. The P2P network is dynamic, where peers come and go (i.e., leave and join the group) for sharing files and data through direct exchange.

The most frequently discussed applications include popular file-sharing systems, such as early Napster. In addition to file-sharing collaborative P2P service, grid computing and instant messaging are key applications of P2P. P2P systems offer a way to make use of the tremendous computation and storage resources on computers across the Internet. Unlike sensor networks and ad hoc wireless networks, P2P networks are overlay networks operated on infrastructured (wired) networks, such as the Internet. However, P2P networks are also highly dynamic, where users join and leave the network frequently. Therefore, the topology of the overlay network is dynamic.

Most current research in this field focuses on the location management which is also called the *lookup problem*. Specifically, how can we find any given data item in a large P2P network in a scalable manner. In serverless approaches, flooding-based search mechanisms are used; these include DFS with depth limit D (as in Freenet) or BFS with depth limit D (as in Gnutella), where D is the system-wide maximum TTL of a message in hops. There are several efficient alternatives; these include iterative deepening to slowly increase the flooding rings, random walks instead of blind flooding to reduce the flooding space, and dominating-set-based searching to reduce the searching scope. In server-based approaches, Napster uses a centralized approach by maintaining a central database. KaZaA uses a hierarchical approach based on the notion of supernode.

Data in P2P networks are sometimes structured to facilitate an efficient searching process through the use of *distributed hash table* (DHT). Each node acts as a server for a subset of data items. The operation *lookup(key)* is supported, which returns the node ID storing the data item with that key. The values of the node could be data items or pointers to where the data items are stored. Each data item is associated with a key through a hashing function. Nodes have identifiers, taken from the same space as the keys. Each node maintains a routing table consisting of a small subset of nodes in the system. In this way, an overlay network is constructed that captures logical connections between nodes. Usually, the logical network is a regular network such as a ring, tree, mesh, or hypercube. When a node receives a query for a key for which it is not responsible, the node routes the query to the neighbor that

makes the most "progress (defined in terms of "distance" between source and destination) towards resolving the query. A promising approach is limited server-based approaches, where location information is limited to a limited region of nodes.

Other issues related to P2P systems include network control, security, interoperability, metadata, and cost sharing. Some open problems include operation costs, fault tolerance and concurrent changes, proximity routing, malicious nodes, and indexing and keyword searching.

Among 13 chapters in this group, one chapter gives a general overview of P2P networks. Two chapters deal with searching techniques, including one on semantics search. One chapter provides an overview of structured P2P networks. The next three chapters are devoted to three specific aspects of structured P2P networks: distributed data structure, state management, and topology construction. One chapter presents some theoretical foundations on tradeoffs between routing table size and network diameter. One chapter deals with overlay optimization. Reliability and efficient issues are covered in two chapters. The discussion on security issues is given in one chapter. The group ends with the application of the peer-to-peer concept in ad hoc wireless networks.

36

Peer-to-Peer: A Technique Perspective

Weimin Zheng, Xuezheng Liu, Shuming Shi, Jinfeng Hu, and Haitao Dong

36.1 Introduction

Peer-to-peer (P2P) systems have drawn much attention from end Internet users and the research community in the past few years. From Napster,[1] the earliest P2P system, appearing in 1999, to popular systems such as Gnutella,[2] Freenet,[3] KaZaA,[4] and BitTorrent,[5] more and more P2P file sharing systems come to fame. In research community, P2P has become one the most active research fields. Many universities and research institutes have research groups focusing on P2P techniques. And P2P has become one of the hottest topics at many conferences and workshops on distributed systems or networking.

Different from traditional client/server architectures, each node in a peer-to-peer system acts as both a producer (to provide data to other nodes) and a consumer (to retrieve data from other nodes). Key features of P2P systems include large-scale self-organization, self-scaling, decentralization, fault tolerance, and heterogeneity. In a P2P system, an increase in the number of peers also adds capacity of the system. There is a large heterogeneity in the properties (including capacity, availability, bandwidth, etc.) of peers in most P2P systems. A typical P2P system often has no central servers to which users connect. Instead, the whole system is self-organized by the interaction of peers. The above features of P2P systems make them suitable for file sharing. In addition to file sharing, there are also some proposals that use P2P to build large-scale, fault-tolerant storage systems,[6–9] do content distribution,[10] or even as a replacement of current networks.[11]

The earliest P2P system was Napster, which maintains a centralized directory to facilitate file sharing. Files shared by peers are indexed in a centralized directory. To search a file, peers first refer to the centralized directory and results are returned after local lookup; and then result files are downloaded directly from peers. Relying on this kind of centralized-lookup and decentralized-download mode, Napster became one of the most popular applications in just a few months. However, Napster was forced to shut down because of legal issues. Then there came decentralized architectures where the search and download processes are both distributed among peers. Existing decentralized P2P networks can be partitioned into two groups: unstructured and structured. Unstructured P2P systems (e.g., Gnutella and Freenet) have few constraints on overlay topology and the mapping of data to peers. Whereas, for structured P2P systems (e.g., CAN,[12] Pastry,[13] Chord,[14] Tapestry,[15] etc.), the topology of overlay and the mapping of data items to nodes are strictly controlled.

Because of the heterogeneous and dynamic properties of peers, it is a challenge for peers to self-organize themselves to build a large-scale P2P system in a decentralized way. Many researchers and system designers have been involved in solving key problems of P2P systems in the past several years.

This chapter analyzes the design issues and applications of P2P systems from a technique perspective. It focuses key design issues of P2P systems and studying how these techniques are used to support practical applications.

36.2 Design Issues

This section discusses the fundamental issues of P2P system design. It is very difficult to build a large-scale distributed system and combine enormous weak, dynamic participants to provision powerful and persistent services, especially in the Internet environment which is prone to failures. The system should scale to large numbers of peers, tolerate unpredictable failures, fully utilize peers' capacities and heterogeneities, guarantee strong availability of data or services, and efficiently provide diverse functionalities.

Therefore, there are many challenge problems to address. First, peers should be perfectly organized so as to facilitate communication and cooperation. Thus, we need to construct an efficient routing infrastructure in P2P networks, an infrastructure that is scalable and fault tolerant, and also exploits network proximity (Section 36.2.1). Second, on top of the underlying routing, we need to deploy services in P2P networks. Thus, we should answer how the services and data are placed and supplied, so as to achieve data availability and full utilization of peers' capacities (Section 36.2.2). In addition, the difficulty of looking for the most suitable service from numerous provided services leads us to build efficient search mechanisms in P2P applications (Section 36.2.3). Finally, for the special cases of group communication (e.g., network meeting, media streaming, and bulk data multiple delivery), only unicast communication and one-to-one data

transmission remain far from sufficient. Thus, we want to implement application-level multicast on top of overlay and P2P networks (Section 36.2.4).

36.3 Routing

Scalability is an extraordinarily important issue for the P2P system. Unfortunately, the original designs for P2P systems have significant scaling problems. For example, Napster introduces a centralized directory service, which becomes a bottleneck for the millions of users' access; Gnutella employs a flooding-based search mechanism that is not suitable for large systems.

To solve the scaling problem, several approaches have been simultaneously but independently proposed, all of which support a distributed hash table (DHT) functionality. Among them are Tapestry,[15] Pastry.[13] Chord,[14] and Content-Addressable Networks (CAN).[12] In these systems, which we call DHTs, files are associated with a key, which is produced, for instance, by hashing the filename, and each node in the system is responsible for storing a certain range of keys. There is one basic operation in these DHT systems: lookup(key), which returns the identity (e.g., the IP address) of the node storing the object responsible for that key. By allowing nodes to put and get files based on their key with such operation, DHTs support the hash-table-like interface. This DHT functionality has proved a useful substrate for large distributed systems, which is promising to become an integral part of the future P2P systems.

The core of these DHT systems is the routing algorithm. The DHT nodes form an overlay network with each node having several other nodes as neighbors. When a lookup(key) operation is issued, the lookup is routed through the overlay network to the node responsible for that key. Then, the scalability of these DHT algorithms depends on the efficiency of their routing algorithms. Each of the proposed DHT systems listed above — Tapestry, Pastry, Chord, and CAN — employs a different routing algorithm. Although there are many details that are different between their routing algorithms, they share the same property that every overlay node maintains $O(\log n)$ neighbors and routes within $O(\log n)$ hops (n is the system scale).

Researchers have made inroads in the following issues related to routing: state–efficiency trade-off, resilience to failures, routing hotspots, geography, and heterogeneity, and have forwarded various algorithms to improve the performance of the initial DHT systems.

This section is organized as follows. First we review the history of research for routing algorithms in P2P system; then we introduce the representative DHT systems; and finally we discuss several issues related to DHT routing algorithms.

36.3.1 $O(\log N)$ DHT Overlay

This section reviews some representative routing algorithms that were introduced in the early period. All of them take, as input, a key and, in response, route a message to the node responsible for that key. The keys are strings of digits of some length. Nodes have identifiers, taken from the same space as the keys (i.e., same number of digits). Each node maintains a routing table consisting of a small subset of nodes in the system. When a node receives a query for a key for which it is not responsible, the node routes the query to the neighbor node that makes the most "progress" toward resolving the query. The notion of progress differs from algorithm to algorithm but in general is defined in terms of some distance between the identifier of the current node and the identifier of the queried key.

36.3.1.1 Plaxton et al.

Plaxton et al.[16] developed perhaps the first routing algorithm that could be scalably used by DHTs. While not intended for use in P2P systems, because it assumes a relatively static node population, it does provide very efficient routing of lookups. The routing algorithm works by "correcting" a single digit at a time: if node number 47532 received a lookup query with key 47190, which matches the first two digits, then the routing algorithm forwards the query to a node that matches the first three digits (e.g., node 47603). To do this, a node needs to have, as neighbors, nodes that match each prefix of its own identifier but differ in the next digit. For a system of $n$nodes, each node has on the order of $O(\log n)$ neighbors.

Because one digit is corrected each time the query is forwarded, the routing path is at most $O(\log n)$ overlay (or application-level) hops. This algorithm has the additional property that if the $O(n^2)$ node-node latencies (or "distances" according to some metric) are known, the routing tables can be chosen to minimize the expected path latency and, moreover, the latency of the overlay path between two nodes is within a constant factor of the latency of the direct underlying network path between them.

36.3.1.2 Tapestry

Tapestry[15] uses a variant of the Plaxton et al.algorithm. The modifications are to ensure that the design, originally intended for static environments, can adapt to a dynamic node population. The modifications are too involved to describe in this short review. However, the algorithm maintains the properties of having $O(\log n)$ neighbors and routing with path lengths of $O(\log n)$ hops.

