Distributed Hash Queues: Architecture & Design

Chad Yoshikawa[1], Brent Chun[2], and Amin Vahdat[3]

[1] University of Cincinnati, Cincinnati OH 45221, USA, yoshikco@ececs.uc.edu
[2] Intel Research Berkeley, Berkeley CA 94704
[3] University of California, San Diego, La Jolla, CA 92093-0114

Abstract. We introduce a new distributed data structure, the Distributed-Hash Queue, which enables communication between Network-Address Translated (NATed) peers in a P2P network. DHQs are an extension of distributed hash tables (DHTs) which allow for *push* and *pop* operators vs. the traditional DHT *put* and *get* operators. We describe the architecture in detail and show how it can be used to build a delay-tolerant network for use in P2P applications such as delayed-messaging. We have developed an initial prototype implementation of the DHQ which runs on PlanetLab using the Pastry key-based routing protocol.

1 Introduction

Delay-Tolerant Networks (DTN) [6] are network overlays that enable communication even in the face of arbitrary delays or disconnections. This is accomplished by using a store-and-forward mechanism which holds packets at interior nodes until forwarding to the next hop in the route is possible. Unlike IP, there is no assumption of an instantaneous source-to-desination routing path nor are there limitations placed on latency or packet loss. In essence, arbitrary delays along the routing path are tolerated by incorporating storage and retransmission in the network itself.

DTNs are useful for enabling messaging over so-called *challenged networks* [5] which have inhernet network deficiences that prohibit communication using standard IP. Examples of challenged networks include satellite-based communication, sensor networks, and ad-hoc mobile networks.

Unfortunately, challenged networks need not be so exotic. Current trends indicate that the Internet itself is becoming a challenged network. The threat of computer virus infection has increased the proliferation and aggressiveness of Internet firewalls. In addition, the dwindling supply of public IP addresses has led to the popularity of NAT gateways which effectively hides machines behind private IP addresses [8]. In both cases, bidirectional communication has been severly constrained (by limiting port numbers) or eliminated altogether (in the case of NAT-to-NAT communication). This restriction severely limits the ability of P2P applications to make use of these NATed nodes. What we are left with is a challenged network where a growing population of *private* machines can only communicate (unidirectionlly) with *public* machines.

In this paper we present a solution to this problem - the distributed hash queue (DHQ). The DHQ provides durable network storage that can be used to facilitate communication between disconnected peers. A sending host places network packets into the DHQ and a receiving host subsequently pulls packets from the DHQ. All queues are named using 160-bit keys and a queue lookup (naming) service has been built on top of the Pastry key-based routing protocol. The DHQ prototype runs on top of the PlanetLab network testbed and the initial implementation consists of appoximately 2500 lines of Java code.

2 Simplified DTN Architecture

As defined by [5] and [4], a general delay-tolerant network provides several different classes of service and delivery options. These include 'Bulk', 'Normal', and 'Expedited' service and 'Return Receipt' and 'Secure' delivery options, among others. In addition, a DTN provides multi-hop routing across several regions using name tuples.

In this paper, we provide an implementation of a simplified DTN architecture than can be extended to the general case. Our architecture consists of a single 'Reliable' class of messaging service and we use 160-bit hash keys for names Delivery options are not provided by default, howevever, they can be added at the application level if desired. Routing is limited to single 'hop' paths, from a NATed network node to another NATed network. node. Multi-hop paths can be built by inserting application-specific route headers into message contents but that is beyond the scope of this paper.

To summarize, then, the DTN that we describe in this paper has the following basic properties:

2-region connectivity. Messages can be routed between two disconnected network regions, i.e. two NATed nodes.

160-bit names Message queues are named by 160-bit keys.

Reliable Delivery Option All message are reliably delivered in the face of up to K network faults. The constant K is a configurable parameter but is set to 3 by default.

3 Background

The DHQ system makes extensive use of the Pastry key-based routing (KBR) protocol. Pastry is used to implement the DHQ name service and to help in replicating queue state. While Pastry is used for the implementation, any KBR protocol would be sufficient. In this section, we give a brief background of the Pastry system. For a complete description, please see [14].

