ENSC 835: HIGH-PERFORMANCE NETWORKS
CMPT 885: SPECIAL TOPICS: HIGH-PERFORMANCE NETWORKS

Scalability and Robustness of the Gnutella Protocol

Spring 2006

Final course project report

Eman Elghoneimy

http://www.sfu.ca/~eelghone

eelghone@sfu.ca

Table of Contents

1. Abstract

Gnutella is a decentralized peer-to-peer (p2p) network protocol for file sharing. There are two major advantages of decentralized protocols. The first one is scalability, and the second one is robustness to failure. In this project, OPNET simulator is used to model the ping pong message exchange of the Gnutella v0.4 protocol. Scalability is tested by adding a large number of nodes. To test the robustness to failure, I arbitrary fail some nodes to see the effect on the network. Results show that node connectivity has a large effect on number of packets exchanged by the node. Faulty networks exchange less packet traffic yet remain connected.

2. Introduction

2.1 Peer-to-peer Network protocols

In peer-to-peer (p2p) networks, the user is the client and the server at the same time. These decentralized systems are used for file sharing applications over the internet. Each user uploads and downloads files to and from the network at the same time with no centralized control. In other words, "the network is the computer", p2p nodes relate to each other side-by-side within a global computing arena [3].
The original Napster protocol is now non-operational. It featured a centralized directory of users' files, which acted as a single of point of failure in the system and caused a copyright violation problem. Next, the Gnutella protocol [1],[2] replaced Napster with a fully distributed decentralized network. Gnutella 'clients' can be purchased from the bearshare website [4].
Other p2p protocols are KazAa which features group leader nodes, Kademlia [5] protocol which is used for the free 'client' e-mule [6], and BitTorrent which is used by 35% of all traffic on the internet [7].

2.2 The Gnutella Protocol

Gnutella is a p2p protocol for distributed search. Each node in the network is called a "**servent**" and acts as a client and server in the same time. Gnutella network is used to search for files in a decentralized manner. HTTP protocol is used to download files from one servent to another directly (not through the Gnutella network). A servent connects to the network by establishing a connection to another node currently on the network.

2.2.1 Gnutella v0.4

Currently, the stable version of the Gnutella protocol is v0.4. Descriptors are used for communicating data between servents. The following is a listing of descriptors and their format taken directly from [1],[2].

Descriptor	Description
Ping	Used to actively discover hosts on the network. A servent receiving a Ping descriptor is expected to respond with one or more Pong descriptors.
Pong	The response to a Ping. Includes the address of a connected Gnutella servent and information regarding the amount of data it is making available to the network.
Query	The primary mechanism for searching the distributed network. A servent receiving a Query descriptor will respond with a QueryHit if a match is found against its local data set.
QueryHits	The response to a Query. This descriptor provides the recipient with enough information to acquire the data matching the corresponding Query.
Push	A mechanism that allows a firewalled servent to contribute file-based data to the network.

Descriptor format:

Fields	Descriptor ID	Payload Descriptor	TTL	Hops	Payload Length
Byte offset	0...15	16	17	18	19...22

Pong descriptor:

Fields	Port	IP Address	Number of Files Shared	Number of Kilobytes Shared	*Optional Pong Data*
Byte offset	0...1	2...5	6...9	10...13	*14...L-1*

The protocol specifies that the servents have to forward ping and pong messages to other servents in the network. A servent drops a pong message for which it has not received a ping message. In addition, each servent recognizes and drops any duplicate messages. Query and QueryHit are handled in a similar manner. Figure 1 represents an example of ping/pong forwarding in a Gnutella network. Figure 2 illustrates the Query, QueryHit and Push routing.

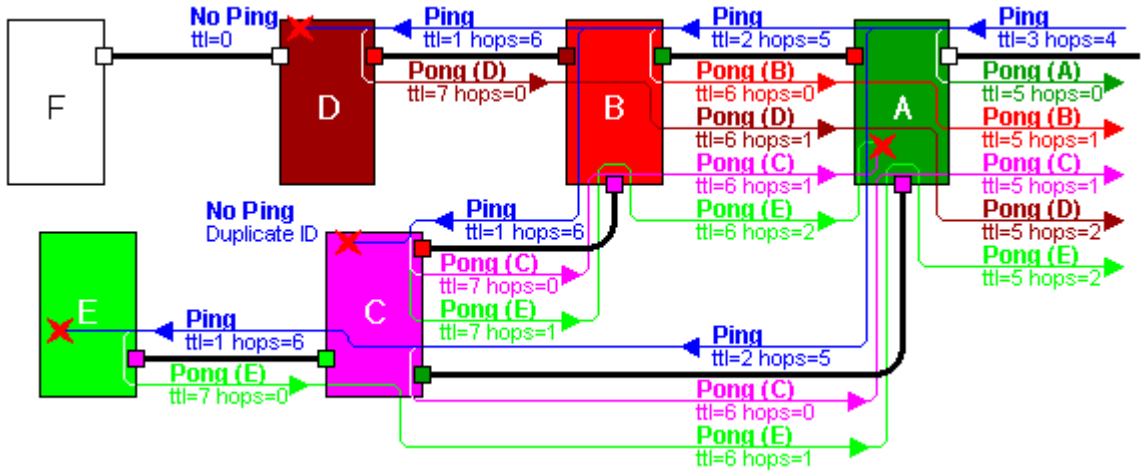

Figure 1. Ping/pong message exchange in a Gnutella network [1].