36.3.1.3 Pastry

Each node in the Pastry[13] peer-to-peer overlay network is assigned a 128-bit node identifier (nodeID). The nodeID is used to indicate a node's position in a circular nodeID space, which ranges from 0 to 2128. The nodeID is assigned randomly when a node joins the system. It is assumed that nodeIDs are generated such that the resulting set of nodeIDs is uniformly distributed in the 128-bit nodeID space. As a result of this random assignment of nodeIDs, with high probability, nodes with adjacent nodeIDs are diverse in geography, ownership, jurisdiction, network attachment, etc.

Assuming a network consisting of n nodes, Pastry can route to the numerically closest node to a given key in less than $O(\log n)$ steps under normal operation and this is a configuration parameter with typical value. Despite concurrent node failures, eventual delivery is guaranteed unless L/2 nodes with adjacent nodeIDs fail simultaneously (L is a configuration parameter with a typical value of 16 or 32). For the purpose of routing, nodeIDs and keys are thought of as a sequence of digits with base 2b. Pastry routes messages to the node whose nodeID is numerically closest to the given key. To support this routing procedure, each node maintains routing state with length of $O(\log n)$. In each routing step, a node normally forwards the message to a node whose nodeID shares with the key a prefix that is at least one digit (or b bits) longer than the prefix that the key shares with the present node's ID. If no such node is known, the message is forwarded to a node whose nodeID shares a prefix with the key as long as the current node, but is numerically closer to the key than the present node's ID. Figure 36.1 is an example of Pastry node's routing table.

Node ID 10233102

Leaf set

10233021	10233033	10233120	10233122

Suffix set

-0-2212102	**1**	-2-2303203	-3-1203203
0	1-1-301233	1-2-230203	1-2-021022
10-0-31203	10-1-32102	2	10-3-23302
102-0-0230	102-1-1302	102-2-2302	3
1023-0-322	1023-1-000	1023-2-121	3
10233-0-01	1	10233-2-32	
0		102331-2-0	
		2	

FIGURE 36.1 State of a hypothetical Pastry node with nodeID 10233102, b = 2, and l = 4. All numbers are in base 4. The top row of the routing table is row zero. The shaded cell in each row of the routing table shows the corresponding digit of the present node's nodeID. The nodeIDs in each entry have been split to show the *common prefix with 10233102–next digit–rest of nodeID*. The associated IP addresses are not shown. (*Source*: From Reference 13.)

36.3.1.4 Chord

Chord[14] also uses a one-dimensional circular keyspace. The node responsible for the key is the node whose identifier most closely follows the key (numerically); that node is called the key's successor. Chord maintains two sets of neighbors. Each node has a successor list of k nodes that immediately follow it in the keyspace. Routing correctness is achieved with these lists. Routing efficiency is achieved with the finger list of $O(\log n)$ nodes spaced exponentially around the keyspace. Routing consists of forwarding to the node closest to, but not past, the key; path lengths are $O(\log n)$ hops.

36.3.1.5 CAN

CAN[12] chooses its keys from a d-dimensional toroidal space. Each node is associated with a hypercubal region of this keyspace, and its neighbors are the nodes that "own" the contiguous hypercubes. Routing consists of forwarding to a neighbor that is closer to the key. CAN has a different performance profile than the other algorithms; nodes have d neighbors and path lengths are $O(dn^{1/d})$ hops. Note, however, that when $d = O(\log n)$, CAN has neighbors and $O(\log n)$ path lengths like the other algorithms.

36.3.2 Improvements

36.3.2.1 State–Efficiency Trade-off

The most obvious measure of the efficiency of these routing algorithms is the resulting path length. Most of the algorithms have path lengths of $O(\log n)$ hops, while CAN has longer paths of $O(dn^{1/d})$. The most obvious measure of the overhead associated with keeping routing tables is the number of neighbors. This is not just a measure of the state required to do routing, but is also a measure of how much the state needs to be adjusted when nodes join or leave. Given the prevalence of inexpensive memory and the highly transient user populations in P2P systems, this second issue is likely much more important than the first. Most of the algorithms require $O(\log n)$ neighbors, while CAN requires only $O(d)$ neighbors.

Xu et al.[17] studied this fundamental trade-off, which is shown in Figure 36.2. We can see that, in a network consisting of n nodes, when n neighbors are maintained at each node, the search cost is $O(1)$, but to maintain so large a routing table will introduce heavy maintenance costs because of the frequent joins and leaves of the P2P nodes. When each node only maintains one neighbor, the search cost is $O(n)$, which incurs intolerable network delay. This plots two endpoints on the trade-off curve shown in Figure 36.2. They also point out that there exist algorithms that achieve better trade-offs than existing DHT schemes, but these algorithms cause intolerable levels of congestion on certain network nodes.

Gupta et al.[18] argue that it is reasonable to maintain complete information at each node. They propose a well-defined hierarchy system to ensure a notification of membership change events (i.e., joins and

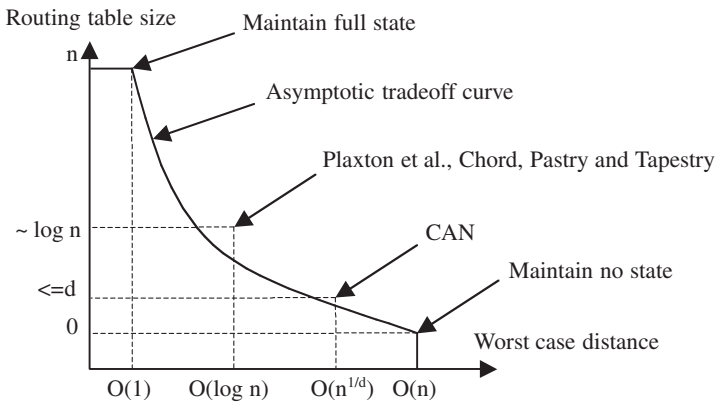

FIGURE 36.2 Asymptotic trade-off curve between routing table size and network diameter.

leaves) can reach every node in the system within a specified amount of time (depending on the fraction of failed queries) and with reasonable bandwidth consumption. There is a distinguishing character in their approach: a node in the system need not probe all the items in its routing table. It only watches a small fraction of nodes; and when membership change event occurs, the node responsible for this event will initial an event multicast in the whole system.

36.3.2.2 Proximity

The efficiency measure used above was the number of application-level hops taken on the path. However, the true efficiency measure is the end-to-end latency of the path. Because the nodes could be geographically dispersed, some of these application-level hops could involve transcontinental links, and others merely trips across a LAN; routing algorithms that ignore the latencies of individual hops are likely to result in high-latency paths. While the original "vanilla" versions of some of these routing algorithms did not take these hop latencies into account, almost all of the "full" versions of the algorithms make some attempt to deal with the geographic proximity of nodes. There are (at least) three ways of coping with geography.

36.3.2.2.1 *Proximity Routing*

Proximity routing is when the routing choice is based not just on which neighboring node makes the "most" progress towards the key, but also on which neighboring node is "closest" in the sense of latency. Various algorithms implement proximity routing differently but all adopt the same basic approach of weighing progress in identifier space against cost in latency (or geography); usually, when choosing the next hop, the node selects, among the possible next hops, the one that is closest in the physical network or one that represents a good compromise between progress in the ID space and proximity. Simulations have shown that this is a very effective tool in reducing the average path latency. But the construction of overlay does not consider physical networks, and its performance is largely dependent on the number of alternative next hops.

36.3.2.2.2 *Proximity Neighbor Selection*

This is a variant of the idea above, but now the proximity criterion is applied when choosing neighbors, not just when choosing the next hop. During the construction of the overlay, nodes choose routing table entries to refer to the topologically nearest among all nodes with nodeID in the desired portion of the ID space. So, its performance depends on the degree of freedom an overlay protocol has in choosing routing table entries without affecting the expected number of routing hops. As mentioned, if the node-pair distances (as measured by latency) are known, the Plaxton/Tapestry algorithm can choose the neighbors so as to minimize the expected overlay path latency. This is an extremely important property that is (so far) the exclusive domain of the Plaxton/Tapestry algorithms. Its mechanism depends on satisfying triangle inequality, which may not hold for Internet.

36.3.2.2.3 *Geographic Layout*

In most of the algorithms, the node identifiers are chosen randomly (e.g.,hash functions of the IP address, etc.) and the neighbor relations are established based solely on these node identifiers. One could instead attempt to choose node identifiers in a geographically informed manner. An initial attempt to do so in the context of CAN was reported by Ratnasamy et al.[12] This approach was quite successful in reducing the latency of paths. There was little in the layout method specific to CAN but the high dimensionality of the keyspace may have played an important role. Recent work[19] suggests that latencies in the Internet can be reasonably modeled by a d-dimension geometric space with $d \geq 2$ This raises the question of whether systems that use a one-dimensional key set can adequately mimic the geographic layout of the nodes. However, this may not matter because the geographic layout may not offer significant advantages over the two proximity methods.

Moreover, these geographically informed layout methods may interfere with the robustness, hotspot, and other properties mentioned in previous sections.

36.3.2.3 Heterogeneity

All the algorithms start by assuming that all nodes have the same capacity to process messages and then, only later, add on techniques for coping with heterogeneity. However, the heterogeneity observed in current P2P populations[20] is quite extreme, with differences of several orders of magnitude in bandwidth. One can ask whether the routing algorithms, rather than merely *coping* with heterogeneity, should instead use it to their *advantage*. At the extreme, a star topology with all queries passing through a single hub node and then routed to their destination would be extremely efficient, but would require a very highly capable hub node (and would have a single point of failure). But perhaps one could use the very highly capable nodes as mini-hubs to improve routing.Chawathe et al.[21] argue that heterogeneity can be used to make Gnutella-like systems more scalable.

It may be that no sophisticated modifications are needed to leverage heterogeneity. Perhaps the simplest technique to cope with heterogeneity, and one that has already been mentioned in the literature, is to *clone* highly capable nodes so that they could serve as multiple nodes; that is, a node that is ten times more powerful than other nodes could function as ten virtual nodes. When combined with proximity routing and neighbor selection, cloning would allow nodes to route to themselves and thereby "jump" in keyspace without any forwarding hops.

