Survey of Locating & Routing in Peer-to-Peer Systems

Fox Harrell, Yuanfang Hu, Guilian Wang, Huaxia Xia
Department of Computer Science and Engineering
University of California, San Diego
{redfox, yhu, guilian, hxia}@cs.ucsd.edu

Abstract Peer-to-peer computing is a term used to describe the current trend toward utilizing the full resources available within a widely distributed network of nodes with increasing computational power. These resources include the exchange of information, processing cycles, cache storage, and disk storage for files. Routing, locating and transmitting of resources, becomes an extremely important issue then. First steps toward robust peer-to-peer systems include extensions of centralized models of resource sharing (e.g. Napster), but more recent attempts acknowledge the limitations of such systems and also address issues of performance, reliability, scalability, maintenance, and usability. We focus on the problem of sharing data and in the context of these issues offer a survey of the following systems: Napster, Gnutella, TRIAD, Pastry, Plaxton, Tapestry, Chord, and CAN. The upshot is that we isolate what we see as both the crucial issues and solutions to the challenge of routing in the peer-to-peer environment, give a comparative summary for all systems. Finally based on framework and analysis, we propose some further questions and potential approaches for future investigation.

Keywords Peer-to-peer, Locating, Routing, Data sharing

1. Introduction

Peer-to-peer systems [10, 11] feature the many of the same desiderata of many software engineering projects: time efficient performance, scalability, ease of maintenance, reliability and usability. There are a variety of techniques used to achieve these goals. In this survey a crucial task was to separate the difference between end goals and features used to attain these goals, for example, it is not immediately clear whether data mobility is an end goal for a peer-to-peer system or simply a mechanism to achieve some optimizations in performance and ease of maintenance. A solution to this problem was to make a list of the crucial features surrounding the issue of routing in distributed systems and to see which affordances each feature allowed us. Sometimes we noticed that several features which afforded different end goals were bundled under one banner, for example, “adaptability” affords us both ease of maintenance and increased performance by exploiting locality of data. So we could see that within adaptability there are various advantages of data managing to support data mobility. In this manner we distilled out a list of five features that seem to capture the essence of the proposed enhancements to improve routing in peer-to-peer systems.

We propose to use this type of framework in which to survey the state of routing in several recent systems. The following chart represents the particular features we noticed and their affects upon desirable traits within the system.

We do not present this chart as exhaustive by any means, but merely as a tool to isolate crucial issues. It presents a framework within which to discuss various enhancements and areas in which necessary improvements are being incorporated. Within the context of each of the following specific systems we analyze specific gains to be made. An additional benefit of our chart is that it allows us to conclude with a synthetic suggestion for the focus of future research in this area.

2. Discussion of Specific Features and Problems Addressed

There are five desirable traits that we felt were relevant for any peer-to-peer system. These traits specifically are(table 1):

Performance This is the total time in data read, insert and delete operations. Factors include the locality of data, load balancing, the efficiency of the locating algorithm, and the efficiency of the routing protocol.

Scalability This includes the ability of the system to remain tractable with an increasing number of nodes and data elements. Factors include the balance of space complexity with time complexity.

Maintenance This includes the amount of human effort required to maintain the system. Factors include the amount of data and topology management that is automated, and the complexity of the code, the data representations, and the network structure.

Reliability This includes the avoidance of failure within the system and the ease of recovery if a failure occurs. Factors include data replication, and node failure detection and recovery, and the existence of multiple guarantees for location information to avoid a single point of failure. Another issue is the availability of multiple paths to data.

Usability This includes the ease of use, availability of control options, and variety of quality services that the system offers the end user. Factors include the flexibility of the querying system and simplicity of the user interface.