In the most basic sense, Pastry maps 160-bit keys to IP addresses. Thus, given any 160-bit key, Pastry will return the closest IP address to that key. This provides the basis of the DHQ name service, since we need to map queue names (160-bit keys) to the host that owns the queue state.

In the Pastry system, the 160-bit key space is configured in a ring (from 0 to $2^{160} - 1$) and the nodes are distributed along the ring. All nodes are assigned a node ID which consists of a 160-bit key and a IP address. Using a consistent hashing algorithm (e.g. SHA1), the IP address is deterministically hashed to a key. In addition to being deterministic, the hashing algorithm also generally provides a uniform distribution of keys. So the nodes are roughly distributed in the 160-bit key space in a uniform manner. For an actual distribution of 8 nodes, see Figure 1.

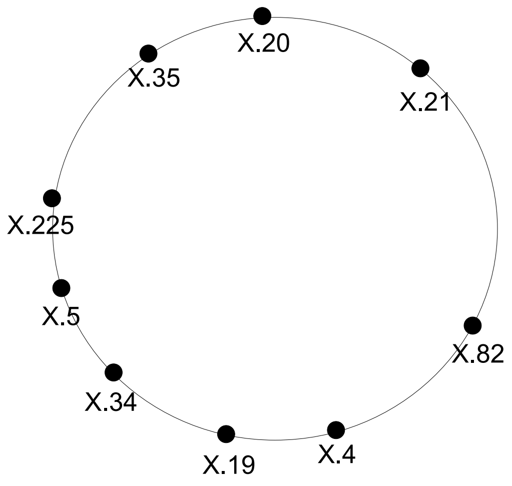

Fig. 1. This figure shows eight Pastry nodes distributed along the 160-bit ID space. The output is obtained by using the Gateway command 'Range' which provides the range of ID space for each node. Only the last decimal is displayed for each IP address in dotted-decimal format.

An important feature of Pastry (and other KBRs) is that the average route path from any node to the owner of an ID is $log(N)$ in the number of nodes in the system. In Pastry, in fact, the average route path is $log_b(N)$ where the base is 16. So the system can potentially scale to a large number of peers.

4 Distributed Hash Queues

The Distributed Hash Queue (DHQ) system provides a queueing service to both public and private peers on the Internet. At the highest level of abstraction, senders push messages to named queues and receivers pop messages from named queues. A request-reply messaging service can be built on top of the queueing service by using the tag field in the queue element structure to match requests with replies.

Senders and receivers are assumed to be applications running on NATed network nodes, e.g. a pair of instant-messaging applications. The DHQ service consists of N nodes running on the PlanetLab which are publically addressable (i.e. have public IP addresses) and participate in a single Pastry ring (group of cooperating nodes). See Figure 2.

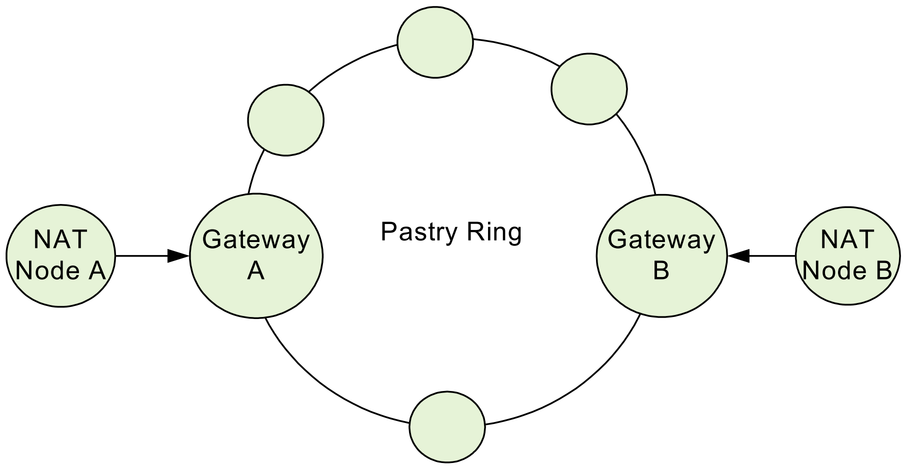

Fig. 2. This figure shows the logical structure of the DHQ service. Two communicating NATed nodes, A and B, connect to the DHQ service via the closest respective gateway node. Once connected, the NATed nodes can issue queue commands, e.g. push and pop.

