ENSC 835: HIGH-PERFORMANCE NETWORKS

Peer-to-Peer Networks as Content Distribution Networks

Fall 2003

FINAL PROJECT REPORT

André Dufour
www.sfu.ca/~adufour/ensc835/
adufour@sfu.ca

Abstract

This project examines the feasibility of using a peer-to-peer (P2P) network as a specialized form of content distribution network (CDN). In it, we consider a particular application of this: using the Gnutella protocol to distribute versioned content throughout the network. Since prompt delivery of updates to all hosts is the critical performance metric in this connection, we simulate several situations and observe the propagation of the content through the network, with special attention to the time it takes for all hosts to be updated.

Table of Contents

Introduction

Peer-to-peer (P2P) communication is emerging as one of the most potentially disruptive technologies in the networking sector[1]. If the degree to which users have adopted pioneering P2P applications such as Napster, ICQ and the host of available Gnutella clients is any indication, the importance of this new paradigm is not to be discounted. We have observed, for instance, that the FastTrack P2P network, which is used by the Kazaa file sharing application, regularly has more than 3,000,000 simultaneous users – a number far too large to be ignored.

For the most part, P2P networks have been used for file sharing, instant messaging and, less visibly, the aggregation of processing resources, as with SETI@Home. Few applications diverge from these apparent niches. Conversely, [2] proposes the use of P2P networks as an alternative to the client-server model for online gaming and shows that it is indeed feasible to use the technology in non-traditional ways. Further to that work, it occurred to us that it might be possible to use a peer-to-peer protocol to solve another problem where the client-server paradigm has traditionally dominated: content distribution.

Content distribution networks (CDNs) typically require large numbers of geographically dispersed servers in order to distribute the processing and bandwidth load and to shorten the distance between users and servers, thus reducing delays. The leading CDN operator, Akamai, operates 12,600 servers worldwide [3]. Clearly there is a large cost associated with deploying and maintaining such an infrastructure, not to mention the network provisioning costs. The $15,000/month/Mbps [3] cost for customers is also quite significant. Thus, there may be some cases where it would be appropriate to replace this type of expensive CDN with a P2P network and handoff the processing and bandwidth demands to the end hosts.

In this project, we evaluate the feasibility of distributing versioned content through the Gnutella P2P network. One particular example of versioned content is an anti-virus signature file. Typically, anti-virus vendors such as Symantec (Norton Antivirus) and Network Associates (McAfee Antivirus) operate CDNs of some sort in order to distribute updates for their customers' virus signature databases. Given their large install base, these CDNs must be fairly substantial and therefore expensive to maintain. It therefore seems relevant to examine the possibility of replacing them with a P2P network where the end hosts query each other for the latest version of content.

We have two primary objectives in this project. Firstly, we wish to determine if it is possible to use the Gnutella P2P network as a CDN for versioned content. If content is introduced into the network and manages to propagate to all hosts that are interested in it, we deem the CDN operation to be successful. If some hosts are not able to download the content, we consider the operation a failure. We examine several different network topologies with this question in mind. Our second objective is to quantify how well successful topologies act as CDNs. Our metric for this evaluation is a function of the time required for all interested hosts to obtain a new version of the content. We used Network Simulator, version 2 (ns-2) to implement our simulation.

In the first part of this report, we provide a brief introduction to peer-to-peer networks and content distribution networks. We then discuss how P2P networks could be used to replace CDNs in our particular scenario. Next, we provide details about our implementation. Finally, we provide results and conclusions.

1. Technology Survey

1.1. Peer-to-Peer Networks

Before discussing the particulars of P2P networks as they relate to this project, it seems appropriate to define more rigorously what we mean by peer-to-peer. The following definition, proposed on OpenP2P.com, accurately captures the idea:

> *P2P is a class of applications that takes advantage of resources -- storage, cycles, content, human presence -- available at the edges of the Internet*
> *[...]*
> *If you're looking for a litmus test for P2P, this is it:*
> *1) Does it treat variable connectivity and temporary network addresses as the norm, and*
> *2) does it give the nodes at the edges of the network significant autonomy?*[4]

In short, we believe that the essence of P2P networking is to aggregate the resources (services, processing power, files) of nodes at the edge of the network. This is in sharp contrast to the classical client-server paradigm where edge nodes have a much more limited participation in network activities than servers do.

P2P networks offer many advantages over the client-server paradigm. Owing to their distributed nature, P2P networks are naturally more fault tolerant than a central server-based approach. Indeed, if a server (or server farm, for that matter) becomes unavailable due to network congestion or equipment failure, the resources it is providing will be unavailable for the duration of the outage. Because resources – files for instance – in P2P networks are distributed on multiple geographically dispersed hosts, localized network and equipment failures cannot easily prevent access to a resource. Also, since P2P technology leverages existing network resources to create an overlay network, it doesn’t require additional capital expenditures or maintenance. One can easily see why this would be an attractive alternative to maintaining costly servers.

One issue with P2P networks is that, once again, because of their decentralized architecture, they are difficult for a single authority to control. We have seen that it is very challenging to prevent piracy on P2P networks such as Gnutella and FastTrack, where terabytes[1] of copyrighted music are routinely available for illegal download. Conversely, some P2P technologies such as Freenet leverage this "feature" to enable free speech without fear of censorship:

> *"To achieve this freedom, the network is entirely decentralized and publishers and consumers of information are anonymous. Without anonymity there can never be true freedom of speech, and without decentralization the network will be vulnerable to attack*[5].*"*

P2P networking technology has been used extensively for file sharing applications. BearShare, LimeWire, Napster, Morpheus and Kazaa are just a few examples. Indeed, the popularity of Napster, as well as its infamous and ultimately unsuccessful legal battle with the Recording Industry Association of America (RIAA), were clearly instrumental in fueling interest in peer-to-peer networks. P2P is also a popular paradigm for instant messaging clients such as ICQ or MSN Messenger. [6] lists dozens of projects and companies that are related to P2P in one way or another. Evidently P2P technologies are gaining popularity and are finding applications in new domains.

1.2. Content Distribution Networks

Because the concept of content distribution networks is well known and well understood, we will not describe it in great detail here. Nevertheless, we highlight a few key characteristics that are relevant to our project.

CDNs are essentially distributed servers, placed throughout the lower-tier internet service providers' networks [7]. They each host a copy of the CDN's customers' content, effectively replicating it in geographically dispersed locations. The intent is to distribute the processing and bandwidth load to multiple locations. This reduces the chance of

[1] Number obtained through personal observation

network bottlenecks forming about a single location and also prevents a single equipment failure from blocking access to the distributed content. CDNs typically also improve performance by directing users to a server located close to their geographical area.

1.3. Peer-to-Peer Networks as Content Distribution Networks: a Natural Progression

As we can see, P2P networks and CDNs are naturally similar. Both attempt to mitigate potential equipment failures by relying on the distribution of resources. They also rely on decentralization to share the processing and network load amongst many hosts or servers.

P2P networks are not directly suitable for use as CDNs because of their loose control structure. Indeed, because no central authority can control the dissemination of content through the network, there must be a built-in mechanism to ensure that the content is genuine and authentic (i.e. that it came from the expected provider). This could easily be achieved by embedding a secret encrypted key into the packages and having the clients verify its presence before deeming the content to be authentic. Such considerations, however, are beyond the scope of this project.

In summary, it seems that P2P networks, with slight modifications, may be suitable as CDNs. Given all the advantages of P2P technologies over the server-based approach employed by CDNs, this is a highly encouraging finding.

It is also worth clarifying what we mean by the distribution of versioned content in the CDN. Versioned content is any type of resource where it is possible to ask: “Does any host in the network have a version of this resource that is higher than mine?” Some examples of versioned content are anti-virus updates, operating system patches and online journals (versioned by date of publication, for instance).

1.4. The Gnutella Network

Gnutella is a P2P protocol used for distributed file sharing. This project simulates a Gnutella network based on version 0.4 of the protocol.

Hosts participating in the Gnutella network are termed *servents* (server + client). Servents wishing to connect to the Gnutella network must obtain the IP address and Gnutella port number of at least one other servent. This is generally done by querying a well known host cache server, although this is not mandated by the protocol [9]. Once the servent is connected to the network, it will try to establish Gnutella connections with other peers until it has filled all its available connections slots, as configured by the Gnutella client. It learns the addresses of servents through interaction with the servents it is already connected to. All the connection messages are in ASCII text format, following syntactical and semantic rules similar to HTTP [9]. Servents also send so-called *ping* messages to probe the network for additional servents to connect to. *Pong* messages are sent in response to pings and contain addresses of additional peers in the network. This connection establishment phase can take several minutes. [8] suggests a possible improvement to the protocol aimed at reducing this connection delay. The network formed by Gnutella 0.4 has a flat hierarchy of peers, unlike the two-tiered network developed by Gnutella 0.6 and subsequent versions.

Once two servents are connected, they exchange binary messages carried over TCP links [9]. These messages are known as descriptors. Each one carries a 16-byte globally unique id as well as a number of other fields that help manage descriptor propagation in the network. The following table, adapted from [9] describes the key fields in Gnutella descriptors.

Field	**Byte Offset**	**Description**
Descriptor ID	0..15	Globally unique identifier for this descriptor. Servents need to know if they are dealing with duplicate messages in some circumstances and this field is used for that purpose.
Payload Descriptor	16	Identifies the type of message this descriptor is carrying: 0x00 = Ping 0x01 = Pong 0x80 = Query 0x81 = QueryHit 0x40 = Push
Time to live (TTL)	17	Number of hops left before this message expires. TTL is decremented each time a servent forwards the descriptor. When TTL = 0, the message is dropped.
Hops	18	Number of servents this message has crossed. This is incremented each time a servent forwards the descriptor.
Payload Length	19..22	Number of bytes in the message payload

As mentioned before, ping and pong messages are used during the connection phase, to locate and report servents available for connection. Query descriptors are used to seek out content in the network using keyword searches. QueryHit descriptors are sent in response to queries in order to report the availability of the requested content at a given servent. Push messages are used to circumvent firewalls that don't allow incoming connections.

When a servent receives a query hit in response to a query it sent, it initiates a direct download from the responding servent. This is done by sending an HTTP GET request similar to the following:

> *GET /get/<File Index>/<File Name>/ HTTP/1.0\r\n*
> *User-Agent: Gnutella/0.4\r\n* (3)
> *Range: bytes=<Start Offset>-\r\n*
> *Connection: Keep-Alive\r\n*
> *\r\n* **[9]**

Only the Query and QueryHit messages are of interest in this project.

Query messages have the following format [9]:

Field	**Byte Offset**	**Description**
Minimum Speed	0..1	Minimum speed in kb/s of servents that should respond to this message.
Search Criteria String	2..N	A space separated list of keywords being searched for.
Nul Terminator	N + 1	The byte 0x00. Nul termination for the search string.
Optional Query Data	N + 2.. L – 1	Reserved for future protocol extentions.

