A Lookup Service for Delaunay Triangulation Overlays: The DeTella Application

John Giordano, Greg Mattes, Pascal Vicaire, James V.S. Watson

University of Virginia
Department of Computer Science
1003 Engineer's Way
Charlottesville, VA 22901
{jcg8f, grm9f, pv9f, jvw3n}@cs.virginia.edu

1. Abstract

In this paper we describe the DeTella distributed overlay directory service. DeTella is composed of network nodes that reside in a logical address space. Data that is inserted into the network is "owned" by a single node and replicated by "neighboring" nodes. Each data item has a logical address that is used to determine which DeTella node owns it. The logical address space that DeTella uses is combined with the HyperCast [3] overlay multicast software for network transport. By using the robust self-organizing properties of the HyperCast software, DeTella can quickly and efficiently reorganize itself in a changing network environment. We feel that our approach is superior to existing distributed directory architectures. The rest of this paper is organized like this: Section 2 gives an introduction to our work, Section 3 discusses previous work in this area, Section 4 outlines our design strategy, Section 5 describes our implementation using HyperCast, Section 6 presents our analysis and testing, Section 7 gives results of our analysis and testing, Section 8 contains our conclusions, finally Section 9 summarizes possible future work.

2. Introduction

Merging a Delaunay triangulation (DT) protocol with a distributed hashing function and lookup service can allow for spontaneously-generated, self-organizing overlay networks that are scalable, fault-tolerant and otherwise robust. Intended as an extension of work in the peer-to-peer domain, we present in this paper our work in designing an architecture and network node software that employ compass routing to distribute key-value pairs in an overlay network, and user functionality to insert, query for, and delete content addresses in the network space. Whereas popular peer-to-peer (PTP) applications such as Napster and Gnutella have fallen victim to legal challenges presented by intellectual property rights owners, our architecture provides a distributed lookup service, but not content availability, access, or distribution. Napster suffered from a centralized lookup service that not only introduced overhead requirements into the network and created a central point of failure, but also served as a prominent target for litigation. Gnutella, which employs service request flooding, a technique which does not scale very well, also distributes content over the network between nodes, again drawing the attention of those concerned with IP rights. Like emerging PTP applications, ours

does not require centralized organization or hierarchy, and in prototype development, provides sufficient features for ease of use.

Herein, we investigate the algorithms and system design required to create and deploy such a service, test our prototype implementation, analyze and report our quantitative & qualitative results, and pose conclusions as well as possible avenues of future work. In Section 3, we review some of the seminal work in extending the scalability & robustness of PTP applications. In Section 4, we discuss the design considerations and algorithm development for employing a DT protocol based on compass routing and featuring replication of key-value pairs among network neighbor nodes. Section 5 describes the implementation of our design and development of additional functionality. In Section 6, we demonstrate the capability of our system through experimentation and analyze the results in Section 7. Finally, we draw our conclusions and discuss potential refinement of and extension to our system. Our work, while not immediately deployable, validates certain claims about distributed services in overlay networks and describes the potential for further exploitation.

3. Background & Related Work

As described above, previous work in the PTP applications domain resulted in very popular products and services that did not scale very well, nor withstand legal challenges posed by those concerned with the nature of content distribution among network nodes. More so than the legal ramifications, the designs behind these products lacked fundamental concern for efficiency, scalability, and robustness, three critical attributes in distributed applications.

3.1 The Chord System

Seeking more theoretically sound approaches, Stoica et al. [1] proposed a lookup service employed at nodes arranged in a flat (linear one-dimensional) key space that provided one fundamental service – mapping a hashed lookup "key" to a value stored at a node in the network. The key is a consistent, unique hashing of some query field that a user seeks to find in the network. The value could take on almost any form – an address such as a URL, a document, list, or other data structure. As a result of its simplicity and elegance in design and functionality, the Chord system reported lookup resolution in no worse than $O(\log N)$ time in an N-node network, with a probability of lookup success not exceeding $O(\log^2 N)$. The Chord protocol provides for the ability to locate keys, scalability of the network as new nodes join, and stability & robustness in the network in the face of node departure – graceful or otherwise.

Stoica et al. conclude that the Chord system solves the problem of centralization by determining key-value location in the network based upon the uniquely hashed keys that are assigned to nodes with equivalent or succeeding m-bit identifiers which are produced by a well-known hashing function. Because of its simplicity, it scales very well. By adding the redundancy of Chord nodes maintaining state information about their successors and replication of those neighbors' key-value pairs in the linear space, the authors provide a provably correct and experimentally sound design.

3.2 The CAN System

A content addressable network (CAN) was introduced by Ratnasamy et al that provides functionality similar to the Chord system – that is, distributed mapping of key-value pairs to nodes instantiated in a d-dimensional Cartesian space on a d-torus [2] with the objective of providing a scalable, robust lookup service to extend previous successes in the PTP domain. The authors pose that their CAN system is not limited to strictly PTP, but could be generalized to large scale or distributed data storage as well. Their design incorporates a unique hashing function to assign key-value pair coordinates in the d-dimensional space, similar to the Chord method, but the CAN nodes successively halve the Cartesian space with their neighbors as they join the network with randomly assigned coordinates. Based on experimental results, a CAN configured with n nodes in a space with d dimensions will consist of path lengths no bigger than $O(d(n^{1/d}))$, thus providing scalability corresponding to that of Chord. The CAN system also includes algorithms to handle the graceful or immediate departure of nodes from the network, where a node's neighbors in the space redundantly store key-value pairs to accommodate failures as well as facilitate departures.

3.3 The Delaunay Triangulation Protocol

While both Chord and CAN demonstrate improvement to the concept of PTP networking, the logical addresses assigned to nodes and key-value pairs have limitations. By employing a DT protocol in an overlay network on the Internet, Liebeherr et al. [3] demonstrated that, at the cost of degraded resource utilization, one can build and maintain very large, highly distributed logical networks. Node assignments can be user generated, pseudo-random, or be assigned based on geographical location. The authors seek to extend their work by introducing adaptive coordinate spaces that take network delays and latencies into account. While not directly concerned with the insertion and maintenance of key-value pairs, the authors implemented a compass routing algorithm that determines network neighborhoods and parent-child relationships by comparing the smallest angles between adjacent nodes in a 2 dimensional grid. One can see how the Chord and CAN functionality of hashing key-value pairs and inserting them into the overlay can be applied to the DT protocol.

Herein, we present our work in creating a DT overlay network employing compass routing that is scalable and robust, if not highly efficient, that provides functionality to insert key-value pairs based on a deterministic hashing of some subset of the key.

4. Design

This section presents the design of our software. We first discuss the hash function used to obtain Cartesian coordinates for a filename. We then describe how a node decides whether some point is within its Voronoi region. We also describe the general functioning of one node. Finally, we explain the implementation of the insert, delete, and query operations.