The DHQ system consists of three services: a reliable naming service, a gateway service (for accepting requests from NATed nodes), and the core reliable queueing service. See Figure 3 which shows the layered structure of the DHQ system.

4.1 Reliable Naming Service

All queue operations operate on named queues and must use the naming service in order to locate the queue owners. The naming service provides a mapping from queue names (160-bit keys) to a set of K locations which replicate the queue state for redundancy. In addition, in order to prevent the naming service itself from becoming a single point of failure in the system, names are replicated across K nodes for fault-tolerance. (In practice, K is chosen to be 3.) The name-to-queue-owners binding is replicated by making use of the Pastry *replica-set* feature which finds the K closest nodes to a particular ID. A queue name is first converted to a Pastry key *key*, and then the Pastry system is used to locate the K node handles which may contain the name binding.

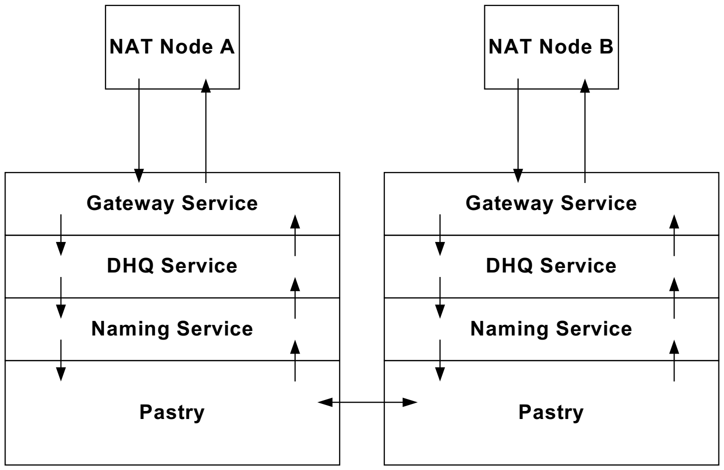

Fig. 3. This figure shows the layered structure of the DHQ system. The arrows indicate communication between layers and between entities. The DHQ and Naming services are implemented as Pastry applications and communicate strictly through Pastry. The NAT nodes connect to the DHQ service via the Gateway service which listens for TCP/IP connections.

For example, consider a lookup of the queue named "foo". First, the name "foo" is converted into a Pastry key foo_{key} which begins with the hex digits $0x338A$.... A request message for a list of name replicas ($LookupReplicasMessage$) is then sent to the Pastry node with ID closest to the key $0x338A$.... This closest node responds with a list of K replica node IDs. A name lookup is then attempted in parallel to each of these replicas, and the first valid response is returned to the caller. (A similar mechanism is used by the PAST storage system [15].) See Figure 4 which shows the operations involved.

Note that the initial $LookupReplicasMessage$ is a single point of failure in the present system. If the node Id closest to foo_{key} is down then the protocol cannot proceed. However, the protocol could be modified to find the second (or N^th) closest node to the key foo_{key}. This second closest node would either contain the name binding or be able to forward the message around the faulty node to another node which could satisfy the name lookup. This is due to the fact each Pastry node maintains a set of L neighbor nodes in the clockwise and counterclockwise directions. Note also that Pastry periodically sends heartbeat messages in order to determine and prune dead nodes. If a dead node is found, ownership of the dead node's ID space is transferred to live nodes. Thus, the single point of failure only exists for a time equal to the heartbeat period and the time to transfer ownership of the ID space.

