Hashing Spatial Content over Peer-to-Peer Networks

Aaron Harwood and Egemen Tanin
Department of Computer Science and Software Engineering
University of Melbourne, Victoria 3010, AUSTRALIA
(aharwood, egemen)@cs.mu.oz.au

Abstract—
The Internet has become the common medium for content distribution. Searching content using keywords is well-known. But there are many shortcomings to it. Users cannot search within the content and on many of the attributes of the content, i.e., other than its name. Content is also becoming increasingly decentralized. New mechanisms allowing access to complex distributed content is needed. We introduce a hashing-based method for accessing complex content over large dynamic networks such as peer-to-peer networks, which uses distributed hash tables in a novel way. In particular, we are interested in spatial content that is becoming popular in databases. Our method is scalable, addressing the needs of todays networks.

I. Introduction

The Internet has become the main medium of interactive content dissemination. Some widely accepted advantages that the Internet has over other mediums is its completely decentralized nature, scalability, and capacity for allowing independent subscriber interaction. However we believe the Internet users have just started to utilize these aspects of the Internet to their full potential. Content and interactions with it is becoming less reliant on centralized systems. Content distribution networks (CDN) are emerging in various forms, with the most general case being the peer-to-peer (P2P) networks [12]. A CDN maintains many data objects and any node in the network can service requests for the objects. Every object has a source and objects may be cached throughout the network to improve access efficiency. CDN nodes can cooperate to build data object indexes for searching and also to provide higher level management functions such as anonymous content publication. Yet, they do not provide a full solution for querying complex content.

Searching complete content in the form of files using keywords is common. But there are many shortcomings to this approach. First, you cannot search within the content (i.e., files). Second, you cannot use many of the attributes of the content for your search other than the content's name (i.e., file name). More elaborate mechanisms that allow users to access complex distributed content over the Internet is needed.

Current CDNs require some form of object indexing or a mechanism for locating objects in the network. Centralized indexes are relatively straight forward to implement but they lead to congestion as the rate of requests increases. Central index replication and query result caching may be used to offset this congestion. But they do not scale well. Recently developed hashing-based distributed indexes does address this problem [14], [16]. Hence, what is also required is a completely decentralized index that we can use to search and access complex content.

In this paper, we introduce a hashing-based method that can be used to access complex content over large scale distributed dynamic networks such as the P2P networks of today. In particular, we are interested in spatial content. Spatial references are becoming a common component in various databases. Cars with positioning systems, personal assistants, demographic data from government servers, even homepages of restaurants with addresses are a part of this emerging wave of spatial content.

In Section II we overview the state of the art methods to access distributed content over large networks. Then we list P2P networks in Section II-C. We discuss our solution to accessing distributed spatial content in Section III.

II. Hashing and distributed hash tables

Hashing is becoming increasingly popular for accessing distributed content over large networks. This section explains consistent hash algorithms in general terms and then shows how these algorithms are important for distributed systems.

A. Consistent hashing

A hash algorithm uses a hash function, H, that maps keys to locations, $H\colon \mathcal{K} \to \mathcal{L}$, where $\mathcal{K}$ is the set of all keys and $\mathcal{L}$ is the set of all locations. A key is usually a unique identifier that represents the data object to be stored, e.g., a file name is a key that represents the file contents. A hash algorithm requires close to constant time to find a data object, rather than for instance $\log_2 k$ time steps required for a binary search, where k is number of keys currently stored. This means that the time required to find an object is independent of the number of objects.

Hash functions can typically take any arbitrarily long binary string as a key and provide a many-to-one mapping to the locations as depicted in Fig. 1. The figure shows L locations depicted as buckets with data objects as shaded rectangles. The key for each object is a text string. When two or more objects hash to the same location then a secondary operation such as a second hashing function is applied or a search algorithm is used. The

We would like to thank Prof. Hanan Samet and Frank Morgan of the University of Maryland at College Park for their help and guidance during this work.

Author names are in alphabetical order.