4.1 Hash Function

A node hashes a file URL to obtain coordinates using the method:

```
I_Address getHash ( String filename );
```

This method returns an I_Address corresponding to coordinates pertaining to the set S with:

```
S = [0…10000] x [0…10000]
```

We must take care to obtain a great variety of x and y coordinates even if filenames are similar (this might happen if all the files begin with "Beethoven," for instance). These coordinates should ideally be randomly distributed within the set S.

4.2 Voronoi Region Test

Given some coordinates, a node N decides whether the corresponding point P is within its Voronoi region using the method:

```
bool isWithinMyVoronoiRegion ( I_Address address );
```

This method identifies the neighbors of N using the getNeighbors() method of the I_OverlaySocket class. If none of the neighbors is closer to P than N, N assumes that P is within its Voronoi region.

4.3 Node Functionality

Each node maintains a table where one entry corresponds to one file. A node keeps information (key and URL) about each file contained in its table. Several nodes keep information about a single file to insure recovery in case of node failure. However, only one node is responsible at a given time for one file: the node containing the file in its Voronoi region. For instance, the table for a node N1 could look like:

Entry Number	File Key	Data	Responsible Node	Iterations Until Deletion
0	(5700 , 2400)	URL\file1.mp3	(N1 = 5678, 2345)	20
1	(5600 , 2300)	URL\file2.mpg	(N1 = 5678, 2345)	20
2	(1400 , 8500)	URL\file3.jpg	(N2 = 1478, 8523)	15
3	(6900, 1200)	URL\file4.dat	(N3 = 6987, 1234)	10

Table 1: Concept for a Node Data Table

From this table, we can deduce that N1 is responsible for storing file1 and file2 information. N1 is as well responsible for asking its neighbors to keep duplicated information about file1 and file2. N1 not only keeps information about the files it is responsible for but also keeps duplicated information concerning file3 and file4, for which respectively N2 and N3 are responsible.

Periodically, each node verifies its neighbor set. Any change to that set whatsoever results in the generation of an internal event concerning a neighborhood change. A timer initiates this periodic operation. Below are two examples of circumstances that characterize a change in neighborhood:

- A neighbor node N2 has disappeared. Each file for which N2 was responsible is checked. If the file is within the Voronoi region of N1, N1 becomes responsible for this file and

asks his neighbors to keep duplicate information about this file. If the file is not within the Voronoi region of N1, N1 discards the file information.

- A new neighbor node N2 has appeared. N1 sends to N2 the information about the files it is responsible for (the files within its Voronoi region). N2 will keep duplicate information about these files. N2 is responsible for asking N1 to keep duplicate information about its files.

If N1 becomes responsible for some new files, if N1 loses responsibility for some file, or if N1 modifies file information, it sends updates to its neighbors. Some inconsistencies might occur if a node N1 dies just before sending updates to its neighbors.

Each time a node receives a packet, it processes the packet's message in a processor thread. There can be multiple processor threads in the system so as to be able to process simultaneous multiple queries and periodical tasks.

If a node knowingly leaves the network, its neighbors must take notice of this departure and adjust the contents of its data table accordingly.

4.3.1 Insert Operation

Nodes initiate insert operations using the method:

```
insert ( I_Address address , String url );
```

The insert operation can be started from any node. The local node tests if the address is in the Voronoi region of the node. If no, a packet containing the (address , url) pair is created and forwarded to a neighbor closer to the file address than the local node. If yes, the (address , url) pair is inserted locally and neighbors are asked to keep duplicate information. The originator of the insertion is notified that the operation was successfully completed.

4.3.2 Delete Operation

Nodes initiate delete operations using the method:

```
delete ( I_Address address , String url );
```

The delete operation can be started from any node. The local node tests if the address is in the Voronoi region of the node. If no, a packet containing the (address , url) pair is created and forwarded to a neighbor closer to the file address than the local node. If yes, the (address , url) pair is deleted from the node table and neighbors are asked to update their information accordingly. After data is deleted, it cannot be inserted for a period of time in order to prevent neighbors who were replicating that data from reinserting stale data.

4.3.3 Query Operation

Nodes initiate query operations using the method:

```
query ( I_Address address );
```

The query operation can be started from any node. The local node tests if the address is in the Voronoi region of the node. If no, a packet containing the (address) singleton is created and forwarded to a neighbor closer to the file address than the local node. If yes, a list of all URLs that correspond to the address is returned to the node that initiated the request. Intermediate nodes that possess the queried information are not allowed to answer the request. A null result might indicate that the desired content is not in the network, but it can also be the result of the network reconfiguration. It is up to the user to decide whether or not to resubmit queries that return no results.

5. Implementation

In this section, we first describe the software that is run at each node of an overlay network distributed lookup service. Each node is modeled as an "event processor" that knows how to handle events generated in the distributed system. This section is intended to provide a framework for implementation as well as outline a scheme for replicating data and recovering from overlay node arrivals and departures. The terms "message" and "event" are used interchangeably in this section, with the intent being that a "message" is a protocol message ("on-the-wire" form) and that an "event" is a data structure used within the software running on a node. The equivalence of the two can be seen easily since the information needed to build an event data structure is contained in a message, while similarly, an event data structure must be used to construct a message.

5.1 High Level System Architecture

At a high level, the software running at a node is seen as a black box with inputs and outputs. The inputs and outputs are events, thus the node is modeled as an event processor. The input is broken into two categories depending on its origin as shown in Table 2.

Input	Output
External Events	Generated Events
Internal Events	

Table 2: High Level Information Flow

External events are those events that are read from a multicast socket, each node has a single multicast socket from which it reads and writes; threads in a node may independently block on both the reading and writing part of the multicast socket. The high-level system overview is depicted in Figure 1:

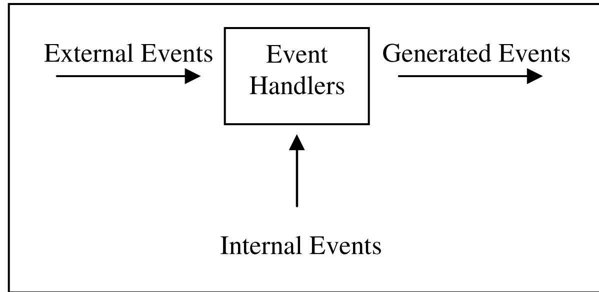

Figure 1: High Level System Overview

5.2 Detailed System Architecture

Considering the system in more detail we see that the system is decomposed into many threads as shown in Table 3.

Thread Name	Number of Threads	
	Multi-Processing	**Single Processing**
ExternalEventReceiver	1	1
NeighborhoodMonitor	1	1
EventDispatcher	1	0
EventProcessor	N > 1	1
WaitingTableMonitor	1	1
Generated EventsSender	1	1
GraphicalUserInterface (GUI)	1	1

Table 3: System Threads

5.2.1 Threads

The ExternalEventReceiver and NeighborhoodMonitor threads are responsible for reading external events and generating internal events respectively. In both cases, these events are enqueued in the system inputQueue to wait for the EventDispatcher to dispatch each event to an EventProcessor.

