A Scalable & Extensible Peer-to-Peer Network Simulator

By

Jonathan B. Harris

B.C.S.

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Master of Computer Science
At

Ottawa-Carleton Institute for Computer Science

Carleton University
Ottawa, Ontario
April 2005

Abstract

Peer-to-peer networks are of great value for tapping the unused resources of computers residing at the edges of the internet. However, the recent rise in popularity of peer-to-peer applications has caused the sizes of their underlying networks to grow significantly. Because of their large size and decentralized nature, it is all but impossible to measure the efficiency of the protocols driving these networks in the real world. To facilitate the evaluation of peer-to-peer protocols prior to deployment in real networks, this thesis presents a scalable simulation architecture which can be extended to support arbitrary peer-to-peer protocols. In particular, the simulator models the competition of messages for bandwidth as traffic increases. For validation purposes, an implementation of multiple versions of the Gnutella protocol is presented. It demonstrates the ability of the simulator to model arbitrary peer behaviour and handle heterogeneous protocols.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Peer-to-Peer

The evolution of the internet gave rise to two distinct classes of machines: servers and clients. Servers are relatively static, in the sense that they have fixed network addresses and high uptimes. By comparison, clients are relatively dynamic, with temporary network addresses and variable connectivity. The predominant internet protocols, such as those used for web-browsing (Hyper Text Transfer Protocol), email (Simple Mail Transfer Protocol), and file transfers (File Transfer Protocol) all communicate on a client-server and server-server level. Servers provide the content, and clients retrieve it.

In a social context, the term peer brings to mind the idea of a colleague or acquaintance of equal stature. Using this definition, it would seem logical to define a peer-to-peer network as a network of machines with similar function and computing power, like the network used for communication between servers on the internet. In practice however, applying the peer-to-peer label to a network signifies a great deal

more. As defined by Clay Shirky [Ora01], peer-to-peer is "a class of applications that takes advantage of resources – storage, cycles, content, human presence – available at the edges of the internet". Thus, peer-to-peer is a term used to define a divergence from the classic client-server model, giving clients the ability to communicate and work together without the aid of the servers. In general, a network can be labelled as peer-to-peer if it satisfies both of the following criteria [Ora01]:

1) It allows for variable connectivity and temporary network addresses.
2) It gives the nodes at the edges of the network significant autonomy.

Peer-to-peer networks are a class of *overlay networks*. Overlay networks are networks that are overlaid on top of an existing network such as the internet, essentially forming a subset of the underlying network. Peer-to-peer networks are of great value for tapping the unused resources of clients residing at the edge of the internet. They allow clients to break free from the traditional master-slave relationship of the past, and to collaborate together as peers.

1.2 Problem

To cope with the variable connectivity of, and to provide a high degree of autonomy to their participating peers, most peer-to-peer networks operate in a decentralized manner, with no central server or servers. This approach is beneficial in the sense that it eliminates any central point of failure, and removes a central barrier to

scalability. The downside of this approach is that it is often all but impossible, or at best prohibitively expensive to measure the performance and scalability of the peer-to-peer protocols employed for communication between peers in the network. For example, the demise of the Napster network [Nap] lead to a surge in popularity of the Gnutella [Gnu] network. Unfortunately, the Gnutella network was poorly equipped to handle the resulting rapid increase in the number of peers, and collapsed under the strain.

For this reason, many developers of peer-to-peer applications employ a simulator to test and measure the performance of peer-to-peer protocols prior to their deployment on a real network. However, the development of a simulator for peer-to-peer networks requires careful consideration of several factors:

- **Scalability** – A frequently asked question about different peer-to-peer networks is whether or not their performance degrades as their populations grow to millions of peers. In some cases, the lack of scalability of a peer-to-peer network only becomes apparent as its size grows to hundreds of thousands or even millions of peers, as witnessed by the collapse of Gnutella in 2000. If a peer-to-peer simulator is to study this phenomenon, it must be itself highly scalable, minimizing the memory required to simulate peers and their communication with each other.

- **Extensibility** – Many peer-to-peer simulators have been developed that are custom-made for the simulation of a specific protocol. Ideally, a peer-to-peer simulator should be extensible, allowing for the simulation of arbitrary peer-to-peer protocols.
- **Peer Behaviour** – Most peer-to-peer networks are constantly evolving. At any time, peers can join or leave the network, or initiate communication with other peers. A peer-to-peer simulator should be able to model this behaviour.
- **Traffic Modeling** – The primary constraint on peer-to-peer networks is the availability of network resources to handle the communication between peers. Some peer-to-peer protocols make more efficient use of these resources than others. To measure the performance of a peer-to-peer protocol, a peer-to-peer simulator needs to provide a relatively realistic model of how network resources are consumed as communication traffic increases.
- **Content Modeling** – The most common and most widely studied peer-to-peer networks are those for distributing, sharing and retrieving content, such as documents, music files or video files. A peer-to-peer simulator needs to be able to model different distributions of content to peers in the network, and model varying degrees of interest in or popularity of content in this distribution.

- **Measurement** – Some peer-to-peer protocols will consume more network resources than others. To measure the relative performance of different varieties of peer-to-peer protocols, a peer-to-peer simulator needs to provide accurate statistics detailing the resources consumed as peers engage in various behaviour.

Several existing peer-to-peer simulators consider one or more of these factors, but as of this date, no peer-to-peer simulator takes all of them into account. This motivates the development of a scalable, extensible peer-to-peer simulator with support for peer behaviour modeling, network traffic modeling and reliable measurement of performance.

1.3 Contributions

This thesis focuses on the development of a scalable and extensible peer-to-peer simulator. Its contributions are summarized as follows:

- **Scalability** – The simulator is designed to be able to simulate peer-to-peer networks with up to 1 million peers, subject to available system memory and traffic density.
- **Extensibility** – The core architecture of the simulator consists of a set of components which can be extended to add support for arbitrary peer-to-peer

protocols. Protocol implementations can be packaged as a plug-in for the simulator, and loaded dynamically at run time.

- **Peer Behaviour** – The simulator can be fed at regular or continuous time intervals with simulated events corresponding to any of the behaviour supported by a protocol implementation.
- **Traffic Modeling** – The Minimum Equal Share bandwidth model is proposed and evaluated as a means for modeling the slow-down in message transmission times as traffic density increases.
- **Content Modeling** – The simulator supports arbitrary distributions of content to peers in the simulated network. Individual content can be assigned a popularity ranking, affecting the frequency with which they are distributed to peers, and selected for search and retrieval.
- **Measurement** – The simulator allows for the collection and reporting of statistics pertaining to each type of message or behaviour supported by a protocol implementation.
- **Gnutella Evaluation** – The simulator is tested by an implementation of multiple versions of the popular Gnuella peer-to-peer protocol. This showcases the simulator's ability to simulate networks with heterogeneous protocols.

1.4 Outline

Chapter 2 provides background information relevant to the simulation of peer-to-peer networks. This includes different categories of peer-to-peer protocols, techniques for capturing their network topologies, and methods for simulating inter-peer communication. This chapter also provides a survey and comparison of existing peer-to-peer simulators.

Chapter 3 describes the design of the core peer-to-peer network architecture that was developed for this thesis to allow the simulation of large peer-to-peer networks. Chapter 4 summarizes how the core simulation architecture was designed for scalability, and describes the process for extending it to support arbitrary peer-to-peer protocols. Chapter 5 describes the Gnutella protocol plug-in that was developed to test the scalability and extensibility of the core architecture.

Chapter 6 presents the experimental results obtained from simulation runs with the Gnutella protocol. It provides an analysis of the results, relating them to the parameters used during each simulation run. Chapter 7 evaluates the success achieved in meeting the goals of the thesis, outlines the contributions made, and discusses future work.

Chapter 2

Background

The purpose of this chapter is to discuss common aspects of existing peer-to-peer systems and simulators which require attention in the design of a scalable and extensible peer-to-peer network simulator.

Section 2.1 gives the reader some background about peer-to-peer protocols. Section 2.1.1 describes the different classifications of peer-to-peer protocols, and Section 2.1.2 gives a high level description of some peer-to-peer protocols that belong to these classifications.

Section 2.2 describes issues related to accurately and realistically representing the network topologies underlying real world peer-to-peer communities. Section 2.2.1 describes some data structures for representing the network topology. Section 2.2.2 discusses some structural models which mimic the organizational structure of real world peer-to-peer communities on the internet. Section 2.2.3 provides a summary of some existing methods for generating topologies with these structural models. Section 2.2.4 briefly discusses issues related to dynamic or evolving network topologies.

Section 2.3 describes some of the issues involved in simulating the communication that occurs over the network topology. This includes different methods for simulating bandwidth constraints and communication delays.

Section 2.4 discusses issues related to the implementation of a peer-to-peer simulator. Section 2.4.1 describes some of the concepts and terminology used in the simulation literature. Section 2.4.2 discusses different approaches for processing events that occur during simulation. Section 2.4.3 discusses techniques for realistically modeling the content stored by peers and their behaviour towards and interaction with the community as a whole. Section 2.4.4 discusses the important issue of scalability which is necessary to allow simulation of large scale peer-to-peer networks.

Section 2.5 surveys some existing peer-to-peer simulators, focusing primarily on their scalability and extensibility. This section concludes with a comparison of the surveyed simulators.

2.1 Peer-to-peer Protocols

As with peers in real life, peers in a community must speak a common language or protocol in order to communicate amongst each other. Understanding peer-to-peer protocols is of critical importance if we wish to accurately simulate and compare them. This section describes different categorizations of peer-to-peer protocols, followed by examples of different protocols from these categories.

2.1.1 Classifications

Peer-to-peer protocols can be classified in a variety of ways. The main classifications are centralized or decentralized and structured or unstructured. Most structured or unstructured protocols fall into the decentralized classification, although there is no hard and fast rule that states that this must be the case. The following sections describe the distinctions between the centralized and decentralized classifications, and the structured and unstructured classifications.

2.1.1.1 Centralized versus Decentralized

Peer-to-peer protocols can be centralized or decentralized[1] [Ora01] [Min01]. Centralized or more specifically centrally coordinated systems such as instant messaging networks and Napster (section 2.1.2.1 Napster) make use of some form of central server that acts as a broker between peers. In decentralized systems (sections 2.1.2.2 – 2.1.2.5), no such intermediary exists.

The difference between centralized and decentralized peer-to-peer protocols can be likened to two different methods for establishing a conversation in real life. Let's say Mary wishes to correspond with a Doctor named Bob, but doesn't have his contact information.

[1] Centralization or decentralization can equally be applied as a classifier for the network topology of peer-to-peer systems.

With a centralized protocol, Mary consults an online telephone directory or yellow pages in order to find the phone number or e-mail address of all doctors named Bob.

With a decentralized protocol, Mary phones or e-mails each of her friends and asks them to check if they know any Doctors named Bob. Unless they know Bob's number, each of Mary's friends contacts each of their friends in turn (except Mary of course) to ask them the same question, and so on. As soon as someone is contacted that knows the contact information of a Doctor named Bob, a sequence of call-backs is initiated until finally the information is communicated back to Mary.

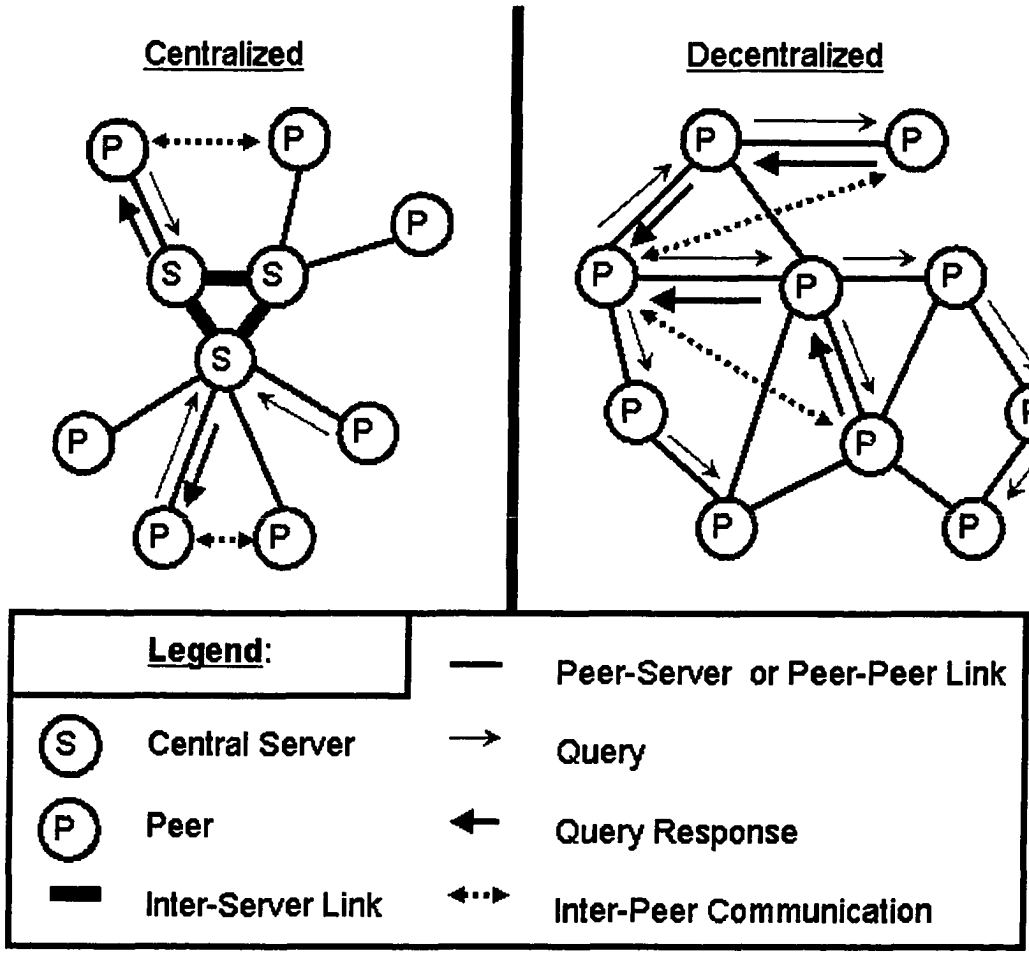

Figure 2.1 Differences in structure and message flow for centralized and decentralized networks

The vulnerability in a centralized network is that they it relies entirely on the online telephone directory or yellow pages in order to initiate all communication, and cannot function without it. Decentralized networks on the other hand can function in all circumstances provided the network topology is available.

For simulations of centralized peer-to-peer systems, explicit representation of the network topology is made mostly redundant by the central server. As such, the remainder of this chapter will focus on decentralized peer-to-peer systems.

2.1.1.2 Structured versus Unstructured

There are two different types of decentralized peer-to-peer protocols, *Structured* (Sections 2.1.2.4 – 2.1.2.5) and *Unstructured* (Sections 2.1.2.2 – 2.1.2.4) protocols [CRB03]. Structured protocols are often called Distributed Hash Table protocols or Second Generation protocols. Unstructured protocols are often called *Broadcast Flooding* protocols or First Generation protocols[2].

Distributed Hash Table (DHT) or structured protocols use the idea that the realm of searchable data can be split up into roughly equal sized pieces. Each document is mapped to a key, and each piece of the search space is uniquely related to a range of keys in a manner similar to conventional hash tables. In an article by Scarlet Pruitt in *Network World Fusion* [Pru02], Kaashoek of MIT is quoted as likening the use of DHT to the use of a distributed filing cabinet. Kaashoek says that "DHT is like having a file cabinet distributed over numerous servers... there is no central server in the system that contains a list of where all the data, or files in the

[2] The use of structured or unstructured protocols has a large impact on the network topology of the peer-to-peer community; however since the network topologies become structured as a result of the protocol, placing the discussion in this section is more fitting.

cabinet, are located. Instead, each server has a partial list of where data is stored in the system. The trick for the researchers is creating a "lookup" algorithm that allows the location of data to be found in a short series of steps".

Searching with DHT is a controlled process, and each peer typically knows exactly to which of its neighbours to forward a search. In practice, DHT protocols (such as Pastry described in section 2.1.2.5 Pastry) can typically guarantee that no more than log (N) peers are contacted when searching for data in a structured peer-to-peer network with N peers. There are costs to be paid for this more efficient search; DHT protocols require that the user provide the exact key associated with the data they are searching for, and in some cases, each arrival or departure of a peer from the network can require up to log (N) messages.

Broadcast Flooding (BF) or unstructured protocols are much more relaxed about the organization of searchable data. Searching with a BF is almost haphazard; each peer typically forwards a search to each of its neighbours, if they have not already seen it. The decentralized portion of Figure 2.1 gives an example of the structure and message flow of a Broadcast Flooding protocol. Some BF protocols use a **selective broadcast** approach in an attempt to give peers some better knowledge about peers to whom a search should be forwarded.

BF protocols are often criticized for their lack of scalability since every peer may be contacted during a search. The use of a time to live (TTL)[3] message variable to enforce a limited horizon for search retransmission can help reduce this cost by a limited extent. Despite their scalability issues, BF protocols do offer many advantages. One advantage is that there is no limitation on the form of the search term. Data can be searched for by key or by key-word. Another advantage is that each peer arrival or departure is relatively inexpensive, usually requiring only a constant number, O(1) of messages.

Chawathe et al. provide a nice discussion of some of the tradeoffs between DHT and BF protocols in their paper *Making Gnutella-like P2P Systems Scalable* [CRB03]. The authors relate DHT and BF protocols analogously to the needle and a haystack scenario. DHT protocols excel at always finding any file that exists on the network, provided you know exactly what you are looking for (finding a needle). BF protocols on the other hand, cannot always guarantee finding a specific file, yet most users typically search for files that are well replicated across the network, and thus easily found (the hay). Based on empirical data observed from real peer-to-peer networks, Chawathe et al. conclude that most users are 'looking for the hay, not the

[3] A TTL variable gets decremented every time a message gets retransmitted. Once its value reaches 0, the message is no longer retransmitted [Ora01].

needle', making the use of BF protocols a more viable approach for certain applications.

2.1.2 Existing Protocols

A variety of peer-to-peer protocols have been devised over time, each having their own unique characteristics. These protocols are typically introduced through new peer-to-peer applications which are designed to exploit them. Rather than giving an exhaustive listing of the details of each protocol (which are often unavailable or confidential), this section will describe general details about several protocols at the application level. The goal of this section is to inform the reader about different protocols and to bring to light some of the requirements that will need to be fulfilled to simulate them.

2.1.2.1 Napster

The application that first brought the peer-to-peer paradigm to the public's attention was Napster [Nap]. Napster was a file sharing application which allowed users to trade (possibly illegally acquired) music files stored in the MP3 file format. At its peak, the Napster network provided more than 2.79 billion files [King01] to some 13.6 million users [BBC02]. Napster ceased to exist in its unrestricted form in 2000

after a much publicized court battle with the RIAA (Recording Industry Association of America).

One of the major factors in Napster's initial success, and the cause of its later downfall was the fact that Napster was a centralized system. Upon joining the Napster network, a list of all songs possessed by a connecting user gets transmitted to a master song list stored on a cluster of centralized servers. Upon disconnection, these songs are then removed from the master list.

The benefit of Napster's centralization was that users' searches for songs were communicated directly to a centralized server, and not to other peers in the network. The central server returned any matches to the querying user in terms of the name of the file, its length, and most importantly, the IP address of the peer possessing that file. To receive a copy of a desired song, peers initiated direct communication with the peer offering that file, thus minimizing the amount of traffic on the network.

Unfortunately, the use of central servers also created a single point of failure in the Napster network. In Napster's case, this failure point was exploited not by malicious hackers, but by the RIAA who won their court battle to have the central servers shut down. Recently, Napster has been reincarnated as a pay-as-you-go style music distribution system, which highlights the vulnerability of centralized peer-to-peer systems to control by a minority!

2.1.2.2 Gnutella

Among the first decentralized peer-to-peer file sharing applications was Gnutella [Gnu] [Ora01] (pronounced new-tella). The Gnutella application is open source, and makes use of the Gnutella protocol, which was one of the first broadcast flooding protocols. Gnutella was initially created by employees of a company called Nullsoft in 2000. The open source community took over its development after Nullsoft's parent company (AOL) shut it down due to piracy concerns.

Gnutella essentially serves as a testing platform to determine the viability of decentralized peer-to-peer systems on the internet. The fact that Gnutella is open source has allowed several researchers to implement 'crawlers' which traverse the Gnutella network. These crawlers are useful for collecting insightful information about the structure and scalability of peer-to-peer networks as well as measuring typical behaviour patterns of their users. One of the papers cited most often by peer-to-peer researchers is the paper *A Measurement Study of Peer-to-Peer File Sharing Systems* [SGG02]. This paper uses two different crawlers to provide a wealth of knowledge about network and usage characteristics of both the Napster and Gnutella networks.

The fragmentation of the Gnutella network into several disconnected pieces in August 2000 provided some important insight into the scalability of FB protocols. It

was determined through analysis [Clip2-00] that the fragmentation was the result of network traffic exceeding the capabilities of dial-up users. Since then, much effort has been put into researching enhancements to FB protocols to reduce the amount of traffic they generate. Examples of these enhancements include pong-caching [Roh], flow control [Roh02b] and ultra peers[4] [SR02].

2.1.2.3 FastTrack

FastTrack is the protocol behind the Kazaa [Kaz] which is currently the most popular file sharing application, offering more files to more users than Napster at its peak [Wik04a]. FastTrack is based on the Gnutella protocol with the added concept of SuperNodes. SuperNodes are essentially peers in the network with high bandwidth and uptime[5]. The idea is that peers with slower connections or low uptimes are connected to SuperNodes, and route all of their searches through them. SuperNodes in turn only route searches to other SuperNodes, reducing the strain on the network, and improving scalability.

Specific details of the FastTrack protocol are difficult to obtain because it is not open source. In fact, FastTrack uses encryption to prevent non-paying authors of client software from connecting to their network. A moderate amount of success has

[4] Ultra peers are synonymous to the super nodes described in Section 2.1.2.3 FastTrack
[5] Uptime is the amount of time a peer is connected to the network.

been achieved by open source developers in reverse-engineering the FastTrack protocol. More details are available at the giFT website [GIFT].

2.1.2.4 Freenet

Freenet [Free] is an open source, decentralized peer-to-peer system that primarily aims to tackle the problems of maintaining anonymity for its users, and preventing censorship of data, but also makes great strides towards achieving scalability [CSW01] [Ora01].

Freenet routes data in a different manner from typical broadcast flooding protocols. A node that receives a query for a key (associated with a document), forwards the query to a single node in its neighbourhood that is believed to be 'closest' to the search key. Thus, Freenet processes searches in a depth first manner, while Gnutella uses a breadth first approach.

Intelligent routing is achieved in Freenet through the use of a data store located at each node. Each entry in this data store contains a key, maybe the associated document, and the address of the node from which the document originated. When a query (by key) arrives at a Freenet node, it checks to see if it has the document associated with the key in its data store. If the document is found, it is returned; otherwise the query is forwarded to the node associated with the closest key in the data store. In the event that a document is found, it is returned back through the chain

of nodes through which the query originated. Along the way, the document is stored (cached) in the data store of each node in the chain. The data store is essentially a stack; older documents in the stack are deleted, leaving only keys and node addresses at the bottom of the stack.

The efficiency of Freenet lies in the tendency for nodes to spontaneously develop a specialization in areas of the key space. This is achieved through the use of the routing table and caching process mentioned above. The caching process also provides the benefit of making document availability proportional to its popularity. This is useful to counteract the so called 'slash-dot' effect[6] seen on the World Wide Web. Freenet also prevents censorship of data since no node can control which area of the key-space it specializes in, this is determined by the data stores of other nodes. Anonymity is achieved by only including the address of the node that forwarded the query at each step. Tampering with documents is also prevented through the use of content hash keys or signed subspace keys which represent a unique signature of the document, and allow the original author to prevent modification.

Freenet represents a form of halfway point between FB and DHT protocols. Like pure FB protocols, Freenet is not guaranteed to return a match for a query, and requires minimal traffic for peer arrivals or departures. However, like DHT protocols,

---

[6] The slash-dot effect refers to the popular website slashdot.org which periodically posts information of interest to many readers. In some cases new information generates such a large amount of traffic of readers that the web server cannot cope and readers are unable to get through.

Freenet imposes some form of organization on data using keys (unfortunately key-word searching is not supported), and typically minimizes the amount of traffic required to find a match for a query.

2.1.2.5 Pastry

Pastry [Pas] [RD01] is a decentralized, distributed hash table protocol. Every node in a Pastry network is assigned a unique identifier (NodeID) randomly chosen from a 128 bit circular identifier space. Queries in Pastry are issued for a 128 bit key, and are routed towards the node which possesses a NodeID closest to this key. In order to efficiently route queries, each node maintains a routing table, a neighbourhood set and a leaf set.

The routing table contains addresses (IP address) for other nodes whose NodeIDs are similar to the current node's in the first X digits. The size of this table is logarithmic (to the base of 2^b, where b is a user defined parameter) in the number of nodes in the entire system. Adjustments in the value of the parameter b allow for a trade-off between the table size and the number of nodes it is necessary to visit in order to complete a query.

The neighbourhood set contains the addresses and NodeIDs of nodes which are closest to the current node in terms of a user definable proximity metric. This

information is used by Pastry to attempt to ensure that nodes listed in the routing table are always as near (according to the proximity metric) to the current node as possible.

The leaf set contains a certain (user definable) number of NodeIDs and addresses of other nodes having the closest NodeID to the current node. Half of the nodes in the leaf set have smaller NodeIDs than the current node, and half have larger.

When a query arrives at a node, the node first checks to see if the key is in the range of the NodeIDs stored in the leaf set, in which case the query is forwarded to the node therein with NodeID closest to the key. If the key is outside of the range, the node computes the number of common digits between the key and the current node's NodeID. The query is then forwarded to a node in the routing table whose NodeID is at least one digit closer to the key than the current node's.

A search for a valid key will almost always return successfully, and on average requires contacting *O(logN)* other peers, although it should be noted that this can be *O(N)* in the worst case when several nodes fail simultaneously. Pastry can also be somewhat cumbersome in terms of storage since the routing table can require *O(logN)* space for the routing table at each node.

One of the main disadvantages of Pastry is that the arrivals/departures of nodes to/from the network require *O(logN)* message between *O(logN)* peers in order to fill at least *O(logN)* rows of the routing table.

2.1.2.6 Others

The protocols listed in the previous sections give a fairly good representation of the different classifications of peer-to-peer protocols, however many other protocols exist. Some of the better known protocols not mentioned here are CAN [RFH+01], Chord [Cho], Kademlia [Kad], NeuroGrid [Neu], P-Grid [PGr] and Tapestry [Tap]. Wikipedia also maintains a fairly comprehensive list of many peer-to-peer protocols and some of the applications that use them [Wik04b].

2.2 Network Topology

A real world peer-to-peer system or network is made up of a collection of peers which interact with each other through an underlying infrastructure called a *network topology*. In order to simulate a peer-to-peer network, it is important to use an accurate and realistic representation of the network topology. This section begins by describing some of the different data structures that can be used to represent the network topology. Later on in this section, some of the known structural properties of real-world peer-to-peer network topologies are described, followed by a description of some of the methods for generating network topologies that exhibit these properties.

2.2.1 Representation

In order to simulate a peer-to-peer network, it is necessary to somehow represent this network topology inside the simulator. The standard approach for representing network topologies is to make use of the mathematical discipline of **graph theory** [Eve79]. In graph theory, a **graph** is defined as a collection of points or vertices (**nodes**) and lines connecting these points (**edges**). Figure 2.2 illustrates these terms. In the context of peer-to-peer or other networks, peers or computers in the network can be represented by nodes, and the communication links among them can be represented by edges. [Ora01]. For the remainder of this document the terms peer-to-peer community, network topology and graph will be used interchangeably.

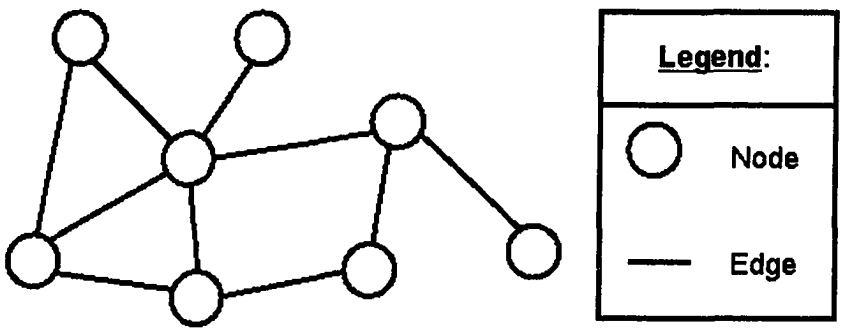

Figure 2.2 An example of a graph with nodes and edges

Two popular data structures that exist for representing graphs are the Edge-List and the Adjacency Matrix [CLR90]. These Data Structures will be described and compared in the following sections.

2.2.1.1 Edge-List Data Structure

As its name implies, the Edge-List (sometimes called an Adjacency-List or Incidence-List) data structure represents the graph using lists of edges for each node. Each of the N nodes in the graph maintains a list, containing references to each of the edges connected to that node. Fast edge insertions and removals can be achieved with this data structure through the use of doubly linked lists (c.f. half edge data structure).

2.2.1.2 Adjacency Matrix Data Structure

For the Adjacency Matrix data structure, an N x N matrix is used to store the M edges between the N nodes. Each node is indexed to both a row and column of the matrix. For example, if node 4 has an (undirected) edge with node 7, then the appropriate data (usually a Boolean flag) is placed in the matrix at row 4, column 7 and row 7, column 4. Node insertions and deletions are costly in this structure since the entire matrix must be copied in order to insert or delete rows and columns.

2.2.1.3 Comparison

The Edge-List is the more popular of these two data structures largely due to its more efficient storage. Node insertions and deletions also can be performed much more efficiently with the Edge-List as illustrated by the Table 2.1.

Table 2.1 Asymptotic run times of operations on Edge-List vs. Adjacency Matrix

Operation	Edge-List	Adjacency Matrix
Storage	O(N+M)	$O(N^2)$
Node insertion time	O(1)	$O(N^2)$
Node deletion time	O(1)	$O(N^2)$
Edge finding time for a node with E edges. (other end node provided)	O(E)	O(1)
Edge insertion time	O(1)	O(1)
Edge deletion time for a node with E edges.	O(E) if only the other end node is provided. O(1) if actual edge is provided	O(1)

2.2.2 Structural Models

The recent rise in popularity of peer-to-peer networks and the internet has motivated researchers from backgrounds ranging from mathematics to the social sciences to study the structural properties of the network topologies that these networks form. The models devised by these researchers provide valuable insight for simulating or approximating real networks. The following sections introduce and describe the two structural models that have been associated with peer-to-peer networks: Small World networks and Power Law networks.

2.2.2.1 Small World Networks

Researchers in the peer-to-peer field have observed that in practice, peer-to-peer networks tend to form **Small World Networks** [Ora01]. Small world networks is a

term coined by Watts and Strogatz in their paper *Collective dynamics of 'small-world' networks* [WS98]. The name for this type of network was inspired by the research of Milgram of Harvard University in the 1960's. Milgram performed an experiment, in which he addressed 160 letters to randomly chosen people, and then mailed them to a different set of 160 randomly chosen people. The initial recipients of the letters were asked to forward their letter to a person in their acquaintance whom they considered mostly likely to know the final recipient. 42 of the 160 letters made it to their final recipient, using an average number of approximately 5.5 intermediaries (hence the term Six Degrees of Separation). This was the first concrete demonstration of the popular phenomenon known as the small-world effect (It's a small world isn't it?!). Small world networks are networks which on average require traversing a small number of edges to connect any two nodes.

The authors determined that there are two measurements of structural properties of these types of graphs that are particularly interesting, the **characteristic path length**, and the **clustering coefficient**. The characteristic path length is defined by Watts and Strogatz as 'the number of edges in the shortest path between two vertices, averaged over all pairs of vertices'. The clustering coefficient is a measure of the average 'cliquishness' of all the vertices. Cliquishness is measured for a vertex v with k neighbours by calculating how many edges exist between all k vertices in the neighbourhood, divided by the total possible number of edges between vertices in the

neighbourhood. The total possible number of edges in a neighbourhood of k vertices is k(k-1)/2 when every vertex is connected to every other.

2.2.2.2 Power Law Networks

Researchers in the peer-to-peer field have also noticed that peer-to-peer networks tend to form **power law networks**. The line between small world and power law networks is grey as opposed to black and white. Graphs can fall into any combination of these categories (both, one or the other, or none).

The first connection between internet topologies and power law networks was made by Faloutsos et al. in their paper *On Power-Law Relationships of the Internet Topology* [FFF99]. A power law graph (network) is one in which the majority of nodes have small degree, and only a few nodes have high degree (the degree of a node is the number of edges connected to it.).

More formally, the fraction of nodes N in a power law random graph with degree greater than D can be expressed by the function $N = D^{-k}$, where k is a network specific constant. It has been observed in practice that the value of k is typically between the values of 1 and 4 [BT02] [BA99]. Such Power law functions are recognizable by the appearance of a straight line when N is plotted versus D on a log-log scale, as illustrated by Figure 2.3.

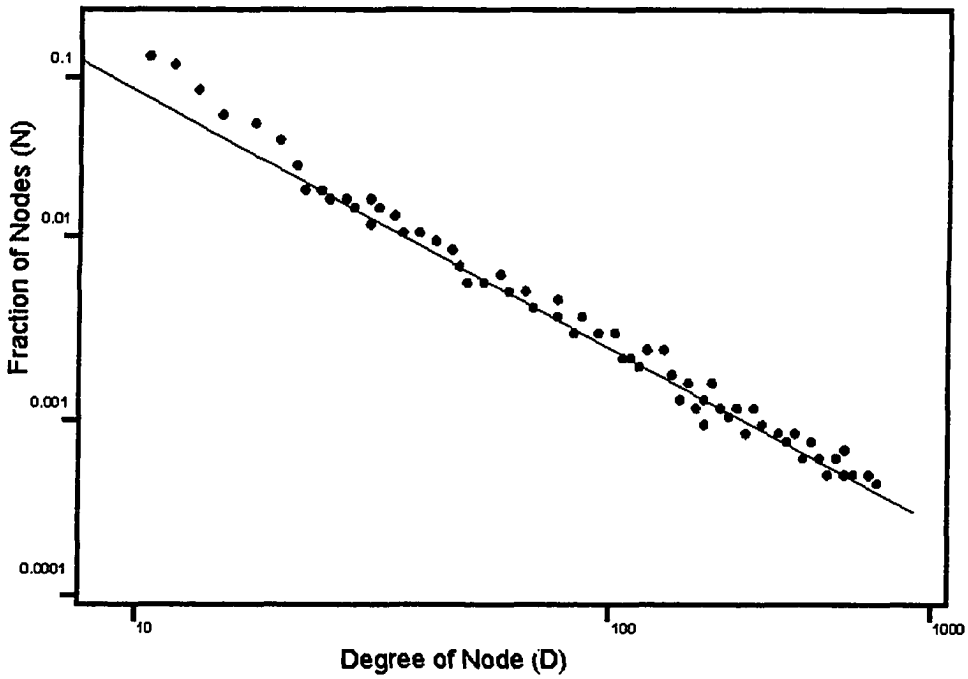

Figure 2.3 Example of a Power Law Distribution plotted on a log-log scale

It should be noted that there has been some debate as to whether peer-to-peer networks form power law networks. For example Mihajlo et al. write: "Upon analysing the Gnutella topology data obtained using our network crawler, we discovered it obeys all ... these power-laws" [JAB01]. While Ripneau et al. using their own Gnutella crawler observe that "there are too few nodes with low connectivity to form a pure power law network." [RFI02].

Power law networks are often criticized for their vulnerability to malicious attacks on highly connected nodes, yet fare much better in the face of random node failures. Saroiu et al. observe from their Gnutella crawler data that "the network fragments only when 60% of the nodes breakdown", but also mention that "This

removal of [~5% of the more highly connected nodes] has effectively shattered the overlay into a large number of disconnected pieces" [SGG02].

2.2.3 Initial population of peers

In order to obtain realistic statistics and results from a peer-to-peer simulation, it is important to realistically model the community in which peers reside. To this end, the standard practice is to load a ready made graph with desirable structural properties into the simulator prior to the simulation run.

In some cases, the ready made graph is generated by the simulator during initialization. However, in most cases, a stand-alone topology generator tool is used to generate the graph, which is then loaded into the simulator via a standard graph file format. The following sections discuss existing models for topology generation, followed by a brief description of existing topology generator tools.

2.2.3.1 Topology Generators

In their paper [TGJ+01], Tanguminarunkit et al. propose grouping topology generators into three families: *random graph* generators, *structural* generators and *degree-based* generators[7]. Random graph generators are the simplest and oldest family of generators. Structural generators are the next oldest family group, and generators

[7] All of these families of generators produce random graphs in the sense that their exact layout is not predictable.

from this family focus on attempting to create a hierarchical topology structure. Degree-based generators are typically newer, and generators from this family focus on the degree distribution of nodes in the generated topology.

2.2.3.1.1 Random Graph Generators

Primitive random graph generators generate graphs by assigning a uniform probability for creating a link between any pair of nodes. The first known random graph generator was proposed by Erdos and Renyi in 1960 [ER60]. The Waxman generator is perhaps the most well known random graph generator. The Waxman generator is almost as simple as the Erdos and Renyi version, except that nodes are assigned randomly to positions in the plane, and the probability of connecting any two pairs of nodes is proportional to their Euclidean distance from each other. [WAX88]. Several variations of the Waxman generator have been developed which make slight alterations to the probability distribution [ZCB96].

2.2.3.1.2 Structure-Based Generators

Structure-based generators attempt to enforce a rigid structure or hierarchy onto the graphs they generate. Two models have been proposed for developing structure-based generators; the *Transit-Stub* model [ZCB96] and the *Tiers* model [Doa96].

The **Transit-Stub** model uses the idea of constructing portions of the graph randomly and then connecting these portions together in a fashion that mimics the structure of the internet. More specifically, the internet can be viewed as a collection of routing domains, where most traffic between two nodes in a domain stays inside that domain. The Transit-Stub model emulates this by classifying portions of the generated graph as transit domains (no restrictions) or stub domains (every path between any pair of nodes must be internal). Figure 2.4 illustrates the different domains of the Transit-Stub model.

Generating graphs using the Transit-Stub model involves generating a random connected graph to represent the connections between transit domains. Each node in this graph is then replaced by a random connected graph which makes up the content of the transit domain. The next step is to generate a number of random connected graphs (stub domains) for connection to each transit domain node. In a final step, a number of extra links are created between transit domains and stub domains or between stub domains.

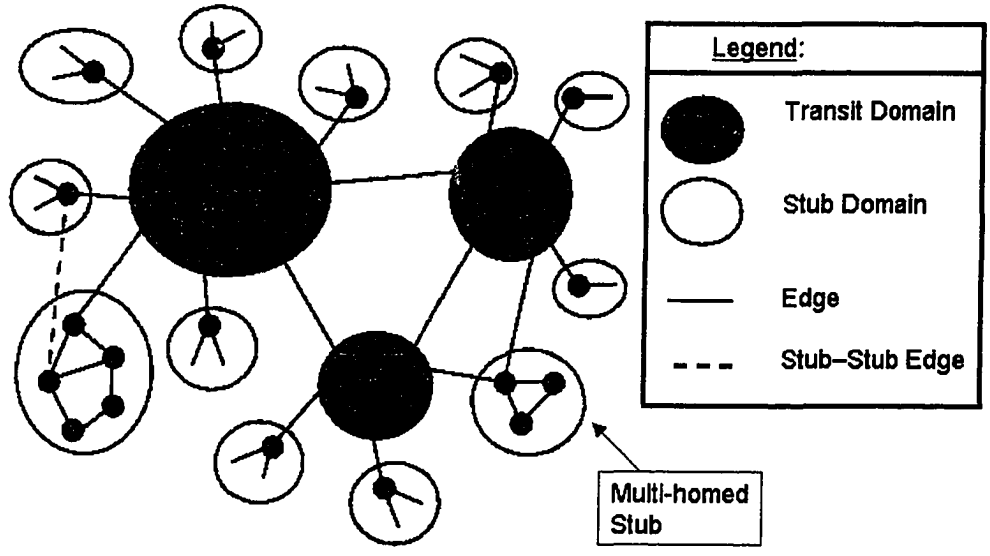

Figure 2.4 Illustration of the Transit-Stub model

The **Tiers** model is similar to the Transit Stub model. In fact, the authors of both the Transit-Stub and Tiers models compare their models in [CDZ97]. The Tiers model replaces the transit and stub domains with 3 levels: WAN, MAN and LAN[8]. The Tiers model also employs a minimum spanning tree algorithm to connect the nodes in sub-graphs and nodes at the LAN level are organized into a star topology. In the final step, 'redundant' edges are added according to a distance metric instead of randomly.

2.2.3.1.3 Degree-Based Generators

Several models have been proposed for degree-based or power law topology generators, many of which are summarized by Bu et al. [BT02] and Tangmunarunkit

[8] Wide, Metropolitan and Local Area Networks

et al. [TGJ+01]. The two main types of models are the *growth* model and the *distribution* model.

Power law topology generators using the **growth** model typically start off with a small graph, and then grow this graph incrementally until the desired number of nodes is reached. The basic ideas behind this model are those of *incremental growth* and *preferential connectivity.*

Incremental growth refers to graphs that are formed by the continual addition of new nodes [MMB00]. At each step, a probability function is used to determine which node(s) a new node will connect to. There are several variations of incremental growth which specify different rules for how many edges to form with the new node, and where to connect them. In some variations of the model, edge addition or rewiring can occur independently of node additions [BA99] [BA00].

Preferential connectivity refers to the preference of new nodes to be connected to existing nodes which are highly connected (have high degree) or popular [MMB00]. Variations on preferential connectivity include schemes for incorporating distances between nodes as part of the probability function.

Power law topology generators using the **distribution** model differ from those using the growth model in that all of the nodes have pre-defined degrees (drawn from a power law distribution) and are present from the graph from the start. Building the

graph is now a question of how to distribute the edges to the nodes in order to match their degrees.

One approach used by Aiello et al. [ACL00] is to make x copies of each node, where x is the degree of each node, and then compute and use a matching of this set in order to determine which nodes to connect with edges. The disadvantage of this approach is that the generated graph may contain duplicate or redundant edges, and the graph is not guaranteed to be connected.

2.2.3.1.4 Small World Generators

Small World networks are often lumped into the same category as power law networks, but the generation of small world networks does not seem to fit into any of the previous families of generators. It is interesting to note that Bu and Towsley use the characteristic path length and clustering coefficients as 'metrics for distinguishing between power law graphs produced by different topology generators' [BT02]. For completeness, this section will describe one technique for generating small world graphs.

In their paper [WS98], Watts and Strogatz noticed that in previous works 'Ordinarily the connection topology is assumed to be either completely regular or completely random. But many biological, technological and social networks lie

somewhere between these two extremes'. Figure 2.5 gives an example of both a regular and random graph.

In order to bridge the gap between regular and random graphs, the authors propose a technique for randomly 'rewiring' a regular (ring lattice) graph. Each edge in the regular graph is rewired at random with probability p ($0 < p < 1$). Rewiring an edge at random involves randomly selecting at least one new end node for that edge. This technique was modified by the authors in a later work [NW99] to eliminate the possibility of creating disconnected graphs. To this end, the rewiring procedure was modified so that edges are added instead of moved.

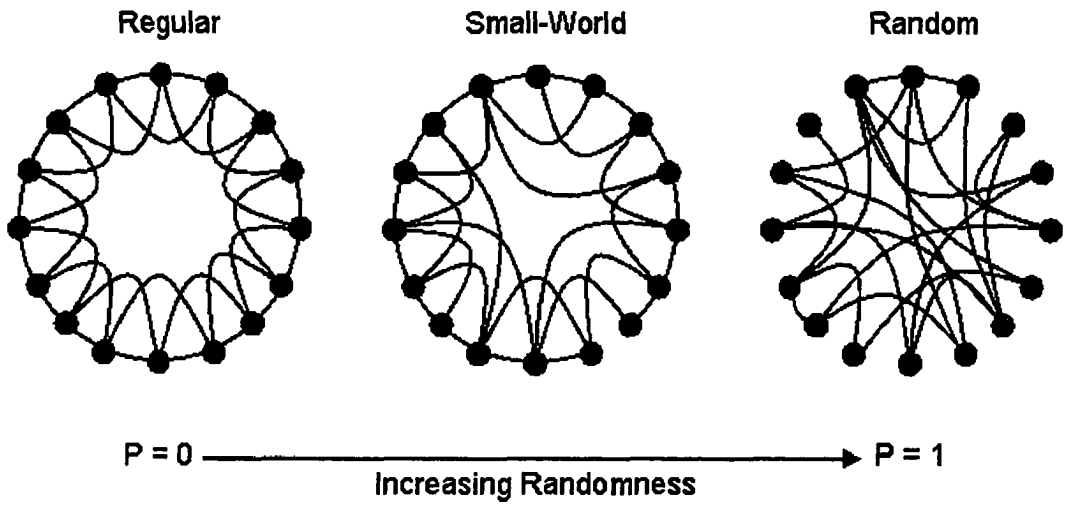

Figure 2.5 Illustration of rewiring process for generating Small World Graphs

Regular graphs typically have long characteristic path lengths and low clustering coefficients, and vice versa for random graphs. Watts and Strogatz noticed

that sampling the probability p (mentioned above) in the range between 0 and 1, they obtained (small world) graphs that were 'highly clustered like a regular graph, yet with small characteristic path length, like a random graph'.

2.2.3.1.5 Comparison

While the random graph family of topology generators is clearly insufficient for accurately modeling today's internet topologies, the subject of which of the other generator families is most appropriate is the subject of much debate [TGJ+01] [JCJ00] [MMB00]. Since degree-based topology generators are newer, the implication is that they are the most appropriate for modeling internet (and therefore peer-to-peer) networks.

Tangmunarunkit et al. [TGJ+01] study the differences between structure and degree based topologies, and ask the question 'Is there any relationship between hierarchical structure and power-law degree distributions?' In answer to their own question, they write: 'while degree-based generators do not explicitly inject hierarchy into the network, the power-law nature of the degree distribution results in a substantial level of hierarchy – not as strict as the hierarchy present in the structural generators, but significantly more hierarchical than, say, random graphs.'

2.2.3.1.6 Existing Topology Generator Tools

This section provides a list of some of the better known network topology generator tools. In Table 2.2, the name of each topology generator tool is provided, along with a reference to a paper describing it (where available), and summary of which of the models from the previous sections are supported.

Table 2.2 Summary of existing topology generators

Name of Tool	Reference	Models supported
BRITE	[MMB00]	Random, Structure-Based, Degree-Based
GT-ITM	[CDZ97]	Random, Structure-Based
Inet	[JCJ00]	Random, Structure-Based, Degree-Based
Tiers	[CDZ97]	Structure-Based

2.2.4 Evolution

Real world peer-to-peer networks are dynamic in nature. The structure of these networks is constantly evolving as new users join or leave the network, forging or breaking links between the peers. The following sections describe the two scenarios that cause the network topology to evolve: Peer arrivals and departures, and Peer discovery. The discussion of how to model the occurrence of these scenarios or events is relegated to section 2.4.3.2 Arrival and Departure.

2.2.4.1 Peer Arrivals and Departures

Peers are typically free to join or exit the community at any time[9]. This corresponds in real life to users opening or exiting a peer-to-peer application, or rebooting or powering off their computers. Several events (computer or network errors) can even cause users to disconnect unintentionally.

An accurate peer-to-peer simulator must provide mechanisms for representing this form of peer behaviour. This can be accomplished via node insert/delete operations on the corresponding graph. A recent measurement study of the Gnutella network showing that most users connect to the network for a period of less than an hour [SGG02], emphasizes the need to make the implementation of these operations as efficient as possible in the simulator.

During a simulation run, we should not be solely concerned with modeling the rate of peer arrivals and departures, but also with how they select a connected peer with which to make first contact. Ideally, node insert operations should be orchestrated so as to preserve the existing structural properties of the graph; however most existing peer-to-peer protocols make use of a host cache containing a fixed list of peers with which to connect.

---

[9] The process/rate of peers arriving/departing to/from the network is often called churn/churn rate.

2.2.4.2 Peer Discovery

Peers cannot form a community network without some knowledge of each other's presence. In real peer-to-peer applications, each peer is aware of a (small) subset of peers from the community, referred to as that peer's **neighbourhood**. Changes to a peer's neighbourhood can occur for a variety of reasons, such as new peers joining or leaving the network, peers contacting another peer during a search, periodic search for new or faster peers to connect to (a.k.a. topology adaptation) or periodic pruning of unresponsive neighbours.

A peer's neighbourhood is analogous to the set of edges stored by its node in the graph. Over time, peers can 'discover' new peers or drop existing peers from their neighbourhood. The specifics of this type of behaviour can vary, depending on which peer-to-peer protocol is in use.

Peer discovery or dropping of peers from a peer's neighbourhood can be represented in a simulator via edge insertion/deletion operations on the corresponding graph. A higher frequency of changes to peers' neighbourhoods than of arrivals or departures of peers from the network, would suggest that more priority should be given to the efficiency of edge operations than to node operations.

2.3 Communication

Peer-to-peer protocols define the language that is spoken in conversations between peers, and the network topology provides the details of whom a peer may converse with. What are missing are the details about how the conversations get sent across the network. This section discusses some of the issues involved in communications between peers that are not covered by the previous sections.

2.3.1 Overlay Network

Peer-to-peer networks on the internet form what is called an *overlay network* [Ora01]. An overlay network is a network which is overlaid on top of another network, essentially forming a subset of this network. At the internet level, a link connecting two nodes on the internet may pass through several other nodes, and many such links or paths may exist between two specific nodes. In an overlay network, these two nodes appear to be connected directly by one link. In reality, any communication over this single link will follow one or many of the paths available between the two nodes on the underlying network.

This design means that the speed of communicating a message between two nodes in an overlay network is dependant to a large extent on the amount of message traffic on the underlying network (the internet). At any given time, the underlying network may be in use by other protocols, such as for browsing the World Wide Web

or downloading files. This type of activity can be simulated by periodically increasing or decreasing the bandwidth (see the next section) of certain links of the overlay.

For the purposes of accurately simulating a peer-to-peer system, the question is whether the goal is to measure the absolute or relative performance of different protocols [CRB03]. In most cases, the primary concern is to measure the relative performance of protocols, and as such modeling the background traffic on a network overlay is not necessary.

2.3.2 Bandwidth Modeling

In real communication networks, we must define the concept of the bandwidth of connections between any two nodes. If we imagine that the underlying network topology is a system of interconnected water pipes, then the *bandwidth* [Ora01] is the width of these pipes. The bandwidth of a pipe determines the maximum amount of water (data) that can pass through it at any given moment in time, which in turn determines how long water (data) takes to get from one end to the other. The *available bandwidth* of a pipe corresponds to how much room is left in the pipe when some water is already flowing through it. When pipes are interconnected in sequence, the amount of water that can be channelled from one end to the other is limited by the narrowest (least available bandwidth) pipe. In communication networks, these 'narrow

pipes' that limit the flow of data are called *bottleneck links* [GB02]. See Figure 2.6 for an illustration.

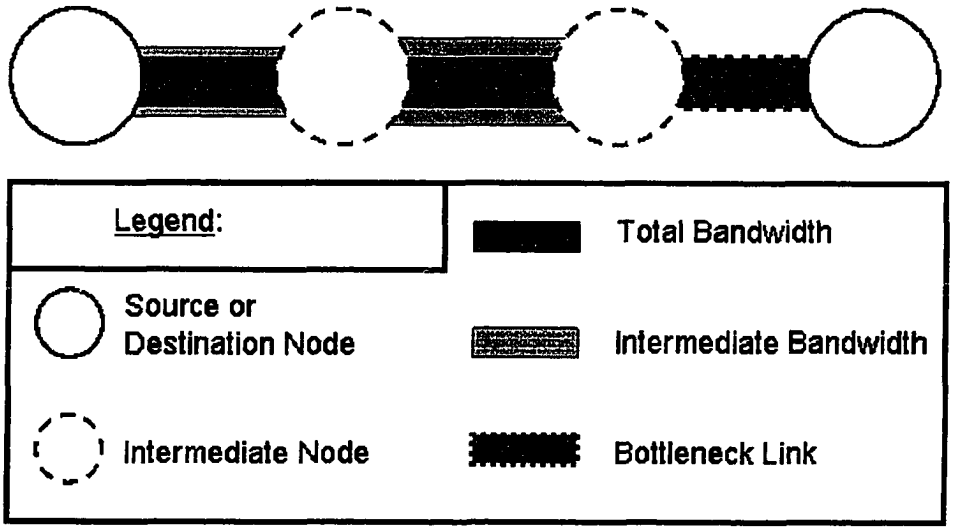

Figure 2.6 Illustration of bandwidth and bottleneck links

In the context of an overlay network, the bandwidth available for a connection between two nodes corresponds to the sum of the bottleneck bandwidths of each of the distinct paths between the same two nodes in the underlying network. Deciding whether or not to account for this added complexity in modeling bandwidth is again a question of whether absolute or relative performance of a protocol is to be measured. For comparing the relative performance of one protocol to another, it is often sufficient to simply model the overlay network, and not be concerned with the complexities of the underlying network. This is supported by observations in the

literature [GB02] [ILE02] that the bottleneck link for any given node in an overlay network usually corresponds to the link closest to it in the underlying network. For example, a dialup user's bandwidth is constrained by the speed of the phone-line connection to their internet service provider.