QueryHit descriptors have the following format [9]:

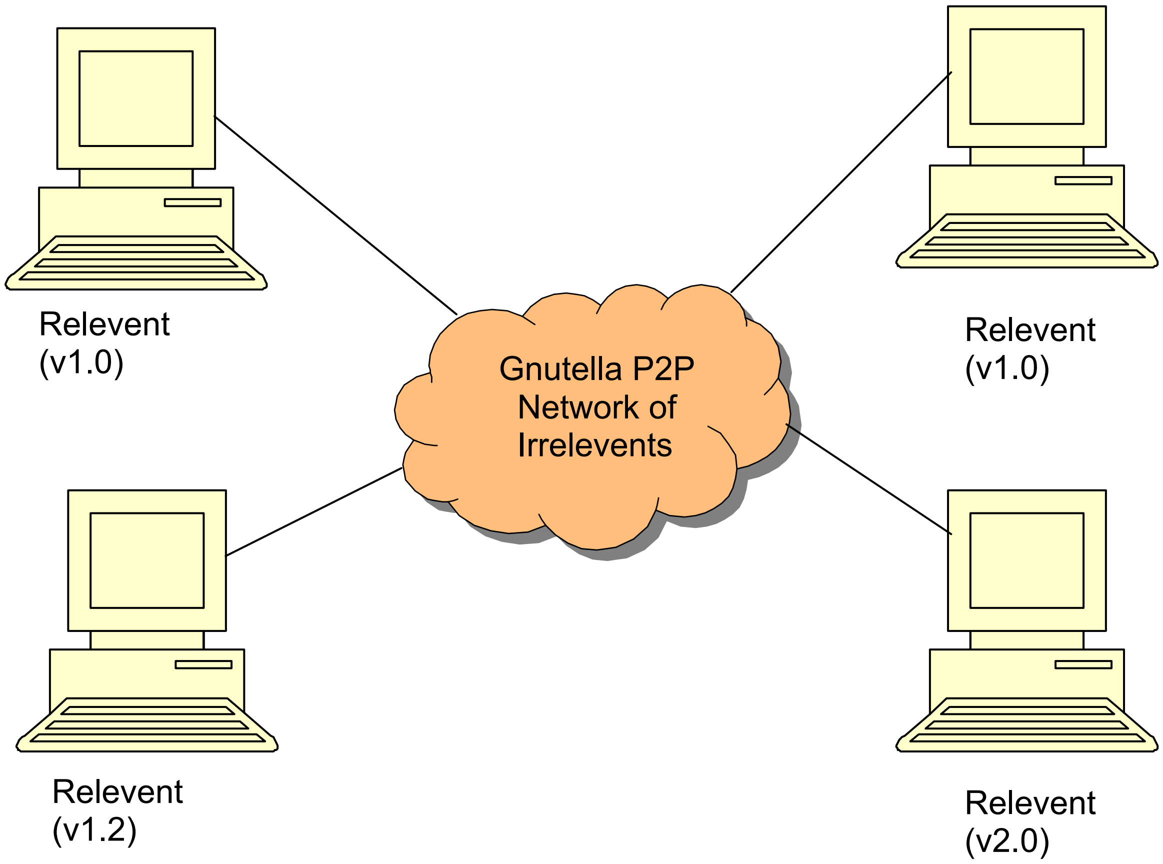
Field	**Byte Offset**	**Description**
Number of Hits	0	The number of matches returned in this message
Port	1..2	The TCP port on which the responding servent is willing to accept incoming connections (i.e. the expected HTTP GET).
Speed	7..10	The maximum upload speed in kb/s of the responding servent.
Result Set	11..10+N	Contains *Number of Hits* successful search results corresponding to the queried data.
Servent Identifier	L – 16.. L – 1	Identifier unique to the responding servent. Required for *push* operations.

2. Implementation

2.1. Objective

In order to facilitate the discussion, we introduce some terminology. Recalling that all nodes in the Gnutella network are servents, we coin the term *relevent* to designate servents that are interested in the versioned content we are considering. This does not necessarily imply that they have the latest version, but they are interested in acquiring it. By extension, *irrelevents* are hosts that don't have an interest in our versioned content. These are just regular Gnutella servents that could be sharing MP3 files, for example. While they don't want to download the content we are considering, the Gnutella protocol requires them to perform Query/QueryHit forwarding. Thus, they will facilitate the propagation of our versioned content even though they don't wish to acquire it for themselves.

Essentially we wish to simulate the situation depicted in the following figure:



As we can see, in this figure, four relevents are connected to the Gnutella network and an unknown number of irrelevents are also connected. This number is assumed to be much higher than the number of relevents. Two of the relevents have version 1.0 of the content, one of them has version 1.2 and one of them has version 2.0. This project aims to examine the propagation behaviour of the content through the network. In particular, we would like to know how long it takes for all relevents to be updated to version 2.0 (the highest version in the network) of the versioned resource.

2.2. Simplifications and Scope

Given the timeframe for this project, it was necessary to make some simplifying assumptions. One of the most important simplifications is that we do not consider how the overlay (i.e. P2P) network is established. Indeed, this rather complex process was the focus of another project in a previous iteration of this course [8]. We ask the question: "Given that the network has converged to some configuration, what is the behaviour now?", rather than actually simulating the connection phase, which is of little interest to us in this investigation. We do, however ensure that the network we create conforms to the Gnutella specification in terms of the maximum number of connections a host is allowed to have. Like in the previous project, we neglect physical network topology, as it is inconsequential to our analysis; for simplicity's sake, chose to mirror Gnutella connections with direct physical connections between the peers. The physical topology of the network is fairly inconsequential to our analysis, but having the links mirror the Gnutella connections makes it easier to visualize the network's behaviour in ns-2's network animator (NAM).

In addition to neglecting the connection phase, we also chose not to implement the HTTP download mechanism used for the actual transfer of content between peers. This would have been a fairly trivial addition to the project, but it would not add much value to our analysis. Indeed, this simple download can be considered a fixed-cost in acquiring updates and is therefore not helpful in comparing the *relative* performance of different network topologies. We therefore consider that once a relevent receives a QueryHit indicating that a higher version is available, it has acquired that version.

We chose to limit our implementation to include only the fields of interest in the Query and QueryHit descriptors. Essentially, this is the search string and the result set. Furthermore, since the servents are only searching for version numbers, we chose to store the search string and result set as integers in the descriptor objects. Also, since each relevent only has one version of the content (the latest one it downloaded), each QueryHit's result set will only contain that one version.

The Gnutella specification [9] states that if a node receives a message containing an identifier it has seen before, it must drop that message and not propagate it further. This requires nodes to keep track of all the identifiers of messages that pass through them, leading to an ever increasing list of IDs. One approach to keep this list manageable is to periodically flush it. Doing this in our simulation would have required the use of timers and introduced a fair amount of complexity and computational overhead. We chose to store path information in the descriptors instead of message IDs in the servents: the descriptors keep a record of all nodes they passed through. Servents inspect this path and don't forward Queries to nodes that have already been visited. This is a slight deviation from the protocol, but it made the implementation far simpler because descriptors have a finite lifetime (their TTL) and therefore disappear naturally, unlike ID lists in servents.

As an additional simplification, we chose to use the Gnutella 0.4 network model [9], which describes a flat network hierarchy. A more scalable two-tiered model is advocated in Gnutella 0.6 [10], but implementing it would have diverted too much time from our principal focus. Furthermore, we were not interested in studying Gnutella for its own sake, but rather as an example of a P2P network. The simple, flat topology was therefore more amenable to our application.

One more important simplification is that we don't consider the connection state of the P2P network to be transient. We act as if the topology that it converged to is the steady state topology. We know this to be untrue because nodes enter and leave the network frequently. However, between the time a relevent to enters the network and the time it gets updated to the latest version, it is reasonable to assume that the connections are *fairly* stable.

Finally, we wish to underscore that fact that since we cannot simulate realistic network topologies with thousands of hosts, it may not be appropriate to generalize our results and infer the behaviour of the real Gnutella network as a CDN[11]. Nevertheless, we can get an idea of the performance on a smaller scale.

2.3. Working Principle

This section describes the flow of our simulation. The details about the various components interaction are presented in the following sections.

The characteristics of the simulation are entirely specified by the parameters in a global configuration TCL file. These parameters are:

- Number of nodes: the total number of servents in the simulation (relevents + irrelevents).
- Average number of links per servent: the average number of peers each servent is exchanging Gnutella messages with.
- Maximum number of links per servent: the upper bound to the number of directly connected peers. Required in order to ensure that our simulation respects the Gnutella protocol, which does constrain this number [9].
- Probability that a servent is a relevent: controls the density of relevents in the network.
- Time between queries: both minimum and maximum values between successive queries sent out by relevents. The time is uniformly distributed between these two values.
- Version update intervals: how often new versions of the content are introduced into the network.
- Topology: speed and type links.
- RNG seed: specified in order to make simulations reproducible.

Our implementation essentially functions as follows:

1. We create a ring topology, ensuring we have a connected graph such that all nodes are reachable.
2. Given the average number of links per servent and the number of nodes, the total number of Gnutella connections is calculated.

3. For each bidirectional connection, two nodes with available connection slots (as determined by the specified maximum number of connections) are selected at random and connected (both physically and with TCP/Gnutella links).
4. One relevent in the network is seeded with the initial content version.
5. Relevents periodically (but not all at the same time) send out Gnutella queries, searching for content with a version number greater than theirs. Irrelevents don't query, but they do forward queries, since they are Gnutella servents. Relevents reply to Queries with QueryHit messages if they can offer a higher version of the content. They also forward Queries and QueryHits if their TTL is not elapsed.
6. From time to time, the network is seeded with a newer version of the content.
7. We then observe the propagation behaviour of the update and see how long it takes for it to reach all relevents. This is done by using a statistics object, which monitors how many relevents have received the latest version of the content at each second.

2.4. Object Implementation Details

This section presents the components involved in the simulation and describes their functionality as well as their relationship with one another. The code for each of these classes is presented in Appendix 1.

<u>GnutDescriptor</u>

Class type: C++, Abstract Base Class

Description: This is the base class for all Gnutella application layer messages exchanged in the simulation. It is derived from ns-2's AppData class, which allows the transfer of objects through the TcpApp's *send*() method. This class encapsulates the functionality common to all Gnutella descriptors and stores the message ID, hops, TTL and the list of nodes traversed.

<u>GnutQueryData</u>

Class type: C++, Concrete

Description: This is a subclass of GnutDescriptor. It specializes by keeping information related to the query string (just an integer version number) as well as the size in bytes of the message. This size is used by ns-2 to calculate the time required to transmit the message over a given link.

<u>GnutQueryHitData</u>

Class type: C++, Concrete

Description: This is a subclass of GnutDescriptor. It specializes by keeping information related to the result set (simply an integer version number) as well as the size in bytes of the message.

