Zero Servers With Zero Broadcasts

Miguel Castro
Microsoft Research
Cambridge, UK
mcastro@microsoft.com

Greg O'Shea
Microsoft Research
Cambridge, UK
gregos@microsoft.com

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

ABSTRACT

To achieve the vision of networks that work without any supporting infrastructure, we need wireless ad hoc technology to replace the cabling infrastructure, but we also need self-configuring network and application services to replace the server infrastructure. Current solutions perform poorly because they either pick a single host to act as the server or they use network wide broadcasts to implement services. We need wireless ad hoc networks with *zero servers* and *zero broadcasts*!

Can we use DHTs to build both network- and application-level services with zero servers and zero broadcasts? This paper starts to answer this question. It shows that it is important to remove broadcasts at all levels of the networking stack and describes how to use the Virtual Ring Routing protocol to achieve our vision.

Categories and Subject Descriptors

C.2 [**Computer Systems Organization**]: Computer Communication Networks

General Terms

Algorithms

Keywords

Wireless ad hoc networks, Distributed Hash Tables

1. INTRODUCTION

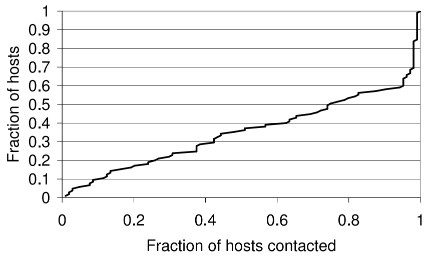
It would be great if networks could work without any supporting infrastructure. This would allow computers to communicate and access services when it is not feasible or convenient to deploy a supporting infrastructure, or when an existing infrastructure fails. Removing the need for deploying and maintaining a supporting infrastructure may also reduce hardware and management costs.

Wireless ad hoc networks provide an important step towards this vision. They enable hosts to communicate without wires, routers, or access points. However, they still rely on servers not only to provide network services, for example DHCP [9] and DNS [16], but also to provide application services, e.g. file systems and instant messaging. We are interested in providing network and application services in small scale scenarios, for example in the home or in a meeting, and in large-scale scenarios. For example, in an office building or office floor, where an office building's wired network is completely replaced by a wireless mesh network [10]. Such networks will need to scale to a couple of hundred hosts.

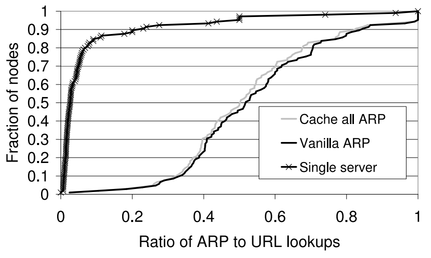
There has been some work towards configuring a network automatically without requiring servers. The IETF Zeroconf working group has defined protocols [2, 4, 3, 12] that enable small networks to self-configure and perform service discovery without DHCP and DNS servers. However, these protocols make extensive use of broadcasts, for example, hosts broadcast ARP [20] packets to allocate IP addresses [2] and they broadcast DNS requests [4] to locate the host responsible for a DNS record. In larger-scale networks, e.g. office mesh networks, the use of broadcasts reduces the bandwidth available to other applications as the network size increases [17].

Additionally, the Zeroconf work does not address the problem of providing application services without servers. Simply assigning a server role to a host in the network does not address the problem. Performance will be poor because, as the network grows, the host is unlikely to have the communication and computation resources required to play the server role. Also, if the hosts to perform server roles are selected arbitrarily then it is necessary to make the service resilient to host failure.

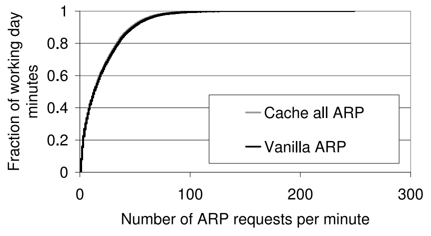
We propose using Distributed Hash Table (DHT) technology [25, 21, 27, 23] to implement both network- and application-level services without using any servers or any broadcasts for wireless networks. DHTs can be used to implement peer-to-peer versions of both network- and application level services. They balance the load of managing hash-table keys across all the hosts in the network and they route messages sent to a key to the host responsible for managing the key. For example, one could implement DNS by hashing the domain name associated with a DNS record to obtain a key and publishing the record by sending a message to the key. Other hosts could then lookup the record by sending a request to the key obtained by hashing the domain name. This implementation balances the load to serve DNS requests across all the nodes in the network and it does



(a) CDF of the fraction of hosts contacted by each host.