In particular, the ExternalEventReceiver thread runs in a loop that blocks on the read half of the multicast socket during each iteration. When a message arrives on the socket, an event is enqueued in the inputQueue.

The NeighborhoodMonitor thread runs in a loop that iterates once for some configurable time interval, e.g. on the order of 10 seconds. Each iteration constructs a set of nodes that are neighbors in the multicast overlay network. If the set of neighbor nodes is different from the set that was constructed in the previous iteration, the NeighborhoodMonitor thread will enqueue a NeighborhoodChanged event every time a change to the neighbor set is detected.

It is the job of the EventDispatcher thread to read events from the inputQueue and dispatch them to EventProcessor threads that handle the events. The number of EventProcessor threads (listed as "N" in Table 3) is fixed at configuration-time, i.e. the number of EventProcessor threads is

constant at run-time. The EventDispatcher keeps track of idle EventProcessor threads so that it knows where to dispatch incoming events.

The EventProcessor threads contain the logic to respond to each event that enters the system. There can be more than one EventProcessor thread so that multiple events can be handled in parallel. It is not clear at this time whether or not having multiple EventProcessor threads will be useful. Our architecture supports easy configuration of the number of these threads, so if it is not advantageous to have many of these threads, we will simply configure the system to have one of them. EventProcessor threads encode the algorithm that implements the distributed lookup scheme. The events cause updates to be made to the Lookup Table that is found on the node. Events may generate other events that will be placed on the outputQueue to be sent to their proper destinations. Generated events destined for the local node, however, are sent directly to the inputQueue.

The WaitingTableMonitor thread is responsible for doing retransmissions of messages that require acknowledgment, but have not received acknowledgment. The period for this timer is user configurable. In addition to retransmission of messages, every third iteration of the WaitingTableMonitor sends messages to nodes that are owners of data that is replicated at the local node. These messages ask the owner if it is still the owner of replicated data. If a message is received that tells the local node that the owner for the replicated data has changed, the local node sends an Insert event into the network for the replicated data.

The WaitingTableMonitor has a third purpose - every five iterations, the deleted counter on deleted items in the lookup table is decremented by one. While this counter is greater than zero, no insertion of the data are allowed (to stop replicas from inserting stale data); when this counter reaches zero, the item is removed from the lookup table and the data may be reinserted

Finally, the GeneratedEventSender thread is responsible for reading generated events from the outputQueue and sending messages to other nodes in the overlay network (not necessarily neighbors). This thread is responsible for "patching" compass routing to account for the fact that data addresses are not always an exact match for node address. In particular, this thread calls our "chooseNeighbor" routine which computes the neighbor node of the current node that is closest to the data address in the event that the data address does not match. Typically, we only need to call this routine after a message has gone 10 hops, as the message may be oscillating around the owning node.

5.2.2 Queues

In addition to threads, the system also has data structures. The system has two queues that are used to hold messages, the inputQueue and the outputQueue. These are data structures that give synchronized access to a list of events (input) and messages (output). In Section 6, we discuss the impact that varying queue length has on system performance.

The programming model for the SynchronizedQueue data structure is that there are two operations: add (Object) and remove (Object). The add method will block if the queue is full, waiting until there is enough room to insert of new element. When there is room for a new

element, add will insert Object at the end of the queue and return. The remove method will block if there is an empty queue. When there is an entry in the queue, remove returns with the first entry.

5.2.3 The Lookup Table

Another data structure in the system is the Lookup Table. Events that enter the system operate on the information contained in the Lookup Table. This table, shown earlier in Table 1, stores information about the state of entries that are stored in the node.

Being "responsible," or being an "owner," means that N1 has the job of ensuring that entry 1 is replicated to all neighbors of N1. It is also N1's job to give responses to queries for entry 1 (nodes replicating the data will not give responses; our replication is for redundancy not load-balancing). As responsible node, N1 will take care of removing information of entry 1 from the network when a DeleteRequest for that data is issued.

Although N1 is not responsible for entry 2, it is responsible for ensuring that a valid owner exists in the network until N1 is told to remove the entry.

5.2.4 Events, Messages & Message Flows

The algorithms that control how entries are inserted, deleted, queried, and replicated in the overlay network are contained in the handlers of the various events. Below, we describe the different types of events and the messages that handlers must pass in order to process these events. Internally generated and processed events (i.e. never seen "on-the-wire") are enclosed in angle brackets, <internal>, events that are not present in all enumerated cases for a particular diagram are enclosed in parenthesis, (optional). Each enumeration case will say whether or not the optional event applies to it.

5.2.4.1 Insert operation

(1) User node inserts data and does not become a replica (user node is not a neighbor of owner node). MakeInsert event is required.

(2) Replica is seeking new data owner and the new data owner tells replica that it should no longer be a replica. No MakeInsert event is required.

(3) An owner node detects that it should no longer be the owner of the data and the new data owner tells previous owner that it should not be a replica. No MakeInsert event is required.