A typical approach to representing the bandwidth available to inter-node links is to associate a maximum and available value with the corresponding edge[10] in the network topology representation. Initializing the representation with maximum bandwidths can be done at random, or can be biased towards certain capacities. In today's world, this might correspond to allocating a certain percentage of the bandwidths to model dial-up users and another percentage to cable modem/DSL users etc...

Consumption of bandwidth occurs as nodes progressively engage in more and more communication. The time for each communication to be completed can be determined by the amount of available bandwidth. When communication begins the available bandwidth on a link is decremented. When the communication is complete, the available bandwidth is incremented. The trick is to share the available bandwidth fairly when more than one message is being communicated.

---

[10] Some peer-to-peer simulators use a simplified model where bandwidth is associated with the nodes. The bandwidth available for communication across a link from that node is the minimum of the available bandwidth of both end nodes of the link.

2.3.3 Transporting Messages

On the internet, messages between nodes (peers) are transported using *network sockets*. Network sockets are essentially channels of communication that are opened between two nodes, using the TCP/IP protocol. There is a large amount of overhead associated with the use of sockets, and as such their use makes little sense in the context of a simulation where all the peers reside on one computer. The only exception might be if the goal is to build a simulator that is also deployable as a real application.

In simulation, the typical approach for transporting messages between nodes is to use simple method or function calls. For example, in an object-oriented setting, if peer A wishes to send a message to peer B, then peer A calls peer B's receive Message method, passing it the contents of the message. This simple approach can be extended by using Remote Method Invocations (RMI) or other similar standards for simulators in distributed, multi-threaded or parallel settings. (See section 2.4.4.1 Parallel, Sequential, Distributed or Multi-Threaded)

2.3.4 Communication Errors

On the internet or other similar networks, communication errors can occur at any time, greatly impacting the time taken to transport messages. In some extreme cases, communication errors may actually prevent a message from being transported at all.

When simulating peer-to-peer protocols, it is important to evaluate their resilience in the face of communication delay or failure. Communication delay on a link between peers is relatively easy to simulate by simply increasing the transfer time of all messages being transported by that link. Critical communication errors can be simulated by simply suppressing a message while it is in transit.

2.3.5 CPU Delay

Some scenarios require processing a message before or after its communication between nodes. For example, a peer receiving a search message may require time to check if it has the matching document, or to search through its routing table to find the appropriate peer to forward the search to. Another example is the encryption and decryption of messages before and after its transmission. Encryption can be particularly computationally expensive, resulting in lengthy delays.

Such delays incurred by the processing speed of the CPU of a node on the network can also be simulated by delaying messages in transit on the appropriate link.

2.3.6 File Transfers

The transfer of files between nodes on a network is usually the lengthiest form of communication that can occur. As such, this form of communication is typically most prone to failure of some form. Deletion of the file by the transmitter or cancellation by

the receiver can also cause the communication to be terminated. In some cases, files can be transferred simultaneously from multiple sources providing extra resilience, yet requiring careful management. The time taken to transfer a file between one peer and another can be simulated proportionally to the bandwidth of the link(s) it traverses.

2.4 Simulation

This section describes some of the important topics with respect to the implementation of a peer-to-peer simulator. Section 2.4.1 introduces some general concepts related to simulation, while Section 2.4.2 explains some of the standard methods for processing the events that occur during simulation. Section 2.4.3 explains some methods for modeling the content and behaviour in a simulation. Methods for assigning content (files) to peers are discussed, along with methods for determining the rate of their behavioural events. Finally Section 2.4.4 provides some discussion about the important issue of scalability for a peer-to-peer simulator.

2.4.1 Simulation Concepts and Terminology

Banks et al. [BNC00] define *simulation* as "the imitation of the operation of a real-world process or system over time". Simulation involves the development of a *simulation model* representing the system which is used to study the behaviour of this system over time, and to "draw inferences concerning the operating characteristics of

the real system". In the peer-to-peer context, the system is the real-world peer-to-peer network, and the model is the simulated peer-to-peer network.

Banks et al. define several classifications of simulation models. The model may be classified as static or dynamic, deterministic or stochastic, or discrete or continuous. A static model represents a system at a specific point in time, whereas a dynamic model represents a system as it changes over time. A deterministic model is one with no random variables, whereas a stochastic model allows random variables (such as random arrivals) as input. A discrete model is one in which the system changes only at a discrete or countable set of points in time, whereas a continuous model allows the system to change continuously over time (e.g. water in a tank continuously evaporating).

Since peer-to-peer networks are constantly changing in an unpredictable fashion, most peer-to-peer simulators use a dynamic and stochastic simulation model. Furthermore, since all of the activities in a peer-to-peer network occur at specific points in time, the common practice is to use a discrete simulation model for peer-to-peer simulation.

The reader is referred to the simulation literature [BNC00] [Ros02] [Tys99] for a more in depth discussion of simulation models.

2.4.2 Event Processing

In order to study the behaviour of a peer-to-peer simulation model over time, it is necessary to develop a framework that models how different events affect the state of the model over time. According to Banks et al., an *event* is defined as "an instantaneous occurrence that may change the state of the system" [BNC00]. They define the *state* of the system as "the collection of variables necessary to describe the system at any given time". The state of a peer-to-peer system is represented by the network topology (Section 2.2 Network Topology) and variables associated with the nodes therein. Banks et al. define two types of events, *endogenous events* and *exogenous events*. Endogenous events are events that occur within the system such as the forwarding of a query from one peer to another. Exogenous events are events that occur in the environment (outside the system) that affect the system such as a peer joining the network. The following sections describe the standard approaches for processing events which affect the state of the simulation model.

2.4.2.1 World Views

In the field of simulation, there are two principle approaches for scheduling and processing events, *activity scanning* and *event scheduling* [BNC00] [Tys99]. Another approach called *process interaction* is a specialization of the event scheduling approach. Banks et al. refer to these approaches as *world views*, because the approach

that is used determines from what perspective or world view events are perceived and handled.

2.4.2.1.1 Event Scheduling

Event scheduling is the predominant world view used by simulation programs, and "constitutes the cornerstone of discrete-event computer simulation" [Tys99]. Event scheduling revolves around the concept of a *future event list* which stores all the events that are scheduled to occur in the future [BNC00]. All events in the future event list are arranged chronologically by time. While some events start and end at the same time, other events denote the beginning of an activity (such as downloading a file) which will end at some time in the future. The processing of an event which starts an activity requires placing an event which marks the end of the activity at the appropriate (future) time in the future event list.

Event scheduling relies on a simulation time clock to determine which events need executing at the current time. Assuming that the simulator has been initialized, the sequence of actions to be performed using the event scheduling approach is provided by Banks et al.:

> Step 1: Remove the first (imminent) event from the future event list.
> Step 2: Advance the simulation clock's time to that of the imminent event.
> Step 3: Execute the imminent event, updating system state as necessary.
> Step 4: Generate any future events related to the imminent event.
> Place any such future events in sorted order in the future event list.
> Step 5: Update any relevant statistics.

This sequence of actions is performed repeatedly by the simulator until either no more events exist to be processed, or a pre-determined simulation clock time is reached.

An important issue when using the event scheduling world view is efficiently maintaining the sorted order of the future event list. A chronologically sorted future event list allows the next imminent event to be found quickly, but may require scanning the entire list when adding, deleting or rescheduling the time of events. A variety of data structures have been suggested for maintaining the future event list such as arrays, linked lists or binary heaps [BNC00] [Tys99].

Arrays are the simplest data structure to implement, yet can require scanning every event for each operation. Linked lists offer some improvement over arrays by allowing events to be removed or inserted in constant time, although the entire list needs to be searched to find the event to delete or position for a sorted insertion. Perhaps the most promising data structure for the future event list is the binary heap. Binary heaps storing N events require scanning at most log (N) events for every operation (including removal of the imminent event and insertion of an event in sorted order) except searching for a specific event, which still requires scanning all the events [CLR90].

2.4.2.1.2 Activity Scanning

Activity scanning is a world view which focuses on the activities or events of a model and the conditions which must be met for them to occur. At each time step, all of the entities (i.e. peers) in the simulation model are scanned to check if they satisfy the conditions for the occurrence of an event, in which case that event is executed. In contrast to the event scheduling world view which uses a variable time increment, the activity scanning world view "uses a fixed time increment and a rule-based approach to decide whether any activities can begin at each point in time" [BNC00].

It should be noted that with the activity scanning world view, the execution of an event for an entity may change the conditions of other entities, requiring repeated scanning of these entities. To combat this inefficiency, some hybrid approaches to activity scanning have been developed which borrow some properties of the event scheduling approach to reduce the number of scanning passes [BNC00] [Tys99].

2.4.2.1.3 Comparison

In the context of simulating large scale peer-to-peer networks, using the event scheduling world view is advantageous since at each time step, only events occurring at that time, or events created at that time must be processed. In contrast, the activity scanning world view requires examining every single peer in the network (possibly multiples) at each time step.

Banks et al. summarize some of the tradeoffs between event scheduling and activity scanning in the following passage:

> *Proponents claim that the activity scanning approach is simple in concept and leads to modular models that are more maintainable and easily understood and analyzed by other analysts at later times. They admit, however, that the repeated scanning to determine whether an activity can begin results in slow runtime on computers.* [BNC00].

2.4.2.2 Existing Simulation Packages

Some existing peer-to-peer simulators make use of simulation libraries or packages that provide implementations of the world views mentioned in the previous section. In general, the question of whether or not to use a simulation package depends on its abilities to match the needs of the simulator. A realistic peer-to-peer simulator needs to be able to handle an extremely large number of peers and a correspondingly high numbers of events. It is unclear whether any of the existing open source simulation packages have the required ability. Perhaps for this reason, the majority of existing peer-to-peer simulators rely on their own world view implementations.

2.4.3 Modeling Peer Content and Behaviour

In order for a peer-to-peer simulator to realistically compare the performance of different protocols, it is necessary to accurately model both the content that peers will offer, and the behaviour of peers with respect to the system. The following sections

describe methods for assigning initial content to the peers and methods for generating the exogenous events that affect the simulation model.

2.4.3.1 Content Distribution

The most basic approach for assigning initial content to peers is to form a pool of available content, and then just randomly distribute it among the peers. More complicated approaches involve ranking the pool of content and then using a so called 'heavy-tailed' distribution function such as Zipf or Pareto distribution [Ada].

Like power-law distributions, Zipf and Pareto distributions are used to "describe phenomena where large events are rare, but small ones quite common" [Ada]. If we assume that content is ranked according its popularity[11], then Zipf and Pareto distributions can be used to assign content to peers such that more popular files are quite common, and unpopular files are rare [SCK03].

Zipf's law allows us to determine how many copies of a file to create based on its rank. If files are each given a popularity ranking r, then the number of copies n can be determined by the function: $n \sim r^x$ (the value of x is typically close to one). Using Zipf's law, we can determine how many copies of files in the available content pool to make, and then randomly distribute the copies among the peers. Pareto's law differs

[11] The popularity of content may be defined in a number of ways such as the frequency with which it is requested by queries.

slightly from Zipf's law in that it allows us to determine the probability of there being more than r copies of a file with rank r.

2.4.3.2 Arrival and Departure Events

As discussed in section in section 2.2.4 Evolution, real peer-to-peer networks are in a constant state of flux; Peers are continuously joining and leaving the network. A simulator must provide some form of mechanism for determining the frequency of these kinds of 'random' exogenous events. In the context of simulation, Banks et al. state that 'the most important model for random arrivals is the Poisson arrival process.' [BNC00]. They also cite several examples of the Poisson process being successfully employed for modeling random arrivals of customers to a restaurant or telephone calls to a telephone exchange.

A Poisson process with rate λ allows us to determine the number of random events that will occur in a fixed time interval of length t. The number of random events per time interval will be biased towards the value λt (See Figure 2.7). More formally, the number of events in a time interval of length t, has the Poisson distribution with mean λ t events.

For x = 0,1,2,..., a Poisson distribution has probability mass function:

$$P[N(t) = x] = \frac{e^{-\lambda t}\lambda t^{x}}{x!}$$

Here P[N(t) = x] denotes the probability that the number of events N(t) occurring in an interval of time t is equal to x. Note that e is Euler's constant approximately equal to 2.7183 and λ and t are parameters. A useful property of the Poisson distribution is that both its mean and variance are equal to λt [12]. Also, Poisson distributions have the 'memory-less' property, that is that the numbers of events in separate time intervals are independent of each other, since they only rely on the rate λ and the length of the time interval t as parameters.

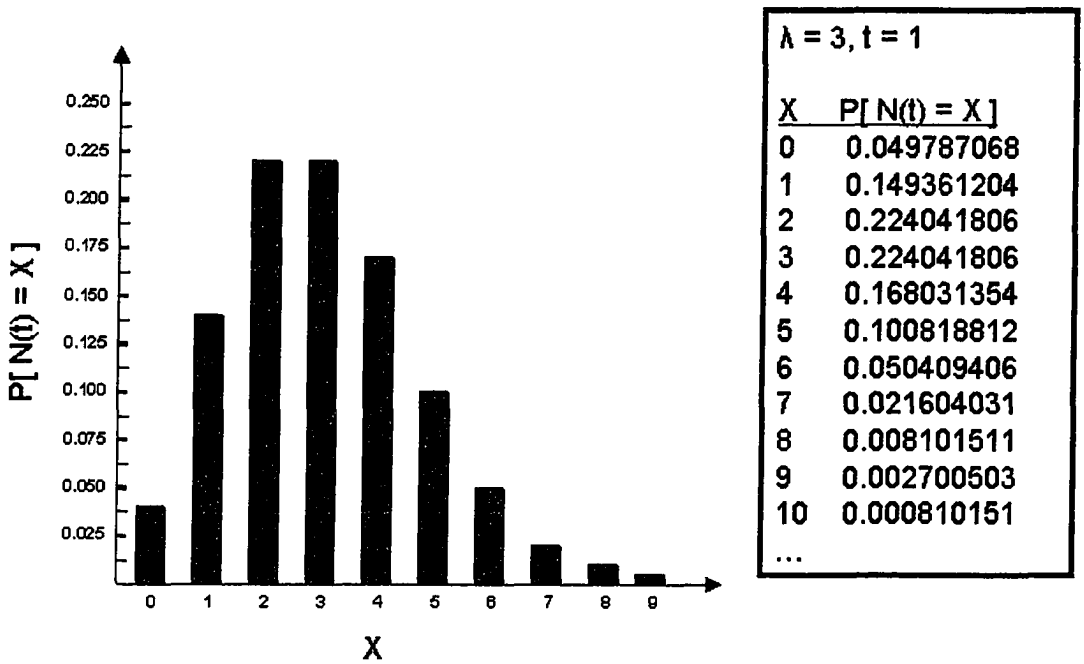

Figure 2.7 Plot of Poisson Probability Mass Function for λ = 3 and t = 1

[12] The mean is a measure of the central tendency of a random variable. The variance is a measure of the spread or variation of the possible values of the random variable around the mean.

Generating a Poisson process is fairly straightforward, and several algorithms are suggested by Ross [Ros02] for this purpose. One of the simpler algorithms proposed by Ross is as follows:

Poisson Process (rate λ, time interval T, list S)
Step 1: $t = 0, I = 0$.
Step 2: Generate a random number $0 < U < 1$.
Step 3: $t = t - 1/\lambda \log U$. If $t > T$ stop.
Step 4: $I = I + 1, S(I) = t$.
Step 5: Go to Step 2.

This algorithm accepts the rate λ, length of the time interval T and empty list S as input. Note that the log of a real number between 0 and 1 will be negative causing the value of t to be continually increased. The results of the algorithm are returned via the list S. S contains I entries, one for each of the I event times that are generated within the time interval of length T.

An algorithm similar to this one can be used to generate the simulated arrival and departure times of peers to and from a peer-to-peer network.

2.4.3.3 Query Events

Simulating query behaviour requires decisions about what the targets of the queries should be, and how often they are to be issued.

In the real world, it has been shown that the subject of queries issued by users of a peer-to-peer system follows a Zipf distribution [Sri03]. In a manner similar to that used in section 2.4.3.1 Content Distribution, a pool of keywords or keys can be

maintained that are ranked according to popularity. Zipf's law can then be used to determine how many times each of these keywords or keys will be issued as a query.

The frequency with which queries are initiated by peers is a random process similar to the rate of arrivals and departures of peers from the system described in section 2.4.3.2 Arrival and Departure. As such, a Poisson process is also a logical choice for simulating the number of queries that will be initiated at each time step in the simulation.

2.4.4 Scalability

Perhaps the most important question about a peer-to-peer simulator implementation is how well does it scale? A simulator that can only represent a few thousand peers cannot expect to realistically measure the performance of a peer-to-peer protocol which when deployed in the real world deals with hundreds of thousands or even millions of peers.

When simulating networks of huge numbers of peers, the primary concern related to scalability is minimizing the memory footprint of each simulated peer and its associated messages. Another important concern is ensuring that the simulation can be run in a reasonable amount of time.

When analyzing these concerns, it is important to remember that a simulator need not capture every last detail of the system from real life. For example, the data

stored at peers can be represented by fixed-sized place holders which contain only necessary information such as the name and the size of its content. Also the links between peers may not need to have any form of explicit representation (ex: store references to other nodes instead of using edge objects).

Decreasing the memory usage of the simulator should lead to faster execution times, but equally important to speed is the efficient implementation of the event processing mechanism (See section 2.4.2 Event Processing). In general it is important to weigh the pros and cons of sacrificing certain details to improve scalability.

2.4.4.1 Parallel, Sequential, Distributed or Multi-Threaded

The implementation of most simulators is sequential. A sequential simulator runs solely on one computer, and all of the events are generated or processed one after another. Some simulators attempt to increase their performance (ex: speed, scalability) by using threads, parallel computation or distributed computation.

Multi-threaded simulators are useful for certain models. Threads enable parts of the simulator to appear to run simultaneously, which would seem to be an ideal model since peers in a real network also run simultaneously. However the use of concurrent threads can have a considerable negative impact on performance due to the overhead required to keep them synchronized.

Distributed or parallel simulators offer a more promising avenue for increasing performance by sharing the work between multiple computers. The limitation of these approaches is that data synchronization must occur between the different computers. Careful thought must be given to the trade-off between processing power and communication overhead.

2.5 Existing Peer-to-peer Simulators

This section surveys some existing peer-to-peer simulators, focusing primarily on simulators that show promise in supporting more than just a single protocol. This section concludes with a brief comparison of the surveyed simulators.

2.5.1 3LS

The 3LS or 3 Layered System peer-to-peer simulator was developed by N. Ting of the University of Saskatchewan [Tin03]. Ting developed a set of criteria for evaluating peer-to-peer simulators. These evaluation criteria are usability (ease of use), extensibility (supporting different protocols), configurability (setting configuration parameters), interoperability (with other applications), level of detail (of simulated environment) and build-ability (simulated code can be deployed as working application).

Ting uses these criteria to rate the relative performance of some existing peer-to-peer simulators, most of which are designed solely to test the performance of a single protocol (with the exception of NeuroGrid). Ting concludes that 'current P2P simulators do not support the customization of the initial network state and are limited in the level of detail and the scalability of the supported models'. It is interesting to note that scalability is mentioned in the conclusion, but not in the evaluation criteria.

Ting's conclusions serve to motivate the development of a generic peer-to-peer network simulator 3LS, with the goal of fulfilling each of the criteria. 3LS' name stems from the idea of splitting the simulator into 3 separate levels; the network level is at the bottom, the protocol level is in the middle and the user level is on top. Communication can only occur between adjacent levels.

The network level contains a two-dimensional matrix (i.e. adjacency matrix) which stores the distance values between the nodes in the simulation. Worker threads are associated with each of the nodes to perform communication with other nodes in the network. Four queues are used at each node to store pending message objects. These queues are the Outbox, the Inbox-For-Network-Delay, the Inbox-For-Processor-Delay and Inbox. The use of these queues is intended to allow the simulation of delays related to network traffic and CPU delays. A 'static step clock' is used to propagate messages between the various queues, and a node receives the message when it reaches its Inbox.

The protocol level contains instances of a peer class. Any protocol implemented by the simulator must extend this class. The Gnutella peer class is used as an example, and apparently this object is used for maintaining the list of a peer's neighbouring peers. Each instance of the protocol class is also linked to an application class which is responsible for passing message objects to other peers.

The user level was not actually implemented, although the idea seems to be to provide a means for scheduling events related to user specific behaviour.

Ting concludes this paper with the claim that all of the evaluation criteria have been fulfilled. In addition, it is reported that 3LS allows the simulation of 'computer hardware', 'network overlay' and 'network delays'. The scalability of the implementation is not discussed directly, although Ting reports that 'the simulation with 255 nodes (50176 events) cannot be performed ... due to memory limitations...'

2.5.2 Anthill

Anthill is a 'framework for the development of agent-based peer-to-peer systems' developed by members of the computer science department of the University of Bologna [Mon01] [BMM02]. Anthill attempts to use certain aspects of biological and social systems to develop a framework for the development and analysis of peer-to-peer applications. As the name implies, Anthill models a peer-to-peer network after a

collection of interconnected nests of ants. A nest corresponds to a peer in the network, and the ants are the messages that travel between them.

Each nest provides facilities such as storage management, communication and topology management, ant scheduling and an interface between the nest and the application. When the application makes a request of the nest, the nest deploys ants into the network to perform the requested service. In the context of a file-sharing application, this request could be a query for documents that match a given key word.

Ants travel between the different nests in the network, attempting to fulfill the service requested of them. Each ant maintains certain information about its state, which may include query requests, matching documents or other data. The algorithm run by the ant may also be included with this state information, allowing for dynamic peer-to-peer algorithms (which evolve over time) to be supported. Upon arrival at a nest, ants use their state information to interact with the facilities made available to it by that nest. An ant 'carrying' a query can request that the nest return all matching documents from its storage facility. If a match is found, the ant will 'pick-up' the matching document(s) and return to the nest where it was born. Ants that do not fulfill their requests within a certain time (TTL) can be killed by a foreign nest.

Anthill is designed to be easily deployable either for simulation or real world use. This is accomplished through the use of the JXTA (Java) project promoted by Sun Microsystems. In the simulation setting, Anthill allows many simulator

parameters, including which protocol to run to be loaded from XML configuration files.

Despite its many promising features, development on Anthill has recently been stopped in favour of PeerSim described in section 2.5.7 PeerSim (A. Montresor, personal communication).

2.5.3 Narses

The Narses network simulator was developed by Giuli et al. [GB02] of Stanford University. Their network simulator uses a 'flow-based' approach for modeling the communication channels between nodes. According to Giuli et al. existing peer-to-peer simulators "do not model network behaviour such as propagation delay, traffic interdependencies and congestion". Also, Giuli et al. state that with real network simulators such as ns [NS] "simulating applications like Gnutella or higher-layer protocols is very expensive". Narses is meant to form a middle ground between the communication complexity of real packet level simulators such as ns and the simplicity of existing peer-to-peer simulators.

Among the simplifying assumptions made by Narses is that communication occurs as part of a flow as opposed to the piece-by-piece packet level transmission used on the internet. Narses also assumes that there are no bottleneck links between any two nodes beyond their immediate neighbourhood. The authors state that "this

simplification implies that not all topologies are appropriate for simulation by Narses".

Narses is implemented in Java. Narses claims to offer portability of its source code to a real world application, based on its use of a transport layer with an interface similar to UNIX sockets. Narses has the ability to run simulations with different levels of detail in the network model. Giuli et al. explain that this can be used for either debugging a small model with high detail or simulating a large model with lower detail. During experimental comparison between Narses at the most detailed level and ns, it was shown that memory usage of Narses was 28% of ns', and Narses ran the simulation 45 times faster.

In Narses, bandwidth is modeled by assigning a bandwidth to each node. At the highest level of detail, Narses allocates the bandwidth of each flow using a technique the authors call 'minimum-share allocation'. Under this technique, the bandwidth is split equally for each flow to or from a node. The bandwidth available to a flow is the minimum of the two bandwidths given to it by the sending and receiving node. At the simplest level of detail, Narses does not take into account the cross-traffic across links.

The scalability of Narses at the highest level of detail is a little doubtful due to the use of an "optimal minimum spanning tree routed at each node" [GB02] used for routing of flows, requiring $O(n)$ storage per node in an n node network. Indeed, Giuli

et al. only report experiments on Narses using a 600 node network generated by the GT-ITM topology generator.

Narses was designed as a general purpose network simulator, and is not restricted to the simulation of peer-to-peer networks. Giuli et al. report on using Narses to study several large distributed applications, although studies of peer-to-peer applications such as Gnutella are the focus of future work.

2.5.4 NeuroGrid Simulator

The NeuroGrid Simulator [Neu] was developed by Joseph, a research associate at the University of Tokyo. The NeuroGrid simulator was originally developed as a simulator for comparing the performance of the Freenet, Gnutella and NeuroGrid protocols. More recently, the simulator has been extended to support DHT protocols such as Pastry. Indeed, the NeuroGrid simulator was designed so as to be as extensible as possible with regards to supporting new protocols. New protocols can be added to the NeuroGrid simulator by extending several abstract classes. These abstract classes include the Node, MessageHandler, Message, Keyword and Document classes [Jos03].

The abstract Node class provides rudimentary functionality such as a list of connected Nodes, a Queue for receiving Messages, a reference to the MessageHandler to use when receiving or sending Messages, a list of seen Messages, a table mapping

(possibly multiple) Keywords to Documents and a knowledge base specific to the NeuroGrid protocol. The other abstract classes allow for the representation of the different types of messages, key/keyword and document types used by other protocols.

The NeuroGrid simulator makes use of an Action Event (i.e. event scheduling) framework to simulate the flow of messages through the represented network. Under this framework, all pending actions (messages) are stored in a global table with their associated time stamp. The simulator loops repeatedly through this table, executing actions that match the current time stamp, and scheduling new actions for the future as necessary. Communication or other failures can be simulated by increasing the time stamp of a pending action.

The NeuroGrid simulator appears to support the use of a ZipF function for distributing data, although its use is only documented for assigning keywords to documents [Jos01a] [Jos02].

One of the simplifying assumptions made by the NeuroGrid simulator is that all links between nodes have equal bandwidth. In fact, no bandwidth data is associated with the list of connected nodes stored at each node. This simplifying assumption is likely one of the principal factors allowing the NeuroGrid simulator to run simulations of up to 100,000 nodes [S. Joseph, personal communication].

2.5.5 P2Psim

P2PSim is a peer-to-peer simulator developed by Gil et al. of the Computer Science and AI Laboratory at the Massachusetts Institute of Technology [GKL+03]. The primary goals of P2PSim are "to make understanding peer-to-peer protocol source code easy, to make comparing different protocols convenient, and to have reasonable performance" [GKL+03].

P2PSim is one of the few peer-to-peer simulators to make use of threads in simulation. Gil et al. state that the use of threads makes "the implementations [of peer-to-peer protocols] look like algorithm pseudo-code, which makes them easy to comprehend" [GKL+03]. P2PSim uses an event queue to store pending events sorted by time stamp. The main program thread repeatedly removes the first event from this queue and executes it within a new thread. The main thread is then blocked until all new threads have finished executing.

Several DHT peer-to-peer protocols have been implemented for P2PSim such as Chord, Tapestry and Kademlia. The authors claim that implementing these protocols was fairly straightforward, although some careful coding was required to handle the concurrent threads. It should be noted that the protocol implementations in P2PSim are based on the concepts of mapping node identifiers to internet protocol addresses, and no actual applications such as file sharing have been implemented [T. Gil, personal communication].

The main components of P2PSim are the Node, Network, Topology, Lookup Generator and Churn Generator objects. Nodes represent individual peers which belong to the Network. The Topology object determines the latency of communication between any two peers in the Network. The Churn Generator models the dynamic aspects of the Network, such as the arrival and departure of Nodes. The Lookup generator models the behaviour of the Nodes by issuing searches for Nodes which are closest to a given key.

One of the simplifying assumptions made by P2PSim is that the network delay model consists only of latency[13], and not bandwidth, queuing delays or packet loss. The latency model used by P2PSim is based on empirical data collected by internet measurement studies. When a Node wishes to send data to another Node, it sends a packet of data consisting of an RPC (Remote Procedure Call) to the Network object. When the Network object receives such a packet, it uses the Topology object to determine the latency between the sender and receiver. This latency determines the time stamp of the delivery event which gets placed in the event queue for this message. At a later time, the delivery event will be removed from the event queue by the main thread, and the corresponding RPC will be executed.

It remains to be seen whether or not the choice to use Multi-threading in P2PSim will impose any significant impediments to performance and scalability. As

[13] Latency is the time between one peer sending a message and another peer receiving it.

of this writing, P2PSim has not been fully tested, although the authors believe that it should scale well to approximately 10,000 peers [T. Gil, personal communication].

2.5.6 Packet Level Simulator

The Packet Level Simulator (PLS) was developed by He et al. of the Networking & Telecommunications Group at the Georgia Institute of Technology [HAR+03]. The primary goal of PLS is to rectify the lack of packet level details in existing peer-to-peer simulators. PLS aims to achieve this goal while maintaining extensibility and scalability.

PLS includes packet level simulation details by running on top of an existing network simulator such as ns [NS]. To allow for a variety of existing network simulators to be used as a base, PLS uses a three layered architecture.

The bottom layer of PLS is the Socket Adaptation Layer. The Socket Adaptation Layer is responsible for adapting the interface of a specific network simulator to a generic one that is usable by the upper layers. This layer provides adaptations for a variety of network simulators, as well as an adaptation which allows packet level details to be bypassed altogether.

The middle layer of PLS is the PeerAgent layer. The PeerAgent is responsible for accepting messages passed to it by the Socket Adaptation Layer and parsing them

in a protocol specific manner. The parsed messaged may be rejected, sent back to the Socket Adaptation Layer for forwarding to another peer, or passed to the top layer.

The top layer of PLS is the PeerApp. The PeerApp layer serves as the interface to the peer-to-peer application users, embodying behaviour related to individual peers. This layer provides functionality for generating user events such as arrival or departure from the network, or queries for content. It also contains the repository of content available at each peer, and is capable of adding to the content after a successful query.

PLS allows for a wide variety of parameters of the simulation run to be set via configuration files. Some of the parameters of PLS allow for content to be distributed with a Pareto distribution, and for peer arrivals to occur based on a Poisson process. A variety of experiments were performed with PLS using the Gnutella protocol over a topology generated with the transit stub model. These experiments showed that low level packet details do have a significant effect on the results of the simulation.

To overcome the barrier to simulator scalability introduced by packet level details, He et al. developed a variation of the ns simulator called pdns [HAR+03] that allows the simulation to be run in parallel over several computers. Using PLS on top of pdns, He et al. were able to run a simulation with 8192 networks nodes containing 1440 peers. They expect to be able to use this setup to run simulation with 100,000 or more network nodes, and tens of thousands of peers.

2.5.7 PeerSim

PeerSim is a peer-to-peer simulator developed by Jelasity et al. [JMB03] as part of the BISON project [Bis]. PeerSim is the successor of the Anthill framework (Section 2.5.2 Anthill) and is being developed by the same authors. The stated goals of PeerSim are extreme scalability, support for dynamicity and modularity of simulator components.

Jelasity et al. propose several simplifying assumptions to support extreme scalability in PeerSim. Low-level or packet-level details of the underlying network are not simulated, and neither is bandwidth or latency. According to the authors, "the simulation model that is adopted by PeerSim ignores concurrency and in fact it is very similar to a cellular automaton model" [JMB03].

The main component of the PeerSim architecture is the configuration manager. The configuration manager is responsible for loading configuration files or command line parameters that decide what form the simulation will take. The configuration manager is the only fixed component in the simulator; the configuration files allow the user to specify which other components should be loaded. Using this framework, PeerSim allows developers to create their own interchangeable components to be used during a simulation.

The network topology used by PeerSim is made up of Node objects, each of which is host to one or more Protocol objects. Communication between nodes is

accomplished via method invocations between Protocol object. Protocols must implement a Linkable interface which allows the simulation environment to obtain information about the neighbourhoods of the nodes. Dynamicity is achieved in PeerSim through the use of a Dynamics object. The Dynamics object may be executed periodically to perform changes to the collection of nodes or the neighbourhoods of individual nodes.

PeerSim uses Observer objects, which like Dynamics objects may be executed periodically. Observer objects are responsible for harvesting statistics from the simulation, and may be customized for individual protocols.

The cornerstone of PeerSim which drives the simulation is the Simulation Engine object. The simulation engine uses an Activity Scanning approach which at each time step scans all of the nodes (in random order) to invoke call-back routines on their associated protocol objects.

According to the authors, PeerSim is capable of performing simulations on networks of up to 1,000,000 nodes [A. Montresor, personal communication]. It should be noted however that as of yet, no actual peer-to-peer protocols have been simulated with PeerSim.

2.5.8 Query Cycle Simulator (P2PSim)

The Query Cycle Simulator (sometimes called P2PSim) is a peer-to-peer simulator developed by Schlosser et al. of Stanford University. The primary focus of the Query Cycle Simulator is to accurately model user behaviour in a peer-to-peer file sharing network [SCK03].

The Query Cycle Simulator uses a model which proceeds in query cycles. In each query cycle, a set of nodes issues queries over the network. As the queries get propagated throughout the network, other nodes may return a match or forward the query to other nodes. According to Schlosser et al., a query cycle ends when all querying peers 'download a satisfactory response'. This statement implies that the simulator is not designed to handle erroneous or unsuccessful queries.

Schlosser at al. divide the discussion about modeling user behaviour in QCC into two categories: Content Distribution and Peer Behaviour. Content Distribution deals with modeling the volume and content of data that each peer carries. Peer Behaviour deals with modeling what queries peers will issue, which query matches they will select for download, and peer uptime and session duration.

QCC uses empirical data obtained by Saroiu et al. [SGG02] to determine the number of files to assign to each peer in the network. The determination of the content of files that each peer stores is motivated by the observation that in the real world, peers are generally more interested in a subset of the total available content. To mimic

this behaviour, QCC organizes files into content categories. Content categories and files within each category are ranked by popularity as governed by a ZipF distribution. Each peer uses this popularity ranking, together with its randomly assigned interest level in each category to determine how many categories, and files within them it stores.

QCC also uses the empirical data from [SGG02] to determine how long peers stay connected to the network. Using this data, each peer is assigned a probability for being online. At each cycle, this probability is used to determine if a new peer will join (or leave) the network. When a peer joins the network, it will preferentially select nodes with higher degree to connect to.

Since no empirical data seems to exist on query rates, QCC uses a Poisson Process to determine of how often peers will issue queries. The target of each query is determined probabilistically by the issuing peer's interest level in each content category, and the popularity ranking of files within the categories. When a peer receives multiple matches in response to a query, one is selected at random for the source of the ensuing download. Downloads are the only activity in QCC which consume bandwidth. In QCC, each peer is assigned a bandwidth at start up based on data from [SGG02].

While QCC's performance is exemplary in terms of modeling of peer behaviour, it suffers from limited scalability. According to one of the authors QCC

does not scale well above 1000 peers, although this is apparently due to the fact that QCC "models actual files" [T. Condie, personal communication].

2.5.9 SimP²

SimP^2 is a peer-to-peer simulator designed by Kant et al. [KI] of Intel Labs. According to the authors, SimP^2 is designed to "address detailed performance characteristics such as queuing delays and message expiry or loss". SimP^2 is made up of two parts, one for generating a set of peer-to-peer network instances, and another which consists of 'parallel' or simultaneous simulation of Gnutella-like protocols over these instances.

The peer-to-peer network instances are generated in the first part using a non-uniform random-graph based model proposed by the authors. Kant et al. propose a 3 tiered model for the nodes in the network instances. Tier 1 consists of 'distinguished nodes' which are connected almost indefinitely, and have high bandwidth and lots of content. Tier 2 consists of 'undistinguished nodes' which are semi permanent nodes with less content and bandwidth than distinguished nodes. Tier 3 consists of highly transient nodes which do not contribute much to the community and are non permanent. Non-uniform network instances are constructed by SimP^2 by connecting each new node to a random number of nodes. Each link to the new node is created with a distinguished node with probability q and to an undistinguished node with

probability 1-q. For simplicity, Kant et al. ignore transient nodes in their graph model, and in fact their network instances are static after creation.

According to Kant et al., the second part of SimP^2 is not limited to working with graphs generated by the first part, but may also accept graphs from an external source. Once provided with a set of graphs or network instances, the second part of SimP^2 randomly selects a graph on which to simulate each successive transaction. The idea is that using a set of parallel network instances yields simulation results that are a better representative of the average performance of the network than a single instance.

SimP^2 models the request or query generation at each peer by using an on-off process. When a peer's request generation is 'on', it generates requests at a constant rate. The lengths of these on-off periods are determined by a Pareto Distribution. SimP^2 also uses a Pareto distribution along with a uniform distribution to determine the distribution of file sizes to the peers.

Unfortunately, SimP^2's requirement of parallel network instances imposes a serious barrier to scalability. This is reflected in the fact that SimP^2 has not been run for simulation of more than 500 peers [R. Iyer, personal communication].

2.5.10 Other Simulators

As noted by Montresor et al. [MCH03], most of the authors of peer-to-peer papers relying on simulation tend to make use of "home-made" simulators. Due to the great

volume of such papers, it is impossible to provide an exhaustive listing and summary of these home-made simulators. In any case, none of these home-made simulators appear to have been designed with extensibility in mind, and as such are limited to the support of a single peer-to-peer protocol at best.

2.5.11 Comparisons

Table 2.3 compares the functionality of the more interesting simulators from the previous sections. Each simulator was ranked according to three criteria: scalability, extensibility and traffic modeling. Scalability was measured by the simulator's ability to simulate large networks of a hundred thousand peers or more. Extensibility was measured as the ease with which the simulator could be configured to run arbitrary peer-to-peer protocols. Traffic modeling was measured as the simulator's capabilities in terms of modeling network traffic, and its impact on the protocol in use.

Data for each column was obtained from available documentation and personal communication with the authors. See Appendix A for the full questionnaire that was sent to the authors.

Table 2.3 Comparison of features of existing peer-to-peer simulators

Simulator	Scalability	Extensibility	Traffic Modeling
3LS	Low	Medium-High	NA
Anthill	NA	Medium-High	NA
Narses	Medium	Medium	High
NeuroGrid	Medium	High	None
P2PSim	Low	NA	Low
Packet Level	Medium-Low	Medium-High	High
PeerSim	Medium	Medium-High	None
Query Cycle	Low	NA	NA
SimP^2	Low	NA	Medium-Low

2.6 Summary

This chapter began in Section 2.1 with a description of several of the current peer-to-peer protocols, along with a discussion of how they can be classified. Section 2.2 provides discussion of the network topologies formed by these protocols, including how they may be represented, generated and evolved for simulation purposes. Section 2.3 describes how the network topology can be used to simulate communication over the simulated network. Section 2.4 provides a summary of the simulation literature, along with some background on how events can be generated and processed efficiently. Finally, this chapter concludes with a survey and comparison of existing peer-to-peer simulators

Chapter 3

Simulator Core

This chapter describes the core architecture of the peer-to-peer simulator developed for this thesis. The core architecture is composed of several key components:

- A set of peers participating in the simulated network.
- A set of connections conveying messages between peers.
- A repository of content to be distributed to peers and events.
- Ordered collections of peer action and inter-peer message events.
- A statistics reporter which collects and reports information about events.

Figure 3.1 provides a high level UML class diagram of the relationships between the key components. Detailed descriptions of the components are provided in sections 3.1 to 3.5, along with detailed UML class diagrams where appropriate.

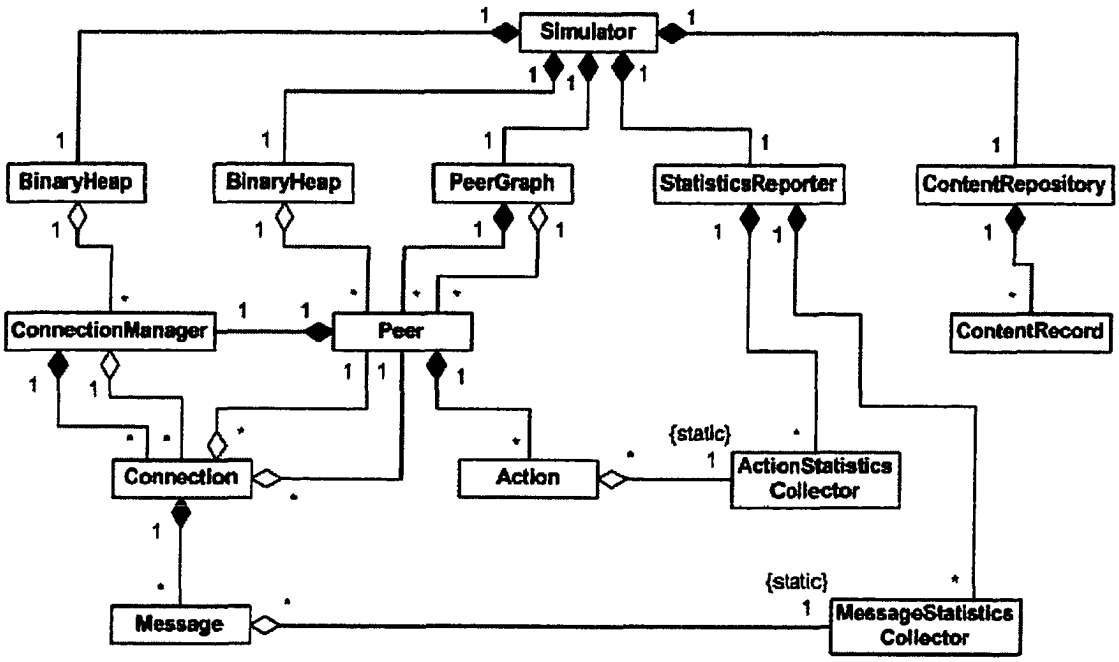

Figure 3.1 Class diagram of the principal components of the simulator core

3.1 Peers

The principal component of the simulator is the list of peers taking part in the simulation. Individual peers in the simulator are represented by peer objects.

The simulator stores all of the individual peer objects in a single peer graph object. When the simulator is initialized, the maximum number of peer objects to be stored in the peer graph must be specified. The peer graph maintains an array of the peer objects, assigning each peer a unique index, which can be used for retrieval purposes.

The simulator allows for peers to be active or inactive. An active peer corresponds to a peer that is currently connected to and participating in the simulated network. This is an important feature for the study of the effects of peer churn on simulated peer-to-peer protocols. To support this feature, the peer graph maintains a list of active peers, and assigns each peer a unique 'active index', allowing for efficient retrieval of all currently active (or inactive) peers in the simulator. Figure 3.2 illustrates the relationship between the peer graph and the peers it stores.

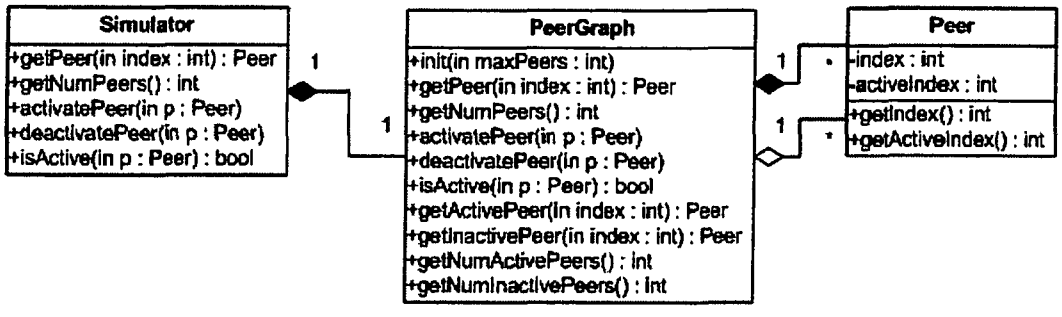

Figure 3.2 Class diagram of peer related components

To control interaction with other peers, each peer object maintains a reference to an instance of the abstract protocol class, which is subclassed to provide details specific to the protocol in use. This is described in more detail in Section 4.2.

3.2 Connections

The simulator is comprised of two concurrent topological structures. The protocol-level topology contains information about which peers a given peer is aware of or

linked to, whereas the communication-level topology maintains information about which peers a given peer is currently transmitting messages to or receiving messages from. This section describes the communication-level topology of the simulator. Discussion of the protocol-level topology is deferred to Section 4.2.2.

In order to track the communication between peers, the simulator maintains a set of connection objects which represent active connections used for the transmission of messages between pairs of peers. Connections between two peers are created whenever the first message is transmitted between them, or the last message is received. This allows the simulator to model the network topology at the flow or overlay level (Section 2.3.1) as opposed to a detailed packet level. Modeling communication to the packet level of detail, where messages are split into individual packets and routed through various intermediate nodes between the source and destination peers is not conducive to scalability. In real-world networks, the intermediate nodes used to route messages between peers are likely not even part of the peer-to-peer network, necessitating simplifications if a simulator is to keep its storage requirements bounded by the number of peers and connections being simulated.

Each peer keeps track of its active connections through the use of two lists, one for incoming connections, and one for outgoing connections. Each connection object stores references to a source and destination peer, and a list of message objects

(Section 3.4.2) travelling from the source peer to the destination peer. The management of these lists of connections is delegated to the connection manager subclass (Section 3.4.4) associated with each peer object, which is specific to the bandwidth model currently in use. Figure 3.3 illustrates these relationships.

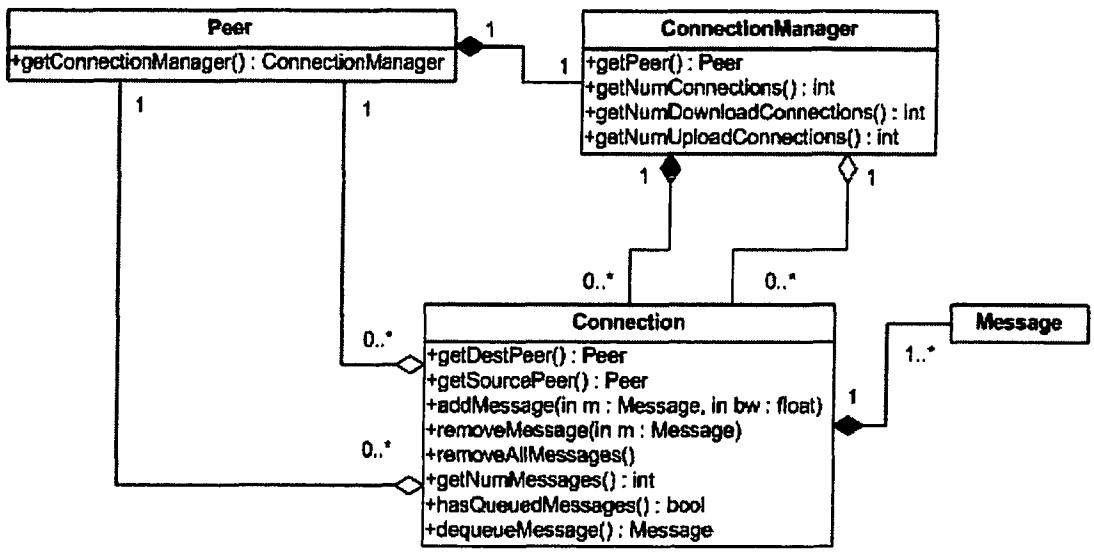

Figure 3.3 Class diagram of connection related components

The interface of the connection class allows for additional messages to be stored in a queue. Messages stored in this queue are not in transmission, and are deemed as 'pending'. The manner in which events are stored in this queue is left up to the protocol implementation, via subclasses of the connection class (Section 4.2.2). Subclasses of the connection class may override the default methods for determining whether a connection has queued messages, and retrieving them when necessary.

3.3 Content

The simulator supports the storage and distribution of generic content records, which may be extended to support content used by arbitrary peer-to-peer protocols. The content record interface is used to represent files or other content types that are transmittable over the simulated network. Classes that implement the content record interface must define methods for retrieving a unique identifier for the content, and the size of the content's identifier and data in bytes. The size of the content's name and data are used when determining the time to transmit content-carrying messages over the network, as described in Section 3.4.2.

All of the content records in the simulator are stored in a central content repository. The content repository interface specifies the methods that a concrete content repository implementation must define for retrieving content records. Concrete content repositories are responsible for populating themselves with content records.

Content records may be distributed to individual peers by providing the simulator with a class which implements the content distributor interface. Content distributors perform their task using a reference to the simulator's peer graph and content repository. Distribution of content involves selecting content record objects to assign to selected peer objects. Because different peer-to-peer protocols store content

in different ways, storage of distributed content is delegated to the protocol object. This is described in more detail in Section 4.2.1.

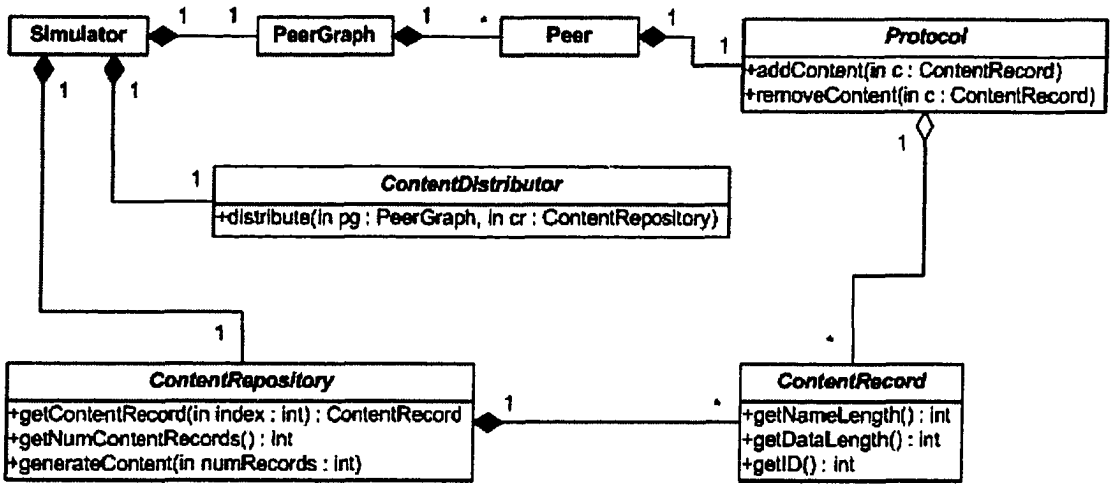

Figure 3.4 Class diagram of content related components

The default content scheme provided by the simulator is a flat content repository, where content records are distributed randomly to the peers. The simulator was designed to support more advanced content repository and distributor implementations, such as those that rank content records using a zipf-like popularity distribution (See section 2.4.3.1). In general, this allows for a wide variety of schemes such as the per-peer interest based scheme employed by the Query-Cycle Simulator (Section 2.5.8).

3.4 Events

The simulator allows for two different types of events: actions, and messages. Actions are used to simulate events local to a given peer, such as those initiated by a user, or other local events initiated at a peer specific to the protocol in use. Messages are used to simulate the communication of all types of messages and associated data between peers in the simulated network.

For the purposes of event processing, the simulator is a discrete-event simulator (Section 2.4.2). The simulator operates in sequential time increments, using a floating point variable to track the current simulation time. Both action and message events store floating point instance variables representing their start and expected simulation times.

3.4.1 Actions

Actions are used by the simulator to represent events or behaviour local to a peer, such as initiating a connection to the simulated network, or commencing a query for content distributed across the simulated network. Each action stores start and expected simulation time fields, additional fields may be added by protocol-specific action subclasses as described in Section 4.2.3.

Whenever an action is added to the simulator, its start time is set to the simulator's current simulation time. The expected time of an action may or may not be

equal to its start time, which allows for actions to be scheduled for the future. Actions are stored in a list maintained by the peer object on which they are to be executed. This allows peers to cancel or reschedule actions when necessary. Figure 3.5 illustrates the relationship between peers and actions.

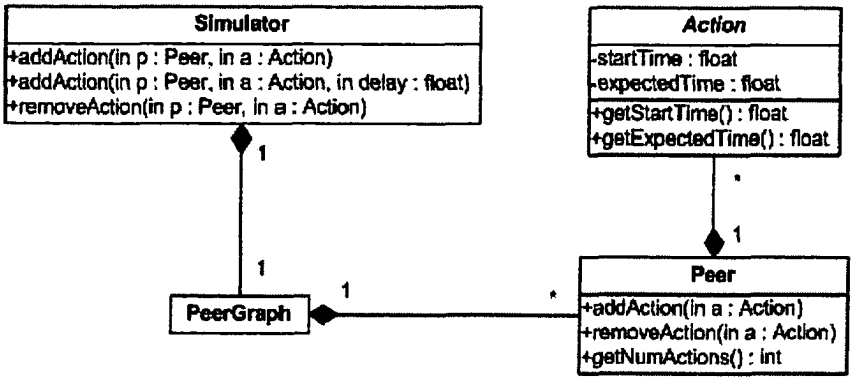

Figure 3.5 Class diagram of action related components

3.4.1.1 Action Generators

Action generators are used to feed the simulator with actions. The simulator maintains lists of two different types of action generators: initial action generators and continuous action generators.

