An Analysis of the Dtella Network at Purdue

by Paul Marks
CS536 Networking Project, Spring 2007

Abstract

In this paper, I perform measurements on the Dtella network at Purdue. I wrote Dtella over the last year, and it forms a pseudorandomly-organized mesh network which results in a wide broadcast domain for chat and file sharing purposes. From a single point on the Dtella network, I analyze the ping response times, neighbor link counts, broadcast packet hop counts, and 24-hour per-link bandwidth utilization. I conclude that the network is running within acceptable bounds, with the exception of a couple minor flukes.

A Description of the Dtella Network

Before my network measurements can be fully understood, it is first necessary to discuss the design of the network itself. Dtella is a program that I wrote, as a personal project, between the months of June 2006 and February 2007. Its purpose is to allow the operation of a Direct Connect (DC) network without the central point of failure inherent in its centralized hub design. DC is a popular system for creating chat and file sharing communities on the Internet or on private LANs, usually with user counts on the order of 1000. A standard DC network consists of a central hub which allows a group of users to see each other, establish 1-to-1 TCP connections for file transfer, send private and public (chat) messages, and search for files. Searching is implemented using a relatively simple method: a user broadcasts a search query which is forwarded by the hub to every other node. Then the nodes search their own file lists, and send replies back to the sender using direct UDP messages. Hence, like the Internet itself, most of the intelligence lies on the edges of the network.

Dtella is written in Python, and provides an emulated "DC Hub" which listens only on the loopback interface. A user runs one of the pre-existing and maturely developed DC clients (such as DC++) and establishes a hub connection to localhost at 127.0.0.1. But instead of there being an actual centralized hub, Dtella creates a decentralized P2P system in its place, which implements the necessary features provided by a hub, specifically presence, broadcast, and unicast. A node must know at all times which other nodes are on the network (presence), must be able to send search and chat messages to every other node (broadcast), and must be able to send a connection request or private message to a specific node (unicast).

In order to connect all the nodes together, I implemented a mesh network in which every node establishes 3 outbound links to other nodes. All communication is done over UDP, such that each node can be uniquely identified by its IP:Port combination. In order to maintain a stable mesh structure even in cases where the network is rapidly changing in size (churn), I used a pseudo-random algorithm for selecting neighbor links. Each active node has an 8-byte ID, consisting of the IP, Port, and a 2-byte random session number. In order to select neighbors, a node generates a pairwise score between itself and every other node, by concatenating its own ID with the remote ID, and running the result through MD5. The 3 nodes with the lowest MD5 results are then chosen as the outbound links. In order to make the scoring symmetrical (i.e. A:B produces the same score as B:A), the two ID values are always concatenated such the the smaller one comes first. This symmetry property, I assumed, would mean that many outbound links would also be chosen as inbound links by the remote node, as both sides will see a relatively small-numbered MD5 score. Thus, I expected that most nodes should not have to maintain a number of links which is significantly higher than 3.

In order to establish an initial connection to the network, I place an encrypted cache of long-running nodes on the network into a DNS TXT record which can be requested by the virgin nodes. Also, each node maintains its own persistent cache of addresses, which can be used in cases where the DNS system isn't available. Finally, it's also possible for a user to manually add the IP:Port of a known node to their local cache.

Every node on the network contains a short (~100 bytes) block of "info" which consists of a nickname, a freeform description, client version information, and amount of data shared by the client. When a new node joins the network, it must collect this info block from every other node in order to construct a complete user list. I solved this problem by using a rapid spidering technique. Because the system is UDP based, there is very little overhead involved in talking to many peers simultaneously. So, once a new node establishes a neighbor connection using the various caches described above, it starts building a hash table of all the active nodes, and continually asks every node for its info, as well as a list of its current neighbors. These new neighbors are also added to the table, and more requests are made, until eventually the table is exhausted and every node is known. In order to accelerate this process somewhat, the info requests are implemented using a limited-range broadcast of 3 hops through the network, such that a single request will solicit replies from approximately 10 nodes, instead of just 1. In practice, joining a Dtella network with 800 nodes usually takes less than ten seconds.