There are five main feature categories that seemed to be the most important contributing factors to the desirable traits. These are:

Naming This is the method used to represent shared data objects, network addresses of the nodes, and the structure of

Table 1. The features related to peer-to-peer data sharing

	Performance	Scalability	Maintenance	Reliability	Usability
Naming	Protocol and naming integration	Hierarchical name spaces	Hierarchical name spaces	–	Semantic flexibility
Structuring	Exploit locality	Dynamic structure	Dynamic structure	Multiple root nodes	–
Locating & Routing	Efficient location	Decentralized algorithms	–	–	–
Data Managing	Locality exploitation and load balance	Relocate data to maximize storage	Relocate data upon topological changes	Replication & relocation of data, load balance	User control of data replication
Topology Updating	Create optimal links	Replication, decentralization	Automatic restructuring	Replication of data and links, node failure recovery	–

routing requests across the network. Using an appropriate addressing scheme works hand in hand with the algorithms used to increase performance. Using hierarchical name spaces and name spaces that take into account the long-term usage of the system increases scalability. A well-structured name space also is more tractable for a human operator that eases maintenance. Semantic flexibility of naming allows for a variety of query patterns that enhances usability.

Structuring This includes the organization of the topology and the data structures maintained at each node which are used for locating and routing. An efficiently structured system minimizes storage requirements, a key factor in enhancing scalability of the system.

Locating & Routing These are the algorithms used to locate data and rout it to a server. Efficient algorithms minimize overhead of requests/queries and increase both scalability and performance.

Data Managing This includes the abilities to add, delete, replicate, and dynamically shift the location of data between nodes. This affects performance because it allows the system to exploit locality and balance the load by distributing data to less congested nodes. It allows the system to scale by relocating data to maximize storage. It allows for reliability by relocating data in case of node failure. Replication also increases reliability by increasing redundancy and locality.

Topological Updating This includes the abilities to add links, add nodes, and delete nodes in the network. This allows for performance to increase by structuring the network to decrease the distance between clients and data nodes. It allows for the system to be decentralized and avoid the problems centralized server. Automatic restructuring of the topology based upon usage minimizes human effort to perform those tasks—easing maintenance. We can duplicate links and create new nodes easily to increase reliability.

These combinations of traits and features are illuminating with regard to specific systems. In the subsequent section we survey several specific systems in this light.

3. Discussion of Individual System Features

3.1 Basic Routing Systems

Napster [1] and Gnutella [2] are two early routing systems which use centralized and decentralized servers respectively. These have been some of the most popular peer-to-peer systems, though they are less interesting from a research standpoint. In this section we describe routing in Napster and Gnutella as base level systems from which we can contrast the approaches and enhancement proposed in the subsequent systems discussed. TRIAD [3] is a content-based routing system that can be seen as a sort of intermediary between completely decentralized and centralized systems. In this section we also offer a brief description of TRIAD as some aspects of content based routing are also incorporated in the more complicated and robust systems described later.

3.1.1 Napster

Napster is a simply structured centralized system. With respect to the features given above, Napster offers no enhancements beyond base functionality in any of them. With regard to the desired traits it has many serious limitations in all of them, perhaps save usability where its socially enforced policies have made it very successful. We present it here as a sort of simplest model (which was very successful socially) to contrast the other systems to. It uses a centralized server to create its own flat namespace of host addresses. When a client makes a request to a server, it searches first over the client's assigned server and then begins search other servers until it finds the correct number of responses e.g. one hundred matching music files. These files are organized according to an array of search criteria.

There are problems with using a centralized server including the fact that there is a single point of failure. Napster does not replicate data. It uses "keepalives" to make sure that its directories are current. Maintaining a unified view is computationally expensive in a system like Napster. Scaling up can be a problem. It has been a very socially successful system though. The focus on Napster as a music sharing system in which users must be active in order to participate

has made it exceedingly popular. Regarding routing, it is simply a centralized directory system using Napster servers.

3.1.2 Gnutella

Gnutella is one of the earliest peer-to-peer file sharing systems that are completely decentralized. In Gnutella, each node is identified by its IP address and connected to some other nodes. All communication is done over the TCP/IP protocol. To join to the network, the new node needs to know the IP address of one node that is already in the system. It first broadcasts a "join" message via that node to the whole system. Each of these nodes then responds to indicate its IP address, how many files it is sharing, and how much space those files take up. So, in connecting, the new node immediately knows how much is available on the network to search through.

Gnutella uses file name as the key. Once a search message is sent out to request a name match, it is propagated through the network. Each node that has matching terms passes back its result set. Each node handles the search query in its own way. To save on bandwidth, a node does not have to respond to a query if it has no matching items. The node also has the option of returning only a limited result set.