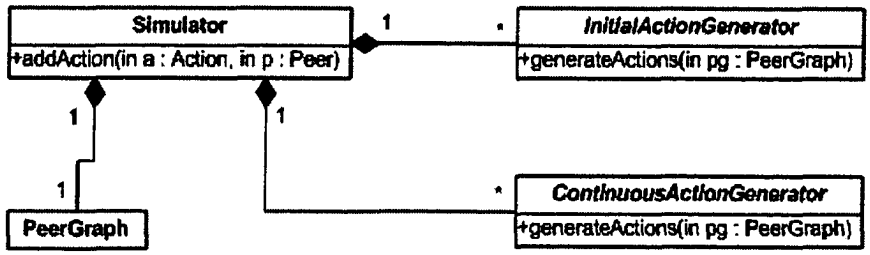

Figure 3.6 Class diagram of action generation related components

Initial action generators are called upon to feed the simulator with actions only once at simulation time 0. The typical use of this form of action generator is to schedule initial actions that will reoccur at regular time intervals. For example, an action generated upon initialization of the simulator may specify that when it is processed, it be re-scheduled for execution at its peer with a predetermined expected simulation time in the future.

Continuous action generators are called upon to feed the simulator with actions at regular time intervals. The action generation time intervals may be specified for each continuous action generator, but this value must be specified as a multiplier with respect to the global action generation time interval stored by the simulator. Another parameter of each continuous action generator is the generation offset. For example, consider an example where the simulator has a global generation interval of 1.5, and a continuous action generator is provided to the simulator with a generation interval of 5 and offset of 3. In this scenario, the continuous action generator will start feeding the simulator with actions at simulation time 4.5 (global generation interval * generation

offset). Thereafter, actions will be fed to the simulator at simulated time intervals 12.0, 19.5, 27.0, etc... (4.5 + global generation interval * generation interval).

For convenience, the simulator provides a method for generating a Poisson distribution using the target rate provided as a parameter. This allows action generators to generate a different number of actions at each interval, yet on average generate them at the target rate.

3.4.2 Messages

Messages are used by the simulator to represent events corresponding to the transmission of data from one peer to another, such as a request to link with another peer to join the simulated network, or the transmission of a query for content stored at another peer. Each message stores start and expected simulation time fields, additional fields may be added by protocol-specific action subclasses as described in Section 4.2.3. In addition, messages are required to implement a method which returns their size in bytes.

Whenever a message is added to the simulator, its start time is set to the simulator's current simulation time. The expected time of a message is determined by the amount of bandwidth allocated to it by the bandwidth model (Section 3.4.4). Messages are stored in a list maintained by each connection object on which they are

being transmitted. This allows for peers to cancel or reschedule messages travelling to or from any other peer when necessary.

In some cases, several messages are added to the simulator simultaneously, corresponding to a broadcast of messages from a single source peer to multiple destination peers. The simulator's message addition interface provides support for this, allowing the bandwidth model to operate more efficiently.

As mentioned in section 3.2, connections support the storage of queued messages. To support this functionality, the simulator provides methods for replacing a message with another that is traveling between the same source and destination peers, which is more efficient than removing and then adding the messages. Whenever a message is to be removed from a connection, the simulator checks if the connection has queued messages, in which case the removed message is replaced with a message from the queue.

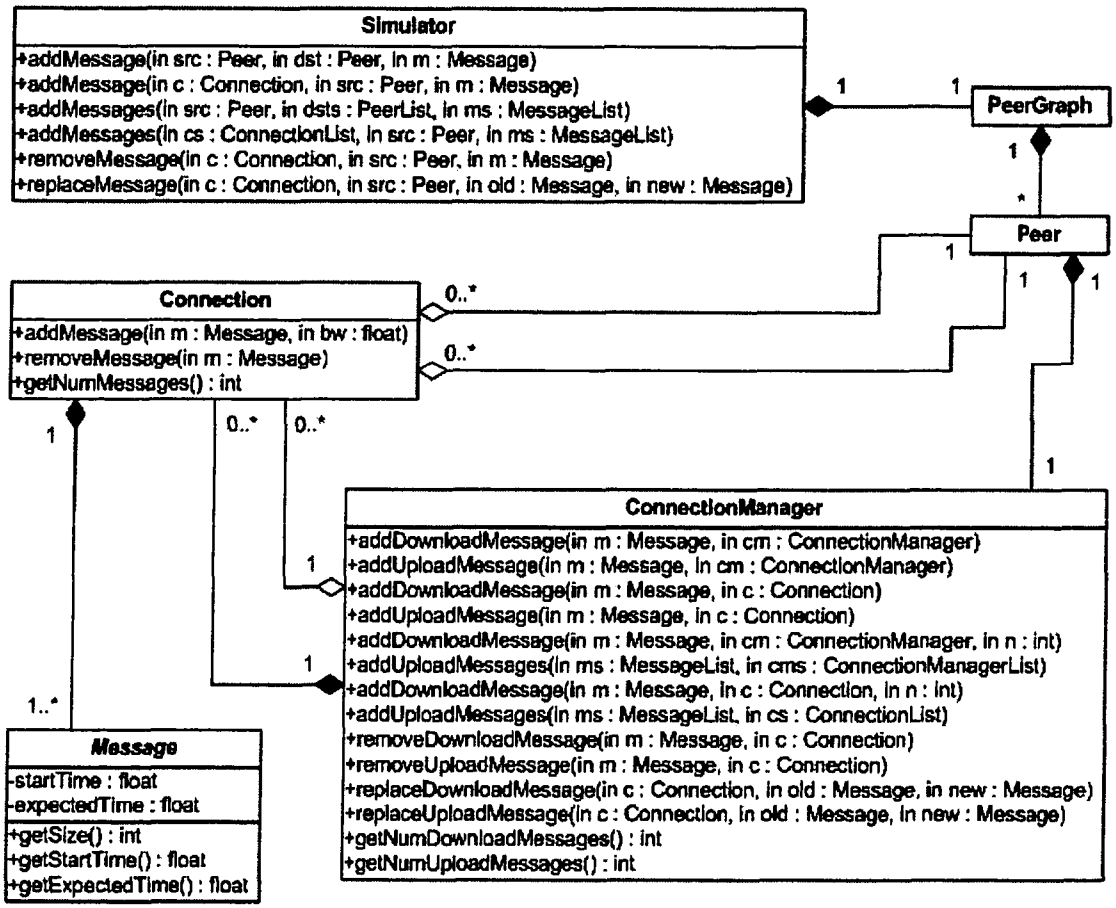

Figure 3.7 Class diagram of message related components

For messages which correspond to the transmission of content (Section 3.3), the size returned should include the size of the content. For example, if a message represents a query for content, its size should include the corresponding content record's identifier size. If a message represents the transmission of content, its size should include the corresponding content record's data size.

3.4.3 Event Processing

A discrete event simulator requires a means for ordering pending events for sequential execution. Section 3.4.3.1 describes how the simulator was designed to efficiently store pending events by increasing order of expected time. Section 3.4.3.2 describes how the simulator uses this order to continuously remove and execute the earliest expected event.

3.4.3.1 Ordering Events

Peers are continuously adding or removing action and message events, which must be processed in increasing order of expected time by the simulator. This requires that the simulator be able to efficiently locate the event(s) that have an expected simulation time closest to the current simulation time.

To this end, each peer (Section 3.1) and associated connection manager (Section 3.4.4) keeps track of the action and incoming message with the minimum or earliest expected time. The minimum action for a peer is easily tracked by storing it at the front of the peer's list of actions. Keeping track of the minimum message is slightly more complicated, due to the fact that the peer's connection manager stores a list of connections, which in turn store lists of messages. In this case, each connection stores the minimum message at front of its list of messages, and the connection manager in turn stores the incoming connection with the earliest minimum message at

the front of its list of incoming connections. This ordering of actions and messages is illustrated in Figure 3.8.

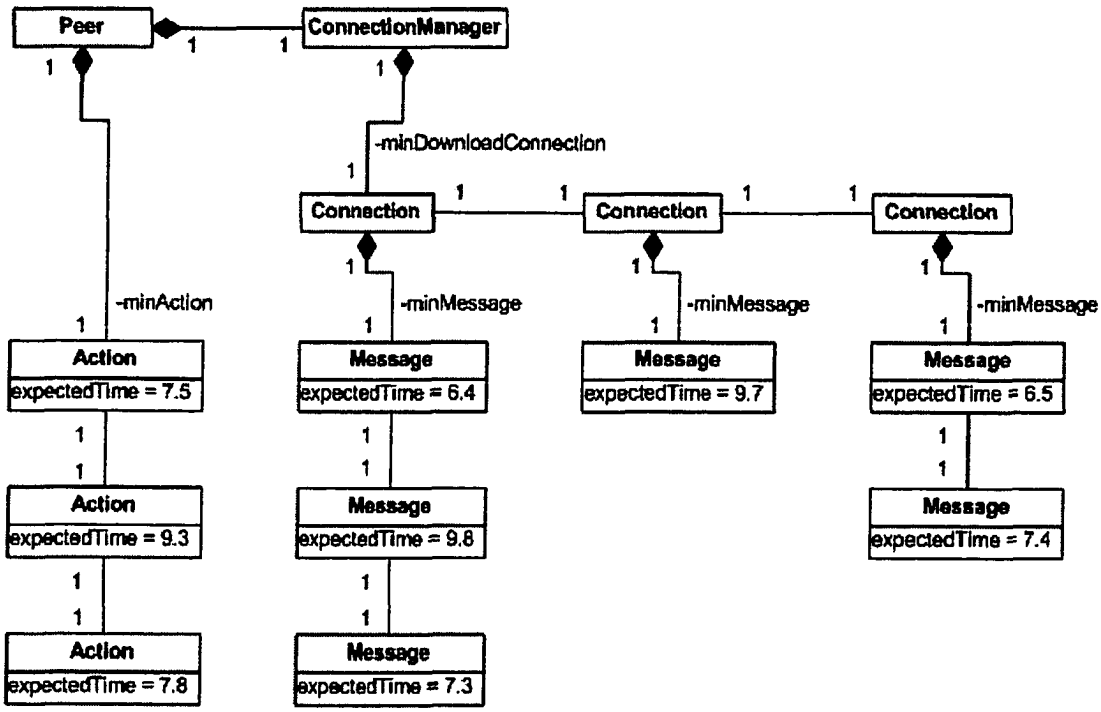

Figure 3.8 Object diagram showing actions and messages ordered for processing

The major advantage of this approach is that the size of the future event list to be processed by the simulator is effectively reduced to a size proportional to the total number of peers in the simulated network.

Continually searching every event in the future event lists to find the earliest action or message events would be highly inefficient. Storing the future event lists in a binary heap data structures allows for the minimum (earliest) event to be accessed in constant time. The cost of the heap structure is that inserting or removing an action or

message may require up to log of the number of peers in the event heap operations, in addition to the time for updating the relevant peer or connection manager's list. This is an acceptable cost when one considers that log of one million elements (peers) results in only a maximum of 19 operations. To support the heap-based future event list, both the peer and connection manager classes implement a heap object interface, using their earliest event's expected time as the heap key. Since the maximum number of elements in the heap is known in advance, it is possible to use an array-based heap implementation, which is highly storage-efficient, requiring only that an integer heap index variable be maintained for each peer and connection manager object. Figure 3.9 illustrates how the simulator's earliest scheduled actions and messages are accessed through the binary heaps which store the future event lists.

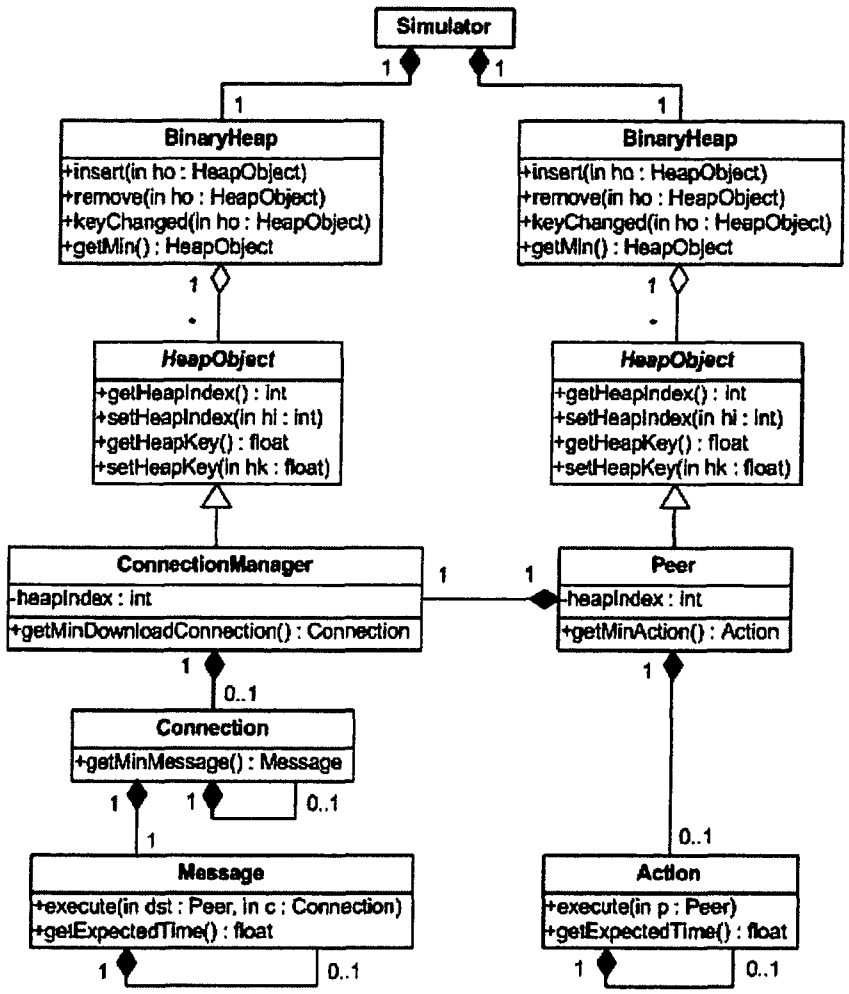

Figure 3.9 Class diagram showing access of actions and messages via binary heaps

3.4.3.2 Executing Events

To process events, the simulator uses the event-scheduling world view described in Section 2.4.2.1. This process requires that the simulator continuously remove and execute the earliest expected event from the binary heaps storing the future action or message event lists. Both actions and messages must implement an execute method

which is called by the simulator when action is performed, or a message is received. Because neither actions nor messages store references to their target peer or connection, this information is passed to them by the simulator upon execution. Section 4.2.3 describes the use of these methods in more detail. Figures 3.10 and 3.11 illustrate the interface for the execute method of actions and messages.

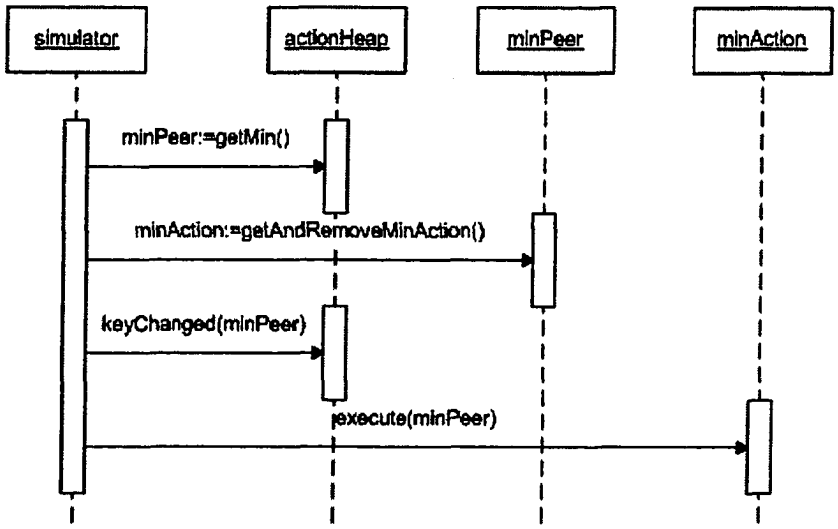

Figure 3.10 Sequence diagram showing how actions are executed

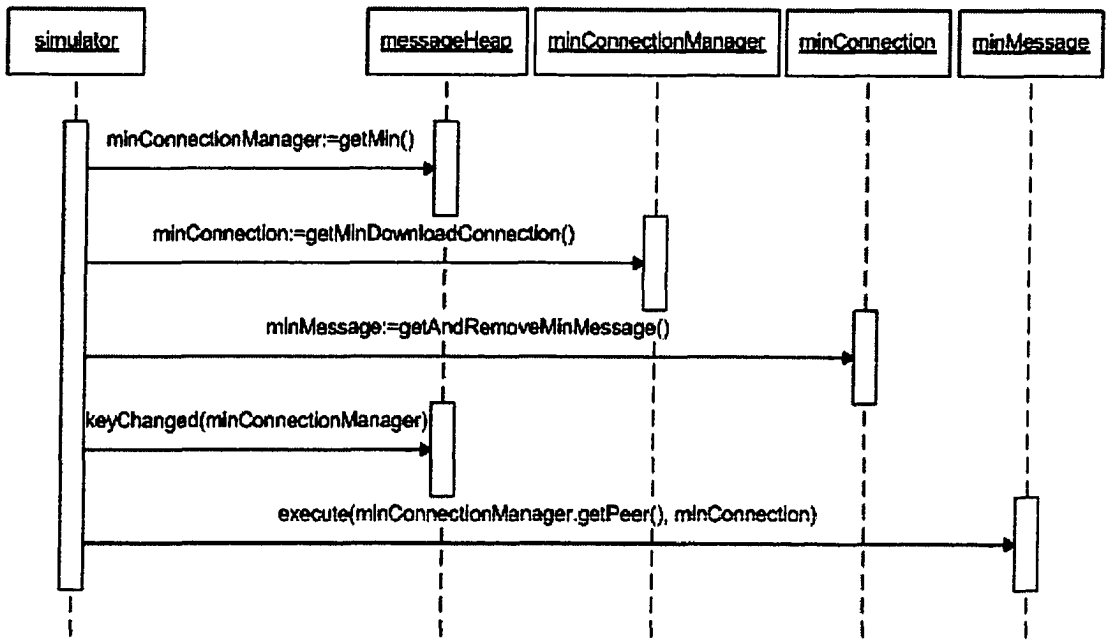

Figure 3.11 Sequence diagram showing how messages are executed

In the case where both the action and message future event lists have an earliest expected event with the same expected time, precedence is given to the action. This precedence is somewhat arbitrary, but is motivated by the fact that the action may cause the peer to become inactive, in which case pending messages for that peer will likely be cancelled.

3.4.4 Bandwidth Models

Upon initialization of the simulator, each peer is assigned maximum upload and download bandwidth values in bytes per simulation time unit. The simulator relies on a class which implements the bandwidth distributor interface for assigning bandwidth

values to peers. This allows for arbitrary bandwidth distributions to be deployed, with total flexibility. For example a bandwidth generator could be configured to deploy 30 percent of peers with dialup bandwidths, 50 percent with broadband bandwidths, and 20 percent with ultra high speed bandwidths.

The peer's bandwidth values are used by the simulator when allocating a portion of bandwidth to individual messages between peers, which in turn affects a message's duration and expected time. The duration of a message is equal to its size divided by its allocated bandwidth. The expected time of a message is equal to its start time plus the duration. If the bandwidth of a message is increased or decreased, the duration of the event relative to the current simulation time will be decreased or increased accordingly.

Figure 3.12 illustrates the interface provided by the simulator for changing the bandwidth allocated to a message.

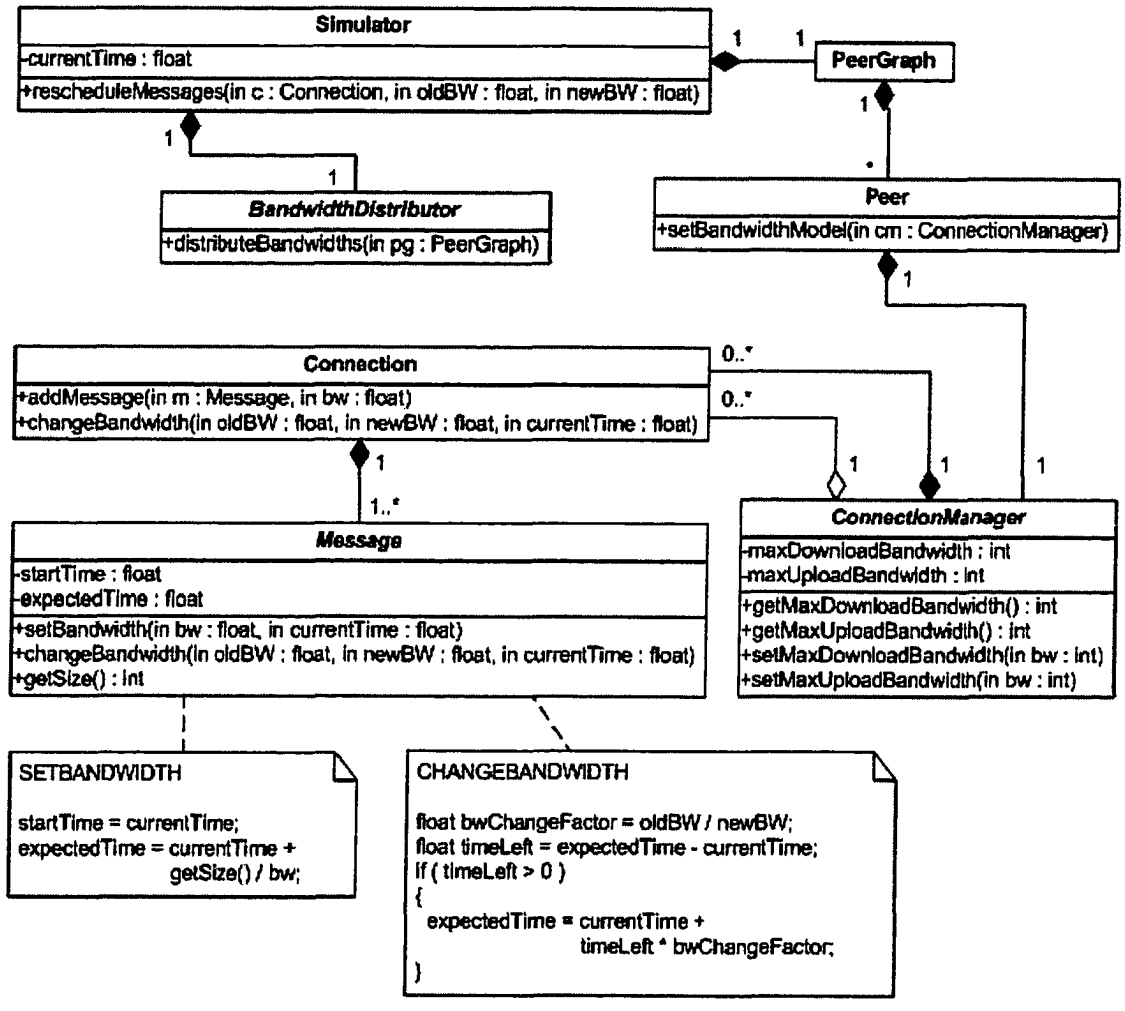

Figure 3.12 Class diagram of bandwidth related components

The details of allocating bandwidth to messages are delegated to the connection manager stored by each peer. Different bandwidth models may be employed by the simulator by subclassing the abstract connection manager class, and distributing instances of the subclass to each peer. At present, the simulator only supports homogeneous bandwidth models for all peers.

3.4.4.1 Minimum Equal Share Bandwidth Model

The default connection manager provided with the simulator is the Minimum Equal Share (MES) bandwidth model, which is effective for modeling bandwidth on overlay networks where the bottleneck bandwidth is defined by the link closest to a peer (See sections 2.3.1 and 2.3.2). With this model, each message between a source and destination peer is assigned a bandwidth which is the minimum value of an equal share of the source peer's upload bandwidth and an equal share of the destination peer's download bandwidth, as illustrated by Figure 3.13.

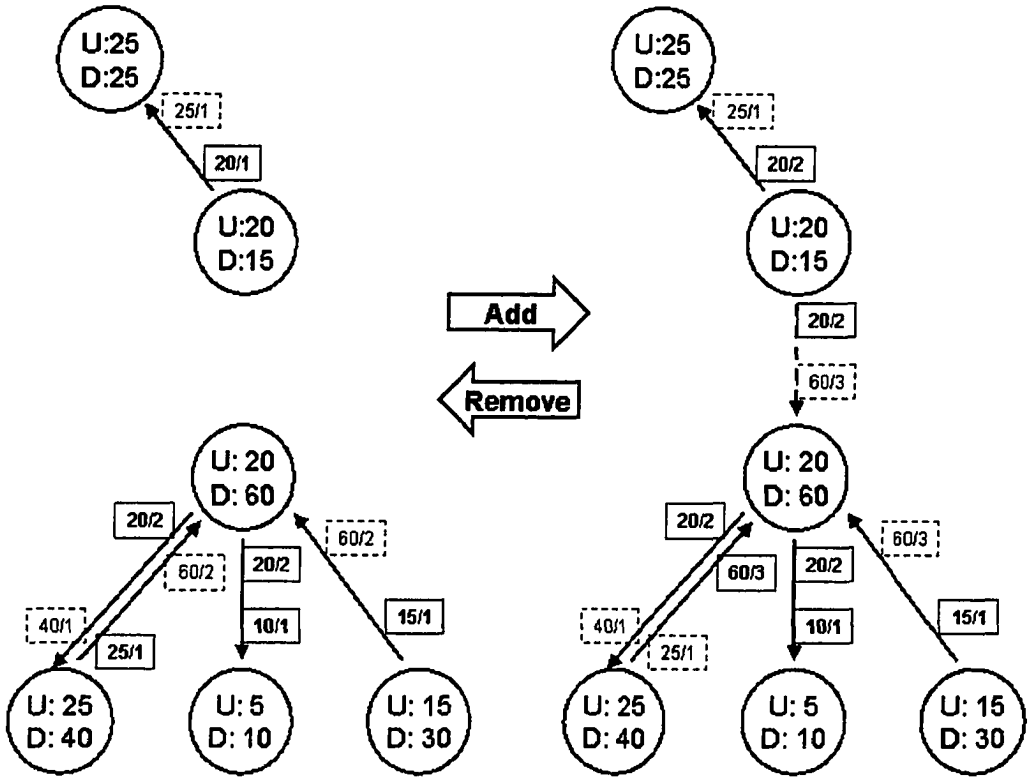

Figure 3.13 Illustration of bandwidth allocation with the Minimum Equal Share model

Figure 3.13 illustrates the minimum equal share bandwidth model in action. Peers are drawn as circles, with associated upload (U) and download (D) bandwidth values. Individual messages between the peers are drawn with directed arrows. At either end of the drawn messages, a text box is used to denote the upload or download bandwidth allocated to it by its source and destination peers. The textbox denoting the minimum bandwidth allocation for a message is drawn with a solid border, while the maximum bandwidth allocation is denoted by a dashed border. The right hand side of the figure shows the changes in message bandwidth allocation when a new message is added (dashed directed arrow). The case for removing a message is symmetric.

The following pseudo code describes the high-level operations that are performed by the MES bandwidth model whenever messages are added or removed. Most of these operations must be performed by any connection manager / bandwidth model implementation. The exception is that other bandwidth models are free to specify which other messages are updated to reflect changes in bandwidth (Figure 3.14: lines 7-10, Figure 3.15: lines 5-8).

Adding a Message between two Peers

```
1.  Find connection between source and destination peer
2.  If no connection exists, create one.
3.  Add message to connection's list (soonest at head)
4.  If is only message for connection
5.    Add Connection to destination peer's download list
6.    Add Connection to source peer's upload list
```

```
7.  If other messages exist for the connection
8.    Update these messages to reflect the decreased share of bandwidth
9.  Update all incoming messages of the destination peer to reflect the decreased
    share of bandwidth.
10. Update all outgoing messages of the source peer to reflect the decreased share
    of bandwidth.
11. Update destination peer's download connection list so that connection with
    earliest message is at the head.
12. If earliest connection for destination peer has changed
13.   If is only message at destination peer
14.     Insert destination peer into message heap
15.   Else
16.     Change key of destination peer in message heap
```

Figure 3.14 Pseudo code for message addition with MES bandwidth model

Removing a Message between two Peers

```
1.  Remove message from connection's list (soonest at head)
2.  If connection no longer has any messages
3.    Remove connection from source peer's upload list
4.    Remove connection from destination's download list
5.  If connection has other messages
6.    Update these messages to reflect the increased share of bandwidth
7.  Update all incoming messages to the destination peer to reflect the increased
    share of bandwidth.
8.  Update all outgoing messages from the source peer to reflect the increased
    share of bandwidth.
9.  Update destination peer's download connection list so that connection with
    earliest message is at the head.
10. If earliest connection for peer has changed
11.   If peer has no messages
12.     Remove peer from message heap
13.   Else
14.     Change key of peer in message heap
```

Figure 3.15 Pseudo code for message removal with MES bandwidth model

The principle advantage of the MES bandwidth model is that it models the slow-down in message transmission times as traffic increases, while minimizing per message storage and processing for the additional or removal of messages. The storage required for each message is kept to a minimum by implicitly determining the bandwidth for each connection via its containing connection manager. The processing for adding or removing a message is also minimized, as discussed later in this section. Another advantage of the MES bandwidth model is that any two messages with the same size that are transmitted at the same time between the same two peers will receive an identical bandwidth allocation, and thus have the same expected time. To take advantage of this fact, a multiple message class was created specifically for same-size messages transmitted at the same time with the MES model. This optimization[14] saves memory by allowing multi-messages to share the overhead cost of the message class and its associated fields.

Whenever a message is added or removed from a connection between a source peer S and a destination peer D, all of the download messages DM(D) of D, and all of the upload messages UM(S) of S are rescheduled. The event ordering model used by the simulator requires that the earliest download connection of each peer must be transporting the earliest download message of that peer, and the rescheduling of upload messages from S may have broken this requirement. The cost of reinstating

[14] This optimization is an application of the Flyweight pattern [GHJ+95]

proper event ordering for all of the peers P in the set UP(S) that are downloading from S is that they must search through their set of DC(P) download connections to find the one storing the earliest download message. The peers and connections that must be processed whenever a message is added or removed from a connection are illustrated by Figure 3.16.

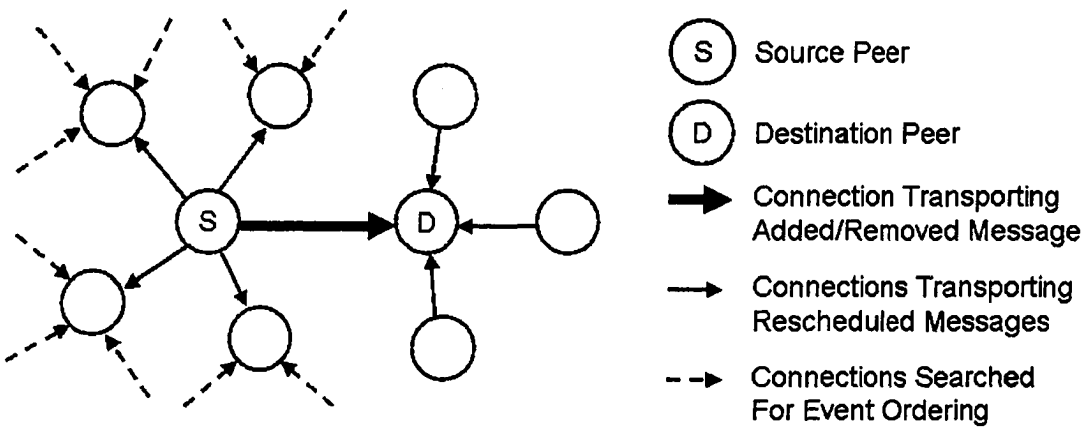

Figure 3.16 Illustration of peers and connections affected by message addition or removal

This results in the following running time for adding or removing a message with the MES Bandwidth Model:

$$O\left(|UM(S)|+|DM(D)|+\sum_{P\in UP(S)}|DC(P)|\right)$$

The amount of processing required for the MES bandwidth model is optimized for situations where a source peer S wishes to broadcast a set of messages to several destination peers DST at once. In this scenario, it would be inefficient to update the

bandwidths of all messages travelling from S for each of the messages being broadcast. Instead, it is more efficient to update these bandwidths only once, as shown in Figure 3.17.

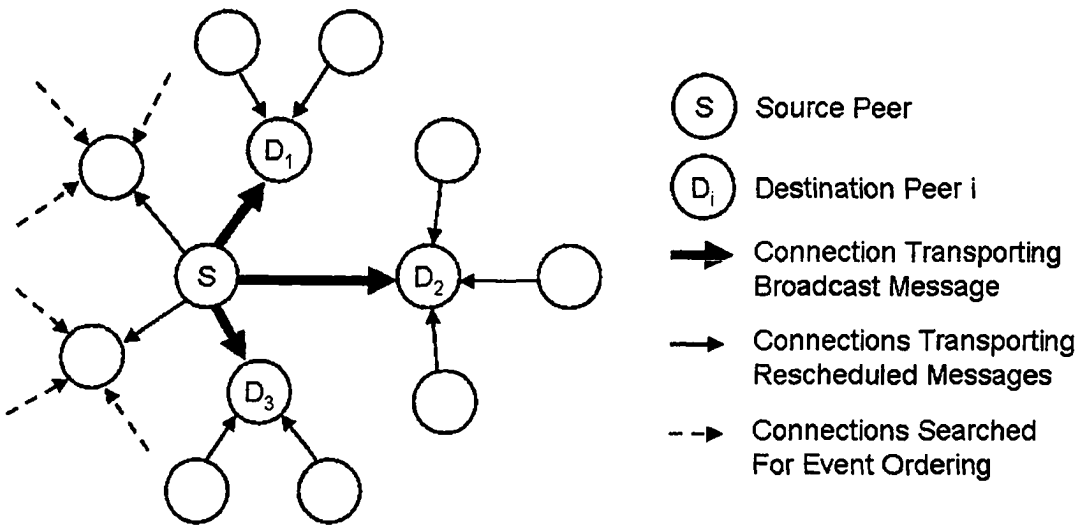

Figure 3.17 Illustration of peers and connections affected by message broadcast

This optimization results in the following running time for broadcasting a set of messages with the MES bandwidth model:

$$O\left(|UM(S)| + \sum_{D_i \in DST} |DM(D_i)| + \sum_{P \in UP(S)-DST} |DC(P)|\right)$$

In addition, when adding, removing or broadcasting messages from a source peer S, *O(|UP(S)| log(# peers downloading in the simulator))* operations may be required to update the message event heap if the minimum download message for any of the peers downloading from S has changed.

The amount of processing required for the MES bandwidth model is also optimized for situations where a message is removed from a connection which has queued messages directed from a peer S to a peer D. In this scenario, it would be inefficient to remove the message and then update the bandwidths of all messages travelling from S, and those travelling to D, only to update these bandwidths again when the queued message is added. Instead, the MES bandwidth model can take advantage of the fact that all messages travelling from S to D are allocated an equal bandwidth. The only change that might occur in this scenario is that the minimum message between S and D may now have a different expected time, requiring a search through the set DC(D) of download connections of the peer D to find the one that is now storing the earliest download message.

This optimization results in the following running time for replacing a message with the MES bandwidth model:

$$O\left(\left|UM(S) \cap DM(D)\right| + \left|DC(D)\right|\right)$$

The MES bandwidth model is similar to the network flow bandwidth model proposed for the Narses network simulator [GB02], except that the Narses network simulator does not allow each node (peer) to have differing upload and download bandwidths, and does not discuss the optimizations that can be made for efficient message storage and processing.

This model also bears many similarities to the one proposed by Michael Iles [ILE02], which also employs a flow-level model. Iles observed that allocating bandwidth bares many similarities to the classic Min/Max Flow problem [CLR90], and thus devised an *O(nlogn)* algorithm for computing the bandwidth of one message in a network with n messages. While Iles' bandwidth model is likely more 'realistic' in terms of modeling the propagating effect of bandwidth changes throughout the network, it does not scale well beyond small network topologies.

3.4.5 Delays

An important realization is that bandwidth may not be the only contributing factor to the duration of message transmission in the simulator. There may be other factors which contribute a delay to transmission times, such as CPU delays for compression, encryption, or communication delays caused by errors, or distance-related latency.

To allow for the simulation of these types of delays, the method for retrieving the expected time in the message class may be overridden by protocol-specific subclasses (Section 4.2). For example, a message subclass may include an extra field which stores an encryption delay field. In this case, the method for retrieving the expected time could be overridden to return the value of the expected time event field plus the value of the encryption delay field. This allows the simulator to simulate

delays that are known prior to the events creation, avoiding the complexity of delaying a message already in transit.

3.4.5.1 Latency

Each peer object stores x and y coordinates, allowing the simulation of geographical or topological distances between them. The value of these coordinates may be specified directly by the protocol at initialization, or provided by a topology generator (Section 4.2.2.1). Message subclasses may choose to store a delay field whose value is determined proportionally to the distance between its source and destination peers.

3.4.6 Removing Groups of Events

The simulator provides a variety of methods for removing several action or message events at once, which are optimized for efficient processing. These methods can be useful when simulating various scenarios, such as disconnection of a peer from the network, or the disconnection of two peers from each other. Figure 3.18 illustrates the interface provided by the simulator for the removal of groups of events.

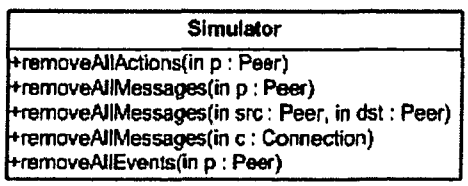

Figure 3.18 Class diagram illustrating methods for removing groups of events

The result of the use of these methods will likely be the cancellation of several actions or messages. In some scenarios, it may be desirable to notify the affected peers so that they can handle the cancellations in a suitable manner. To this end, the action and message classes provide default cancel methods which are called by the simulator whenever instances are removed from the simulator. These methods may be overridden by action or message subclasses to perform arbitrary operations, in a manner similar to the execute method which is described in Sections 3.4.3.2, and 4.2.3.

3.5 Statistics

It is important to be able to track the progress of a simulation run. This section describes the components of the simulator that allow statistics about actions and messages to be collected and reported.

3.5.1 Statistics Reporter

The simulator maintains a reference to a statistics reporter which is responsible for reporting statistics about the operation of the simulator over regular simulation time intervals. The reporting time interval may be specified for the statistics reporter as a multiplier with respect to the global reporter time interval stored by the simulator. As each time interval elapses, the simulator notifies the statistics reporter of this fact so

that it can update and report statistics pertaining to the latest interval. Aside from storing global simulation statistics, the primary responsibility of the statistics reporter subclass is the storage and maintenance of a list of statistics collector objects, which are described in section 3.5.2.

3.5.2 Statistics Collectors

Statistics collector objects are responsible for collecting the statistics pertaining to the creation, deletion and execution of instances of individual action or message event subclasses during successive reporting time intervals. There are two different types of statistics collectors; action statistics collectors, and message statistics collectors. Statistics collectors may be created and bound to action or message subclasses by the constructor of the statistics reporter object. The statistics reporter uses two hash tables [CLR90] to store bindings of statistics collectors to action or message subclasses. The keys of these hash tables are the Java class objects associated with each action or message subclass, and the values are the statistics collector object bound to them. Whenever an operation is performed on an action or message event, the simulator notifies the statistics collector bound to the event's class so that it can update its collected statistics.

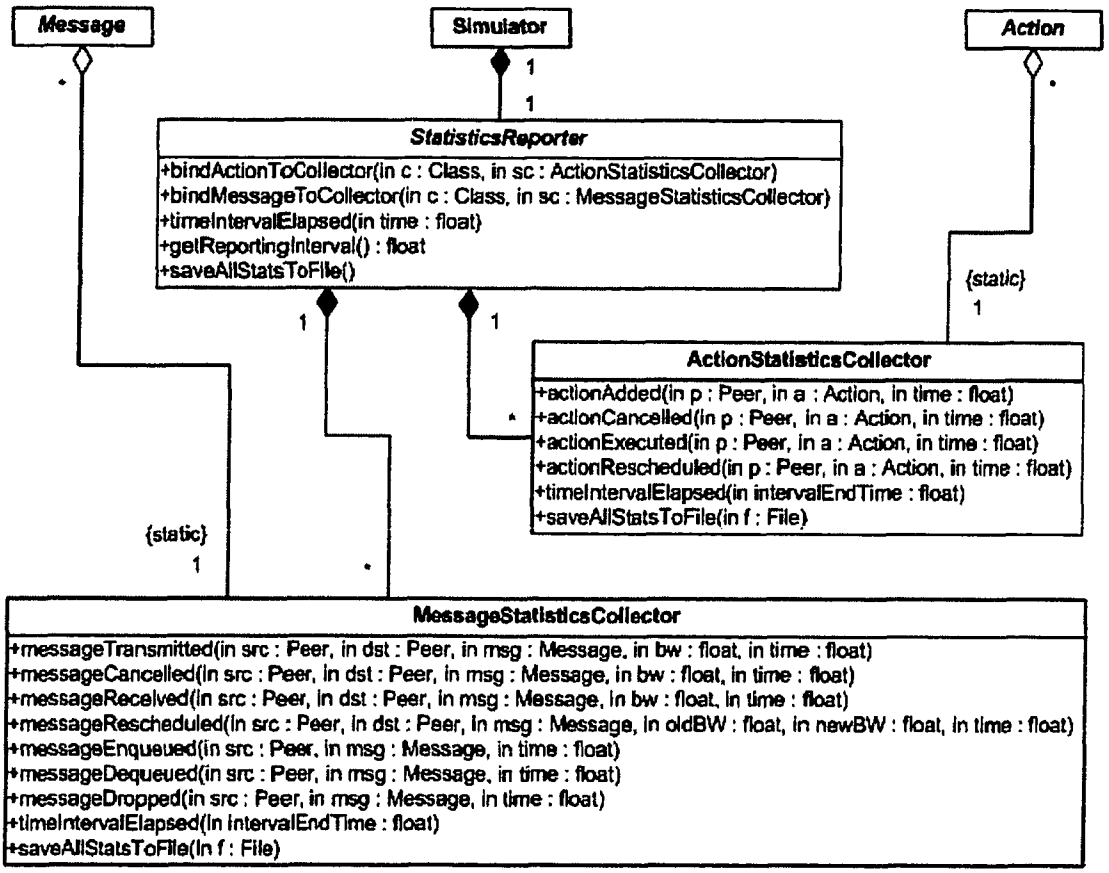

Figure 3.19 Class diagram showing actions and messages bound to statistics collectors

Statistics collectors maintain statistics for events both at a global, and per-peer level of detail. Global level statistics are used to count the number of operations that were performed on all instances of the corresponding event, such as the total number of events executed. Peer-level statistics are used to track the effect of event operations on each peer, providing valuable insight into their impact on the simulated network. Peer level statistics are maintained in arrays with values for each peer, accessed using the unique index value stored by each peer (Section 3.1). For example, when a

statistics collector is notified of an action being added for peer 0, it increments the value at index 0 of its 'number of actions per peer' array by one.

The statistics maintained by statistics collectors are only ever retrieved by the statistics reporter at the end of each reporting time interval, which means that the peer-level statistics only convey information about that exact moment in simulated time. To combat this problem, statistics collectors may optionally maintain an average value for each peer-level statistic throughout the reporting time interval. For example, if peer 4 had only one action for half of the reporting time interval, then the value of 0.5 is stored at index 4 of the 'average number of actions per peer' array.

For simulated networks with large numbers of peers, interpreting the sheer amount of statistics collected for each peer at every time interval would be cumbersome to say the least. A simpler and more manageable approach is to use statistical methods to get a general picture of how the events were distributed across the peers. For this purpose, statistics collectors can calculate the mean, standard deviation, skewness and kurtosis of the peer-level statistics during each reporting time interval. These statistical values are often referred to as the 1^{st}, 2^{nd}, 3^{rd} and 4^{th} moments [Pap84] of a distribution respectively. Each statistics collector may be configured with an integer value between 0 and 4 that specifies the largest moment that it should calculate. (0 disables peer-level statistics.)

Table 3.1 Available measures of peer-level statistics

Moment	Name	Formula	Description
1	Mean	$\bar{x} = \frac{\sum x_i}{n}$	Un-weighted average of a distribution.
2	Standard Deviation	$\sigma = \sqrt{\frac{\sum (x_i - \bar{x})^2}{n-1}}$	Deviation of a distribution from the mean.
3	Skewness	$s = \frac{\sum (x_i - \bar{x})^3}{(n-1)\sigma^3}$	Degree of asymmetry of a distribution.
4	Kurtosis	$k = \frac{\sum (x_i - \bar{x})^4}{(n-1)\sigma^4} - 3$	Degree of peakedness of a distribution.

Table 3.2 lists the statistics collected by action statistics collectors over each simulation time interval.

Table 3.2 Statistics collected for actions

Global-Level
Number of actions added
Number of actions rescheduled
Number of actions cancelled (removed)
Number of actions executed (& removed)
Peer-Level (Current, Average)
Number of actions

Table 3.3 lists the statistics collected by message statistics collectors over each simulation time interval.

Table 3.3 Statistics collected for messages

Global-Level
Number of messages transmitted (added)
Number of messages rescheduled (bandwidth change)
Number of messages cancelled (removed)
Number of messages received (executed & removed)
Number of messages enqueued
Number of messages dequeued
Number of messages dropped
Peer-Level (Current, Average)
Number of messages in transmission (Incoming / Outgoing)
Size of all messages in transmission (Incoming / Outgoing)
Bandwidth allocated to all messages in transmission (Incoming / Outgoing)
Number of messages queued pending transmission (Outgoing)
Number of message dropped (Outgoing)

3.5.4 Statistics Persistence and Visualization

The statistics stored by statistics reporters and their statistics collectors are saved to a data file at the end of each simulation run. In order to facilitate the visualization, the data collected for each time interval during a simulation run is stored in a format which is compatible with the open-source JFreeChart [Jfr05] data plotting library. This allows any of the data to be plotted on a chart, typically with the simulation time values plotted on one axis, and the collected statistics plotted on the other. To enhance the analysis of the collected statistics, a graphical user interface was implemented which allows for multiple statistics data files to be loaded. Multiple sets of the

statistics from the same or different files can be selected for plotting to a chart, allowing for comparative analysis. The results presented in chapter 6 were generated using this functionality.

3.6 Configurability

The core simulator architecture described in this chapter was designed to be highly configurable. A variety of general parameters can be specified in order to influence the operation of the simulator, irrespective of the peer-to-peer protocol that is being studied. The following is a list of these configurable parameters, along with references to the section of this chapter where they are described.

- **Bandwidth distributions** – Distributions of varying upload and download bandwidth values can be assigned to peers through the use of a class which implements the bandwidth distributor interface (Section 3.4.4).
- **Bandwidth models** – Different bandwidth modeling techniques for determining how messages between peers consume bandwidth can be employed through the use of a subclass of the connection manager class (Section 3.4.4).
- **Content generation, distribution & selection** – Different varieties of content repositories can be generated, with arbitrary schemes for distribution of content to peers and selection of content for search queries. This can be

accomplished by classes which implement the content repository and content distributor interfaces (Section 3.3).

- **Protocols** – The simulator core can be extended to support the simulation of arbitrary peer-to-peer protocols. This is described in detail in Chapter 4.

Refer to Appendix C for a description of how these configurable parameters can be set prior to execution of a simulation run.

3.7 Summary

This chapter described in detail each of the components that make up the core of the simulator. Section 3.1 described how the simulator keeps track of the active and inactive peers that may be present in the network. Section 3.2 described how the simulator keeps track of connections used for inter-peer communication and Section 3.3 describes how the simulator stores a repository of content available for distribution to and retrieval from peers in the network. Section 3.4 described how events are handled by the simulator. Sections 3.4.1 and 3.4.2 described action events which are local to a peer, and message events which are communicated between peers. Subsequently, Section 3.4.3 described how the simulator imposes an ordering on the events to be able to execute them efficiently, and Sections 3.4.4 and 3.4.5 described how the simulator uses bandwidth and other forms of delay to determine the time between a message's transmission and reception. Finally, Section 3.5 described the

infrastructure provided by the simulator for the collection and reporting of statistics pertaining to action or message events, and section 3.6 provided a review of the parameters that can be used to configure or extend the functionality of the simulator core.

All of the components described in this chapter were designed to minimize memory usage, and to allow for the simulation of networks of peers running arbitrary protocols. These aspects of the simulator design are described in detail in Chapter 4.

Chapter 4

Scalability and Extensibility

The primary objectives during the development of the peer-to-peer simulator were scalability and extensibility. Scalability in this context refers to the ability of the simulator to simulate networks with a large number of peers, in a time-efficient manner. Extensibility refers to the ability of the simulator to support the simulation of networks with peers running arbitrary peer-to-peer protocols. Section 4.1 discusses how the simulator was designed to be scalable, and section 4.2 describes how the simulator core can be extended to support arbitrary peer-to-peer protocols.

4.1 Scalability

4.1.1 Storage

The key concept behind the scalability of the simulator is the minimization of memory usage by core components which occur more frequently than others. If we consider a modest traffic scenario where each peer (Section 3.1) in the simulator is communicating with at least one other peer, we know that the number of connections (Section 3.2) must be greater than or equal to the number of peers. Also, since

connections only exist when at least one message (Section 3.4.2) is being transmitted across them, we know that the number of messages is greater than or equal to the number of connections. Similarly, the number of actions (Section 3.4.1) is typically greater than or equal to the number of peers.

Number of Messages >= Number of Connections >= Number of Peers Number of Actions >= Number of Peers

Using these expressions, it makes sense to minimize the memory usage of messages and actions, even at the expense of extra memory usage by connections or peers.

Messages and actions are kept lightweight by only storing their start and expected times, and references for storing them in the lists at the appropriate connection or peer. The use of singly linked lists for storing the lists of messages pertaining to a connection or actions pertaining to a peer means that only one reference to the next event in the list is required for each message or action. Another advantage of singly linked lists is that they need not be explicitly represented as an object, saving the cost of an object header for each peer or connection. Instead, peers and connections simply store a reference to the first action or message in their list.

Messages and actions are freed from the requirement of storing their target peer or connection because of the event processing (Section 3.4.3) utilized by the simulator. Whenever a message is processed, it is always retrieved from the event

heaps via references to the relevant destination peer and connection, which are passed to the message upon execution. Similarly, actions are always retrieved for processing via references to the relevant peer. The statistics collection method employed by the simulator (Section 3.5.2) also frees actions and messages from the requirement of storing instance-level references to their statistics collectors.

Messages require more information than actions; namely size and bandwidth. The storage required for this information is marginalized by storing the size of each type of message in static or class fields, and by using a bandwidth model such as the Minimum Equal Share Bandwidth model (Section 3.4.4.1) which allows bandwidth for the message to be determined implicitly from its source and destination peers.

Connections are relatively lightweight, yet require more storage than messages or actions. Connections store references to two end peers, a list of messages being transported, and references to the next incoming/outgoing connections at the source and destination peers. Connections are stored in singly linked lists at their source and destination peers, minimizing the number of required references. Under typical network traffic scenarios, the storage used for the transmission of messages over connections is mitigated by the fact that most connections will be transporting multiple messages.

The simulator can be operated so that bi-directional communication between two peers employs either two directed connections, or one undirected connection. The

choice of whether to run the simulator with directed or undirected connections depends on the typical traffic scenarios of the protocol in use. For protocols with a high percentage of simultaneous bi-directional messaging, using undirected connections will result in one less object header, and one pair less of source & destination peer references. Conversely, if a protocol only rarely has messages traveling to and from the same end peers, using directed connection will be more efficient.

Table 4.1 displays the memory usage in bytes for each of the core components of the simulator. For a detailed explanation of how these values were calculated, refer to Appendix B.

Table 4.1 Memory usage in bytes per core simulator component

Component	Memory Usage (bytes)
Action	20
Message	20
Directed Connection	32
Undirected Connection	48
Peer Total	140
Peer	60
Connection Manager	68
Protocol	12

4.1.2 Performance

Another important factor in the scalability of the simulator is its performance. A scalable simulator must achieve the goal of being able to simulate a large number of

peers and events, yet balance this against the need to achieve results in a timely manner. This section summarises the six key operations of the simulator related to adding and removing events, and lists the asymptotic running time of each operation.

The following is a list explaining the notation used in the asymptotic running times:

- NA = Total number of peers with scheduled actions.
- NM = Total number of peers with scheduled download/incoming messages.
- S = Source peer of a message.
- D = Destination peer of a message.
- T = Target peer of an action.
- DST = Set of all destination peers of a broadcast message.
- UM(X) = Set of all messages being uploaded by peer X.
- DM(X) = Set of all messages being download by peer X.
- DC(X) = Set of all download/incoming connections to peer X.
- UP(X) = Set of all peers to which a peer X is uploading messages.
- M(X, Y) = Set of all messages travelling from peer X to peer Y.
- A(X) = Set of all actions pertaining to a peer X.

Transmitting, Broadcasting or Removing Messages

Transmitting, broadcasting or removing messages requires processing to update the bandwidths and expected times of messages according to the bandwidth model in use, and processing to ensure the correct event ordering required by the simulator. [Note: these running times assume that the Minimum Equal Share bandwidth model is in use].

Table 4.2 Asymptotic running times of message transmission, broadcast and removal

Transmit	$O\left(\lvert UM(S)\rvert + \lvert DM(D)\rvert + \sum_{P \in UP(S)} \lvert DC(P)\rvert + \lvert UP(S)\rvert * \log_2(NM)\right)$
Broadcast	$O\left(\lvert UM(S)\rvert + \sum_{D_i \in DST} \lvert DM(D_i)\rvert + \sum_{P \in UP(S)-DST} \lvert DC(P)\rvert + \lvert UP(S)\rvert * \log_2(NM)\right)$
Remove	$O\left(\lvert UM(S)\rvert + \lvert DM(D)\rvert + \sum_{P \in UP(S)} \lvert DC(P)\rvert + \lvert UP(S)\rvert * \log_2(NM)\right)$

Enqueuing or Dequeing Messages

As mentioned in Sections 3.2 and 3.4.2, the simulator supports queuing of messages on the connections on which they are to be transported. The processing time required to enqueue or dequeue a message on a connection depends on the implementation of the queue in the protocol-specific connection subclass.