<u>GnutPathList</u>

Class type: C++, Concrete

Description: This class contains a list of all the node numbers traversed by a message. Each GnutDescriptor object contains one instance of this class. As a query is forwarded, each node it traverses adds its number to this list. When servents receive a Query, they search its GnutPathList and only forward it to nodes that are not in the list. This path list is also essential to the proper routing of QueryHit messages. The Gnutella protocol specification [9] requires that QueryHits be back propagated along the path taken by the Query that triggered them. Thus, the GnutPathList is used as a stack: the nodes are pushed onto the stack as a Query traverses them and popped (in the reverse order) as a QueryHit makes it way back to the originating servent.

GnutAgent

Class type: C++/oTcl split object, Concrete Base Class

Description: This is a subclass of ns-2's Process class. It represents irrelevents in our simulation, i.e. regular Gnutella servents. It implements all the Query and QueryHit forwarding logic, but does not itself generate Queries. It contains an associative map of node numbers and ns-2 *TcpApp*s. Given a servent ID (equivalent to a node number in our simulation), the map will return the TcpApp object that is handling the connection to that servent. It was necessary to use Process as the base class rather than Application because Applications can only support a single connection.

GnutAgents register a callback method with each of their TcpApps in order to receive any data coming in on their links. Once they get the data, they apply Gnutella's protocol logic to take the appropriate action. If the descriptor's TTL is not zero, the GnutAgent will examine the object's GnutPathList. After incrementing the hop count and decrementing the TTL, the GnutAgent will forward the message to all its peers that don't appear in the GnutPathList. If the TTL is zero, the message is effectively consumed and not propagated any further.

The GnutAgent is also responsible for updating the NAM display to indicate the last action it took. This part of the implementation is done in oTcl because of that language's natural affinity with NAM.

ReleventAgent

Class type: C++/oTcl split object, Concrete

Description: This is a subclass of GnutAgent. In addition to inheriting all of the forwarding logic provided by its superclass, ReleventAgent implements the functionality for sending Queries and QueryHits, as well as the capability to update the version of the content it has.

Like GnutAgents, ReleventAgents register a callback method with all their TcpApps. When they receive data, they determine if the message is a Query or a QueryHit. If it is a Query, they examine the query string and check if their version of the content is higher than the one in the Query. If it is, they will construct a new GnutQueryHitData object based on the GnutQueryData object received and send it back through the same link it was received from. The ReleventAgent will then delegate to its superclass to make Query forwarding decisions. If the received message is a QueryHit, the ReleventAgent will examine the GnutPathList object it contains to see if it is the intended recipient of the hit. If it is, it will check the version in the result set and update to that version if it is higher than its own. It will also consume the QueryHit and not propagate it further. At that point, the ReleventAgent will inform the Stats object that it has updated to a new version so that the simulation can keep track of how many nodes have been updated. If, on the other hand, the QueryHit was intended for another relevent, the ReleventAgent will invoke the base class, GnutAgent, to apply QueryHit forwarding logic.

ReleventAgents periodically flood all their links with Queries, searching for content with a version higher than their own. To do this, they construct a GnutQueryData object with their version as a query string. They then send that object to all their peers using the TcpApps' *send*() method.

Some of the ReleventAgent's functionality is implemented in oTcl. In particular, the functions that relate to updating the NAM display to reflect the last action taken by the servent are written in oTcl. Also, the timer which triggers the periodic queries is armed in Tcl as this is the only way to interact with the NS simulator.

<u>Stats</u>

Class type: oTcl, Concrete Singleton

Description: This object is responsible for recording the results of the simulation. The test harness notifies it when it introduces a new version of the content into the network. When relevents update to that version, they also inform the Stats object. Every second, the Stats object tallies how many relevents have not been updated to the latest version and records this to a log file. This file is the result set we will analyze.

3. Simulation Results

3.1. Analysis Framework

Before presenting our simulation results, it seems appropriate to introduce a framework for analyzing them. We simulated a number of different topologies, differing only by their connectivity parameters (average and maximum number of links per node) and the prevalence of relevents in the network. We call the prevalence of relevents the *density* of the network. A dense network has a high proportion of relevents and sparse networks have relatively few relevents with respect to the total number of servents.

In order to answer the first important question we set out to examine – can P2P networks be used as CDNs? – we look at the number of unupdated relevents as a function of time. Each time a new version of the content is introduced into the network, that is to say that one relevent is seeded with it, the number of relevents not updated should be the total number of relevents less one. If the network is acting successfully as a CDN, this number should drop to zero over time. If the steady state result is non-zero, some relevents were not able to get the latest version and the scenario is a failure.

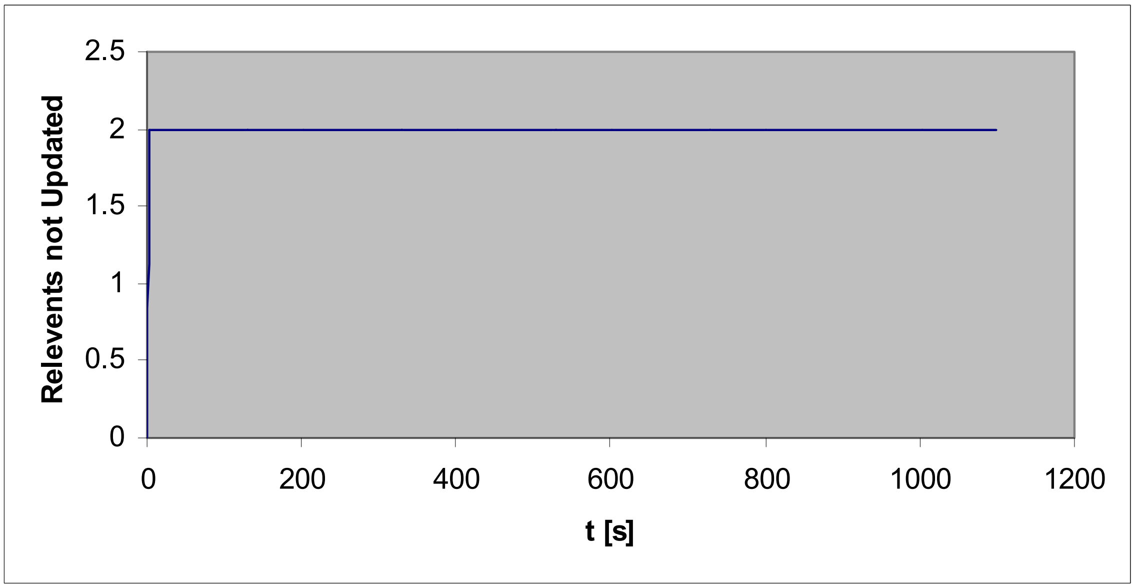
The second issue we wished to investigate was how well the different topologies functioned as CDNs. We hoped to infer some relationship between the varying parameters – connectivity and density – and the performance of the network. This performance is a function of the time elapsed from the moment an update is introduced into the network until the last relevent is updated (propagation time). Furthermore, the value of our metric should be inversely proportional to the number of relevents. This arises from the fact that if one scenario has a large number of relevents and another has a smaller number of relevents, but they both require the same propagation time, the network with fewer nodes is exhibiting poorer relative performance. The last point to consider in the derivation of our objective function is the fact that it should be an average over several trials within each scenario. A trial is the measurement of the propagation time after each introduction of a new version. Thus, our metric, the *normalized update time*, is defined as follows:

$$U = \frac{1}{T}\sum_{j=1}^{T}\frac{t_{p_j}}{n},$$

where T is the number of trials, t_{p_j} is the propagation time at the j^th^ trial and n is the number of relevents in the network. We plot U as a function of the connectivity and density parameters in order to examine their effect on network performance.

3.2. Results

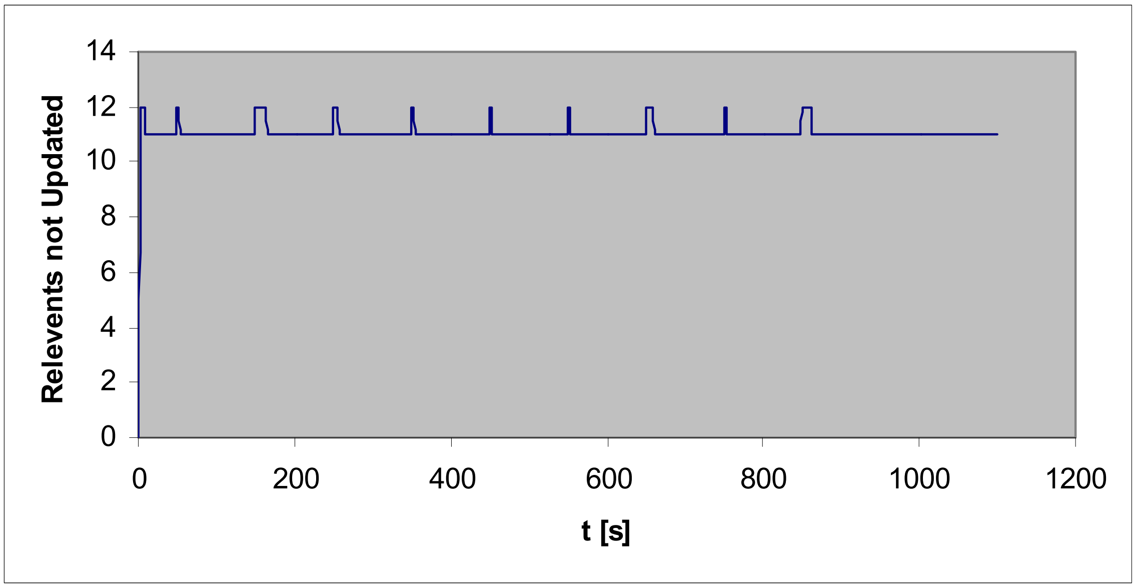
Figure 3.2-1: Weakly Connected, Sparse Network
avg conns = 2, max cons = 2, 5% Relevents



As we can see in figure 3.2-1, for a weekly connected, sparse network, the steady state value is 2, revealing that two relevents are never able to update to the latest version over time. Thus, this is a scenario where P2P networks don’t work as CDNs.