After the client node receives response from other nodes, it uses HTTP to download the files it wants.

Gnutella is completely decentralized. So there's no single point of failure and the scalability is also a little better than Napster. But the nodes are organized loosely, so the costs for node joining and searching are O(N), which means that Gnutella cannot grow to a very large scale.

3.1.3 TRIAD

TRIAD is not a comprehensive peer-to-peer system, but a solution to the problem of content based routing. Its goal is to reduce the time need to access content. Despite that it is focused on the performance problem, it also represents improvements in other traits. The core idea in TRIAD is network-integrated content routing. It is an intermediary system between a centralized model and a fully decentralized model because it relies upon using replicated servers. So a client can go through one of a variety of servers to reach content as long as each server hosts the content. Integrated into the system are content routers to act as both IP routers and name servers.

The main idea is that the content routers hold "name to next hop" information so that all routing is done through adjacent servers so that each step is on the path to the data, avoiding some of the back and forth calling of traditional DNS. They also explore piggybacking connection set-up on the name lookup so that immediately upon locating the data the connection is already established. Reliability is increased because the system topology is structured so that there are multiple paths to content. TRIAD increases performance by proposing its name based content routing as a topological enhancement. This reduces a lot of the overhead from a DNS based system. Its protocols make it easier to maintain the system by using routing aggregates instead of a large number or individual names.

The core ideas in TRIAD relate to peer-to-peer because in such a system end users' machines can act as either content routers or servers, or both. At minimum this system could replace the centralized servers of a Napster type system.

3.2 Pastry

Pastry [4] is a generic peer-to-peer content location and routing system based on a self-organizing overlay network of nodes connected via the Internet. It is completely decentralized, scalable, fault-resilient, and reliably routes a message to the live node with a nodeId numerically closest to a key with that message; it automatically adapts to the arrival, departure and failure of nodes.

Naming Each node in the Pastry peer-to-peer overlay network has a unique 128-bit nodeId, this nodeId is assigned randomly when a node joins the system by computing a cryptographic hash of the node's public key or its IP address. With this naming mechanism, Pastry makes an important assumption that nodeIds are generated such that the resulting set of nodeIds is uniformly distributed in the nodeId space. Each data also has a 128-bit key. This key can be the original key, or generated by a hash function. The data is stored in the node whose id is numerically closest to it key.

Structuring Each Pastry node maintains a routing table, a neighborhood set and a leaf set.

- **Routing table** Assuming a network consisting of N nodes, a node's routing table is organized into logN rows with 2b-1 entries each row. The nth row of the routing table contains the nodeIds and IP addresses of those nodes, whose nodeId shares the present node's nodeId in the first n digits but different in the n+1 digit. If there are more than 2b-1 qualified nodes, the closest 2b-1 nodes will be selected, according to proximity metric.
- **Neighborhood set** Neighborhood set contains the nodeIds and IP addresses of the nodes that are closest to the present node.
- **Leaf set** Leaf set contains the nodeIds and IP addresses of the half nodes with numerically closest larger nodeIds, and half nodes with numerically closest smaller nodeIds, relative to the present node's nodeId.

Locating & Routing Given a message, the node first checks to see if the key falls within the range of nodeIds covered by its leaf set. If so, the message is forwarded directly to the destination node, namely the node in the leaf set whose nodeId is closest to the key. If the key is not covered by leaf set, then the routing table is used and the message is forwarded to a node that shares a common prefix with the key by at least one

more digit. In certain cases, it is possible that the appropriate entry in the routing table is empty or the associated node is not reachable, in which case the message is forwarded to a node that shares a prefix with the key at least as long as the present node, and is numerically closer to the key than the present node's nodeId. Such a node must be in the leaf set unless the message has already arrived at the node with numerically closest nodeId.

Data Managing Pastry supports dynamic data object insertion and deletion, but does not explicitly support for mobile objects.

Topology Updating Pastry supports dynamic node join and departure.

3.3 Plaxton

Plaxton et al. present in [5] a distributed data structure, called a Plaxton mesh, optimized to support a network overlay for locating named objects and routing of messages to those objects.