(b) CDF of the ratio of ARP requests to URL lookups per host.



(c) CDF of the number of ARP requests per one-minute window.

Figure 1: Traffic changes with serverless web caching.

not use broadcasts.

DHTs have traditionally been implemented as overlays on top of existing network routing protocols but this approach can perform poorly in wireless ad hoc networks. The problem is that moving from a server-based implementation to a DHT-based implementation can significantly change the traffic patterns in the network: it increases the number of flows to distinct destinations and shortens their lifetime. This can impact the underlying network protocols, for example, it causes reactive routing protocols [14, 19] and the ARP protocol to perform poorly because of a large increase in the number of network-wide broadcasts. We present results of experiments driven by real traces that shows one ARP request is generated for every two pages requested when using a DHT-based Web cache [13].

We propose using Virtual Ring Routing (VRR) [1] to implement services. VRR implements both traditional point-to-point routing and DHT functionality with no broadcasts. Unlike reactive protocols, it performs well with a DHT workload and, unlike proactive protocols [18, 5], it performs well with mobility [1]. VRR offers an API to write DHT applications at user level. We used this API to implement ARP and we present some encouraging preliminary performance results. Our approach can remove the need for servers and it can scale to larger networks than previous approaches because it eliminates broadcasts.

The rest of the paper is organized as follows. Section 2 examines how traffic patterns change when servers are replaced by DHT-based service implementations. Section 3.1 provides an overview of VRR and Section 3.2 describes the DHT functionality it provides. Section 3.3 describes an ARP implementation that uses VRR's DHT functionality to eliminate broadcasts and Section 3.4 presents some preliminary performance results. We conclude in Section 4.

2. TRAFFIC CHANGES WITH NO SERVERS

Moving from a client/server implementation to a DHT implementation of a service changes traffic patterns in the network. These changes can invalidate assumptions that the underlying network protocols rely on for performance. We ran experiments with an example service to illustrate this problem but the problem applies to any service.

We simulated, using a discrete event simulator, a DHT-based Web caching service [13] to illustrate how traffic patterns change. In particular we evaluated the impact of these changes on the performance of the Address Resolution Protocol (ARP). These changes would have a similar impact on other protocols, for example, on reactive routing protocols [1]. The simulation was driven by a trace of requests received by the centralized Web proxy server in our Microsoft Research Cambridge building during the period 9/7/2001 to 8/8/2001. The trace contains entries for 105 unique IP addresses, which we assume are 105 unique nodes, and represents an example workload for a service running on an office wireless mesh.

In the peer-to-peer Web caching service, each node runs a local Web cache proxy. When a node x requests a URL, the local proxy hashes the URL to obtain a key and uses the DHT to send the request to the node responsible for the key, which we call the *root node* for the key. When the root node receives the request, it checks its local cache for a page with the requested URL. If it has a valid copy, it returns the page to x. Otherwise, it retrieves the page from the origin web server, adds the page to the cache, and returns it to x.

Initially, for each node we measured the fraction of the nodes in the network that were the root node for at least one URL requested by the node. Figure 1(a) shows the cumulative distribution function for these measurements. As expected, the number of nodes contacted by each node is high: the median is 75 nodes as opposed to one in the client/server implementation.

This change in traffic patterns impacts the performance of the ARP protocol, which is used by IPv4 to discover the mapping between IPv4 addresses and physical hardware addresses. In ARP, each node maintains a cache of IP to physical address mappings. When a node uses an IP address of another node on the same link, ARP checks the cache. If no mapping is found, the ARP protocol is used to discover the mapping. Normally this is achieved by flooding the local link with an ARP request. If a node receives an ARP request containing its IP address, it responds directly to the source of the ARP request and the source adds the mapping to its ARP cache. In Windows, the default policy is to timeout ARP cache entries after 10 minutes or if unused for 2 minutes.