Dtella's primary network function is to create an application-level broadcast domain, so it was of course necessary to implement a means of broadcasting. I chose to value reliability and redundancy over bandwidth efficiency, as Dtella is meant to be run across high-speed LANs with many nodes disappearing randomly due to churn. Thus, to broadcast a message, it is simply forwarded across every neighbor link, excluding the links through which the message has already arrived in the reverse direction. Each broadcast message is given a unique ID, by means of MD5 hashing, which is kept in memory for a couple minutes so that the same message will never be forwarded twice by the same node. As an extra safety, each message also has a hop limit, initially 64, which is decremented at every hop through the network. If a hop limit ever does reach 0, it is discarded.

In order to maintain a consistent list of users, each node broadcasts its info into the network at an interval of approximately 15 minutes, plus a randomized flutter value. Each node listens for these broadcasts, and removes any nodes from its local list if they should fail to announce on time. This method is superficially similar to that used by routers in the OSPF protocol. In addition to these long term periodic broadcasts, each node sends a ping to all of its neighbors several times per minute. If a node detects that a neighbor has inexplicably stopped pinging, it will broadcast a failure message into the network, which will reduce that node's expiration time to 15 seconds. If the node then fails to refresh itself during this interval, it disappears. In practice, if you unplug the ethernet cable of an active node, it will be gone from the network within 30 seconds.

The remaining capabilities, chat, search, private messaging, and connection requests, are trivially implemented on top of the Dtella network. Chat and search are implemented as broadcast messages, and private messages and connection requests are sent as UDP directly to the destination address. This network overview glazes over many of the finer points which are less relevant to this measurement project, such as sequence numbers, packet acknowledgment and retransmission, encryption, bridging chat with an IRC network, and Dtella's IRC-centric predecessor, DCgate.

Related Work

I based Dtella's structure loosely on concepts I learned from Distributed Hash Table (DHT) networks such as Chord [5]. Both use randomized ID values as a means of maintaining a predictably random structure as the network grows.

Analysis and measurement of peer-to-peer networks in general has been a popular research topic in recent years, due to their explosion of popularity on the Internet. Reference [3] discusses taking measurements similar to what I've done, albeit in a much larger network in which it's not possible to determine the global state.

Analysis Procedure

When designing the Dtella system, I made some rough assumptions about the nature of the network and algorithms:

1. Being on a local network, the majority of nodes should have low-latency ping times.
2. The somewhat-symmetric neighbor selection algorithm should tend to produce nodes with a small number of neighbor connections.
3. The randomly organized structure of the network should mean that broadcast packets reach any destination without traversing a large number of hops.
4. The network isn't carrying much more than control information, so total bandwidth usage per link should stay relatively small.

My goal in this paper will be to test each of these assumptions, evaluate how well they were met, and, if necessary, propose possible changes which could result in a more healthy network. In order to collect data, I modified the Dtella source code to add some active and passive monitoring facilities which write their results to a text file. Then I wrote small Python programs to post-process this data, and finally fed it into gnuplot 4.2 for data visualization purposes.

Analysis: Ping Times

In order for the Dtella network to provide a good user experience when broadcasting chat and search queries, it is helpful for many of the nodes to have short ping times. I will define “short” to be under 100 milliseconds, but the definition is fairly arbitrary.

I added a piece of code to my Dtella client which gathers a list of all the online nodes, and sends an invalid connection request to each one, at a rate of 10 per second. Because the request is invalid, the remote node always replies with a rejection message. Effectively, this pair of messages can be used as a ping and pong. I grouped the response times by on-campus location and fed my data into gnuplot, producing Figure 1. This ping data was gathered on April 28, 2007, at approximately 8:00 pm, from my desktop located in Earhart hall.