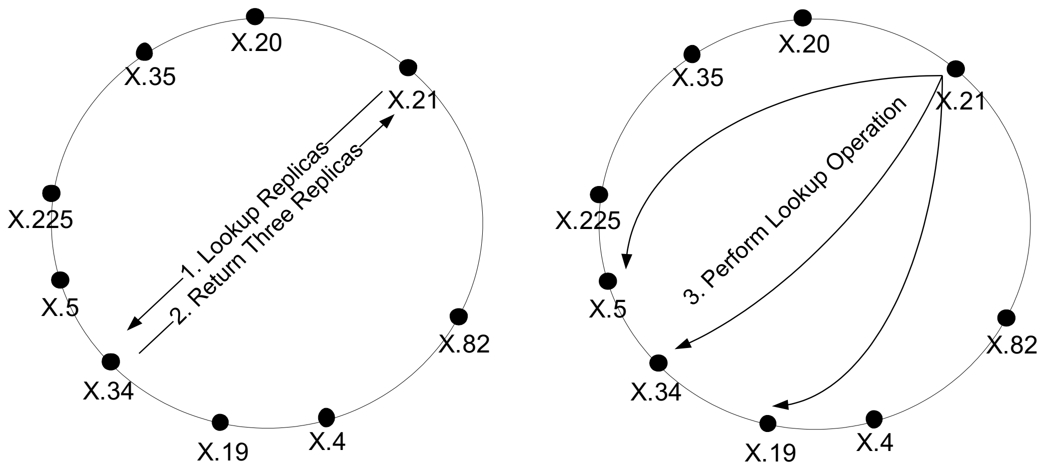

Fig. 4. This figure shows the steps taken during a lookup operation on the Reliable Naming Service. First, a set of replicas is fetched from the node closest to the name key. Then, a lookup request is multicast to these nodes and the first valid response is used.

4.2 Gateway Service

In the DHQ system, the NATed peer nodes do not participate in the Pastry ring, i.e. they do not own a part of the Pastry ID space. This is by design since NATed nodes are assumed to be highly dynamic and would introduce a high churn rate [13] into the system which would decrease stability. Instead, NATed nodes communicate to the Pastry ring nodes using a Gateway Protocol over standard TCP/IP. Commands are sent as human-readable single-line ASCII strings in order to ease parsing and debugging. In addition, this simple protocol makes the process of creating DHQ clients much simpler. The only requirement for a DHQ client is that it must support TCP/IP and be capable of sending ASCII strings. In fact, during the debugging process, a telnet client was used to connect to the ring and issue push and pop commands. The list of gateway commands is described below.

Alive *queue_name* lists the nodes which contain a live copy of *queue_name*. This list decreases monotonically as nodes fail until the queue is fixed using the Fix command (see below).

BlockingPop *queue_name* blocks until the queue has at least one element then returns that element.

Create *queue_name* $(IPaddress1, IPaddress2, etc.)$ creates a queue named *queue_name* on the machines represented in the IP address list. In the case that no list is given, the current gateway node and its neighbors are used to replicate the queue.

Delete *queue_name* delete a queue from the system. This removes the name *queue_name* from the naming service so that the queues are effectively deleted.

Fix *queue_name* ensure that the queue name *queue_name* is K-replicated and the queue state is K-replicated. For each queue, a Fix command is issued periodically by the system (every 2 minutes) in order to maintain the replication factor of each queue.

Range This returns the ID space that the gateway node is responsible for. This is used for debugging purposes and to map out the distribution of the ID space to each node. See Figure 1 for a graph produced using the Range command.

NameAlive *queue_name* - This returns the set of nodes that are replicating the name binding for *queue_name*. This set is not usually the same returned by the Alive command.

Peek *queue_name* - Return the first element from the queue *queue_name* without removing it.

Pop *queue_name* - Destructively return the first element from the queue.

Push *queue_name* **"value"** - Push an element onto the queue *queue_name* consisting of the string "value".

QueueInfo *queue_name* - Used for debugging. Return a string represntation of the queue size and contents.

Where *queue_name* - Return the list of queue replicas. This is a superset of the nodes returned by the Alive command.

For example, the following set of commands will create a queue named 'foo' and push three items onto it:

1. create foo
2. push foo "First Element"
3. push foo "Second Element"
4. push foo "Third Element"

NATed nodes attach to Gateway nodes by using a bootstrap process that is as follows. First, a NATed node contacts a seed node that it obtained via some out-of-band process. Then, the NAT node executes the nearby-node algorithm from [1] in order to find the closest (in terms of network latency) Gateway node. In our experience, the nearby-node algorithm tended to be biased towards returning the seed node and an improved algorithm based on Vivaldi [2] network coordinates is currently underway.