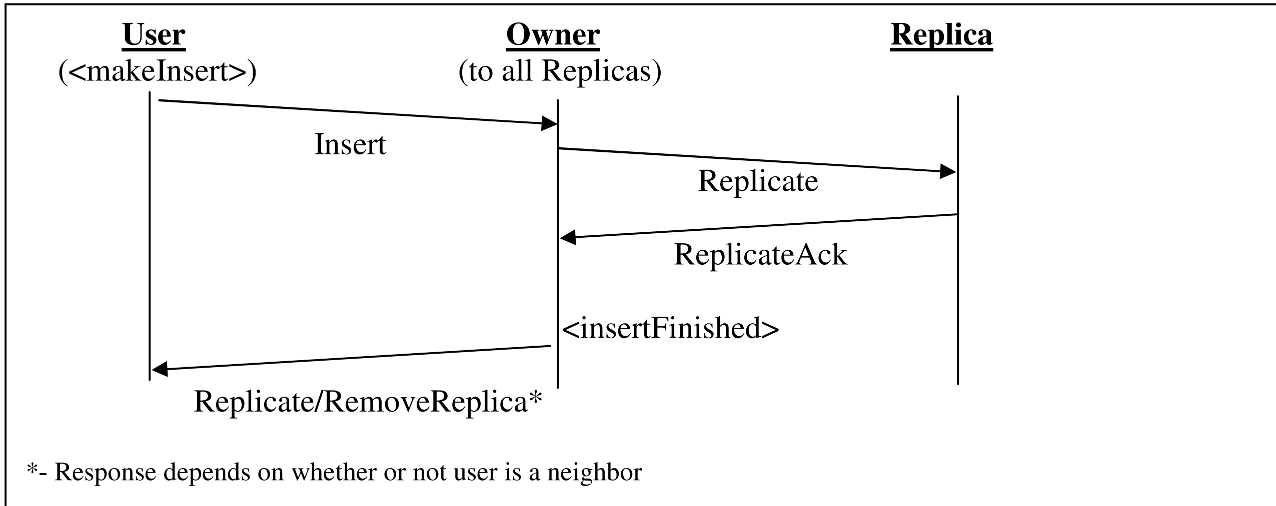

Figure 2: The Insert Event

5.2.4.2 Owner replicates to all neighbors

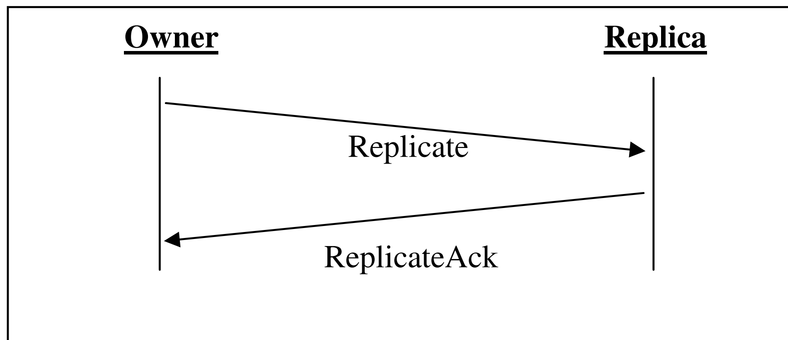

Figure 3: The Replicate Event

5.2.4.3 User queries for data item

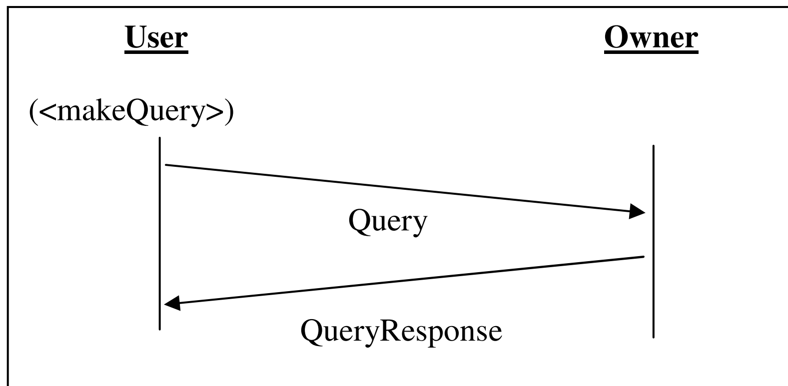

Figure 4: The Query Event

5.2.4.4 Change in state of neighbor set

NeighborhoodChanged happens when the neighborhood monitor detects a change in the neighbors list. Many other flows may be triggered by this event. For example an owner node may compute that it should no longer be the owner for some data.

5.2.4.5 Checking and Acknowledging Data Ownership

This event happens every 3 waiting table periods. This allows a node to determine if the owner nodes for any of the entries in its data table are still in the overlay.

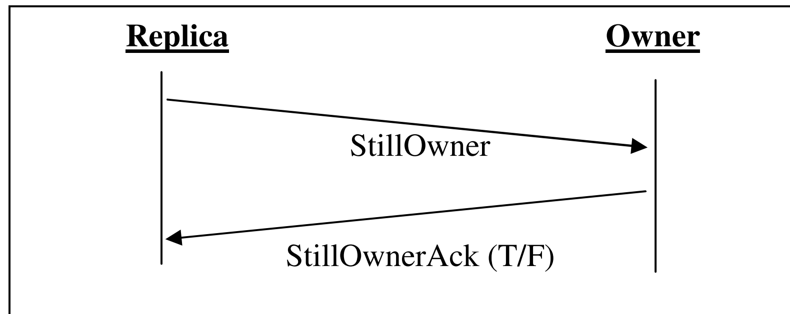

Figure 5: The StillOwner Event

We do this in order to deal with the following situation:

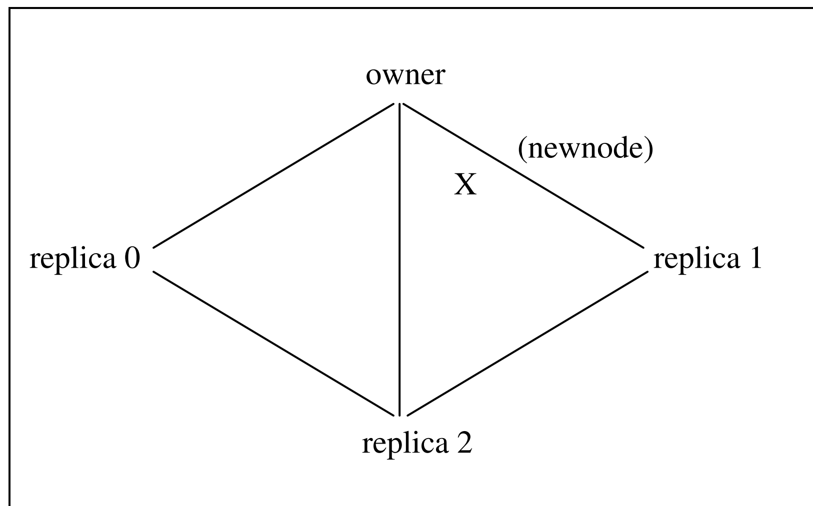

Figure 6: Ownership Scenario

As in Figure 6, if newnode joins the network and assumes ownership of X, replica 1, replica 2 and the owner node will see the neighborhood change and the owner will now be a replicate of X; replica 0 will not see the neighborhood change and will continue to replicate X, even though it is not a neighbor of newnode. Eventually, replica 0 will send StillOwner to the old owner, which will respond with StillOwnerAck(false). Upon receipt, replica 0 will insert its replica of X.

5.2.5 Message Details & Pseudocode

In order to more precisely describe the logic by which messages are generated and passed, we include the detailed descriptions and pseudocode segments in an appendix to this paper. Within each description, the "Algorithm" part is run by receiver of event and the "Node Not Alive" part is run by sender of event.

5.2.6 Graphical User Interface

We provide the user with a simple yet functional graphical user interface (GUI) to interact with the network, insert, delete or query for content. This simple GUI was built using Java Swing components and interfaces directly with the code for the network nodes. Be launching an instance of the GUI, a node is created on the local host, which will, if configured properly, join an overlay with other nodes that might be running. The GUI provides the capability for the user to extract the values returned from a query (URLs) and copy them to the URL field in a web browser. The user can then access the content that was indexed in the overlay lookup service, thus separating content access and distribution from the overlay network.

The cross-platform capability of Java and Java Swing allows the GUI to be run on almost all architectures and O/S varieties – x86 Windows & Linux, Sun Solaris & Motif, Mac, etc. It is similar in appearance and functionality to existing PTP clients, but obviously it does not facilitate content availability and distribution onto the overlay. A screenshot of the GUI is included in Figure 7.