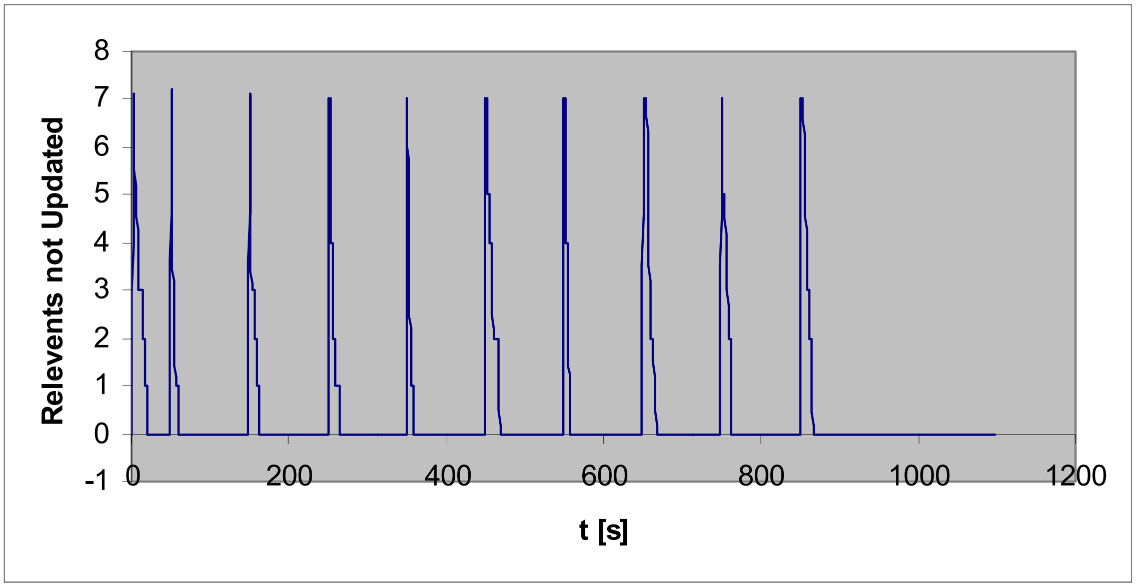
The figure 3.2-2 shows another failure scenario with the same connectivity parameters, but a larger proportion of relevents. We can see that 11 relevents are never able to update to the latest version of the content.

Figure 3.2-2: Weakly Connected, Medium Sparse Network
avg conns = 2, max cons = 2, 25% Relevents



If the density of the network is further increased to 50% relevents, thus forming a very dense network, we observe that all relevents are eventually able to obtain the latest version of the content. This situation is depicted in figure 3.2-3. The width of the spikes correspond to the t_{p_j} values in the formula for *U*.

Figure 3.2-3: Weakly Connected, Medium Sparse Network
avg conns = 2, max cons = 2, 50% Relevents



While this type of graph is convenient for visually seeing the number of relevents not updated dropping to zero, it does not lend itself well to quantifying the performance of the various network topologies we examined. The previous figures were provided only to illustrate the difference between the failure cases and an example of success. The following figures in this section show the normalized update time *U* as a function of the connectivity and density parameters for all cases where $U < \infty$, that is to say all successful cases.

Figure 3.2-4 shows that *U* seems to decrease quasi-linearly as the average connectivity of the relevents is increased. We indeed expected to see that *U* would decrease as relevents were in contact with more and more peers, thus increasing their chances of being close to a relevent with the desired content. This experiment was carried out using a network where 40% of the nodes were relevents and the maximum number of connections for any given node was 8.

Figure 3.2-4: Normalized Update Time vs Average Number of Connections

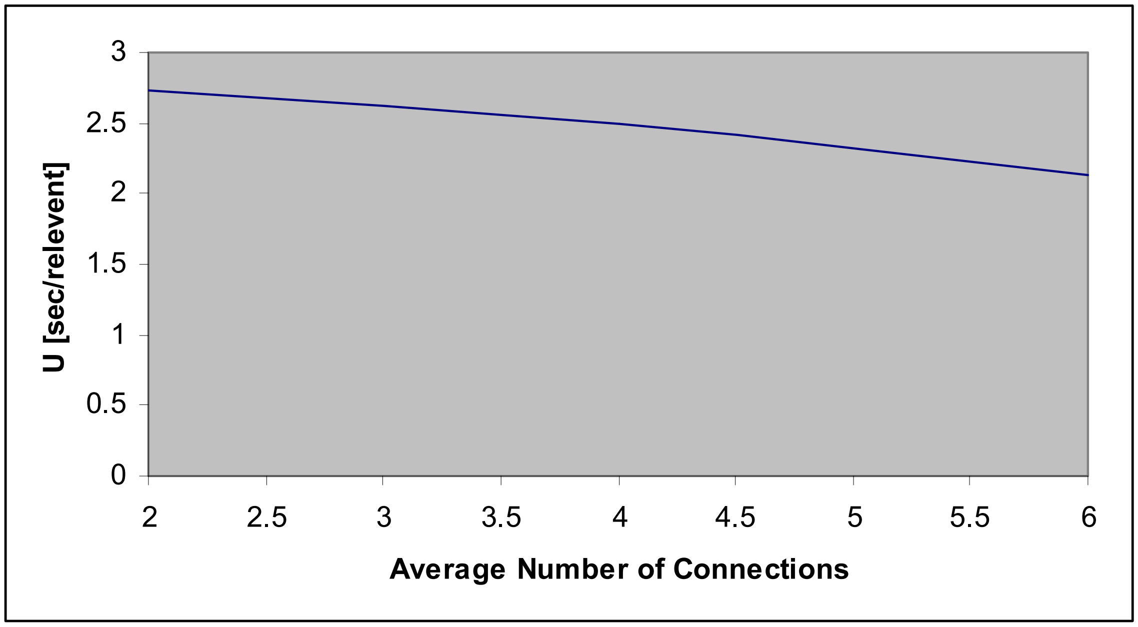
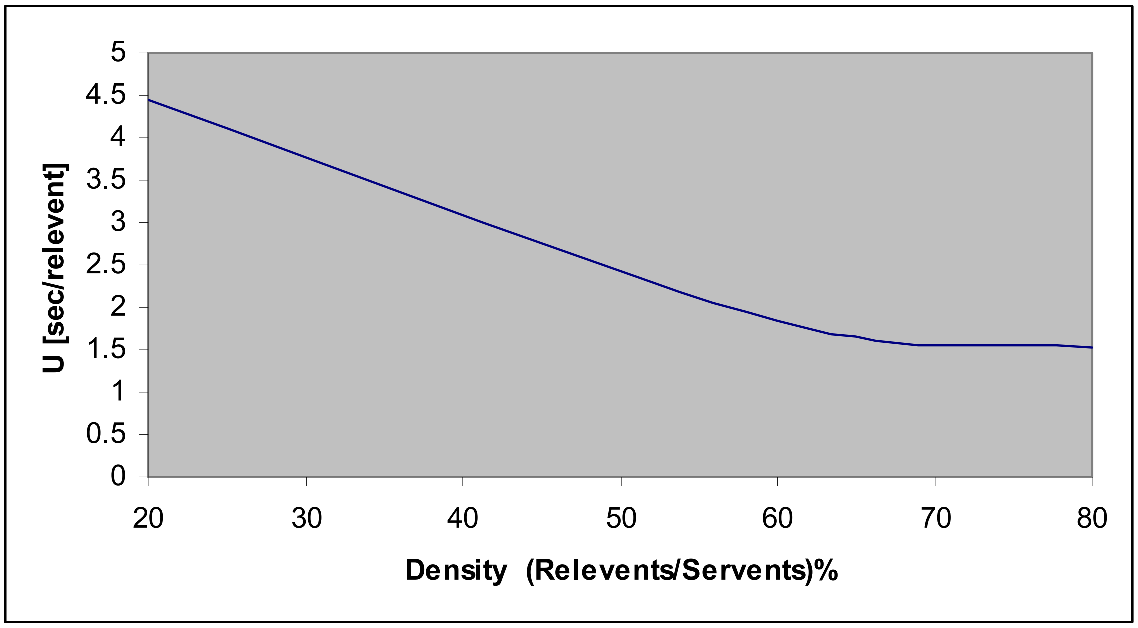


Figure 3.2-5 shows the dependency of *U* on the density of the network. We can see that as the proportion of relevents increases, the normalized update time decreases sharply. We note, however, that once the density reaches a certain threshold, just below 70%, we observe little improvement in *U* as the density increases further. Presumably all relevents are close enough to each other that adding further ones does not significantly improve the situation. Nevertheless, the decreasing trend is as expected. This experiment was carried out by holding the average number of connections of each servent fixed at 3 and the maximum number of connections at 8.

Figure 3.2-5: Normalized Update Time vs Network Density



It is very important to note that the simulations carried out in order to obtain the results presented here were very CPU and memory intensive. Thus, it was necessary to restrict them to a small number of nodes. In all cases, 50 nodes were used. Even with that small number, the simulations required several hours to run, no doubt due to the large number of connections involved and the complex message processing. As a result, the number of sample points on the graphs presented here is small and we may therefore be mislead as to the real trend. Furthermore, because of the small number of nodes, it is difficult to make claims as to the performance of real-sized networks. Indeed, the Gnutella network is larger than our simulation by a factor of 1000 and it is doubtful that our implementation accurately captures the complex behaviour involved in the real network. Nevertheless, our results are pleasing in that they agree with the intuitive notions that the update time should decrease as connectivity and network density increase.

4. Discussion and Conclusions

In this project, we set out to examine the possibility of using P2P networks as content delivery networks for versioned content. We wished to determine if the idea was feasible as well as obtain a rough idea of how certain network parameters influence performance. We conducted simulations using a 50-node Gnutella network with different connectivity and density values and observed that the normalized update time decreased as the connectivity and density of the network increased. We also noted that once connectivity reached a certain threshold, performance stopped improving as the average number of connections per servent increased.

Gnutella is a fairly complex protocol and one of the major challenges in this project was to make coherent simplifications that facilitated the implementation yet still captured the essence of the protocol. Another important challenge was designing a suitable metric for evaluating the performance of the various network topologies. Furthermore, it was a non-trivial task to capture all the network parameters we wished to vary in our test harness. All these issues notwithstanding, the most difficult part of the project was simply implementing the Gnutella protocol in ns-2.

There are many opportunities for further work and improvement to this project. Its most significant shortcoming is the small scale simulation upon which we based our analysis. Although we will most likely never be able to simulate Internet-sized networks [11], we can still hope to simulate networks with a few hundred or perhaps a few thousand nodes. It is likely that our code can be optimized and be run on a more powerful computer in order to make such simulations realistic. Also, assuming simulations can be run faster, it would be desirable to sample a broader range within our chosen parameters – and indeed additional parameters as well – and observe their effect on the propagation behaviour of versioned content. Also, for the sake of completeness, it would be appropriate to integrate this project with past work related to the Gnutella P2P network, such as [8], as well as to implement the HTTP download phase. Our project gives a glimpse of the behaviour of P2P networks as CDNs, but to get a more complete picture, further work is in order.

5. References

[1] A. El Saddik, and A. Dufour, "*Peer-to-Peer Suitability for Collaborative Multiplayer Games*," In Proceedings of the Seventh IEEE International Symposium on Distributed Simulation and Real Time Applications, Delft, Netherland 25th - 28th Oct. 2003.