36.4 Data Placement

The DHT (distributed hash table)[12–15] is a basic manner of data placement in peer-to-peer systems. In such structured systems, a unique identifier is associated with each data item as well as each node in the system. The identifier space is partitioned among the nodes and each node is responsible for storing all the items that are mapped to identifiers within its portion. Thus, the system provides two basic functions: (1) put (id, item), which stores an item with its identifier ID, and (2) get(id), which retrieves the item. How to find the appropriate node for a given ID is by the routing algorithms mentioned in Section 36.2.1.

Many DHT-based systems[6–8] assume that item IDs and node IDs are all randomly chosen by a consistent hashing function.[22] This offers a significant advantage for the system; that is, items are evenly distributed among all the nodes.

To improve the availability of the items, replication is desired. A simple and useful replication method is putting one item on k nodes whose node IDs are closest to the item's ID, where k is a constant value. For example, PAST[6] deploys Pastry as its routing infrastructure and a node replicates its data on the k closest nodes in its leaf set. Because node IDs are chosen randomly, these k nodes are scattered around the world with high probability, thus reducing the probability of simultaneous replica failures.

Other replication strategies on DHT include simultaneously using k different hashing functions for data items and simultaneously running k different routing infrastructures.[12]

However, all these methods cannot escape from a significant issue: unbalanced load on DHT nodes. This can come from the following reasons:

- There is a $\square(\log N)$ imbalance factor in the number of items stored at a node because it is impossible to achieve perfect randomicity.
- Items are not of identical size.
- Nodes are heterogeneous. This is a serious and unavoidable reality in the peer-to-peer world.[20] Some applications associate semantics with item IDs, which makes distribution of item IDs skewed. For example, a peer-to-peer database might stores tuples using their primary keys as tuple IDs.

In PAST,[6] items are first intended to be stored on k contiguous nodes in Pastry's leaf set. But once a node in the leaf set is overloaded, items on it must be moved to the other nodes in the leaf set, leaving corresponding pointers. Once all the nodes in the leaf set are overloaded, a node out of one leaf set must then undertake the responsibility of item storing. How to maintain the pointers, as well as the pointers to the pointers, significantly increases the complexity of the system. Therefore, other systems tend toward another choice, performing load balancing in a direct way:

An overloaded node claims its status and lets a light node take over part of its data.[23,24] This work first involves a concept of *virtual servers*. A virtual server works as a peer in the P2P infrastructure, while a physical node can simultaneously host many virtual servers. When a node is overloaded, it attempts to move one or many virtual servers to other nodes.

Then the key problem turns to how to find a light partner, in the peer-to-peer environment, for a heavy node to transfer its load. Methods can be classified into three types of schemes: (1) one-to-one, (2) one-to-many, and (3) many-to-many, from the simplest to the most complicated.

36.4.1 One-to-One

Each light node periodically picks a random ID and then routes to the node that is responsible for that ID. If that node is a heavy node, then a transfer can take place between the two nodes.

36.4.2 One-to-Many

This scheme is accomplished by maintaining *directories* that store load information about a set of light nodes in the system. We use the same DHT system to store these directories. Assume that there are d directories in the system. A light node l is hashed into a directory using a well-known hash function h' that takes values in the interval $[0, d)$. A directory i is stored at the node that is responsible for the identifier $h(i)$, where h is another well-known hash function. A light node l will periodically advertise its target load and current load to the node $i = h(h'(l))$,which is responsible for directory $h'(l)$. In turn, the heavy nodes will periodically sample the existing directories. A heavy node n picks a random number $k \in [0, d)$and sends the information about its target load and the loads of all its virtual servers to node $j = h(h'(k))$. Upon receiving such a message, node j looks at the light nodes in its directory to find the best virtual server that can be transferred from n to a light node in its directory. This process repeats until all the heavy nodes become light.

36.4.3 Many-to-Many

The many-to-many scheme is similar to the one-to-many scheme. The difference is that all the heavy nodes, as well as the light ones, advertise their target load and current load to the directories, and the nodes in charge of the directory determine how to transfer virtual servers, by a heuristic algorithm that is more complex than in the one-to-many scheme. Details are provided by Rao et al.[23]

Godfrey et al.[24] have examined the many-to-many scheme in a dynamic P2P environment and the results show that a such mechanism achieves load balancing for system utilization as high as 90 percent while moving only about 8 percent of the load that arrives into the system. Moreover, in a dynamic system, less than 60 percent of the load is moved due to node arrivals and departures. Compared to a similar centralized algorithm, such a distributed algorithm is only negligibly worse.

Although these balancing algorithms make the data placement fit for the heterogeneous peer-to-peer systems, another serious problem remains unresolved, that is, *the bandwidth cost*. Peer-to-peer systems maintain an invariant of k replicas, even in the dynamic environment. So when a node leaves, all its data must be reproduced on some other nodes, which is equivalent to transferring these data. Unfortunately, in peer-to-peer storage, nodes store a large amount of data and enter and leave the system frequently. This makes data always in transfer, which needs a very high bandwidth that is out of practice in today's Internet world. Blake and Rodrigues[25] argue that such a scenario will get worse and worse, not better and better, over time.

So, DHT-based storage can be eployed only in a relatively stable system, such as a data center. However, if the nodes are stable, DHT (or say, structured routing overlay) is not a necessity because full-connection topology is then practical.

A compromise method is not directly storing items by DHT, but instead letting users determine where and how to replicate their data by themselves, and only employing DHT to store the replica lists.[26] A replica list is much smaller than physical items, so moving it when nodes change will not cost too much bandwidth.

36.5 Data Lookup and Search

As more and more data is stored in P2P systems, upper applications demand that the infrastructure provides search capacity. There are two kinds of P2P search functionalities: (1) data lookup and (2) keyword searching. Data lookup takes the object ID as an input to get the corresponding data item or its hosting peer(s). Keyword searching means feeding one or some keywords into the system and getting some data items that contain the keywords. Data lookup and keyword searching are nearly the same for unstructured P2P networks because they process queries by adopting flooding or random walk[27] approaches. For structured P2P networks, the two search functionalities are quite different. As nearly all structured P2P systems presented by now actually implement distributed hash tables (DHTs), the data lookup functionality can be naturally supported. However, keyword searching is not supported directly by DHTs. In this section we review some data lookup and search techniques proposed in recent years.

36.5.1 Search in Unstructured P2P Networks

36.5.1.1 Basic Search Techniques

Flooding and random walk are two basic searching mechanisms in unstructured networks. Gnutella[2] is the most famous unstructured P2P system and uses flooding to deal with queries. To locate a file using flooding, a node sends a query to all its neighbors, which in turn propagate the query to their neighbors, and so forth until the query reaches all the peers within a certain radius of the query initiator. With random walk, a query is forwarded to a randomly chosen neighbor at each step until sufficient query results are found. Random walk is recommended by Lv et al.[27] as a replacement strategy for flooding.

36.5.1.2 Search Optimizations

Peer flooding and random walk have proved inefficient in answering search requests. To make the search process more efficient and scalable, some optimization mechanisms must be adopted. Search in unstructured P2P networks can be boosted in three directions. The first one is content aggregation, that is, peers summarizing the content from other peers. With content aggregation, peers know more information about the system, and therefore there is hope for a boost in search efficiency. The second direction is content clustering. With content clustering, all contents in the system are clustered (maybe implicitly) by semantic or user interest, and clustering information is used by peers to reduce the number of nodes that must be accessed in searching. As the third direction, peer heterogeneity can be exploited.

36.5.1.2.1 *Directed Flooding and Biased Random Walk*

Directed flooding means that each peer sends query messages to a subset of its neighbors that hopefully have more high-quality search results than other neighbors. Biased random walk means that in random walk, rather than forwarding incoming queries to randomly chosen neighbors, each peer selects a neighbor that has the highest probability of having query results. Compared with pure flooding and random walk, directed flooding and biased random walk not only reduce the number of nodes bothered to process a query, but also improve the quality of results. To intelligently select one or some neighbors, a peer must maintain statistics on its neighbors. Statistical information of a node can be collected based on its basic information (degree, capacity, availability, latency, bandwidth, etc.) and the contents of it.Adamic et al.[28] suggest that queries are forwarded to high-degree nodes. A number of neighbor-selection heuristics are listed Yang and Garcia-Molina,[29] including the number of results for previous queries, the largest number of messages forwarded, etc. Routing indices[30] is another way to collect node statistics and enable directed flooding and biased random walk. Different from other statistics collecting mechanisms, routing indices collect statistics of nodes in a "direction" rather than from a specified neighbor.

36.5.1.2.2 *r-Hop Replication*

In the r-hop replication technique, a node maintains an index over the data of all nodes within r hops of itself. When a node receives a query message, it can answer the query on behalf of every node within r hops

of itself. In this way, search overhead can be dramatically reduced. r-hop replication (especially one-hop replication) is adopted as a building block in many search boosting proposals.[21,29]

36.5.1.2.3 *Super-Nodes*

A super-peer network[4,31] is a P2P network in which all super-nodes are connected as a pure P2P network and each super-peer is connected to a set of ordinary peers. A super-node, which typically has high capacity, is a node that operates as a centralized server to a set of ordinary peers. Each ordinary peer connects to one or several super-nodes. For each query initiated from an ordinary peer, the ordinary peer submits the query to one of its super-nodes and receives query results from it. Every super-node indexes the contents of all its attached ordinary peers and then answers queries on behalf of them. In a super-peer network, queries are only processed by super-nodes, and both flooding and random walk mechanism can be adopted. Because queries are processed only by super-nodes that have relatively higher capacities than ordinary peers, searches in super-peer networks are much more efficient than in ordinary unstructured P2P networks. Yang and Garcia-Molina[31] studied the behavior of super-peer networks and presented an understanding of their fundamental characteristics and performance trade-offs.

36.5.1.2.4 *Guide Rules and Interest-Based Shortcuts*

These kinds of optimizations are in the "content clustering" direction, assuming that contents and peers in the system have semantic locality. A guide rule is a set of peers that satisfies some predicate. A group of peers belonging to a certain guide rule should contain contents that are semantically similar. Cohen et al.[32] proposed to build associative overlays based on guide rules and adopt two algorithms, RAPIER and GAS, for improving search efficiency. Sripanidkulchai et al.[33] observed the phenomenon of interest-based locality; that is, if a peer has a data item that one is interested in, then it is likely that it will have other data items that one is also interested in. All peers in the system are clustered by creating interest-based shortcuts between peers sharing similar interests. And search efficiency is improved by using these shortcuts.