Figure 7: Screenshot of the GUI

6. Analysis & Testing

In this section, we report and analyze the results of several experiments conducted in order to quantify the performance of our overlay application. These experiments were conducted entirely on machines located within the Department of Computer Science at the University of Virginia. We acknowledge that such a testbed does not account for the myriad of performance issues inherent to large-scale distributed networks and their applications, such as propagation delays and latency, competition for resources, etc., and that our results and conclusions pertain strictly to the design and implementation presented herein. Given more time, we feel that a wider range of experiments run on long-distance networks would yield results that could be generalized to similar implementations and better inform the reader of the benefits and drawbacks of the DT protocol being applied to overlay networks.

Since testing and analyzing our system requires that many nodes be instantiated in a relatively short period of time, we automated the testing process by using Expect scripts to successively launch nodes with identical configuration parameters (except for random address assignments) into an overlay network, and then populate the nodes with key-value pairs. Expect is a scripting tool that allows for communication with interactive programs on remote hosts. It runs on top of Tcl/Tk and greatly facilitates the automation of interactive testing [4].

We designed experiments that not only populate our overlay with key-value pairs, but more importantly, we query the overlay for existing data and report on percentages of successful queries and delays in results being returned, while varying the number of nodes in the overlay and the insertion or query period – that is, the time between successive operations in the given experiment. Most of our results are based on nodes with unbounded inputQueues, but we also show the results that varying the size of inputQueues has on system performance. In each experiment, we allow the network 2 minutes to stabilize after node instantiation before running our experimental scripts. We feel this is a reasonably low stabilization time, but we do not measure and analyze the effect of beginning our scripts as the network is stabilizing in order to determine precisely when stabilization has occurred. For each experiment, the nodes run for about 5 minutes.

7. Results

7.1 Insert Operation Experiment

Our first experiment analyzes the performance of the network with N = 32, 256 and 512 nodes in inserting key-value pairs with increasing frequency. Figures 8 and 9 show that in large networks (N = 512), file insert delay increases exponentially when insertion frequency is less than 30 seconds. In smaller networks, this delay only grows linearly. Large networks, however, cannot reasonably assure the user that desired content will be inserted, even when the operations are 60 seconds apart. Smaller networks give the best assurance (85% - 90%) of successful insertion when the interval between operations is that long, but that assurance decreases exponentially and becomes unreasonable as insertion frequency increases.

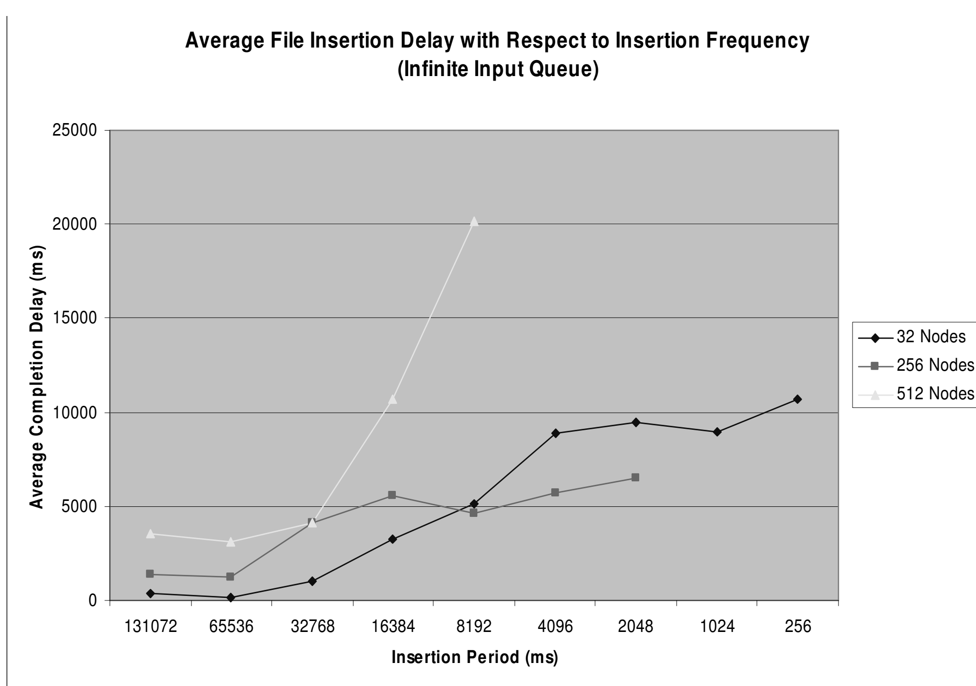

Figure 8

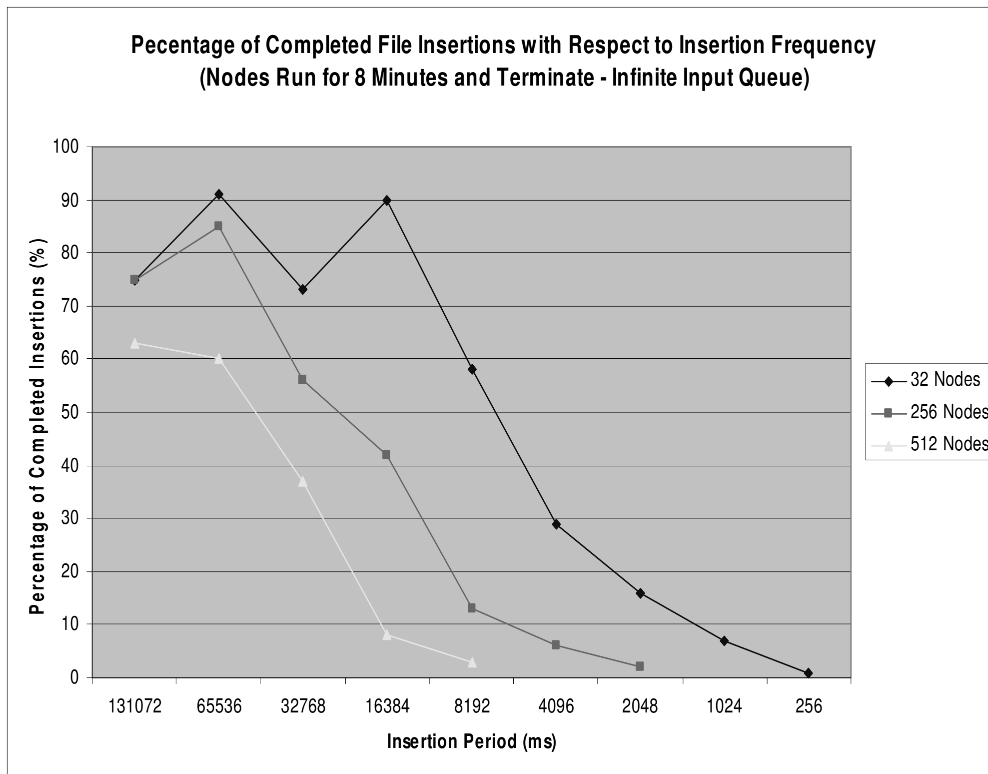

Figure 9

Next, we analyze insertion performance with increasingly larger networks, characterized by insertion delay time and percentage of successful operations. We observe in Figures 10 and 11 that in networks up to 256 nodes, insertions will take up to 5 seconds, but that as networks grow in size beyond 256, insertions take exponentially longer to complete, while the level of assurance

that the insertions will be successful drops exponentially in networks in excess of 128 nodes. Also, in the case where insert operations take place at a rate faster than about every 15 seconds, successful insertion is unreasonably low (below 60%), and becomes unreasonably low as networks grow from 128 nodes, even when inserts are taking place a minute apart.