Whenever a message is removed from a connection with queued messages, the simulator replaces the removed message with the first one from the connection's queue. In this situation, the earliest message for the connection may need updating,

which in turn may require updating the minimum incoming connection of the destination peer.

Table 4.3 Asymptotic running time of message replacement

Replace	$O(\|M(S, D)\| + \|DC(D)\| + \log_2(NM))$

Adding or Removing Actions

Adding or removing actions only requires processing to ensure the correct event ordering required by the simulator. Namely, the earliest expected action must be at the head of the list of actions stored by each peer.

Table 4.4 Asymptotic running times of action addition and removal

Add	$O(\|A(T)\| + \log_2(NA))$
Remove	$O(\|A(T)\| + \log_2(NA))$

As the asymptotic running times listed in this section show, the simulator minimizes the amount of processing required for each of the six key operations. In general, the amount of processing required to process an event at a peer is bounded by the number of events at that peer. This is a crucial feature if the simulator is to handle networks with large numbers of peers, and even larger numbers of events.

4.2 Protocol Extensibility

Each peer (Section 3.1) in the simulator stores a reference to an instance of the abstract protocol class. The simulator supports simulations of peers running arbitrary

peer-to-peer protocols via subclasses of this abstract protocol class. The only strict requirement of protocol subclasses is that they store a reference to their associated peer object, which provides a large degree of freedom when simulating different varieties of peer-to-peer protocols.

Protocols subclasses are distributed to the peers via the Protocol Factory class which implements the protocol distributor interface, as shown in Figure 4.1. Protocol distributors are free to distribute any protocol subclass to the peers contained in the peer graph. This allows for the simulation of heterogeneous peer-to-peer networks. Refer to Appendix C for a description of how the protocol to use can be specified prior to execution of the simulator.

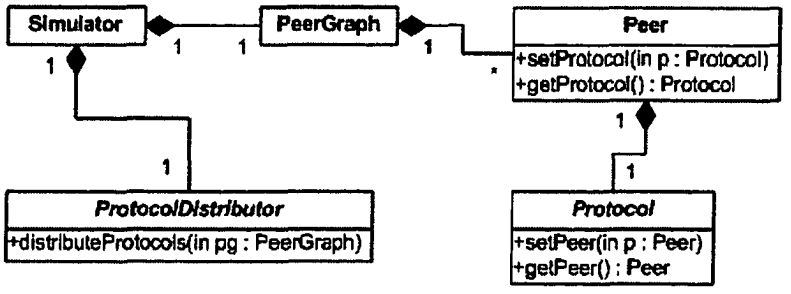

Figure 4.1 Class diagram illustrating protocol distribution related components

Sections 4.2.1 to 4.2.5 provide an overview of how protocol subclasses can be used to extend the simulator, allowing it to simulate arbitrary peer-to-peer protocols.

4.2.1 Content

As described in section 3.3, the storage of a peer's content (ex: the files a peer is sharing with the network) is delegated to the protocol object. To facilitate the distribution of content, protocol subclasses must implement two methods for adding or removing content based on content records which are provided as parameters (See figure 4.2).

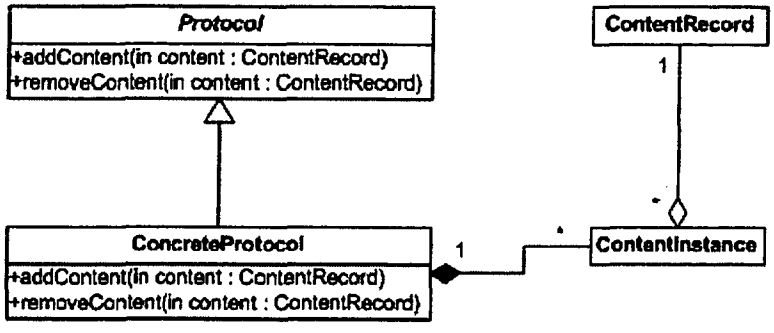

Figure 4.2 Class diagram of how a concrete protocol may choose to store content

Protocol subclasses are free to store content in the manner most suited to their corresponding peer-to-peer protocol. This may involve storing a list of direct references to the content records, or references to a wrapper class (see figure 4.2), which enable more advanced search and retrieval. For example, the Freenet protocol (Section 2.1.2.4) stores content in a stack, along with locator information for the peer that forwarded that content.

4.2.2 Links

Another key responsibility of protocol subclass instances is the storage and management of links between their associated peers. For many protocols, links between peers are bi-directional (i.e. if peer A has a link to peer B, then peer B has a link to peer A). To facilitate the addition and removal of bi-directional links, a subclass of the abstract protocol factory class must be created, which implements methods for adding and removing links (see figure 4.3). The concrete protocol factory class can leverage intimate knowledge of the corresponding concrete protocol class interfaces to add or remove links in the appropriate manner.

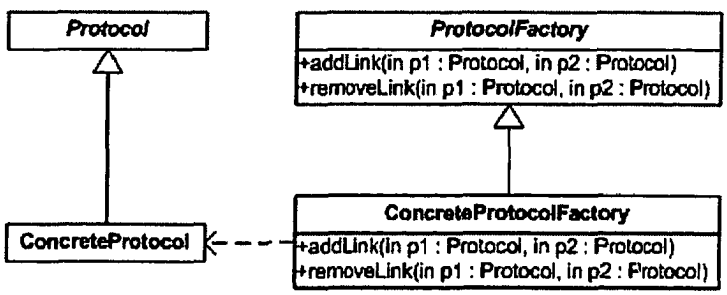

Figure 4.3 Class diagram of components related to link addition and removal

The use of a protocol factory for adding or removing links provides a generic interface for use by the simulator. This allows any protocol-level topology to be loaded by a topology generator (Section 4.2.2.1), regardless of which actual peer-to-peer protocol is in use. This also allows concrete protocol subclasses to store links in a manner of their choosing. For example, a highly-storage efficient approach would be

to have each protocol instance simply store a list of references to other protocol instances (figure 4.4-a). A more demanding protocol might require a plethora of data for each link, in which case it would make more sense to represent each link explicitly via a link object with associated fields (figure 4.4-b). Finally, some protocols may require access to communication-level information, such as the amount of communication-level message traffic currently traversing a link. In this case, a viable approach is to represent links via subclasses of the connection class of the simulator core (figure 4.4-c). Representing links via subclasses of the connection class also allows for messages to be queued on the connection

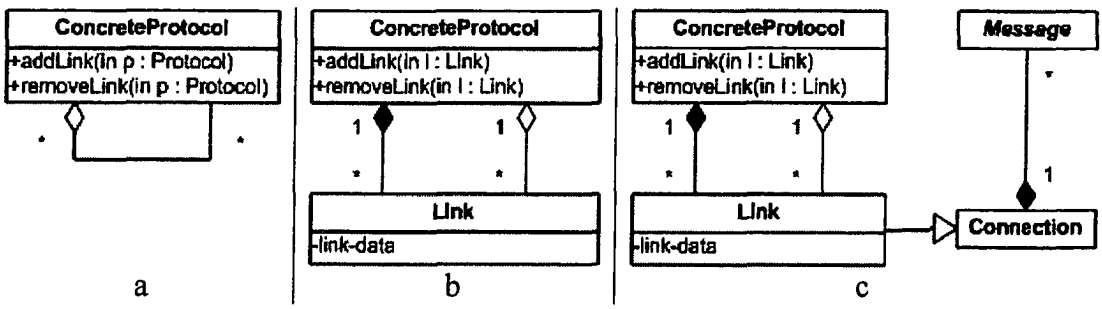

Figure 4.4 Class diagrams of different approaches for protocols to store links

4.2.2.1 Topology Generation

The simulator can be initialized or boot-strapped with pre-determined protocol-level topologies by providing it with a class which implements the topology generator interface. Topology generator classes are passed a reference to the simulator's peer graph (and associated protocol factory) which they can utilize to add protocol-level

links between pairs of peers. Figure 4.5 illustrates the components utilized by the topology generator interface for creating protocol-level links between peers.

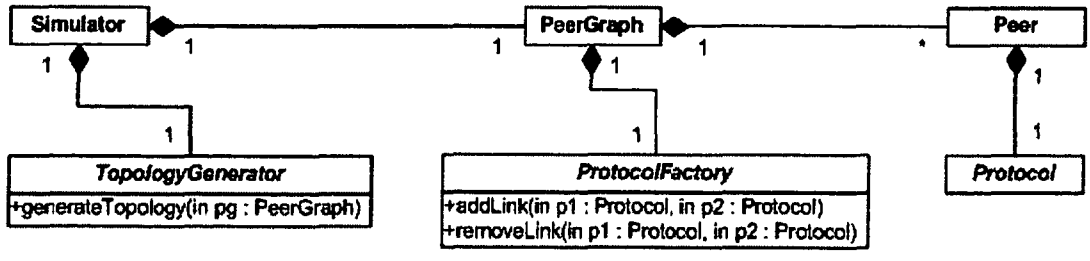

Figure 4.5 Class diagram of topology generator related components

Topology generator classes may choose to generate hard-coded or random protocol-level topologies, or they may elect to load the topology from a file generated by a third-party topology generator. The default topology generators included with the simulator include a simple, hard-coded topology generator used for testing, and a more advanced topology generator which loads the topology from an output file generated by the BRITE topology generator (Section 2.2.3.1.6). Topologies generated by BRITE include information about the x and y coordinates of each node in the topology, and this information can be passed to each protocol to simulate message delays proportional to the geometric distance between peers (See Section 3.4.5.1).

4.2.3 Event Handling

The final responsibility of concrete protocol subclasses is event handling, or more specifically the handling of executed actions or messages. Protocol subclasses should

implement methods that can accept any action or message that can be triggered on them as parameters, as illustrated in Figure 4.6.

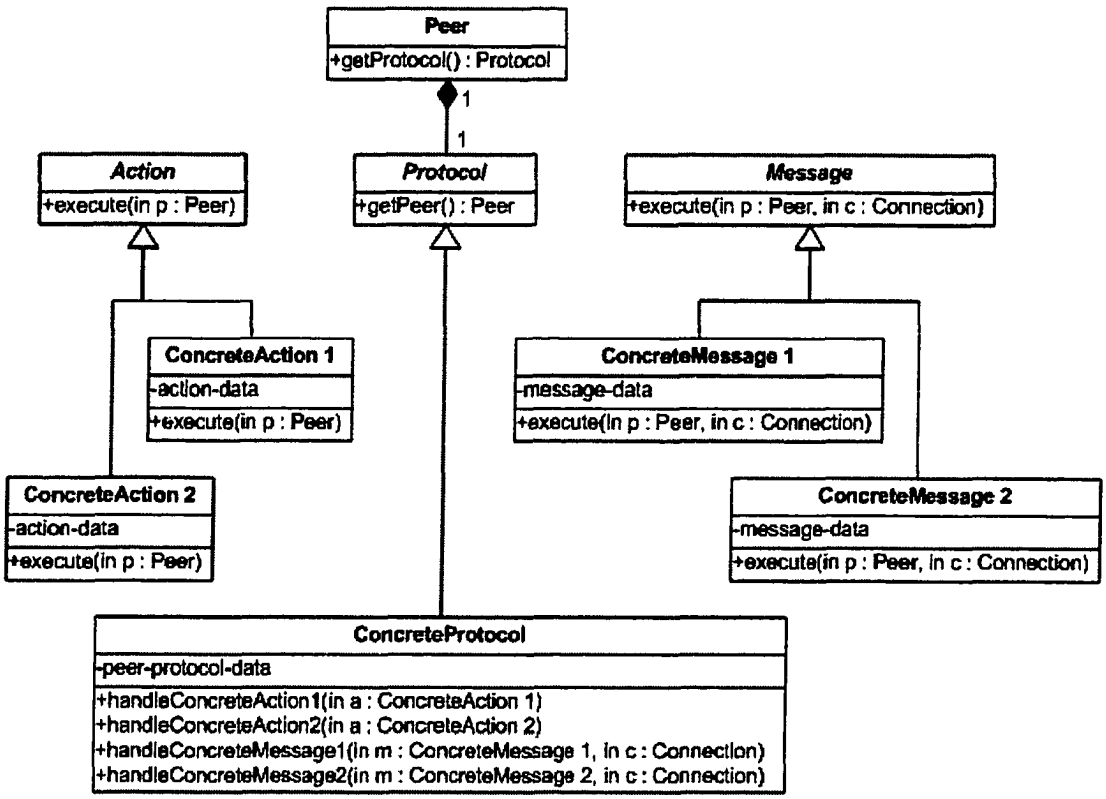

Figure 4.6 Class diagram showing event subclasses interacting with protocol subclasses

Subclasses of the action class may be created to correspond to any protocol-specific action local to a peer, such as connection to the network, or the initiation of a search for content on the network. Action subclasses may add additional fields for protocol-specific data, and must implement an execute method which takes the target peer as a parameter. The action can be handled by calling the appropriate method of

the protocol subclass associated with the peer from the execute method (see figure 4.7).

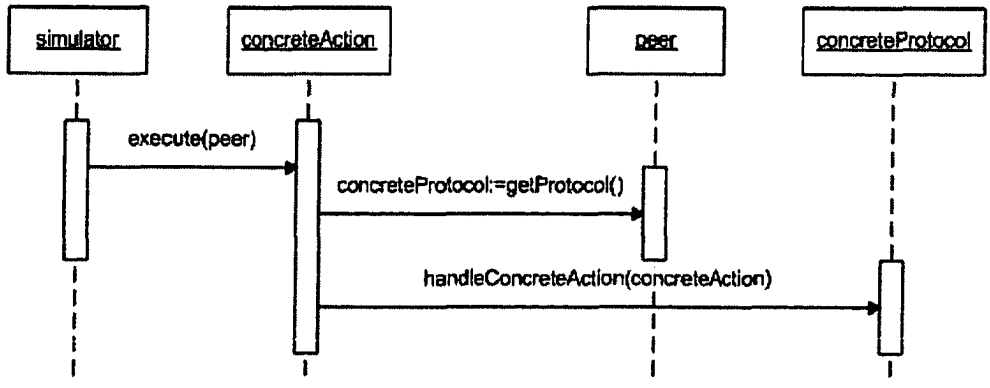

Figure 4.7 Sequence diagram showing concrete actions being handled by concrete protocols

Subclasses of the message class may be created to correspond to any protocol-specific message that can be passed between and to peers, such as a request to join the network, or the transmission of a search for content stored somewhere in the network. Message subclasses may add additional fields for protocol-specific data, and must implement an execute method which takes the destination peer and transporting connection as parameters. The message can be handled by calling the appropriate method of the protocol subclass associated with the destination peer from the execute method (see figure 4.8). The message handling method of the destination peer's concrete protocol subclass may require a reference to the connection used to transport the message in order to resolve the source peer of the message.

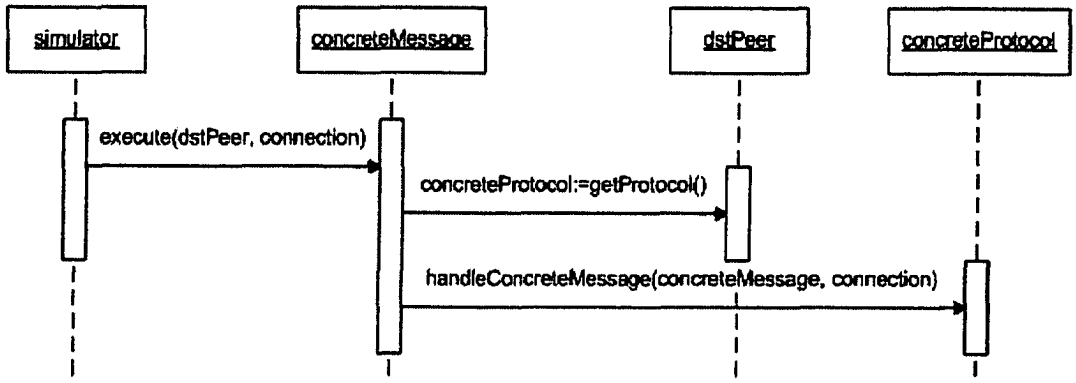

Figure 4.8 Sequence diagram showing concrete messages being handled by concrete protocols

A major benefit of handling events in this manner is that the amount of code implemented in the action and message classes is kept to a minimum. Instead, all of the logic for handling events for a given protocol is grouped together in one place in the appropriate protocol subclass. Another benefit is that different protocol subclasses can handle the same actions or messages in different ways, which facilitates the simulation of peer-to-peer networks with heterogeneous protocols.

4.2.4 Peer behaviour

As described in section 3.4.1.1, actions are fed into the simulator by action generators. An action generator should be implemented for each of the action subclasses implemented by the simulator. The choice of whether to implement an initial or continuous action generator depends on whether the actions will be **initial** and recurring (ex: polling the network for connectivity information) or **continuous** and sporadic (ex: user queries the network for content). In either case, action generators

must implement the required methods that when called add actions to the simulator using the interface described in Section 3.4.1. The generation methods of action generators are responsible for determining which action subclass to instantiate, as well as the number of actions to create, and their target peers.

4.2.5 Statistics

As described in section 3.5, the simulator requires that a statistics collector object be bound to each protocol-specific action or message subclass (See figure 3.19). If it is of interest to collect additional statistics for protocol-specific actions or messages, the action and message statistics collector class may be subclassed to override the default methods, or add new ones corresponding to event operations other than addition, removal or rescheduling.

It is also possible to configure a protocol-specific statistics reporter subclass to store lists of statistics that are not bound to a specific action or message subclass. These statistics lists may be added as instance variables of the reporter subclass, and the time interval elapsed method of the reporter subclass may be overridden to trigger the collection of statistics for populating them with data.

4.3 Summary

This chapter began with Section 4.1 which summarizes the techniques that were used to minimize the memory overhead of the most frequently used components of the simulator core, and to optimize the time required to process them. A table summarizing the memory overhead of each of these components was provided, allowing for rough estimates of the total system memory required to simulate peer-to-peer networks with various population sizes and traffic densities (See Section 5.5).

Section 4.2 gave a detailed description of how the components of the simulator core may be extended to support the simulation of arbitrary peer-to-peer protocols. The core components which may be extended for this purpose include: actions, messages, connections and protocols. The action and message classes may be subclassed to support any form of protocol-specific behaviour or communication. The connection class may be subclassed to provide protocol-level links with access to low-level message transmission information. The protocol sub-class may be subclassed to handle the execution of action and message sub-classes, and to store content and protocol-level links in the manner most suitable to the protocol. This section also included a discussion of how topologies of protocol-level links can be generated or loaded through the use of topology generator tools.

The ease with which the simulator core can be extended to support arbitrary peer-to-peer protocols is illustrated by the Gnutella protocol implementation, which is described in Chapter 5.

Chapter 5

Gnutella Implementation

Gnutella is a continuously evolving protocol, and its network is made up of heterogeneous peers, each running their own, possibly different version of the Gnutella protocol.

The original or *standard* version of Gnutella [Lim] consists primarily of two general types of messages, queries, and pings. Queries are used to search the network for peers storing desired content, while pings are used to explore the network to maintain and increase peers' connectivity to one another. Both queries and pings are handled in the same manner; when a peer receives a query or ping message (that it has not already seen), it forwards the message to every other peer to which it is linked, that is it forwards the message to its neighbourhood. For each message, this process is repeated a fixed number of times determined by the originating peer, which sets a maximum hop count value, thus enforcing a horizon on message retransmission. Figure 5.1 illustrates this flooding of ping or query messages on a sample network, with each peer marked with the hop count of the received message. The peer where

the message originates is marked with an X, and redundant messages are shown as dashed arrows.

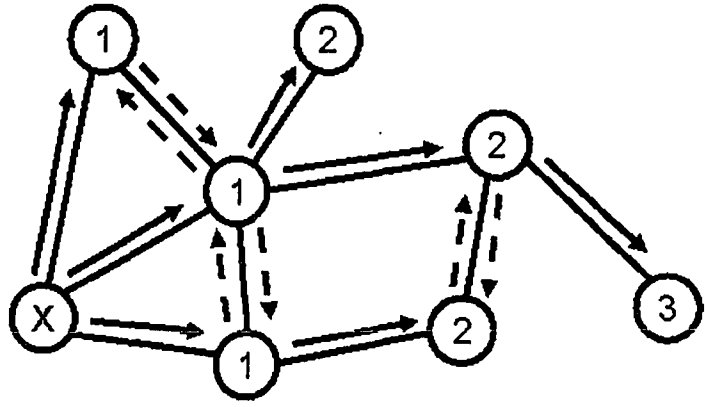

Figure 5.1 Flooding of query or ping messages in a sample Gnutella network

In addition to forwarding messages, each peer that has content matching a query, will send back a query reply to the peer who forwarded it the query. Similarly, each peer that is accepting new connections will send back a pong in response to each ping that it receives. A peer that receives a query reply or pong message will forward that message back along the original transmission path until it reaches the originator. Each query reply or pong message maintains a hop count value, which is incremented by one each time the message is transmitted by a peer along the return path. This form of message flooding with replies traced back along the original path is useful for allowing a peer to communicate with a large proportion of peers in an unstructured network such as Gnutella, yet it has several drawbacks. It necessitates that each peer maintain routing information to trace back the query replies or pongs, and the peers

closest to the originating peer are forced to handle a great deal of return message traffic. Figure 5.2 illustrates the back tracing of replies in the sample network from Figure 5.1, with the each peer marked with the number of replies.

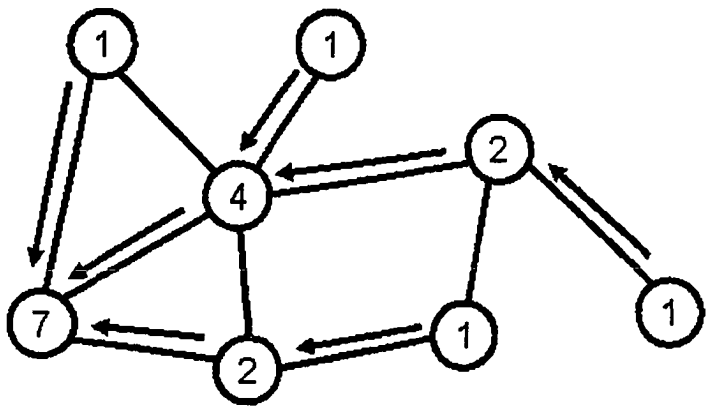

Figure 5.2 Back tracing of message replies in a sample Gnutella network

To reduce the burden of ping/pong traffic, Gnutella protocol developers conceived a new version of Gnutella which introduces the notion of *pong caching* [Roh]. Pong caching consists of three key elements: pong limiting, pong caching, and ping multiplexing. Pong limiting specifies that peers return or trace back a maximum number of pongs (typically 10) in response to a ping. Pong caching specifies that peers maintain a cache of a number of the most recent pongs they have received. Pongs in the cache are grouped by hop count, so that pongs with a good distribution of hop count distances can be returned in response to a ping. To ensure that pongs remain 'fresh', the cache is emptied at intervals, and is refilled by broadcasting pings to neighbouring peers. Ping multiplexing specifies that peers maintain the number of

pongs required by each of their neighbouring peers. Thus, if a peer receives a ping from a neighbouring peer and its cache is empty or only partially full, it may defer responding with pongs, because it knows how many pongs to forward to each of its neighbours as its cache fills up.

Another improvement conceived by Gnutella protocol developers for reducing message traffic is *flow control* [Roh02b]. As can be seen in figure 5.2, certain peers in the Gnutella network may be required to handle a great many messages. For peers with low bandwidth, high message traffic translates to low bandwidth allocation per message. Under extreme message traffic conditions, bandwidth allocation per message may be so low that the messages are delayed until they are no longer relevant. Flow control addresses this problem by capping the maximum number of messages being sent by a peer to each of its neighbours, and storing additional messages in fixed-size LIFO (last-in first-out) queues for each neighbour connection. Each of the queues is divided into sub-queues for each type of Gnutella message. The maximum sizes of the sub-queues are ordered by the priority of their message type: query reply queue size > query queue size > pong queue size > ping queue size. A larger queue size means that more of that type of message will be selected for transmission as bandwidth becomes available. Most of the sub-queues operate on a LIFO basis to minimize the latency introduced as a message spends time in the queue. The exception to this is the query reply sub-queue, where the query replies are sorted by the number of replies that the

current peer has already sent back for the corresponding query. This gives preferential transmission to rarer query replies.

The simulator's Gnutella plug-in allows for the simulation of a network with peers running two different versions of the Gnutella protocol in tandem: Standard and Pong Caching. The Gnutella Protocol Factory class can be configured to distribute varying ratios of instances of the Gnutella Standard or Gnutella Pong Caching protocol subclasses to the peers stored in the peer graph. In addition, the Gnutella protocol distributor allows for the specification of the ratio of peers to be run with or without flow control for either protocol version.

5.1 Storing Content

Both of the Gnutella protocol subclasses in the Gnutella plug-in store content records (Section 3.3) distributed to their associated peer in a Red Black Tree [CLR90], which is keyed by the content records' unique identifier. Red Black Trees require $O(n)$ storage for n elements, and provide $O(log(n))$ insertion, removal, and search times. The decision to use a Red Black Tree was motivated by the fact that searches are the most frequent content-related operation performed in Gnutella. They are performed every time a query is received by a peer.

To support the storage of content records in this manner, a Gnutella content wrapper class (See Figure 4.2) was created which stores a reference to a content record, and references to the parent, left child and right child elements in the tree.

5.2 Events

The Gnutella plug-in supports query and ping events as described in the introduction to Chapter 5. Sections 5.2.1 and 5.2.2 describe how support for both of these types of events was implemented. Section 5.2.3 describes how support for connection & disconnection events was implemented. Where appropriate, UML class diagrams are provided to illustrate the interface of and relationship between the principle components described in each section.

5.2.1 Queries and Replies

The submission of a query request by a peer is simulated by adding a Gnutella query action instance to the simulator. A query action requires three pieces of information: the content record that is being requested, the identity of the peer who is requesting that content, and a unique identifier for the query. The simulator is fed at regular time intervals with Gnutella query actions via the Gnutella *query (continuous) action generator* (Section 3.4.1.1) class, which can use arbitrary peer and content selection strategies. For example, the random peer and content selection strategies select

random peers or content records from the peer graph or content repository. The average number of query actions to be fed to the simulator for each action generation interval can also be specified, which provides the target rate for a Poisson distribution of queries over all generation intervals.

Gnutella protocol classes implement a *handle query action* method, which is called by the execute method of the Gnutella query action class. When this method is called, Gnutella query messages which contain the action's content record and unique identifier are broadcast to all of the Gnutella protocol instances[15] in the query originator protocol instance's neighbourhood. The hop count value of these messages is set to 1, and the maximum hop count is set to that of the protocol instance. Because all query messages corresponding to the same query action will require a reference to the same content record and unique identifier, an instance of the Gnutella query record class is used to hold all this information. This allows Gnutella query messages to simply store a reference to a query record instead of the content and unique identifier, thus reducing memory usage.

Gnutella protocol classes also implement a *handle query message* method, which is called when a Gnutella query message is successfully received by a protocol instance. Upon receiving a query message, each protocol instance checks to make sure that it has not previously received a query message with the same unique identifier. To

[15] A protocol instance is bound to every peer in the simulated network

keep track of the queries it has previously seen, each protocol instance stores a singly linked list of query travel records. Whenever a new query message is received by a protocol instance, the query message's query record is used to create a query travel record, which is added to the list.

If a protocol instance has not already seen a query, and the query message's hop count value is less than its maximum value, it broadcasts the query message with an incremented hop count value to all of the protocol instances in its neighbourhood, except for the protocol instance from which it received the query message in the first place. Protocol instances which receive a query message also check to see if they are storing content which matches the content record being requested by the query. If a content match is found, the protocol instance returns a Gnutella query reply message with a hop count value of 1 to the protocol from which it received the query message. The query reply message stores a reference to the appropriate query record, and a reference to the protocol instance returning the reply.

The final query-related method implemented by Gnutella protocol classes is the *handle query reply message* method, which is called when a Gnutella query reply message is successfully received by a protocol instance. There are two possibilities when a protocol instance receives a query reply message; either the protocol instance is the originator of the query, or it is simply a protocol instance which forwarded the query across the network. In the latter case, the protocol instance needs to increment

the query reply message's hop count value, and then send it back to the protocol instance from which it earlier received the query message with the same unique identifier. To do so, the protocol instance needs to have a reference to the protocol instance that forwarded it the query message. This is accomplished by storing a reference to the forwarding protocol instance inside the query travel record that was created when the query message was received.

Gnutella protocol instances require query travel records to determine which queries they have already seen, and to route query replies back to the originator. However, once all query replies have been routed back to the originator there is no longer a need for that query's query travel records to be stored by the protocols. Taking advantage of this fact, the Gnutella query record stores a list of all query travel records, along with a count of all the instances of Gnutella query and query reply messages. When this count of instances reaches zero, all of the travel records in this list are removed from the protocol instance which own them. This results in a considerable savings in memory usage per protocol instance. The only cost is that each travel record must store a reference to the protocol instance which owns it.

Figure 5.3 illustrates the relationships between the classes of the Gnutella plug-in which are related to queries and query replies.

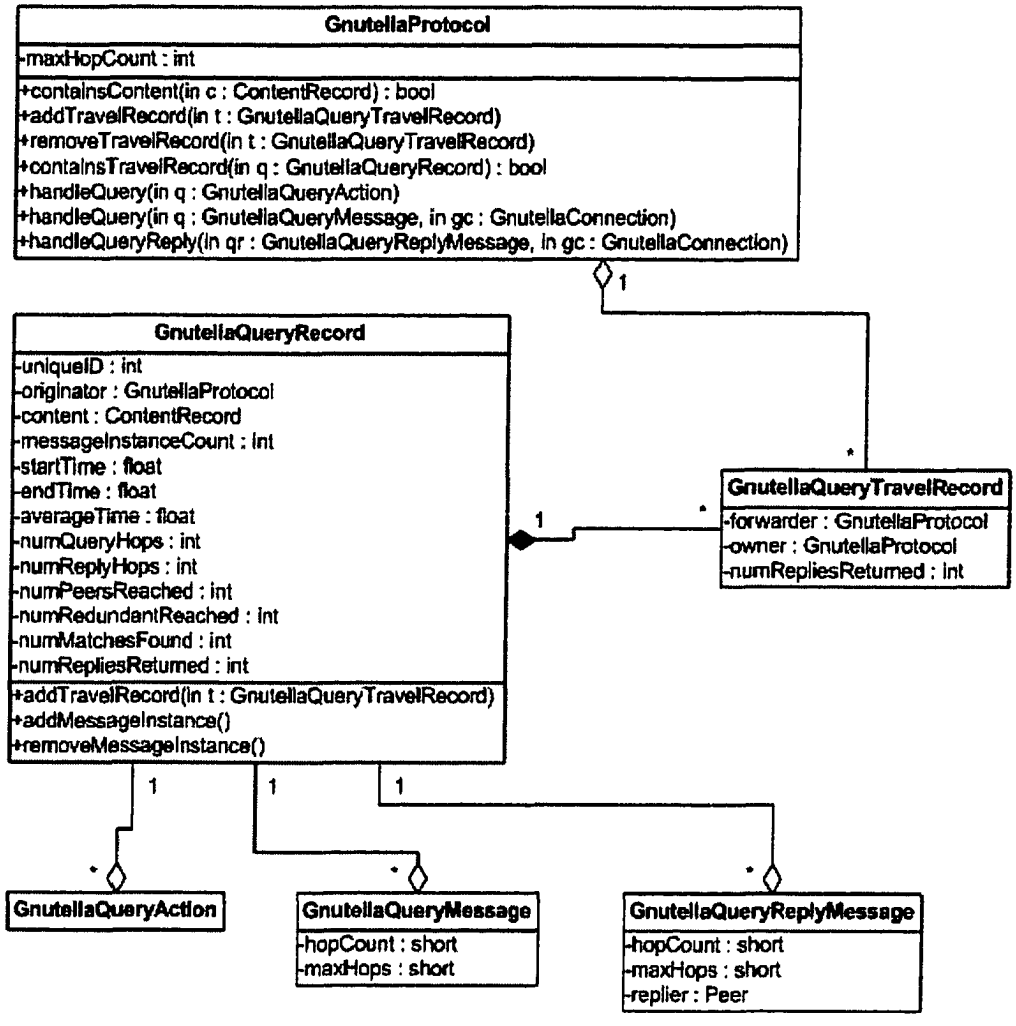

Figure 5.3 Class diagram of Gnutella Query related components

5.2.2 Pings and Pongs

To allow for the two different versions of Gnutella which handle pings and pongs differently, two subclasses of the Gnutella protocol class were created as illustrated in Figure 5.4.

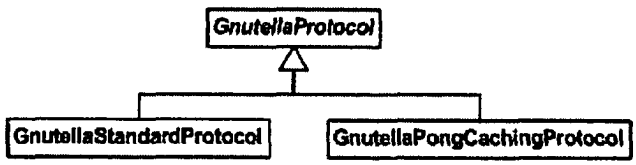

Figure 5.4 Class diagram of Gnutella protocol subclasses for standard and pong caching versions

The *standard* version of Gnutella processes ping and pongs in exactly the same way that queries and query replies are handled (Section 5.2.1). The principle difference is that with pings, pongs are returned if a protocol instance is accepting new links, whereas with queries, query replies are returned if a protocol instance has the content being sought. Figure 5.5 illustrates the relationships between classes related to pings and pongs for standard Gnutella peers.

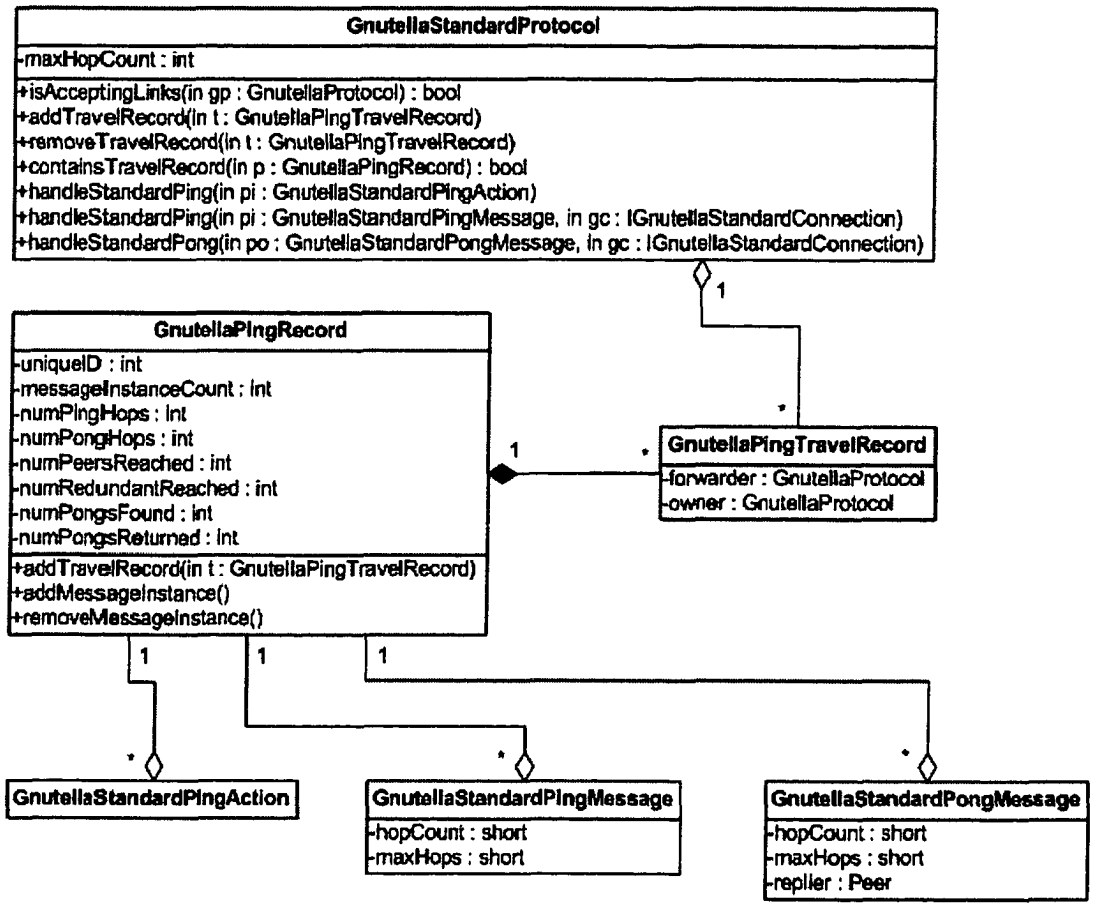

Figure 5.5 Class diagram of Gnutella standard ping/pong related components

The *pong caching* version of Gnutella also uses ping travel records, but only to determine whether or not a pong caching protocol instance has already seen a ping sent by a standard protocol instance. Ping travel records are not required for returning pongs, because pong caching protocol instances do not forward pings to their neighbours; they only return a fixed number of pongs with a good distribution of hop

counts from their pong cache, and possibly a pong from themselves if they are accepting new links.

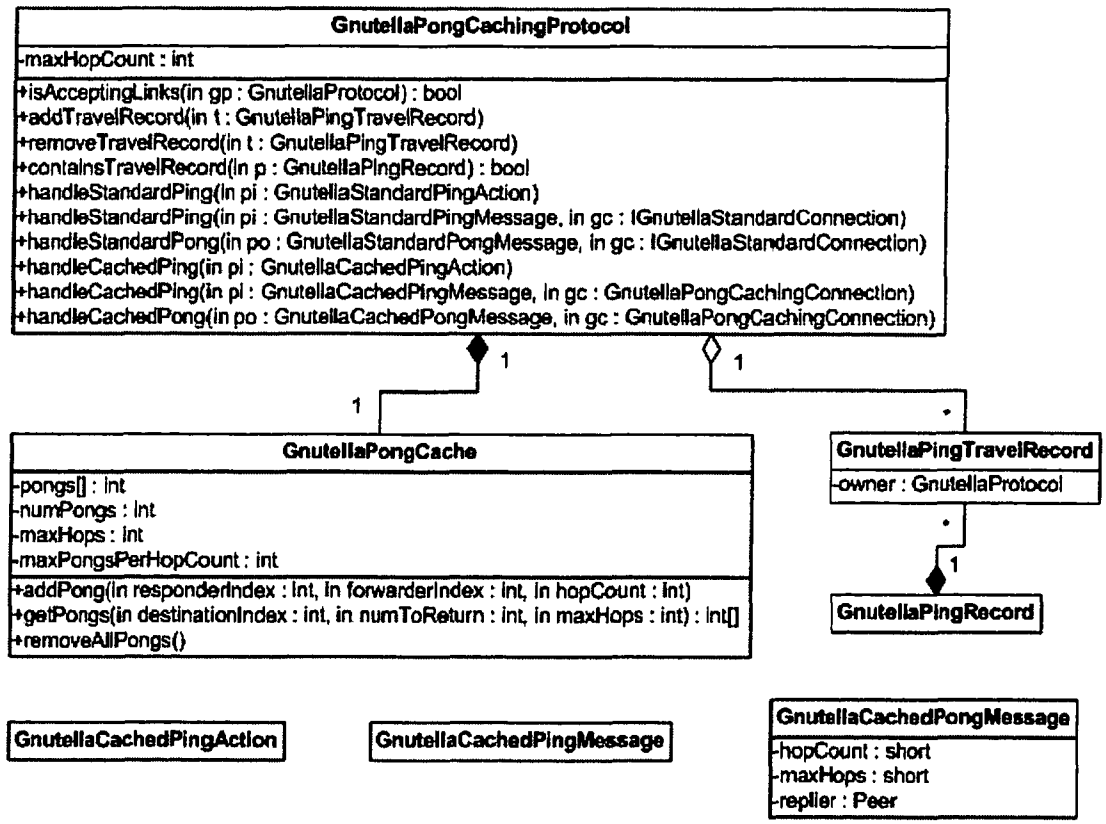

Figure 5.6 Class diagram of Gnutella pong caching ping/pong related components

The pong caching protocol subclass was implemented so as to be backwards compatible with the standard Gnutella protocol subclass. As illustrated by Figure 5.6, the principle difference is that the pong caching protocol subclass stores a reference to a Gnutella pong cache, and implements event-handling methods for cached ping action and message subclasses. Cached ping messages are only ever sent between

pong caching protocol instances, and therefore do not need to store references to a Gnutella ping record or hop count information, thus reducing memory usage.

The pong cache was implemented so that when pongs are inserted, they are grouped by hop count, ensuring that the cache has an even distribution of pongs with all possible hop counts. The pong cache has a fixed size, with a maximum number of pongs per hop count. As new pongs are inserted, the oldest pong with the same hop count is the one that is removed. Each pong stores the index of the peer that responded with the pong, and also the index of the peer that transmitted the pong. This allows for pongs that are retrieved from the cache to be filtered so that redundant pongs are never returned. Redundant pongs for a peer either originated from or were forwarded by that peer. To minimize memory usage, the pong cache was implemented as one long array of integers. The contents of the pong cache are illustrated in Figure 5.7.

Hop Count	Number of Pongs	Oldest Pong Index	Data (Responder Index, Forwarder Index)
1	4	0	3,3 | 4,4 | 5,5 | 2,2 | |
2	6	2	7,4 | 9,2 | 11,5 | 6,4 | 12,3 | 10,3
3	2	0	21,5 | 8,2 | | | |
4	0	0	| | | | |

Figure 5.7 Illustration of the contents of the Gnutella pong cache

The initiation of a ping by a peer is simulated by adding a Gnutella *standard or cached ping action* instance to the simulator. Standard Ping actions require two pieces of information; the identity of the peer who is initiating the ping, and a unique identifier for the ping. The only information required by cached pings is the identity of the peer who is initiating it. The simulator is fed with Gnutella ping actions via the Gnutella standard and cached *ping (initial) action generator* classes, which schedules standard pings for all peers, and cached pings for pong caching peers. Both types of ping actions are scheduled to occur with a delay, determined by a random distribution between 0 and the ping interval specified by each peer. Upon execution, ping actions are re-added to the simulator to be executed again at a time specified by their peer's ping interval.

The Gnutella pong caching protocol class implements two *handle ping action* methods, one which is called by the execute method of standard ping actions, and another which is called by the execute method of cached ping actions. Whenever the standard ping action handling method of a pong caching protocol is called, standard pings are sent out to all of the standard protocol instances in its neighbourhood. Whenever the cached ping handling method is called, the pong cache is emptied, and cached pings are sent to all pong caching instances in the protocol's neighbourhood.

The Gnutella pong caching protocol also implements two *handle ping message* methods, which are called when standard or cached ping messages are successfully

received by a protocol. Upon receiving a ping message (standard pings that have already been seen are ignored) from a requesting protocol, each protocol attempts to retrieve 10 acceptable pongs from its cache and send them back to the requester. Pongs are deemed acceptable if they were not originally sent by the requesting protocol, and if their hop count plus the hop count of the received ping message do not exceed the maximum hop count value specified by the ping message. If there are insufficient acceptable pongs in the cache, the deficit number required is stored in an instance variable of the link upon which the ping was received. For standard pings, the ping record and hop count of the ping message is also stored by the link (See figure 5.9). This provides pong caching protocols with enough information to send pongs back to their pong caching or standard neighbours as their pong cache fills up with new pongs.

The final ping-related methods implemented by Gnutella pong caching protocols are the *handle pong message* methods, which are called when standard or cached pong messages are successfully received by a protocol. When a cached pong message is received by a caching protocol, it is added to the pong cache, and sent to any of the protocol's neighbours that were not sent enough pongs in response to a ping. When a standard pong is received, it is not added to the pong cache; instead it is relegated to a backup list of pongs which may be used by the protocol if it cannot find any caching protocols to connect to. Standard pongs are not added to the pong cache

because it is more desirable to return only cached pongs generated by pong caching protocols. This encourages protocols looking to increase their connectivity to form links with pong caching protocols, thus minimizing ping/pong traffic in the network as a whole.

5.2.3 Connections and Disconnections

In order to test the effects of peer churn on the simulated Gnutella network, the Gnutella plug-in supports the connection or disconnection of a peer.

The Gnutella implementation assumes that each peer / protocol instance is initially provided with links to other protocol instances by a topology generator. The disconnection of a peer is then simulated by adding a Gnutella *disconnect action* instance to the simulator. A disconnect action requires only the identity of a peer which is currently active (See section 3.1). The simulator is fed at regular time intervals with Gnutella disconnect actions via the Gnutella *disconnect (continuous) action generator* (Section 3.4.1.1) class, which can use an arbitrary active peer strategy. For example, the random active peer selection strategy selects random active peers from the peer graph. The average number of disconnect actions to be fed to the simulator for each action generation interval can also be specified, which provides the target rate for a Poisson distribution of disconnections over all generation intervals.

Gnutella protocol classes implement a *handle disconnect action* method, which is called by the execute method of the Gnutella disconnect action class. When this method is called, the protocol instance is deactivated, and Gnutella disconnect messages are sent to all linked protocol instances, prior to removing each link. In addition, the disconnected protocol instance adds references to a connect list for each of the protocol instances to which it was previously linked, or from which it had previously received pongs. This facilitates future re-connection(s).

Gnutella protocol classes implement a *handle disconnect message* method, which is called when a Gnutella disconnect message is successfully received by a protocol instance. The disconnect message serves as a notification that a protocol instance has lost a connection, and that it may wish to form one or more new links with other protocol instances. Whenever a protocol instance wishes to form new links to other protocol instances, it makes use of its connection list. The contents of the connect list is defined as the union of its prior contents with protocol instances corresponding to each pong recently received by the protocol instance. Protocol instances for which links already exist are excluded.

Protocol instances attempt to add new links by sending Gnutella *connect request* messages to protocol instances in the connect list. A protocol instance will sequentially attempt to form new links with other protocol instances until it has reached its maximum number of links, or links have been requested for every protocol

instance in the connect list. When a protocol instance is no longer attempting to add new links, the connect list is cleared to reduce memory consumption.

Gnutella protocol classes implement a *handle connect request* method, which is called by the execute method of the Gnutella connect request class. When this method is called, the protocol instance checks whether or not it is accepting new links. In the affirmative case, a Gnutella *connect accept* message is sent back. In the negative case, a Gnutella *connect reject* message is sent back, along with a number of pongs to aid the requesting protocol instance's quest to forge new links. The *handle connect accept* and *handle connect reject* methods of the Gnutella protocol classes are responsible for either adding a link, or adding the extra pongs to the connect list, when the appropriate message is received.

The connection of a peer is simulated in an identical manner to disconnection, except that instances of the Gnutella *connect action* and *connect (continuous) action generator* are used.

Gnutella protocol classes also implement a *handle connect action* method, which is called by the execute method of the Gnutella connection action class. When this method is called, the protocol instance is activated, and it attempts to add links to other protocol instances, using the connect list it created when it disconnected from the network.

The relationships between the classes of the Gnutella plug-in that are related to connections and disconnections are illustrated by Figure 5.8.

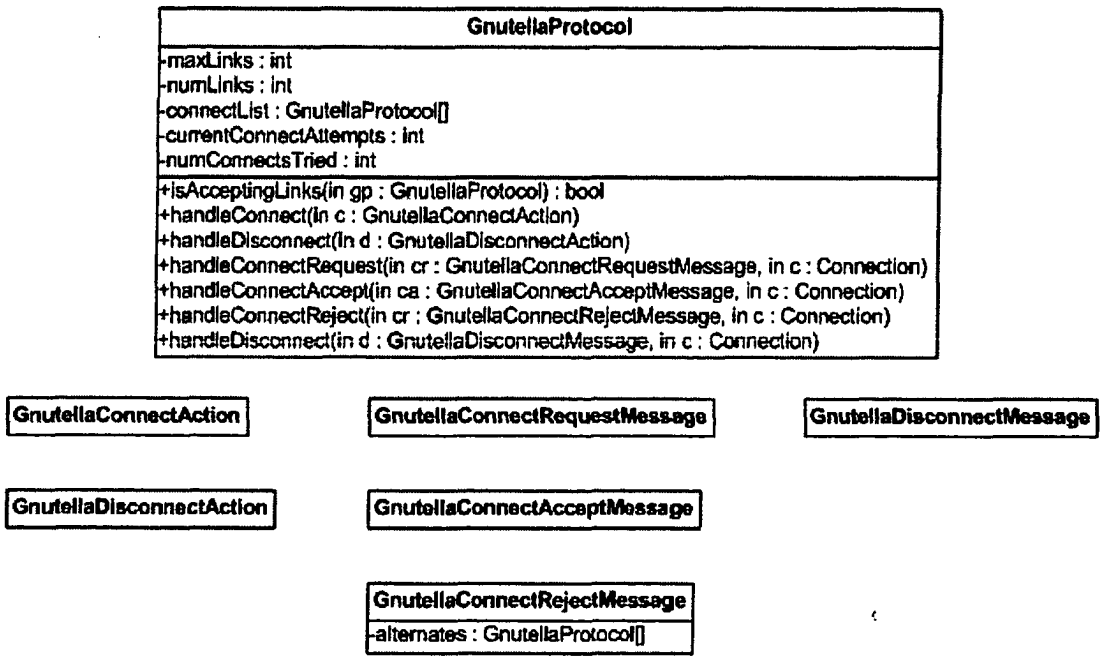

Figure 5.8 Class diagram of Gnutella connect / disconnect related components

5.3 Links

In order to store protocol-level links between peers, Gnutella protocol instances use references to instances of the Gnutella link class. Three different subclasses of the Gnutella link class were created to handle protocol-level links between the two different Gnutella protocol subclasses. The Gnutella standard link class is used to represent links between pairs of Gnutella standard protocol instances, while the Gnutella pong caching link class is used to represent links between pairs of Gnutella

pong caching protocol instances. Similarly, the Gnutella standard and pong caching link class is used to represent links between pairs of standard and pong caching protocol instances. The principle difference between the different subclasses are the fields that they store (see figure 5.9) in order to support the pong caching protocol described in section 5.2.2.

Both the Gnutella standard and Gnutella pong caching protocol subclasses implement methods for the addition or removal of the different subclasses of Gnutella links that they may store. When a link is created between two protocol instances, the Gnutella protocol factory class is responsible for creating an instance of the proper Gnutella link subclass, and adding it to the protocol instances using the appropriate methods. For link removal, a similar procedure is followed by the Gnutella protocol factory class.

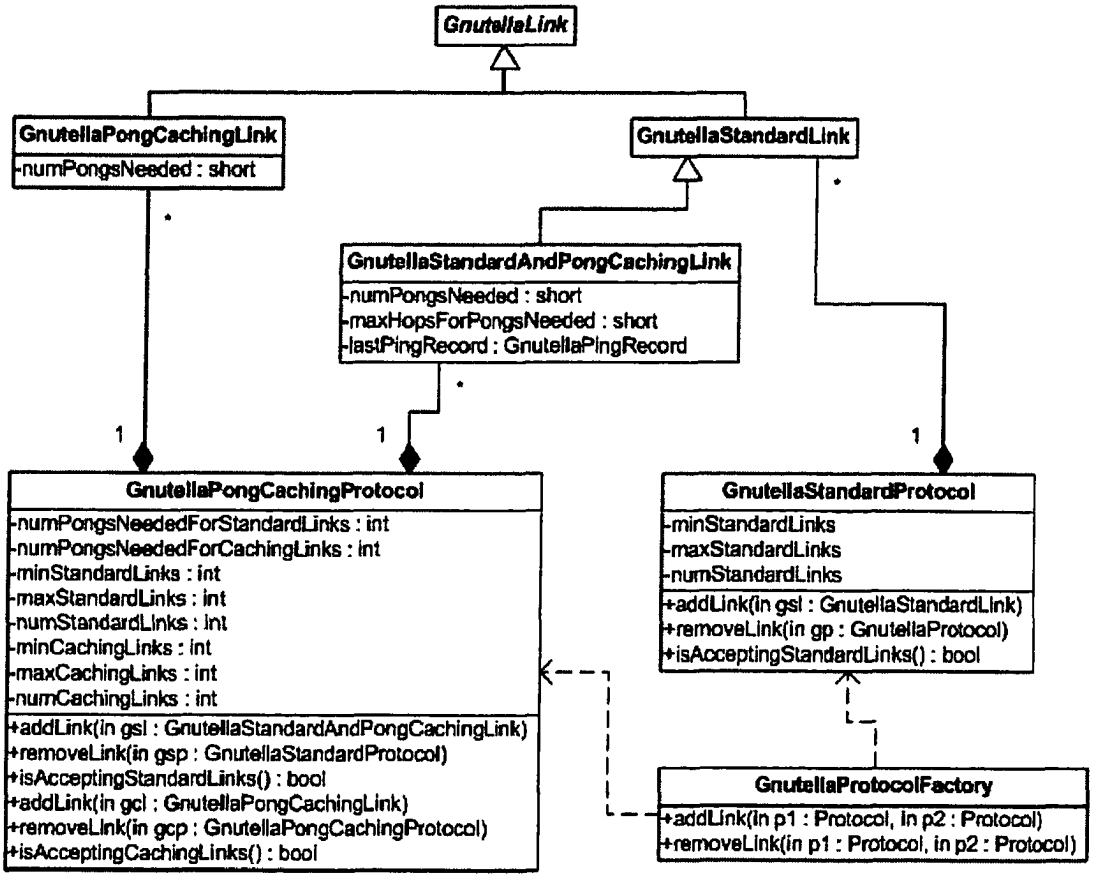

Figure 5.9 Class Diagram of Gnutella link related components

5.3.1 Flow Control

To support flow control, the Gnutella link class stores a fixed-length list of queued messages. The message queue was implemented as several fixed-length arrays, one for each of the Gnutella-specific fields stored by Gnutella messages. This

optimization[16] reduces the memory required to store each queued message, eliminating the overhead incurred by subclassing the message class (See Table 4.1).