36.5.1.3 Case Study: Gia

Gia[21] is a decentralized and unstructured P2P file-sharing system that combines several existing search optimization techniques in its design. The four boosting schemes used in Gia include (1) a dynamic topology adaptation protocol, (2) an active flow control scheme, (3) a one-hop replication mechanism, and (4) a search protocol based on biased random walks.

The dynamic topology adaptation protocol ensures that high-capacity peers are indeed the ones with high degrees, and that low-capacity peers are within short reach of high-capacity ones. Each peer maintains a level of satisfaction value, determined by its capacity and neighbors. Each node tends to gather more neighbors to improve its satisfaction level until it is up to 1. Each node also tends to use high-capacity nodes to replace low-capacity neighbors without dropping already poor-connected neighbors. Dynamic topology adaptation is a key protocol for Gia to explicitly make use of node heterogeneity to speed up search efficiency.

The active flow control scheme is used for avoiding overloaded hot spots. It explicitly acknowledges the existence of heterogeneity and adapts to it by assigning flow-control tokens to nodes based on available capacity. Each peer periodically assigns flow-control tokens to its neighbors. A node can send a query to a neighbor only if it has received a token from that neighbor, thus avoiding overloaded neighbors. To provide an incentive for high-capacity nodes to advertise their true capacity, Gia peers assign tokens in proportion to the neighbor's capacities, rather than distributing them evenly between all neighbors.

To improve search efficiency, Gia uses a one-hop replication mechanism, that is, each peer maintains an index of the contents of all its neighbors. Indices are incrementally updated between neighbors periodically.

Gia uses a biased random walk search protocol in which each peer selects the highest-capacity neighbor for which it has flow-control tokens and sends the query to that neighbor.

The combination of dynamic topology adaptation, one-hop replication, and biased random walk generates a similar effect to that of the super-node mechanism. However, compared with the super-node

network, Gia provides a more adaptive mechanism. The active flow-control scheme makes Gia one of few proposals to consider node capacity constraints in search optimization. Refer to Chawathe et al.[21] for more details on Gia.

36.5.2 DHT-Based Keyword Searching

Structured P2P networks actually implement a distributed hash table (DHT) layer by firmly controlling the topology of overlay networks and the mapping from data items to peers. Upper applications can insert a <key, value> pair into the system and retrieve a data item by its key. As DHT can only directly support search by an opaque key, to support keyword searching, some mechanism is needed to map keyword queries to unique routing keys of DHT.

Most existing DHT-based keyword searching proposals use the inverted index, the most widely used indexing structure in full-text searching systems, as the basic index structure. An inverted index comprises many inverted lists, one for each word. An inverted list for a word contains all the identifiers of documents in which the word appears. In a P2P keyword searching system, the logically global inverted index must be partitioned and placed at certain peers. There are two basic P2P index partitioning and placing strategies: (1) local indexing and (2) global indexing. With local indexing, each host maintains a local inverted index of the documents for which it is responsible. Using this strategy, each query must be flooded to all peers. Unstructured P2P networks use this strategy by default. The global indexing strategy assigns each keyword undivided to a single node, and each node maintains the inverted lists of some keywords. Global indexing is the basic strategy used by structured P2P networks.

36.5.2.1 Global Indexing and its Optimizations

The most obvious keyword searching scheme in structured P2P networks is global indexing, that is, partition the index by keyword. By this scheme, each keyword (and its corresponding inverted list) is mapped by hashing to a unique key of DHT layer and thus to a unique peer of the system. To answer a query consisting of multiple keywords, the query is sent to peers responsible for those keywords. Their inverted lists are transmitted over the network and intersected to get a list of documents that contain the keywords. Using this strategy, for a query that contains k keywords, at most k nodes need to be contacted. However, as keyword intersection requires that one or more inverted lists be sent over the network, bandwidth consumption and transmission time can be substantial. To alleviate this problem, some optimizations have been proposed. Here are some of them.

36.5.2.1.1 *Compression*

To reduce communication overhead of the inverted list intersection, data can be transferred in a compressed way. Bloom filters[34] are the most frequently used compression methods in existing optimization proposals.[35,36] A bloom filter is a hashing-based data structure that can summarize the contents of a set with a relatively small space, at the cost of a small probability of false positives. For two nodes whose inverted lists are to intersect using bloom filters, one node compresses its inverted list and sends the bloom filter to another node. The receiving node intersects the bloom filter and its inverted list, and sends back the result list of documents. The original sender then removes false positives and sends final query results to end users. Li et al.[35] employed four rounds of inverted list intersections using compressed bloom filters and obtained a compression ratio of roughly 50. Li et al.[35] also tried other compression techniques (e.g., gap compression, adaptive set intersection, document clustering, etc.).35

36.5.2.1.2 *Caching*

Caching has been widely used in many sub-fields of computer science to improve efficiency. For P2P systems with global indexing, caching schemes can be divided into two categories: (1) inverted list caching and (2) result caching. Inverted list caching means that peers cache the inverted lists sent to them for some queries to avoid receiving them again for future queries. Result caching, on the other hand, caches query

results rather than inverted lists. Because keyword popularity in queries roughly follows a Zipf distribution, it is hoped that a small cache of inverted lists or query results can reduce communication overhead and improve search efficiency remarkably. A data structure called *view tree* is proposed[37] to efficiently store and retrieve prior results for result caching. Reynolds and Vahdat[36] [efficiently] analyzed the effect of inverted list caching and proposed to cache bloom filters instead of inverted lists. This has two effects: (1) storage space is saved because bloom filters are more compact than inverted lists; and (2) caching bloom filters spends more communication overhead in answering queries than caching inverted lists. That is because, by caching bloom filters, the caching node needs to send the intersection results of its inverted list and the bloom filter to another node. Query results can be returned to end users directly by caching inverted lists.

36.5.2.1.3 *Pre-computation*

Using pre-computation, the intersection of some inverted lists can be computed and stored in the system in advance. Li et al.[35] tried to choose 7.5 million term pairs (about 3 percent of all possible term pairs) from the most popular terms of their dataset and pre-compute intersections. As a result, they obtained an average communication reduction of 50 percent. Gnawali[38] proposed a keyword-set search system (KSS) in his thesis. In KSS, the logically global index is partitioned by sets of keywords rather than keywords; and each query is divided into sets of keywords accordingly. In answering a query, the document list for each set of keywords is retrieved and intersected if necessary. KSS can be viewed as a kind of pre-computation. The effects of adopting pre-computation are twofold. Pre-computation reduces query processing overhead, but on the other hand, it also dramatically improves index building overhead and storage usage. Pre-computation shares some similar features with caching. Their relationship and the comparison of their effects remain to be studied.

36.5.2.1.4 *Incremental Intersection*

In most keyword searching systems, users nearly always need a few most relevant documents rather than all query results. This allows adopting the incremental intersection of inverted lists for reducing communication overhead and improving efficiency. Incremental intersection means that, in the intersection of two (or more) inverted lists, instead of transferring an inverted list as a whole to another node, the list is partitioned into multiple blocks and transferred incrementally. The intersection process terminates before the whole inverted list is transferred when some conditions are satisfied. Reynolds and Vahdat[36] combine the bloom filter technique and incremental intersection by dividing an inverted list into chunks and sending bloom filters of each chunk incrementally. To generate precise results when adopting incremental intersection, some algorithms must be used to guarantee that top k (k indicates the number of results users required) results have been generated when the intersection process terminates. One of the most frequently used algorithms is Fagin's algorithm.[39] Fagin's algorithm has some constraints to ranking functions that judge the relevance between each document and a given query. Unfortunately, some important ranking functions (e.g., term proximity) are not applicable to Fagin's algorithm. We face three choices in this circumstance. The first choice is compromising the quality of search results, either by abandoning the ranking functions not supported by Fagin's algorithm, or including such ranking functions while terminating early without the guarantee of generating top k results. As a second choice, we can negatively cease using incremental intersection to guarantee query result quality. The third choice may be the best choice; however, it needs our effort; that is, discover algorithms other than Fagin's algorithm that can support existing important ranking functions.

There also have been other optimizations. P-VSM[40] improves search efficiency by sacrificing query quality to some extent. In P-VSM, each document is represented by a vector by using the vector space model. And then the m most important components (keywords) of the document are indexed in the P2P system. Compared with the basic global indexing scheme, P-VSM reduces the overhead of index building and query processing by only indexing a small portion of keywords. However, the omission of some terms may affect its search precision. Another problem is that the IDF (inverse document frequency) values for terms needed in the vector space model must be globally computed.

36.5.2.2 Local Indexing

36.5.2.2.1 *Partition by Document*

P-LSI takes CAN[12] as its P2P substrate to implement keyword searching. The basic idea of P-LSI is to control the placement of documents such that documents stored close to each other in CAN are also close in semantics. Each document has a semantic vector that can be generated by LSI (latent semantic indexing), which uses SVD (singular value decomposition) to transform and truncate a matrix of documents and terms. To build the index, the index of each document is inserted into CAN using the semantic vector of the document as the key. Query processing in P-LSI follows a two-phase process. In answering a query, as the first phase, the semantic vector of the query (note that each query can be regarded as a, although short, document) is computed and the query is routed to a peer using its semantic vector as the key. In the second phase, the destination peer floods the query only to peers in a predefined radius r. Because the index of documents is stored by semantics, the search mechanism of P-LSI can retrieve highly relevant documents with high probability, while bothering only a small fraction of peers.

P-LSI is similar to the guide rules[32] and interest-based shortcuts[33] proposals in unstructured P2P networks. P-LSI uses information retrieval techniques to discover the latent semantic between documents, while the later two proposals are comparatively more ad hoc.

36.5.2.3 Hybrid Indexing Schemes

Global indexing and local indexing each have their advantages and drawbacks. There have been some combinations of the two for the purpose of achieving good trade-offs between them.

36.5.2.3.1 *eSearch*

The primary index partitioning and placement strategy used in eSearch[41] is global indexing; that is, an inverted index is distributed based on keywords, and each node is responsible for maintaining the inverted lists of some keywords. To avoid transmitting large amounts of data in query processing, eSearch also replicates document information on nodes that are responsible for at least one keyword of the document, called *metadata replication*. Metadata replication is implemented by having each node store the complete lists of terms for documents in its inverted lists. It is the combination of global index and document replication that makes the index structure "hybrid." With this hybrid index structure, a multi-keyword query need only be sent to any one of the nodes responsible for those terms and do a local search. Refer to Figure 1 in Ref. 41 for an intuitive description of the index structure of eSearch.