[2] A. El Saddik, and A. Dufour, "*Peer-to-Peer Communication through the Design and Implementation of Xiangqi*," In Proceedings of the International Conference on Parallel and Distributed Computing, Klagenfurt, Austria 26th - 29th August 2003.

[3] CDN shortlist (printable version):
http://www.hostingtech.com/connectivity/02_06_cdn_print.html,
(accessed October 28, 2003).

[4] What is P2P… and What Isn't [Nov. 24, 2000]:
http://www.openp2p.com/pub/a/p2p/2000/11/24/shirky1-whatisp2p.html?page=2,
(accessed October 27, 2003).

[5] The Freenet Project:
http://www.freenetproject.org/,
(accessed October 27, 2003).

[6] OpenP2P.com P2P Directory Listings by Category [Jun. 09, 2003]:
http://www.openp2p.com/pub/q/p2p_category,
(accessed October 28, 2003).

[7] J. Kurose, and K. Ross, "*Computer Networking: A Top Down Approach Featuring the Internet*", 2nd edition. Addison-Wesley, July 2002.

[8] A. Fouron, and B. Fraser, "*Dynamic Connection Creation in the Gnutella Network,*" Graduate Work, Simon Fraser University, 2003.

[9] Gnutella – Stable – 0.4:
http://rfc-gnutella.sourceforge.net/developer/stable/index.html,
(accessed October 10, 2003).

[10] Gnutella Protocol Development:
http://rfc-gnutella.sourceforge.net/developer/testing/index.html,
(accessed October 10, 2003).

[11] S. Floyd and V. Paxson, ``Difficulties in simulating the Internet," *IEEE/ACM Transactions on Networking*, vol. 9, no. 4, pp. 392 - 403, August 2001.