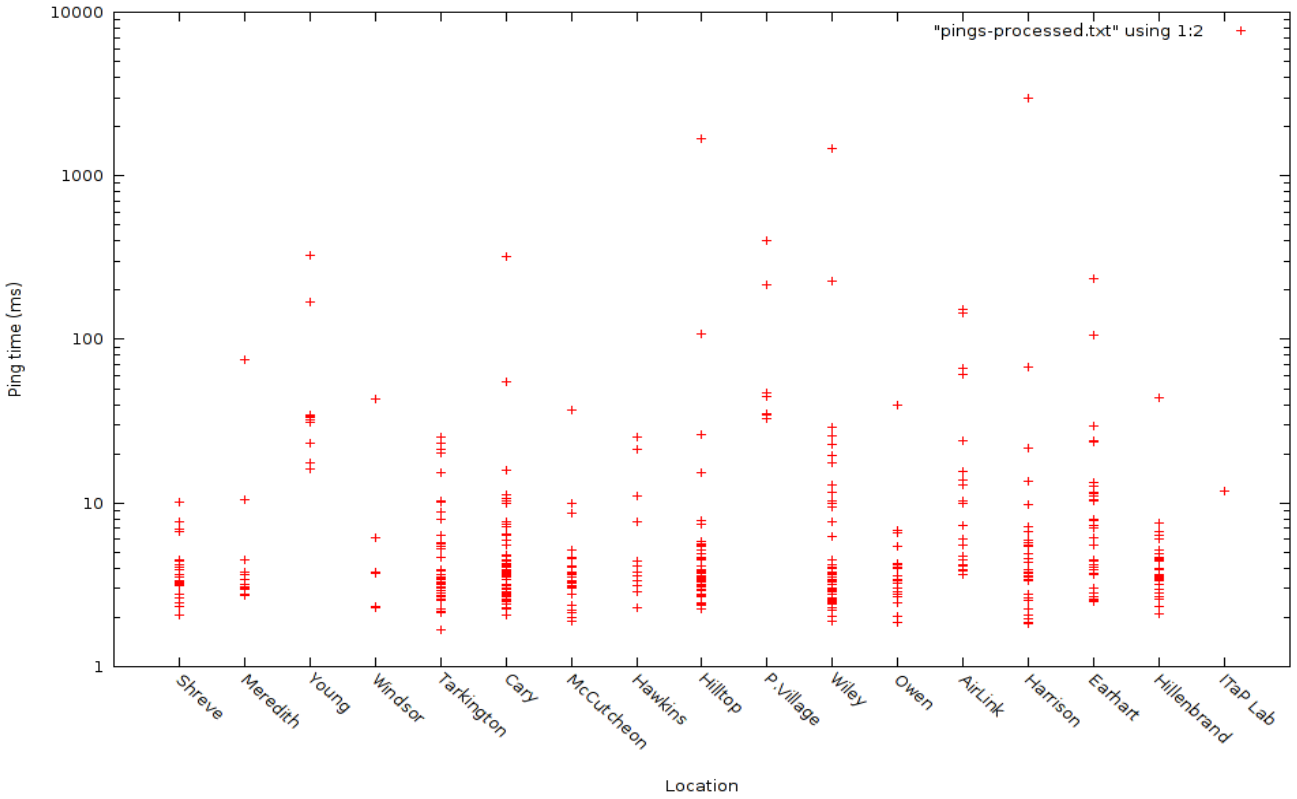


Figure 1. Ping response times on Dtella

It is immediately evident from Figure 1 that the vast majority of nodes have response times under 100ms. In fact, a large number of nodes reply in under 10ms. This is good news for the network, as it means that most messages will be forwarded quickly. The handful of outliers do not pose a significant threat to network stability, because messages will find faster paths around these nodes in most cases.

From Figure 1, though, it is also possible to gain some insight into the structure of Purdue's residential network. Notice that none of the nodes in Young Hall and Purdue Village have any response times below 10ms. These stand out from the rest of the network because their service is provided through ADSL instead of Ethernet. Also, I theorize that many of the higher-latency nodes in the Ethernet-based dorms are a result of people running laptops connected behind their personal wireless access points.