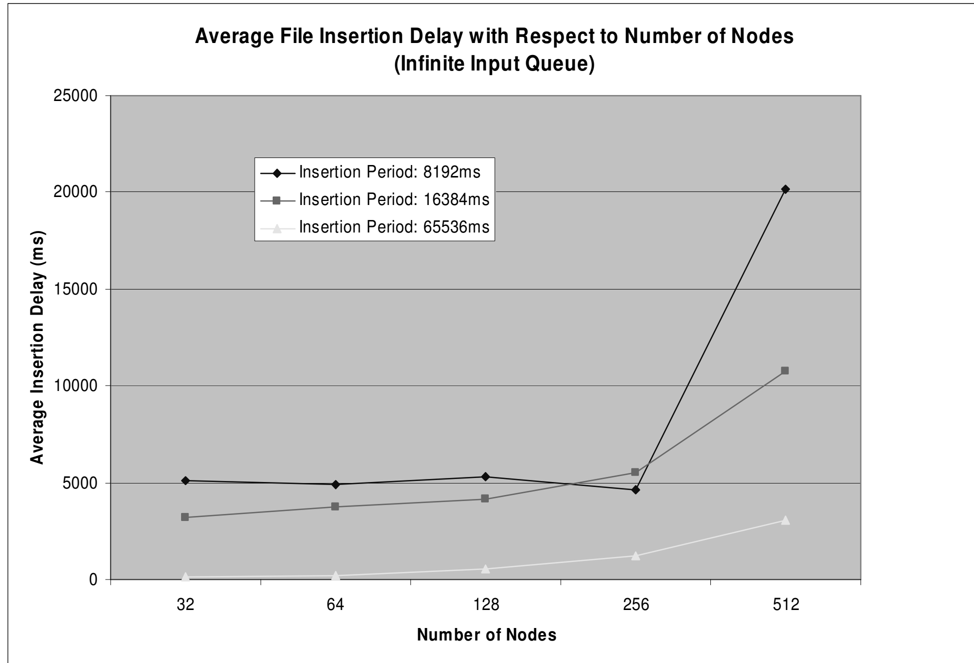

Figure 10

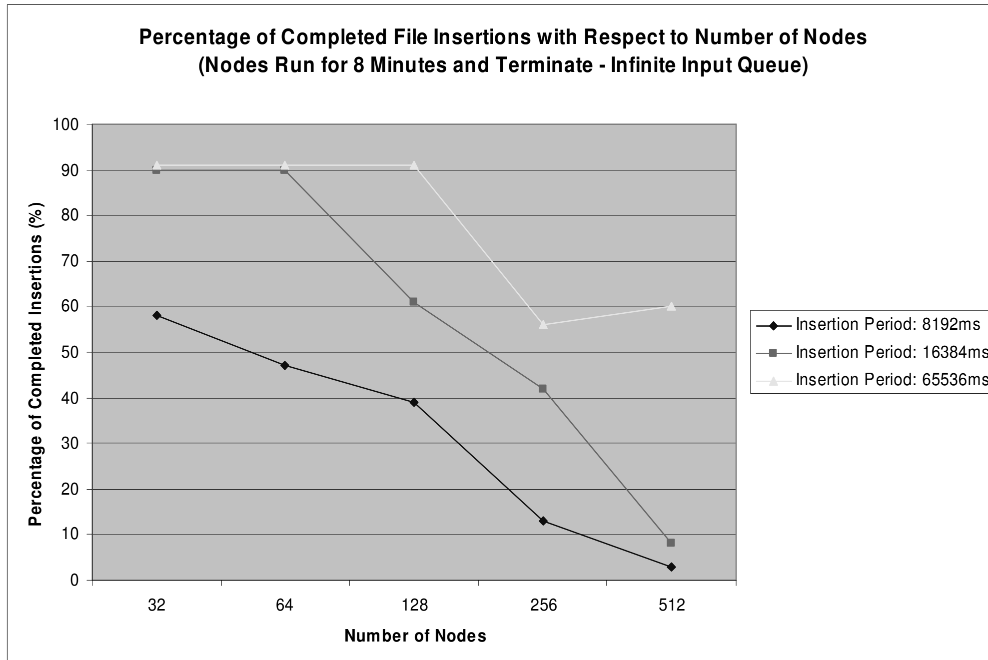

Figure 11

7.2 Query Operation Experiment

Our second experiment analyzes the performance of the network with N = 32, 512 nodes in querying for content with increasing frequency. In Figures 12 and 13, we observe that query delay in small networks remains consistent even when queries are made in quick succession, but in large networks, delay begins to increase exponentially as query frequency increases beyond 8 seconds. Query success remains reasonable with frequencies up to this point, but as the frequency increases past 8 seconds, query success decreases exponentially and becomes unreasonable at about 4 seconds (below 60%).