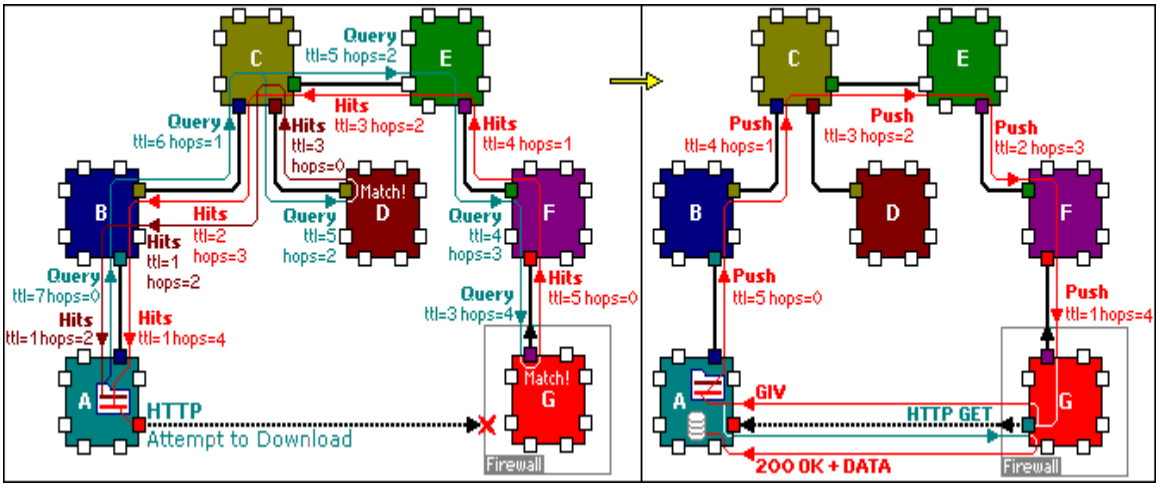

Figure 2. Query/QueryHit/Push routing in a Gnutella network [1].

2.2.2 Gnutella v0.6

In Gnutella v0.4, servents with slow internet connection had problems in the Gnutella net due to the volume of messages nodes were required to forward to other nodes. More structure is added to the network in Gnutella v0.6 [2]. Nodes are of two types: **ultrapeer** and **leaf**.

Ultrapeers are connected to each other. Each ultrapeer forwards to its leaf node only the queries that it can handle. A leaf node can turn into an ultrapeer if it satisfies several conditions such as bandwidth, uptime and operating system. In addition, there must be a need of an ultrapeer in the network which is determined from the ultrapeer handshaking protocol. A handshake protocol is used when a node wants to join the network as well.

Another feature of v0.6 is pong cache. Pong messages are cached at the servent and there is a maximum number of pong messages that each sevent send. Discriptors are the same as those of v0.4.

2.3 Simulation of the Gnutella Protocol

2.3.1 Scope of the project

In this project, I focus on examining the dynamics of joining the Gnutella network using ping and pong messages. I verify the design using a small network. In addition, I test the robustness to failure and the scalability of the protocol.

2.3.2 OPNET Modeler

OPNET is a network modeling software package. It is used to model commercial networks using standard models for workstations, servers, routers, etc. Network, node and process editors are used to specify the functionality of the nodes in a network. OPNET provides a GUI for design, simulation and result viewing. In addition, standard models are provided in the OPNET package, such as TCP and IP models. Members can download (and upload) custom models from the OPNET website (www.opnet.com). This project was developed using OPNET Education version 11.0 under UNIX.

3. Implementation

3.1 Packet format

A custom packet format is implemented as shown in Figure 3. The packet contains the following fields:

- *orig_pkt_id*: original packet id generated by OPNET is used as a unique identifier,
- *message_id*: is 1 for ping and 2 for pong,
- *ttl*: time to live,
- *hops*: number of hops, and
- *sender_objid*: the OPNET object id, used to identify sender of packet

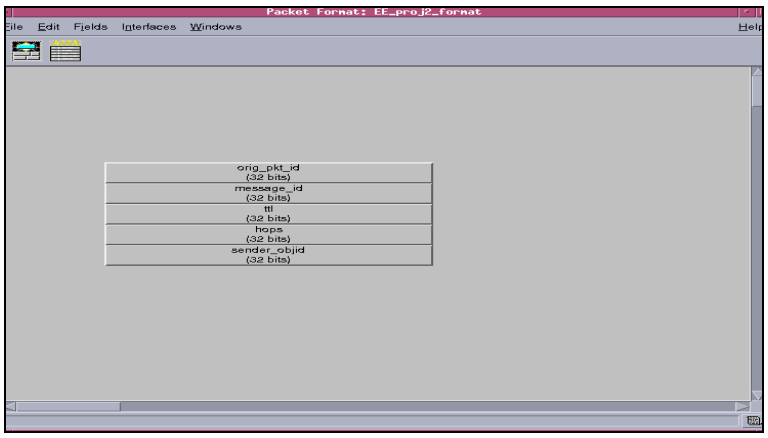

Figure 3. Packet format

3.2 Link model

A duplex link that supports the customized packet is used.

3.3 Peer (servent) node model

There is one type of node used in this model. The peer node, as shown in Figure 4, consists of three point-to-point transmitters, three point-to-point receivers, standard OPNET simple_source and a processing unit. The simple_source generates periodic ping messages. The processing unit sends out those messages through the transmitters to other peers. In addition, the processing unit handles packets from the receivers, which will be explained in the next section.