The hybrid index structure used by eSearch actually trades storage space for communication cost and search efficiency. However, its communication cost in index publishing is much more than that of the more commonly proposed global indexing scheme. To overcome this drawback, eSearch enhances the above naïve hybrid scheme by two optimizations. We call the first optimization *metadata selection,* which means that only the most important terms for each document are published. To implement metadata selection, each document is represented as a vector using the vector space model (VSM). And then, the top elements (or their corresponding terms) are selected. The second optimization involves adopting of overlay source multicast. Overlay source multicast provides a lightweight way to disseminate metadata to a group of peers and reduces the communication cost compared with other data disseminating schemes.

36.5.2.3.2 *MLP*

Multi-level partitioning (MLP)[42] is a P2P-based hybrid index partition strategy. The goal of MLP is to achieve a good trade-off among end-user latency, bandwidth consumption, load balance, availability, and scalability.

The definition of MLP relies on a node group hierarchy. In the hierarchy, all nodes are logically divided into k groups (not necessarily of equal size), and nodes of each group are further divided as k sub-groups. Repeat this process to obtain a node hierarchy with l levels. Groups on level l are called leaf groups. Given the hierarchical node groups, the logically global inverted index is partitioned by document among groups; that is, each group maintains the local inverted index for all documents belonging to this group. For each leaf group, the index is partitioned by keywords among nodes; that is, global index is built inside each leaf

group. MLP processes queries by the combination of broadcasting and inverted list intersection. Take a query (initiated from a node A) containing keyword w_1 and w_2 as an example. The query is broadcast from node A to all groups of level 1, and down to all groups of next levels, until level l is reached. For each group in level l, the two nodes that contain the inverted lists corresponding to keyword w_1 and w_2, respectively, are responsible for answering the query. In each group, the two inverted lists are intersected to generate the search results of the group. And then, the search results from all groups are combined level by level and sent back to node A. The implemented of MLP on top of SkipNet[43] is demonstrated by Shi et al.[42]

MLP has two features: (1) uniformity and (2) latency locality. Uniformity implies that all nodes must have roughly the same number of inverted lists on them. This is required for the balance of load and storage among peers in the system. Latency locality infers that intra-group latency should be smaller than inter-group latency on each level of the node hierarchy. This is needed for reducing end-user latency and bisection bandwidth consumption. With latency locality, all nodes in a group are roughly in the same sub-network. Therefore, bisection backbone bandwidth is saved by confining inverted list intersection inside each sub-network.

Compared with local indexing, MLP avoids flooding a query to all nodes by having only a few nodes in each group to process the query. MLP can dramatically reduce bisection backbone bandwidth consumption and communication latency when compared with the global indexing scheme.

36.5.2.3.3 *Hybrid Search Infrastructure*

It well known that while flooding-based techniques (with local index structure) are effective for locating popular items, but are not suitable for finding rare items. On the other hand, DHT-based global indexing techniques excel at retrieving rare items, but are inefficient in processing queries containing multiple popular terms. Loo et al.[44] proposed a simple hybrid index building and search mechanism, in which global indexing is used to index rare data items and local indexing with flooding is used for indexing and retrieving popular data items. Based on this hybrid mechanism, a hybrid search infrastructure is built. A key problem in hybrid index building and query processing is the identification of rare items. The hybrid search infrastructure utilizes selective publishing techniques that use some heuristics to identify rare items.

36.6 Application-Level Multicast

In some cases, we meet the requirements of data transmission or communication among more than two end-hosts (i.e., the group communication). In group communication, many end-hosts take part in a common task and exchange data simultaneously and cooperatively to implement more complicated applications beyond only one-to-one connection. For example, in considering an application for a network meeting with many attendant people, the system needs to transmit the voice or picture of the talker to all other persons. Other examples for group communication include online video stream and efficient delivery for bulk contents. In these applications of group communication, the *multicast* is a key component that facilitates efficient data delivery from one source to multiple destinations. Although network-level IP multicast was proposed over a decade (e.g., network architectures for multicast[45–47] and reliable protocols[48,49]), the use of multicast in applications has been very limited because of the lack of widespread deployment and the issue of how to track group membership. To alleviate this problem, several recent proposals have advocated an alternative approach, termed "*application layer multicast*" or "*end-host multicast,*" that implements multicast functionality at the application layer using unicast network-level services only, forming an overlay network between end-hosts.

Application-level multicast has a number of advantages over IP multicast. First, because it requires no router support, it can be deployed incrementally on existing networks. Second, application-level multicast is much more flexible than IP multicast and can adapt to diverse requirements from the application. The end-to-end argument[50] suggests that functions placed at low levels of a system (e.g., the network level) may be redundant or of little value when compared with the cost of providing them at that low level, partly because of insufficiency in low levels to accomplish the function alone and partly because of incomplete knowledge about an upper level. For example, to deal with the usual problem that various links in the

distribution tree have widely different bandwidth, a commonly used strategy is to decrease the fidelity of the content over lower-bandwidth links. Although such a strategy has merit for video streaming that must be delivered live, there exist content types that require bit-for-bit integrity, such as software. Obviously, such application-aimed diversity can only be supported in the overlay layer, using application-level multicast rather than IP multicast in the network layer.

As a result, the overlay-based application-level multicast has recently become a hot category in research, and many algorithms and systems are being proposed for achieving scalable group management and scalable, reliable propagation of messages.[51–56] For such systems, the challenge remains to build an infrastructure that can scale to and tolerate the failure modes of the general Internet, while achieving low delay and effective use of network resources. That is, to realize an efficient overlay-based multicast system, there are three important problems to be addressed.

In particular, multicast systems based on overlay should mainly consider the following problems:

What is the appropriate multicast structure? In addition, how can it be constructed in the (mostly decentralized) overlay networks?

How can overlay networks manage the numerous participating hosts (users), especially when there is huge diversity between users, either in their capacities or in behaviors?

How can the multicast algorithm and overlay networks adapt themselves to an Internet environment whose links are prone to frequent variations, congestion, and failures?

The first problem often reduces to how to construct and maintain an efficient, fault-tolerant spanning tree on top of overlay networks, because a spanning tree is the natural structure for multicast. Beyond single tree structures, recent research on peer-to-peer networks further take advantage of "perpendicular links" between siblings of the tree to overcome some of the drawbacks of tree structure. The second problem is often called "group management" in IP multicast; and in overlay and peer-to-peer networks, it becomes even more difficult because of the large system scale and decentralization. Indeed, sometimes the overlay is made up of only volunteer hosts, and we are unable to count on a dedicated, powerful host to take charge of the entire group management. In this case, it is helpful to employ suitable self-organized infrastructures presented in previous sections to perform decentralized group management. The third problem is typically solved by actively measuring the Internet and multicast connections, and using adaptive approaches for data transmission. Overlay networks usually consider the network connections between pairs of overlay nodes as direct, independent network links, regardless that some connections may actually share the same network link between routers. Thus, by measuring the underlying "black box" (mainly the connectivity and available bandwidth of overlay connections), the overlay network will become aware of the runtime status and may further perform self-adaptation to improve the performance.

The following sections explain how existing designs and approaches address the above three fundamental problems of overlay-based multicast.

36.6.1 Construction of Multicast Structures

To perform multicast on overlay, the primary problem lies in constructing the multicast structure, that is, deciding the paths for data flows. Consider the scenario that there is one source and a number of destinations, and the goal is to deliver data from the source to all destinations as quickly as possible. Most applications of group communication can be simplified to such a scenario. To deal with numerous destinations, some destination hosts act as intermediate nodes in the data transmission path and forward received data to other destinations. Thus, many approaches employ a multicast tree as the basic structure for one-source-multiple-destination application. Theoretically, the problem now becomes one of constructing a minimum spanning tree (MST) rooted at the source, given the overlay connective graph $G = (V, E)$ with weighted edges, where V represents all end hosts and E characterizes the overlay connections. Although there is an efficient algorithm for constructing an MST,[57] in practice the one-source-multiple-destination problem is not that easy, because it is usually impracticable to get the entire overlay connective graph G

in the Internet, and because the resulting MST usually contains nodes with high degree that which will be overloaded in multicast. So, in practice many multicast algorithms do not seek to find the optimal tree solution, but turn to construct the near-optimal spanning tree that has bounded node degree and is easily built based on only partial knowledge of connective graphs. Therefore, the considerations of a multicast structure are mainly the following: (1) improvement of latency and bandwidth; (2) fault tolerance; and (3) ease of construction, especially in a decentralized manner.

Overcast[62] is motivated by real-world problems to deliver bandwidth-intensive content using Internet, which is faced with content providers. An overcast system is an overlay consisting of a central source, any number of internal Overcast nodes (standard PCs with permanent storage) sprinkled throughout a network fabric, and clients with standard HTTP connection. Overcast organizes the internal nodes into a distribution tree rooted at the source and using a simple tree-building protocol. The goal of Overcast's tree algorithm is to maximize bandwidth to the root for all nodes, which is achieved by placing a new node as far away from the root as possible without sacrificing bandwidth to the root. Once a node initializes, it begins from the root a process of self-organization with other nodes. In each round, the new node considers its bandwidth to the current node as well as the bandwidth to the current node *through each of the current node's children*. If the bandwidth through any of the children is as high as the direct bandwidth to the current node, then one of these children becomes current and a new round commences. In this way, the new node can locate itself further away from the root while also guaranteeing available bandwidth. Therefore, the tree-building protocol is inclined to locate a node to a parent that is very near the node, mostly within localized Internet area. Thus, Overcast takes advantage of the multicast tree to maximize available bandwidth and the power for delivering content to large-scale hosts.

Bayeux[54] utilizes a prefix-based routing scheme that it inherits from an application-level routing protocol (i.e., the Tapestry) and thus it combines randomness for load balancing and scales to arbitrarily large receiver groups, with the properties of locality and tolerance to failures in routers and network links. Bayeux follows the idea of using the "natural multicast tree" associated with DHT-based routing schemas. Because we can easily derive a tree structure from most DHT-based routing schemas, it is beneficial to use such a tree for overlay multicast and thus leave the failure recovery and proximity issues to overlay's routing layer, as Bayeux and other systems do.[63] In Bayeux, the Tapestry overlay assists efficient multipoint data delivery by forwarding packets according to suffixes of listener node IDs, and the node ID base defines the fan-out factor used in the multiplexing of data packets to different paths on each router. A multicast packet only needs to be duplicated when the receiver node identifiers become divergent in the next digit. In addition, the maximum number of overlay hops taken by such a delivery mechanism is bounded by the total number of digits in the Tapestry node IDs. Therefore, Bayeux has a bounded multicast approach; that is, both hops (critical to latency) and host's out-degree (critical to bandwidth) are guaranteed to be small.