Appendix 1: Code Listing

```
###############################################################################
# File: gnut_globals.tcl
# Author: Andre Dufour
#
# Contains configuration options for the simulation. This is the only file
# That should be changed in order to modify the parameters of the simulation.
###############################################################################

# Maximum number of connections any node can have
set max_conns       10

# Average number of connections a node has
set avg_conns        8

# Number of nodes in the simulation
set nnodes          10

# Seed for the random number generator. Specifying this
# ensures that is it possible to reproduce simulations.
set seed            26
set rng [new RNG]
$rng seed $seed

# Percentage probability that a node is a "relevent"
set relprob         40

# Minimum and maximum delay in seconds between queries for a given node.
set min_query_delay  1
set max_query_delay 20

# Physical link characteristics
set link_speed "100Mb"
set link_delay "10ms"
set link_queue "DropTail"

# Validation of options
if {$avg_conns > $max_conns} {
    error "avg_conns must be smaller than or equal to max_conns"
}

if {$min_query_delay > $max_query_delay} {
    error "min_query_delay must be smaller than or equal to max_query_delay"
}

# Calculated globals
set nconns [expr ($avg_conns * $nnodes / 2) - $nnodes]
```

```
##############################################################################
# File: real_topology.tcl
# Author: Andre Dufour
#
# Core of the simulation.
##############################################################################
#import Global constants
source gnut_globals.tcl

#Misc globals
set agentnum 0
set cur_flow_id 0

#Only keep the headers we need; conserve memory
remove-all-packet-headers
add-packet-header ARP IP TCP

#Create a simulator object
set ns [new Simulator]

# Setup routing - distance vector
$ns rtproto DV

#Open the nam trace file
#set nf [open real.nam w]
#$ns namtrace-all $nf

#Clear out the old stats file
file delete "stats.out"

##############################################################################
#Define a 'finish' procedure
proc finish {} {
#        global ns nf
#        $ns flush-trace
        #Close the trace file
#        close $nf
        #Execute nam on the trace file
#        exec nam real.nam &
        exit 0
}
##############################################################################
#Get a random node
proc getRndNodeIdx { maxIdx } {
    global rng
    return [expr round(floor([$rng uniform 0 $maxIdx]))]
}
##############################################################################
proc startqueries {} {
    global rellist gnut_apps
    for {set i 0} {$i < [llength $rellist]} {incr i} {
        $gnut_apps([lindex $rellist $i]) startQueries
    }
}
##############################################################################
proc stopqueries {} {
    global rellist gnut_apps
    for {set i 0} {$i < [llength $rellist]} {incr i} {
        $gnut_apps([lindex $rellist $i]) stopQueries
    }
}
##############################################################################
proc connectNodes { n1 n2 } {
    global n ns agentnum agent conns cur_flow_id conn_list tcp_apps gnut_apps
    global link_speed link_delay link_queue

    #Physically connect the two nodes
    $ns duplex-link $n($n1) $n($n2) $link_speed $link_delay $link_queue

    #Create a TCP agent for each of the nodes participating in the connection
    set agent_1 $agentnum
    set agent_2 [expr $agentnum + 1]
    set agent($agent_1) [new Agent/TCP/FullTcp]
    $ns attach-agent $n($n1) $agent($agent_1)
    incr agentnum
    set agent($agent_2) [new Agent/TCP/FullTcp]
    $ns attach-agent $n($n2) $agent($agent_2)
    incr agentnum
    $agent($agent_1) set fid_ $cur_flow_id
    $agent($agent_2) set fid_ $cur_flow_id
    $ns color $cur_flow_id red
    incr cur_flow_id
    $ns connect $agent($agent_1) $agent($agent_2)
    $agent($agent_1) listen
    $agent($agent_2) listen

    #Create some TCP apps and connect them too
    set tcp_apps($agent_1) [new Application/TcpApp $agent($agent_1)]
    set tcp_apps($agent_2) [new Application/TcpApp $agent($agent_2)]

    $tcp_apps($agent_1) connect $tcp_apps($agent_2)

    #Now, tell the gnut apps about which TcpApps are theirs and which node
    #they are connected to.
    $gnut_apps($n1) gnutconnect $n2 $tcp_apps($agent_1)
    $gnut_apps($n2) gnutconnect $n1 $tcp_apps($agent_2)

    incr conns($n1)
    incr conns($n2)
    lappend conn_list($n1) $n2
    lappend conn_list($n2) $n1
}
##############################################################################
# Main

set conns_avail [list]

# Create the nodes and GnutApps
for {set i 0} {$i < $nnodes} {incr i} {
    set n($i) [$ns node]

    #Create the gnut app. Can be either a relevent or an irrelevent
    if {[expr round(floor([$rng uniform 0 100]))] <= $relprob} {
        # it's a relevent
        set gnut_apps($i) [new Relevent $ns $n($i)]
        lappend rellist $i
    } else {
        # it's an irrelevent
        set gnut_apps($i) [new Gnut $ns $n($i)]
    }

    $gnut_apps($i) setnodenumber $i
    set conns($i) 0
    set conn_list($i) $i
    $n($i) label "[$gnut_apps($i) getversion]"
    lappend conns_avail $i
}

# First, create a ring topology to ensure that we have a single connected
# graph.
for {set i 0} {$i < [expr $nnodes - 1]} {incr i} {
   connectNodes $i [expr $i + 1]
}
connectNodes 0 [expr $nnodes - 1]

# Now, while there are still connections available, select two nodes
# that have room for new connections at random and connect them.
for {set i 0} {($i < $nconns) && ([llength $conns_avail] > 1)} {incr i} {
    set valid_conn 0
    while {$valid_conn == 0} {
        set n1 [lindex $conns_avail [getRndNodeIdx [llength $conns_avail]]]
        set n2 [lindex $conns_avail [getRndNodeIdx [llength $conns_avail]]]

        if {($n1 != $n2) && ([lsearch $conn_list($n1) $n2] == -1)} {
            connectNodes $n1 $n2
            set valid_conn 1
        }
    }

    if {$conns($n1) == $max_conns} {
        # n1 has reached its max number of conns. Remove it from the
        # available list
        set delidx [lsearch $conns_avail $n1]
        set conns_avail [lreplace $conns_avail $delidx $delidx]
    }

    if {$conns($n2) == $max_conns} {
        # n2 has reached its max number of conns. Remove it from the
        # available list
        set delidx [lsearch $conns_avail $n2]
        set conns_avail [lreplace $conns_avail $delidx $delidx]
    }
}

set stats [new Stats $ns [llength $rellist]]
for {set i 0} {$i < [llength $rellist]} {incr i} {
    $gnut_apps([lindex $rellist $i]) setStats $stats
}


# Kick off the simulation
$ns at 1.0   "$stats timerTick"
$ns at 1.0   "startqueries"
$ns at 2.0   "$stats newVersion 18"
$ns at 2.0   "$gnut_apps([lindex $rellist 0]) setversion 18"
$ns at 50.0  "$stats newVersion 19"
$ns at 50.0  "$gnut_apps([lindex $rellist 0]) setversion 19"
$ns at 150.0  "$stats newVersion 20"
$ns at 150.0  "$gnut_apps([lindex $rellist 0]) setversion 20"
$ns at 250.0  "$stats newVersion 21"
$ns at 250.0  "$gnut_apps([lindex $rellist 0]) setversion 21"
$ns at 350.0  "$stats newVersion 22"
$ns at 350.0  "$gnut_apps([lindex $rellist 0]) setversion 22"
$ns at 450.0  "$stats newVersion 23"
$ns at 450.0  "$gnut_apps([lindex $rellist 0]) setversion 23"
$ns at 550.0 "stopqueries"
$ns at 650.0 "finish"

#Run the simulation
$ns run
```

```
/*
 * File: gnutagent.h
 * Author: Andre Dufour
 *
 * Description: Header file for the Gnutella servents that participate in
 *              the CDN simply by forwarding queries and query hits.
 *
 */

#ifndef __GNUT_AGENT_H__
#define __GNUT_AGENT_H__

#include <map>
#include "ns-process.h"
#include "tclcl.h"

// Forward declarations
class AppData;
class TcpApp;
class GnutDescriptor;


class GnutAgent : public Process {
public:
    // = Foundation

    GnutAgent(void);

    virtual ~GnutAgent(void);

    // = Action

    // Process commands from TCL
    virtual int command(int argc, const char*const* argv);

    // Process recv'd packets
    void process_data(int, AppData* data);

protected:
    virtual void processQuery(GnutDescriptor* aData);

    // Return true if the hit was for this node, false otherwise.
    virtual bool processQueryResponse(GnutDescriptor* aData);

    // Establish a Gnutella application level connection between
    // this node and the specified node/app.
    int gnutConnect(unsigned int aNodeNumber, TcpApp* aApp);

    // Flood a descriptor out on all connections except the one
    // it came from.
    void floodMsg(GnutDescriptor* aDescr) const;

    // = Access

    // For some reason, the implementors of the name() method neglected to make
    // it a const method even though it doesn't change instance variables.
    // This is inconvenient when we want to call name() from within a const
    // method - we can't. So, the following method plays a dirty little trick
    // to make the call possible :(
    const char* getName() const;

    // = Types

    typedef std::map<unsigned int, TcpApp*> ConnMap;

    // = Constants
    // These constants are the strings that will be displayed in
    // NAM. They describe the action last taken by a GnutAgent.

    // Initiated a query
    static const char* const INIT_QUERY;

    // Forwarded a query
    static const char* const FW_QUERY;

    // Dropped a query with expired TTL
    static const char* const DR_QUERY;

    // Generated a query hit
    static const char* const INIT_QUERY_HIT;

    // Forwarded a query hit
    static const char* const FW_QUERY_HIT;

    // Updated version
    static const char* const UPDATED;

    // = Data

    unsigned int                       mNodeNumber;

    // Map associating node numbers and the TCP connections
    // to reach those nodes.
    ConnMap                            mConns;

private:
    // = Action

    // Return true if the message should be processed further.
    // On false, the message should be ignored.
    bool preProcessQuery(GnutDescriptor* aData) const;
    bool preProcessQueryResponse(GnutDescriptor* aData) const;

    // Propagate message as appropriate
    void postProcessQuery(GnutDescriptor* aData) const;
    void postProcessQueryResponse(GnutDescriptor* aData) const;

    // Disable copy constructor and assignment operator for safety.
    GnutAgent(const GnutAgent&);
    GnutAgent& operator=(const GnutAgent&);
};

#endif /* __GNUT_AGENT_H__ */
```

```
/*
 * File: gnutagent.cc
 * Author: Andre Dufour
 *
 * Description: Implementation for the GnutAgent agent type in ns-2.
 *
 */

#include <assert.h>
#include <stdio.h>

#include "gnututil.h"
#include "gnutagent.h"
#include "gnutdescriptor.h"
#include "gnutquerydata.h"
#include "gnutqueryhitdata.h"
#include "../webcache/tcpapp.h"

// Definition of static class members
const char* const GnutAgent::INIT_QUERY     = "IQ";
const char* const GnutAgent::FW_QUERY       = "FQ";
const char* const GnutAgent::DR_QUERY       = "DQ";
const char* const GnutAgent::INIT_QUERY_HIT = "IQH";
const char* const GnutAgent::FW_QUERY_HIT   = "FQH";
const char* const GnutAgent::UPDATED        = "UPD";

//============================================================================
GnutAgent::
GnutAgent(void)
: mNodeNumber(0)
{ }
//============================================================================
GnutAgent::
~GnutAgent(void)
{
    // Something went really wrong!
    assert(false);
}
//============================================================================
int
GnutAgent::
command(int argc, const char*const* argv)
{
    if (argc >= 2)
    {
        // "gnutconnect"
        if (strncmp(argv[1], "gnutconnect", strlen("gnutconnect")) == 0)
        {
            if (argc == 4)
            {
                // Connect to new sevent
                TcpApp* app = (TcpApp*)TclObject::lookup(argv[3]);
                return gnutConnect(atoi(argv[2]), app);
            }
            else
            {
                GNUT_DBG_PRINT("Wrong number of args for gnutconnect\n");
                return TCL_ERROR;
            }
        }
        // "getversion"
        else if (strncmp(argv[1], "getversion", strlen("getversion")) == 0)
        {
            Tcl::instance().resultf(" "); // No label for IRs
            return TCL_OK;
        }
        // "setnodenumber"
        else if (strncmp(argv[1], "setnodenumber", strlen("setnodenumber")) == 0)
        {
            if (argc == 3)
            {
                mNodeNumber = atoi(argv[2]);
                return TCL_OK;
            }
            else
            {
                GNUT_DBG_PRINT("Didn't specify node number\n");
                return TCL_ERROR;
            }
        }
    }
    GNUT_DBG_PRINT("Got some weird command .%s.\n", argv[1]);
    return TCL_ERROR;
}
//=============================================================================
// Private
int
GnutAgent::
gnutConnect(unsigned int aNodeNumber, TcpApp* aApp)
{
    if (aApp != 0)
    {
        // Don't want duplicate connections
        assert(mConns.find(aNodeNumber) == mConns.end());
        mConns[aNodeNumber] = aApp;
        aApp->target() = this;
        return TCL_OK;
    }
    return TCL_ERROR;
}
//=============================================================================
void
GnutAgent::
process_data(int, AppData* aData)
{
    GnutDescriptor* descr = static_cast<GnutDescriptor*>(aData);
    switch (descr->type())
    {
    case GNUT_QUERY:
        if (preProcessQuery(descr))
        {
            processQuery(descr);
            postProcessQuery(descr);
        }
        break;
    case GNUT_QUERY_RESPONSE:
        if (preProcessQueryResponse(descr))
        {
            if (processQueryResponse(descr) == false)
            {
                // Only consider forwarwind response if it was not
                // for this servent.
                postProcessQueryResponse(descr);
            }
        }
        break;
    default:
        GNUT_DBG_PRINT("%u got an unknown AL message\n", mNodeNumber);
        abort();
        break;
    }
}
//=============================================================================
void
GnutAgent::
processQuery(GnutDescriptor* aData)
{
    // TODO: update display to reflect last action.
    GNUT_DBG_PRINT("IR %u not doing anything w/ query\n", mNodeNumber);
}
//=============================================================================
bool
GnutAgent::
preProcessQuery(GnutDescriptor*) const
{
    // Queries are always processed.
    return true;
}