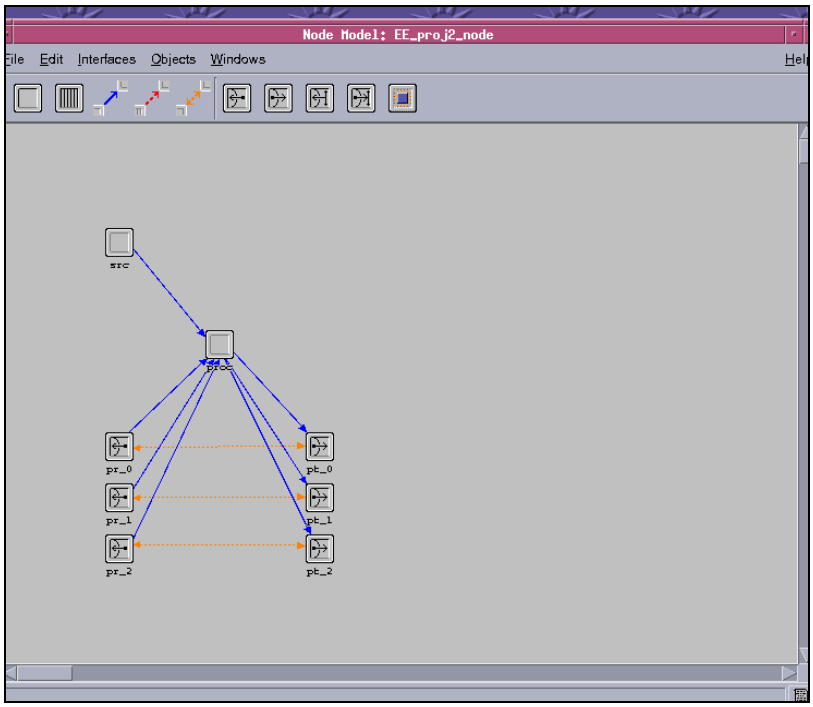

Figure 4. Node model

3.4 Process model

There are three states in the process model: init, idle and procRCV. State variables and statistics handlers are initialized in the **init** state. The init state is a forced state leading unconditionally to the idle state.

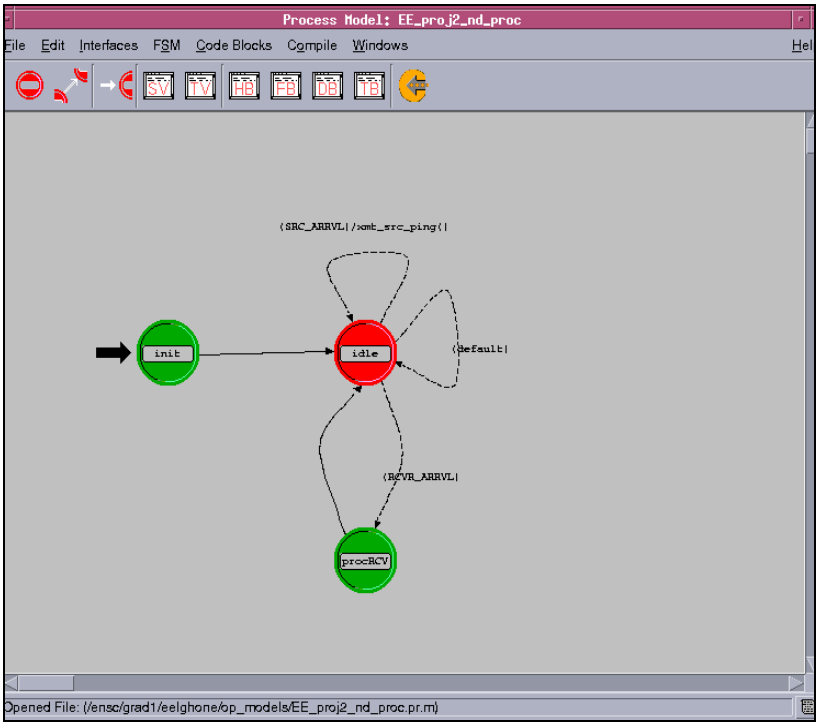

Figure 5. Process model

In the **idle** state, the processing unit is waiting for the next packet doing nothing. Once a packet arrive (from the source, or one of the receivers), a state transition occurs. There are two types of state transitions (in addition to the default one). The first transition is due to a **packet received from the source**. This invokes the xmt_src_ping() function, then back to the idle state. The xmt_src_ping() function reads the packet sent from the src, set the values of the packet properly, save packet information to the ping cache, and transmit the packet to the transmitters.
Another type of state transition is caused by receiving a **packet from one of the receivers**. This causes the transition to another forced state called procRCV. In this state, the received packet is read, if the packet is a **ping** message, the following steps are followed:

1. Reply with a pong message, set ttl = received_hops + 1
2. If this packet is a duplicate packet, discard the packet and exit.
3. Save the packet to the cache.
4. If ttl > 1, forward the message to other peers, set ttl = TTL_INIT

For **pong** messages, the same steps are followed, except for step 1.

3.5 Network models (scenarios)

Seven network models were implemented to verify and simulate the protocol. All network models use the same type of node (section 3.1) and the same type of link (section 3.2). Since each node has 3 transceivers, one node can be connected with up to three other peer nodes.