However, tree-based multicast has some inherent drawbacks and is sometimes insufficient for full use of available resources in a cooperative environment. The reason is that in any multicast tree, the burden of duplicating and forwarding multicast traffic is carried by the small subset of the hosts that are interior nodes in the tree. The majority of hosts are leaf nodes and contribute no resources. This conflicts with the expectation that all hosts should share the forwarding load. The problem is further aggravated in high-bandwidth applications, such as video or bulk file distribution, where many receivers may not have the capacity and availability required of an interior node in a conventional multicast tree. In addition, in tree structures, the bandwidth is guaranteed to be monotonically decreasing moving down the tree. Any loss high up the tree will reduce the bandwidth available to receivers lower down the tree. Despite a number of techniques being proposed to recover from losses and hence to improve the available bandwidth in an overlay tree (e.g., Birman et al.[51] and Byers et al.[58]), fundamentally, the bandwidth available to any host is limited by the bandwidth available from that node's single parent in the tree.

To overcome the inherent limitations of the tree, recent techniques propose to use "perpendicular links" to augment the bandwidth available through the tree (e.g., Bullet[59]) or to construct multicast *forest* in place of single tree (e.g., SplitStream[60]). In essence, these techniques share the same idea: that we should make use of every host in the overlay to enhance data transmission, rather than use only interior nodes of a single tree.

The key idea in SplitStream[60] is to split the content into k stripes and to multicast each stripe using a separate tree. Hosts join as many trees as there are stripes they wish to receive, and they specify an upper bound on the number of stripes they are willing to forward. The challenge is to construct this forest of multicast trees such that an interior node in one tree is a leaf node in all remaining trees and the bandwidth constraints specified by the nodes are satisfied. This ensures that the forwarding load can be spread across all participating hosts. For example, if all nodes wish to receive k stripes and they are willing to forward k stripes, SplitStream will construct a forest such that the forwarding load is evenly balanced across all nodes while achieving low delay and link stress across the system. Figure 36.3 illustrates the stripes in Splitstream and the multicast forest.

The challenge in the design of SplitStream is to construct such a forest of interior-node-disjoint multicast trees in a decentralized, scalable, efficient, and self-organizing manner. A set of trees is said to be "interior-node disjoint" if each node is an interior node in at most one tree and a leaf node in the other trees. SplitStream exploits the properties of Pastry's routing and Scribe's group member management (Scribe[61]; see details of Scribe in the section of group member management) to facilitate construction of interior-node-disjoint trees. Recall that Pastry normally forwards a message toward nodes whose nodeIDs share progressively longer prefixes with the message's key. Because a Scribe tree is formed by the routes from all members to the groupID (see below), the nodeIDs of all interior nodes share some number of digits with the tree's groupID. Therefore, it can be ensured that k Scribe trees have a disjoint set of interior nodes simply by choosing groupIDs for the trees that all differ in the most significant digit. Figure 36.3 illustrates the construction. The groupID of a stripe group is also called the stripeID of the stripe. To limit a node's out-degree, the "push-down" method, which is a built-in mechanism in Scribe, is used. When a node that has reached its maximal out-degree receives a request from a prospective child, it provides the prospective child with a list of its current children. The prospective child then seeks to be adopted by the child with lowest delay. This procedure continues recursively down the tree until a node is found that can take another child. Moreover, to fully use the spare capacity in some powerful hosts, these hosts are organized in an independent group of Scribe, called the spare capacity group. All SplitStream nodes that have less children in stripe trees than their forwarding capacity limit are members of this group. When there is a forwarding demand that the current node cannot afford, SplitStream will seek an appropriate node in the spare capacity group to help the fully loaded node in supplying stripe.

Bullet[59] aims at high-bandwidth multicast data dissemination, for example, delivering bulk content to multiple receivers via the Internet. Rather than sending identical copies of the same data stream to all nodes, Bullet proposes that participants in a multicast overlay cooperate to strategically transmit *disjoint* data sets

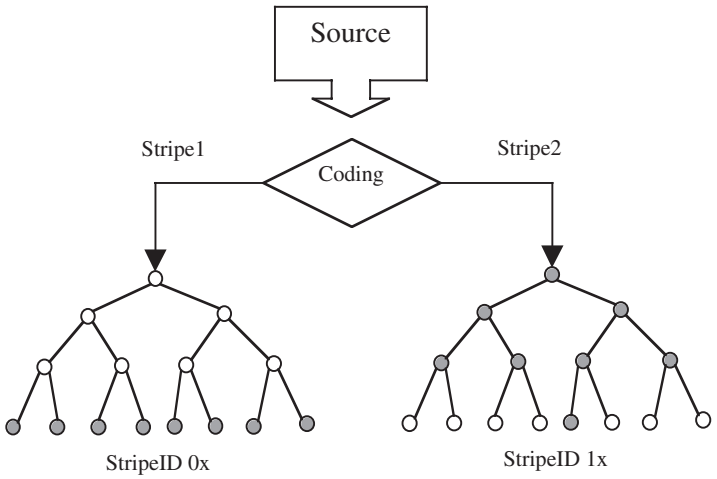

FIGURE 36.3 Stripes and forest of multicast trees in Splitstream.

to various points in the network. Nodes still receive a set of objects from their parents, but they are then responsible for locating hosts that hold missing data objects. Bullet has a distributed algorithm that aims to make the availability of data objects uniformly spread across all overlay participants. In this way, it avoids the problem of locating the "last object" and achieves persistent high-bandwidth data dissemination.

36.6.2 Group Member Management (Scribe)

In addition to the multicast structure, there are always requirements for managing participants, especially organizing hosts into many different groups according to the hosts' properties and also delivering messages to all members belonging to a certain group. Group member management is a basic overlay service and usually a necessary component for many multicast applications.

For this purpose, Scribe[61] is proposed as a decentralized application-level multicast infrastructure, aimed at member management for a large number of groups. Scribe is built on top of Pastry and takes advantage of Pastry's remarkable properties in network proximity, fault tolerance, and decentralization to support simultaneously a large number of groups. Any Scribe node can create a group; other nodes can then join the group, or multicast messages to all members of the group (provided they have the appropriate credentials). Each Scribe group has a unique groupID and the Scribe node with a nodeID numerically closest to the groupID acts as the rendezvous point for the associated group. The rendezvous point is the root of the multicast tree created for the group. When a Scribe node wishes to join a group g, it asks Pastry to route a JOIN message with the g's groupID as the key. This message is routed towards the g's rendezvous point. Each node along the route checks its list of groups to see if it is currently a forwarder of g; if so, it accepts the node as a child, adding it to the children table. Otherwise, it creates an entry for the g, adds the source node as a child and becomes a new forwarder for g by sending a JOIN message to the next node along the route from the joining node to the rendezvous point. So, the membership management mechanism is efficient for groups with a wide range of memberships, varying from one to all Scribe nodes. Pastry's randomization properties ensure that the tree is well balanced and that the forwarding load is evenly balanced across the nodes. This balance enables Scribe to support large numbers of groups and members per group. Furthermore, joining requests are handled locally in a decentralized fashion. In particular, the rendezvous point does not handle all joining requests. Therefore, Scribe can be used as a basic component for overlay-based multicast applications, and the notable example is SplitStream, which uses Scribe to form interior-disjoint multicast trees and organize hosts' spare capacities.

CoopNet[64] focuses on the problem of distributing "live" streaming media content from a server to a potentially large and highly dynamic population of interested clients. CoopNet uses multicast through clients to alleviate the heavy load in a streaming server and help a server tide over crises such as flash crowds, rather than replace the server with a pure peer-to-peer system. So, CoopNet benefits from a centralized management, which is much more efficient than a decentralized approach. There is a dedicated manager (a workstation) for managing the joins and leaves of clients. The manager stores the entire structure of the multicast tree (or trees) in its memory. When a client begins to receive live streaming, the client contacts the manager for JOIN operation. The manager then chooses an appropriate link place in the multicast tree from its memory, and responds with the designated parent to the client. In this way, the JOIN operation is very efficient and only needs one round of messaging. Although centralized management is not self-scaling, it is shown in CoopNet that such management is a rather lightweight task, and that a laptop with a 2 GHz Mobile Pentium 4 processor can keep up with about 400 joins and leaves per second. So, centralized management is also feasible and scales to large systems in some practical scenarios.

36.6.3 Overlay-Based Measurement and Adaptation for Multicast

In addition to multicast structure construction and group member management, an overlay application must continually adapt itself to the variational Internet environment. As is well known, the Internet is a highly dynamic environment and prone to unpredictable partitions, congestion, and crowd flashes. So, the multicast structure well-constructed at early time may become very inefficient after a period of time.

Consequently, overlay networks should be able to learn Internet variations by means of repetitive overlay-based measurement and re-estimation of its connections, and also adaptively transform the multicast structure in response.

Many probing-based techniques to measure overlay connections have been proposed65–70; they light-weight message probing to estimate the latency and available bandwidth between host pairs. In these techniques, the RTT (round-trip time) and 10-KB TCP probing are commonly utilized. In addition, some approaches[71] manage to estimate the bottleneck bandwidth. Although probing messages and lightweight estimations cannot fully characterize the property of connections, they are usually helpful in failure detection and path selection. In general, overlay networks regard the connections between end-host pairs as "black-boxes," and the measurement out-of-the-box is sufficient for overlay-based multicast systems.

After detecting variations, overlay networks must adapt themselves and multicast structures to improve performance. Many overlay designs employ localized adjustments of multicast structures, allowing hosts to dynamically change to a better service node. In Overcast,[62] to approximate the bandwidth that will be observed when moving data, the tree protocol measures the download time of 10 kbytes. This approach gives better results than approaches based on low-level bandwidth measurements such as using ping. In addition, a node periodically reevaluates its position in the tree by measuring the bandwidth to its current siblings (an up-to-date list is obtained from the parent), parent, and grandparent. Just as with the initial building phase. a node will relocate below its siblings if that does not decrease its bandwidth back to the root. The node checks bandwidth directly to the grandparent as a way of testing its previous decision to locate under its current parent. If necessary, the node moves back up in the hierarchy to become a sibling of its parent. As a result, nodes constantly reevaluate their position in the tree and the Overcast network is inherently tolerant of non-root node failures and Internet congestion.