The message queue is divided into sub-queues for query reply, query, pong and ping messages. The query reply sub-queue was implemented as an array-based binary heap, keyed by the number of replies previously returned for each query reply. The other sub-queues were implemented as array-based LIFO queues. The maximum total message queue size, and the proportion allocated to each of the sub-queues is determined by the sending protocol instance's ratio values.

To determine whether messages can be transmitted or should be added to the queue, instances of the Gnutella link class need to know if the number of messages in transmission between their source and destination protocol instances has exceeded a pre-determined threshold specified by the source protocol instance. To gain access to the number of messages in transit, the Gnutella link class was implemented as a subclass of the simulator core's connection class.

When using flow control, Gnutella protocol instances wishing to send a message to a protocol instance to which they are linked must first check to see if the appropriate Gnutella link instance can transmit the message. In the affirmative case, the message is added to the simulator using the standard interface described in Section

[16] This optimization is an application of the Flyweight pattern [GHJ+95]

3.4.2. In the negative case, the message is added to the link's queue, using the methods illustrated in Figure 5.10.

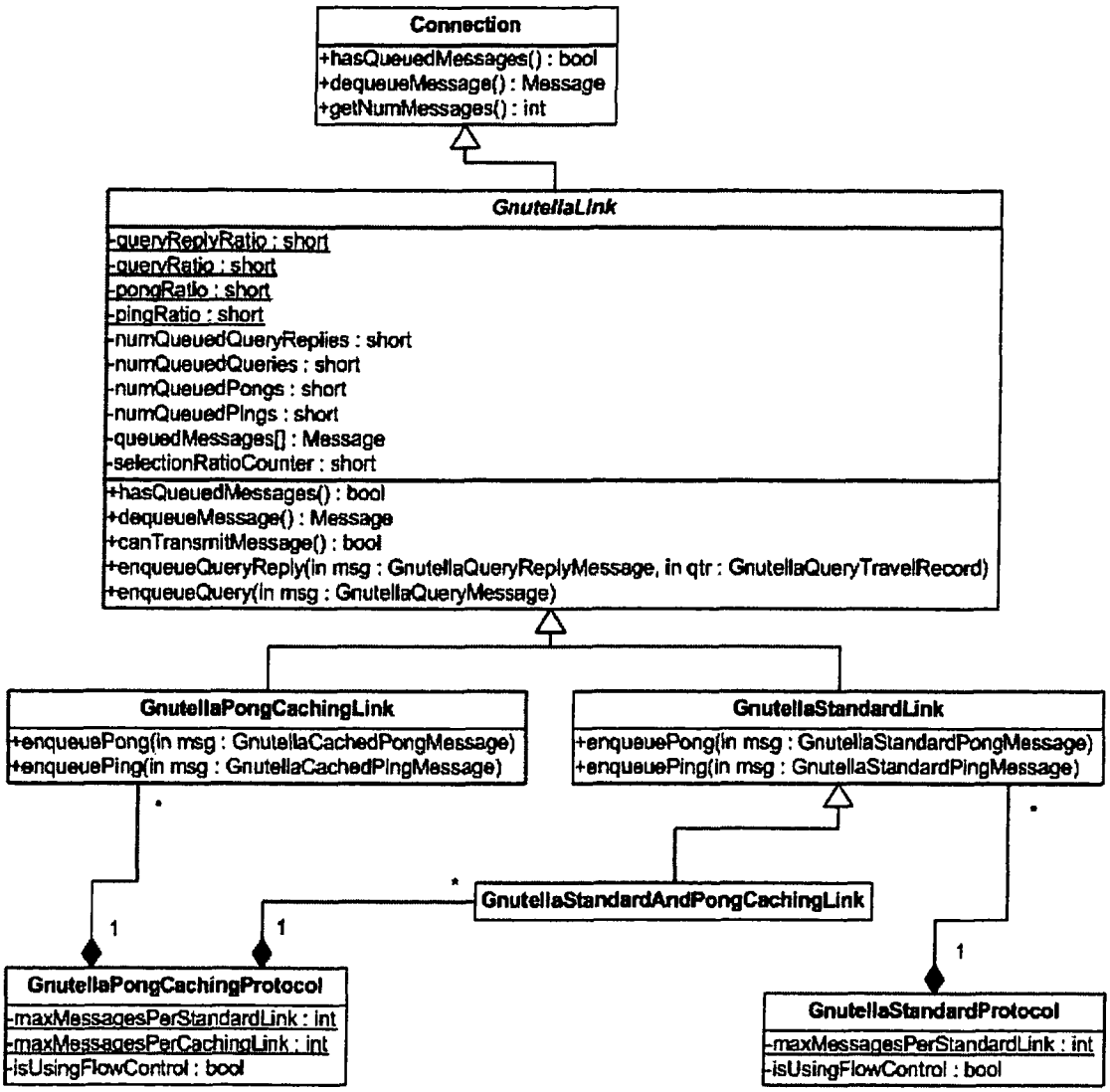

Figure 5.10 Class diagram of Gnutella flow control related components

The Gnutella link class supports the dequeuing of messages by overriding the default methods specified for the connection class. This allows the simulator to automatically dequeue a link's queued messages for transmission as other messages in transmission over the link are removed (See section 3.4.2). The Gnutella link class' dequeue method was implemented so that messages of differing types are removed in round-robin fashion, giving preference to message types with a higher priority/ratio, without neglecting message types with a lower priority/ratio.

5.4 Statistics

To collect and report statistics specific to Gnutella, a Gnutella reporter class was created as a subclass of the core's statistics reporter class (Section 3.5.1). This class binds statistics collectors (Section 3.5.2) to each Gnutella action and message class, and also collects additional statistics not directly associated with actions and messages.

The method of the statistics reporter class that is called whenever a reporting time interval elapses was overridden by the Gnutella statistics reporter to collect the following additional statistics per reporting time interval:

- The number of active queries for which messages are still pending.
- The total number of query travel records being stored by all peers.

- The total number of query replies that reached peers that issued queries.
- The average simulation time taken for all query replies to reach their originators.
- The number of active standard pings for which messages are still pending.
- The total number of ping travel records being stored by all peers.

5.5 Scalability Analysis

One of the principle objectives of this thesis is the development of a peer-to-peer simulator that can simulate large networks. By tabulating the memory of all the component classes implemented for the Gnutella plug-in, it is possible to calculate the total amount of system memory required to simulate Gnutella networks of different sizes running different versions of the protocol.

Table 5.1 displays the memory usage in bytes for each of the components of the Gnutella plug-in. The memory requirements for each component are broken down into the amount required by Gnutella-specific data, and the amount for the extension of a core simulator component, if required. The memory usage of the core simulator components is listed in Table 4.1. For a detailed explanation of how these values were calculated, refer to Appendix B.

Table 5.1 Memory usage in bytes per Gnutella component

Gnutella Component	Memory Usage (bytes)		
	Gnutella	Core	Total
Gnutella Standard Protocol	60	140	200
Gnutella Pong Caching Protocol	92	140	232
Gnutella Pong Cache	36	0	36
Gnutella Pong List	32	0	32
Gnutella Content	24	0	24
Gnutella Standard Link	124	48	172
Gnutella Standard & Pong Caching Link	132	48	180
Gnutella Pong Caching Link	128	48	176
Gnutella Query Record	76	0	76
Gnutella Query Travel Record	32	0	32
Gnutella Ping Record	72	0	72
Gnutella Ping Travel Record	28	0	28
Gnutella Query Action	4	20	24
Gnutella Query Message	8	20	28
Gnutella Query Reply Message	12	20	32
Gnutella Standard Ping Action	4	20	24
Gnutella Standard Ping Message	8	20	28
Gnutella Standard Pong Message	12	20	32
Gnutella Cached Ping Action	0	20	20
Gnutella Cached Ping Message	0	20	20
Gnutella Cached Pong Message	8	20	28

Using the data from table 5.1, it is possible to calculate the total system memory required to simulate Gnutella networks with various sizes and parameters.

Tables 5.2 through 5.5 show the calculations of total system memory required to simulate Gnutella networks with sizes ranging from 100,000 to 1,000,000 peers. The parameters that were used in these calculations are as follows:

- # Standard Peers – The number of peers in the network that store a reference to an instance of the Gnutella Standard Protocol class.
- # Pong Caching Peers – The number of peers in the network that store a reference to an instance of the Gnutella Pong Caching class.
- # Pongs per Pong Cache – The number of pongs in the pong cache stored by instances of the Gnutella pong caching class.
- # Pongs per Reserve List – The number of pongs stored in the reserve pong list stored by all Gnutella protocol instances.
- # Actions per Peer – The average number of actions stored by each peer, scheduled for future execution.
- # Content per Peer – The average number of Gnutella Content instances stored by each peer.
- # Standard Connections – The total number of Gnutella Standard or Gnutella Standard and Pong Caching Link instances stored by all Gnutella protocol instances. The actual number of link class instances is half of this value, since each link is stored by the protocol instance on either end.
- # Pong Caching Connections – The total number of Gnutella Pong Caching Links instances stored by all Gnutella Pong Caching protocol instances. Again, the actual number of link class instances is half of this value.

- # Messages per Connection – The maximum number of messages in transmission over each of the links.
- # Queued Messages per Connection – The maximum number of messages queued for transmission on each link. A value is specified for each of the possible types of messages. Because arrays are used to store the queues, the memory usage is the same whether the queues are full or empty. The storage required per message is limited to the Gnutella-data listed in Table 5.1.
- # Active Queries – The number of queries that are active in the Gnutella network. An active query is one for which there are messages still pending on links.
- # Query Travel Records per Query – The average number of query travel records stored by Gnutella protocol instances, as used for routing an active query's messages.
- # Active Standard Pings - The number of standard pings that are active in the Gnutella network. An active standard ping is one for which there are messages still pending on links.
- # Ping Travel Records per Standard Ping - The average number of ping travel records stored by Gnutella protocol instances, as used for routing an active standard ping's messages.

Table 5.2 Memory for network with 100,000 peers and 8 connections per peer

Parameter	Value	Memory Usage (Gigabytes)	Percentage
# Standard Peers	20,000	0.004	0.44%
# Pong Caching Peers	80,000	0.017	2.05%
# Pongs per Pong Cache	35	0.021	2.54%
# Pongs per Reserve List	20	0.015	1.76%
# Actions per Peer	2	0.004	0.53%
# Content per Peer	10	0.022	2.65%
# Standard Connections	160,000	0.013	1.55%
# Pong Caching Connections	640,000	0.052	6.21%
# Messages per Connection	10	0.209	24.69%
# Queued Query Replies per Connection	6	0.054	6.35%
# Queued Queries per Connection	5	0.030	3.53%
# Queued Pongs per Connection	4	0.026	3.10%
# Queued Pings per Connection	3	0.006	0.71%
# Active Queries (% of peers)	2	0.000	0.02%
# Query Travel Records per Query	6,000	0.358	42.33%
# Active Standard Pings (% of Standard peers)	5	0.000	0.01%
# Ping Travel Records per Standard Ping	500	0.013	1.54%
Total		0.845	100.00%

Table 5.2 illustrates that a Gnutella network with 100,000 peers can be simulated on a conventional desktop machine with 1 Gigabyte of system memory. The calculations for this chart assume that each peer has on average at most 8 links or connections with other peers, and each of these connections is in use for the communication of at most 18 messages. With these parameters, approximately 46 percent of the memory used is dedicated to connections and messages between peers. The biggest consumer of system memory for this simulation would be the query travel records used by Gnutella protocol instances to filter out redundant query messages and route query replies back to the query's originator.

Table 5.3 Memory required for network with 100,000 peers and 10 connections per peer

Parameter	Value	Memory Usage (Gigabytes)	Percentage
# Standard Peers	20,000	0.004	0.37%
# Pong Caching Peers	80,000	0.017	1.74%
# Pongs per Pong Cache	35	0.021	2.16%
# Pongs per Reserve List	20	0.015	1.50%
# Actions per Peer	2	0.004	0.45%
# Content per Peer	10	0.022	2.25%
# Standard Connections	200,000	0.016	1.65%
# Pong Caching Connections	800,000	0.066	6.59%
# Messages per Connection	12	0.313	31.47%
# Queued Query Replies per Connection	6	0.067	6.74%
# Queued Queries per Connection	5	0.037	3.75%
# Queued Pongs per Connection	4	0.033	3.30%
# Queued Pings per Connection	3	0.007	0.75%
# Active Queries (% of peers)	2	0.000	0.01%
# Query Travel Records per Query	6,000	0.358	35.96%
# Active Standard Pings (% of Standard peers)	5	0.000	0.01%
# Ping Travel Records per Standard Ping	500	0.013	1.31%
Total		0.994	100.00%

Table 5.3 illustrates the effect of increasing the average number of connections per peer, and the number of messages per connection relative to the parameters used for Table 5.2. Assuming that the number of peers reached by queries does not increase as the number of connections increases, the relative importance of the memory used for query travel records is reduced. With a conventional desktop machine, it is still possible to simulate a network of this size.

Table 5.4 Memory required for network with 100,000 peers without pong caching

Parameter	Value	Memory Usage (Gigabytes)	Percentage

# Standard Peers	100,000	0.019	0.53%
# Pong Caching Peers	0	0.000	0.00%
# Pongs per Pong Cache	35	0.000	0.00%
# Pongs per Reserve List	20	0.015	0.43%
# Actions per Peer	2	0.004	0.13%
# Content per Peer	10	0.022	0.64%
# Standard Connections	1,000,000	0.082	2.34%
# Pong Caching Connections	0	0.000	0.00%
# Messages per Connection	12	0.313	8.95%
# Queued Query Replies per Connection	6	0.067	1.92%
# Queued Queries per Connection	5	0.037	1.07%
# Queued Pongs per Connection	4	0.045	1.28%
# Queued Pings per Connection	3	0.022	0.64%
# Active Queries (% of peers)	5	0.000	0.01%
# Query Travel Records per Query	7,000	1.043	29.84%
# Active Standard Pings (% of Standard peers)	10	0.001	0.02%
# Ping Travel Records per Standard Ping	7,000	1.825	52.21%
Total		3.496	100.00%

By contrast, Table 5.4 illustrates the inherent difficulties involved in simulating a pure Gnutella network without pong-caching. In this scenario, every peer is repeatedly flooding the network with standard ping messages, requiring extensive system memory to store the ping travel records that are distributed peers along the ping's path. More than 50 percent of system memory is consumed by ping travel records, and the total memory requirement increases almost fourfold.

Table 5.5 Memory required for network with 1,000,000 peers and 10 connections per peer

Parameter	Value	Memory Usage (Gigabytes)	Percentage
# Standard Peers	200,000	0.037	0.25%
# Pong Caching Peers	800,000	0.173	1.17%
# Pongs per Pong Cache	35	0.215	1.45%

# Pongs per Reserve List	20	0.149	1.01%
# Actions per Peer	2	0.045	0.30%
# Content per Peer	10	0.224	1.51%
# Standard Connections	2,000,000	0.164	1.11%
# Pong Caching Connections	8,000,000	0.656	4.43%
# Messages per Connection	10	2.608	17.63%
# Queued Query Replies per Connection	6	0.671	4.53%
# Queued Queries per Connection	5	0.373	2.52%
# Queued Pongs per Connection	4	0.328	2.22%
# Queued Pings per Connection	3	0.075	0.50%
# Active Queries (% of peers)	3	0.002	0.01%
# Query Travel Records per Query	10,000	8.941	60.46%
# Active Standard Pings (% of Standard peers)	5	0.001	0.00%
# Ping Travel Records per Standard Ping	500	0.130	0.88%
Total		14.788	100.00%

Thus, to estimate the memory requirements for the simulation of a Gnutella network of 1 million peers, it is important to carefully select the appropriate parameters. Table 5.5 illustrates the total amount of system memory required to simulate a Gnutella network of 1 million peers with the use of pong caching and flow control. Equally, the simulation is limited to a maximum of 180 million concurrent messages, and a maximum of 30,000 concurrently active queries. Simulations of the network can then be achieved on a machine equipped with 16 Gigabytes of system memory, which is not unreasonable by today's standards.

5.6 Summary

This chapter has described the implementation of the Gnutella plug-in used to showcase the extensibility of the simulator core, and to test its scalability. The chapter

began with a description of the Gnutella protocol and two of its optimizations: pong caching and flow control. Section 5.1 then gave a brief description of how distributed content was stored by peers running the Gnutella protocol. Section 5.2 described how the simulator core's protocol, action and message classes were extended to handle Gnutella query, ping and connection related messages. The simulation of peers with a heterogeneous distribution of protocols was also show-cased via the standard and pong caching protocol subclasses. Next, Section 5.3 described how the simulator core's connection class was extended to represent protocol-level links between peers running the Gnutella protocol, and to provide an implementation of flow control. Section 5.4 discussed how the simulator core's statistics collection and reporting functionality was extended to collect Gnutella-specific statistics. Finally, Section 5.5 summarized the memory overhead incurred by the Gnutella implementation, and provided estimates of the total system memory required to simulate Gnutella networks of varying size and with various parameters.

The results and analysis obtained by performing simulation runs with the Gnutella plug-in are provided in Chapter 6.

Chapter 6

Results & Analysis

The functionality, scalability and extensibility of the simulator core were validated by performing simulation runs with the Gnutella plug-in described in Chapter 5. The Gnutella plug-in was utilized by the simulator in a variety of scenarios, each with differing network and protocol-specific parameters. This chapter describes the results of these simulations.

The primary function of these runs is to establish that the simulator, in its principal features, does succeed in reproducing the behaviour of real peer-to-peer networks. Section 6.1 illustrates the effects on message traffic when simulating different proportions of peers with high and low maximum bandwidths. Section 6.2 focuses on the effects on message traffic of simulations with different proportions of pong caching and standard Gnutella peers. Section 6.3 illustrates the effects on the network as the amount of query related message traffic increases or decreases. Section 6.4 illustrates the effects of flow control on message traffic, reinforcing its importance for managing the number of simulated messages. Section 6.5 shows how simulation

runs can be greatly affected by differing protocol-level topologies, and illustrates that the simulator can handle changing topologies. Finally, in Section 6.6 the results of simulation runs on large topologies of 100,000 peers are presented.

6.1 High vs. Low Bandwidth Distributions

As described in section 3.4.4, the simulator supports the distribution of varying maximum upload and download bandwidths to the peers in the simulated network. The purpose of this section is to verify that the MES bandwidth model employed by the simulator is able to model the increased message transmission delays as available bandwidth decreases. To this end, statistics were obtained from three separate simulation runs of a Gnutella network with varying proportions of peers with high and low upload and or download bandwidth allocations.

Table 6.1 lists the *common* parameters used for all of these simulation runs, and Table 6.2 lists the *different* parameters used for each of them. For an explanation of the parameters and a description of how they were provided to the simulator, refer to Appendix C.

Table 6.1 Common Parameters Used for High vs. Low Bandwidth Distributions Simulation Runs

Parameter	Value
Random Number Seed	51957253545678
Time to Stop (Simulated Time Units)	100.0
Number of Peers	10,000
Average Number of Links per Peer	12
Topological Structure of Inter-Peer Links	BRITE Waxman (Random)
Number of Content Records in Repository	5000
Average Number of Content Records per Peer	10
Percentage of Pong Caching Gnutella Peers	95
Percentage of Standard Gnutella Peers	5
Max Pongs per Hop Count in Pong Cache	5
Cached Pongs Returned per Ping	10
Cached Ping Interval (Simulated Time Units)	4.0
Standard Ping Interval (Simulated Time Units)	40.0
Max Hop Count for Queries and Standard Pings	7
Percentage of Peers Using Flow Control	100
Max Messages per Connection	12
Max Queued Query Replies per Connection	6
Max Queued Queries per Connection	5
Max Queued Pongs per Connection	4
Max Queued Pings per Connection	3
Query Generation Interval (Simulated Time Units)	5.0
Query Generation Rate (Poisson) per Generation Interval	10

Table 6.2 Different Parameters Used for High vs. Low Bandwidth Distributions Simulation Runs

Parameter	Simulation Run 1	Simulation Run 2	Simulation Run 3
Maximum Upload / Download Bandwidth per Peer (bytes per Simulated Time Unit)	100 % 10k / 10k	50 % 10k / 10k 50 % 4k / 10k	33 % 10k / 10k 33 % 4k / 10k 33 % 400 / 400

Table 6.2 shows how the maximum bandwidth available to peers was gradually decreased for each of the simulation runs. In the first of these simulation runs, all peers were assigned uniform maximum upload and download bandwidths. In

the second run, the maximum upload bandwidth was decreased by 60 percent for 50 percent of the peers. Finally, in the third run, the maximum upload and download bandwidth of a third of the peers was decreased well below the previous values. To highlight the effects of different bandwidth distributions on the simulated Gnutella network, several statistics were chosen for graphical display. Figure 6.1 illustrates the decrease in the average amount of bandwidth allocated to messages per peer as the proportion of lower bandwidth peers increases. Figure 6.2 illustrates how this decrease in allocated bandwidth corresponds to more in-transit messages accumulating for each peer. Finally, Figure 6.3 shows the significant effects of reduced bandwidth on query response times.

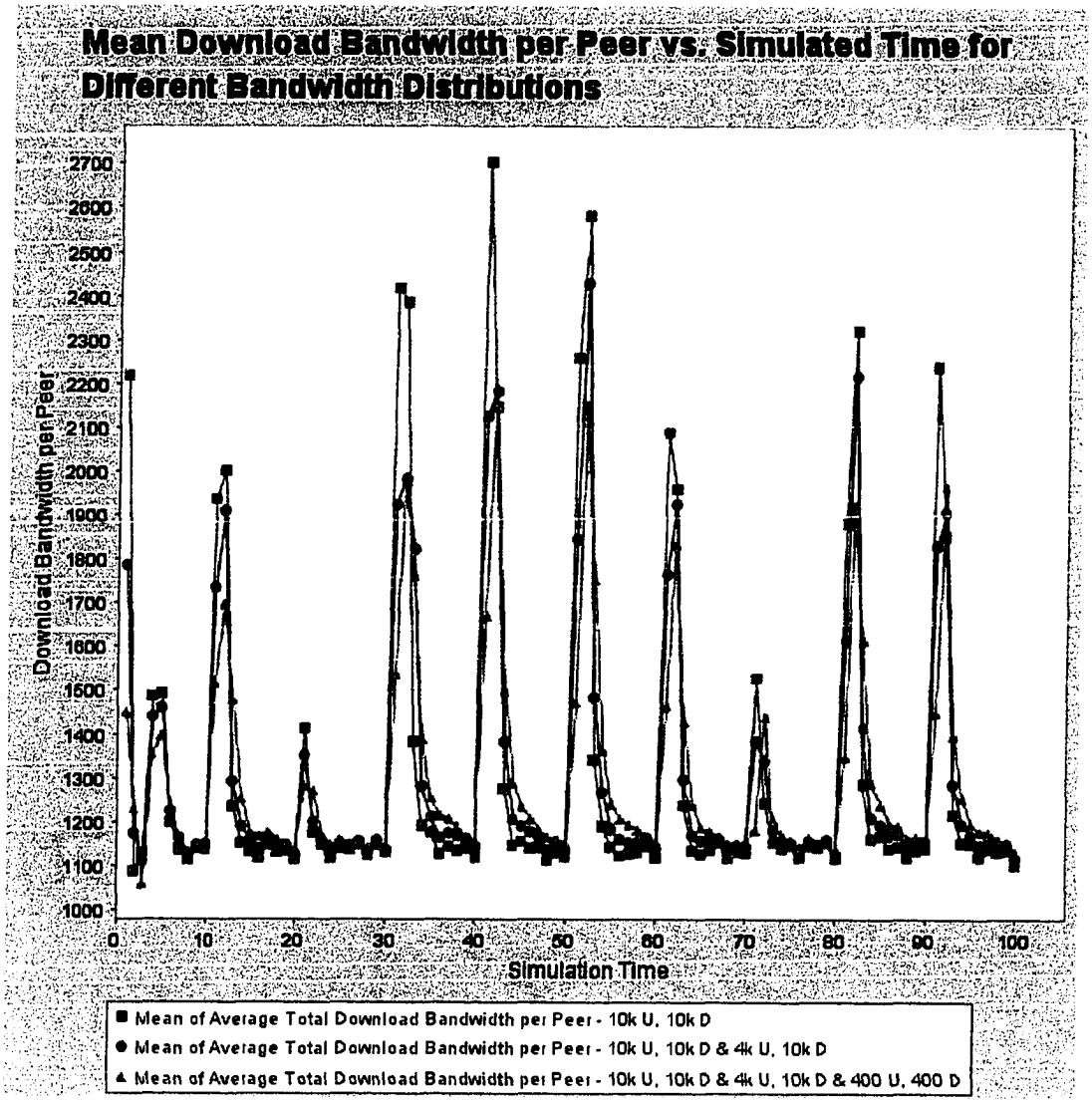

Figure 6.1 Mean Bandwidth Used per Peer for Different Bandwidth Distributions

Figure 6.1 illustrates the effects on bandwidth used by individual peers as the total amount of bandwidth available to the network decreases. When more bandwidth is available, peers take advantage of this bandwidth to transmit messages faster. When less bandwidth is available, peers cannot assign as much bandwidth for message

transmission, which results in messages taking longer to transmit. This can be observed in the figure, where, on average, peers in the networks with lower total available bandwidth are seen to consume bandwidth for longer time intervals.

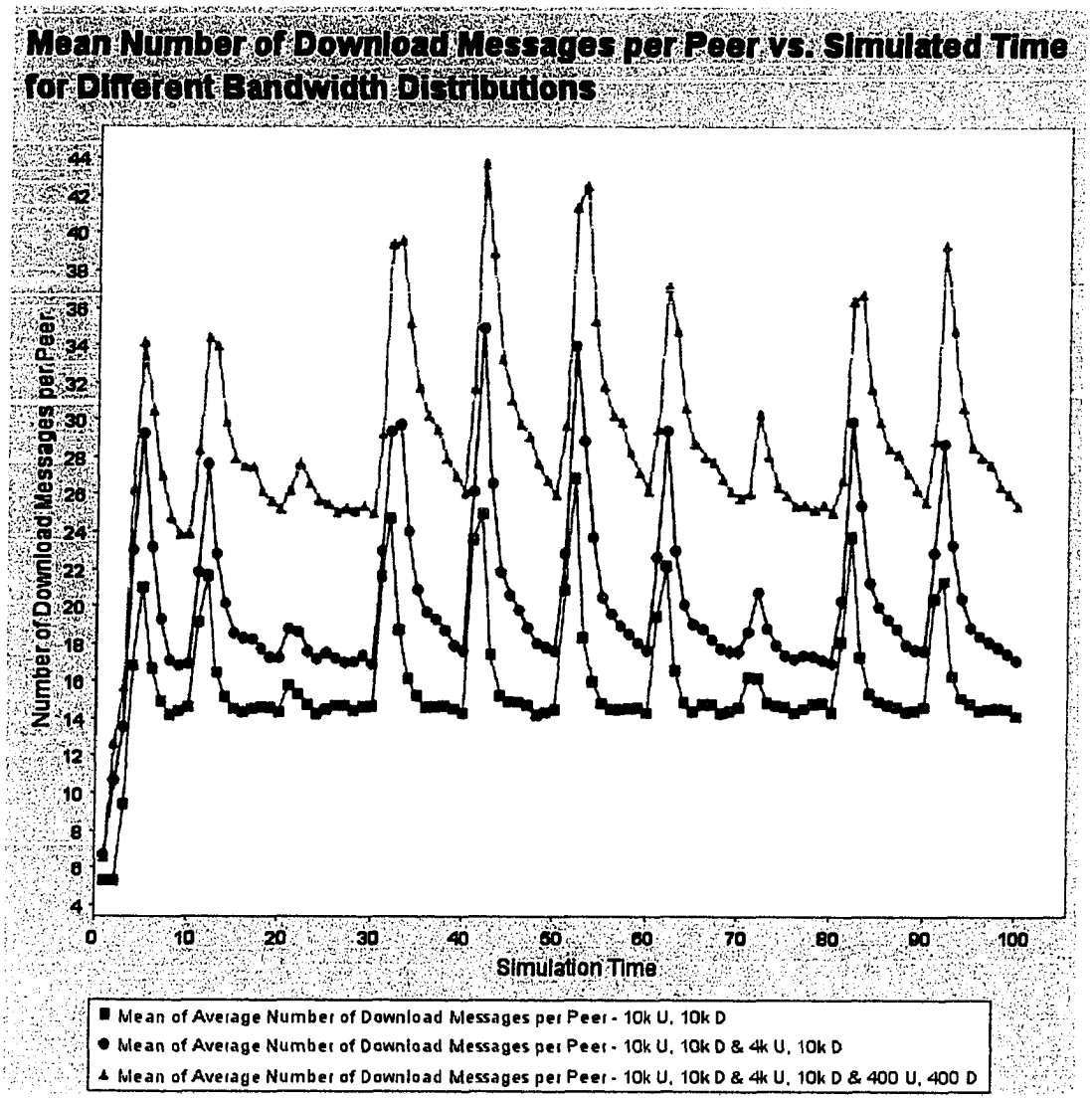

Figure 6.2 Mean Number of Messages per Peer for Different Bandwidth Distributions

Figure 6.2 shows statistics which confirm that peers in networks with lower total available bandwidth tend to take longer to transmit messages. As the amount of bandwidth available to the network decreases, the average number of concurrent message transmissions to or from peers increases.

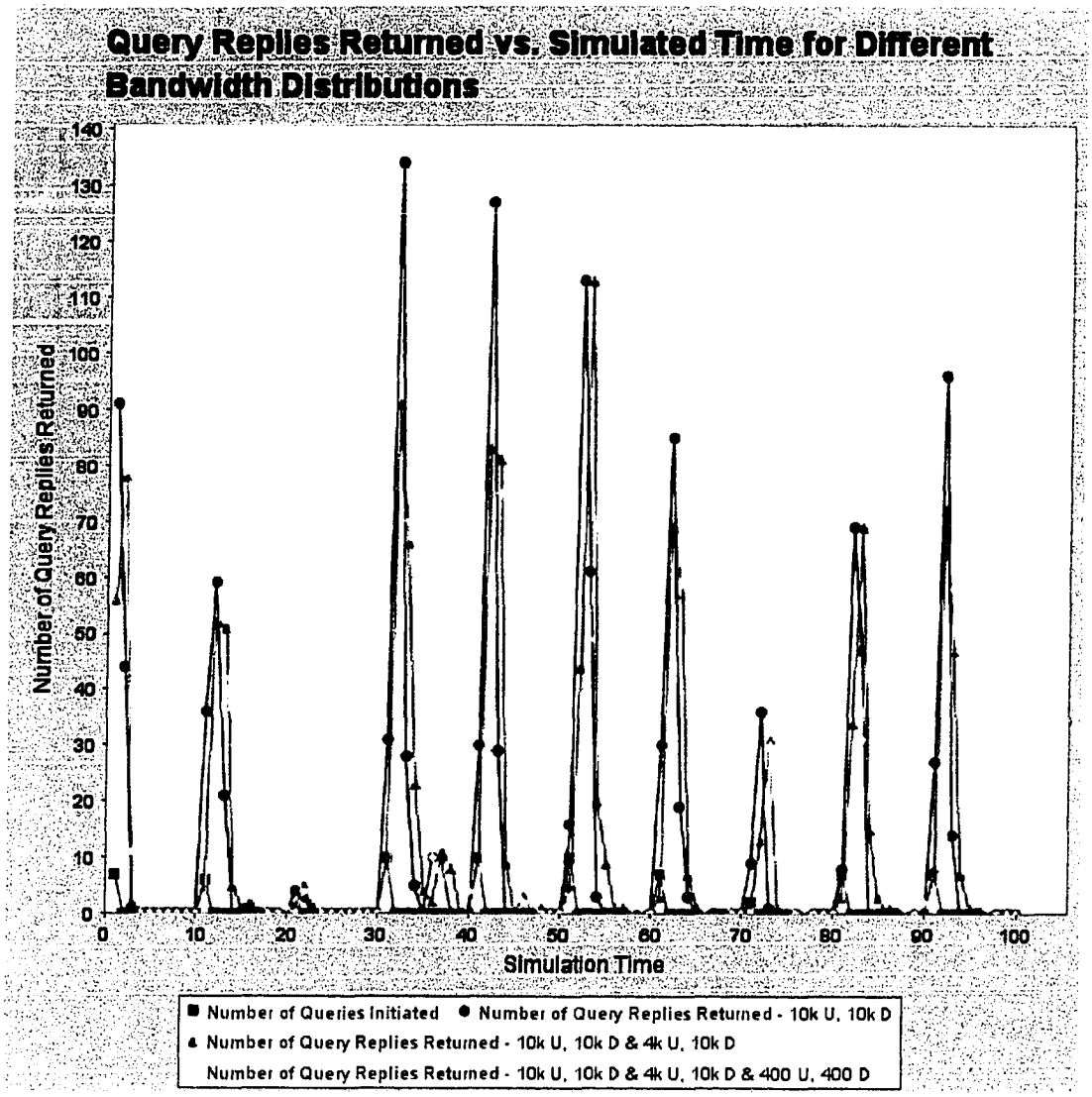

Figure 6.3 Number of Query Replies Returned for Different Bandwidth Distributions

Figure 6.3 shows the effects of decreased total bandwidth on the rate at which query replies are returned in response to queries. In networks with higher available bandwidth, more query replies are returned in the time intervals immediately after the corresponding query is issued. Conversely, in networks with lower available bandwidth, more query replies are returned in later time intervals.

Thus, the statistics presented in Figures 6.1 to 6.3 illustrate that the MES bandwidth model performs well at modeling the slow-down in message transmission times as the amount of available bandwidth decreases. Correspondingly, with this model, it would be expected that increasing the amount of simultaneous message traffic on the network would have a similar effect to decreasing the amount of available bandwidth. This is confirmed by the results obtained in Sections 6.2 and 6.3, where simulation runs are performed with varying message traffic densities.

6.2 Standard vs. Pong Caching

As described in the introduction to Chapter 5 and in Section 5.2.2, one of the enhancements made to the Gnutella protocol was the introduction of Pong Caching. The intention of this enhancement was to reduce the burden of ping and pong message traffic, and in a network formed entirely of Pong Caching peers, the advantages are clear. What is less clear, however, is how the presence of Standard peers in a network of Pong Caching peers affects network performance. To investigate this aspect,

statistics were obtained from three separate simulation runs of a Gnutella network with varying proportions of Standard and Pong Caching peers.

Table 6.3 lists the *common* parameters used for all of these simulation runs, and Table 6.4 lists the *different* parameters used for each of them. For an explanation of the parameters and a description of how they were provided to the simulator, refer to Appendix C.

Table 6.3 Common Parameters Used for Standard vs. Pong Caching Simulation Runs

Parameter	Value
Random Number Seed	51957253545678
Time to Stop (Simulated Time Units)	100.0
Number of Peers	10,000
Average Number of Links per Peer	12
Topological Structure of Inter-Peer Links	BRITE Waxman (Random)
Maximum Upload Bandwidth per Peer (bytes per Simulated Time Unit)	10,000
Maximum Download Bandwidth per Peer (bytes per Simulated Time Unit)	10,000
Number of Content Records in Repository	5000
Average Number of Content Records per Peer	10
Max Pongs per Hop Count in Pong Cache	5
Cached Pongs Returned per Ping	10
Cached Ping Interval (Simulated Time Units)	4.0
Standard Ping Interval (Simulated Time Units)	40.0
Max Hop Count for Queries and Standard Pings	7
Percentage of Peers Using Flow Control	100
Max Messages per Connection	12
Max Queued Query Replies per Connection	6
Max Queued Queries per Connection	5
Max Queued Pongs per Connection	4
Max Queued Pings per Connection	3
Query Generation Interval (Simulated Time Units)	5.0
Query Generation Rate (Poisson) per Interval	10

Table 6.4 Different Parameters Used for Standard vs. Pong Caching Simulation Runs

Parameter	Simulation Run 4	Simulation Run 5	Simulation Run 6
Percentage of Pong Caching Gnutella Peers	100	90	80
Percentage of Standard Gnutella Peers	0	10	20

As shown in Table 6.4, the proportion of standard Gnutella peers was varied from zero to twenty percent of the total network population. The upper limit of twenty percent was chosen because it provides sufficient information to deduce the trend in performance degradation, while allowing the simulator to run in a reasonable time frame. To highlight the effects of different proportions of Standard vs. Pong Caching peers on the simulated Gnutella network, several statistics were chosen for graphical display. Figures 6.4 through 6.6 show for each simulation run, the mean of the bandwidth used per peer for each of the different messages types. Figures 6.7 and 6.8 show comparisons over all simulation runs of the number of messages in transmission per peer, both for all messages, and for query related messages. Finally, Figure 6.9 illustrates the effects of the different proportions on query response time.

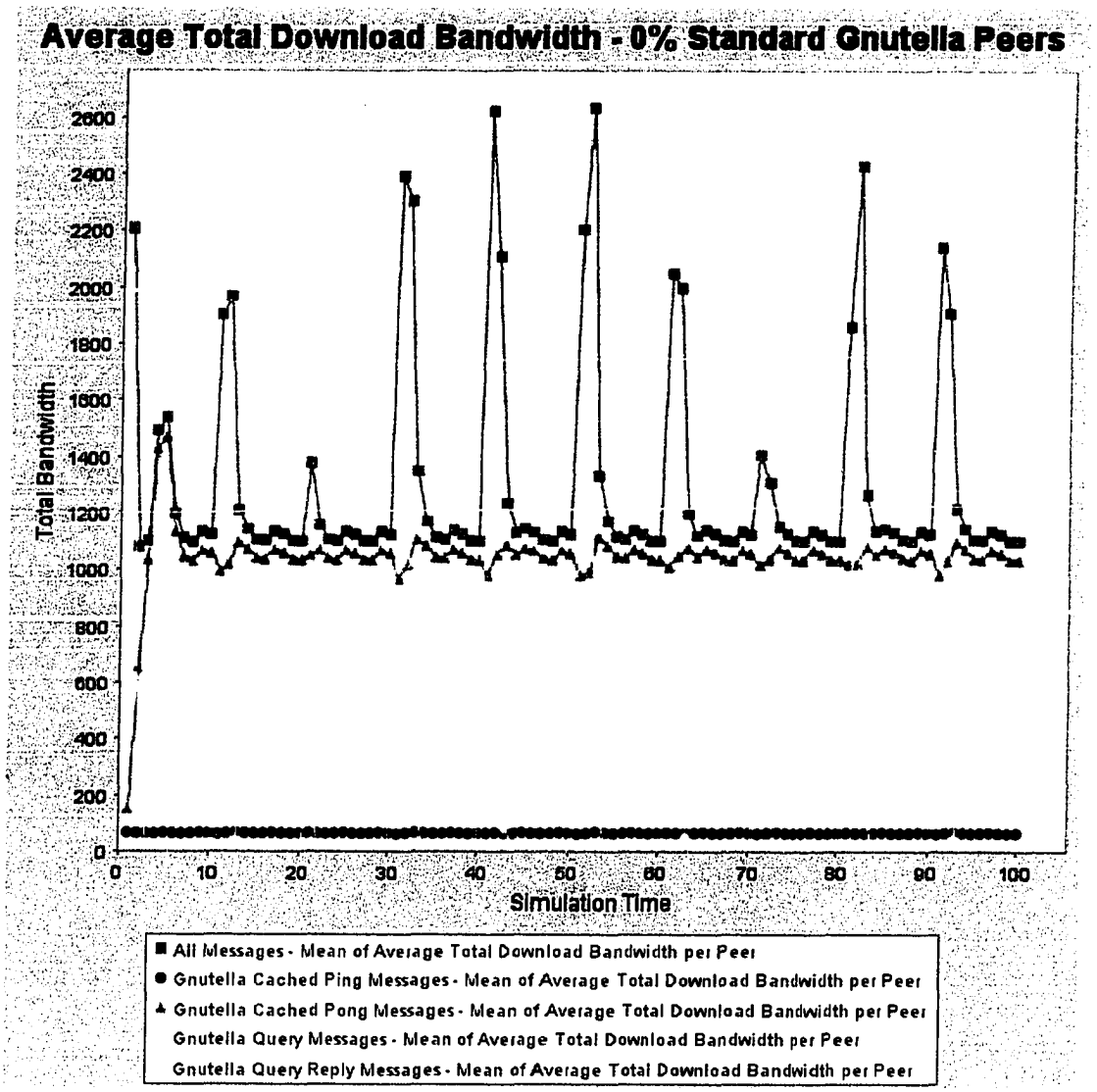

Figure 6.4 Average Bandwidth Used per Peer in a Gnutella Network with 0% Standard Peers

Figure 6.4 shows queries competing with cached pongs for bandwidth. Cached pongs require on average 1000-1100 units of bandwidth per peer, fluctuating very little in demand as the network evolves with time. By contrast, the queries, which are sent out each 10 simulated time units, cause the spikes of around 1000 units which are

clearly visible. This figure serves as a standard against which the subsequent data, shown in Figures 6.5 and 6.6 should be compared.

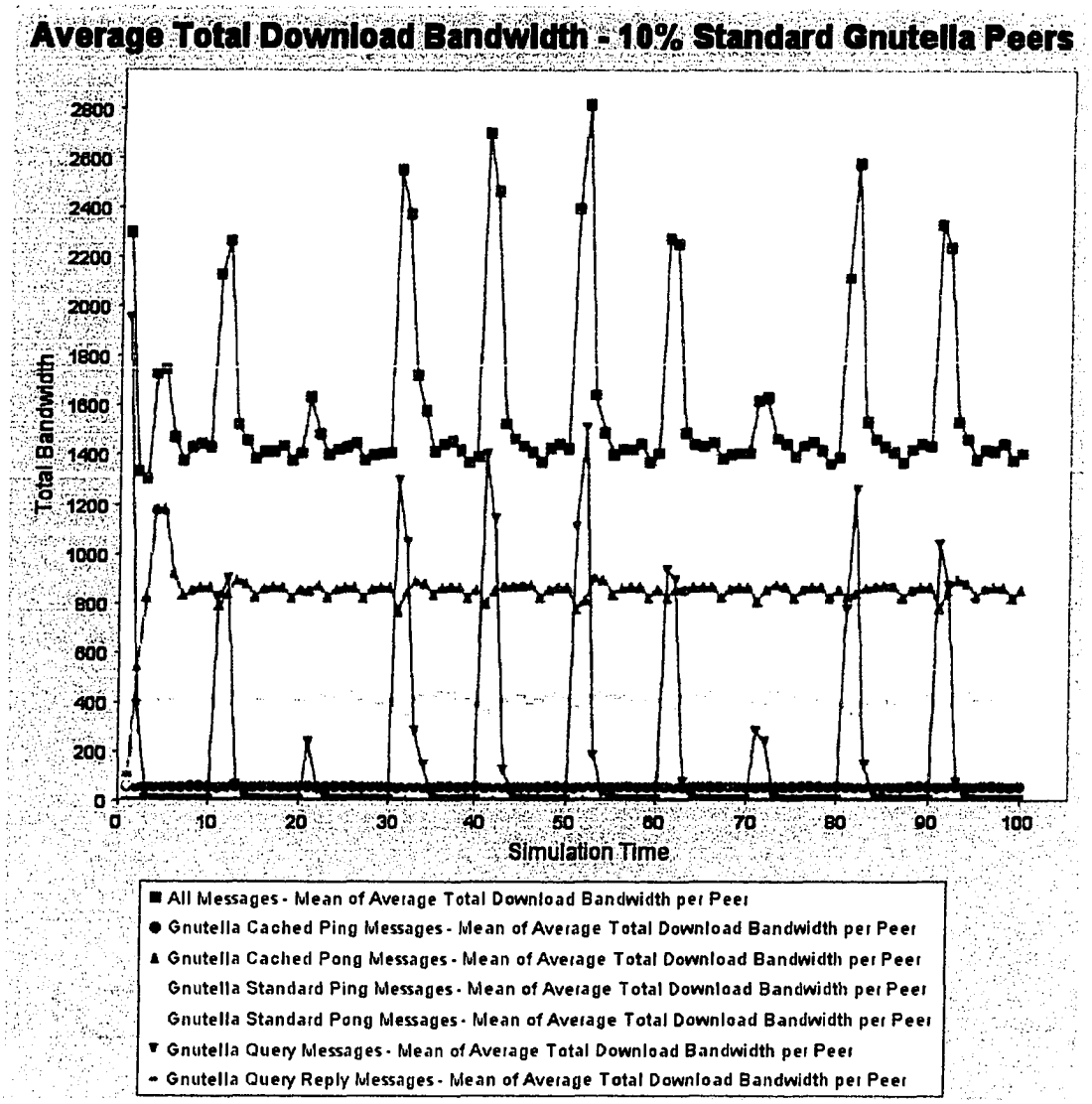

Figure 6.5 Average Bandwidth Used per Peer in a Gnutella Network with 10% Standard Peers

Figure 6.5 shows data for a network with 10% standard Gnutella peers. As expected, it shows that total bandwidth requirements on peers in the network are

marginally higher, since queries are competing with cached pongs and standard pongs for bandwidth. Cached pongs require on average 800-900 units of bandwidth per peer, compared with the previous 1000-1100, which makes sense since there are 10 percent less pong caching peers than before. The biggest factor in the change of total bandwidth requirements for this network is the introduction of standard pong messages. These messages require 50 percent of the bandwidth of cached pongs. This occurs even though the caching peers, which are 9 times more numerous, are issuing cached pings at 10 times the rate of standard pings.

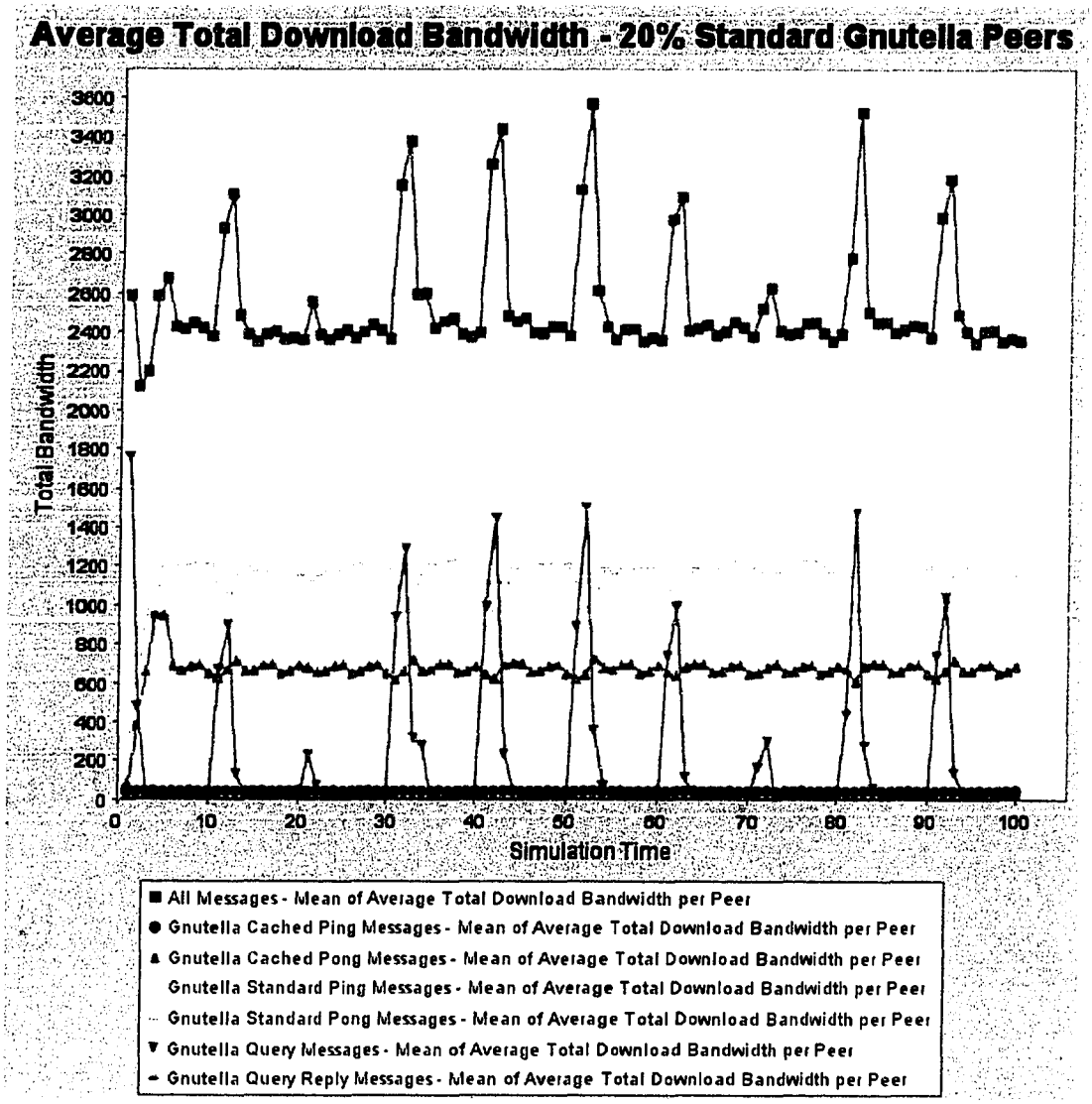

Figure 6.6 Average Bandwidth Used per Peer in a Gnutella Network with 20% Standard Peers

Figure 6.6, with 20% standard peers, shows that the total bandwidth used by peers in the network is substantially higher than before. This is because queries are competing for bandwidth not only with cached pongs but also with a significant number of standard pongs whose effect is predominant. Thus, whereas cached pongs

require on average 700 units of bandwidth per peer, since there are again 10 percent less pong caching peers, the standard pongs now require twice the bandwidth of cached pongs. This occurs, even though the caching peers are 5 times more numerous, and issue cached pings at 10 times the rate of standard pings.

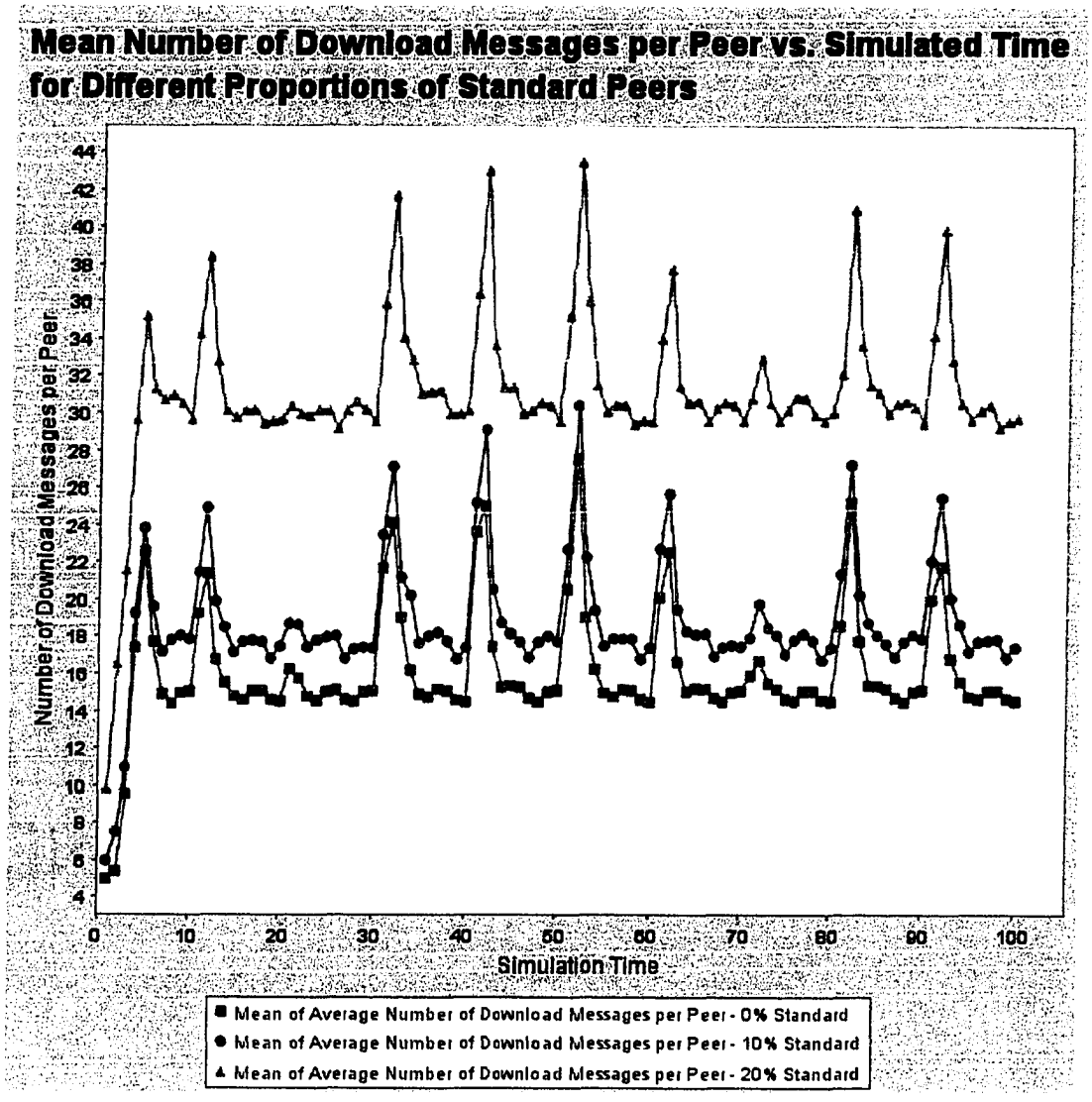

Figure 6.7 Mean Number of Download Messages per Peer for Different Proportions of Standard Peers

Figure 6.7 compares the number of messages being transmitted to peers for all three runs, averaged over each simulated time interval. Higher values of these statistics imply either quick and frequent transmissions, or slow and steady

transmissions. The upper three curves in the figure show that the average number of messages simultaneously in transmission to peers increases significantly as the number of standard peers increases. This increase is significantly larger as the proportion of standard peers increases from 10 to 20 percent. The likely cause of this effect is that more standard peers are directly linked to each other, which results in a larger number of pongs being routed amongst them.