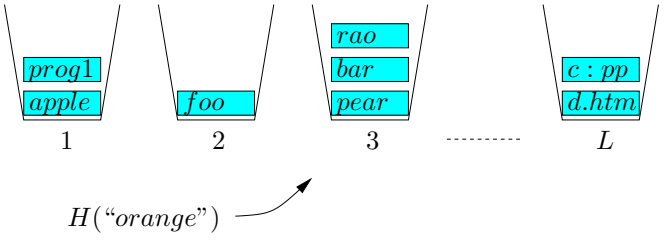
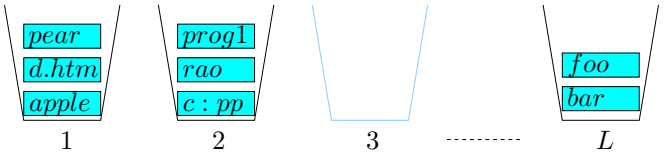


Fig. 1. A basic hash function.

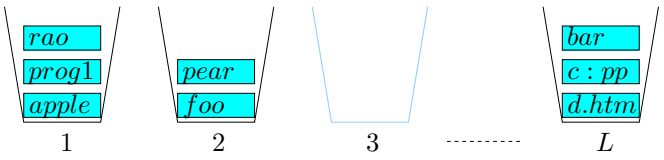
widely known SHA-1 algorithm can hash arbitrary strings onto $L = 2^{160}$ different locations.

Using $L = 2^{160}$ locations means that it is highly unlikely for any two objects to hash to the same location (so secondary operations are not usually required). Also it is important to note that a good hash algorithm like SHA-1 will provide a distribution of keys over buckets that is close to uniformly random. As a consequence, similar keys do not hash to similar locations. For example the location $H(\text{“}apple\text{”})$ is independent of the location $H(\text{“}apples\text{”})$.

An important class of hash functions are those that are *consistent*. A consistent hash function minimizes the degree of remapping required when the set of locations changes. Basically, the set of locations will change when either: (i) an existing location is removed or (ii) a new location becomes available. The concepts are explained using the illustrations in Fig. 2. In Fig. 2(a), an *inconsistent* hash function requires the objects from location 3 and *all the other objects* to be rehashed due to location 3 being removed. Comparing Fig. 2(a) and Fig. 1 note that inconsistent hashing has relocated many of the existing data objects due to the change. In Fig. 2(b) the removal of location 3 only leads to the relocation of the objects at location 3 – consistent hashing reduces the effect of a change. Similarly when a new location is made available, only a small number (proportional to the ratio of the number of items to the number of locations) of objects need relocating.



(a) An inconsistent hash function after remapping caused by the removal of location 3.



(b) A consistent hash function after remapping caused by the removal of location 3.

Fig. 2. Examples of hashing (in)consistency.

It is known that some hash algorithms provide optimum consistency, i.e., the degree of remapping is minimized. This class of hash algorithms are important building blocks of efficient content distribution networks and other systems that require distributed data and processing.

B. Distributed hashing

A distributed hash algorithm uses a number of servers to maintain some number of locations. A server represents a computer on the Internet, e.g., a `http` server. In the simplest sense, if there are M servers and L locations then each server should maintain roughly L/M locations. Although there are many methods for a distributed system to implement a distributed hash algorithm, we explain a method that has recently become widely known as the *Chord* method [16].

The Chord method maps both keys and servers to locations, as depicted more generally in Fig. 3. The figure shows a hash algorithm using 2^t different locations, from 0 to $2^t - 1$. Each server has a unique Internet Protocol (IP) address (and port number which is not shown) that is used plainly as a key.

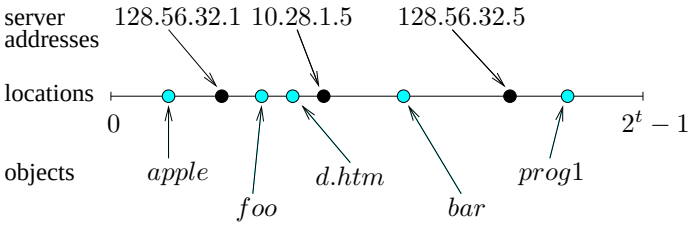


Fig. 3. Hashing server addresses and object keys to the same locations.