Naming In Plaxton, objects and nodes have identifiers independent of their location and semantic properties, in the form of random fixed-length bit-sequences represented by a common base. The system assumes entries are roughly evenly distributed in both node and object namespaces, which can be achieved by using hashing algorithms.

Structuring Each node's involved auxiliary memory is partitioned into two parts, namely the neighbor table and pointer list of the node. For each node X, the neighbor table consists of (logN)/b levels. Each level represents a matching suffix up to a digit position in the ID. A given level of the neighbor table contains a number of entries equal to b, the base of the ID. Each node X also maintains a pointer list Ptr(X) with pointers to copies of some objects in the network in form of <A, Y, k> which represents node Y has a copy of object A with cost function value of k. At most one record associated with any object is maintained in Ptr(X). For each object, it is always guaranteed that there is a node that maintains the location mapping for it.

Locating & Routing During a location query, the client sends message destined for object A is initially routed towards A's root. At each step, if the message encounters a node that contains the location mapping for A, it is immediately redirected to the server containing the object. If the message reaches the root, it is guaranteed to find a mapping for the location of A. The local neighbor table is used to incrementally route messages to the destination ID digit by digit. This approach is similar to longest prefix routing in the CIDR IP address allocation architecture [6]. This routing method guarantees that any existing unique node in the system will be found within at most $\log_b N$ logical hops, in a system with an N size namespace using IDs of base b. A way to visualize this routing mechanism is that every destination node is the root node of its own tree, which is a unique spanning tree across all nodes. Any leaf can traverse a number of intermediate nodes en route to the root node

Data Managing Insertion of object A on node X is done through sending an insert message, or called publish message, to the root node of A. At each hop Y along the way, the publish message updates the Ptr(Y) by adding <A, X, k>. Where multiple copies of A exists, only the reference to the closest copy is saved at each hop to root. By sending a delete message for A generated by node X to A's root eventually removes all <A, X, *> from the Ptr(Y) where Y is in the neighbor sequence from A. But Plaxton does not explicitly support mobile objects. The complexity is O(C) for data insert operations and is O(ClogN) for delete where C is the maximum cost for communication between any two nodes.

Topology updating Plaxton does not support dynamic node insertion or deletion, and doesn't handle node failure, hence the Plaxton Mesh is a static data structure.

3.4 Tapestry

Tapestry [6] is an overlay infrastructure designed as a routing and location layer in OceanStore [7]. Tapestry mechanisms are modeled after the Plaxton scheme, its naming, structuring, and core locating & routing scheme are the same as Plaxton, but they provide adaptability, fault-tolerance against multiple faults, and introspective optimizations.

Structuring As the same as Plaxton structuring scheme, each node has a neighbor map, which is organized into routing levels, and each level contains entries that point to a set of nodes closest in network distance that matches the suffix for that level. Each node also maintains a back pointer list that points to nodes where it is referred as a neighbor. They are used in node integration algorithm to generate neighbor maps for a node, and to integrate it into Tapestry. However, instead of assigning a single root for an object in Plaxton, Tapestry uses a distributed algorithm, called Surrogate Routing, to incrementally compute a unique root node for a object; and moreover each object gets multiple root nodes through concatenating a small globally constant sequence of salt values to each object ID, then hashing the result to identify the appropriate roots.

Locating & Routing When locating an object, tapestry performs the same hashing process with the target object ID, generating a set of roots to search. Where multiple copies of data exist in Plaxton, each node en route to the root node only stores the location of the closest replica to it. Tapestry, however, stores locations of all such replicas to increase semantic flexibility. There are only some small modifications I routing mechanism for improving fault-tolerance, e.g. in case of bad links encountered, routing can be continued by jumping to a random neighbor node.

Data managing Although the insertion and deletion of objects are the same as Plaxton, except Tapestry send publish and delete message to multiple roots, Tapestry provide explicit

support for mobile objects. The complexity for insert and delete is the same as Planxton.

Topology Updating Node insertion is easily implemented through populating neighbor maps and neighbor notification. Node deletion is more trivial. It is worth notice that Tapestry provides two introspective mechanisms to allow Tapestry to adapt to environmental changes. First, in order to adapt to the changes of network distance and connectivity, Tapestry nodes tune their neighbor pointers by running a refresher thread which uses network Pings to update network latency to each neighbor. Second, Tapestry presents an algorithm that detects query hotspots and offers suggestions on locations where the additional copies can significantly improve query response time.