//=============================================================================
bool
GnutAgent::
preProcessQueryResponse(GnutDescriptor* aData) const
{
    assert(aData->getPath().isNodeNextInPath(mNodeNumber));
    // If this isn't for us, return false?
    return true;
}
//=============================================================================
void
GnutAgent::
postProcessQuery(GnutDescriptor* aData) const
{
    if (aData->getTtl() > 1)
    {
        GnutQueryData* query = static_cast<GnutQueryData*>(aData);
        GnutQueryData* newQuery = new GnutQueryData(*query);

        newQuery->addNodeToPath(mNodeNumber);

        floodMsg(newQuery);
        Tcl::instance().evalf("%s updateLabel %s", getName(), FW_QUERY);
    }
    else
    {
        GNUT_DBG_PRINT("%u dropping query with expired ttl\n", mNodeNumber);
        Tcl::instance().evalf("%s updateLabel %s", getName(), DR_QUERY);
    }
}
//==========================================================================
bool
GnutAgent::
processQueryResponse(GnutDescriptor* aData)
{
    GNUT_DBG_PRINT("IR %u processing query response\n", mNodeNumber);

    // IR's should never be the ultimate destination for a hit.
    assert(!(aData->getPath().isHitForNode(mNodeNumber)));
    assert(aData->getTtl() > 1);
    return false;
}
//==========================================================================
void
GnutAgent::
postProcessQueryResponse(GnutDescriptor* aData) const
{
    if (aData->getTtl() > 1 && (!(aData->getPath().isHitForNode(mNodeNumber))))
    {
        // Take ourselves out of the path and forward it on.
        GnutQueryHitData* hit = static_cast<GnutQueryHitData*>(aData);
        GnutQueryHitData* newHit = new GnutQueryHitData(*hit);
        newHit->removeNodeFromPath();
        newHit->nextHop();
        GNUT_DBG_PRINT("new path = ");
        newHit->getPath().printPath();

        ConnMap::const_iterator out_conn = mConns.find(newHit->getPath().getNextNode());
        assert(out_conn != mConns.end());

        GNUT_DBG_PRINT("%u forwarding query hit to %u\n", mNodeNumber, out_conn->first);
        out_conn->second->send(newHit->getSize(), newHit);
        Tcl::instance().evalf("%s updateLabel %s", getName(), FW_QUERY_HIT);
        GNUT_DBG_PRINT("Query hit forwarded\n");
    }
    else
    {
        GNUT_DBG_PRINT("%u dropped hit with expired ttl or because it was the dest\n");
    }
}
//==========================================================================
void
GnutAgent::
floodMsg(GnutDescriptor* aDescr) const
{
    for (ConnMap::const_iterator iter = mConns.begin();
         iter != mConns.end();
         iter++)
    {
        if (aDescr->getPath().isNodeInPath(iter->first))
        {
            // Don't send back where it has already been.
            GNUT_DBG_PRINT("%u not sending msg to %u - already in path.\n",
                   mNodeNumber, iter->first);
            continue;
        }
        GnutDescriptor* msg = static_cast<GnutDescriptor*>(aDescr->copy());
        msg->nextHop(); // important: decrement ttl and increment hop cnt
        iter->second->send(msg->getSize(), msg);
        GNUT_DBG_PRINT("%u sent msg to %u with ttl = %u\n", mNodeNumber, iter->first,
msg->getTtl());
    }

    GNUT_DBG_PRINT("Done flooding method\n");
}
//=============================================================================
const char*
GnutAgent::
getName() const
{
    GnutAgent* this_agent = const_cast<GnutAgent*>(this);
    return this_agent->name();
}
//=============================================================================
// TCL Binding
static class GnutAppClass : public TclClass
{
public:
    GnutAppClass() : TclClass("Gnut") {}
    TclObject* create(int, const char*const* argv)
    { return (new GnutAgent()); }
} class_gnut_app;
//=============================================================================
###############################################################################
# Constructor
Gnut instproc init { aNs aNode } {
    $self next

    $self instvar mNs mNode
    set mNs $aNs
    set mNode $aNode
    $mNode color black
}
###############################################################################
Gnut instproc updateLabel { aLastAction } {
    $self instvar mNode

    $mNode label "$aLastAction"
}
###############################################################################
```

```
/*
 * File: gnutdescriptor.h
 * Author: Andre Dufour
 *
 * Description: Abstract base class for Gnutella messages
 *
 */


#ifndef __GNUTDESCRIPTOR_H__
#define __GNUTDESCRIPTOR_H__

#include <tcl.h>
#include "gnutpathlist.h"
#include "ns-process.h"
#include "app.h"

class GnutDescriptor : public AppData
{
public:
    // = Foundation

    // Create a descriptor with a new GUID and the default TTL value
    GnutDescriptor(AppDataType aType);

    // Create a descriptor based on the provided one.
    GnutDescriptor(AppDataType aType, const GnutDescriptor& aDescr);

    virtual ~GnutDescriptor(void);

    // = Access

    // Get the size of this message in bytes.
    virtual unsigned long getSize(void) const = 0;

    // Get message id
    inline unsigned long getId(void) const { return mId; }

    // Get the number of hops traversed by this message
    inline unsigned char getHops(void) const { return mHops; }

    // Get the ttl of this message
    inline unsigned char getTtl(void) const { return mTtl; }

    // Access to the path list
    inline const GnutPathList& getPath(void) const { return mPathList; }

    // = Action

    // Add specified node as the most recent in the path
    void addNodeToPath(unsigned int aNodeNumber);

    // Remove the most recent node in path
    void removeNodeFromPath(void);

    // Adjust hops and ttl ctrs
    void nextHop(void);

    // Reset hop and ttl ctrs to initial values
    void resetCounters(void);

    // Number of hops message can traverse before expiring
    enum {GNUT_DEFAULT_TTL = 16};

private:
    // = Data

    GnutPathList                mPathList;
    unsigned long               mId;
    unsigned char               mHops;
    unsigned char               mTtl;

    static unsigned long        theNextId;

    // Disable copy constructor and assignment operator for safety.
    GnutDescriptor(const GnutDescriptor&);
    GnutDescriptor& operator=(const GnutDescriptor);
};
#endif /* __GNUTDESCRIPTOR_H__ */
```

```
/*
 * File: gnutdescriptor.cc
 * Author: Andre Dufour
 *
 * Description: Abstract base class for Gnutella messages
 *
 */

#include <tcl.h>
#include "gnutdescriptor.h"
#include "ns-process.h"
#include "app.h"

// Init static variable
unsigned long GnutDescriptor::theNextId = 0;

//=============================================================================
GnutDescriptor::
GnutDescriptor(AppDataType aType)
: AppData(aType),
  mId(theNextId++),
  mHops(0),
  mTtl(GNUT_DEFAULT_TTL)
{ }
//=============================================================================
GnutDescriptor::
GnutDescriptor(AppDataType aType, const GnutDescriptor& aDescr)
: AppData(aType),
  mPathList(aDescr.mPathList),
  mId(aDescr.mId),
  mHops(aDescr.mHops),
  mTtl(aDescr.mTtl)
{
    assert(aDescr.mTtl > 0);
}
//=============================================================================
GnutDescriptor::
~GnutDescriptor(void)
{ }
//=============================================================================
void
GnutDescriptor::
addNodeToPath(unsigned int aNodeNumber)
{
    mPathList.printPath();
    mPathList+=aNodeNumber;
}
//=============================================================================
void
GnutDescriptor::
removeNodeFromPath(void)
{
    mPathList--;
}
//=============================================================================
void
GnutDescriptor::
nextHop(void)
{
    assert(mTtl > 1);
    mTtl--;
    mHops++;
}
//=============================================================================
void
GnutDescriptor::
resetCounters(void)
{
    mHops = 0;
    mTtl = GNUT_DEFAULT_TTL;
}
//=============================================================================
/*
 * File: gnutpathlist.h
 * Author: Andre Dufour
 *
 * Description: Encapsulates a list of nodes traversed by a query
 *              so that the corresponding query hit can be back
 *              propagated properly.
 *
 */

#ifndef __GNUTPATHLIST_H__
#define __GNUTPATHLIST_H__

#include <list>

class GnutPathList
{
public:
    // = Foundation

    GnutPathList(void);

    virtual ~GnutPathList(void);

    // = Access

    // Checks whether the specified node is the next one on the path.
    // If it is not, a Gnutella having that node number and receiving
    // a query hit message with this path must ignore the query hit
    // and must not propagate it.
    bool isNodeNextInPath(unsigned int aNodeNumber) const;

    // Checks whether the specified node is anywhere on the path. If it is,
    // agents must not send queries there again.
    bool isNodeInPath(unsigned int aNodeNumber) const;

    // Checks whether the query hit message is in response to a query
    // that originated at the specified node.
    bool isHitForNode(unsigned int aNodeNumber) const;

    // Return the first node on the reverse path (i.e. the last node
    // visited). Needed to route query hit messages on reverse path.
    unsigned int getNextNode(void) const;

    // = Action

    // Adds the specified node to the path. Every node on the
    // query's path must do this.
    GnutPathList& operator+=(unsigned int aNodeNumber);

    // Removes an element from the path. Every node on the
    // query hit's path (the reverse path) must do this.
    GnutPathList& operator--(int);

    // TODO: debug
    void printPath(void) const;

private:
    // = Types
    typedef std::list<unsigned int> PathList;

    // = Data

    // List of nodes traversed
    PathList                mList;

    // Disable assignment operator for safety
    GnutPathList& operator=(const GnutPathList&);
};

#endif /* __GNUTPATHLIST_H__ */
```

```
/*
 * File: gnutpathlist.cc
 * Author: Andre Dufour
 *
 * Description: Implementation for a list of nodes traversed by a query
 *              so that the corresponding query hit can be back
 *              propagated properly.
 *
 */

#include <algorithm>
#include "gnututil.h"
#include "gnutpathlist.h"

//=============================================================================
GnutPathList::
GnutPathList(void)
{ }
//=============================================================================
GnutPathList::
~GnutPathList(void)
{ }
//=============================================================================
bool
GnutPathList::
isNodeNextInPath(unsigned int aNodeNumber) const
{
    return aNodeNumber == mList.back();
}
//=============================================================================
bool
GnutPathList::
isHitForNode(unsigned int aNodeNumber) const
{
    GNUT_DBG_PRINT("Is hit for node? Next node = %u. Size = %u\n", mList.back(),
mList.size());
    return isNodeNextInPath(aNodeNumber) && (mList.size() == 1);
}
//=============================================================================
bool
GnutPathList::
isNodeInPath(unsigned int aNodeNumber) const
{
    return std::find(mList.begin(), mList.end(), aNodeNumber) != mList.end();
}
//=============================================================================
unsigned int
GnutPathList::
getNextNode(void) const
{
    assert(mList.size() >= 1);
    return mList.back();
}
//=============================================================================
GnutPathList&
GnutPathList::
operator+=(unsigned int aNodeNumber)
{
    assert(mList.max_size() > (mList.size() + 1));
    mList.push_back(aNodeNumber);
    return *this;
}
//=============================================================================
GnutPathList&
GnutPathList::
operator--(int)
{
    assert(mList.size() > 0);
    mList.pop_back();
    return *this;
}
//=============================================================================
// TODO: debug remove!!!
#include <stdio.h>
void
GnutPathList::
printPath(void) const
{
    GNUT_DBG_PRINT("Path size = %u\n", mList.size());
    for (PathList::const_iterator it = mList.begin();
         it != mList.end();
         it++)
    {
        GNUT_DBG_PRINT(", %u", *it);
    }

    GNUT_DBG_PRINT("\n");
}
//============================================================================
/*
 * File: gnutquerydata.h
 * Author: Andre Dufour
 *
 * Description: Header file for gnutella query data object
 *
 */

#ifndef __GNUTQUERYDATA_H__
#define __GNUTQUERYDATA_H__

#include "gnutdescriptor.h"

class GnutQueryData : public GnutDescriptor
{
public:
    // = Foundation

    // Create new query
    explicit GnutQueryData(unsigned int aSearchString);

    // Copy constructor
    // Create object based on existing query.
    // NB: the ttl and hop count should normally be adjusted by
    //     calling the appropriate method in the base class.
    explicit GnutQueryData(const GnutQueryData& aQuery);

    virtual ~GnutQueryData();

    // = Access

    // Get the value of the search string
    unsigned int getSearchString(void) const;

    // Get size of message in bytes
    unsigned long getSize(void) const;

    // = Interface AppData

    AppData* copy(void);

private:
    // = Data

    // Simplification: string is only the version number.
    unsigned int                    mSearchString;

    // Disable assignment operator for safety
    GnutQueryData& operator=(const GnutQueryData&);
};

#endif /* __GNUTQUERYDATA_H__ */
```

```
/*
 * File: gnutquerydata.cc
 * Author: Andre Dufour
 *
 * Description: Implementation for gnutella query data object
 *
 */

#include <stdio.h>
#include "gnutquerydata.h"
#include "ns-process.h"


//=============================================================================
GnutQueryData::
GnutQueryData(unsigned int aSearchString)
: GnutDescriptor(GNUT_QUERY),
  mSearchString(aSearchString)
{ }
//=============================================================================
GnutQueryData::
GnutQueryData(const GnutQueryData& aQuery)
: GnutDescriptor(GNUT_QUERY, aQuery),
  mSearchString(aQuery.mSearchString)
{ }
//=============================================================================
GnutQueryData::
~GnutQueryData(void)
{ }
//=============================================================================
unsigned int
GnutQueryData::
getSearchString(void) const
{
    return mSearchString;
}
//=============================================================================
unsigned long
GnutQueryData::
getSize(void) const