The Chord method derives its name from the use of modulo arithmetic and the clarity of representation achieved by depicting the hash locations in a circle. Fig. 4 depicts the locations using a circle and also shows other details of the Chord method.

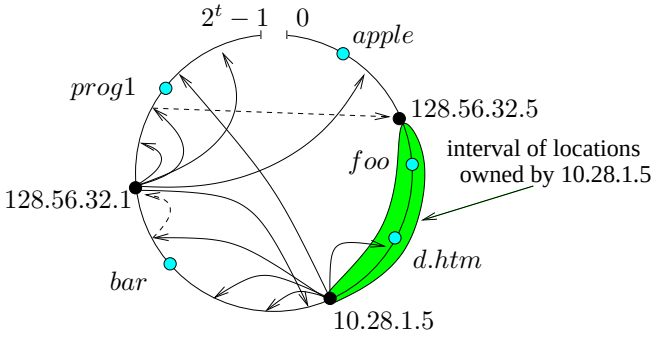


Fig. 4. Details of the Chord method.

Each server in the Chord maintains a table of up to t other servers, where t is logarithmically proportional to the number of locations. Each entry in the table represents an interval that is twice as large as the interval of the previous entry. The entries in the table for two servers are shown as arrows from the servers in Fig. 4. Each interval is associated with a successor server. These are shown as dashed arrows.

Without giving an explicit algorithm, consider the case when server 10.28.1.5 is trying to locate the object with key *prog*1. The server scans through the table and finds the interval which contains the key and the associated successor server. In this case *prog*1 is in the interval defined by the third and fourth arrows (counting clockwise from the server). The request for the object is then forwarded to server 128.56.32.1, which repeats the process and finally forwards the request to server 128.56.32.5. In general, it is proven that a request to locate an object must be forwarded up to $\log_2 n$ times, where n is the number of servers in the system.

The Chord method is consistent. As depicted in Fig. 5, the addition of a new server requires relocating only the objects on one existing server. The removal of a server requires relocating only the objects that were owned by that server. In general, servers can be added and removed

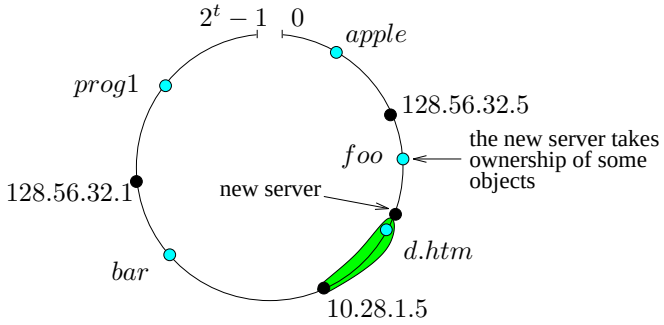


Fig. 5. Inclusion of a new server.

from the Chord arbitrarily since the overhead is logarithmically proportional to the number of servers. There are many more aspects to the Chord algorithm, including *caching* requests and objects for object locality and *virtual servers* for load balancing improvements. However in this paper we direct our discussion to a higher level system implementation perspective.

C. Peer-to-peer content distribution networks

Use of P2P systems on the Internet are gaining momentum. Napster [2] and Gnutella [1] were the first and most well-known of these systems. Napster was a P2P system where the directory service is centralized on servers and people exchanged music content that they had on their disks. Gnutella was a more general P2P content exchange system. Unfortunately, it suffered from scalability issues, i.e., floods of messages between peers to map connectivity in the system were required. Other systems followed these, each addressing a different flavor of sharing over the Internet. Multiple P2P storage systems have recently emerged. PAST [9], Freenet [6], Free Haven [8], Eternity Service [4], CFS [7], and OceanStore [11] are some popular P2P storage systems.

Some of these systems, like Freenet, have focused on anonymity while others, like PAST, have focused on persistence of storage. Also, other approaches, like SETI@Home [3] and distributed shells [17], made other resources, such as idle CPUs, work together over the Internet to solve large scale computational problems. Along with these systems, researchers have developed underlying routing and indexing utilities to address the core problems of these systems. These are for optimally finding and accessing an object in a decentralized dynamic distributed network over a wide-area. Examples of some lower level utility systems are [5], [13], [14], [15], [16], [18] including the Chord method. Recent research efforts continue to examine detailed aspects of these utilities [10].