The implemented scenarios are as shown in figure 6:

1. Ring 3-node model (**ring_3n**): A simple 3-node network with a ring topology. Ring topology is the most complicated form with two transceivers.
2. Ring 4-node model (**ring_4n**): 4-node ring topology network.
3. Ring 4-node model with a faulty node (**faulty_ring_4n**): 4-node ring topology with a failed node representing a node in the network down or disconnected.
4. Fully connected 4-node model (**fullyconn_4n**): Four nodes all connected to each other. Since each node can connect to a maximum of 3 other nodes, this is the largest fully connected topology possible.
5. Ring 15-node model (**ring_15n**): 15 nodes connected in a ring topology.
6. 15-node random mesh model (**mesh_rnd_15n**): 15 nodes connected in a mesh topology. Each node is randomly assigned one to three connections to other nodes.
7. 50-node random mesh model (**mesh_rnd_50n**): 50 nodes connected in a random mesh topology with one to three connections per node.

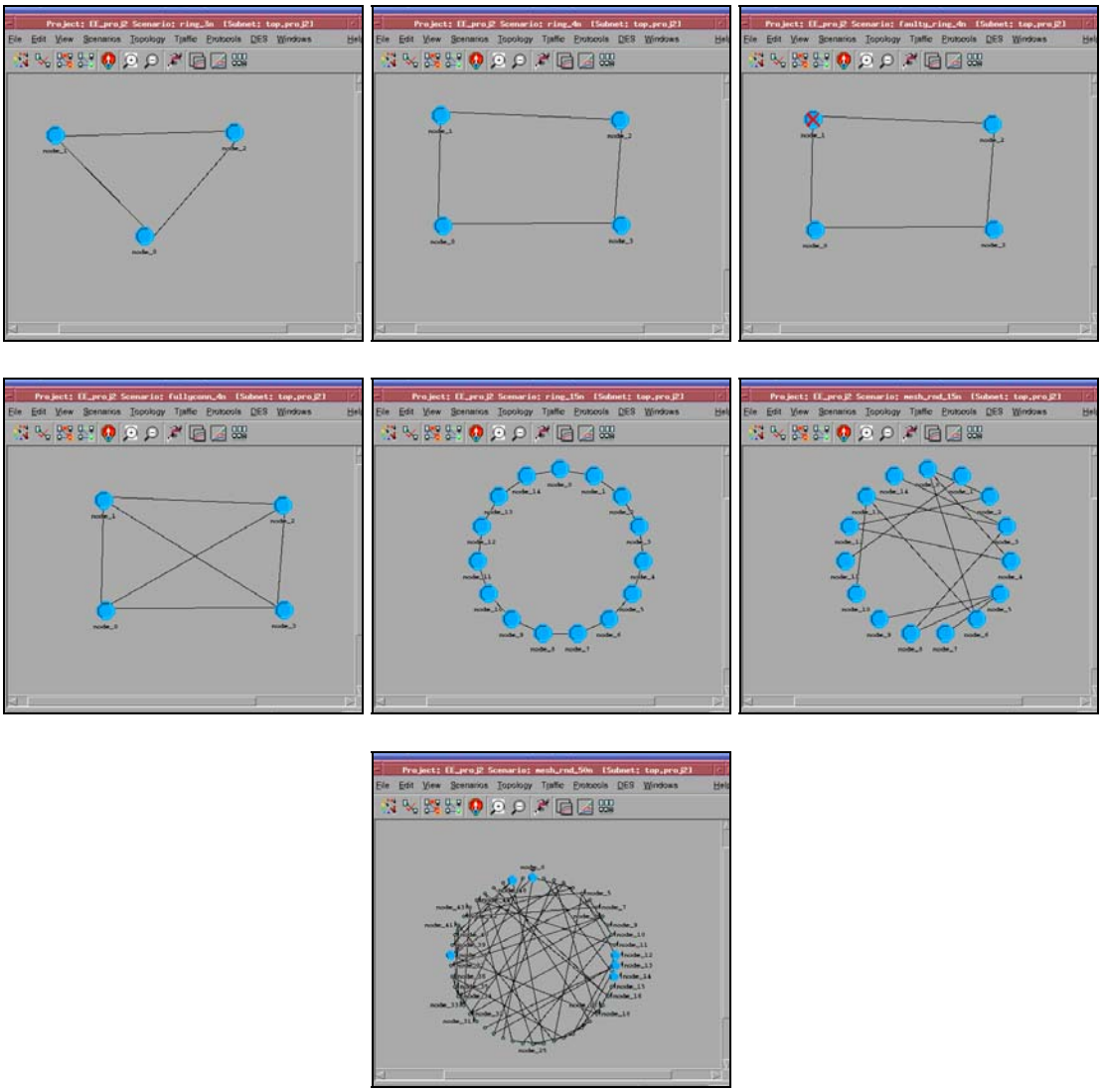

Figure 6. Network model scenarios

4. Simulation

4.1 Model verification

Simple networks were used to verify the model implementation. Debug messages were printed to trace the protocol implementation. Figure 7 shows a sample of the debug output of fullyconn_4n model.

```
node_1 rcv:pk_in 8,pk_out 12
        ...received from rcvr 0: msg_id 1, pkt_id 10, orig_pkt_id 3, ttl 4, hops 1, sender node_3
        ...gen pong to xmtr 0: pkt_id 46, orig_pkt_id 10, ttl 2, hops 0, sender node_1
                Wrote ping to ping cache, ping_ptr=4
        ...fwd ping to xmtr 1: orig_pkt_id 3, ttl 3, hops 2, sender node_3
        ...gen ping to xmtr 2: orig_pkt_id 3, ttl 3, hops 2, sender node_3
node_2 rcv:pk_in 8,pk_out 15
        ...received from rcvr 2: msg_id 1, pkt_id 27, orig_pkt_id 3, ttl 4, hops 1, sender node_3
        ...gen pong to xmtr 2: pkt_id 48, orig_pkt_id 27, ttl 2, hops 0, sender node_2
        ....destroy duplicate ping packet
node_3 rcv:pk_in 8,pk_out 16
        ...received from rcvr 0: msg_id 2, pkt_id 24, orig_pkt_id 3, ttl 1, hops 0, sender node_2
                Wrote pong to pong cache, pong_ptr=1
node_0 rcv:pk_in 8,pk_out 14
        ...received from rcvr 2: msg_id 1, pkt_id 25, orig_pkt_id 3, ttl 4, hops 1, sender node_3
        ...gen pong to xmtr 2: pkt_id 49, orig_pkt_id 25, ttl 2, hops 0, sender node_0
        ....destroy duplicate ping packet
node_3 rcv:pk_in 9,pk_out 16
        ...received from rcvr 1: msg_id 2, pkt_id 26, orig_pkt_id 3, ttl 1, hops 0, sender node_0
        ....destroy duplicate pong packet
node_2 rcv:pk_in 9,pk_out 16
        ...received from rcvr 1: msg_id 2, pkt_id 22, orig_pkt_id 2, ttl 1, hops 0, sender node_1