Analysis: Neighbor Counts

As I stated earlier, each node in the Dtella network establishes 3 outbound links with other nodes. The number of inbound links currently has no limit, but my assumption was that the pseudo-random algorithm with its slight preference toward symmetry would result in a network in which many nodes have only 3 links total, and none have significantly more than 6 links.

In order to gather this data, I modified the network sync/spidering algorithm to save a log of how many neighbors each node reports in its reply. After processing the data and feeding results into gnuplot, I produced Figure 2. This data was collected on April 28, 2007, at approximately 8:30 pm.

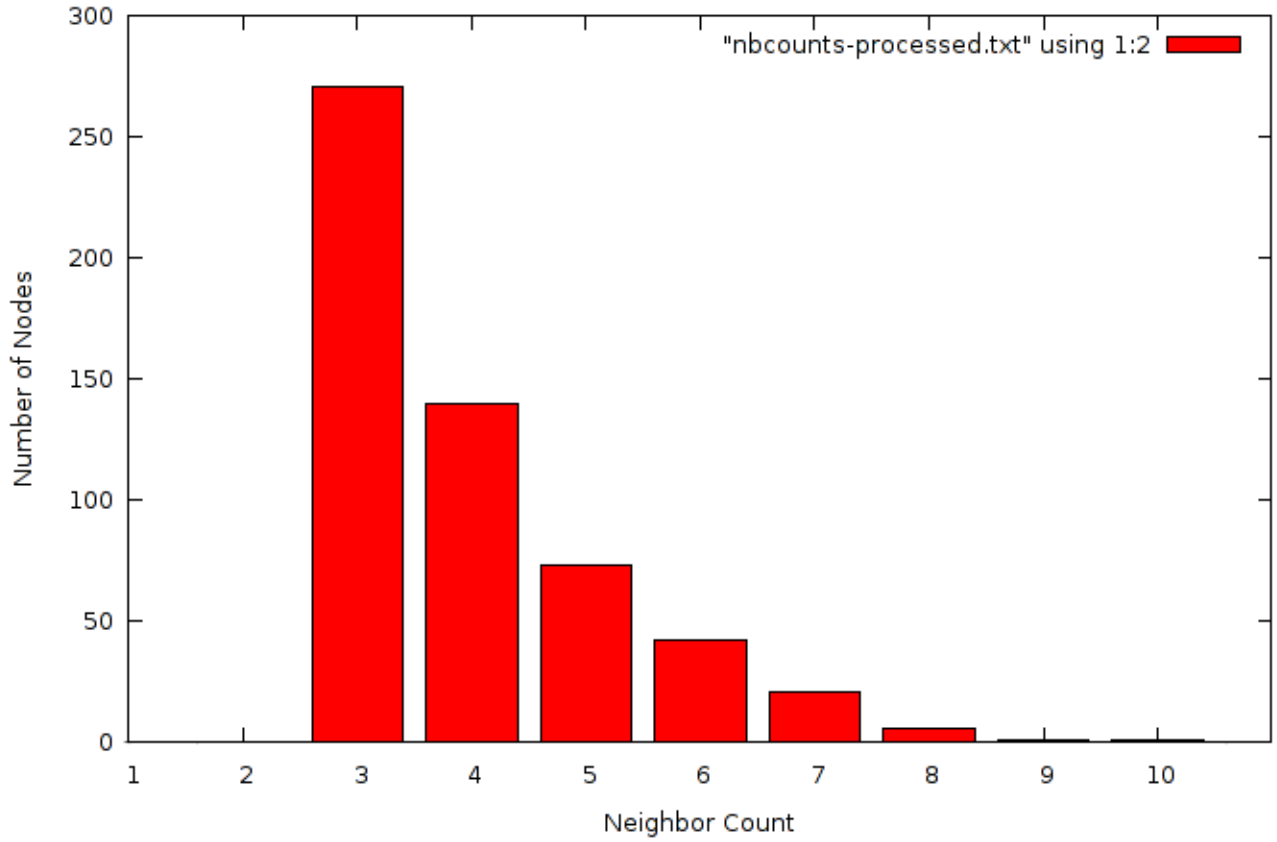


Figure 2. Distribution of neighbor link counts on Dtella

It is evident from Figure 2 that the majority of nodes have a neighbor count of 4 or less, which is good news. With each increasing number of links, the count seems to drop off exponentially. I am slightly concerned by the fact that a node exists with a hop count of 10. In previous runs, I even saw a count of 12. These counts may be unlikely, but in a large enough sample size, unlikely things happen all the time. Based on this evidence, it could potentially be a good idea to have nodes limit their number of total connections to a reasonable value, such as 8.

Analysis: Hop Counts

Broadcast messages on Dtella are routed through the mesh network using a simple flood algorithm. The hop limit of a message always begins at 64, and counts down each time it's forwarded. I found that I could monitor the hop limit value of incoming broadcast packets, and subtract the value from 65 to determine how many hops the packet had traversed before reaching my node.

First, I did some calculations to estimate how many hops I should expect on the network. If we have 800 nodes, and each time a packet is forwarded from a node, it reaches 3 new nodes, then the hop count for reaching all nodes is:

$$\log_3(800) = 6.08$$

Thus, we should expect packets to arrive in approximately 6 hops, given ideal conditions. I ran my hop counting code on April 26, 2007 at 4:00pm for a period of approximately 15 minutes. The results are

shown here in Figure 3.