Once the closest node is found, the NATed node opens up a socket connection to the gateway over a well-known port number. Once connected to the Gateway Service, the NAT node issues commands (one per line) and receives any responses (e.g. to pop messages) over the network stream. If a connection is lost, the Gateway can restart the bootstrap process to find a better node or try to connect directly to the Gateway again. Gateway commands are translated directly into queue operations which are then handled by the Reliable Queue Service which is described below.

4.3 Reliable Queue Service

Queues are replicated across a set of K nodes (K is 3 in practice) which are specified upon creation of the queue. The set can either be specified by the NAT client or automatically picked by the DHQ system. The advantage of letting the client pick is that locality can be optimized, while the advantage of letting the system pick is that load can be balanced. In both cases, the creation of the queue also creates a binding from the queue's name to its queue replicas.

Queues are implemented as priority queues where the message timestamps denote priority. This provides a total ordering on messages given synchronized global clocks. Given weaker time synchronization, however, the priority queues still serve a purpose: they provide a consistent ordering of packets in replicated queues. Therefore, if messages are replicated across a set of K queues, the priority feature ensures that messages will be seen by queue readers in the same order regardless of which queue is accessed. In the face of message loss, ordering is still preserved. For example, given messages $m1$, $m2$, $m3$ with increasing timestamps, receivers will receive $m3$ before $m1$ if the message $m2$ is dropped. Implementing an improved total-ordering policy is the subject of our current research.

After a name binding is found, the set of nodes which 'own' the queue state are returned - called the queue owners. The queue operation (e.g., push, pop, peek) is then multicast to the queue owners. In the case of a push, the push message is sent and control returns immediately to the calling program. For a pop (or peek) operation, the message is sent and the first valid reply is returned to the caller. (Valid replies are replies that are not errors.) Thus, if a queue is originally owned by 5 nodes and 2 are currently down (or have been rebooted), then three valid replies should be returned to the user. The first such valid reply is used as the value of the pop, and the other two are discarded. (Optionally, all replies can be returned to the user and the user can use a voting method or some other protocol to decide the correct value to use.)

The set of queue operations that are supported includes the set of Gateway commands plus some additional commands. Only the additional commands are listed below:

CreateQueueReplica *queue_name queue_state* This message is sent, along with serialized queue state, to a node in order to manually replicate a queue.

GetQueueState *queue_name* This command is used to fetch the entire state of a queue from a remote node.

PingQueue *queue_name* Determine if a queue exists.

WatchQueue *queue_name* This message is scheduled periodically using the Pastry $schedule - message$ primitive. A WatchQueue message, when received by the QueueService, will automatically fix a queue and maintain the invariant that the queue and its name binding has K replicas.

Push and pop operations are multicast to all of the K queue owners in order to attempt to preserve queue consistency. For a push operation, we chose not to use a synchronous two-phase commit protocol such as ABCAST [7], but rather we use a best-effort send which attempts to send the message to all live queues.

While this does not guarantee consistency, with a large enough value of K it does probabilistically guarantee that the message will not get dropped.

In order to preserve the queue state over long delays, it is important that the queues be able to survive many faults. For example, in a delay-tolerant network, days or even weeks may go by before a message can successfully be delivered. Therfore, the DHQ needs to durably store messages so that they can survive multiple faults. This is handled by the initial queue replication, and a periodic process which re-replicates queue state every S seconds. In practice, we have used a value of S to be 120 seconds, although this value is tuneable and should be set according to the environment in which the DHQ is operating. In this initial implementation, the faults that we are trying to survive are mainly the periodic reboots of PlanetLab nodes. Currently, we are assuming fail-stop nodes which simplifies the implementation. Future work will be to survive other kinds of failures and to improve the consistency guarantees of the system.

4.4 Replication Factor

In this section we show that the choice of 3 as the replication factor for queue names and queue state is a reasonable one. While it does not guarantee resiliency, the replication factor does provide a probabalistically high guarantee against data loss. A recent survey of the PlanetLab system, consisting of 242 nodes over a period of three months shows that the number of simultaneous reboots (nodes that reboot within 5 minutes of each other) is always less 8% and typically less than or equal to 3 nodes. See Figure 5.