Multiple P2P systems are built on top of the lower level utilities. For example, PAST is built on the Pastry [15] routing and indexing service. All of these routing and indexing utilities help the nodes of the systems find complete objects. We differ from these systems by supplying a method to query the content within an object and using other attributes of the content rather than just the name of it. We are specifically interested in the spatial attributes of the content.

III. Hashing distributed spatial content

Spatial databases contain objects with positions in space, e.g., cities that have a longitude and latitude coordinate. In a more general sense an object, G, is described by a region of some n-dimensional space. A spatial query is also defined as an object, Q, and the result of the query is all objects in the database that intersect with Q.

Unlike documents that are accessed by name (i.e., the query is simply a file name), spatial objects and spatial queries require intersection computations – the query is actually a *search* for objects that satisfy it. In $\mathbb{R}^2$ space, a query can be efficiently executed by first recursively subdividing the space into a quadtree and indexing the objects into the leaves of the tree[1]. Each query can similarly be subdivided to find which leaves it belongs to, which can exponentially reduces the average number of intersection calculations per query, for instance each level of the tree reduces the number of possible resulting objects by a factor of 4.

Distributed spatial queries require associating responsibility of spatial regions to the nodes in the system. If a node is responsible for a region of space then it is responsible for query computations that intersect that region of space. While it is possible to divide up space into a regular grid, this trivial division does not maintain the scalability of a quadtree.

We hash *control points*, e.g., $H(\text{“}(5,2)\text{”})$ is the location of $(5,2)$ on the Chord, to associate responsibility of a spatial region with a node in a P2P system. Each node is responsible for the regions of space surrounding the control points that hash to that node. We allow the control points to be dynamically determined using a globally known function to recursively subdivide space (or to superdivide as may be required when extending the current spatial domain). Fig. 6 depicts some control points and an example hashing. Objects are inserted into the distributed hash table by resolving them into spatial regions and hashing the control points of those regions. A copy of the object is stored at each node that becomes responsible for a control point. Alternatively a pointer to the original node that stores the object can be copied to each node, the data movements depend on the complexity of the object and the complexity of the intersection cal-

[1]We discuss $\mathbb{R}^2$ space throughout the remainder of this paper.

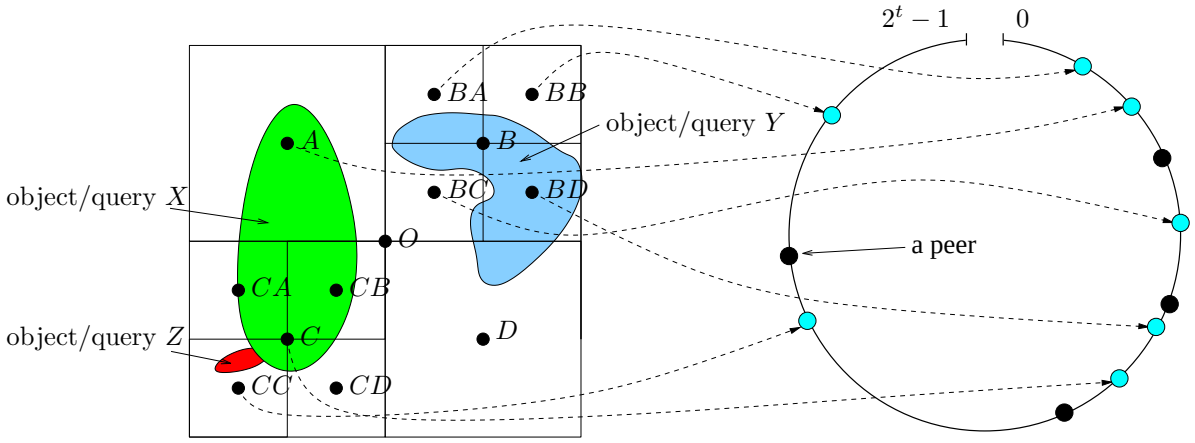


Fig. 6. Spatial objects/queries, X, Y, Z, 3 levels of control points, and example hashing to a distributed hash table. Z intersects X.

culations. Any node in the P2P network can have a small database of objects, contributing to the larger system.