Using the simulator with the same trace, we measured the impact of the DHT-based Web cache on ARP. We modeled the ARP cache on each host, and experimented with two different caching policies for ARP: *vanilla ARP*, which complies with the current standards, and *cache all ARP*, which is optimized to improve performance when ARP requests are flooded. In *vanilla ARP* when host x receives an ARP request from any host y and x has a valid entry for y in its cache, x resets the expiration timers for the entry. In *cache all ARP*, whenever x receives an ARP request from y, x inserts an entry for y in its ARP cache. Both policies used the default ARP cache timeouts in Windows with infinite cache sizes.

The first experiment measured the ratio of ARP requests to URL requests issued by each node. Figure 1(b) shows a cumulative distribution function of these measurements for the DHT-based Web cache with the two ARP caching policies, and for the client/server Web cache with a single server. The DHT-based Web cache implementation performs similarly with both ARP cache policies; the median for both is approximately one ARP request for every two URL requests. We also measured the number of ARP requests issued per unit time across the network. Figure 1(c) shows a cumulative distribution function of the number of ARP requests issued in one minute windows for minutes between 08:00 and 18:00 (that is during working days). The performance is similar with both ARP cache policies: the median is 14 ARP requests per minute and the 95th percentile is one ARP request per second.

The results show that changing service implementations from a client/server model to a DHT model can have a dramatic impact on traffic and protocol usage patterns. This change is necessary to support services without servers because it balances load across all the nodes in the network. However, it can lead to very poor performance by increasing the number of broadcasts performed by underlying network protocols.

To achieve the vision of a network without any supporting infrastructure, we need to build both network- and application-level services in a wireless ad hoc network without broadcasts and without overloading any node in the network. The next section describes how we can use VRR [1] to do this.

3. ZERO BROADCAST SERVICES

VRR implements both traditional point-to-point network routing and DHT functionality in a wireless ad hoc network without using any broadcasts. We propose to implement both application and network-level services using VRR.

There has been a significant amount of work on developing DHT versions of application-level services for the Internet, for example, email and instant messaging services [15], Web caching [13], file backups [24, 6], and version control systems [26]. We plan to adapt these applications to run on a wireless ad hoc environment using VRR.

There has been relatively little work on implementing network-level services and protocols using DHT functionality. We describe how we implemented ARP using VRR to eliminate the performance problem that we demonstrated in the previous section. We plan to implement DHCP, DNS and Service Discovery using a similar approach to avoid the broadcasts in dynamic configuration of IP addresses [2], multicast DNS [4], SSDP [12] and DNS-SD [3].

This section starts with a brief overview of VRR followed by a description of the DHT functionality it offers. Then it presents our ARP protocol without broadcasts and some preliminary performance results.

3.1 VRR

VRR uses random unsigned integers to identify nodes, and organizes the nodes into a *virtual ring* in order of increasing identifier (with wrapping around zero). Node identifiers are fixed, unique, and location independent. To maintain the integrity of the virtual ring with node and link failures, each node maintains a virtual neighbor set (or *vset*) of cardinality r containing the node identifiers of the $r/2$ closest neighbors clockwise in the virtual ring and the $r/2$ closest neighbors counter clockwise. Each node also maintains a physical neighbor set (or *pset*) with the identifiers of nodes that it can send messages to and receive messages from at the link layer.

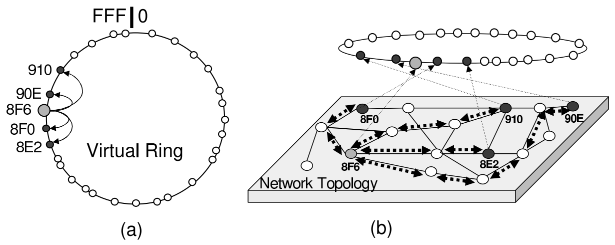


Figure 2: Relationship between the virtual ring and the physical network topology.

Figure 2(a) shows an example virtual ring with a 12-bit identifier space (with identifiers in base 16). It also shows the vset of the node with identifier $8F6$ with $r = 4$.