{
    // 23 bytes for Gnut descriptor header
    // 1 byte for min speed
    // Some bytes for the search string
    // 1 byte for null termination
    return 23 + 1 + 1 + 1;
}
//=============================================================================
AppData*
GnutQueryData::
copy(void)
{
    return new GnutQueryData(*this);
}
//=============================================================================
/*
 * File: gnutqueryhitdata.h
 * Author: Andre Dufour
 *
 * Description: Header file for gnutella query hit data object
 *
 */


#ifndef __GNUTQUERYHITDATA_H__
#define __GNUTQUERYHITDATA_H__

#include <tcl.h>
#include "gnutdescriptor.h"

class GnutQueryData;

class GnutQueryHitData : public GnutDescriptor
{
public:
    enum { MAX_HIT_STRING_LEN = 200 };

    // = Foundation

    // Construct a query hit message as a response to the specified
    // query
    GnutQueryHitData(const GnutQueryData& aQuery,
                     unsigned int aHitString);

    // Create object based on existing query; can adjust hops and TTL.
    // NB: ttl and hop counts should normally be adjusted by the caller
    //     on the new object.
    explicit GnutQueryHitData(const GnutQueryHitData& aQuery);

    virtual ~GnutQueryHitData(void);

    // = Access

    unsigned int getHitString(void) const;

    // Get size of message in bytes
    unsigned long getSize(void) const;

    // = Interface AppData

    AppData* copy(void);

private:
    // = Data

    // Simplification: hit string is just a version number.
    unsigned int            mHitString;

    // Disable assignment operator for safety
    GnutQueryHitData& operator=(const GnutQueryHitData&);
};

#endif /* __GNUTQUERYHITDATA_H__ */
```

```
/*
 * File: gnutqueryhitdata.cc
 * Author: Andre Dufour
 *
 * Description: Implementation for gnutella query hit data object
 *
 */

#include <tcl.h>
#include "gnutqueryhitdata.h"
#include "gnutquerydata.h"
#include "gnutdescriptor.h"

//=============================================================================
GnutQueryHitData::
GnutQueryHitData(
    const GnutQueryData& aQuery,
    unsigned int aHitString)
: GnutDescriptor(GNUT_QUERY_RESPONSE, aQuery),
  mHitString(aHitString)
{ }
//=============================================================================
GnutQueryHitData::
GnutQueryHitData(const GnutQueryHitData& aQuery)
: GnutDescriptor(GNUT_QUERY_RESPONSE, aQuery),
  mHitString(aQuery.mHitString)
{ }
//=============================================================================
GnutQueryHitData::
~GnutQueryHitData(void)
{ }
//=============================================================================
unsigned int
GnutQueryHitData::
getHitString(void) const
{
    return mHitString;
}
//=============================================================================
unsigned long
GnutQueryHitData::
getSize(void) const
{
    // 23 bytes for Gnut descriptor header
    // 1 byte for min speed
    // Some bytes for the search string
    // 1 byte for null termination
    return 23 + 1 + 1 + 1;
}
//=============================================================================
AppData*
GnutQueryHitData::
copy(void)
{
    return new GnutQueryHitData(*this);
}
//=============================================================================
/*
 * File: gnututil.h
 * Author: Andre Dufour
 *
 * Description: utility functions/macros for ns-2 Gnutella implementation
 *
 */

#ifndef __GNUTUTIL_H__
#define __GNUTUTIL_H__

#include <stdio.h>

//#define GNUT_DEBUG

#ifdef GNUT_DEBUG
#define GNUT_DBG_PRINT(args...) printf(args)
#else
#define GNUT_DBG_PRINT(args...)
#endif

#endif /* __GNUTUTIL_H__ */
```

```
/*
 * File: releventagent.h
 * Author: Andre Dufour
 *
 * Description: Header file for Gnutella servents participating in the CDN
 *              as active content seekers.
 *
 */

#ifndef __RELEVENTAGENT_H__
#define __RELEVENTAGENT_H__

#include "gnutagent.h"
#include "ns-process.h"
#include "tclcl.h"

// Forward declarations
class AppData;
class TcpApp;


class ReleventAgent : public GnutAgent {
public:
    // = Foundation

    ReleventAgent(void);

    virtual ~ReleventAgent(void);

    // = Action

    void processQuery(GnutDescriptor* aData);

    bool processQueryResponse(GnutDescriptor* aData);

    // Parse command from TCL
    int command(int argc, const char*const* argv);

protected:
    // = Data
    unsigned int        mVersion;

private:
    // = Action
    void initiateQuery(void) const;
    void setVersion(unsigned int aVersion);

    // Disable copy constructor and assignment operator for safety.
    ReleventAgent(const ReleventAgent&);
    ReleventAgent& operator=(const ReleventAgent&);
};

#endif /* __RELEVENTAGENT_H__ */
```

```
/*
 * File: releventagent.cc
 * Author: Andre Dufour
 *
 * Description: Implementation for the ReleventAgent agent type in ns-2.
 *
 */

#include <assert.h>
#include <stdio.h>

#include "gnututil.h"
#include "releventagent.h"
#include "gnutquerydata.h"
#include "gnutqueryhitdata.h"
#include "../webcache/tcpapp.h"

//============================================================================
ReleventAgent::
ReleventAgent(void)
: mVersion(0)
{
    GNUT_DBG_PRINT("Relevent created\n");
}
//============================================================================
ReleventAgent::
~ReleventAgent(void)
{
    // Something went really wrong!
    assert(false);
}
//============================================================================
int
ReleventAgent::
command(int argc, const char*const* argv)
{
    if (argc >= 2)
    {
        // "gnutconnect"
        if (strncmp(argv[1], "gnutconnect", strlen("gnutconnect")) == 0)
        {
            if (argc == 4)
            {
                // Connect to new sevent
                GNUT_DBG_PRINT("About to perform connection (gnutconnect)\n");
                TcpApp* app = (TcpApp*)TclObject::lookup(argv[3]);
                return gnutConnect(atoi(argv[2]), app);
            }
            else
            {
                GNUT_DBG_PRINT("Wrong number of args for gnutconnect\n");
                return TCL_ERROR;
            }
        }
        // "sendquery"
        else if (strncmp(argv[1], "sendquery", strlen("sendquery")) == 0)
        {
            initiateQuery();
            return TCL_OK;
        }
        // "setversion"
        else if (strncmp(argv[1], "setversion", strlen("setversion")) == 0)
        {
            if (argc == 3)
            {
                GNUT_DBG_PRINT("Setting version\n");
                setVersion(atoi(argv[2]));
                return TCL_OK;
            }
            else
            {
                GNUT_DBG_PRINT("Didn't specify what version to set to.\n");
                return TCL_ERROR;
            }
        }
        // "getversion"
        else if (strncmp(argv[1], "getversion", strlen("getversion")) == 0)
        {
            GNUT_DBG_PRINT("Getting version\n");
            Tcl::instance().resultf("%d", mVersion);
            return TCL_OK;
        }
        // "setnodenumber"
        else if (strncmp(argv[1], "setnodenumber", strlen("setnodenumber")) == 0)
        {
            if (argc == 3)
            {
                GNUT_DBG_PRINT("Setting node number\n");
                mNodeNumber = atoi(argv[2]);
                return TCL_OK;
            }
            else
            {
                GNUT_DBG_PRINT("Didn't specify node number\n");
                return TCL_ERROR;
            }
        }
    }
    GNUT_DBG_PRINT("Got some weird command .%s.\n", argv[1]);
    return TCL_ERROR;
}
//=============================================================================
void
ReleventAgent::
initiateQuery(void) const
{
    Tcl& tcl = Tcl::instance();

    GNUT_DBG_PRINT("About to initiate query\n");
    GnutQueryData* query = new GnutQueryData(mVersion);
    query->addNodeToPath(mNodeNumber);
    floodMsg(query);
    GNUT_DBG_PRINT("Relevent %d about to reset timer\n", mNodeNumber);
    tcl.evalf("%s reloadQueryTimer", getName());
    tcl.evalf("%s updateLabel %s", getName(), GnutAgent::INIT_QUERY);
}
//=============================================================================
void
ReleventAgent::
setVersion(unsigned int aVersion)
{
    Tcl& tcl = Tcl::instance();
    mVersion = aVersion;
    tcl.evalf("%s updateLabel %s", getName(), GnutAgent::UPDATED);
    tcl.evalf("%s declareUpdate %u", getName(), aVersion);
}
//=============================================================================
void
ReleventAgent::
processQuery(GnutDescriptor* aData)
{
    Tcl& tcl = Tcl::instance();
    GnutQueryData* query = static_cast<GnutQueryData*>(aData);
    GNUT_DBG_PRINT("Relevent %u processing query\n", mNodeNumber);

    if (query->getSearchString() < mVersion)
    {
        // We can offer them a higher version. Send a query hit message.
        GnutQueryHitData* hit = new GnutQueryHitData(*query, mVersion);
        hit->resetCounters();

        ConnMap::const_iterator out_conn = mConns.find(hit->getPath().getNextNode());
        assert(out_conn != mConns.end());

        GNUT_DBG_PRINT("%u sending query hit to %u\n", mNodeNumber, out_conn->first);
        out_conn->second->send(hit->getSize(), hit);
        hit->getPath().printPath();
        GNUT_DBG_PRINT("Query hit sent\n");
        tcl.evalf("%s updateLabel %s", getName(), GnutAgent::INIT_QUERY_HIT);
    }
    else
    {
        GNUT_DBG_PRINT("%u's version isn't larger than the requested one\n");
        //tcl.evalf("%s updateLabel", getName());
    }
}
//=============================================================================
bool
```

```
ReleventAgent::
processQueryResponse(GnutDescriptor* aData)
{
    GNUT_DBG_PRINT("Relevent %u processing query response\n", mNodeNumber);
    GNUT_DBG_PRINT("Path=");
    aData->getPath().printPath();
    bool retval = false;

    if (aData->getPath().isHitForNode(mNodeNumber))
    {
        retval = true;
        GnutQueryHitData* hit = static_cast<GnutQueryHitData*>(aData);
        unsigned int hitVer = hit->getHitString();

        if (hitVer > mVersion)
        {
            GNUT_DBG_PRINT("%u got hit with higher version - changing ver.\n",
mNodeNumber);
            setVersion(hitVer);
        }
    }

    return retval;
}
//============================================================================
// TCL Binding
static class ReleventAppClass : public TclClass
{
public:
    ReleventAppClass() : TclClass("Relevent") {}
    TclObject* create(int, const char*const* argv)
    { return (new ReleventAgent()); }
} class_relevent_app;
//============================================================================
############################################################################
# File: releventagent.tcl
# Author: Andre Dufour
#
# TCL portion of the implementation for a relevent agent.
############################################################################

source gnut_globals.tcl

############################################################################
# Constructor
Relevent instproc init { aNs aNode } {
    $self next

    $self instvar mNs mNode mStopQueries
    set mNs $aNs
    set mNode $aNode
    set mStopQueries 0
    $mNode color blue
}
############################################################################
Relevent instproc updateLabel { aLastAction } {
    $self instvar mNode

    $mNode label "[$self getversion] $aLastAction"
}
############################################################################
Relevent instproc startQueries {} {
    global rng min_query_delay max_query_delay
    $self instvar mNs

    set update_delay [expr [$rng uniform $min_query_delay $max_query_delay]]

    $mNs after $update_delay "$self sendquery"
}
############################################################################
# This stops the scheduling of further queries. It won't cancel already
# scheduled ones.
Relevent instproc stopQueries {} {
   $self instvar mStopQueries
   set mStopQueries 1
}
############################################################################
Relevent instproc reloadQueryTimer {} {
    $self instvar mNs mStopQueries
    global rng min_query_delay max_query_delay

    if {$mStopQueries == 0} {
        set update_delay [expr [$rng uniform $min_query_delay $max_query_delay]]
        $mNs after $update_delay "$self sendquery"
    }
}
############################################################################
Relevent instproc setStats { aStats } {
    $self instvar mStats

    set mStats $aStats
}
############################################################################
Relevent instproc declareUpdate { aVersion } {
    $self instvar mStats

    $mStats declareUpdate $aVersion
}
############################################################################
```

```
##############################################################################
# File: stats.tcl
# Author: Andre Dufour
#
# Handles statistic gathering during the simulation
##############################################################################
source gnut_globals.tcl

Class Stats

##############################################################################
# Constructor
Stats instproc init { aNs aNumRelevents } {
    $self instvar mNs mNumRelevents mLatestVersion mNotUpdated mTime

    $self next

    set mNs $aNs
    set mNumRelevents $aNumRelevents
    set mLatestVersion 0
    set mNotUpdated 0
    set mTime 0
}
##############################################################################
# Called when a new version is introduced into the network
Stats instproc newVersion { aVersion } {
    $self instvar mLatestVersion mNotUpdated mNumRelevents

    if { $aVersion <= $mLatestVersion } {
        error "Invalid lower version number"
    }

    set mLatestVersion $aVersion
    set mNotUpdated $mNumRelevents
}
##############################################################################
# Relevents call this when they update their version
Stats instproc declareUpdate { aVersion } {
    $self instvar mLatestVersion mNotUpdated

    if { $aVersion == $mLatestVersion } {
        incr mNotUpdated -1
    }
}
##############################################################################
# Stats are logged every second
Stats instproc timerTick {} {
    $self instvar mTime mNotUpdated mNs

    set statsfile [open stats.out a]
    incr mTime

    puts "Simulating time = $mTime"
    puts $statsfile "$mTime $mNotUpdated"
    close $statsfile

    $mNs after 1.0 "$self timerTick"
}
##############################################################################
```