Let X, Y and Z be objects or queries. Note that in Fig. 6, Y will actually be distributed to *many* nodes (some nodes may be responsible for more than one associated control point) while Z may only be stored at a single node. Parallelism is a direct consequence of having multiple control points and multiple nodes. All the nodes can handle the pieces of a query simultaneously.

Considering X as a query and Z as an object, resolving X to control points A and C will not be sufficient to detect the intersection of Z which is resolved to the single control point CC, i.e., the control points must match in order to give a correct query result. However, the level of resolution can be computed on a per object basis. We can use *downward* flags at higher level control points to indicate that some object does exist at a lower level. When inserting an object at a level i control point, a downward flag is hashed to the parent level $j = i - 1$ control point, which indicates the presence of an object at level i. This *flag hashing* is continued up through the quadtree until reaching a control point for which the downward flag already exists. Clearly, for a large number of objects, downward flags will exist for the origin point O and more than likely for the first few levels of control points (assuming a uniform distribution of objects).

Similarly, an *upward* flag is used to indicate that some objects are hashed at a higher level. In this case, an object insertion is also followed by propagating upward flags to the lower levels for which control points have already been hashed.

The use of flags allows a query to proceed from *any* level of the quadtree, dictated by the desired *initial* parallelism. However, as described, the method generally requires every query to propagate to the highest level for which control points have been hashed and to the lowest level in those regions where objects are known to exist. While the later case leads to the query propagating down to the leaves of the quadtree, the former case leads to queries propagating up to the root which suggests a bottleneck. This is overcome by requiring that no object is hashed to control points at a level less than some globally constant value called the *fundamental level*. For example, control points, O, A, B, C, and D in Fig. 6 may be forbidden to be used, which is using a fundamental level equal to 2.

In general, the number of nodes (parallelism) that store an object is selected when the object is stored and so can be controlled to a suitable value. The parallelism of a query is not so controllable because it depends on the location of existing objects and the levels to which these objects are hashed. However query parallelism is desirable and in general a high level of query parallelism is good. Redundant intersection computations can occur when an object intersects the query in more than one region. Eliminating this redundancy is a direction for further investigation.

A. Spatial algorithms

Let u be a control point, then let

$$D(u) = (upward, downward, list)$$

if control point u exists (i.e., has already been hashed) and $D(u)$ is empty if control point u does not exist. The default values for a $D(u)$ are $(0, 0, empty)$ which are used when an algorithm implicitly creates a new point u. Here, $upward, downward \in \{0, 1\}$ are flags and $list$ is a list of objects that intersect the region, $R(u) = (x_1, y_1, x_2, y_2)$, defined by $u = \big((x_2 - x_1)/2, (y_2 - y_1)/2\big)$ *and* are controlled by the owner of that control point. Each u has a parent control point, $P(u)$, and a set of four children, $\{C(u, 1), C(u, 2), C(u, 3), C(u, 4)\}$.

Given a control point u, it is always possible for any node to determine $R(u)$. This requires an *origin region*, $R(o) = (0, 0, 1, 1)$ (the unit square) with control point $o = (0.5, 0.5)$ and a method of sub/super-division that covers all space. Call o a (or the) 0-level control point. Let $L(u)$ be the level of control point u. In Fig. 6, $L(O) = 0$ and object Y has been resolved into 4 regions at level 2 plus 1 region at level 1.