VRR sets up and maintains routing paths between a node and each of its virtual neighbors. These are called *vset-paths*. Since node identifiers are random and location independent, the virtual neighbors of a node will be randomly distributed across the physical network. Therefore, vset-paths are multi-hop in most cases. They are also bidirectional because membership in the vset is symmetrical (if node x is in the vset of node y then node y is in the vset of x).

The routing information for a vset-path is stored in the *routing tables* of the nodes along the path. Each node maintains a routing table with information about the vset-paths to its virtual neighbors and other vset-paths that are routed through the node. A routing table entry identifies the two vset-path endpoints and the next hop towards each endpoint. This information is maintained proactively, i.e., it is maintained even when there is no traffic along the vset-path.

Figure 2(b) shows the mapping between the virtual ring and the physical network topology and it shows the vset-paths between node $8F6$ and its virtual neighbors.

VRR does not setup or maintain paths between nodes that are not virtual neighbors because vset-paths can be used to route packets between any pair of nodes in the network. VRR nodes route packets to destination identifiers by forwarding them to the next hop towards the path endpoint whose identifier is numerically closest to the destination identifier from among all the endpoints in their routing table.

If there is a correct vset-path between each node and its virtual neighbors, VRR can route between any pair of nodes by following the vset-paths between neighboring nodes along the ring. But VRR does better because each node uses not only the vset-paths to its virtual neighbors but also vset-paths between other nodes that happen to be routed through it. The following approximate analysis provides some intuition into how this works. If each node maintains r vset-paths to its virtual neighbors and the average path length is p, the total number of routing table entries in an n node network is nrp. Therefore, each node will have on average rp entries for vset-paths in its routing table: r entries for the paths to its virtual neighbors and $r(p-1)$ additional entries for vset-paths through the node. If we assume that these

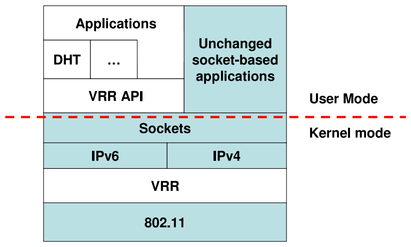


Figure 3: Software architecture.

additional vset-paths end at nodes that are selected randomly and uniformly, the probability that a random node has a path to a random destination is $O(rp/n)$. Therefore, a packet is expected to reach a node that has a vset-path to the destination after visiting $O(n/(rp))$ nodes, which will add only a constant stretch if p grows with $\sqrt{n}$ (as in wireless ad hoc networks).

VRR does not use flooding at all and it uses only location independent identifiers to route. All control and data packets are routed as described above without any translation to location based addresses. In particular, control messages to setup new vset-paths are routed using the existing vset-paths. A detailed description of VRR can be found in [1].

The version of VRR used in this paper is implemented using the Mesh Connectivity Layer (MCL) toolkit from Microsoft Research [8]. MCL adds a new kernel module that appears as a virtual network adapter to the Windows TCP/IP stack, which allows the use of unmodified IP-based protocols and applications over VRR. This implementation uses 48-bit virtual MAC addresses as node identifiers.

3.2 DHT functionality

VRR provides not only point-to-point network routing between two nodes but also a distributed hash table (DHT) [25, 21, 27, 23]. VRR routes messages sent to a numerical key to the node whose identifier is numerically closest to the key. This is the node responsible for managing information associated with the key. The key management load can be balanced across all the nodes in the network by selecting both node identifiers and keys uniformly at random from the identifier space.

VRR is designed to provide consistent routing to the node responsible for a key with high probability even with node mobility and failures. It uses local failure detection and a guaranteed failure notification mechanism to detect node and link failures quickly and with low overhead [1]. Additionally, VRR does not need to transfer hash table data across nodes when they move because node identifiers are fixed and independent of the topology. In contrast, proposals to implement DHTs using coordinate-based routing [22, 11] require data shuffling when nodes move.