3.5 Chord

Chord [8] is a distributed lookup protocol designed by MIT. It supports fast data locating and node joining/leaving.

Naming Each machine is assigned an m-bit nodeID, which is got by hashing its IP address. Each data record (K, V) has its unique key K. In Chord, it is also assigned an m-bit ID by hashing the key, P=hash(K). This ID is used to indicate the location of the data.

Structuring All the possible $N=2^m$ nodeIDs are ordered in a one-dimensional circle; the machines are mapped to this virtual circle according to their nodeIDs. For each nodeID, the first physical machine on its clockwise side is called its *successor node*, or succ(nodeID).

Each data record (K, V) has an identifier *P=hash(K)*, which indicates the virtual position in the circle. The data record (K, V) is stored in the first physical machine clockwise from *P*. This machine is called the *successor node* of *P*, or succ(P).

To do routing efficiently, each machine contains part of the mapping information. In the view of each physical machine, the virtual cycle is partitioned into 1+logN segments: itself, and logN segments with length 1, 2, 4, …, N/2. The machine maintains a table with logN entries; each entry contains the information for one segment: the boundaries and the successor of its first virtual node. In this way, each machine only need O(logN) memory to maintain the topology information. And we will see that the information is sufficient for fast locating/routing.

Locating & Routing On query for a record with key *K*, the virtual position is first be calculated: *P=hash(K)*. The locating can start from any physical machine. Using the mapping table, the successor of the segment that contains P is selected to be the next router until *P* is lies between the start of the segment and the successor (this means the successor is also P's successor, i.e., the target). The distance between the target and the current machine will decrease by half after each hop. Thus the routing time is O(logN).

Data Managing For high availability, the data can be replicated using multiple hash functions; we can also replicate the data at the *r* machines succeeding its data ID. All the data operation is in O(logN) time.

Topology Updating In Chord, machines can join and leave at any time. For normal node arrival and departure, the cost is $O(\log^2 N)$ with high probability, but in the worst case, the cost is O(N). The node failure can also be detected and recovered automatically if each node maintains a "successor-list" of its *r* nearest successors on the Chord ring.

3.6 CAN

CAN (Content-Addressable Network) [9] is a distributed hash-based infrastructure that provides fast lookup functionality on Internet-like scales.

Naming In CAN, the machines are addressed by their IP addresses. Each data record has its unique key *K*. A hash function assigns a d-dimensional vector *P=hash(K)* for each key, which corresponds a point in d-dimensional space. In CAN, the point indicates the virtual position for the data.

Structuring CAN maintains a d-dimensional virtual space on a "d-torus ". The virtual space is partitioned into many small d-dimensional zones. Each physical machine corresponds to one zone and stores the data that are mapped to this zone by the hash function. In the d-dimensional space, two nodes are neighbors if their coordinate spans overlap along d-1 dimensions and about along one dimension. Each machine knows the zones and IP addresses of its neighbors.

Locating & Routing For a given key, the virtual position will be calculated; then starting from any physical machine, the query message is passed through the neighbors until it find the IP address of the target machine. In a d-dimensional space, each node maintains 2d neighbors [at most 4d, in fact] and the average routing path length is $(d/4)(n^{1/d})$ hops. If d=(logN)/2, it will achieve the same scaling properties as Chord.

Data Managing CAN supports data insertion and deletion in $(d/4)(n^{1/d})$ hops. In CAN, a machine can also copy its data to one or more of its neighbors. This is very useful for load balance and fault tolerance.

Topology Updating CAN supports dynamic machine joining and leaving. It can also detect and recover node failure automatically. The average cost for machine joining is $(d/4)(n^{1/4})$ hops; for machine leaving and failure recovering, it's constant time.