36.7 Application

In recent years, P2P applications and practical systems have become a large category. A practical P2P system always needs integrated considerations in many aspects and combines many of the above approaches into one design. In this section we review a typical P2P application (i.e., P2P-based network storage) and see how the design issues that were previously discussed are implemented in practice.

36.7.1 Network Storage

Storage is a main application based on peer-to-peer architecture, in which storage spaces of all the nodes are combined and offered to users as a whole storage service for data storing, replication, backup, and other uses. Major projects on P2P storage systems include OceanStore,[8] PAST,[6] CFS,[7] Farsite,[72] Freenet,[3] and Granary.[26] As a practical application, peer-to-peer storage systems face many technical problems, such as routing, data location, data access control, replication, cache, update manner, service model, and etc. Not all projects focus on all these problems, and different projects choose different resolutions. Figure 36.4 provides a brief summary of the major aspects of current projects.

In the following we introduce OceanStore as a tangible example, for its research almost covers all the areas. The key design parameters of other projects will also be mentioned when necessary.

OceanStore is designed to be a very large storage pool, spanning the globe and offering reliable and efficient storage service to end users, especially to mobile clients, such as ubiquitous computing devices. Users can approach their data in OceanStore anytime, anywhere, and through any kind of device. Figure 36.5 illustrates the basic architecture of OceanStore, showing that the system is composed of a multitude of highly connected *pools*. Most of these pools are dedicated servers, provided by a confederation of companies, and any organization can be invited to participate in the system, even airports or small cafés, as long as they offer some storage servers. Users pay their fees for occupied spaces in the system, perhaps monthly, and all their data would be safe and never be leaked to other users or even to the system managers. Of course, users are permitted to grant data access authority to other users.

	Routing & Location	Access Control	Replication	Cache
OceanStore	Tapestry	√	√	√
CFS	Chord	–	√	√
Farsite	Broadcast	√	√	–
Freenet	DFS	√	–	–
PAST	Pastry	√	–	–
Granary	Tourist	√	√	–

	Update Manner	Service Model	Others
OceanStore	on block & archival	API & facades	Introspection
CFS	whole file replace	file system	load balance
Farsite	whole file replace	file system	
Freenet	whole file replace	file system	
PAST	file create & reclaim	file system	smart card
Granary	on object level	object storage	object query

FIGURE 36.4 Brief comparison of P2P storage projects.

To accomplish such goals, many technical problems should be tackled, and these are introduced in the following one by one.

36.7.1.1 Service Model

Different storage systems are based on different data units, including block, file, object, etc. The atomic data unit in OceanStore is "*object*," a piece of data with a global unique identifier. However, such a data form shows no substantial difference with the *file*, which is widely chosen by many other projects e.g., CFS, Farsite, Freenet, and PAST).

OceanStore offers a set of its own APIs, which provides full access to OceanStore functionality in terms of sessions, session guarantees, updates, and callbacks. Applications with more basic requirements are supported through facades to the standard APIs. A facade is an interface to the APIs that provides a traditional, familiar interface. For example, a transaction facade would provide an abstraction atop the OceanStore API so that the developer could access the system in terms of traditional transactions.

Another object-based system, Granary uses a more complex object model, which consists of a collection of user-defined attributes, just like the object concept in the OOP. This enables data query service for Granary, a significant functionality for many applications.

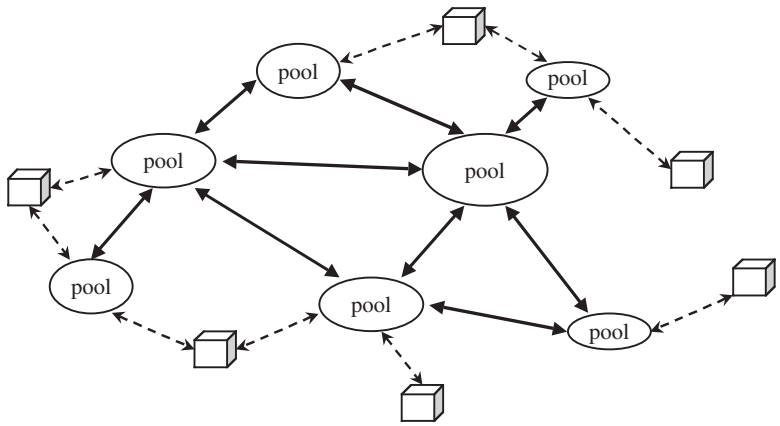

FIGURE 36.5 Architecture of OceanStore.

36.7.1.2 Data Location and Routing

PAST and CFS directly use DHT to store their data, which cannot avoid the trouble of massive data movement. OceanStore and Granary only use DHTs to store the metadata, mainly consisting of the replica lists. This greatly reduces the amount of data stored by DHTs and leave sufficient space for replication-algorithm design.

The basic routing layer in OceanStore is Tapestry,[15] which was introduced in Section 36.2.1. Moreover, a modified Bloom filter algorithm, called an attenuated Bloom filter, was combined. The attenuated Bloom filter[73] summarizes object items on a node's neighbors, in which whether a neighbor does not hold a given item can be judged accurately, while whether it does hold the item can be judged with high probability.

36.7.1.3 Access Control

OceanStore supports two primitive types of access control: *reader restriction* and *writer restriction*. More complicated access control policies, such as working groups, are constructed from these two.

To prevent unauthorized reads, all data is encrypted and the encryption key is distributed to those users with read permission. To revoke read permission, the owner must request that replicas be deleted or re-encrypted with the new key.

To prevent unauthorized writes, OceanStore requires that all writes be signed so that well-behaved servers and clients can verify them against an access control list (ACL). The owner of an object can securely choose the ACL for an object by providing a signed certificate. The specified ACL may be another object or a value indicating a common default. An ACL entry extending privileges must describe the privilege granted and the signing key, but not the explicit identity, of the privileged users. Such entries publicly are readable so that servers can check whether a write is allowed.

Peer-to-peer storage systems run on the open Internet, so the service provider must keep the data encrypted and unknown to other users or the system managers. Thus, almost every project involves access control design, and most are similar in spirit.

Farsite employs an interesting method — convergent encryption — to encrypt the data: first hashing the data content and then using the hashing result as the key to encrypt the data, by a symmetric encryption algorithm. When granting read authority to another user, the owner delivers the symmetric key, encrypted by the receiver's public key. In this way, the same files will always generate the same keys and then produce the same encrypted result, even if they belong to different owners. Therefore, the system can directly judge whether two files are identical, without knowing their content. This allows the system to reduce redundant storing of the same files. A following backup project Pastiche[74] also employs such an algorithm.

36.7.1.4 Replication

Objects in OceanStore are replicated and stored on multiple servers. This replication provides availability in the presence of network partitions and durability against failure and attack. A given replica is independent of the server on which it resides at any one time, and thus is called a floating replica.

Because object written authority can be granted to others, simultaneously updating must be coped with; that is, all the update requests should be uniformly ordered before executing. OceanStore separates all the replicas of a given object into two tiers, the primary tier performing a Byzantine agreement protocol to commit the updates and the secondary tier propagating the updates among themselves in an epidemic manner.

Most other projects do not assume such sophisticated replication. They only let the owner have the write permission and detour the replica consistence design.

36.7.1.5 Archival Storage

OceanStore objects exist in both active and archival forms. An active form of an object is the latest version of its data, together with a handle for update. An archival form represents a permanent, read-only version of the object. Archival versions of objects are encoded with an erasure code and spread over hundreds or thousands of servers; as data can be reconstructed from any sufficiently large subset of fragments, the result is that nothing short of a global disaster could ever destroy information.

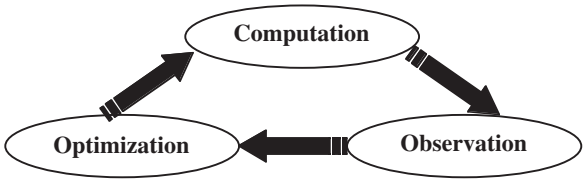

FIGURE 36.6 Introspection in OceanStore.

Erasure coding[75,76] is a process that treats input data as a series of fragments (say, n) and transforms these fragments into a greater number of fragments (say, $2n$ or $4n$). The essential property of the resulting code is that any n coded fragments are sufficient to construct the original data. In that OceanStore spans the globe, and all the fragments are identified by hashing and distributed in a DHT manner, all the fragments would be dispersed around the world, thus greatly improving the availability of archival data.

36.7.1.6 Introspection

OceanStore consists of millions of servers with varying connectivity, disk capacity, and computational power. Servers and devices will connect, disconnect, and fail sporadically. Server and network load will vary from moment to moment. Manually tuning a system so large and varied is prohibitively complex. Worse, because OceanStore is designed to operate using the utility model, manual tuning would involve cooperation across administrative boundaries.

To address these problems, OceanStore employs introspection, an architectural paradigm that mimics adaptation in biological systems. As shown in Figure 36.6, introspection augments a system's normal operation (computation) with observation and optimization. Observation modules monitor the activity of a running system and keep a historical record of system behavior. They also employ sophisticated analyses to extract patterns from these observations. Optimization modules use the resulting analysis to adjust or adapt the computation.

OceanStore takes advantage of introspection mainly in two ways: (1) cluster recognition, which detects clusters of strongly related objects; and (2) replica management, which adjusts the number and location of floating replicas in order to service access requests more efficiently.

References

1. Napster. http://www.napster.com.
2. Gnutella. http://gnutella.wego.com.
3. I. Clarke, O. Sandberg, B. Wiley, and T.W. Hong. Freenet: a distributed anonymous information storage and retrieval system. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability,* Berkeley, CA, June 2000. http://freenet.sourceforge.net.
4. KaZaA. http://kazaa.com.
5. BitTorrent. http://bitconjurer.org/BitTorrent/.
6. A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of ACM SOSP'01,* 2001.
7. F. Dabek, M.F. Kaashoek, D. Karger, et al. Wide-area cooperative storage with CFS. In *Proc. ACM SOSP'01,* Banff, Canada, Oct. 2001.
8. J. Kubiatowicz, D. Bindel, Y. Chen, et al. 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 2000),* 2000.
9. A. Muthitacharoen, R. Morris, T.M. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation,* 2002.