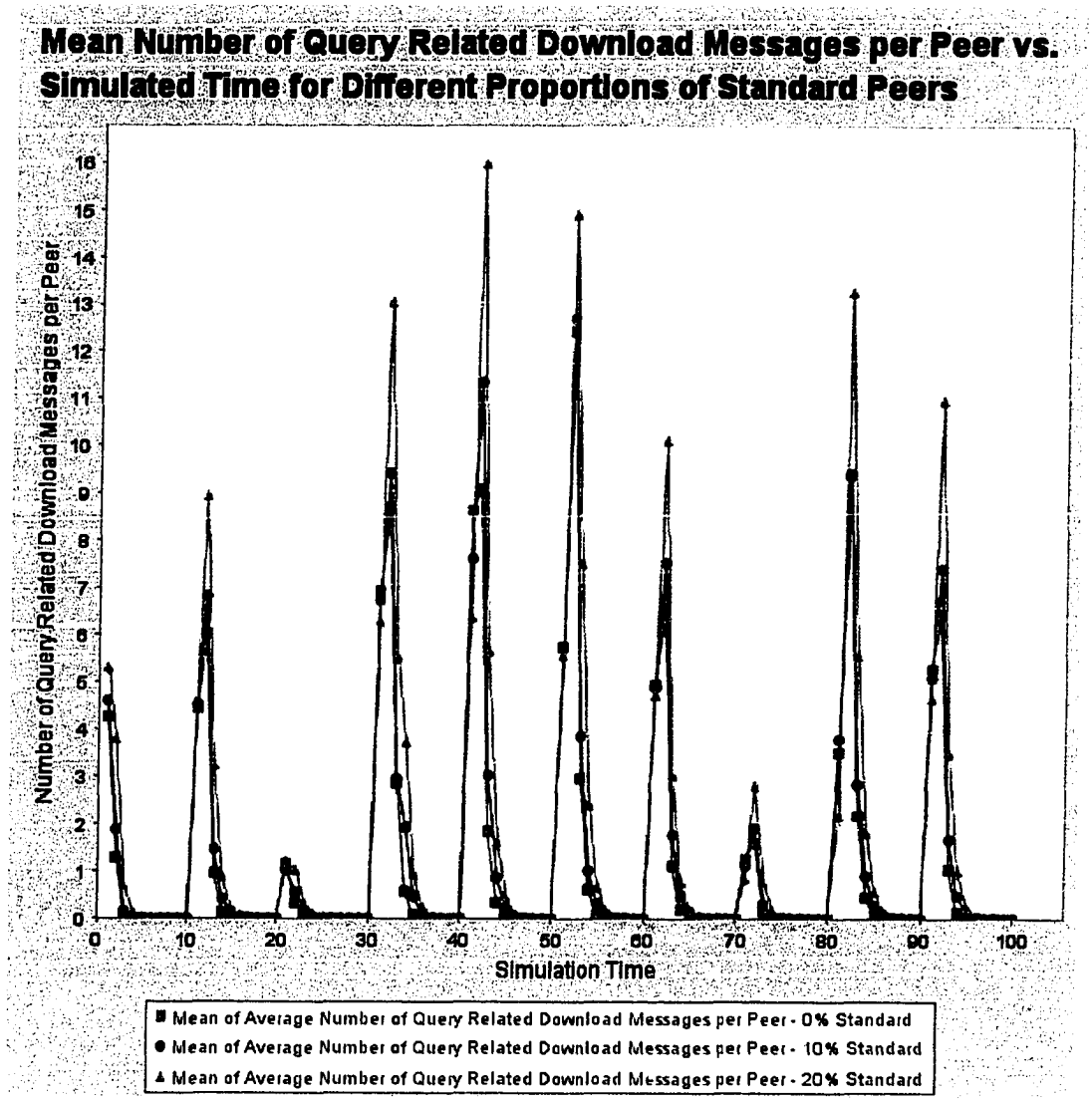

Figure 6.8 Mean Number of Query Related Download Messages per Peer for Different Proportions of Standard Peers

Figure 6.8 shows the mean number of query related messages in simultaneous transmission to peers for all three runs, averaged over each simulated time interval. The significance of these values is that as the number of standard peers increases, so

does the number of query related messages in transmission. This signifies that query related messages are taking longer to transmit because of a decreased share of bandwidth. Once again, the increases are more significant as the proportion of standard peers changes from 10 to 20 percent.

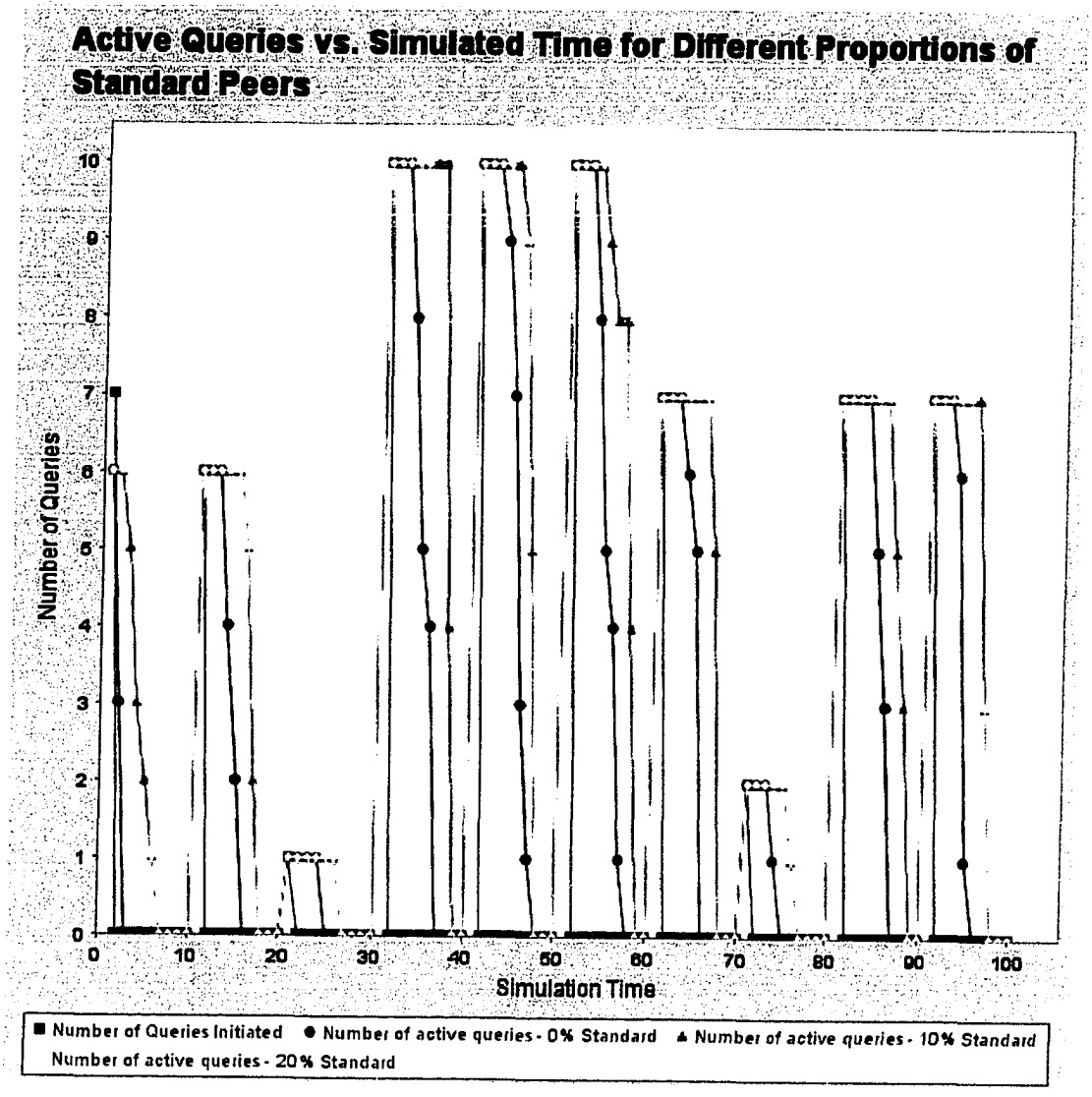

Figure 6.9 Number of Active Queries for Different Proportions of Standard Peers

A different comparison between the three runs is shown in Figure 6.9. Here, the statistic used is the number of active queries, which is defined as a query for which related messages are still in transmission. As seen in Figures 6.4 to 6.8, more standard peers means less bandwidth allocated to query related messages, which in turn means that on average there are more query related transmissions at any given time. One would expect that this would lead to queries taking longer to complete. Figure 6.9 demonstrates that this is indeed the case.

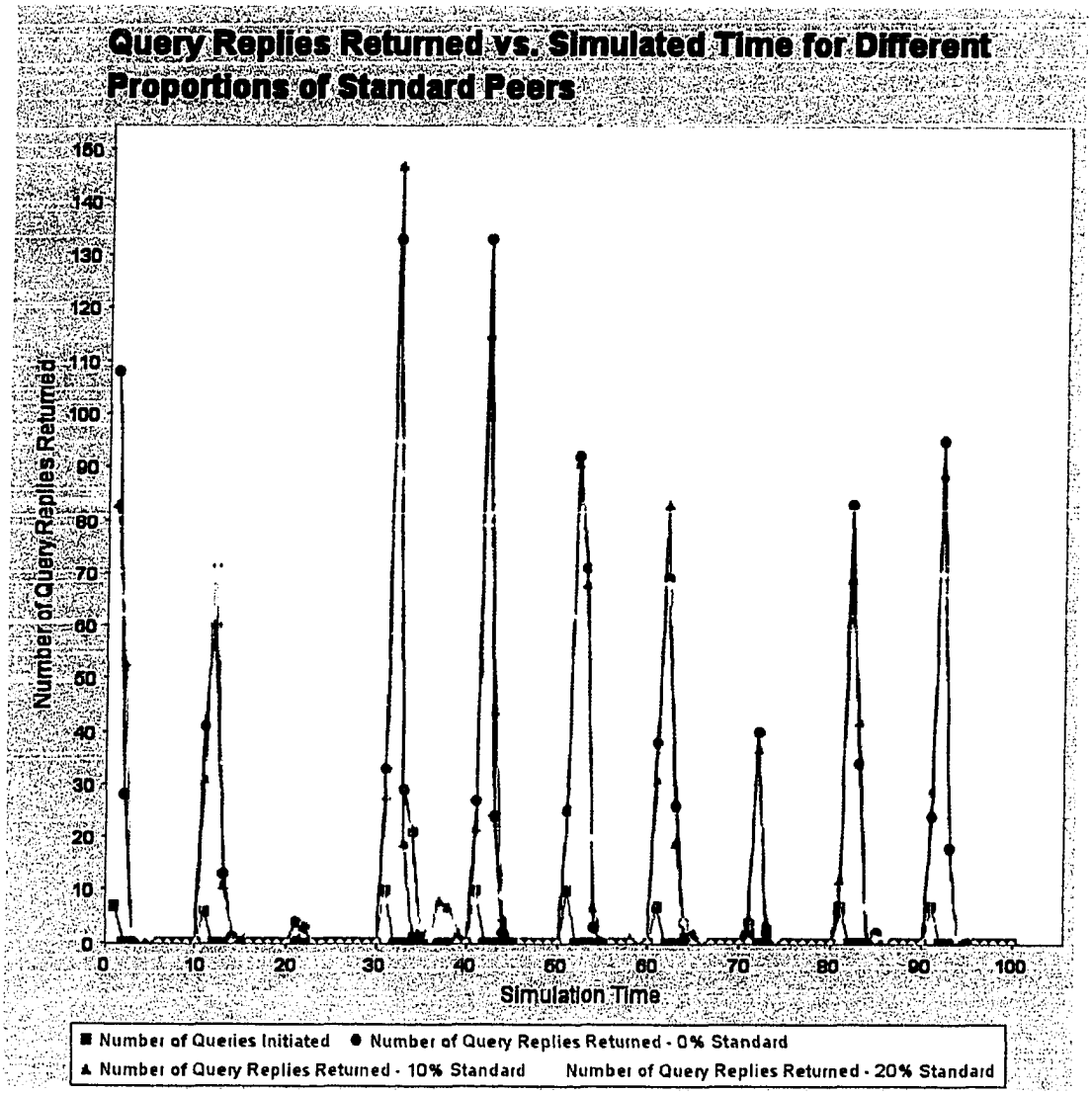

Figure 6.10 Number of Query Replies Returned for Different Proportions of Standard Peers

The effects illustrated by Figure 6.9 are shown even more clearly by Figure 6.10. This figure shows the number of query replies returning to the originating peer during each time interval for each of the three simulation runs. With lower percentages of standard Gnutella peers, query replies are returned faster; more query

replies are returned in the simulated time intervals immediately following the initiation of the corresponding query.

In general, the statistics produced by the simulator for the three simulation runs with different proportions of standard and pong caching Gnutella peers show that the simulator does reflect the expected behaviour of the real world Gnutella network. The introduction of pong caching significantly reduces the burden of ping and pong message traffic on the network, and frees up more resources for query traffic.

6.3 High vs. Low Query Traffic

As described in the introduction to Chapter 5 and in Section 5.2.1, the simulator's Gnutella plug-in supports the initiation of queries for content. These queries cause query messages to be broadcast or flooded throughout the network, searching for peers storing the desired content. To investigate the impact of query traffic on the Gnutella network, three simulation runs were performed with queries being issued at different rates and intervals.

Table 6.5 lists the *common* parameters used for these simulation runs, and Table 6.6 lists the *different* parameters used. For an explanation of the parameters and a description of how they were provided to the simulator, refer to Appendix C.

Table 6.5 Common Parameters Used for High vs. Low Query Traffic Simulation Runs

Parameter	Value
Random Number Seed	51957253545678
Time to Stop (Simulated Time Units)	100.0

Number of Peers	10,000
Average Number of Links per Peer	12
Topological Structure of Inter-Peer Links	BRITE Waxman (Random)
Maximum Upload Bandwidth per Peer (bytes per Simulated Time Unit)	10,000
Maximum Download Bandwidth per Peer (bytes per Simulated Time Unit)	10,000
Number of Content Records in Repository	5000
Average Number of Content Records per Peer	10
Percentage of Pong Caching Gnutella Peers	95
Percentage of Standard Gnutella Peers	5
Max Pongs per Hop Count in Pong Cache	5
Cached Pongs Returned per Ping	10
Cached Ping Interval (Simulated Time Units)	4.0
Standard Ping Interval (Simulated Time Units)	40.0
Max Hop Count for Queries and Standard Pings	7
Percentage of Peers Using Flow Control	100
Max Messages per Connection	12
Max Queued Query Replies per Connection	6
Max Queued Queries per Connection	5
Max Queued Pongs per Connection	4
Max Queued Pings per Connection	3

Table 6.6 Different Parameters Used for High vs. Low Query Traffic Simulation Runs

Parameter	Simulation Run 1	Simulation Run 7	Simulation Run 8
Query Generation Interval (Simulated Time Units)	10	2	2
Query Generation Rate (Poisson) per Generation Interval	5	5	15

Table 6.6 shows the interval between query generations and the query generation rates for the three runs. In the first of the runs, queries were issued at the nominal rate of 5 per every 10 simulated time units. In the second run, queries were

issued at shorter intervals, with 5 per every 2 simulated time units. Finally, in the third run, the query generation rate was increased to 15 per every 2 simulated time units. To highlight the effects of different query generation rates and intervals on the simulated Gnutella network, several statistics were chosen for graphical display.

Figure 6.11 to 6.13 illustrate that individual queries take longer to complete when there are more simultaneous queries active in the network. Figure 6.14 illustrates that a higher proportion of concurrent queries in the network results in a significantly higher level of query related message traffic to peers in the network. Figure 6.15 illustrates the effects of this higher level of query related traffic on the transmission times of other types of messages, namely pings and pongs. Finally, Figure 6.16 illustrates how a higher number of concurrently active queries results in queries competing against each other for resources, causing an increase in the average time in which query replies are returned.

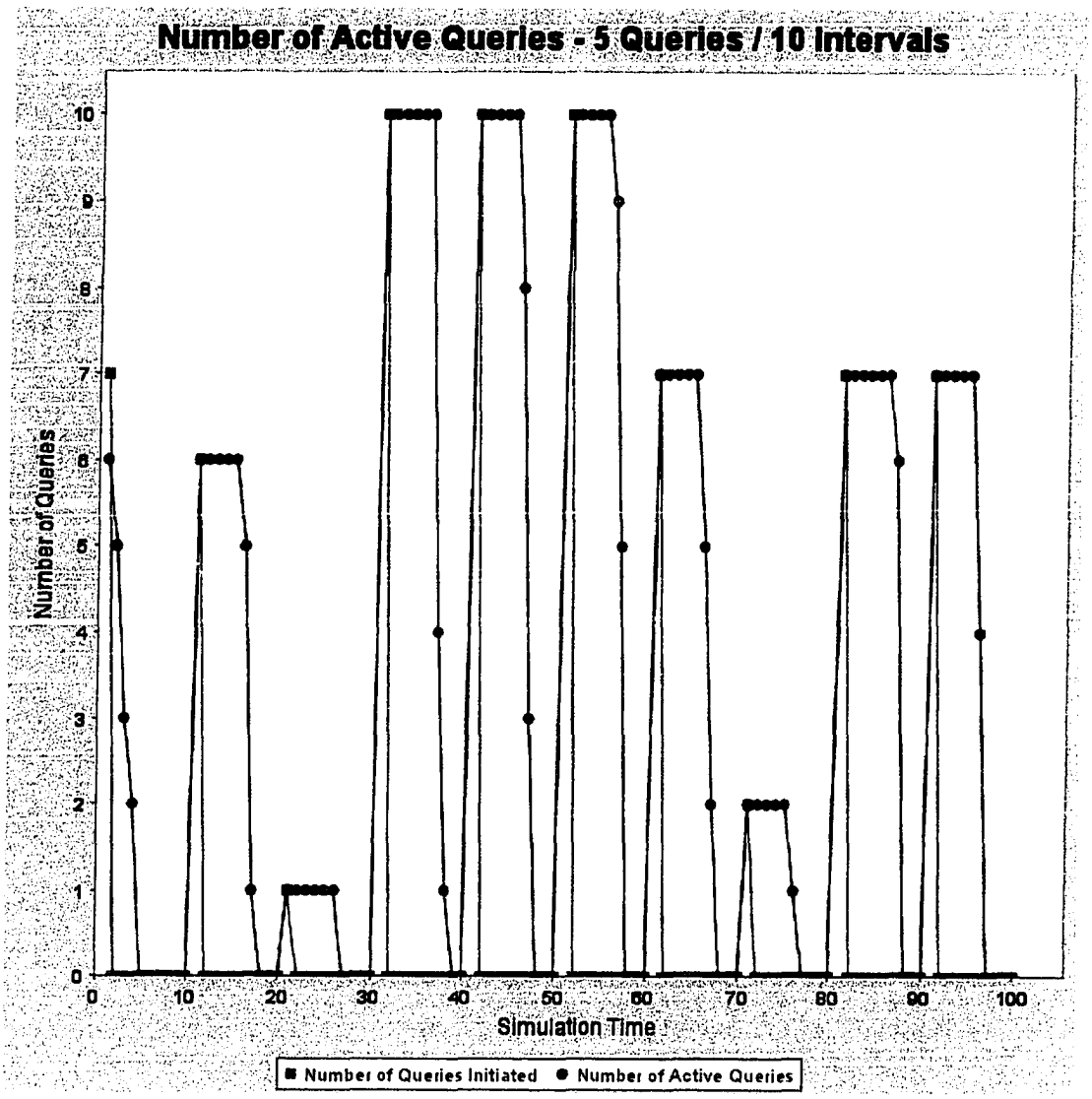

Figure 6.11 Number of Active Queries – 5 Queries per 10 Simulated Time Units

Figure 6.11 illustrates that query initiations for the first simulation run were scheduled to occur only periodically, at well spaced intervals. In this scenario, the load of query traffic on the network is minimal, and no messages pertaining to a query

are still in transit by the time the next batch of queries is initiated, 10 simulated time units later.

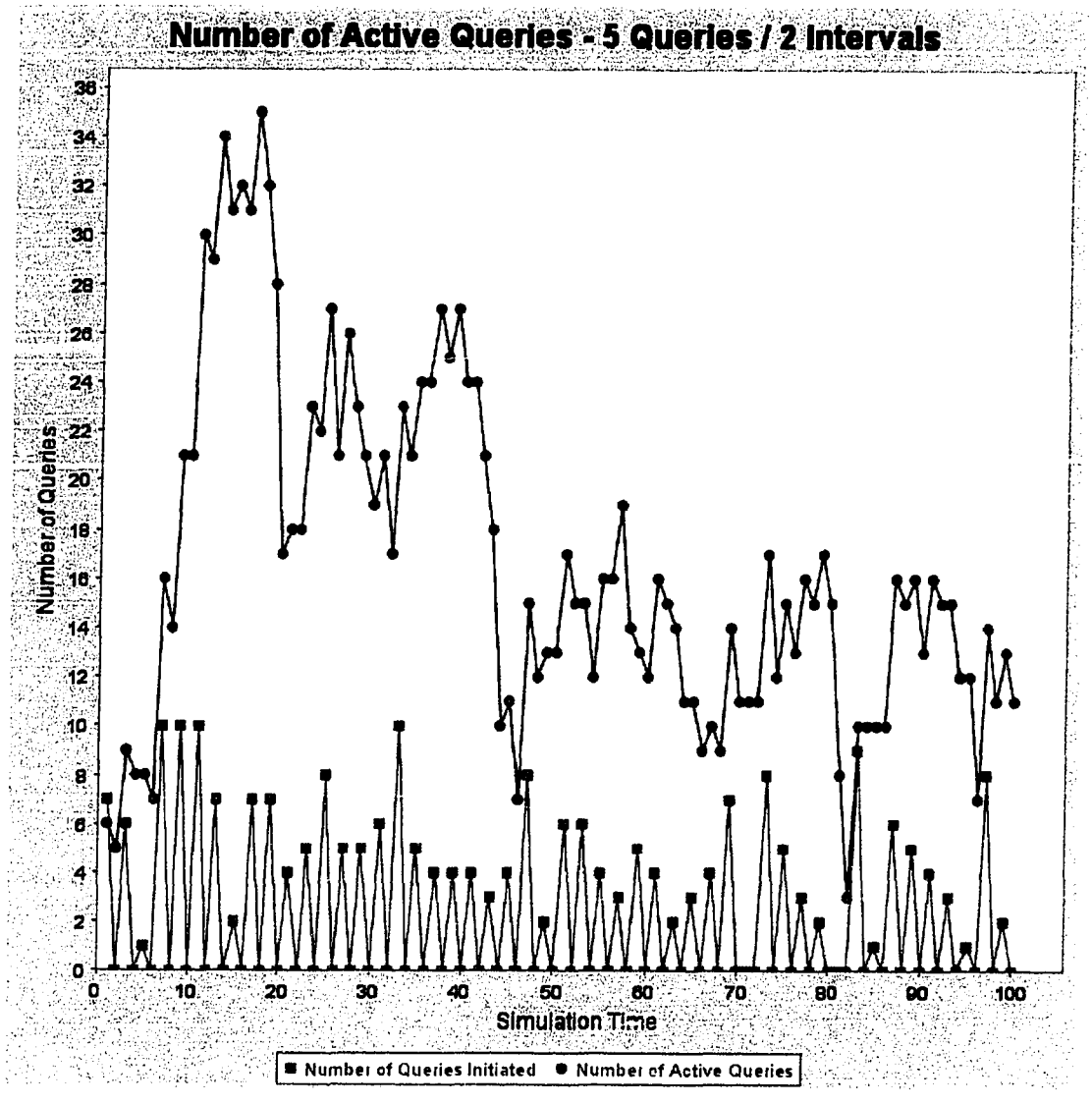

Figure 6.12 Number of Active Queries - 5 Queries per 2 Simulated Time Units

By contrast, Figure 6.12 illustrates the effects of increasing the rate of query initiations. In this scenario, queries are initiated at more frequent time intervals, before

the queries initiated in the previous interval have had a chance to complete. Thus, the number of concurrently active queries outpaces the number of queries initiated over each interval.

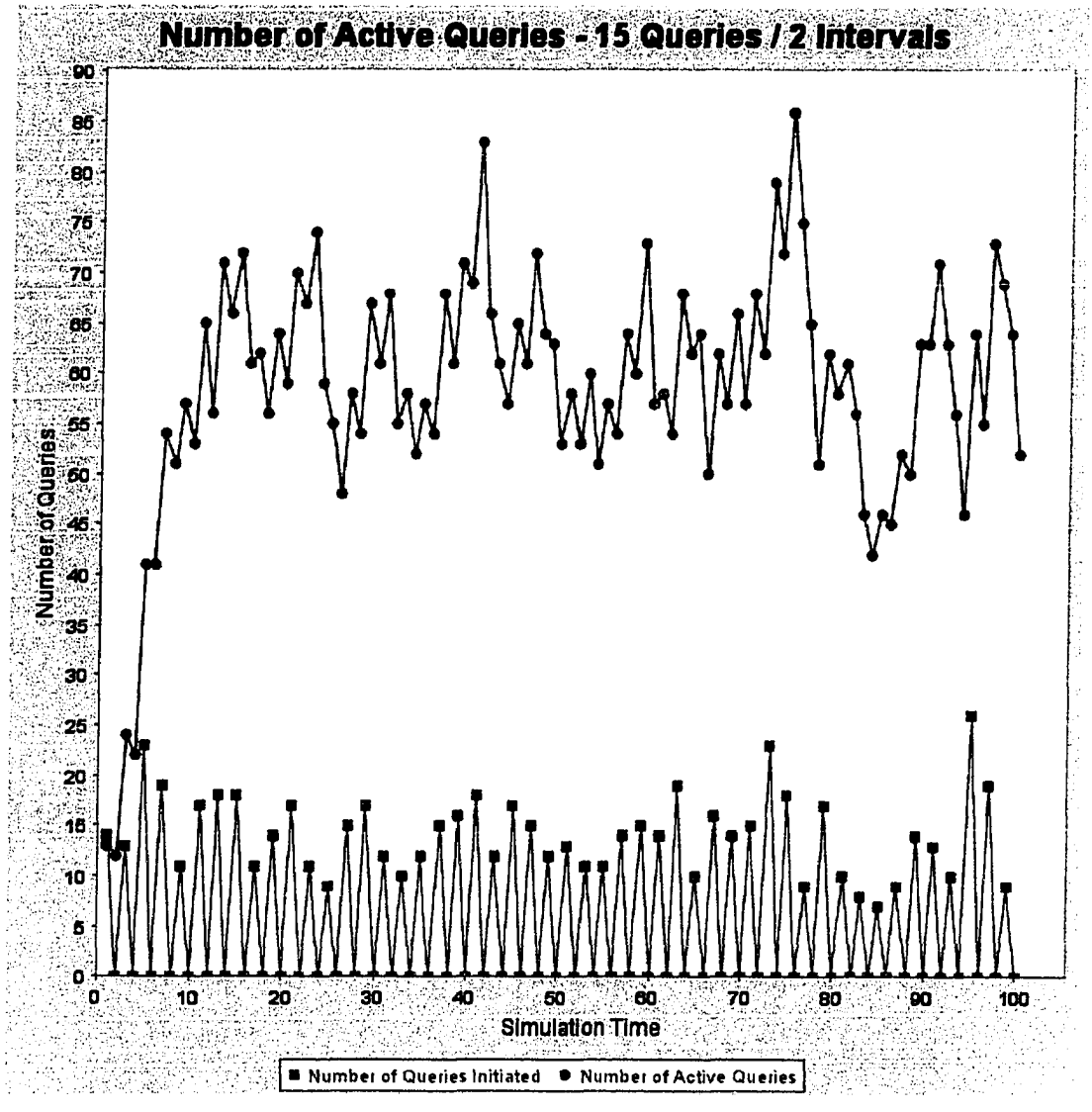

Figure 6.13 Number of Active Queries - 15 Queries per 2 Simulated Time Units

Figure 6.13 illustrates the effects of increasing both the rate and number of query initiations. In this scenario, the number of concurrently active queries dramatically outpaces the number of queries initiated over each interval, signifying that more and more queries are competing with each other for network bandwidth.

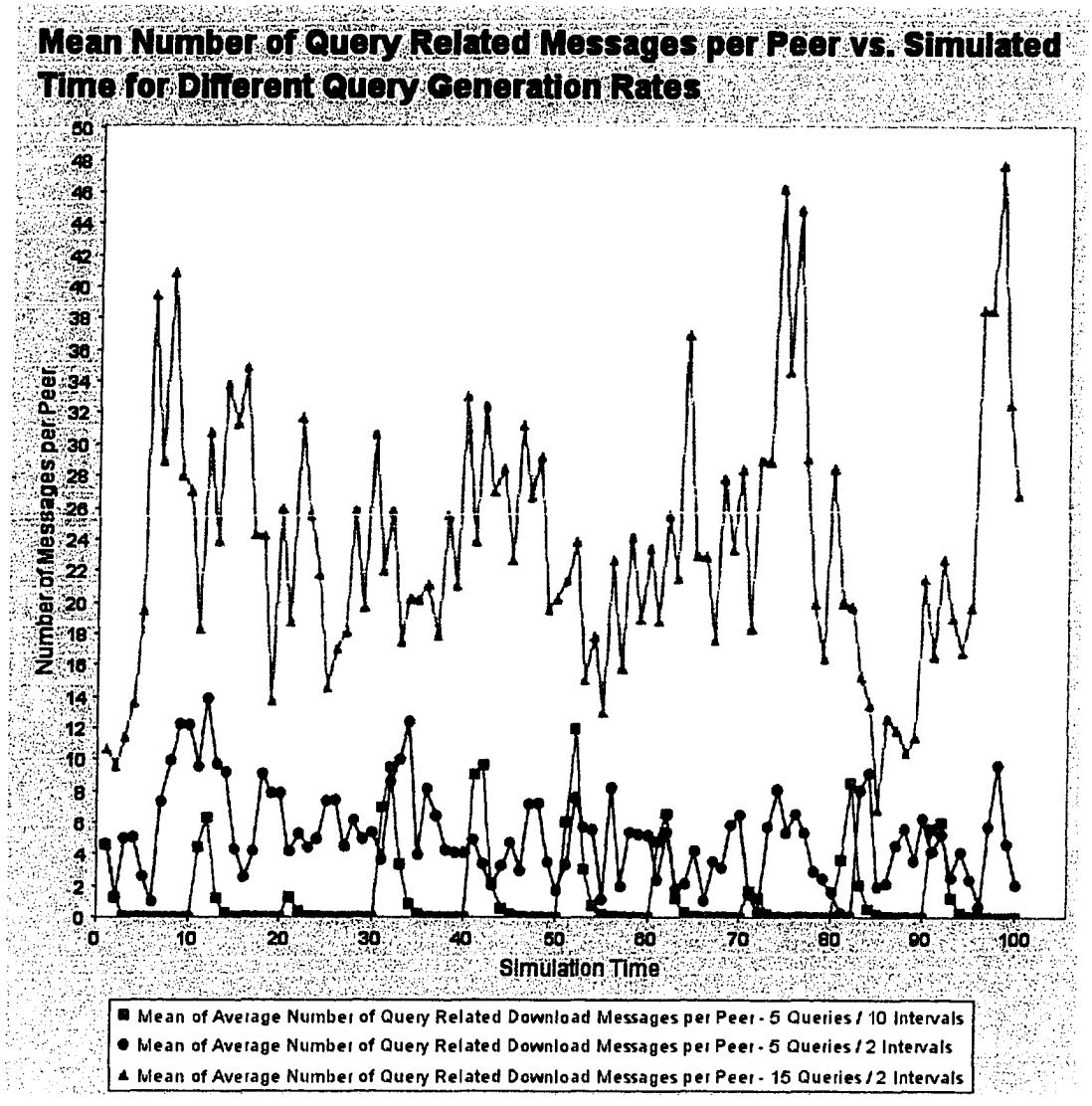

Figure 6.14 Mean Number of Query Related Messages per Peer for Different Query Generation Rates

Figure 6.14 provides a comparison of the average number of query related messages in transmission for each of the three simulation runs. As the rate and number of initiated queries increases, so does the number of query related messages in

transmission. In the case of the third simulation run with the highest query rate, the significantly higher number of query related messages in transmission illustrates not only that more query related messages are being transmitted, but that they are taking longer to transmit because they are competing with each other for bandwidth.

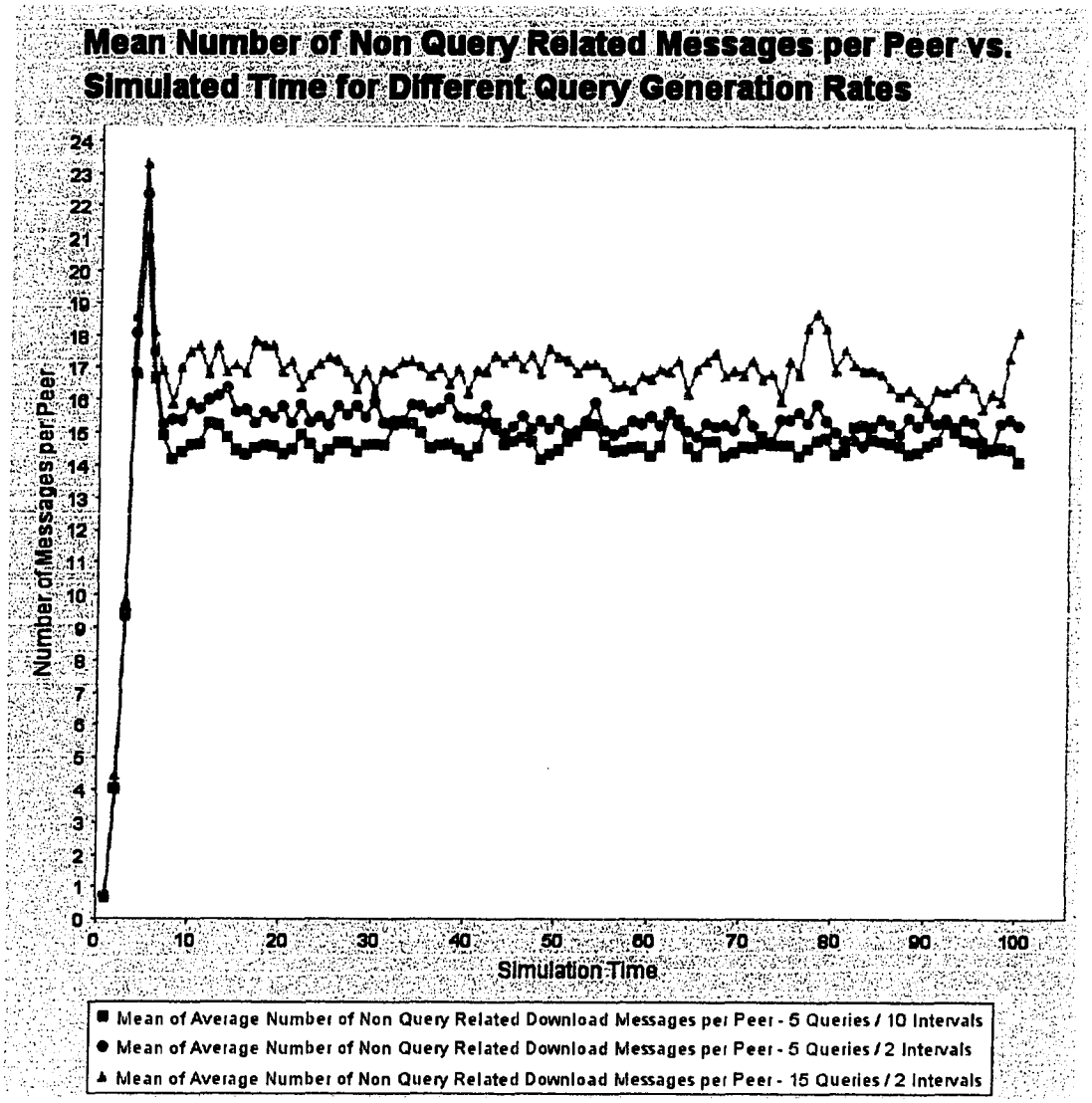

Figure 6.15 Mean Number of Non Query Related Messages per Peer for Different Query Generation Rates

Figure 6.15 illustrates the effects of an increased rate and number of query initiations on non query related traffic, namely pings and pongs. When a higher proportion of query related messages are being transmitted over the network, less

bandwidth is available to non query related messages. This causes the transmission times of non query related messages to increase, which is witnessed by the increased number of them in transmission, as shown. However, this effect on non query related messages is less dramatic than that on query related messages. This is because query related messages are typically less numerous than non query related messages, and they more highly concentrated around certain peers over shorter time intervals.

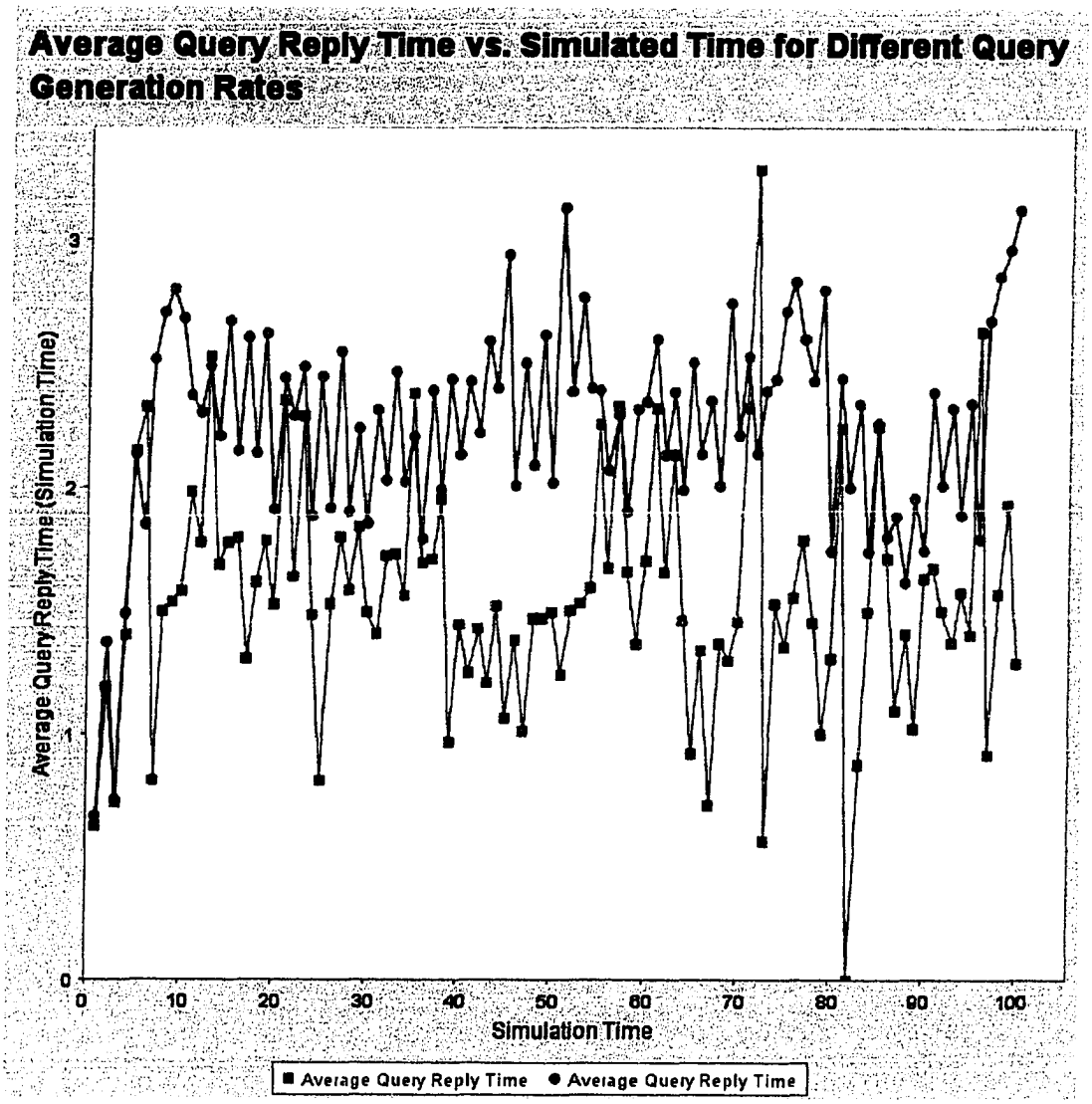

Figure 6.16 Average Query Reply Time for Different Query Generation Rates

Figure 6.16 illustrates how an increase in the number of concurrently active queries has a negative impact on the average time taken for query replies to return to the peers which initiate the corresponding query. The upper curve in this figure shows that increasing the number of queries initiated by a factor of 3 causes an increase of

approximately 1 simulated time unit in the average time taken for query replies to be returned. However, this average query reply time is a rather simplistic statistic. It is accumulated, as each query reply is returned, by adding to an accumulator variable the simulated time elapsed since the corresponding query was initiated. At the end of each time interval, the value of the accumulator variable is divided by the number of queries returned. This measurement gives a good general picture of average query response times when there are many concurrently active queries, but is less effective when queries are sporadic. For this reason, the statistics for the first simulation run are not shown in this figure.

6.4 Flow Control

As described in the introduction to Chapter 5 and in Section 5.3.1, the simulator's Gnutella plug-in supports flow control. Flow control provides a scheme for limiting the number of messages in transmission over a connection between any two peers. Messages are dropped only when strictly necessary, and the type of message to drop is selected in an intelligent manner, so as to prioritize certain message types over others. To investigate the impact of flow control on the Gnutella network, three simulation runs were performed, one with low message traffic, and two with high message traffic. The high traffic simulation runs were performed both with and without flow control.

Table 6.7 lists the *common* parameters used for these simulation runs, and Table 6.8 lists the *different* parameters used. For an explanation of the parameters and a description of how they were provided to the simulator, refer to Appendix C.

Table 6.7 Common Parameters Used for Flow Control Simulation Runs

Parameter	Value
Random Number Seed	51957253545678
Time to Stop (Simulated Time Units)	100.0
Number of Peers	10,000
Average Number of Links per Peer	12
Topological Structure of Inter-Peer Links	BRITE Waxman (Random)
Number of Content Records in Repository	5000
Average Number of Content Records per Peer	10
Percentage of Pong Caching Gnutella Peers	95
Percentage of Standard Gnutella Peers	5
Max Pongs per Hop Count in Pong Cache	5
Cached Pongs Returned per Ping	10
Cached Ping Interval (Simulated Time Units)	4.0
Standard Ping Interval (Simulated Time Units)	50.0
Max Hop Count for Queries and Standard Pings	7
Max Messages per Connection	12
Max Queued Query Replies per Connection	6
Max Queued Queries per Connection	5
Max Queued Pongs per Connection	4
Max Queued Pings per Connection	3
Query Generation Interval (Simulated Time Units)	1.0
Query Generation Rate (Poisson) per Generation Interval	3

Table 6.8 Different Parameters Used for Flow Control Simulation Runs

Parameter	Simulation Run 9	Simulation Run 10	Simulation Run 11
Percentage of Peers Using Flow Control	100 %	100 %	0 %
Maximum Upload / Download Bandwidth per Peer	100 % 10k / 10k	100 % 400 / 400	100 % 400 / 400

(bytes per Simulated Time Unit)			

Table 6.8 shows the maximum bandwidth available to peers and the percentage of peers using flow control for the three runs. In the first of the runs, a large amount of bandwidth was made available to the peers, resulting in fast message transmission times, and low traffic. In the second run, a significantly lower amount of bandwidth was made available, resulting in slower message transmission times, and more concurrent traffic. Finally, in the third run, the same low amount of bandwidth from the second run was made available, but flow control was turned off for all peers. To highlight the effects of flow control on the simulated Gnutella network, several statistics were chosen for graphical display.

Figure 6.17 illustrates how in low bandwidth networks, flow control helps constrain the number of message transmissions to the number the network is able to receive. Figure 6.18 illustrates the extensive use of the flow control queues for low bandwidth networks. Figure 6.19 illustrates the side-effect of flow control, as queue utilization increases, more messages are dropped. Finally, Figures 6.20 and 6.21 illustrate the total number of messages in transmission in networks with low bandwidth, both with and without flow control.

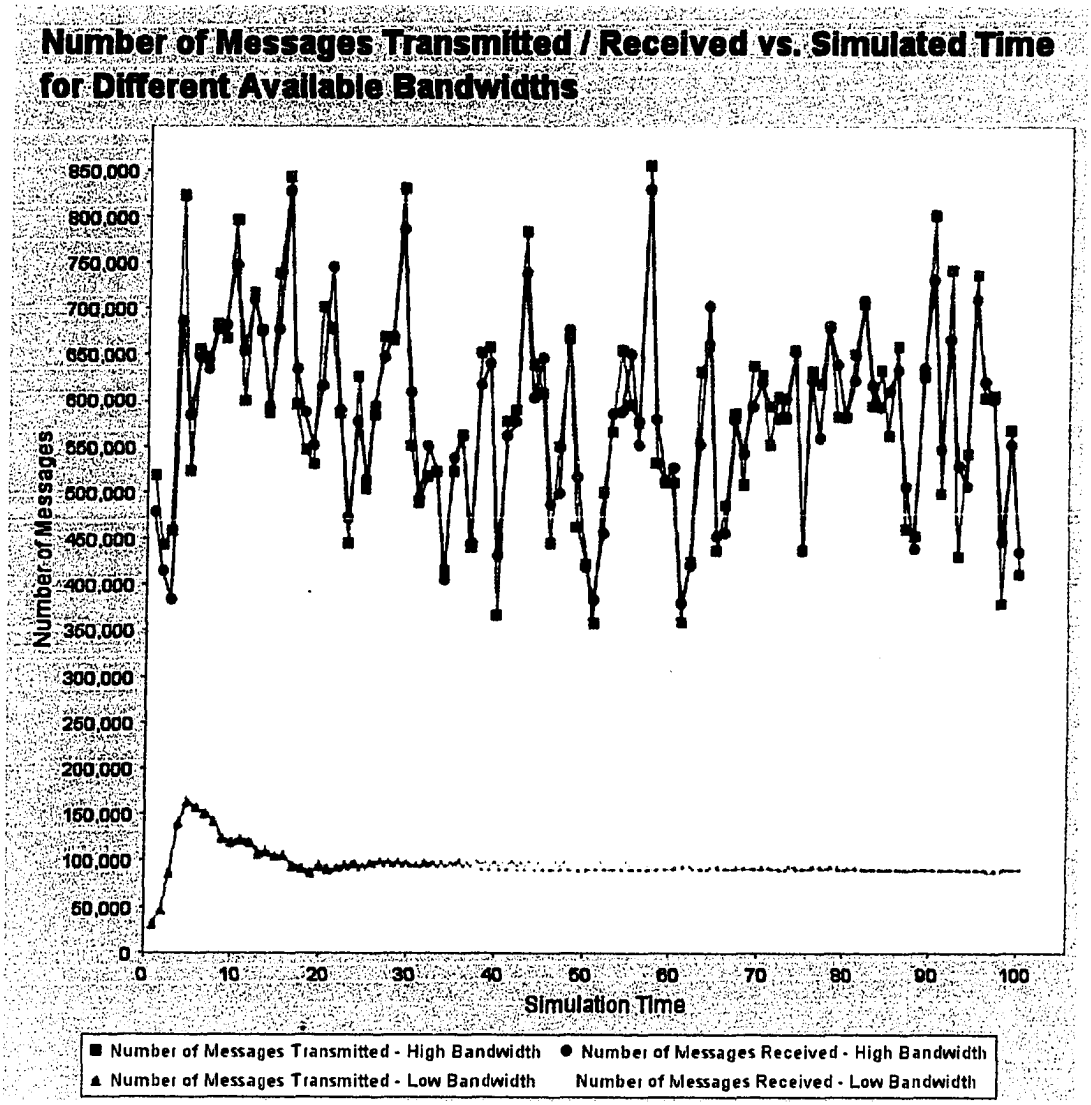

Figure 6.17 Number of Messages Transmitted & Received with Flow Control for Different Available Bandwidths

Figure 6.17 illustrates the total number of messages transmitted and received for the two simulation runs where all peers are using flow control. For the second of these simulation runs, where peers were given lower available bandwidth, messages

take longer to transmit. This is visible in the lower curves in the figure which show that initially, there are significantly more message transmissions than receptions per time interval. The purpose of flow control is to limit the resulting build-up of messages in the network, preventing message transmissions from outpacing message receptions. The effect of flow control is visible in the figure at simulation time 20, where message transmissions are curtailed to the level of receptions.

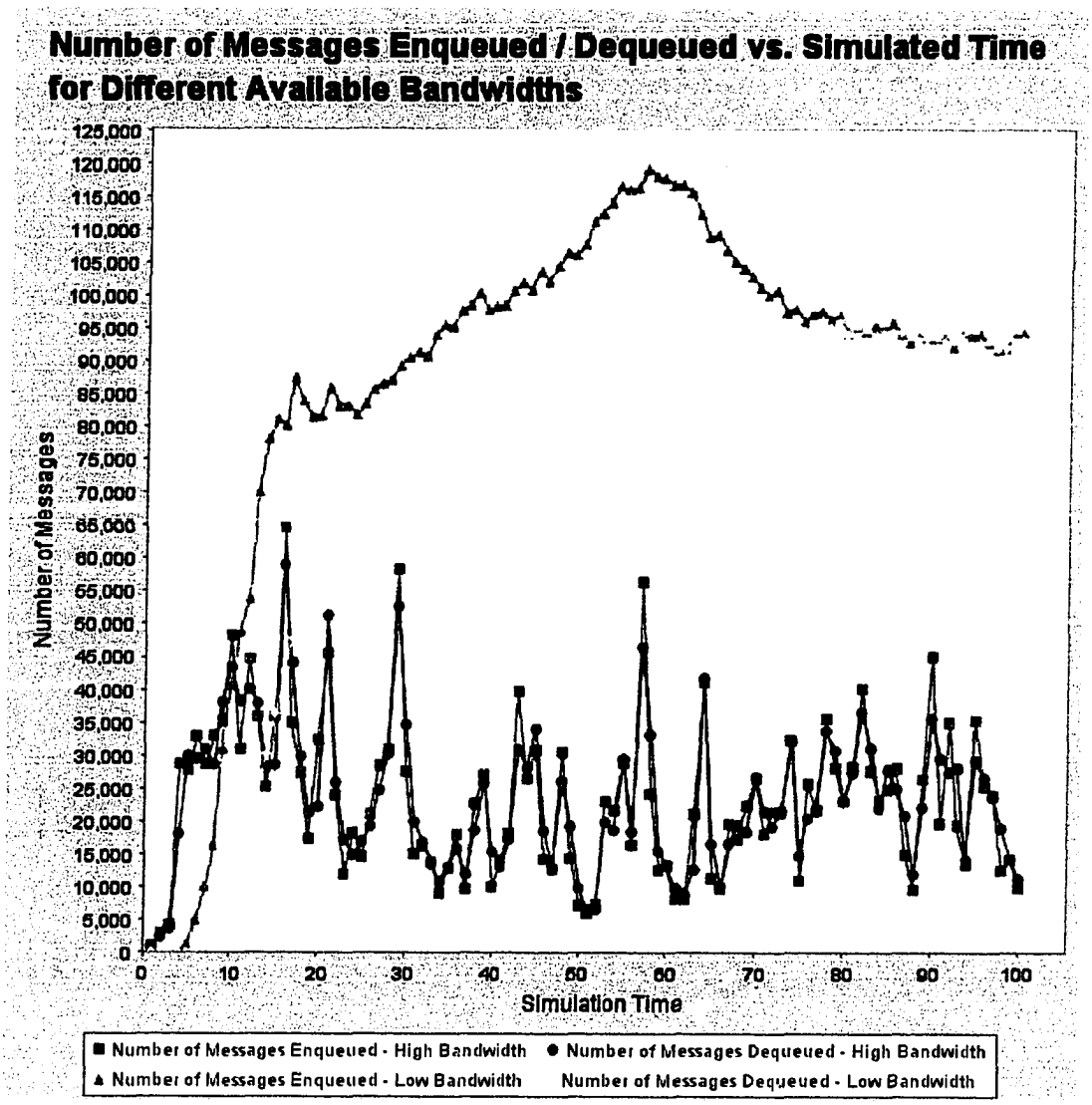

Figure 6.18 Number of Messages Enqueued & Dequeued with Flow Control for Different Available Bandwidths

Figure 6.18 illustrates the total number of messages enqueued and dequeued for the simulation runs where all peers are using flow control. The lower curves show that with high available bandwidth, only a small proportion of messages are queued

prior to transmission. The upper curves show that the simulation run with lower available bandwidth makes significant use of the flow control queue, with virtually all messages being queued prior to transmission.