4. Comparative Property Analysis

In this chapter, we give the detailed information about the features of each system in table 2. Next we compare the

Table 2. The features of the systems

	Napster	Gnutella	TRIAD	Pastry	Plaxton	Tapestry	Chord	CAN
Decentralized	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
[(0)]Space cost	O(N)	Depends	O(logN)	O(logN)	O(logN)	O(logN)	O(logN)	2d
Data read	[(1)]O(1)	O(N)	O(logN)	O(logN)	O(logN)	O(logN)	O(logN)	[(4)]$O(N^{1/d})$
Data insert	O(1)	O(1)	O(logN)	O(logN)	O(logN)	O(logN)	O(logN)	[(4)]$O(N^{1/d})$
Data delete	O(1)	O(1)	O(logN)	O(logN)	$O(\log^2 N)$	$O(\log^2 N)$	O(logN)	[(4)]$O(N^{1/d})$
Node insert	O(1)	O(N)	–	[(2)]O(logN)	Hard	O(logN)	[(3)]O(logN)	[(4)]$O(N^{1/d})$
Node delete	O(1)	O(1)	–	[(2)]O(logN)	Hard	O(logN)	[(3)]O(logN)	O(1)
Node failure	–	–	No	O(logN)	No	–	O(logN)	O(1)
Locality	Yes	–	–	Yes	Yes	Yes	No	No
Load balance	–	No	No	No	No	No	–	Yes

Note:

(*): "–" means the data is not accessible or special technique is not necessary. "No" means that specific technique is not provided.

(0): Here the space is the storage used for node routing.

(1): All the time cost is based on the "hops", i.e., the number of messages between the nodes.

(2) (3). The node insertion/deletion in Pastry or Chord needs O(N) time in the worst cases.

(4). The basic complexity in CAN is $O(N^{1/d})$, where d is the dimension of the node space, N is the number of real nodes (while in Pastry, Plaxton, Tapestry, and Chord, N is the size of virtual space). The constant coefficient is about $3*2^d/d$. If d=(logN)/2, it will achieve the same scaling properties as O(logN).

Table 3. The overall comparison between the systems

	Napster	Gnutella	TRIAD	Pastry	Plaxton	Tapestry	Chord	CAN
Performance	Bad	Bad	Good	Good	Good	Good	Good	Good
Scalability	Bad	Bad	Good	Good	Good	Good	Good	Good
Reliability	Bad	Good	Good	Good	Moderate	Good	Good	Good
Maintenance	Best	Best	Moderate	Good	Bad	Good	Good	Best
Usability	Good	Good	Simple	Simple	Simple	Good	Simple	Simple

the systems in their performance, scalability, scalability, reliability, and maintenance (table 3).

4.1 Performance

We say Napster is bad because it uses a central server that is likely to be over-loaded. The server needs large storage to maintain the information about all the nodes and data; the response time will increase when the number of nodes and requests exceed the capability of the server.

Though Gnutella is completely decentralized, its performance is not satisfying. Because the nodes are organized loosely, the costs for node joining and searching are O(N).

All the other systems perform well. They all have logN-like performance.

4.2 Scalability

Napster needs O(N) storage and computing power, Gnutella needs O(N) routing time cost, so we say that Napster and Gnutella is not satisfying.

4.3 Reliability

Napster has the single point of failure. The centralized server also is easy to be attacked by DoS.

All the other systems are better than Napster by using decentralized organization to eliminate the single point of failure. The additional mechanisms to achieve reliability are listed below:

Pastry: Routing in Pastry can be random, i.e., the choice among multiple nodes can be made randomly. In the event of a malicious or failed node along the path, the query may be repeated several times by the client, until a route is chosen that avoids the bad node.

Plaxton: Because routing only requires nodes match a certain suffix, there is potential to route around any single link or server failure by choosing another node with a similar suffix. However, Plaxton's reliability is limited by its single root failure because it is the node that every client relies on to provide an object's location information, and lack of ability to adapt because correlated access patterns to objects are not exploited, potential trouble spots are not corrected before they cause overload or cause congestion problems over the wide-area.

Tapestry: Using multiple root nodes gains redundancy and simultaneously makes it difficult to target a single node with a denial of service attack against a range of IDs. Redundancy is increased by supplementing the basic links with additional neighbor links. Further, the Tapestry infrastructure continually monitors and repairs neighbor links through a form of *introspection*, and servers slowly repeat the publishing process to repair pointers.