10. M. Castro, P. Druschel, A.-M. Kermarrec, et al. SplitStream — high-bandwidth multicast in cooperative environments. *SOSP'03,* 2003.
11. J. Eriksson, M. Faloutsos, and S. Krishnamurthy. PeerNet: pushing peer-to-peer down the stack. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03),* 2003.
12. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. *Annual Conference of the Special Interest Group on Data Communication (SIGCOMM 2001),* August 2001.
13. A. Rowstron and P. Druschel. Pastry: scalable, distributed object location and routing for large-scale peer-to-peer systems. *International Conference on Distributed Systems Platforms (Middleware 2001),* November 2001.
14. I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup service for Internet applications. *Annual Conference of the Special Interest Group on Data Communication (SIGCOMM 2001).* August 2001.
15. B. Zhao, Jo. Kubiatowicz, and A. Joseph. Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing. Technical Report UCB/CSD-01-1141, Computer Science Division, University of California, Berkeley. April 2001.
16. C. Plaxton, R. Rajaraman, and A. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the ACM SPAA,* Newport, RI, June 1997, pp. 311–320.
17. J. Xu, A. Kumar, and X. Yu. On the fundamental tradeoffs between routing table size and network diameter in peer-to-peer networks, *IEEE Journal on Selected Areas in Communications,* 22(1), 151–163, January 2004.
18. A. Gupta, B. Liskov, and R. Rodrigues. One hop lookups for peer-to-peer overlays. In *Proceedings of the Ninth Workshop on Hot Topics in Operating Systems,* Lihue, Hawaii, May 2003.
19. E. Ng, E. and H. Zhang. Towards global network positioning. In *Proceedings of ACM SIGCOMM Internet Measurement Workshop 2001,* November 2001.
20. S.Saroiu, P.K. Gummadi, and S.D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking 2002 (MMCN'02),* January 2002.
21. Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making Gnutella-like P2P systems scalable. In *Proceedings of ACM SIGCOMM 2003,* Karlsruhe, Germany, August 2003.
22. D. Karger, E. Lehman, T. 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 *29th ACM Annual Symposium on Theory of Computing. (STOC 1997).* May 1997.
23. A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in structured P2P systems. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS'03).* February 2003.
24. B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, I. Stoica. Load balancing in dynamic structured P2P systems. In *The 23rd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2004).* March 2004.
25. C. Blake and R. Rodrigues. High availability, scalable storage, dynamic peer networks: pick two. In *9th Workshop on Hot Topics in Operating Systems (HOTOS IX).* May 2003.
26. Granary project. Homepage at: http://166.111.68.166/granary/.
27. Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In Proc. *ACM ICS 2002,* 2002.
28. L.A. Adamic, R.M. Lukose, A.R. Puniyani, and B.A. Huberman. Search in power-law networks. *Physical Review E 64* (2001).
29. B. Yang and H. Garcia-Molina. Efficient search in peer-to-peer networks. In *ICDCS'02,* 2002.
30. A. Crespo and H. Garcia-Molina. Routing indices for peer-to-peer systems. In *ICDCS'02,* 2002.
31. B. Yang and H. Garcia-Molina. Designing a super-peer network. In *ICDE'03,* 2003.
32. E. Cohen, A. Fiat, and H. Kaplan. Associative search in peer to peer networks: harnessing latent semantics. In *INFOCOM'03,* 2003.
33. K. Sripanidkulchai, B. Maggs, and H. Zhang. Efficient content location using interest-based locality in peer-to-peer systems. In*INFOCOM'03,* 2003.

34. B.H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM,* 13(7):422–426, 1970.
35. J. Li, B.-T. Loo, J.M. Hellerstein, M.F. Kaashoek, D.R. Karger, and R. Morris. On the feasibility of peer-to-peer Web indexing and search. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03),* 2003.
36. P. Reynolds and A. Vahdat. Efficient peer-to-peer keyword searching. In *Middleware'03,* 2003.
37. Bhattacharjee, S. Chawathe, V. Gopalakrishnan, P. Keleher, and B. Silaghi. Efficient peer-to-peer searches using result-caching. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03),* 2003.
38. O.D. Gnawali. A Keyword Set Search System for Peer-to-Peer Networks. Masters thesis, Massachusetts Institute of Technology, June 2002. UCB/CSD-01-1141, University of California, Berkeley, April 2001.
39. R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Symposium on Principles of Database Systems,* 2001.
40. C. Tang, Z. Xu, and M. Mahalingam. Peersearch: efficient information retrieval in peer-to-peer networks. In *Proceedings of HotNets-I, ACM SIGCOMM,* 2002.
41. C. Tang and S. Dwarkadas. Peer-to-peer information retrieval in distributed hashtable systems. In *USENIX/ACM Symposium on. Networked Systems Design and Implementation (NSDI'04),* March 2004.
42. S. Shi, G. Yang, D. Wang, J. Yu, S. Qu, and M. Chen. Making peer-to-peer keyword searching feasible using multi-level partitioning. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS'04),* 2004.
43. N.J.A. Harvey, M.B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: a scalable overlay network with practical locality properties. In *USITS'03,* 2003.
44. B.-T. Loo, R. Huebsch, I. Stoica, and J.M. Hellerstein. The Case for a hybrid P2P search infrastructure. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS'04),* 2004.
45. S. Deering and D. Cheriton. Multicast routing in datagram internetworks and extended LANs, *ACM Transactions on Computer Systems,* 8(2), May 1990.
46. S.E. Deering. Multicast Routing in a Datagram Internetwork, Ph.D. thesis, Stanford University, December 1991.
47. S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C. Liu, and L.Wei. The PIM architecture for wide-area multicast routing, *IEEE/ACM Transactions on Networking,* 4(2) April 1996.
48. S. Floyd, V. Jacobson, C.G. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing, *IEEE/ACM Transaction on Networking,* 5(4):784–803, December 1997.
49. J.C. Lin and S. Paul. A reliable multicast transport protocol. In *Proc. of IEEE INFOCOM'96,* 1996, pp. 1414–1424.
50. J.H. Saltzer, D.P. Reed, and D.D. Clark. End-to-end arguments in system design, *ACM Transactions on Computer Systems,* 4(2); 277–288, November 1984.
51. K.P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast, *ACM Transactions on Computer Systems,* 17(2); 41–88, May 1999.
52. P. Eugster, S. Handurukande, R. Guerraoui, A.-M. Kermarrec, and P. Kouznetsov. Lightweight probabilistic broadcast. In *Proceedings of The International Conference on Dependable Systems and Networks (DSN 2001),* Gothenburg, Sweden, July 2001.
53. L.F. Cabrera, M.B. Jones, and M. Theimer. Herald: achieving a global event notification service. In *HotOS VIII,* Schloss Elmau, Germany, May 2001.
54. S.Q. Zhuang, B.Y. Zhao, A.D. Joseph, R.H. Katz, and J. Kubiatowicz. Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of the Eleventh International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV 2001),* Port Jefferson, NY, June 2001.
55. Y.-H. Chu, S.G. Rao, and H. Zhang. A case for end system multicast. in In *Proceedings of ACM Sigmetrics,* Santa Clara, CA, June 2000, pp. 1–12.

56. P.T. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The Many Faces of Publish/Subscribe, Technical Report DSC ID:2000104, EPFL, January 2001.
57. H. Gabow. Two algorithms for generating weighted spanning trees in order. *Siam J. Computing,* 6:139–150, 1977.
58. J.W. Byers, J. Considine, M. Mitzenmacher, and S. Rost. Informed content delivery across adaptive overlay networks. In *Proceedings of ACM SIGCOMM,* August 2002.
59. D. Kostic, A. Rodriguez, J.R. Albrecht, and A. Vahdat: Bullet: high bandwidth data dissemination using an overlay mesh. In *Proceedings of the 19th ACM Symposium on Operating System Principles,* October 2003. pp. 282–297.
60. M.Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: high-bandwidth content distribution in cooperative environments. In *Proceedings of the 19th ACM Symposium on Operating System Principles,* October 2003.
61. M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. SCRIBE: a large-scale and decentralized application-level multicast infrastructure. *IEEE JSAC,* 20(8), October 2002.
62. J. Jannotti, D.K. Gifford, and K.L. Johnson. Overcast: reliable multicasting with an overlay network. In *USENIX Symposium on Operating System Design and Implementation,* San Diego, CA, October 2000.
63. S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Application-level multicast using content-addressable networks. In *Proceedings of the 3rd International Workshop on Networked Group Communication,* November 2001.
64. V.N. Padmanabhan, H.J. Wang, and P.A. Chou. Resilient peer-to-peer streaming. In *11th IEEE International Conference on Network Protocols (ICNP'03),* November 04–07, 2003, Atlanta, GA.
65. T.S.E. Ng, Y.-H. Chu, S.G. Rao, K. Sripanidkulchai, and H. Zhang. Measurement-based optimization techniques for bandwidth-demanding peer-to-peer systems. In *IEEE INFOCOM'03,* 2003.
66. R.L. Carter and M. Crovella. Server selection using dynamic path characterization in wide-area networks. In *Proceedings of IEEE INFOCOM,* 1997.
67. M. Sayal, Y. Breitbart, P. Scheuermann, and R. Vingralek. Selection algorithms for replicated web servers. In *Proceedings of the Workshop on Internet Server Performance,* 1998.
68. Z. Fei, S. Bhattacharjee, E.W. Zegura, and M.H. Ammar. A novel server selection technique for improving the response time of a replicated service. In *Proceedings of IEEE INFOCOM,* 1998.
69. K. Obraczka and F. Silva. Network latency metrics for server proximity. In *Proceedings of IEEE Globecom,* December 2000.
70. K. Hanna, N. Natarajan, and B.N. Levine. Evaluation of a novel two-step server selection metric. In *Proceedings of IEEE ICNP,* November 2001.
71. K. Lai and M. Baker. Nettimer: a tool for measuring bottleneck link bandwidth. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems,* March 2001.
72. W. Bolosky, J. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (Sigmetrics 2000),* June 2000.
73. S.C. Rhea and J. Kubiatowicz. Probabilistic location and routing. In *The 21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2002),* June 2002.
74. L.P. Cox, C.D. Murray, and B.D. Noble. Pastiche: making backup cheap and easy. In *5th Symposium Operating Systems Design and Implementation (OSDI'02),* December 2002.
75. M. Luby, M. Mitzenmacher, M. Shokrollahi, D. Spielman, and V. Stemann. Analysis of low density codes and improved designs using irregular graphs. In *30th ACM Annual Symposium on Theory of Computing. (STOC 1998),* May 1998.
76. J. Plank. A tutorial on Reed-Solomon coding for fault tolerance in RAID-like systems. *Software Practice and Experience,* 27(9):995–1012, September 1997.