The procedures `InsertObject()` and `SeedQuery()` are used to insert objects into the system and to make queries for objects in the system. The other procedures are recursive and called as appropriate. Let $D(o) = (0, 0, empty)$ be the initial starting control point, i.e., the control point exists but there are no objects yet. Other control points do not exist.

```
procedure FilterDown(control point u)
  for i := 1 to 4
      v := C(u,i)
      if D(v) exists then
         set D(v) upward to 1
         FilterDown(v)
done

procedure Resolve(object X,
                  control point u,
                  reslevel r)
  if X intersects R(u) then
     if L(u) = r then
        set D(u) list to include X
        for i := 1 to 4
            FilterDown(C(u,i))
     else
       set D(u) downward to 1
       for i := 1 to 4
           Resolve(X,C(u,i),r)
done

procedure InsertObject(object X)
  select an r to give reasonable parallelism
  t := o
  while X is not contained within R(t)
        t := P(t); set D(t) downward to 1
  Resolve(X,t,r)
done

procedure Query(query X,
                control point u,
                direction d)
  if X intersects R(u) then
     intersect all objects in D(u)
     record intersecting objects in result
  if direction is down
     if D(u) downward is 1
        for i := 1 to 4
            Query(X,C(u,i),down)
  else
    if D(u) upward is 1
       Query(X,C(u,i),up)
done

procedure SeedQuery(query X)
  select an r to give reasonable parallelism
  for each intersecting R(u) at level r
      Query(X,u,down)
      Query(X,u,up)
done
```

IV. Conclusion and future work

With the recent paradigm shifts, content and interactions with the content is becoming less reliant on centralized systems. CDNs are emerging in various new forms such as the P2P networks. CDN nodes can cooperate to build data object indexes for searching content. Searching complete content in the form of files using keywords is common. There are many shortcomings to this approach. More effective mechanisms that allow users to access complex distributed content over the Internet are needed. In this paper, we introduced a hashing-based method that can be used to access complex content over large scale distributed dynamic networks such as the P2P networks of today. In particular, we are interested in spatial content, however we believe our method can be extended to arbitrary spaces. We adopted the Chord approach for our solution to querying such complex content because Chord is an efficient distributed hashing algorithm. We believe our approach addresses the current dynamic and distributed nature of the new networks. We are in the process of refining our algorithms and we plan to test our approach on a prototype P2P system in near future.

References

[1] Gnutella. http://www.gnutella.com/.

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

[3] SETI@Home. http://setiathome.ssl.berkeley.edu/.

[4] R. Anderson. The Eternity Service. In *Proceedings of the PRAGOCRYPT'96*, pages 242–252, Prague, Czech Republic, September 1996.

[5] S. Bhattacharjee, P. Keleher, and B. Silaghi. The design of TerraDir. Technical Report CS-TR-4299, Department of Computer Science, University of Maryland at College Park, October 2001.

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

[7] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the ACM SOSP'01*, pages 202–215, Banff, AL, October 2001.

[8] R. Dingledine, M. J. Freedman, and D. Molnar. The Free Haven Project: Distributed anonymous storage service. In *Proceedings of the ICSI Workshop on Design Issues in Anonymity and Unobservability*, pages 67–95, Berkeley, CA, July 2000.

[9] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of the IEEE HotOS VIII Workshop*, pages 65–70, Schloss Elmau, Germany, May 2001.

[10] A. Harwood and M. Truong. Multi-space distributed hash tables for multiple transport domains. In *Proceedings of IEEE International Conference on Networks (to appear)*, Sydney, Australia, 2003.

[11] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent store. In *Proceedings of the ACM ASPLOS'00*, pages 190–201, Cambridge, MA, November 2000.

[12] A. Oram. *Peer-to-peer: Harnessing the power of disruptive technologies*. O'Reilly and Associates, Sebastopol, CA, 2001.

[13] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. *Theory of Computing Systems*, 32(3):241–280, May 1999.

[14] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM'01*, pages 161–172, San Diego, CA, August 2001.

[15] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the ACM Middleware'01*, pages 329–350, Heidelberg, Germany, November 2001.

[16] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the ACM SIGCOMM'01*, pages 149–160, San Diego, CA, August 2001.

[17] M. Truong and A. Harwood. Distributed shell over peer-to-peer networks. In *Proceeding of the 2003 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 269–275, Las Vegas, Nevada, 2003.

[18] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB-CSD-01-1141, Department of Computer Science, University of California, Berkeley, April 2001.