Chord: The good reliability is achieved by maintaining multiple data replicas and multiple successors. In the case of using *r=O(logN)* successors, even if every node fails with probability 1/2, with high probability the location algorithm can still find the closest living successor to the query key in expected time O(logN). CAN deals with data replication and node failure recovery very efficiently by replicating data on neighbor nodes.

CAN: It also maintains multiple data replicas in the neighbor. It also provides a "takeover" mechanism for fault recovery.

But the systems of TRAID, Pastry, Plaxton, Chord and CAN cannot tolerant malicious node, since they don't allow the application to define the selection operator.

4.4 Maintenance

For Napster and Gnutella, the nodes are organized loosely, so no work is necessary to making the system consistent.

In Plaxton the maintenance is very hard due to object's root mapping scheme and lack of support for node insertion, removing and failure. In order to achieve a unique mapping between document identifiers and root nodes, the Plaxton scheme requires global knowledge at the time that the Plaxton mesh is constructed to order the potential root nodes. The static nature of the Plaxton mesh means that insertions could only be handled by using global knowledge to recompute the function for mapping objects to root nodes.

Pastry, Chord, Tapestry and CAN all support dynamic node arrival and departure.

4.5 Usability

Napster supports the searching for music files. The central server can search its database and find a number of "optimal" files for user.

On the client's view, Gnutella provides similar function as Napster, but it supports different file types. On the server's view, Gnutella allows each node decide its own sharing mechanisms on different files.

Tapestry also has good usability due to its semantic flexibility. Tapestry allows the application to define the selection operator. Each object may include an optional application-specific metric in addition to a distance metric. Applications can then choose an operator to define how objects are chosen.

TRIAD is not a comprehensive system but only a solution to the problem of content based routing.

For Pastry, Plaxton, Chord and CAN, basically they only support looking up for a data given a key.

5. Future Work

Application Models Each of these systems proposes fundamental enhancements to the general problem of routing in peer-to-peer systems. What has rested outside of the scope of any of these projects is discussion of specific applications of these systems within particular domains. It makes sense that each system could be better at supporting a particular given domain than another. Furthermore, there may be a level at which facilities for specific application level needs can be embedded at system level, facilitating adaptation of a peer-to-peer locating and routing system to a particular application. Note that this idea is much in the same sense that although different systems may require different security levels a system typically provides protection mechanisms which act as support for different security policies.

We have isolated several different application types and needs specific to these applications:

- **Business Servers** These feature less dynamic data and topological structures. They have less concurrency and update requirements. Error and fail protections should be as strong possible. In general we want them to be: scalable and robust.
- **Consumer Based File Sharing** These are systems like Napster. These features less dynamic data but often require dynamic topological structures. Error and fail protection need not be perfectly guaranteed. In general we want them to be: scalable, efficient, and flexible.
- **Collaborative Applications** These feature dynamic data, but often less dynamic topologies. They require accurate and usually frequently updated data. Error and failure protections should have a high level of guarantee. In general we want them to be: robust, reliable, frequently updateable.
- **Commercial Applications** These systems are represented by e-commerce, or B-to-B (or C-to-C) business applications. These feature dynamic data, they possibly feature dynamic topologies. In general we want them to be: highly accurate, frequently and efficiently updateable.

An analysis of the need of different applications in this manner can be used in conjunction with our property/feature analysis to help decide upon an appropriate system to implement.

Security issues None of the routing algorithms provides satisfying security on data sharing. It is an important issue to combine proper security scheme on the routing algorithms. The security scheme should fit the requirements of the specific application and should not add much overhead on the routing algorithm.

Locating & Routing of Processing Resources Sharing computational resources is a related problem with similar locating and routing needs as data sharing in peer-to-peer systems, although the computing power request messages have more complex content such as control information, security information and code. Some systems which are

meant to be used within particular domains may not need to transmit code—code but be included on each server during set-up so that only control information would need to be transmitted and routing information which returns results back to specific clients.

