ENSC 427: Communication Networks

Robustness of Gnutella network

Spring 2009

Final Project

Hao Su 301006404
Email: hsu@sfu.ca

Ken Wu 301007515
Email: ksw3@sfu.ca

Webpage:
www.sfu.ca/~hsu

Table of content

1. ABSTRACT

Like most other P2P protocols and networks, Gnutella builds a virtual network at the application level with its own routing mechanisms. However, unlike P2P networks such as Napster, which uses a client-server structure for file searching, Gnutella is a true P2P system that uses a decentralized system even for file searching, via Ping and Pong descriptors. This makes Gnutella a very robust to failure. In this project, OPNET modeler will be used to simulate a simplified Ping and Pong message-exchanging Gnutella v0.4 protocol. We will test the robustness of such a network by monitoring Packets received individual nodes in the network. We will implement a variety of different network topologies, and fail certain nodes of these topologies to test the robustness to failure. As well, we try to find out what the TTL, or the time-to-live value has on the robustness of the Gnutella network. Our results show that the node connectivity has a great impact on the traffic of the network, but the topology does not. Also, faulty networks will receive fewer packets at each node than one that is fully connected, but it would be very difficult to completely cut off individual nodes from the network, unless a large number of nodes in the network are disconnected. Our results are inconclusive as to how TTL will affect the robustness of the Gnutella network.

2. INTRODUCTION AND BACKGROUND

2.1 P2P networks

A peer-to-peer (P2P) network is formed by two or more clients connected together to share resources directly with one another. In effect, each peer node in the network can be seen as both the server and client and are equal to each other, with no robust machine to act solely as a server. This type of network is refereed to as "decentralized", in the sense that it differs from the client-server model where communication is usually to and from a central server. For example, a non-P2P file transfer system have a file transfer protocol (FTP) server where many clients initiated the download, and the server(s) react to satisfy these requests. Unlike a P2P protocol, the clients and the servers are itself.

Typically, P2P networks are used for connecting nodes via largely ad hoc connections. This type of networks is often used for files sharing and real time telephony traffic. Some P2P networks and applications such as Napster and OpenNAP use a client-server structure for some of its tasks such as file searching, and uses P2P structures for the remaining tasks. However, networks such as Gnutella and Freenet uses what is referred to as true P2P, in which all tasks use the same P2P structures. In this project, we focus on examining the robustness of the Gnutella network by implementing an extremely simplified model of a Gnutella (version 0.4) network with different topologies, failed nodes and time to live (TTL) values, then collecting data such as number of packets sent and received for these topologies.

2.2 Gnutella Network

Gnutella is a protocol for distributed search developed by Justin Franel and Tom Pepper of Nullsoft in early 2000, and it was becoming the most popular file sharing network on the internet by late 2007. Although the Gnutella protocol is referred to as a "true" P2P network, it is greatly facilitated by directory servers that inform peers of the network addresses of other peers. In the Gnutella protocol, each of these client/server peers is called a servent. Each time a servent wants to do a search; it sends a request to all of its actively connected servents. These servents in turn forward the request, and so on, until the packet records a predetermined number of maximum "hops" from the sender.

2.2.1 Descriptors

The Gnutella protocol consists of a set of 5 descriptors used to communicate data between the servents: **Ping, Pong, Query**, **QueryHit** and **Push**. **Ping** is used to actively discover hosts on the network, and it is sent out to all of the servents that are connected to the sender. A servent that receives a **Ping** descriptor is expected to respond with one or more of the **Pong** descriptors, which includes the address of a connected Gnutella servent and information regarding the amount of data that it has made available to the network. The **Query** descriptor is used primarily as a searching mechanism for searching files, and sends a **QueryHit** descriptor back if indeed there is a

match. Finally, the **Push** descriptor allows a firewalled servent to contribute file-based data to the network.

2.2.2 Descriptor header formats

Once a servent connects to the network, it communicates to the other servents by sending and receiving descriptors preceded by a header. The byte structure of the header is shown as follows:

Table 1. Descriptor Header

Message ID	Payload Descriptor	TTL	Hops	Payload Length
0……15	16	17	18	19……..22

The following table describes each of the fields in the Descriptor header

Table 2. Descriptor header byte field

Descriptor ID / Message ID	The first 16 bytes of the descriptor header, represents a unique ID for identifying the descriptor on the network.
Payload Descriptor	1 byte of string describing what type of descriptor is expected to follow the header: 0x00 = Ping 0x01= Pong 0x40= Push 0x80= Query 0x81=QueryHit
TTL	Stands for Time To Live. This byte of string represents the number of time that the descriptors still have left to be forwarded by the Gnutella network. The TTL will be decremented each time the descriptor is forwarded until the TTL reaches 0.
Hops	A byte of string representing the number of times that the descriptor has been forwarded. It will be incremented every time the descriptor is forwarded. TTL + Hops will always be a constant number; this property can be used for error checking.
Payload Length	4 byte string that stores the length of the descriptor that is expected to follow the header. This will give the servent heads-up on where the next descriptor header is going to be.

2.2.3 Descriptor payload formats

2.2.3.1 Ping

The Ping descriptor has no payload, and has zero length; therefore it is simply represented by its descriptor header with a payload descriptor of 0x00 and a payload length of 0x00000000.

2.2.3.2 Pong

The Pong descriptor payload is made up of 14 bytes of information whose byte format is shown in table 3.