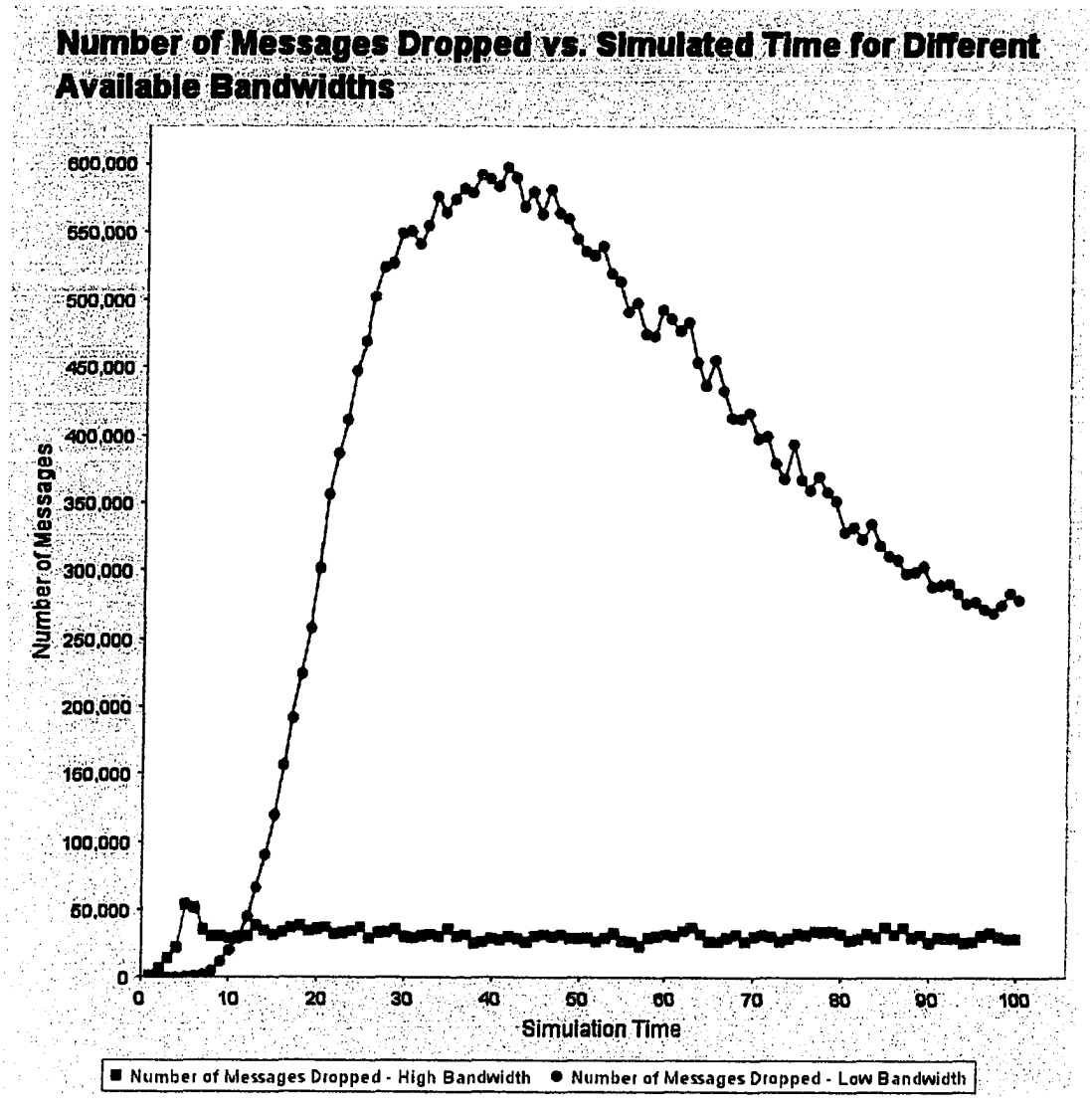

Figure 6.19 Number of Messages Dropped with Flow Control for Different Available Bandwidths

Figure 6.19 illustrates the number of messages dropped from the flow control queue for the simulation runs where all peers are using flow control. The upper curve shows that a significant number of messages are being dropped by the simulation run with lower available bandwidth. This is a symptom of high utilization of the flow control queue. When the queue becomes full, previously queued messages must be dropped in order to make room to enqueue new messages.

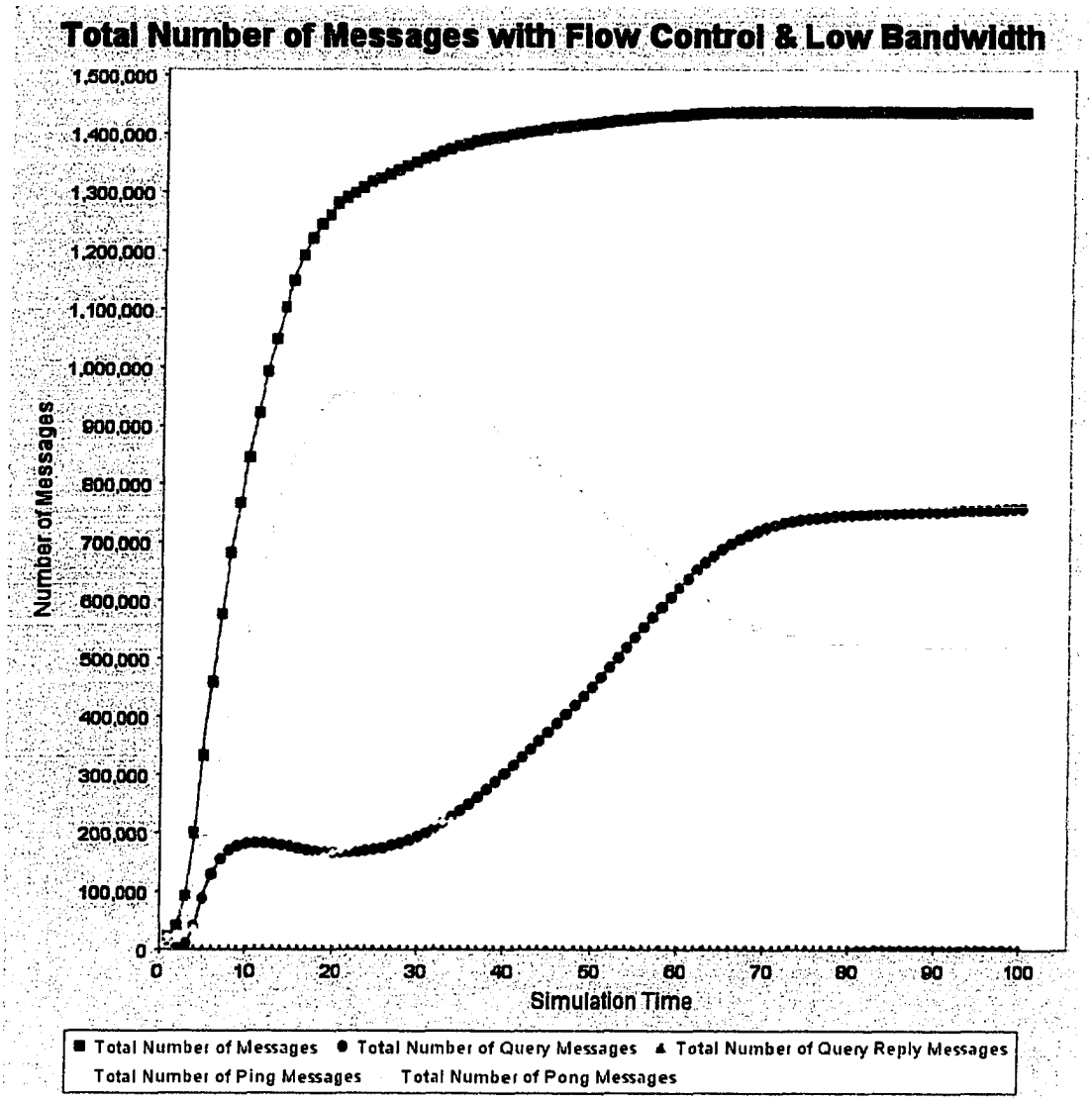

Figure 6.20 Total Numbers of Messages in Transmission with Flow Control and Low Bandwidth

Figure 6.20 illustrates how flow control regulates the number of messages in transmission in a network with low available bandwidth. The curves in this figure show that the flow control algorithm implemented by Gnutella is successful in prioritizing the transmission of query messages over pong messages.

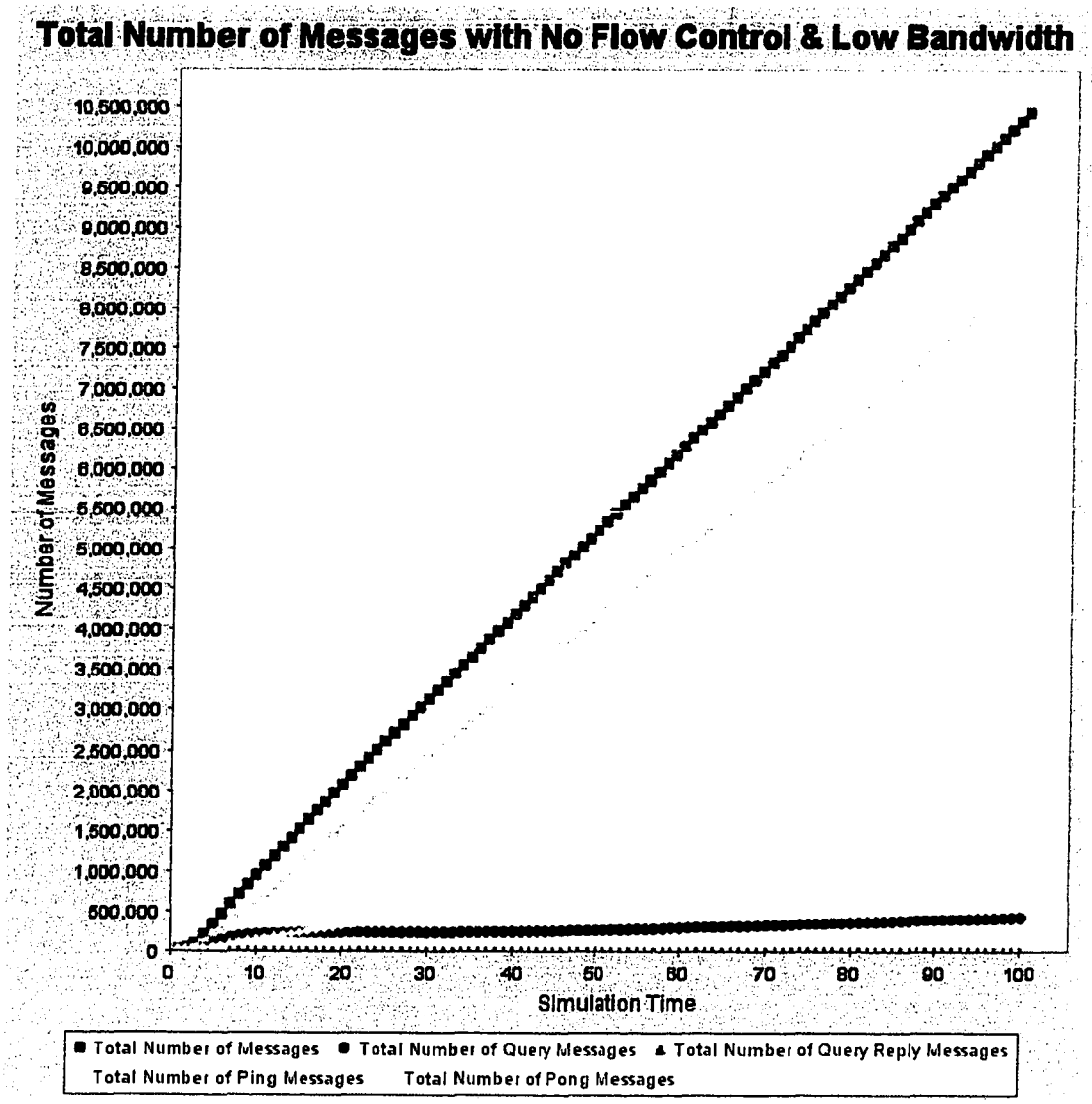

Figure 6.21 Total Numbers of Messages in Transmission with No Flow Control and Low Bandwidth

Figure 6.21 illustrates the effects of the absence of flow control on a network with low available bandwidth. Without flow control, the number of messages in transmission on the network grows without bounds. A protocol which exhibits

unbounded growth in the number of concurrent message transmissions is not scalable, and presents a serious barrier to simulation.

6.5 Changing Topology

6.5.1 Random vs. Power Law Topologies

As described in Section 2.2.2.2, there has been a considerable amount of debate over whether the network topology evolved by Gnutella exhibits power law characteristics. Therefore it is of interest to investigate the performance of the Gnutella network when inter-peer links form a power law distribution. To this end, two simulation runs were performed, one with a random topology, and the other with a power law topology.

Table 6.9 lists the *common* parameters used for these simulation runs, and Table 6.10 lists the *different* parameters used. For an explanation of the parameters and a description of how they were provided to the simulator, refer to Appendix C.

Table 6.9 Common Parameters Used for Random vs. Power Law Simulation Runs

Parameter	Value
Random Number Seed	51957253545678
Time to Stop (Simulated Time Units)	100.0
Number of Peers	10,000
Average Number of Links per Peer	12
Maximum Upload Bandwidth per Peer	10,000
Maximum Download Bandwidth per Peer	10,000
Number of Content Records in Repository	5000
Average Number of Content Records per Peer	10
Percentage of Pong Caching Gnutella Peers	95
Percentage of Standard Gnutella Peers	5
Percentage of Peers Using Flow Control	100

Max Pongs per Hop Count in Pong Cache	5
Cached Pongs Returned per Ping	10
Cached Ping Interval (Simulated Time Units)	4.0
Standard Ping Interval (Simulated Time Units)	50.0
Max Hop Count for Queries and Standard Pings	7
Max Messages per Connection	12
Max Queued Query Replies per Connection	6
Max Queued Queries per Connection	5
Max Queued Pongs per Connection	4
Max Queued Pings per Connection	3
Query Generation Interval (Simulated Time Units)	10.0
Query Generation Rate (Poisson) per Generation Interval	5

Table 6.10 Different Parameters Used for Random vs. Power Law Simulation Runs

Parameter	Simulation Run 1	Simulation Run 12
Topological Structure of Inter-Peer Links	BRITE Waxman (Random)	BRITE Power Law

Table 6.10 shows that in the first run, a random graph was used to initialize the protocol-level links between peers. Likewise, in the second run, a power law graph was used for initializing the protocol-level links, meaning there were a few highly connected peers, and many poorly connected peers. To highlight the effects of the different topologies on the simulated Gnutella network, several statistics were chosen for graphical display.

Figure 6.22 illustrates the different amounts of bandwidth used for the runs with random or power law topologies. Similarly, Figure 6.23 illustrates the different number of messages in transmission for the two runs, providing insight into which topological structure results in the highest degree of concurrent message traffic.

Finally, Figure 6.24 illustrates the effects of the different topological structures on the responsiveness of Gnutella query replies.

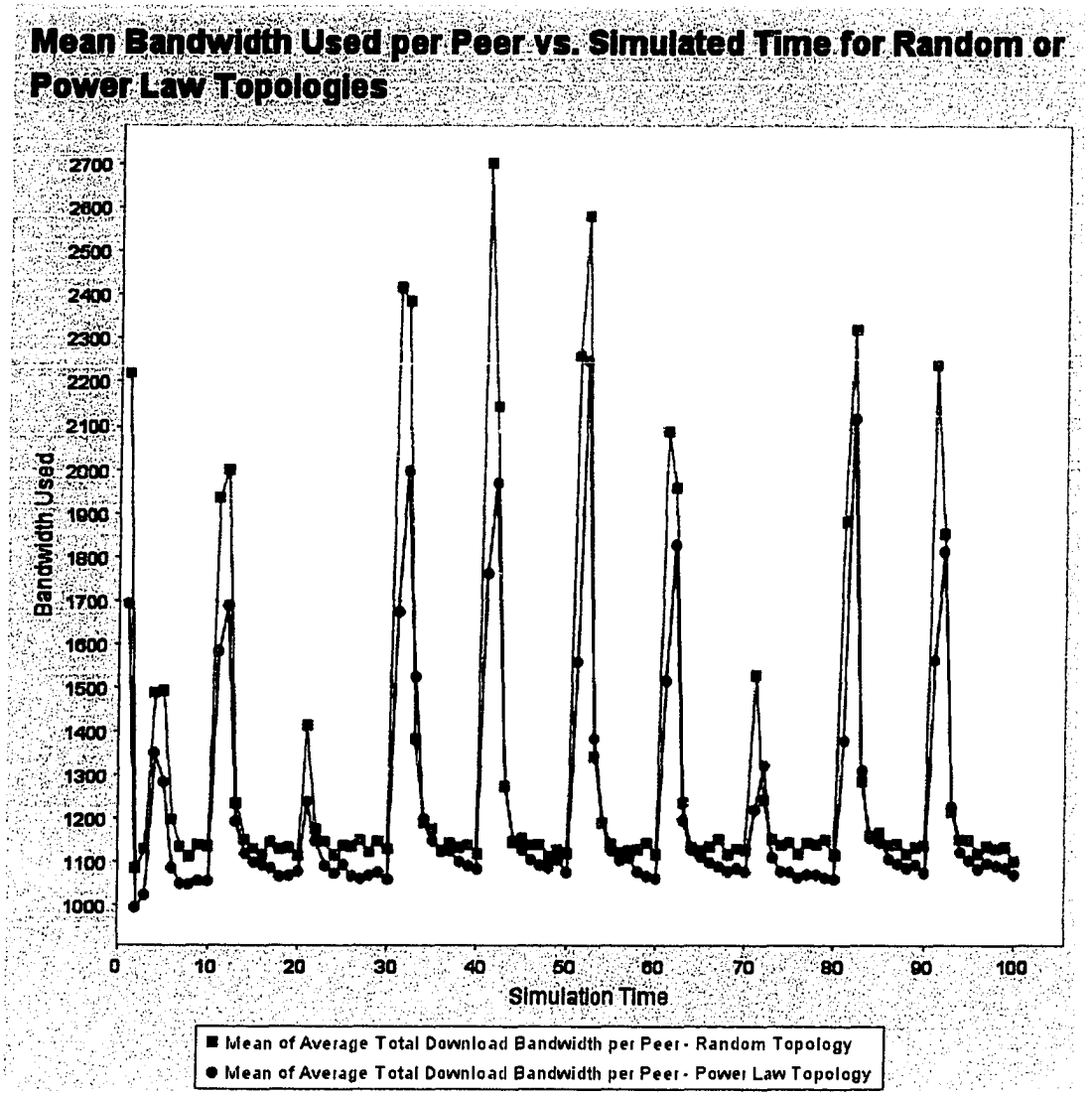

Figure 6.22 Mean Bandwidth Used per Peer for Random or Power Law Topologies

Figure 6.22 illustrates that on average, more bandwidth is used by peers in the random network than in the power law network. This is a result of traffic being more

evenly distributed for the random network topology. In the random network topology, there are fewer peers with a high number of connections, which in turn means that there are fewer peers that impose constraints on the bandwidth allocated to messages. Conversely, in a power law topology, several peers are highly connected, meaning that they constrain messages to a low share of their bandwidth. These highly connected peers are connected to many peers with low connectivity, which have a large proportion of unused bandwidth.

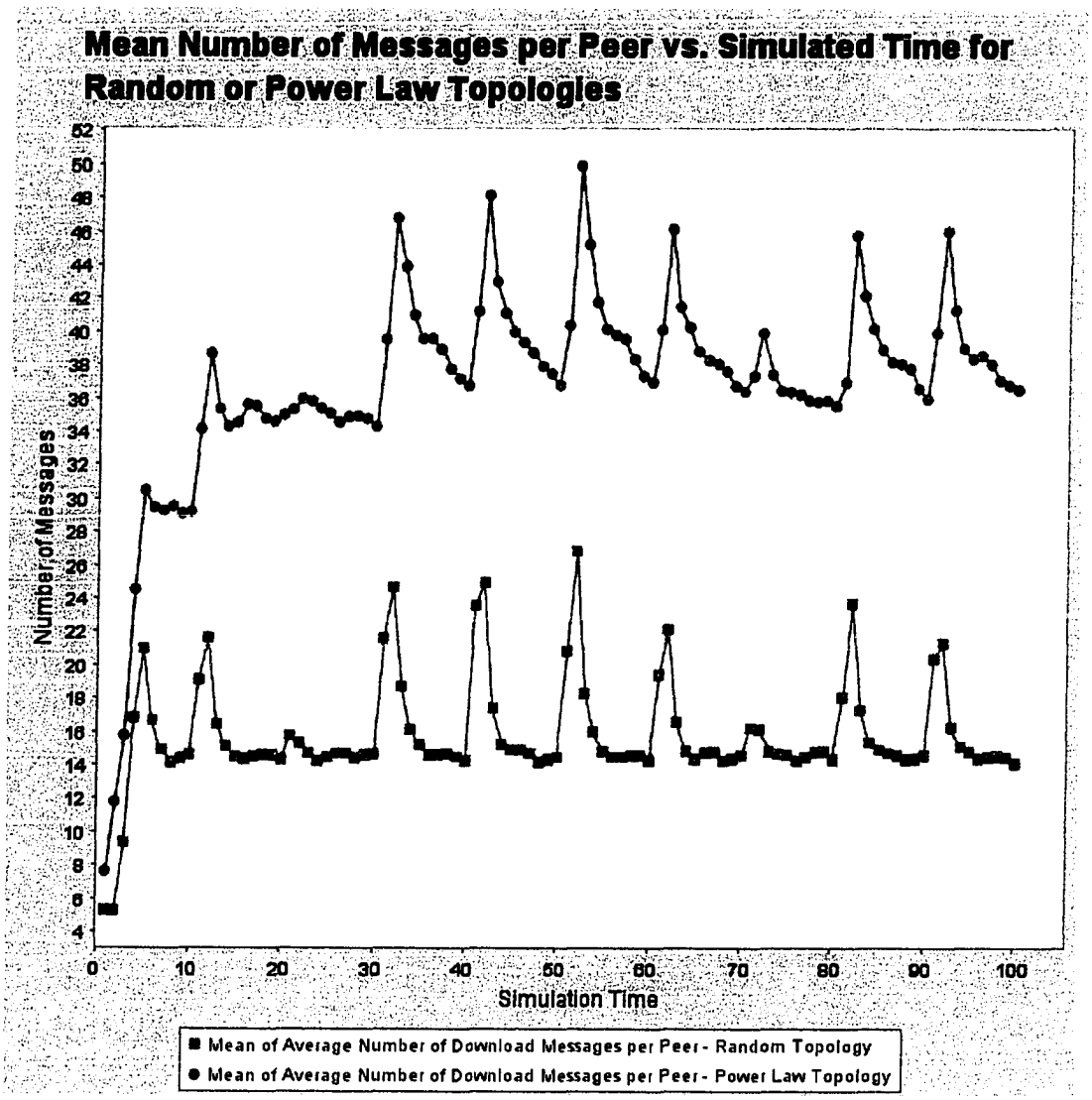

Figure 6.23 Mean Number of Messages per Peer for Random or Power Law Topologies

Figure 6.23 illustrates average number of messages in transmission to peers for the two simulation runs. As expected, the results with the power law topology show that there are many more concurrent message transmissions than with the random

topology. This is because less bandwidth is being allocated to messages, which in turn delays their transmission.

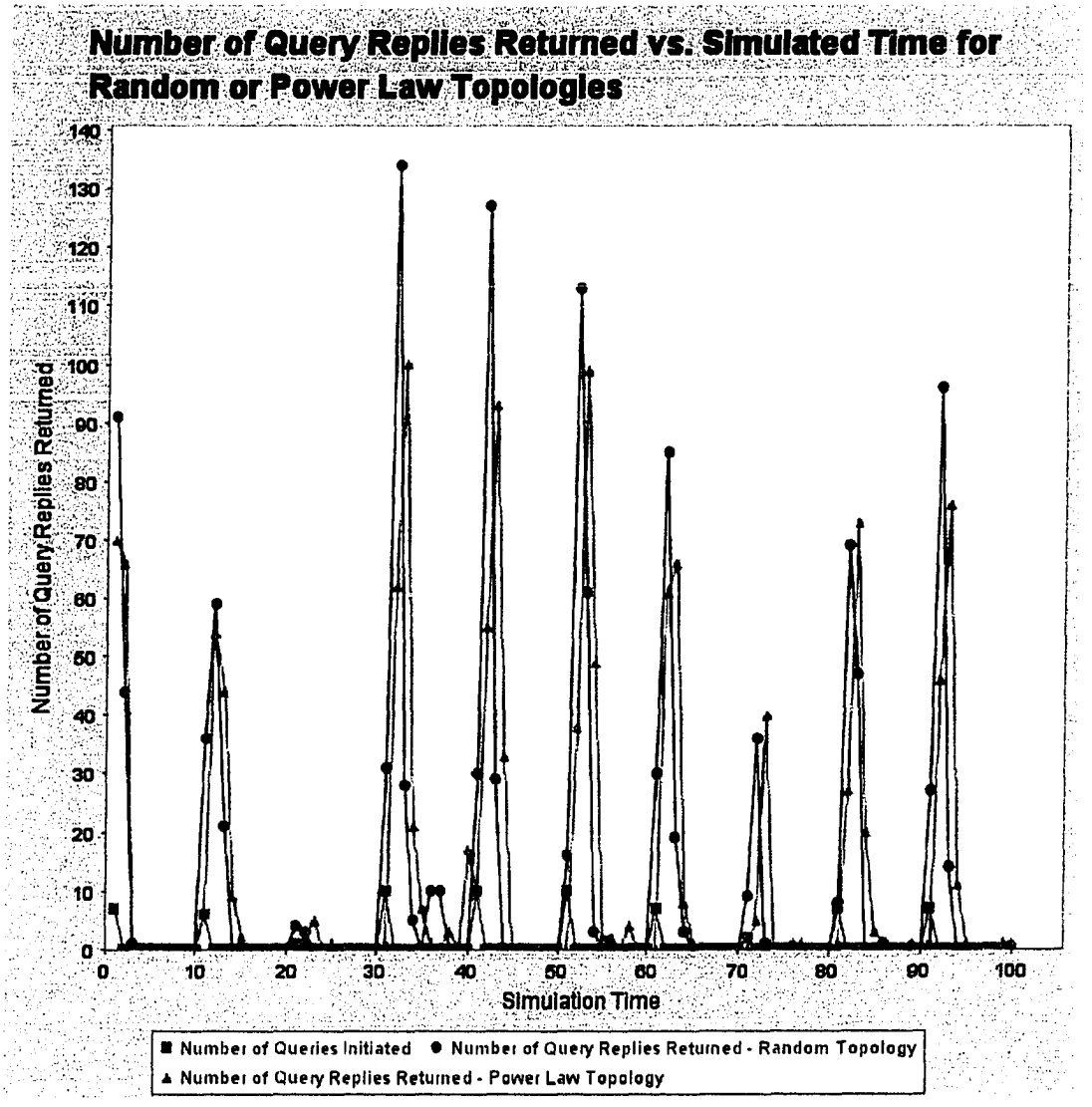

Figure 6.24 Number of Query Replies Returned for Random or Power Law Topology

Figure 6.24 illustrates the number of query replies returned per time interval for the two simulation runs. Because the random topology contains fewer highly connected peers for which higher traffic creates a bottleneck in terms of available bandwidth, query replies are returned much more quickly.

6.5.2 Dynamic Topology

As described in Section 5.2.3, the simulator's Gnutella plug-in supports the disconnection and reconnection of peers to or from the network. The disconnection of a peer from the network causes the removal of links from the protocol-level topology and the cancellation of any messages to or from that peer. To form new links when reconnecting to the network, a peer first uses the addresses of the peers to which it was linked to prior to disconnection. If an insufficient number of these peers are accepting links, then attempts are made to form links with peers from which pongs were previously received. To investigate the effects of peer disconnections and reconnections in the Gnutella network, two simulation runs were compared, one with a static topology, and the other with a high number connections and reconnections occurring over short time intervals.

Table 6.11 lists the *common* parameters used for these simulation runs, and Table 6.12 lists the *different* parameters used. For an explanation of the parameters and a description of how they were provided to the simulator, refer to Appendix C.

Table 6.11 Common Parameters Used for Dynamic vs. Static Simulation Runs

Parameter	Value
Random Number Seed	51957253545678
Time to Stop (Simulated Time Units)	400.0
Number of Peers	10,000
Average Number of Links per Peer	12
Topological Structure of Inter-Peer Links	BRITE Waxman (Random)
Maximum Upload Bandwidth per Peer	10,000
Maximum Download Bandwidth per Peer	10,000
Number of Content Records in Repository	5000
Average Number of Content Records per Peer	10
Percentage of Pong Caching Gnutella Peers	98
Percentage of Standard Gnutella Peers	2
Percentage of Peers Using Flow Control	100
Max Pongs per Hop Count in Pong Cache	5
Cached Pongs Returned per Ping	8
Cached Ping Interval (Simulated Time Units)	4.0
Standard Ping Interval (Simulated Time Units)	50.0
Max Hop Count for Queries and Standard Pings	7
Max Messages per Connection	12
Max Queued Query Replies per Connection	6
Max Queued Queries per Connection	5
Max Queued Pongs per Connection	4
Max Queued Pings per Connection	3
Query Generation Interval (Simulated Time Units)	20.0
Query Generation Rate (Poisson) per Generation Interval	5

Table 6.12 Different Parameters Used for Dynamic vs. Static Simulation Runs

Parameter	Simulation Run 13	Simulation Run 14
Maximum Number of Links for Standard Peers	10	Not Enforced
Maximum Number of Links for Pong Caching Peers	12	Not Enforced
Disconnect Generation Interval (Simulated Time Units)	2.0	None
Disconnect Generation Offset (Simulated Time Units)	30.0	None

Disconnect Generation Rate (Poisson) per Generation Interval	200	None
Connect Generation Interval (Simulated Time Units)	2.0	None
Connect Generation Offset (Simulated Time Units)	31.0	None
Connect Generation Rate (Poisson) per Generation Interval	200	None

Table 6.12 shows that the topology of the simulation runs were dynamic and static respectively. For the first run, the network was initialized using a random topology which remained fixed for 30 time intervals. At the 30th time interval, a group of approximately 200 peers disconnects; at the 31st interval, they attempt to reconnect with a maximum number of protocol-level links. This process of disconnection and then reconnection at every other time interval continues thereafter. The second run did not include any disconnections or reconnections, and serves as a benchmark for comparison with the evolved topology of the first run. To highlight the effects of peer disconnections and reconnections on the simulated Gnutella network, several statistics were chosen for graphical display.

Figure 6.25 illustrates the number of active or connected peers in the dynamic network for each time interval. Figure 6.26 illustrates the number of protocol-level links between peers in the dynamic network for each time interval. Figures 6.27 and 6.28 compare the number of concurrent message transmissions for both the dynamic

and static simulation runs. Finally, Figure 6.29 illustrates the effects of a dynamic topology on the amount of time taken for queries to complete.

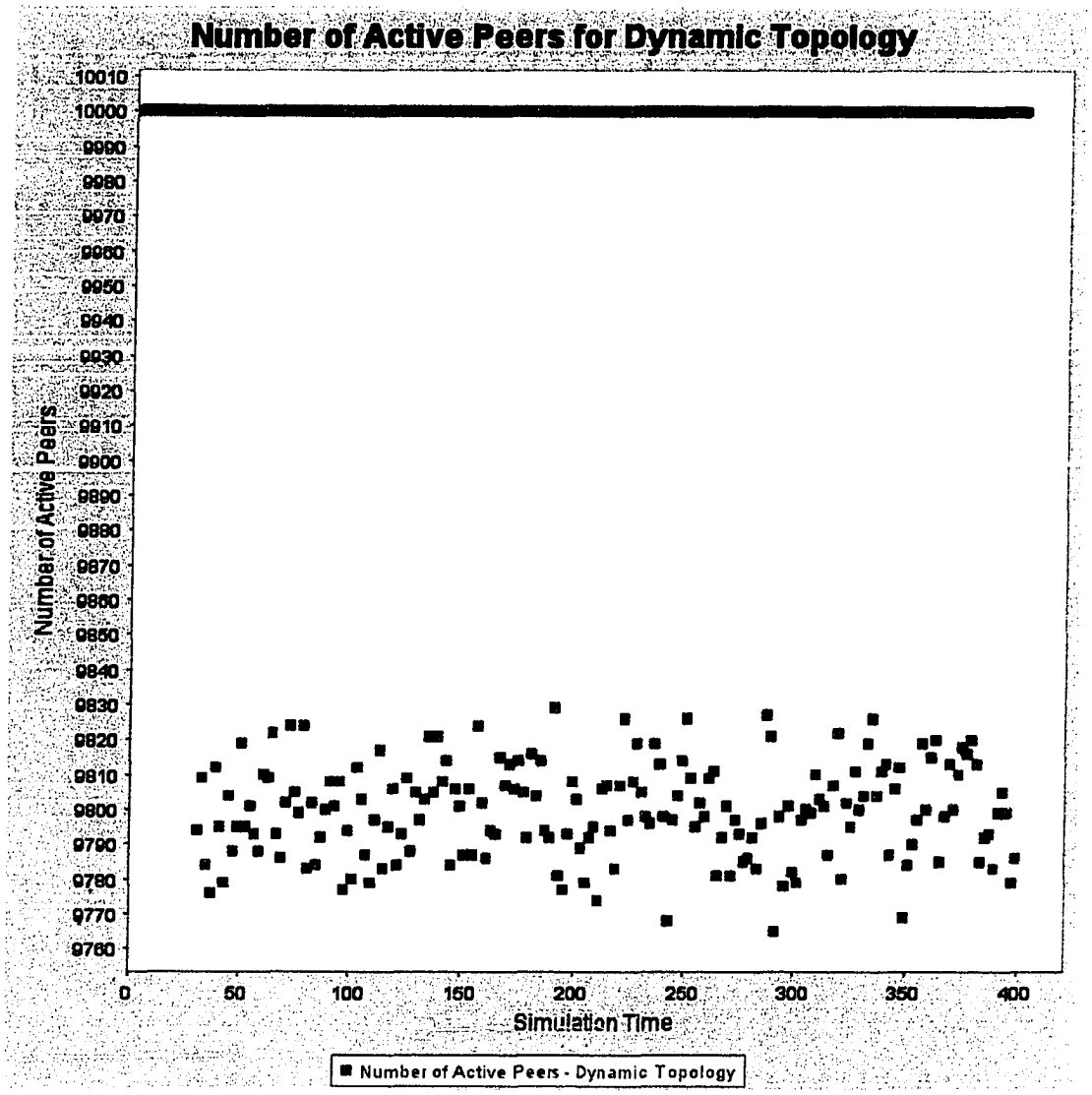

Figure 6.25 Number of Active Peers in a Dynamic Gnutella Network

Figure 6.25 illustrates the number of active peers for the run with a dynamic topology. At alternating time intervals, random selected groups of approximately 200

peers were disconnected, and then reconnected to the network. Notice that the number of active peers oscillates between approximately 9,800 and the maximum value of 10,000.

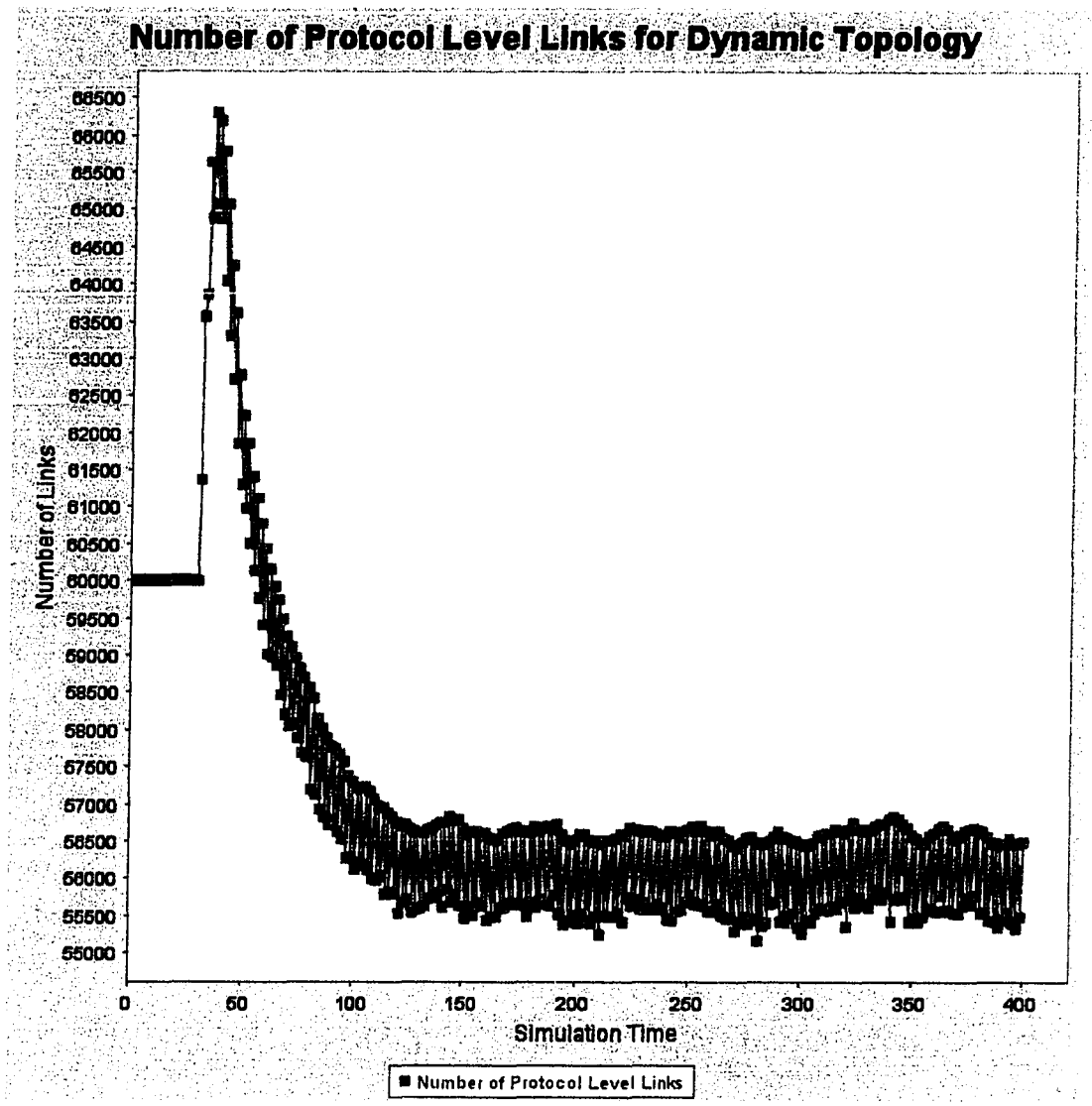

Figure 6.26 Number of Protocol Level Links in a Dynamic Gnutella Network

Figure 6.26 illustrates the change in the number of protocol-level links between peers for the run with a dynamic topology. The initial rise in the number of protocol-level links is the result of the maximum number of links per peer specified as a parameter for the simulation run. Many peers in the initial topology had a small number of links, but as they disconnect and reconnect, they form more links to other peers than they had prior to disconnection. After a few time intervals, this growth in the total number of links in the network is counteracted by highly connected peers disconnecting, and forming fewer links upon reconnection.

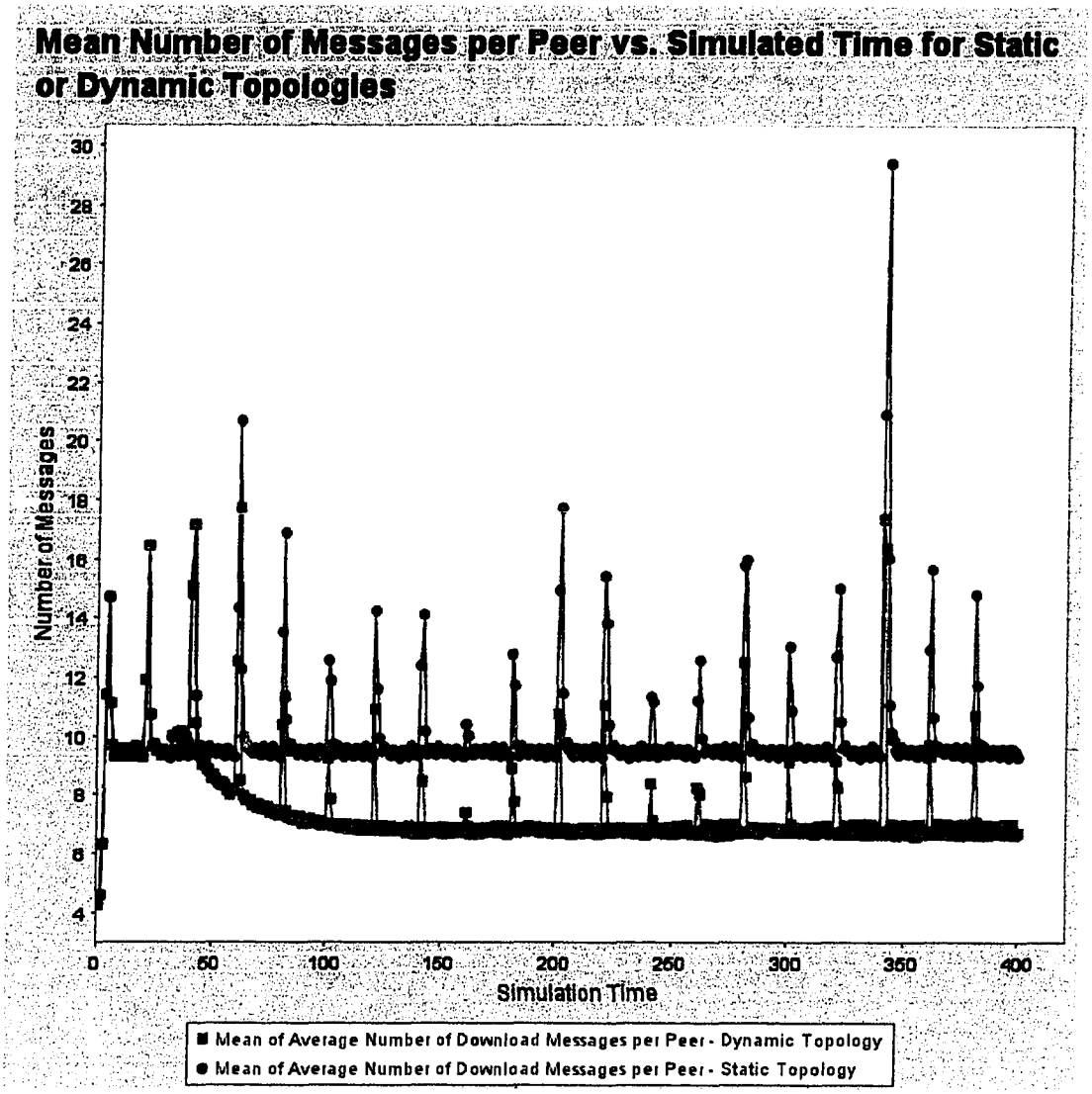

Figure 6.27 Mean Number of Messages per Peer for Static or Dynamic Network Topologies

Figure 6.27 illustrates the average number of concurrent message transmissions per peer for both the static and dynamic simulation runs. Because the dynamic topology eventually contains fewer and more evenly distributed protocol-level links, peers are transmitting and receiving fewer messages. This decrease in the

number of transmissions and receptions results in less bandwidth being consumed, making for faster message transmission, and significantly less traffic on the network as a whole.

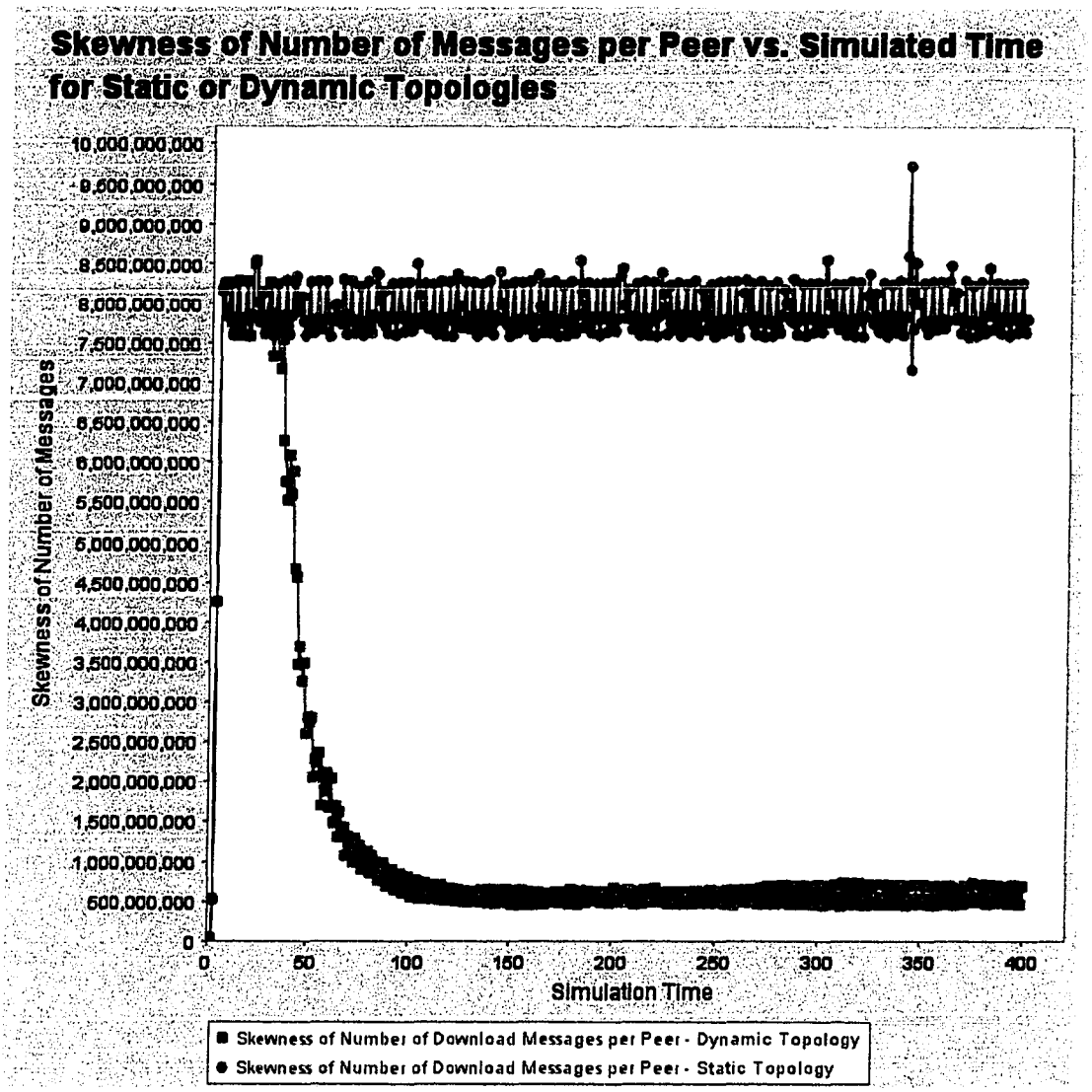

Figure 6.28 Skewness of Number of Messages per Peer for Static or Dynamic Network Topologies

Figure 6.28 illustrates the skewness of the number of messages being transmitted to peers for both the static and dynamic topology. A higher positive skewness value signifies that more peers have a significantly higher number of messages than the mean. Because the dynamic topology eventually equalizes the number of protocol-level links for all peers, bottlenecks for message transmission are removed, resulting in more evenly distributed message traffic across the network, and a lower positive skewness.

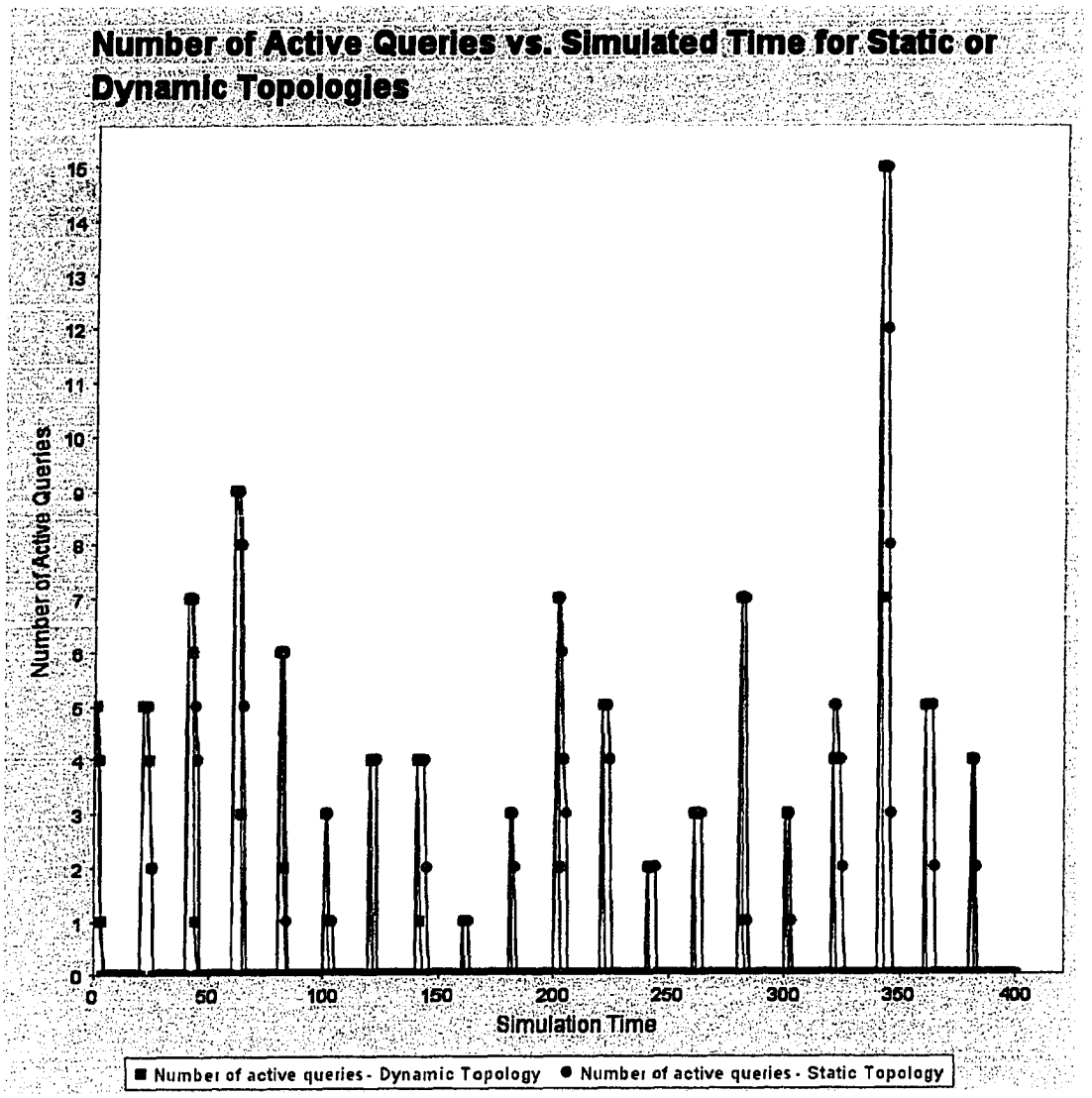

Figure 6.29 Number of Active Queries for Static or Dynamic Network Topologies

Figure 6.29 illustrates the effects of a static or dynamic topology on query completion time. As seen in Figures 6.27 and 6.28, the dynamically evolved Gnutella network topology results in fewer peers carrying a high amount of message traffic. As

expected, this leads to faster transmission of query related messages, which in turn leads to faster query completion times.

6.6 Large Topology

All of the results presented so far have been obtained from simulation runs with topologies with the relatively modest size of 10,000 peers. This section shows the results for simulation runs with a larger network topology of 100,000 peers. The purpose of this section is twofold: first, to validate that the simulator can handle large topologies, and second, to investigate the effects of differing maximum query hop counts.

As described in the introduction to Chapter 5, the horizon of query transmissions in the Gnutella network is limited by a maximum hop count value which is typically set to 7. With the topologies of 10,000 peers described in previous sections, virtually all peers are reached by queries even with modest maximum hop count values of 2 or 3. Even with a network topology of 100,000 peers, a limitation is that all peers are within a 7 hop radius of each other and thus queries will reach every peer. Therefore, to investigate the effects of differing maximum query hop counts, two simulation runs were performed with topologies of 100,000 peers, and differing maximum hop count values of 4 and 5 respectively.

Table 6.13 lists the *common* parameters used for these simulation runs, and Table 6.14 lists the *different* parameters used. For an explanation of the parameters and a description of how they were provided to the simulator, refer to Appendix C.