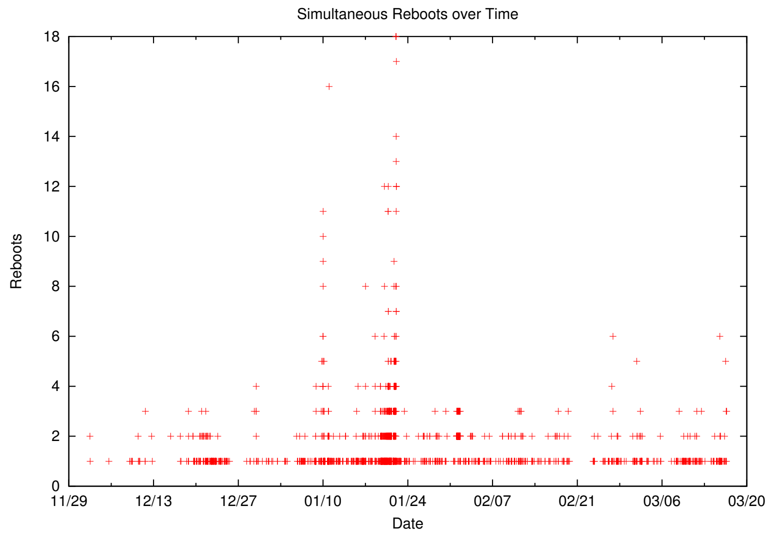

Fig. 5. This figure shows the number of nodes that are unavailable (due to system reboots) over time on the PlanetLab system.

If we assume that queues are replicated at three randomly chosen nodes, then the probability of choosing three bad nodes in this system of 242 nodes is given by:

$$\frac{B}{N} * \frac{B-1}{N-1} * \frac{B-2}{N-2} \tag{1}$$

where B is the maximum number of rebooted nodes at any time (i.e. bad nodes) and N is the total number of nodes in the system. In the case of our measurements, B is 18 and N is 242. Thus, the probabilty of choosing three bad nodes is less than 1% (around 0.035%).

Since the queues are periodically re-replicated (every two minutes) to maintain the three replica invariant, we have reason to believe that the system will do a good job of maintaining queue state. Current research is being done to quantify how much queue state is lost and what countermeasures can be taken to prevent queue state loss.

5 Related Work

In this paper, we have described a mechanism for allowing communication to a NATed network node with a private IP address. Some related work in this area has attempted to tackle this very problem including AVES [9] and i3 [16]. In AVES, the NAT gateway (and DNS server for performance reasons) is modified in order to support incoming connections to private IP hosts. A public network waypoint address serves as the virtualization of the private IP address, and relays IP packets from a public IP address to the private IP address through the modified AVES NAT gateway. The main constaint on the AVES solution is that it requires gateway software modifications which may not be administratively possible by all NATed clients. In addition, while AVES does provide general bi-directional communication from host-to-host, it still makes the assumptions of low RTT and packet loss and therefore is not a candidate for building a complete delay-tolerant network (DTN).

The Internet Indirection Infrastructure (i3) is another possible choice as a substrate for building a DTN. In i3, packets are sent not to an IP address but rather to a rendezvous node identified by an m-bit key, called k. An overlay network (Chord is used in the i3 implementation) then routes data packets to the node associated by $successor(k)$ in the Chord system. Any interested parties can register triggers with the rendezvous node (again, using the key k to identify the rendezvous node). The triggers then forward packets to the interested nodes. What i3 provides through this indirect communication is the ability for recipients to be mobile. For example, if a host moves from address 128.A.B.C to 128.X.Y.Z, then it simply must refresh its trigger to point to its new IP address. A recipient's mobility, however, is still limited to the public Internet since triggers forward packets using IP. In addition, i3 does not provide network storage for packets as is required by a DTN - packets are simply forwarded by a trigger as soon as they arrive. If the destination host is currently unavailable, then the packet is lost and must be retransmitted by the source node. In a DTN, it is the network that provides network storage and/or retransmission before failing.