Extensive Benchmark Comparisons A few of the surveyed systems provided data about performance but this does not properly address benchmarking for comparison to other peer-to-peer systems. The diverse nature of topologies and applications for peer-to-peer systems causes benchmarking to be a difficult task. The ideal situation would be to contrast each system on the same scale of nodes, but this is infeasible because the systems are meant to be scalable to include a large number of user/servers and also the topologies are often meant to be easily updateable—functionality which would additionally have to be benchmark tested. Another possibility is to establish node topologies whose complexities are proportional and generalizable so that results from different systems upon different topologies could be normalized and contrasted. In most likelihood this is infeasible as well. Probably the best situation is to implement a battery of systems on a moderate scale and include given data object on a version of each system. Base performance could be examined in this way, but still there are issues with varying degrees of data mobility affecting results. If we fix data then we may be crippling significant performance

advantages of one system over another. All of these concerns render benchmarking of peer-to-peer systems as an open research question.

Establishment of Criteria and Methodologies for Comparative Evaluation Beyond comparing benchmarks of different systems, a qualitative comparison between peer-to-peer systems is valuable and necessary when trying to decide which system is appropriate to a specific application. Beyond our analytical survey, *the key original contribution of this paper is the development of comparative criteria for evaluation of peer-to-peer systems*. Our criteria are large grain, but still have allowed us to make qualitative judgments on a general level regarding specific systems, even independently of considering specific applications. The analysis of desirable system properties in the light of the most important system features has proven fruitful. It has allowed us to specifically isolate uses of individual mechanisms in relationship to performance. This is important because merely having a mechanism does not guarantee that it is being used in a manner that fully exploits the advantages it is capable of providing, for example, data mobility can provide additional reliability be relocating data in case of topological changes, but it can also aid in the quite different problem of load balancing.

Simulation may also be a good technique for qualitative insight regarding the systems. We could test systems with very large numbers of nodes, as it would be unlikely that a researcher could actually test the system in such a real scenario. This simulation data could provide us with results such as: performance under scenarios involving high degrees of topological change, and high degrees of data movement. These types of results would be nearly impossible to qualitatively compare in a real system as individual systems may deviate greatly from initial allocation of data depending upon policies specific to the system.

A further extension would be to make this analysis more rigorous and continue the discussion in relation to specific applications. Systems could be ranked and classified depending upon which properties they offer strong support for and by trade-offs of system complexity and usability. Thus, when selecting a system one can choose the simplest system that is guaranteed to be adequate to her/his needs. Our criteria now allow for this type of decision to be made, but we believe that a further research problem outside of the scope of a survey paper would be to make these criteria more rigorous.

6. Summary

In the paper we address some issues on peer-to-peer data sharing. In the context of these issues we offer a survey of the following systems: Napster, Gnutella, TRIAD, Pastry, Plaxton, Tapestry, Chord, and CAN. The upshot is that we isolate what we see as both the crucial issues and solutions to the challenge of routing in the peer-to-peer environment, give a comparative summary for all systems. Finally based on framework and analysis, we propose some further questions and potential approaches for future investigation.

References

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

[2] Knowbuddy's Gnutella FAQ, *http://www.rixsoft.com/Knowbuddy/gnutellafaq.html*

[3] David Cheriton et. al. *http://www.dsg.stanford.edu/triad*, July 2000.

[4] A. Rowstron and P. Druschel, *Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems*. Accepted for Middleware, November 2001. http://research.microsoft.com/~antr/PAST/.

[5] C. Greg Plaxton, Rajmohan Rajaraman, Andréa W. Richa. *Accessing nearby copies of replicated objects in a distributed environment*. ACM Press New York, NY, USA Pages: 311-320 Series-Proceeding-Article. 1997. ISBN:0-89791-890-8.

[6] John Kubiatowicz, David Bindel, 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), November 2000.

[7] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. *Tapestry: An Infrastructure for Fault-tolerant Wide-*

area Location and Routing, U. C. Berkeley Technical Report UCB//CSD-01-1141, April 2000.

[8] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*, ACM SIGCOMM 2001, San Diego, CA, August 2001.

[9] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, Scott Shenker. *A Scalable Content-Addressable Network*. In Proceedings of the ACM SIGCOMM, 2001.

[10] O'Reilly P2P Directory.
http://www.openp2p.com/pub/q/p2p_category.

[11] Peer-to-Peer Working Group.
http://www.peer-to-peerwg.org/.