Table 6.13 Common Parameters Used for Large Topology Simulation Runs

Parameter	Value
Random Number Seed	51957253545678
Time to Stop (Simulated Time Units)	100.0
Number of Peers	100,000
Average Number of Links per Peer	10
Topological Structure of Inter-Peer Links	BRITE Waxman (Random)
Maximum Upload Bandwidth per Peer	15,000
Maximum Download Bandwidth per Peer	15,000
Number of Content Records in Repository	15,000
Average Number of Content Records per Peer	10
Percentage of Pong Caching Gnutella Peers	97
Percentage of Standard Gnutella Peers	3
Percentage of Peers Using Flow Control	100
Max Pongs per Hop Count in Pong Cache	5
Cached Pongs Returned per Ping	8
Cached Ping Interval (Simulated Time Units)	3.0
Standard Ping Interval (Simulated Time Units)	60.0
Max Messages per Connection	10
Max Queued Query Replies per Connection	5
Max Queued Queries per Connection	4
Max Queued Pongs per Connection	3
Max Queued Pings per Connection	2
Query Generation Interval (Simulated Time Units)	10.0
Query Generation Rate (Poisson) per Generation Interval	10

Table 6.14 Different Parameters Used for Large Topology Simulation Runs

Parameter	Simulation Run 15	Simulation Run 16
Max Hop Count for Queries and Pings	4	5

Table 6.14 shows that the first simulation run with a large topology used a maximum hop count of 4, while the second used a maximum hop count of 5. To highlight the effects of different maximum hop counts on a large Gnutella network, several statistics were chosen for graphical display.

Figure 6.30 compares the average number of peers reached per query for the two simulation runs. Figure 6.31 illustrates the average number of concurrent message transmissions per peer for these runs. Finally, Figures 6.32 and 6.33 illustrate respectively, the differences in the number of query replies returned, and the differences in the number of query replies queued.

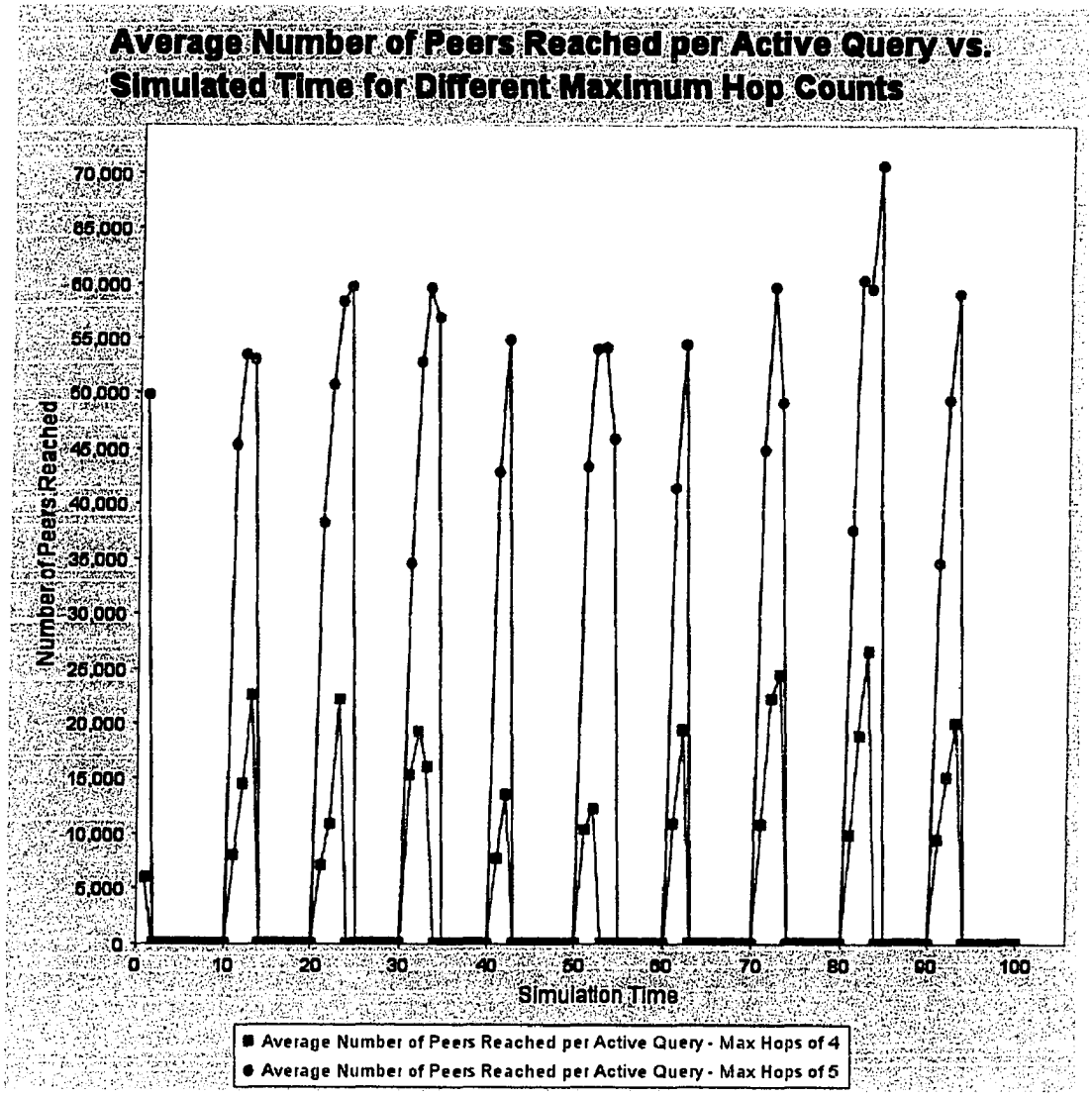

Figure 6.30 Average Number of Peers Reached per Query for Different Maximum Hop Counts

Figure 6.30 illustrates the effects of different maximum hop count values on the number of peers reached by queries. With a conservative maximum hop count of 4, queries reach approximately one quarter of the peers in the network. A modest increase in the maximum hop count to a value of 5 results in a significant increase in

the number of peers reached per query, more than half of the peers in the network are now reached.

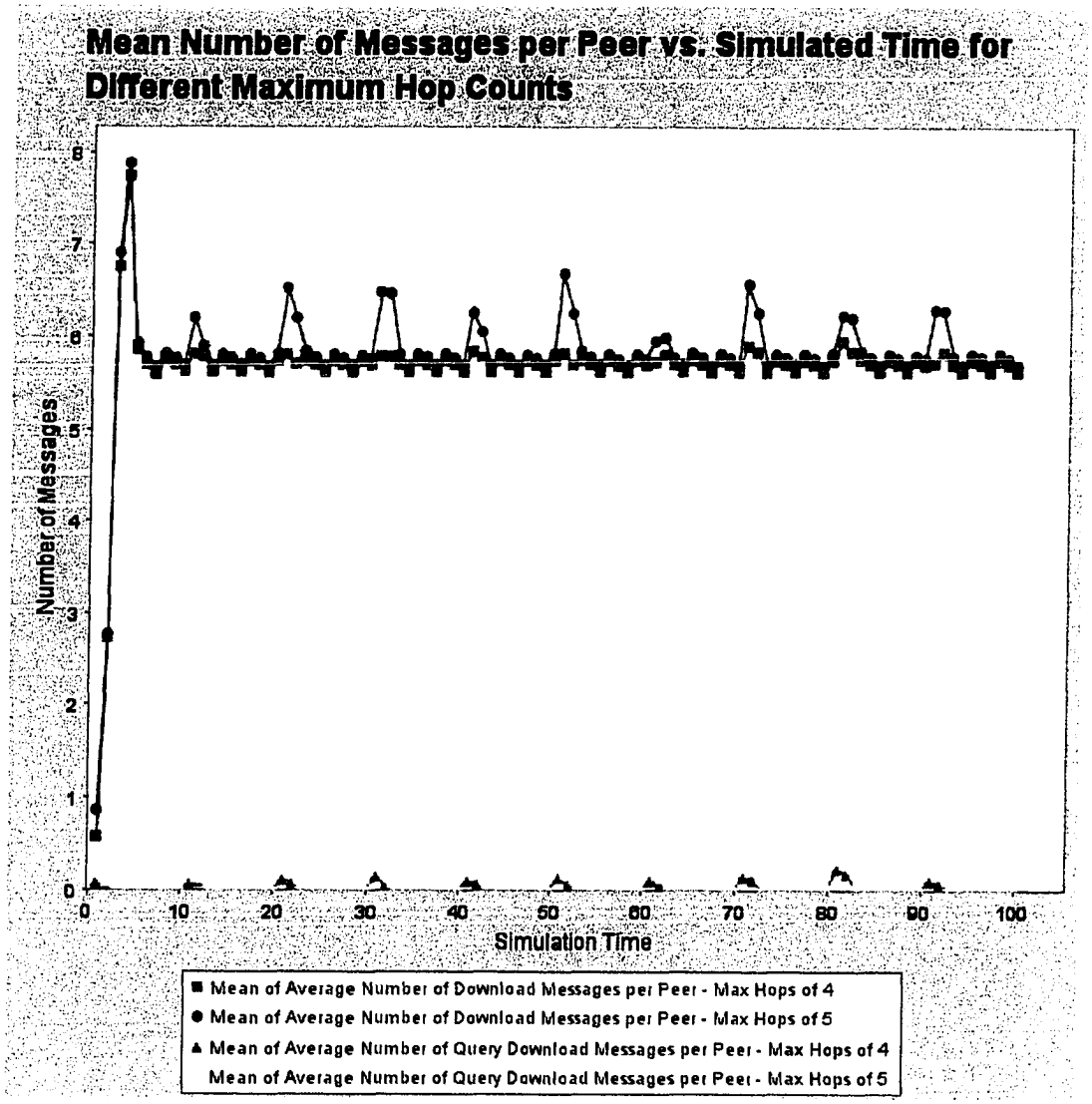

Figure 6.31 Average Number of Messages per Peer for Different Maximum Hop Counts

Figure 6.31 illustrates the effects of different maximum hop count values on the number of concurrent message transmissions in the large network topology. As

expected, a higher maximum hop count leads to a higher number of concurrent query message transmissions, since more peers are being reached by the queries.

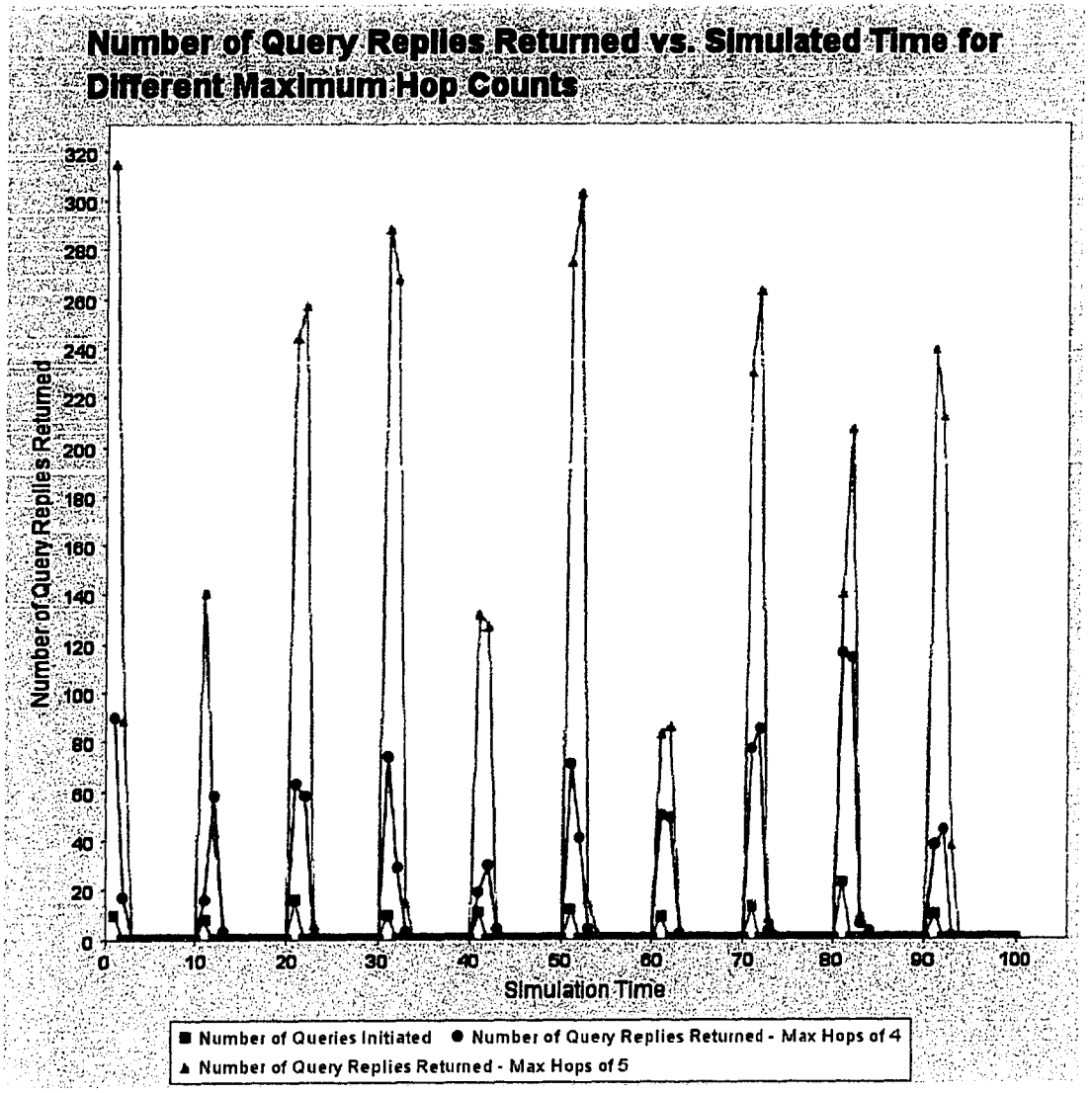

Figure 6.32 Number of Query Replies Returned for Different Maximum Hop Counts

Figure 6.32 illustrates the number of query replies returned to peers initiating queries. An increased maximum hop count value results in more peers being reached by queries, which in turn results in more query replies.

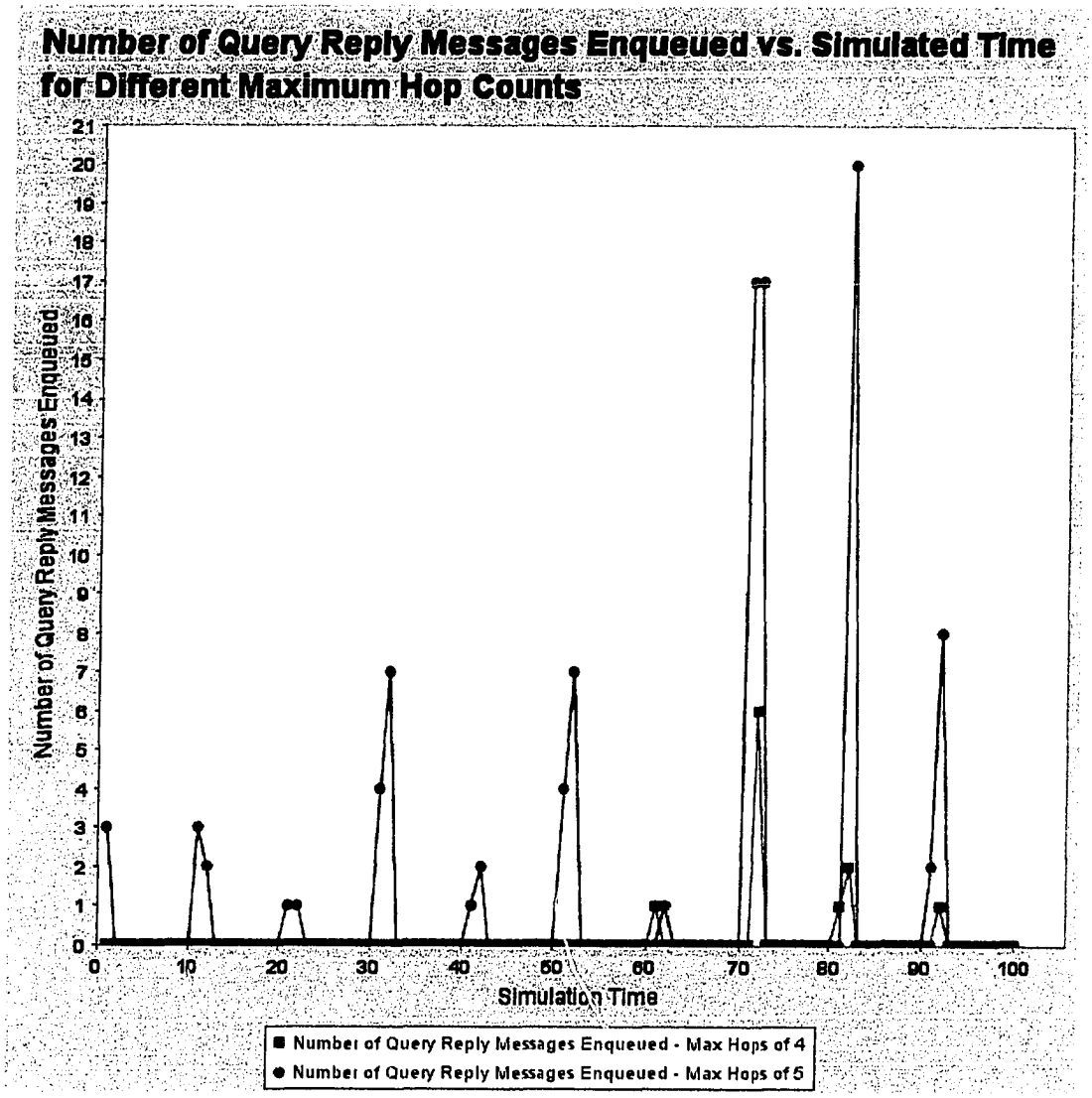

Figure 6.33 Number of Enqueued Query Reply Message for Different Maximum Hop Counts

Figure 6.33 illustrates the total number of query reply messages that were enqueued in the flow control queue for each of the simulation runs. As seen in Figure 6.32, an increased maximum hop count value results in more query replies being returned. This in turn leads to a higher amount of query traffic at peers closest to the query initiator, causing a larger number of query reply messages to be enqueued prior to transmission, as seen in this figure.

6.7 Summary

This chapter has presented the results of 16 simulation runs with the Gnutella protocol. The majority of these runs were performed with topologies of 10,000 peers, while two of them were performed with topologies of 100,000 peers. The benchmark machine used was a 1.9 GHz AMD system with 1024 MB of memory. The time taken to perform each of the simulation runs is listed in Appendix D. In general terms, these results demonstrate that the simulator exhibits the functionality, scalability and extensibility that were the primary objectives of this thesis.

More specifically, the results illustrate that the simulator is successful in modeling and evaluating a diverse set of scenarios. Namely:

- Different bandwidth distributions – As the amount of bandwidth available to individual peers is increased or decreased, message transmission times are seen to vary correspondingly.

- Heterogeneous protocols – By varying the proportions of peers which are running different protocols, performance enhancement or degradation of the network is observed
- Varying traffic densities – By varying the rate of occurrence of peer behaviour which causes messages to be transmitted, the performance of a protocol under varying degrees of message traffic is seen.
- Flow control - As message traffic increases, the role of flow control in preventing an unbounded growth in the number of message transmissions is observed.
- Dynamic or evolving topologies – As peers disconnect and reconnect to the network, properties of the evolved topology and the performance of the protocol are evaluated.
- Topologies of varying size – As protocols are run on topologies of larger size, their performance is evaluated.

Chapter 7

Conclusions and Future Work

This chapter summarizes the contributions made by this thesis, and suggests areas or directions for future work.

7.1 Conclusion

This thesis has presented a contribution to the study of peer-to-peer networks. In particular, it has developed a comprehensive, scalable and extensible simulation model for such networks, and demonstrated the application of the simulator to the Gnutella protocol. As stated in the introduction, the development of such a simulator required careful consideration of several factors. Over and above its scalability and extensibility, it was required to model other crucial aspects of real networks, namely the behaviour of peers participating in the network, and the consumption of network resources as communication traffic increases. In addition, it was required to provide accurate statistics detailing the resources consumed as peers engaged in various behaviours.

The major contributions of the thesis are summarized as follows:

- **A comprehensive review** – As a preamble to the development of the simulator, Chapter 2 presents a comprehensive review of methods for generating network topologies for use in simulation, and provides a review of existing peer-to-peer network simulators. The material presented in this chapter was the basis of a presentation at the *5th International Conference on Internet Computing* [HD04].
- **Scalability** – The event-processing components of the simulator were designed to maximize scalability both in terms of storage and performance, as discussed in Chapter 4. The simulator is in principle able to simulate peer-to-peer networks with up to 1 million peers on a workstation with 16 GB of memory. In Chapter 5, the major factors limiting the size of simulated Gnutella networks are identified and the simulator optimised to accommodate them.
- **Extensibility** – The core architecture of the simulator consists of a set of components which can be extended to add support for arbitrary peer-to-peer protocols (Chapter 3 & 4). This design was validated by simulation runs using various versions of the Gnutella protocol (Chapter 5).
- **Peer behaviour** – The simulator can be fed at regular or continuous time

intervals with simulated events corresponding to any of the behaviours supported by a protocol implementation. Thus, with the Gnutella implementation, there is support for queries, query replies, ping, pongs, connections and disconnections.

- **Traffic modeling** – The Minimum Equal Share bandwidth model is developed as a means for modeling the slow-down in message transmission times as traffic density increases. It also permits the modelling of heterogeneous networks where peers of different intrinsic bandwidths are present simultaneously.
- **Measurement** – The simulator allows for the collection and reporting of a variety of statistics pertaining to each type of message or behaviour supported by a protocol implementation. A graphical interface was developed to simplify the selection and presentation of statistical information collected in particular simulation runs.
- **Gnutella evaluation** – In Chapter 6, the simulator is validated by an implementation of multiple versions of the popular Gnutella peer-to-peer protocol. This showcases the simulator's ability to simulate networks with heterogeneous protocols, and illustrates the need for protocols to curb the number of transmitted messages using flow control or other forms of message queues. The results obtained from the Gnutella simulations confirm that Pong

Caching increases the performance of the Gnutella network. The results also illustrate that Gnutella suffers a decrease in performance when peers have equal available bandwidth, and the topology of inter-peer links exhibits a power law distribution.

In general terms, the simulator was exceedingly successful in meeting all of the objectives set for it. Its modular design allowed the focus of Gnutella protocol implementation to remain fixed on the specifics of handling Gnutella messages, without being distracted by low level details pertaining to communication or statistics collection.

7.2 Future Work

The comprehensive nature of the simulator means that it provides a fertile environment for further work.

The Minimum Equal Share bandwidth model (Section 3.4.4.1) utilized by the simulator is effective for modeling the slow-down of message transmissions as the number of messages increases. However, because this model was developed to be scalable, it is fairly simplistic, and can result in under-utilization of bandwidth available to peers. More complex models exist which would correct this under-utilization, but they would require processing every message in the simulator every

time a message is added or removed, drastically limiting the scalability. An interesting avenue for future work would be the development of a bandwidth model which provides better bandwidth utilization per peer, without requiring the processing of an excessive number of messages.

The current implementation of the simulator only allows for it to be run on a single workstation. An interesting avenue of future work would be to modify the simulator to operate in a distributed manner, utilizing multiple workstations or processors. This could increase the system memory and processing power available to the simulator, but would require careful management of messages between peers residing on different physical workstations or processors.

The focus of many peer-to-peer protocols is the organization of a network for efficient location and retrieval of content. In some cases, the distribution of content across the network is purely random. In other cases, the distribution of content is more structured, with higher replication of more popular content (See Section 2.4.3.1). The simulator's content repository and distributor interfaces allow for arbitrary schemes for content distribution to peers. This functionality could be used to compare the performance of different peer-to-peer protocols in terms of minimizing resource utilization for the location and retrieval of content distributed in different ways.

The simulator supports dynamic network topologies where peers can connect or disconnect at any time. Different peer-to-peer protocols specify different rules for

how a peer should manage its connections, resulting in networks with varying topological structures. The interface of the simulator's protocol class would allow for straightforward analysis of evolved protocol-level topologies, either through classes provided to the statistics reporter, or by saving the topology to a file for analysis by an external tool.

The Gnutella plug-in used to validate the simulator provides implementations of the original or standard versions of the Gnutella protocol, as well as enhanced versions which include pong caching and flow control. One of the more recent enhancements made to the Gnutella protocol is the concept of UltraPeers [Roh02a] [SR02]. UltraPeers are Gnutella peers with high bandwidth and uptimes, which act as proxies through which peers with lower available resources communicate with the network. The features of the simulator would make it possible to extend the Gnutella plug-in with an UltraPeer version of the protocol, allowing for an exploration of the resulting performance increases of the Gnutella network.

Another aspect of the Gnutella protocol that would be interesting to study through simulation is its use of host caches. Host caches are lists of peer addresses maintained by web servers to help new peers connect to the network. If a new peer joining the network does not have access to the addresses of peers already connected to the network, it can retrieve addresses from a host cache. It is known that the use of host caches in Gnutella has a considerable impact on the evolved topology of the

network, and thus an impact on the performance of the network as a whole. To date, the only existing work which studies the effects of host caches [KAD+04] measures only how quickly peers are able to connect to the network, and tracks the number of peers retrieved from host caches with which successful connections were formed. Valuable insight towards improvements to Gnutella's host caching system could be obtained from the simulator.

A final area for future work is the implementation of other peer-to-peer protocols for analysis. As described in Section 4.2, the simulator core was designed to be as generic as possible, leading to a high degree of extensibility, and is therefore an ideal platform for this kind of investigation.

Appendix A

Questionnaire Sent to Authors

The following questionnaire was sent to authors of peer-to-peer simulators for the review of existing work conducted in Chapter 2.

1) What is the maximum numbers of peers that the simulator has been tested with? How well does it run with this many peers (i.e. does it scale well)?

2) What types of initial network topologies are supported? (Ex: Import from file, generated [random, transit-stub, power-law, small-world?], hard-coded [ring, lattice?]) If more complicated generation models were used (ex: power law) please give some details of the method employed.

3) Can peers join and/or leave the network after the initial topology is loaded? If yes, what kind of process(es) are used to model their arrival and departure rates (ex: Poisson Process)? How is the decision made about which peer to disconnect, or where to connect new peers.

4) Is each peer's neighbourhood static or dynamic (i.e. Can peers change which peers they are connected to after joining)?

5) Which general peer 'failures' can be simulated by the simulator? (Ex: sudden disconnection) What other failures can be simulated? (Ex: query & file transfer failures)

6) Can peers perform malicious actions (ex: disrupting queries, modifying files)? If yes, please elaborate.

7) Can some peers index the content of other peers? (i.e. a knowledge base or routing table)

8) What kind of hierarchies can the peers be organized into in the simulation? (Ex: All peers are identical, some peers are Super Nodes)

9) Which protocols are supported by the simulator? (Ex: Gnutella, Chord, Freenet)

10) Are dynamic protocols supported by the simulator (i.e. protocol adapts or evolves during simulation)?

11) Can new protocols be easily integrated into the simulator? If yes, what steps are required to integrate a new protocol? (Ex: Write a new class, or provide an XML file to load the protocol from)

12) Does the simulator support more than just file sharing? If yes, list the applications.

13) How are the links between peers represented in the simulator? (ex: graph object with adjacency list or adjacency matrix).

14) How are messages sent along links between peers? (ex: remote method invocation, local method call, flow-based mode, network socket) Can communication errors be simulated on these links between peers?

15) Does the simulator simulate a network overlay (the simulated peer-to-peer network is overlaid on top of another network)? If yes, describe how this was implemented. (Ex: Bandwidth of links between two peers varies due to traffic unrelated to the overlay)

16) What kind of bandwidth modeling is used by the simulator (i.e. how is bandwidth allocated to links, and assigned or consumed by peers? What activities in the simulator consume bandwidth?)

17) Is the simulator implementation Parallel or Sequential, Single or Multi-Threaded, or Distributed (note that a simulator can be sequential and multi-thread for example)?

18) What programming language was used for the implementation of the simulator? Is the simulator based on an existing simulator / simulation tool (Ex: SimJava, Sim++)?

19) Is the simulator easily portable to a real application (i.e. converted into a real peer-to-peer application that works on the internet)? What steps would be required to make this transition (Ex: Flag at runtime, recompilation, or source code modification)?

20) What types of queries are supported? (Ex: keyword search, key-based search etc...)

21) What kind of query routing is supported? (Ex: broadcast, selective broadcast, key-based)

22) Can queries for non existent items be simulated?

23) Are file transfers supported in the simulation? If yes, can files be downloaded simultaneously from multiple sources?

24) Is file caching supported? (Ex: Copies of a requested file are left on the peers along the path between requester and responder)

25) If file transfers are supported, can files be deleted? Does the simulator handle cases where an attempt is made to download a file that has been deleted since it was returned as a query hit? If peers can index the content of other peers, how are indices affected by file deletions?

26) What functionality does the simulator provide for determining which peers will initiate certain queries and/or initially store certain files? (Ex: Random, Zipf, by Content & Popularity etc...)

27) What kind of simulation approach is used in the simulator? Ex: Activity Scanning approach, Event Scheduling approach. If event scheduling was used, how was the event list implemented?

28) What is the query generation rate per peer (Ex: Uniform, pre-defined, random etc...) How are these queries propagated through the simulator? (Ex: The simulator

processes one query at a time from start to finish, or queries can be initiated at any time from any peer and are processed simultaneously)

29) What kind of queuing is used for messages related to queries and / or file transfers? (Ex: FIFO, None) If queues were used for messages, is a queue associated with each peer, or with each link between two peers?

30) Does the simulator provide the ability to simulate delays incurred by computation performed at peers? (Ex: Peers visited along the path of a message must index into a routing table or decrypt the message)?

31) What statistics are measured / reported by the simulator?

32) What type of visualization is provided of either the topology or the statistics? (ex: graphs or charts are drawn, or the a log file can be saved in a format readable by some visualization tool)

33) What features of the simulator are configurable via the user interface? Is the user interface text based (command line) or a GUI?

34) Are there other features of the simulator that deserve mention that are not covered by this questionnaire? If yes, please elaborate.

35) At present the following Peer-to-peer simulators are being surveyed: 3LS, Anthill, Narses, NeuroGrid, P2PSim (a.k.a. Query-Cycle), PeerSim and SimP^2 (SimP-squared). If you know of a simulator not in this list that should be, please list it below.

Appendix B

Calculation of Memory Usage

This section describes how the memory usage of both the core and Gnutella classes were computed in Tables 4.1 and 5.1.

In Java, each instance of a class requires memory for its *object header* and for each of the *fields* it stores [Rou02]. In general, the fields stored by a class instance can be grouped into three categories: arrays, object references, and primitive variables. Object references and primitive variables have fixed memory requirements. Arrays are slightly more complicated because they have fixed memory requirements, in addition to memory requirements for the object references or primitive variables they contain.

Table B.1 gives a breakdown of the factors contributing to the memory requirements of class instances in Java.

Table B.1 Memory Requirements for Java Objects

Contributing Factor	Memory Requirement (bytes)
Object Headers	8
Arrays (Overhead Only)	12
Object References	4
Primitive Variables	4

Thus, for a Java class whose fields are 3 object references, 5 primitive variables, and 1 array containing 16 object references, the total memory requirement of instances of this class is: Object header (8) + 3 object references (3 x 4) + 5 primitive variables (5 x 4) + 1 array (12) + 16 object references in the array (16 x 4). The total memory required for this Java class is therefore 8 + 12 + 20 + 12 + 64 = 116 bytes.

Appendix C

Simulation Parameters

C.1 Cross Reference to Descriptions of Simulation Parameters

Table C.1 Cross Reference to Simulator Parameter Descriptions

Parameter	Described in Section
Random Number Seed	See Note 1 below
Time to Stop (Simulated Time Units)	2.4.2.1.1
Number of Peers	3.1
Average Number of Links per Peer	See Note 2 below
Maximum Upload Bandwidth per Peer (bytes per Simulated Time Unit)	3.4.4
Maximum Download Bandwidth per Peer (bytes per Simulated Time Unit)	3.4.4
Topological Structure of Inter-Peer Links	4.2.2.1
Number of Content Records in Repository	3.3
Average Number of Content Records per Peer	3.3
Percentage of Pong Caching Gnutella Peers	4.2 & 5.2.2
Percentage of Standard Gnutella Peers	4.2 & 5.2.2
Max Pongs per Hop Count in Pong Cache	5.2.2
Cached Pongs Returned per Ping	5.2.2
Cached Ping Interval (Simulated Time Units)	5.2.2
Standard Ping Interval (Simulated Time Units)	5.2.2
Max Hop Count for Queries and Standard Pings	5
Percentage of Peers Using Flow Control	5.3.1
Max Messages per Connection	5.3.1
Max Queued Query Replies per Connection	5.3.1
Max Queued Queries per Connection	5.3.1
Max Queued Pongs per Connection	5.3.1

Max Queued Pings per Connection	5.3.1
Query Generation Interval (Simulated Time Units)	5.2.1
Query Generation Rate (Poisson) per Generation Interval	5.2.1
Maximum Number of Links for Standard Peers	5.2.3
Maximum Number of Links for Pong Caching Peers	5.2.3
Disconnect Generation Interval (Simulated Time Units)	5.2.3
Disconnect Generation Offset (Simulated Time Units)	5.2.3
Disconnect Generation Rate (Poisson) per Generation Interval	5.2.3
Connect Generation Interval (Simulated Time Units)	5.2.3
Connect Generation Offset (Simulated Time Units)	5.2.3
Connect Generation Rate (Poisson) per Generation Interval	5.2.3

Note 1: The random number seed used to initialize the various random number generators employed by the simulator can be set at run time. This allows for reproducible simulation runs.

Note 2: The average number of links per peer is determined by the topology generator.

C.2 Passing Parameters to the Simulator

The parameters to be used by the simulator for a given run are provided via an XML configuration file. The simulator makes use of a configuration loader class to parse the XML configuration file for parameters. The configuration loader validates the XML using an XML schema file to ensure that it has the format shown in Figure C.1.

```
<simulator timeToStop="" randomSeed="">
  <protocolFactory class="">
  </protocolFactory>
  <topologyGenerator class="">
  </topologyGenerator>
  <bandwidthDistributor class="">
  </bandwidthDistributor>
  <bandwidthModel class="">
  </bandwidthModel>
  <contentRepository class="">
  </contentRepository>
  <contentDistributor class="">
  </contentDistributor>
  <initialActionGenerators>
    <initialActionGenerator class="">
    </initialActionGenerator>
  </initialActionGenerators>
  <continuousActionGenerators>
    <continuousActionGenerator class="" numActionsPerTime=""
     actionGenerationInterval="" actionGenerationOffset="">
    </continuousActionGenerator>
  </continuousActionGenerators>
</simulator>
```

Figure C.1 Format of XML simulator configuration file

Each of the XML elements listed in this configuration file corresponds to one of components of the simulator core which are described in Chapter 3. The exception is that the protocol factory component is described in Section 4.2.

Figure C.2 contains the XML configuration file used for Simulation Run 13 of Chapter 6. Note the presence of the parameter XML elements which allow for arbitrary parameters to be passed to be passed to the components.

```
<?xml version="1.0" encoding="UTF-8"?>
<simulator
 xmlns="http://www.scs.carleton.ca"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance"
 xsd:schemaLocation="http://www.scs.carleton.ca simulatorConfig.xsd"
 timeToStop="400" useTimeAsRandomSeed="false"
```

```
 randomSeed="51957253545678">
  <protocolFactory class="gnutella.GnutellaFactory">
    <parameter name="ratioGnutellaStandard" value="2"/>
    <parameter name="ratioGnutellaWithPongCaching" value="98"/>
    <parameter name="maxLinksForStandard" value="10"/>
    <parameter name="maxLinksForPongCaching" value="12"/>
    <parameter name="standardPingInterval" value="50"/>
    <parameter name="cachedPingInterval" value="4"/>
    <parameter name="maxHops" value="7"/>
    <parameter name="maxPongsPerHopCount" value="5"/>
    <parameter name="pongsPerPing" value="8"/>
    <parameter name="percentUsingFlowControl" value="1"/>
    <parameter name="maxEventsPerConnection" value="12"/>
    <parameter name="maxQueuedQueryReplies" value="6"/>
    <parameter name="maxQueuedQueries" value="5"/>
    <parameter name="maxQueuedPongs" value="4"/>
    <parameter name="maxQueuedPings" value="3"/>
  </protocolFactory>
  <topologyGenerator class="jP2PSim.topology.BriteTopologyGenerator">
    <parameter name="maxPeers" value="10000"/>
    <parameter name="fileName" value="10thousand-6-random.brite"/>
  </topologyGenerator>
  <bandwidthGenerator
   class="jP2PSim.bandwidth.RandomBandwidthDistributor">
    <parameter name="numBandwidthClasses" value="1"/>
    <parameter name="ratio1" value="2"/>
    <parameter name="maxUploadBW1" value="10000"/>
    <parameter name="maxDownloadBW1" value="10000"/>
    <parameter name="ratio2" value="2"/>
    <parameter name="maxUploadBW2" value="4000"/>
    <parameter name="maxDownloadBW2" value="10000"/>
    <parameter name="ratio3" value="2"/>
    <parameter name="maxUploadBW3" value="400"/>
    <parameter name="maxDownloadBW3" value="400"/>
  </bandwidthDistributor>
  <bandwidthModel
   class="jP2PSim.bandwidth.EqualShareConnectionManager"/>
  <contentRepository class="jP2PSim.content.RandomContentRepository">
    <parameter name="maxContent" value="5000"/>
    <parameter name="numContent" value="5000"/>
  </contentRepository>
  <contentDistributor
class="jP2PSim.content.RandomContentDistributor">
    <parameter name="avgContentPerPeer" value="10"/>
  </contentDistributor>
  <initialActionGenerators>
    <initialActionGenerator
```

```
      class="gnutella.GnutellaPingActionGenerator">
    </initialActionGenerator>
  </initialActionGenerators>
  <continuousActionGenerators>
    <continuousActionGenerator
     class="gnutella. GnutellaQueryActionGenerator"
     numActionsPerTime="5" actionGenerationInterval="20"
     actionGenerationOffset="0">
      <parameter name="peerSelectionStrategy"
       value="jP2PSim.RandomPeerSelectionStrategy"/>
      <parameter name="contentSelectionStrategy"
       value="jP2PSim.content.RandomContentSelectionStrategy"/>
    </continuousActionGenerator>
    <continuousActionGenerator
     class="gnutella.GnutellaDisconnectActionGenerator"
     numActionsPerTime="200" actionGenerationInterval="2"
     actionGenerationOffset="30">
      <parameter name="peerWithoutActionSelectionStrategy"
       value="jP2PSim.RandomPeerWithoutActionSelectionStrategy"/>
    </continuousActionGenerator>
    <continuousActionGenerator
     class="gnutella.GnutellaConnectActionGenerator"
     numActionsPerTime="200" actionGenerationInterval="2"
     actionGenerationOffset="31">
      <parameter name="peerWithoutActionSelectionStrategy"
       value="jP2PSim.RandomPeerWithoutActionSelectionStrategy"/>
    </continuousActionGenerator>
  </continuousActionGenerators>
</simulator>
```

Figure C.2 Sample XML simulator configuration file

Appendix D

Time Taken for Simulation Runs

Table D.1 lists the (real) time taken for each of the simulation runs performed for the results presented in Chapter 6.

Table D.1 Time Taken for Simulation Runs of Chapter 6

Run	Description	Total Time Taken
1	High Available Bandwidth	11 minutes, 54 seconds
2	Medium Available Bandwidth	13 minutes, 34 seconds
3	Low Available Bandwidth	15 minutes, 39 seconds
4	0% Standard Gnutella Peers	9 minutes, 52 seconds
5	10% Standard Gnutella Peers	20 minutes, 17 seconds
6	20% Standard Gnutella Peers	58 minutes, 4 seconds
7	Medium Query Traffic	30 minutes, 57 seconds
8	High Query Traffic	102 minutes, 34 seconds
9	Flow Control, High Available Bandwidth	36 minutes, 51 seconds
10	Flow Control, Low Available Bandwidth	7 minutes, 2 seconds
11	No Flow Control, Low Available Bandwidth	29 minutes, 50 seconds
12	Power Law Topology	25 minutes, 51 seconds
13	Dynamic Topology, Long Run	23 minutes, 43 seconds
14	Static Topology, Long Run	24 minutes, 30 seconds
15	Large Topology, Low Max Hop Count	43 minutes, 22 seconds
16	Large Topology, Medium Max Hop Count	51 minutes, 43 seconds

Bibliography

[AB00] R. Albert and A.-L. Barabasi. Topology of Evolving Networks: Local Events and Universality. *Physical Review Letters*, 85(24):5234-5237, 2000.

[ACL00] W. Aiello, F. Chung and L. Lu, A random graph model for massive graphs, *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing*, (2000) 171-180.

[Ada] Lada A. Adamic, Zipf, Power-law, Pareto – a ranking tutorial, http://ginger.hpl.hp.com/shl/papers/ranking/ranking.html, 2004

[BA99] A. Barabasi and R. Albert. Emergence of scaling in random networks. *Science*, 286:509, October 1999.

[BBC02] Napster Faces Liquidation, http://news.bbc.co.uk/1/hi/business/2234947.stm, September 2002

[Bis] The BISON (Biology-Inspired techniques for Self-Organization in dynamic Networks) project website. http://www.cs.unibo.it/bison/, 2004

[BMM02] O. Babaoglu, H. Meling, and A. Montresor. *Anthill: A Framework for the Development of Agent-Based Peer-to-Peer Systems.* In Proc. of the 22th Int. Conf. on Distributed Computing Systems, Vienna, Austria, July 2002.

[BNC00] Jerry Banks, Barry Nelson, and John Carson. *Discrete-Event System Simulation.* 3rd Edition, Prentice Hall, 2000.

[BT02] Tian Bu and Don Towsely. On Distinguishing between Internet Power Law Topology Generators, *Proceeedings of IEEE INFOCOM '02*, 2002.

[CDH+02] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Topology-aware routing in structured peer-to-peer overlay networks. Technical Report MSR-TR-2002-82, Microsoft Research, 2002.

[CDZ97] K. L. Calvert, M. B. Doar, and E. W. Zegura. "Modeling Internet Topology." IEEE Communications Magazine 35, 6 June 1997.

[Cho] The Chord Project Website. http://www.pdos.lcs.mit.edu/chord/, 2004

[CSW01] I. Clarke, O. Sandberg, B. Wiley, and T.W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In Proc. International Workshop on Design Issues in Anonymity and Unobservability, volume 2009 of LNCS, p. 46-66. Springer-Verlag, 2001.

[Clip2-00] DSS group. Gnutella: To the Bandwidth Barrier and Beyond, http://lambda.cs.yale.edu/cs425/doc/gnutella.html, 2004.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*, The MIT Press, 1st edition, 1990.

[CRB03] Yati Chawathe, Sylvia Ratnasamy, Lee Breslau, "Making Gnutella-like P2P Systems Scalable", Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM'03), p. 407-418, 2003.

[Doa96] M. Doar, "A Better Model for Generating Test Networks," in Proceeding of IEEE Global Telecommunications Conference (GLOBECOM), November 1996.

[ER60] P. Erdos and A. Renyi. On the Evolution of Random Graphs, Mat Kutato Int. Ko zl, vol. 5, no. 17, pp. 17-60, 1960.

[Eve79] Shimon S. Even. *Graph Algorithms*. Computer Software Engineering Series. Computer Science Press, Rockville, MA, 1979.

[FFF99] Michalis Faloutsos, Petros Faloutsos and Christos Faloutsos. On Power-Law Relationships of the Internet Topology. In SIGCOMM, pages 251-262, 1999.

[Fis03] A. Fisk, *Gnutella Ultrapeer Query Routing v0.1*. May 2003, http://f1.grp.yahoofs.com/Proposals/search/ultrapeer_qrp.html, 2005

[Free] Freenet – The Free Network Projcet, http://freenet.sourceforge.net/, 2005

[GB02] TJ Giuli and M. Baker. *Narses: A Scalable, Flow-Based Network Simulator*. Technical Report cs.PF/0211024, Computer Science Department, Stanford University, Stanford, CA, USA, Nov 2002.

[GHJ+95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[GIFT] BerliOS Developer: Project Info – giFT FastrTrack http://developer.berlios.de/projects/gift-fasttrack/, 2003

[GKL+03] Thomer M. Gil, Frans Kaashoek, Jinyang Li, Robert Morris, Jeremy Stribling. *p2psim: A simulator for peer-to-peer protocols*. To appear. 2003.

[Gnu] OSMB, LLC, Gnutella.com http://www.gnutella.com, 2005

[HAR+03] Q. He, M. Ammar, G. Riley, H. Raj, R. Fujimoto, *Mapping Peer Behavior to Packet-level Details: A Framework for Packet-level Simulation of Peer-to-Peer Systems*. In Proc. MASCOTS 2003.

[HD04] J. Harris, D. Deugo, *Towards a Peer-to-Peer Simulator*, in *Proceedings of the 5th International Conference on Internet Computing* (IC'04), 2004.

[Ile02] Michael Iles, *Adaptive Resource Location in a Peer-to-Peer Network*, Master's Thesis, Carleton University, 2002.

[JAB01] M. Jovanović, F. Annexstein, and K. Berman: *Modeling peer-to-peer network topologies through "small-world" models and power laws*, IX Telecommunications Forum TELFOR 2001, Belgrade

[JCJ00] C. Jin, Q. Chen, and S. Jamin. Inet: Internet Topology Generator. Tech. Rep. CSE-TR-433-00, EECS Department, University of Michigan, 2000.

[Jfr05] *JFreeChart Website*, http://www.jfree.org/jfreechart/, 2005

[JMB03] M. Jelasity, A. Montresor, O. Babaoglu, *A Modular Paradigm for Building Self-Organizing Peer-to-Peer Applications*, To appear in Proceedings of ESOA03: International Workshop on Engineering Self-Organising Applications, November 2003.

[Jos01a] S. Joseph. *Adaptive Routing in Distributed Decentralized Systems: NeuroGrid, Gnutella and Freenet.* In proceedings of the workshop on Infrastructure for Agents, MAS, and Scalable MAS, at Autonomous Agents 2001 Montreal, Canada. 2001.

[Jos01b] S. Joseph, *NeuroGrid Simulation Results*, available online at http://www.neurogrid.net/ng-simulation.html, 2001.

[Jos02] S. Joseph, NeuroGrid: Semantically routing queries in peer-to- peer networks. In: Intl. Workshop on Peer-to-Peer Computing (at Networking 2002).

[Jos03] S. Joseph, *An Extendible Open Source P2P Simulator*. In P2P Journal, available online at http://www.p2pjournal.com/issues/November03.pdf. November 2003.

[Kad] New York University, Kademelia: the optimized peer network, http://kademlia.scs.cs.nyu.edu/, 2005

[KAD+04] P. Karbhari, M. Ammar, A. Dhamdhere, H. Raj, G. Riley, and E. Zegura. Bootstrapping in Gnutella: A Measurement Study. In Proceedings of the 5th annual pass & active measurement workshop [PAM2004], 2004.

[Kaz] Sharman Networks Ltd. Kazaa Homepage http://www.kazaa.com, 2005

[KI] K. Kant, R. Iyer, *Modeling and Simulation of Adhoc/P2P Resource Sharing*.

[King01] Brad King. *File Trading in the Crosshairs*, Wired News, available online at http://www.wired.com/news/mp3/0,1285,47296,00.html, 2005

[KM02] T. Klingberg, R. Manfredi. *Gnutella Protocol Development – Gnutella 0.6 Working Draft*, June 2002, http://rfc-gnutella.sourceforge.net/developer/

[Lim] *The Gnutella Protocol Specification v0.4*, http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf, 2005

[Mon01] A. Montresor, *Anthill: a Framework for the Design and the Analysis of Peer-to-Peer Systems*, in 4th European Research Seminar on Advances in Distributed Systems, 2001.

[MB03] A. Montresor, O. Babaoglu. *Biology-Inspired Approaches to Peer-to-Peer Computing in BISON*. In Proceedings of the Third International Conference on Intelligent System Design and Applications, Tulsa, Oklahoma, August 2003.

[MCH03] A. Montresor, G. Di Caro, P. Heegaard, *Architecture of the Simulation Environment*. Technical Report: D11, BISON project, University of Bologna. 2003.

[Min01] N. Minar, *Architecture for Overlay Networks*, http://www.openp2p.com/pub/a/p2p/2001/12/14/topologies_one.html

[MMB00] A. Medina, I. Matta, and J. Byers. On the Origin of Power Laws in Internet Topologies. Computer Communication Review, 30(2):18--28, 2000.

[Nap] Napster Inc. Napster Homepage http://www.napster.com, 2004

[Neu] Neurogrid P2P search, http://www.neurogrid.net, 2005

[NS] Ns Project, http://www.isi.edu/nsnam/ns/, 2005

[NW99] M. E. J. Newman and D. J. Watts. *Renormalization group analysis of the small-world network model.* Phys. Lett. A 263 (1999) 341.

[Ora01] Andy Oram, editor. *Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology.* O'reilly, Sebastopol, CA, March 2001.

[Pap84] A. Papoulis, *Probability, Random Variables, and Stochastic Processes*, 2nd Edition, McGraw-Hill, pp. 145-149, 1984.

[Pas] The Pastry project website. http://research.microsoft.com/~antr/Pastry/, 2004

[PGr] The P-Grid Consortium, P-Grid the Grid of Peers, http://www.p-grid.org/, 2004

[Pru02] Scarlet Pruitt, *Universities tapped to build secure 'Net*, Network World Fusion, September 2002, available online at http://www.nwfusion.com/news/2002/0925iris.html

[RD01] A. Rowstron and P. Druschel, "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-toPeer Systems," Lec. Notes. Comp. Sci., vol. 2218, pp. 329-350, 2001.

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

[RFI02] M. Ripneau, I. Foster, and A. Iamnitchi. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *IEEE Internet Computing Journal*, 6(1), January/February 2002.

[Roh] Christopher Rohrs, *LimeWire: Ping Pong Scheme*. http://ww.limewire.org/developer/pingpong.html, 2005

[Roh02a] Christopher Rohrs, *Query Routing for the Gnutella Network*. Version 1.0, May 16, 2002. http://www.limewire.com/developer/query_routing/keyword%20routing.htm

[Roh02b] Christopher Rohrs, *SACHRIFC: Simple Flow Control for Gnutella – Working Draft*, March 1, 2002. http://www.limewire.com/developer/sachrifc.html

[Ros02] Sheldon M. Ross. *Simulation*, Academic Press, 3rd edition, 2002.

[Rou02] Vladimir Roubtsov. *Java Tip 130: Do You Know Your Data Size?*, JavaWorld, http://www.javaworld.com/javaworld/javatips/jw-javatip130.html, 2005

[SCK03] M. T. Schlosser, T. E. Condie, S. D. Kamvar, *Simulating a P2P File-Sharing Network.* 2003

[SGG02] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. *Proceedings of Multimedia Computing and Networking 2002* (MMCN '02)

[SML+01] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In ACM SIGCOMM, Aug. 2001.

[Sou] *Gnutella Protocol Development – The Web Caching System*, http://rfc-gnutella.sourceforge.net/developer/testing/gwc.html, 2005

[SR02] Anurag Singla, Christopher Rohrs, Lime Wire LLC. *Ultrapeers: Another Step Towards Gnutella Scalability*. Version 1.0, Nov. 26, 2002. http://groups.yahoo.com/group/the_gdf/files/Proposals/Ultrapeer/Ultrapeers_proper_format.html

[Sri03] Kunwadee Sripanidkulchai, *The popularity of Gnutella queries and its implications on scalability*, http://www-2.cs.cmu.edu/~kunwadee/research/p2p/gnutella.html, 2003

[Tap] Tapestry. The Tapestry website. http://www.cs.berkeley.edu/~ravenben/tapestry/, 2004

[TGJ+01] H. Tangmunarunkit, R. Govindan, S. Jamin, S. Shenker, and W. Willinger. Network topologies, power laws and hierarchy, Tech. Rep. TR01-746, Technical Report, University of Southern California, 2001.

[Tin03] N. Ting, *A Generic Peer-to-Peer Network Simulator*, Proceedings of the 2002-2003 Grad Symposiun, Computer Science Department, University of Saskatchewan, 10 April 2003.

[Tys99] J. Tyszer. *Object-Oriented Computer Simulation of Discrete-Event Systems*. Kluwer Academic Publishers. 1999.

[Wax88] B. M. Waxman. Routing of Multipoint Connections. *IEEE Journal of Selected Areas in Communication 6*, 9 (December 1988) 1617-1622.

[Wik04a] Wikipedia – The Free Encyclopedia, *FastTrack*, http://en.wikipedia.org/wiki/FastTrack, 2004

[Wik04b] Wikipedia - The Free Encyclopedia, *Peer-to-Peer*, http://en.wikipedia.org/w/wiki.phtml?title=Peer-to-peer&redirect=no, 2004

[WS98] D. Watts and S. Strogatz. Collective dynamics of small-world networks. *Nature*, 393:440-442, June 1998.

[ZCB96] Ellen W. Zegura, Keneth L. Calvert, and S. Bhattacharjee. How to model an internetwork. In Proceedings of the INFOCOM '96, 1996. 18