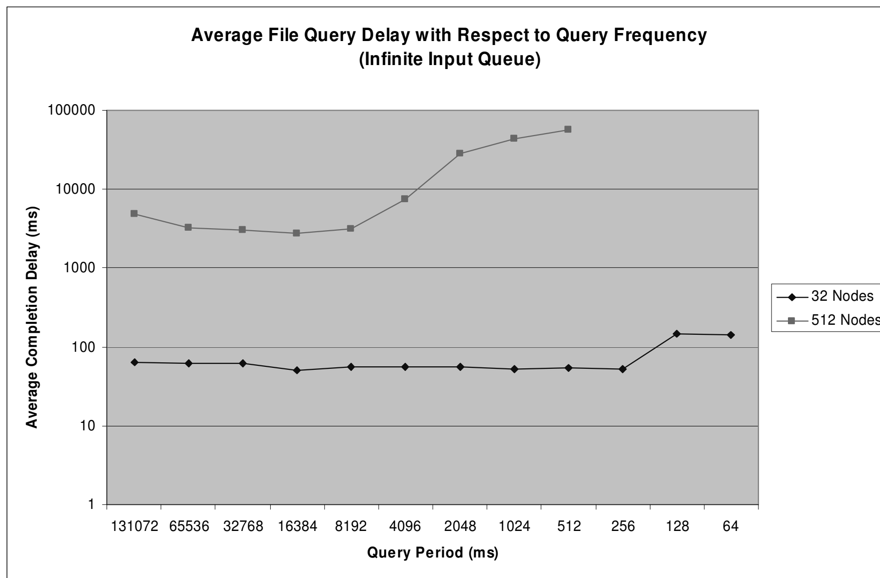

Figure 12

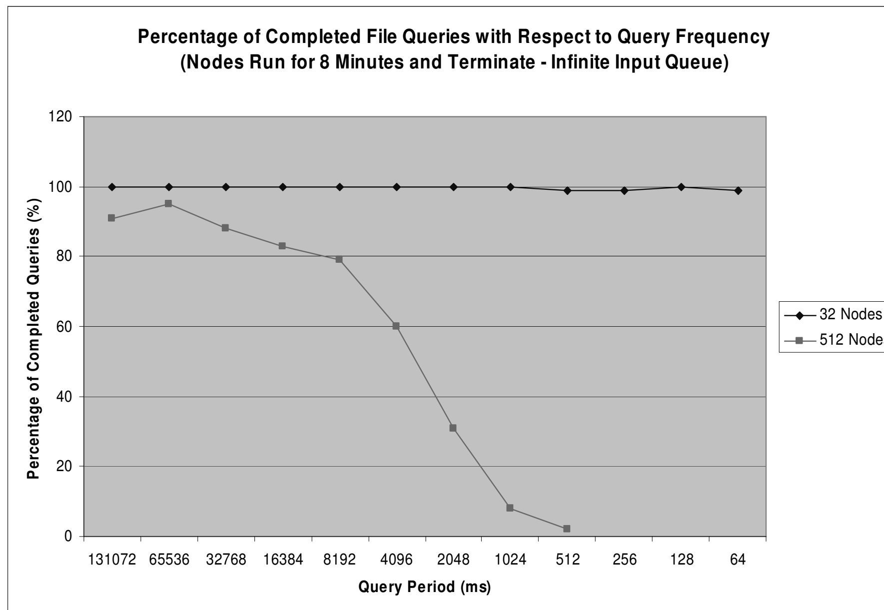

Figure 13

Next, we analyze the results from this experiment with increasing numbers of nodes in the network, characterized by query delay and successful query completion. We observe in Figures 14 and 15 that with query frequency of about 30 seconds, delay remains consistently reasonable even in large (N = 512) networks, and with a very short query frequency of about 0.5 seconds, query delay remains below 30 seconds in networks up to 256 nodes. A 30 second query period yields reasonable query success, while in networks exceeding 128 nodes, query success becomes unreasonable with a query frequency of 0.5 seconds.

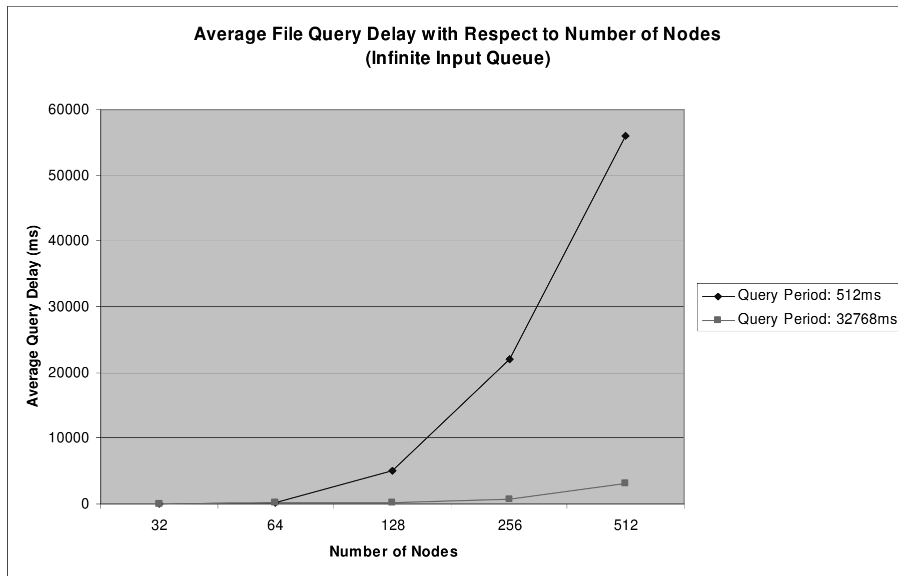

Figure 14

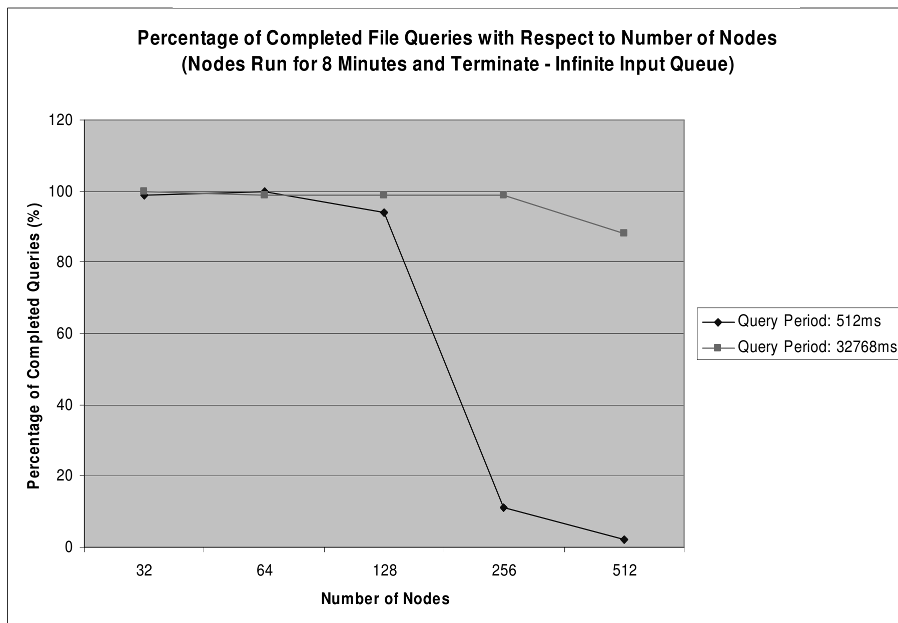

Figure 15

7.3 Variable inputQueue Size Experiment

In our final experiment, we analyze network performance in inserting content with increasing frequency while, varying the size of our inputQueue. We observe in Figures 16 and 17 that an unbounded queue begins to contribute to network congestion after insertion frequency reaches about 30 seconds. As this frequency increases, insertion delay grows exponentially with unbounded queues, but remains fairly stable when bounded at 1, 10 and 1,000 entries. Insertion success is reasonable as the insertion frequency approaches about 8 seconds when the queue length is bounded at 1,000, but with smaller queue sizes, insertion success becomes unreasonable after insertion frequency increases beyond about 30 seconds. From our experiments, we see that the optimal queue length appears to be about 1,000 while an unbounded queue is undesirable.