VRR is implemented as a device driver in the kernel but it exposes DHT functionality to user-level applications to simplify development. This is done by tunneling UDP messages between the driver and an application through the unmodified IP stack and socket layers (as shown in Figure 3). Applications send DHT messages to a reserved IP address that is the same across all VRR machines. These messages include the destination key in the UDP payload. The local driver intercepts these messages and routes them to the node responsible for the destination key, which is the node whose identifier is numerically closest to the most significant 48 bits in the key. The driver in the node responsible for the key sets the destination address in the message to another reserved IP address that is used by applications to receive DHT messages. Then, it sends the message up the IP stack to be received by the DHT application responsible for the destination key. Additionally, applications can query the local driver to obtain relevant information, for example, the set of identifiers in the node's vset. Applications also receive notification messages when the vset changes. This interface is very similar to the key-based routing API described in [7].

We implemented a simple C# layer that provides a DHT interface on top of the low level key-based routing API. The DHT API is shown in Figure 4. The constructor takes a reference to a `VRRInterface` object that implements the key-based routing API and a `VSet` object that implements the interface to obtain information about the vset from the driver. The DHT class replicates key/value pairs in the nodes in the vset of the node responsible for the key. The identifier supplied to the constructor allows multiple DHT applications in the network. The `Put` methods allow applications to store key/value pairs in the DHT. The user can specify a callback to invoke if a `Put` fails. The `Get` method retrieves the value associated with the key supplied as argument from the DHT. The argument `callback` is invoked with the reply or a failure notification.

3.3 Zero broadcast ARP

In this section we describe, as an illustrative example of a network service implemented using a DHT, the ARP protocol implemented as a user-level DHT-based program running on the VRR nodes. In general, routing protocols implemented at layer 2.5 [8] (like MCL and VRR), make the wireless network appear as a single Ethernet network to the higher layers. Conventionally, ARP is implemented using network broadcasts in these layer 2.5 implementations. In contrast our ARP implementation uses a DHT to store the mappings between IP addresses and 48-bit VRR addresses.

Figure 5 shows the source code for the `ARPService` class, which implements the DHT-based ARP protocol. We omit some of the support functions from the code for clarity. Each VRR node in the network creates an instance of `ARPService` when it is initialized. The constructor receives a `VRRInterface` object through which it communicates with the VRR driver, a `VSet` object and an `Intercept` object. It creates a `DHT` object and registers with the `Intercept` service. `Intercept` allows all packets from a particular protocol (ARP in this case) to be intercepted by the driver and forwarded to the specified application at user-level. The constructor also uses `dht.Put` to insert a mapping between the hash of the IP address bound to the local VRR interface and the virtual MAC address of the local VRR interface. `PutFailure` is called if `dht.Put` is not completed successfully and it simply re-issues the `dht.Put`.

Whenever an ARP request is generated by the IP stack, the VRR driver intercepts the packet and it is tunneled to the user-level where the `ReceivedRaw` method is invoked. The `ReceivedRaw` method extracts the IP address being resolved from the packet, hashes the address to obtain a DHT key, and performs `dht.Get` to get the MAC address associated with the key. When the `dht.Get` completes, `GetResponse` is invoked by the DHT implementation with the key, the corresponding value, and the original ARP packet as arguments. It generates an ARP response using the original ARP packet and the returned value. This response is passed down

```
public interface DHTGetResponse { // called when a response to a Get is received
  void GetResponse(Key key, byte[] value, object cargs);
}
public interface DHTPutResponse { // called when a Put fails
  void PutFailure(Key key, byte[] value, object cargs);
}

public class DHT : Application, VSetInfo {
  // create a new DHT application on interface vrr with identifier id
  public DHT(VRRInterface vrr, VSet vset, int repFactor, byte id) {...}
  // Put value in the DHT under key
  public void Put(Key key, byte[] value) {...}
  // Put value in the DHT under key and request a callback in failure with cargs
  public void Put(Key key, byte[] value, DHTPutResponse callback, object cargs) {...}
  // Get value under key from the DHT; the callback is invoked with the response and cargs
  public void Get(Key key, DHTGetResponse callback, object cargs) {...}
}
```