        ....destroy duplicate pong packet
node_1 rcv:pk_in 9,pk_out 15
        ...received from rcvr 1: msg_id 1, pkt_id 3, orig_pkt_id 3, ttl 4, hops 1, sender node_3
        ...gen pong to xmtr 1: pkt_id 50, orig_pkt_id 3, ttl 2, hops 0, sender node_1
        ....destroy duplicate ping packet
node_0 rcv:pk_in 9,pk_out 15
        ...received from rcvr 1: msg_id 1, pkt_id 2, orig_pkt_id 2, ttl 4, hops 1, sender node_2
        ...gen pong to xmtr 1: pkt_id 51, orig_pkt_id 2, ttl 2, hops 0, sender node_0
        ....destroy duplicate ping packet
node_3 rcv:pk_in 10,pk_out 16
        ...received from rcvr 2: msg_id 2, pkt_id 35, orig_pkt_id 0, ttl 2, hops 0, sender node_1
                Wrote pong to pong cache, pong_ptr=2
        ...fwd pong to xmtr 0: orig_pkt_id 0, ttl 1, hops 1, sender node_1
        ...gen pong to xmtr 1: orig_pkt_id 0, ttl 1, hops 1, sender node_1
node_2 rcv:pk_in 10,pk_out 16
        ...received from rcvr 0: msg_id 2, pkt_id 37, orig_pkt_id 1, ttl 2, hops 0, sender node_3
                Wrote pong to pong cache, pong_ptr=2
        ...fwd pong to xmtr 1: orig_pkt_id 1, ttl 1, hops 1, sender node_3
        ...gen pong to xmtr 2: orig_pkt_id 1, ttl 1, hops 1, sender node_3
node_1 rcv:pk_in 10,pk_out 16
        ...received from rcvr 2: msg_id 2, pkt_id 32, orig_pkt_id 0, ttl 2, hops 0, sender node_3
                Wrote pong to pong cache, pong_ptr=2
        ...fwd pong to xmtr 0: orig_pkt_id 0, ttl 1, hops 1, sender node_3
```

Figure 7. Log messages from fullyconn_4n scenario

Animation was also used to view the packet exchange in the models. Figure 8 shows the packet flow animation of fullyconn_4n model on the network and node levels.

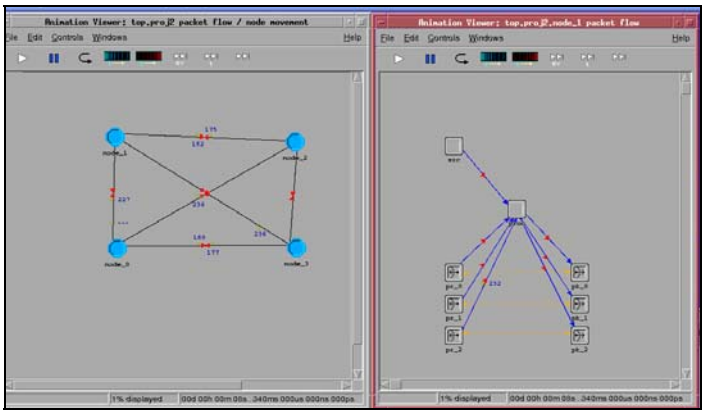

Figure 8. Packet flow animation of fullyconn_4n scenario

4.2. Scalability and fault-tolerance

Network models of different sizes are used to test the scalability of the model. The model fault-tolerance is tested by using a scenario with a failing node. Results from different scenarios are compared.

4.3 Simulation measures

Several object (node and link) statistics are collected during the simulation. Global network statistics such as end-to-end delay are not useful since the network is a virtual network rather than a physical one.

4.3.1 Node statistics

The number of packets received by the node processing unit (**packet count in**) is calculated and updated throughout the simulation. The number of packet sent by the processing unit is also calculated (**packet count out**).

4.3.2 Link statistics

(**throughput packets/sec ←**), (**throughput packets/sec →**) and (**packet-loss ratio**) were chosen among OPNET link statistics.