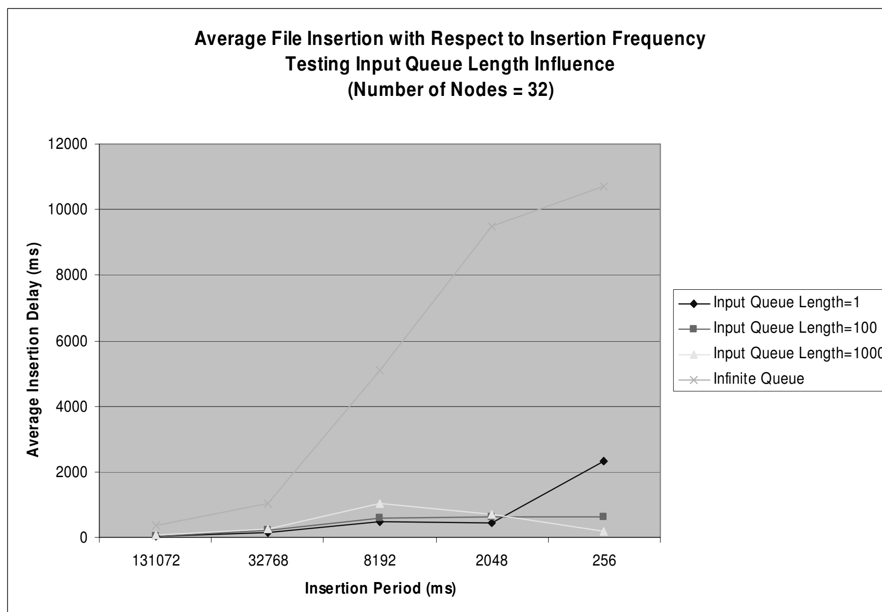

Figure 16

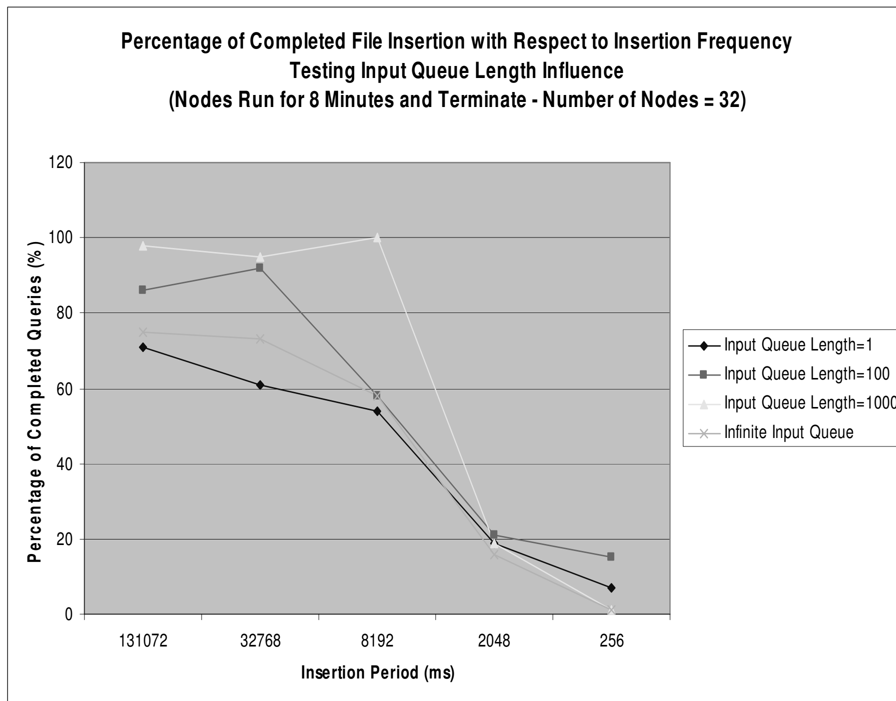

Figure 17

8. Conclusions

We have examined and implemented a DT protocol in order to create a distributed lookup service for an overlay network. Relative to previous work in the field, we have created a system that is robust and scalable, self-organizing and adaptive to state changes. Our design implements compass routing and incorporates the concept of Voronoi regions, as an extension of previous work. Our system employs a unique hashing algorithm to distribute key-value pairs in a 2-dimensional grid and allows users to search the key space for desired content.

With regard to other PTP applications, our system provides reasonable performance and scalability in terms of searching the key space for desired content, while separating the access to and distribution of content from the overlay network. We observe reasonable performance up to 256 nodes, but users may accept performance limitations manifested in insert operations more readily than those for query operations, where we observe less stringent limitations.

Bounding the queue length for processes yields desirable performance advantages over unbounded queues, since the network can become quite congested during periods of high usage. While our tests yield conclusive results for the implementation described herein, a more expansive testbed would be required to draw generalized conclusions pertaining to DT protocol usage in an overlay network.

9. Future Work

Our implementation, while straightforward in design, bears all of the complexity typically found in systems with distributed algorithms. As a result of this complexity, we have limited the functionality of our system in order to at least meet, if not exceed, minimal requirements. We feel that additional functionality and capability could be added to our system. The most promising improvements would be in:

- Remote replicas: In addition to replicating data to the neighboring nodes of an owner node, the data could be replicated to "remote" nodes, i.e. non-neighboring nodes. This could increase robustness in the presence of failures at the expense of increased network traffic on inserts.

- Replicas for load balancing: In our system replicas are not used for load balancing of query requests. We made this choice because a replica can never know when data has been deleted, so if a replica served queries, it could respond with invalid data. A possible extension of our design would allow for replicas to respond to queries, perhaps sometimes returning data that is not very stale would be useful. If the load balancing replicas are all neighbors of the owner node, query messages would have to arrive near the owner node in logical address space. If combined with "remote" replicas, the load balancing would be distributed across the logical address space.

- File Content Insertion: In our current system insertion operations merely maps a file name (hashed to a logical address) to a URL. We could augment the system to actually insert the contents of the file into the overlay network. In this way, the contents of the file would be replicated, not just it's location.

References

[1] I. Stoica, R. Morris, D. Karger, M. Frans Kaashoek and H. Balakrishnan. *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications* in Proc. *ACM SIGCOMM* August 2001, pp. 149-160.

[2] S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Shenker. *A Scalable Content Addressable Network* in Proc. *ACM SIGCOMM* Auagust 2001, pp. 161-172.

[3] J. Liebeherr, M. Nahas, W. Si. *Application-Layer Multicasting With Delaunay Triangulation Overlays*. In IEEE Journal on Selected Areas in Communications, Vol. 20, Oct. 2002, pp. 1472-1488.

[4] P. Raines and J. Tranter. *Tcl/Tk In a Nutshell*. O'Reilly & Assoc, Sebastopol, CA. 1999. p.146.