Figure 4: The DHT API that exposes VRR's DHT functionality to user-level applications.

```
public class ARPservice : Application, DHTGetResponse, DHTPutResponse
{
    private VRRInterface vrr;
    private DHT dht;

    public ARPservice(VRRInterface vrr, Intercept intercept, VSet vset)
    {
        this.vrr = vrr;
        this.dht = new DHT(vrr, vset, Parameters.ARP_DHT); // Create a new DHT instance for the ARP Service
        intercept.RegisterApplications(Protocols.ARP, this); // Register that all ARP protocol packets should be intercepted
        dht.Put(SHA1(vrr.GetIPv4Address()), vset.myVRRAddress(), this, null); // Insert IPv4 to VRR Address mapping in DHT
    }

    // All ARP requests issued by the driver are delivered via this callback
    public void ReceivedRaw(byte[] payload) {
        dht.Get(Key.Hash(ExtractIPv4AddressARPRequest(payload)), payload, this);
    }

    public void GetResponse(Key key, byte[] value, object cargs) {
        if (value == null) return; // Not found in DHT
        // Send the ARP response to the VRR driver - specifying LOOPBACK
        vrr.Transmit(new Header(null,LOOPBACK), ARPReplyPacket((byte []) cargs, value));
     }

    // When the DHT fails to insert the key this method is called
    public void PutFailure(Key key, byte[] value, object cargs) {
        dht.Put(key, value, this, cargs);
    }
}
```

Figure 5: ARP service implementation using the VRR user level APIs. The function `vrr.GetIPv4Address` gets the IP address bound to the VRR interface, `ExtractIPv4AddressARPRequest` extracts the IP address that is being resolved from an ARP request packet. The function `ARPReplyPacket` takes an ARP request packet and a VRR address and creates an ARP response packet.

to the VRR driver using the `vrr.Transmit` method specifying that the packet should be looped back, i.e., sent by the driver back up the IP stack to update the ARP cache.

This example demonstrates how easy it is to implement DHT-based versions of network-level services using the VRR driver and user-level libraries. Since our implementation of ARP replaces network-wide broadcasts by a single DHT lookup, it avoids the performance problems we discussed in the previous section.

3.4 ARP evaluation

We deployed the DHT-based ARP implementation on a 41 machine 802.11a testbed. The machines run Windows XP and are distributed across two floors of our office building (Figure 6). Most machines are placed in offices and a small number in cubicles in open-plan areas. Each machine has a NetGear WAG 311 wireless network card, and the diameter of the network is 5. VRR is configured with a vset size of four ($r = 4$), and a hello period of two seconds.

We ran an experiment to compare the performance of the DHT-based ARP implementation with a broadcast implementation of ARP used in the MCL. Each machine pinged all other machines starting from an empty ARP cache. Then, it pinged all other machines again (with a populated ARP cache). We repeated this process five times and computed the average ping times in both cases. The difference between these times is equal to the average delay to resolve ARP requests. Figure 7 shows these delays averaged across all machine pairs for both implementations. The resolution latency is low in both cases. The broadcast implementa-

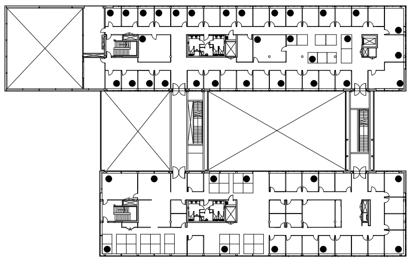


Figure 6: Floor plan of the 41 machine 802.11a PC testbed.

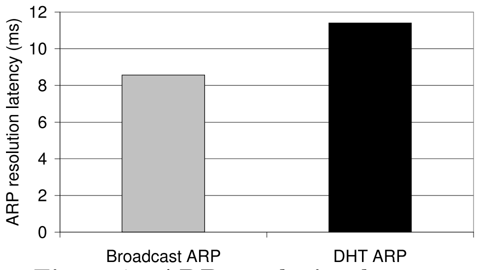


Figure 7: ARP resolution latency.

tion performs better because the network is not loaded, but the number of messages per ARP resolution is only 4.78 in the DHT implementation and it is more than 41 messages in the broadcast implementation. The performance of the DHT-based ARP implementation could also be improved by moving it into the kernel.