Figure 9 shows the simulation measures collected during the simulation.

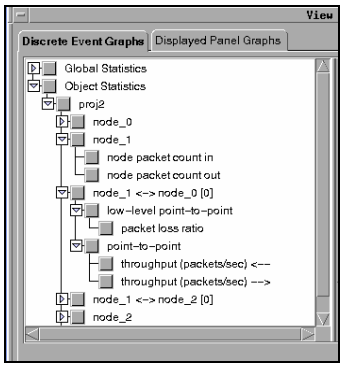

Figure 9. Simulation measures

4.4 Simulation runs

Simulations of all seven scenarios were run for 200 seconds. The nodes start generating ping messages after five seconds from starting the simulation. Two simulation sets were run with two different values for ping messages inter-arrival time (IAT). This value was changed by configuring the OPNET simple_source src module in the node model. The first value is one second, which is the value recommended by the protocol. The second value is 30 seconds. This value was chosen to allow for all messages generated in response the preceding ping message to 'die out', due to the time-to-live (ttl) assigned to each packet.

5. Results

5.1 Node results

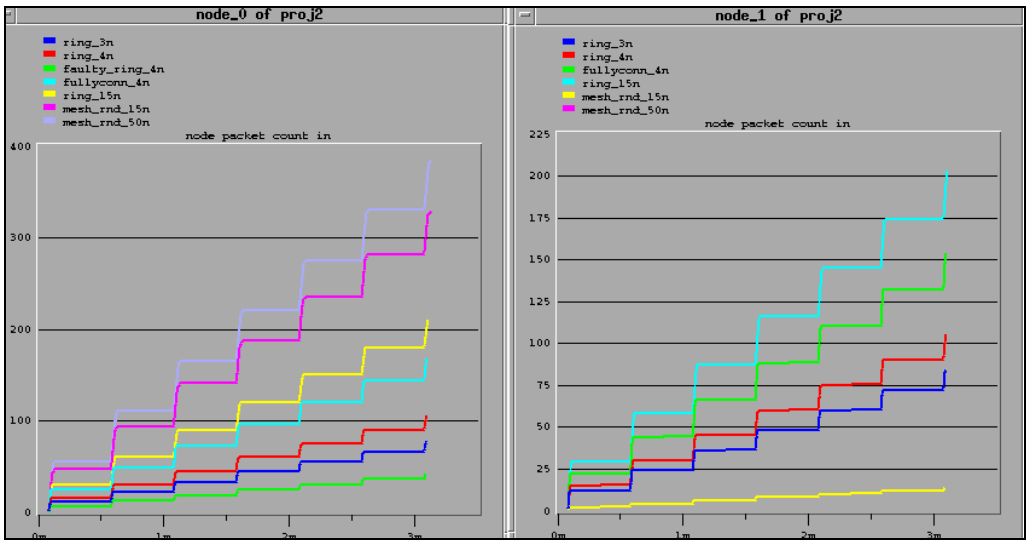

Figure 10.a. Packet count in of node_0 and node_1, ping generation IAT = 30 sec.

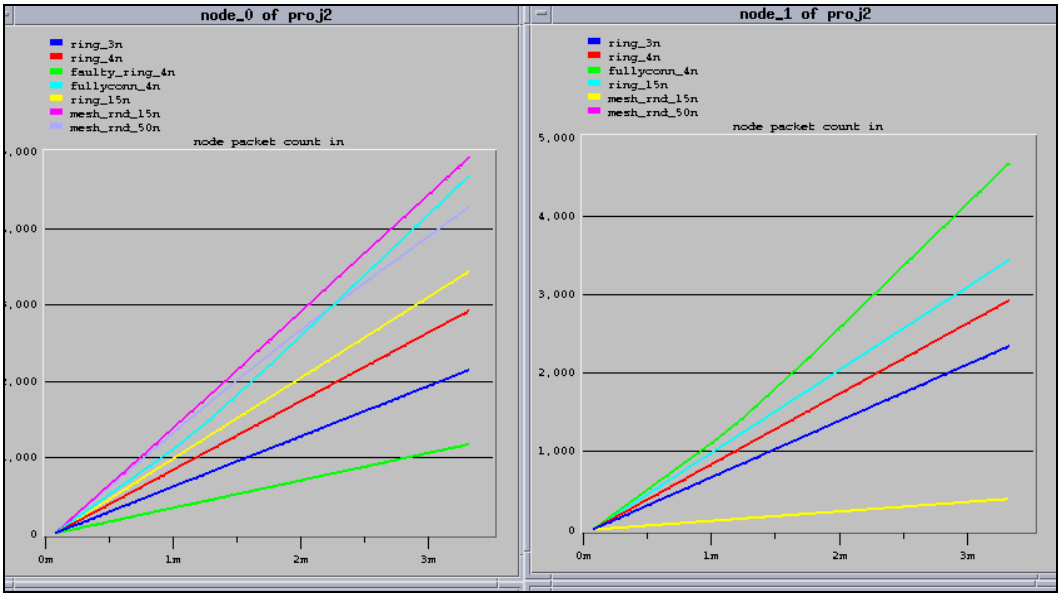

Figure 10.b. Packet count in of node_0 and node_1, ping IAT = 1 sec.