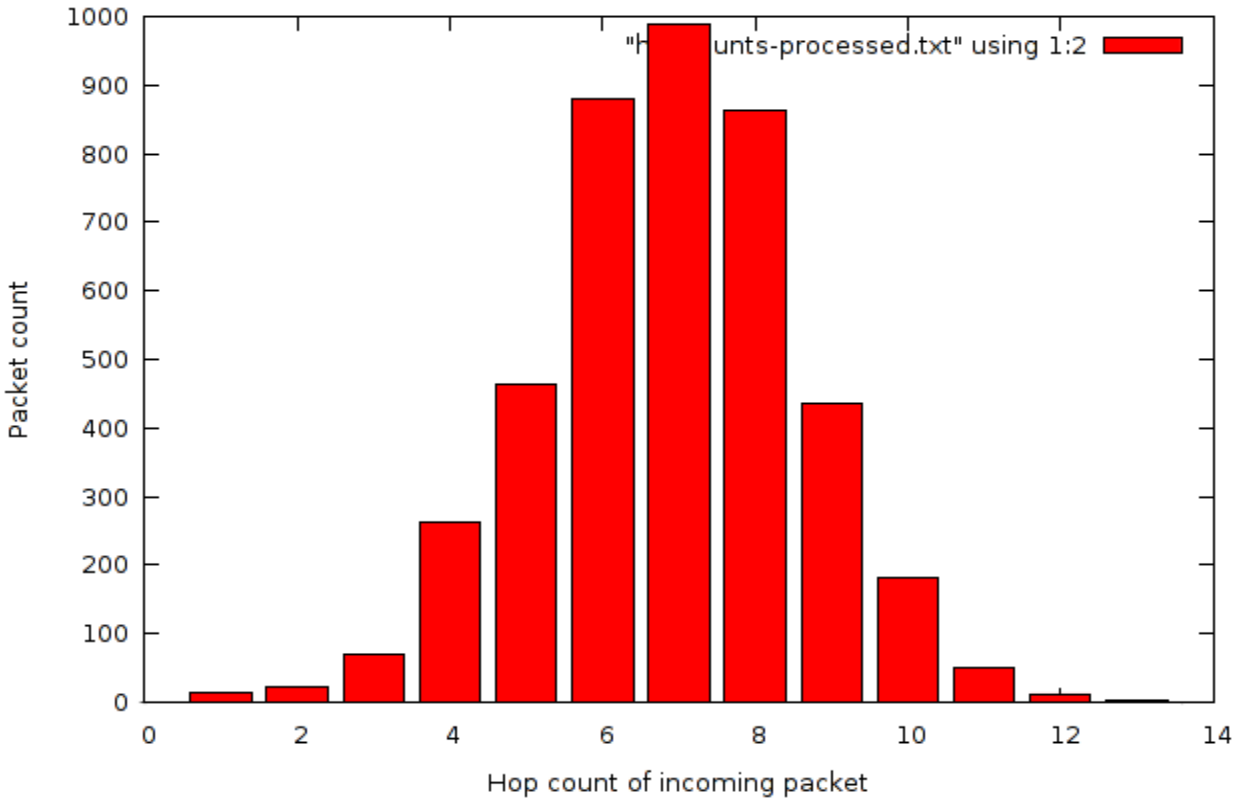


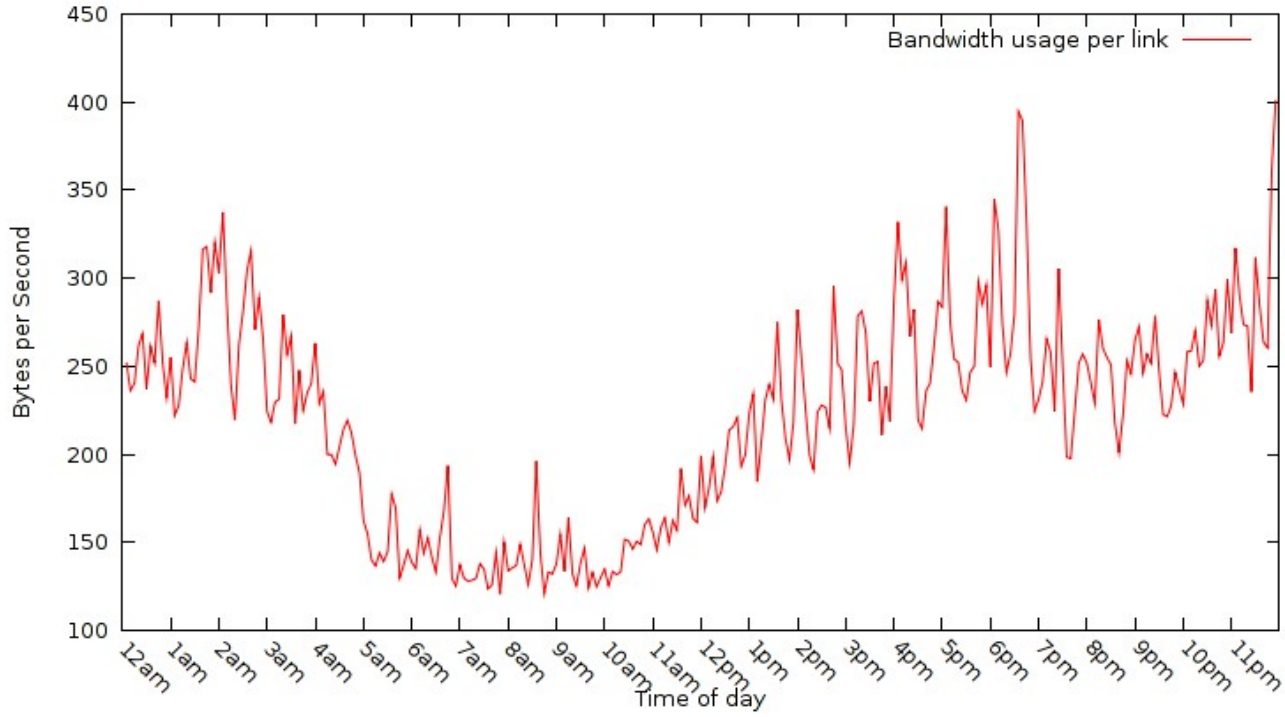
Figure 3. Broadcast packet hop counts on Dtella

As Figure 3 shows, the hop counts on this network follow a nearly perfect normal distribution curve centered at 7. This peak at 7 is very close to my estimate of 6, and the slight deviation is likely the result of the network being a randomly-connected mesh, and not a perfectly-balanced efficient tree. Still, if we consider a bad case of 10 hops, each at 100ms, then a message can still reach the entire network within 1 second. This is certainly well within the bounds necessary for a stable network.