In general, key-based routing (KBR) [3] protocols such as distributed hash tables (DHT), provide name indirection (using m-bit keys) and network storage. However, they do not address the problem of communication to private IP addresses nor do they provide a messaging service. In the case of DHTs, the assumed workload is a collection of large files that map one-to-one with a key identifier [12]. If a messaging layer were built on top of a DHT, then only one outstanding message at a time could exist (given the one-to-one key-to-value mapping). Thus, while a DHT solves some of the problems needed by a general DTN (network storage and naming), it does not provide private IP connectivity nor does it provide the right abstraction for messaging.

The POST system [10] provides secure and reliable messaging between disconnected hosts. Like DHQ, POST is built on top of key-based routing protocol and provides message storage in the network. The main differences between the two systems is the fact that the POST system design assumes bidirectional communication between hosts (it is a P2P messaging system) and is focused on secure messaging. While end-to-end security can be added on top of DHQ at the application layer, it is not a focus of this paper.

The IP Next Layer (IPNL) system [11] provides connectivity to NATed hosts by extending IP addresses to be a triple of a public IP address, realm ID, and private IP address. Other network communication remains the same, so that the IPNL does not handle the long storage delays that are inhernet to DTNs. Also, while IPNL provides a general purpose NAT-to-NAT communication mechanism, it does so by modificating the IP layer and therefore requires router modifications.

6 Conclusion

In this paper we have described the architecture and design decisions involved in building a distributed-hash queue (DHQ) service. The primary reason for building such a service is to provide rendezvous communication for private NATed peers in a P2P system. The requirements of any such service is to provide a set of public waypoints and data durability. In our system, we use the PlanetLab testbed to provide public waypoints. In addition, in order to survive faults and provide long-lived data durability we have chosen to replicate both the queue names and the queue state. The API of our distributed hash queue service has been described in detail. Our planned future work includes evaluating the system that we have implemented in terms of its performance overhead (over point-to-point communication protocols) and resilience to faults.

References

1. M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Exploiting network proximity in distributed hash tables. In O. Babaoglu, K. Birman, and K. Marzullo, editors,

International Workshop on Future Directions in Distributed Computing (FuDiCo), pages 52–55, June 2002.
2. R. Cox, F. Dabek, F. Kaashoek, J. Li, and R. Morris. Practical, distributed network coordinates. In *Proceedings of the Second Workshop on Hot Topics in Networks (HotNets-II)*, Cambridge, Massachusetts, November 2003. ACM SIGCOMM.
3. F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a common api for structured peer-to-peer overlays. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS03)*, Berkeley, CA, 2003.
4. Delay-Tolerant Network Architecture. ftp://ftp.rfc-editor.org/in-notes/internet-drafts/draft-irtf-dtnrg-arch-01.txt, 2003.
5. K. Fall. A delay-tolerant network architecture for challenged internets. Technical Report IRB-TR-03-003, Intel Research, Feb. 2003.
6. K. Fall. Delay-tolerant networks. In *Proceedings of ACM SIGCOMM 2003*, Karlsruhe, Germany, Aug. 2003.
7. B. Glade, K. Birman, R. Cooper, and R. van Renesse. Lightweight process groups in the isis system, 1993.
8. Y. hua Chu, A. Ganjam, T. E. Ng, S. G. Rao, K. Sripanidkulchai, J. Zhan, and H. Zhang. Early experience with an internet broadcast system based on overlay multicast. Technical Report CMU-CS-03-214, CMU, Dec. 2003.
9. T. S. E. Ng, I. Stoica, and H. Zhang. A waypoint service approach to connect heterogeneous internet address spaces. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 319–332. USENIX Association, 2001.
10. A co-operative messaging infrastructure. http://freepastry.rice.edu/POST/default.htm, 2004.
11. P. F. Ramakrishna. Ipnl: A nat-extended internet architecture. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 69–80. ACM Press, 2001.
12. S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The oceanstore prototype. In *Proceedings of USENIX File and Storage Technologies (FAST)*, 2003.
13. S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a dht. Technical Report CSD-03-1299, UCB, Dec. 2003.
14. A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Nov. 2001.
15. A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 188–201. ACM Press, 2001.
16. I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. In *Proceedings of ACM SIGCOMM Conference (SIGCOMM '02)*, Aug. 2002.