4. CONCLUSION AND FUTURE WORK

This paper describes our initial work on developing efficient zero Infrastructure scalable wireless networks. We show how to implement application- and network-level services with zero servers and zero broadcasts using VRR. We also show that moving from server-based to DHT-based service implementations can significantly change the traffic patterns in the network. These changes can cause network protocol implementations that rely on network-wide broadcasts to perform poorly. We present an example that illustrates a general solution to this problem: an implementation of the ARP protocol that uses a DHT to eliminate broadcasts.

We are currently building an entire set of network-level services, such as DNS and DHCP, using VRR. We are also building and evaluating a number of application-level DHT-based services that have been developed for the Internet to understand how to efficiently implement them in wireless ad hoc networks.

5. REFERENCES

[1] M. Caesar, M. Castro, E. Nightingale, G. O'Shea, and A. Rowstron. Virutal Ring Routing: Network routing inspired by DHTs. In *Sigcomm'06*, 2006.

[2] S. Cheshire, B. Aboba, and E. Guttman. Dynamic Configuration of IPv4 Link-Local Addresses (RFC 3927), May 2005. `http://ietf.org/rfc/rfc3927.txt`.

[3] S. Cheshire and M. Krochmal. DNS-Based Service Discovery (Internet Draft), June 2005.

[4] S. Cheshire and M. Krochmal. Multicast DNS (Internet Draft), June 2005.

[5] T. Clausen and P. Jacquet. OLSR RFC3626, Oct. 2003. `http://ietf.org/rfc/rfc3626.txt`.

[6] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. ACM SOSP'01*, Banff, Canada, Oct. 2001.

[7] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a common API for structured P2P overlays. In *IPTPS*, Feb 2003.

[8] R. Draves, J. Padhye, and B. Zill. Comparison of routing metrics for static multi-hop wireless networks. In *SIGCOMM'04*, Aug. 2004.

[9] R. Droms. Dynamic Host Configuration (RFC 2131), Mar. 1997. `http://ietf.org/rfc/rfc2131.txt`.

[10] J. Eriksson, S. Agarwal, P. Bahl, and J. Padhye. Feasibility study of mesh networks for all-wireless offices. In *Mobisys*, June 2006.

[11] R. Fonseca, S. Ratnasamy, J. Zhao, C. Ee, D. Culler, S. Shenker, and I. Stoica. Beacon vector routing: Scalable point-to-point in wireless sensornets. In *NSDI'05*, May 2005.

[12] Y. Goland, T. Cai, P. Leach, Y. Gu, and S. Albright. Simple Service Discovery Protocol (Internet Draft), Oct. 1999.

[13] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized peer-to-peer web cache. In *PODC*, July 2002.

[14] D. Johnson and D. Maltz. Dynamic source routing in ad hoc wireless networks. In *Ad Hoc Networking*, 2001. Chapter 5, pg 139-172, Addison-Wesley.

[15] A. Mislove, A. Post, C. Reis, P. Willmann, P. Druschel, D. Wallach, X. Bonnaire, P. Sens, J.-M. Busca, and L. Arantes-Bezerra. POST: A secure, resilient, cooperative messaging system. In *HotOs*, May 2003.

[16] P. Mockapetris. Domain Names - Implementation and Specification (RFC 1035), Nov. 1987. `http://ietf.org/rfc/rfc1035.txt`.

[17] S. Ni, Y. Tseng, Y. Chen, and J. Sheu. The broadcast storm problem in a mobile ad hoc network. In *MOBICOM*, Aug. 1999.

[18] C. Perkins and P. Bhagwat. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In *SIGCOMM'94*, Aug. 1994.

[19] C. Perkins and E. Royer. Ad hoc on-demand distance vector routing. In *Mobile Computing Systems and Applications*, Feb. 1999.

[20] D. Plummer. Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware (RFC 826), Nov. 1982.

[21] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM'01*, Aug. 2001.

[22] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker. GHT: A geographic hash table for data-centric storage. In *WSNA'02*, Sept. 2002.

[23] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware*, Nov. 2001.

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

[25] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM'01*, Aug. 2001.

[26] A. Yip, B. Chen, and R. Morris. Pastwatch: A distributed version control system. In *NSDI*, May 2006.

[27] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: an infrastructure for fault-resilient wide-area location and routing. In *Technical report UCB//CSD-01-1141, U.C. Berkeley*, Apr. 2001.