The number of packets received by a node (packet count in) is plot against simulation time. Figure 10 shows the results from node_0 and node_1. At 5 seconds, packets are generated and received by the nodes. In figure 10.a the inter-arrival time (IAT) of generating ping messages is 30 seconds, number of packets is increased in small intervals, then remains constant till the next set of ping messages are generated. Whereas in figure 10.b IAT = 1 second, the packets are continuously being received by the nodes. Please note that in node_1 results, mesh_rnd_15 and mesh_rnd_50 curves coincide. Note also that node_1 does not have results for faulty_ring_4n scenario since node_1 was

chosen to be the faulty node (see figure 6). Figures representing sent packets (packet count out) are similar to the ones shown in figure 10.
Comparing number of packets in different scenarios, we can see that in all graphs:

faulty_ring_4n < ring_3n < ring_4n < fullyconn_4n

This result is expected. A network with a faulty node has less packets and a fully connected network has more packets.
Interesting observation is comparing mesh_rnd_15 and mesh_rnd_50 to ring_15. While random mesh models have higher packets in node_0, they have fewer packets in node_1 than ring_15 model. Looking at the connectivity of node_0 and node_1 (figure 11), we can see that node_0 is connected to 3 nodes, while node_0 is connected only to one node.
Another surprising result is that fullyconn_4n have more packets than mesh_rnd_50 in the case of IAT = 1 sec. This shows the importance of IAT and the node connectivity. It can also be seen that mesh_rnd

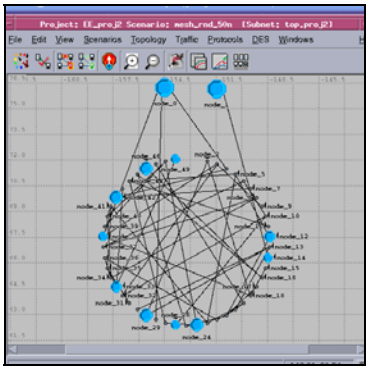

Figure 11. Connectivity of node_0 and node_1 in mesh_rnd_50n scenario

5.2 Link statistics

Link statistics are of less importance than the node statistics since the links are actually virtual links rather than physical links. We can see from figure 12 that packet-loss ratio is equal to 1 in faulty_ring_4n scenarios and equal 0 in other scenarios. Throughput into node_1 is also zero in the case of faulty_ring_4n scenario. Figures 12.a and 12.b show a periodic behaviour due to the high IAT. Fully_conn_4n has higher throughput than faulty_ring_4n, ring_3n and ring_4n (figures 12.a, 12.c).

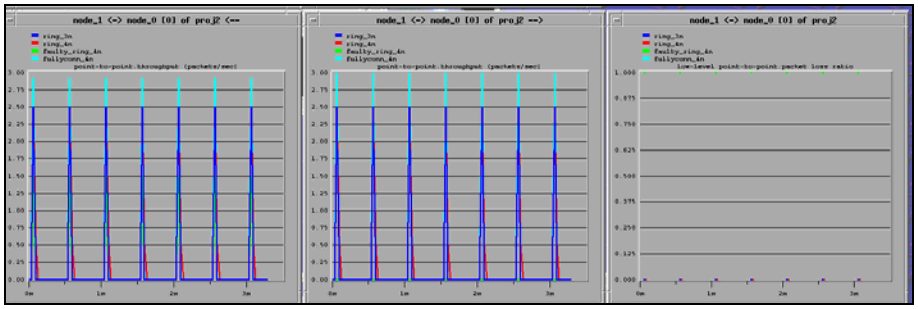

Figure 12.a. Throughput and packet-loss ratio of a link in the first four scenarios, ping IAT = 30 sec.

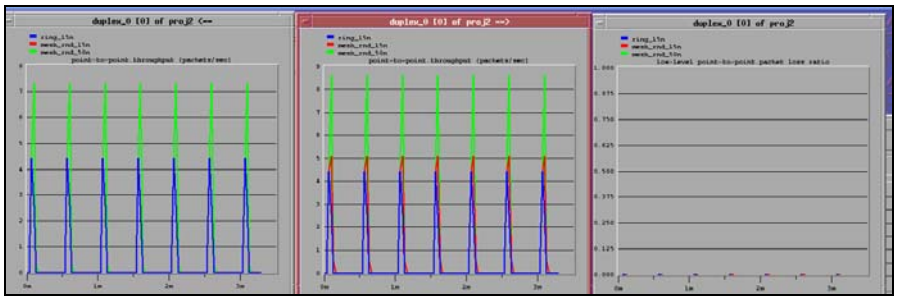

Figure 12.b. Throughput and packet-loss ratio from a (different) link in the rest of the scenarios, ping IAT = 30 sec.

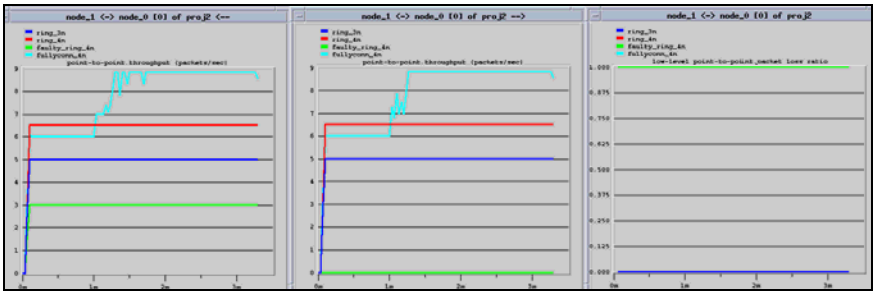

Figure 12.c. Throughput and packet-loss ratio from a link in the first four scenarios, ping IAT = 1 sec.