Analysis: Bandwidth Usage

The node status broadcast interval for Dtella is tuned such that, in a network with no churn, a status broadcast (~100 bytes) should be received from another node at a rate of approximately once per second. When a broadcast covers the network, it will usually cross every neighbor link in one direction only. Thus, the baseline bandwidth usage should be around 100 bytes per second, per link.

In order to measure actual bandwidth usage, I attached hooks to the send and receive functions, which increment byte counters every time a packet is transferred. Every 10 seconds, the byte counters are flushed to a log file, and reset to zero. I ran this bandwidth logger for a period of exactly 24 hours, from 7:30pm on April 27, 2007 to 7:30pm on April 28, 2007. A post-processing utility in Python summed up the send and receive values, and produced a series of 5-minute average bandwidth usage values. These results are shown in Figure 4.



Bandwidth usage (per link) on Dtella

According to Figure 4, it's apparent that activity on Dtella dips to a minimum for the first few hours after sunrise (on a Saturday) when all the college students are asleep. At this time, my estimates for baseline activity are fairly accurate (100 B/sec vs 130 B/sec). During the times of higher activity, the network is filled with node enter/exit messages (caused by churn), as well as a flow of chat and search requests. This additional activity raises average bandwidth usage to approximately 300 Bytes/sec per link during peak times.

It's important to note, however, that these values are "per link," and we determined earlier that every node has at least 3 links, and some have 8 or more. If we consider a bad case of 8 links, each consuming 300 Bytes/sec, then we have 2.4kB/sec, which is a tiny fraction of a 100Mbit ethernet connection, although it is about 10% of the upload capacity of an ADSL line. I consider these values to be well within the bounds of reason.

Now, let's consider how much bandwidth is consumed across the entire LAN. We'll assume 800 nodes, with an average of 4 links per node, each using 300 B/s per link:

$$800 * 4 * 300 = 960{,}000$$

Thus, the total bandwidth usage for Dtella across the entire LAN is under 1 MByte per second. Considering that a single file transfer can consume 10MBytes per second, it is clear that the bandwidth usage of Dtella itself is insignificant when compared to the file transfer clients which attach to it.

Conclusions

The Dtella network has been operational since February, so I already knew that it was at least capable of holding itself together. But before now, I had never performed an experimental analysis of the detailed network characteristics. With the exception of some unfairness due to the possibility of high-

degree nodes, I would conclude that the network is running within acceptable bounds. Furthermore, this analysis confirms the viability of pseudorandomly-organized mesh networks when operating in a homogeneous low-latency environment.

Future Work

Dtella is an open source project, and it's likely that in the future people will establish Dtella networks which operate independently of the Purdue LAN. It could potentially be interesting to redo the measurements in this paper on a Dtella system which is operating across a more heterogeneous network, such as the public Internet. It would also be useful to place measurement points at several locations through the network, in order to determine the actual broadcast propagation time, and to verify that all broadcasts are in fact effectively covering the whole network.

References

[1] Guha, Daswani, Jain. An experimental study of the Skype Peer-to-Peer VoIP System. IPTPS 2006.
[2] Klemm, Lindermann, Vernon, Waldhorst. Characterizing the query behavior in peer-to-peer file sharing systems. ACM SIGCOMM 2004.
[3] Stutzbach, Rejaie, Duffield, Sen, Willinger. On unbiased sampling for unstructured peer-to-peer networks. ACM SIGCOMM 2006.
[4] Stutzbach, Rejaie. Understanding churn in peer-to-peer networks. Internet Measurement Conference 2006.
[5] Stoica, Morris, Liben-Nowell, Karger, Kaashoek, Dabek, Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. IEEE/ACM Transactions on Networking 2003.
[6] Dtella. http://www.dtella.org/ 2007
[7] DC++. http://dcplusplus.sf.net/ 2007