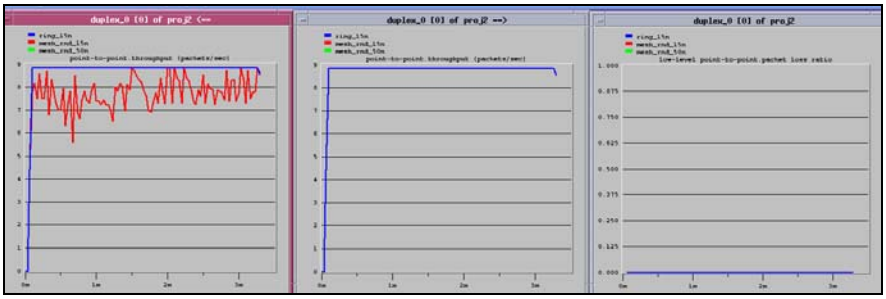

Figure 12.d. Throughput and packet-loss ratio from a (different) link in the rest of the scenarios, ping IAT = 1 sec.

6. Discussion and conclusions

In this project, OPNET is used to model and simulate peer to peer networks. Gnutella protocol is studied and the ping pong message exchange is implemented. The scalability of the model and its robustness to failure is tested using several simulation scenarios.

Results show that large networks do not always carry higher number of packets than small networks. Connectivity of the nodes plays an important role in the amount of message exchange. Heavily connected nodes in a large network exchange more packets than nodes in smaller networks while loosely connected nodes in a large network exchange fewer packets than those in smaller networks. A small fully connected network exchanges a total number of packets larger than a large randomly connected network in the case of the protocol-recommended one second ping generation inter-arrival time. This indicates that the protocol is scalable. It also highlights the importance of the number of peers each node can connect to.

Results also show that networks with faulty nodes still function by exchanging messages throughout the network. However, the number of packets is less than that of similar networks with no faulty nodes. In addition, it was shown that ping generation inter-arrival time have a big effect on the number of packets in the network, which is expected.

Challenges in implementing the project:

The main challenge in this project was to simulate the dynamic nature of the network. OPNET doesn't allow for dynamic creation and failing of objects during simulation. Therefore, simulation scenarios have to be picked fixed rather than dynamic. It would be interesting to vary the network topology in response to received pong messages.

Another difficulty in modeling with OPNET is that programming is allowed only inside each node model, namely the process model. There is no main program that can be used to set the network structure or node behaviour. While this poses a difficulty in modeling, it actually represents the true nature of decentralized systems. In such systems, all the functionality of the system lies in the individual nodes rather than one centralized process.

Initially, I implemented the network using a hub node and several peer nodes in a star topology. However, Gnutella protocol specifies that each node routes the messages to other nodes in the network. Therefore, the hub node functionality was merged into the peer node.

Future work:

1. Perform more analysis to the results by examining more node results, larger scenarios and other faulty networks.
2. Model the rest of the messages of the protocol: Query, QueryHit and Push.
3. Model Gnutella v0.6 with the ultrapeer-leaf architecture.
4. Examine using TCP OPNET model for a lower-level implementation of the Gnutella protocol.
5. Investigate other simulation tools that can allow dynamic change of topology during simulation.

7. References

1. "The Annotated Gnutella Protocol Specification v0.4", http://rfc-gnutella.sourceforge.net/developer/stable/index.html, accessed April 2006.
2. "Gnutella Protocol Development – RFC-Gnutella 0.6", http://rfc-gnutella.sourceforge.net/developer/testing/index.html, accessed April 2006.
3. L. Gong, "Peer-to-Peer Networks in Action," *IEEE Internet Computing*, vol. 5, no. 1, 2002, pp. 37-39.
4. www.bearshare.com, accessed April 2006.
5. P. Maymounkov and D.Mazieres, "Kademlia: A Peer-to-peer Information System Based on the XOR Metric," *Lecture Notes in Computer Science*, vol. 2429, revised papers from the First International Workshop on Peer-to-peer Systems, 2002, pp. 53-65.
6. www.emule-project.net, accessed April 2006.
7. www.wikipedia.org/wiki/BitTorrent, accessed April 2006.

Other references:

1. Sen, S., Jia Wang, "Analyzing peer-to-peer traffic across large networks," *IEEE/ACM Transactions on Networking*, vol. 12, no. 2, 2004, pp. 219 – 232.

2. Manini, D., Gaeta, R., Sereno, M., "Performance Modeling of P2P File Sharing Applications," *Workshop on Techniques, Methodologies and Tools for Performance Evaluation of Complex Systems*, pp. 34 – 43, 2005.

3. He Q., Ammar M., Riley G., Raj H., Fujimoto, R., "Mapping peer behavior to packet-level details: a framework for packet-level simulation of peer-to-peer systems," *IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems*, pp. 71 – 78, 2003.

4. Juan Li, Son Vuong, "Ontology-Based Clustering and Routing in Peer-to-Peer Networks," *International Conference on Parallel and Distributed Computing, Applications and Technologies*, pp. 791 – 795, 2005.

5. Tsungnan Lin, Hsinping Wang, Jianming Wang, "Search performance analysis and robust search algorithm in unstructured peer-to-peer networks," *IEEE International Symposium on Cluster Computing and the Grid*, pp. 346 – 354, 2004.

8. Code listing

Submitted in a separate file (code_listing.txt), which includes:

Node code:
EE_proj2_nd_proc.pr.c

Debug output:
ring_3n_verify.out
fully_conn_4n_verify.out