Table 3. Pong descriptor payload format

Field	Port	IP address	Number of Files	Number of kilobytes shared
Bytes in descriptor payload	0……..1	2……5	6……9	10……13
description	Port number that the responder can accept connections	The IP address of the responding host	Number of files that the responding host is sharing with the Gnutella network	Number of data in kilobytes that the responding host is sharing with the Gnutella network

2.2.3.3 Query

The Query descriptor payload is made up of 3 or more bytes of information whose byte format is shown in table 4.

Table 4. Query descriptor payload format

Field	Minimum speed	Search criteria
Bytes in Descriptor Payload	0……1	2……X
description	The minimum speed (in KB/sec) at which the responding servent should be able to communicate. If not then don't respond.	The string that describes the query. The length of this string is determined by the payload length in the descriptor header.

2.2.3.4 Query Hit

The QueryHit descriptor payload format is shown below in table 5

Table 5. QueryHit descriptor payload format

Field	Number of hits	Port	IP address	Speed	Result Set	Servent Identifier
Bytes in payload	0	1……2	3……6	7……10	11……n-1	n……n+16
Description	# of query hits	Port number for responding host	IP address of the responding host	Speed (kB/sec) of responding host	Set of responses to the query, includes file index, size and name	16 byte string for identifying the responding servent

2.2.3.5 Push

The Query descriptor push is made up of 26 bytes of information whose byte format is shown in table 6.

Table 6. Push descriptor payload format

Field	Servent identifier	File index	IP address	Port
Bytes in payload	0…….. 15	16…….. ….19	20…….. .23	24……... 25
Description	16 byte string identifying servent being requested to push	Index identifying file to be pushed	IP address of the host servent with the file to be pushed	The PORT to which the file will be pushed to

2.2.4 Descriptor routing

The Gnutella protocol requires that the servents route network traffic to according to a specific routing rules to ensure fast connection setup and data transfers. When a servent receives a ping or a pong descriptor, it will forward these descriptors to neighbor servents which are directly connected except the servent that received the descriptor from. It may also decide to reply with a pong same path that the ping was sent if the time to live counter is not zero. To avoid extra traffic, a servent receiving a pong descriptor with a descriptor ID will discard the Pong if it has not seen a Ping with the same descriptor ID. For the same reasons, when a servent receives a Query descriptor, it will reply with a QueryHit descriptor only in the same path that the Query descriptor came from, and discard any received QueryHit descriptor for which it had not received a Query descriptor previously. The same can be said about the Push Descriptors. Furthermore, a servent receiving a descriptor that it had already received before will try not to forward the descriptor any further.

Once a QueryHit descriptor is received by a servent, it may initiate data transfer directly from the targeted servent to try to download the file described by the QueryHit descriptor's result set. The protocol used for download is HTTP, and is done outside of the Gnutella network.

In Fig.1 shows the routing of the Ping and Pong descriptors in the Gnutella network with TTL and hops parameters. Fig. 2 shows the routing of the Query, QueryHit and Push descriptors

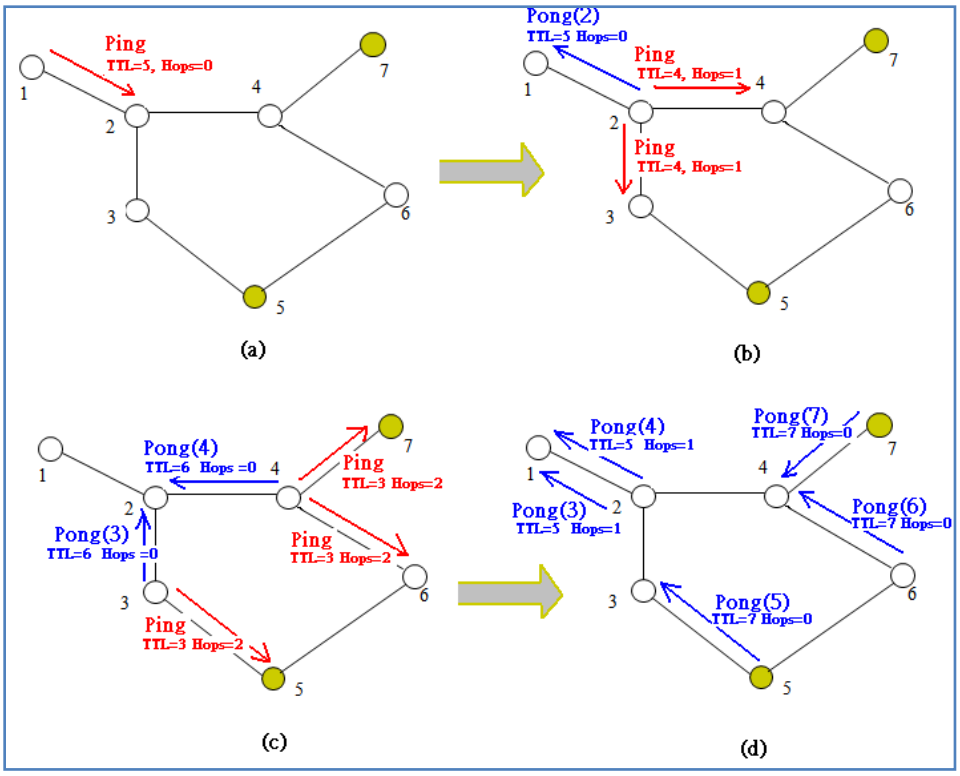

Figure 1. Ping and pong routing in Gnutella

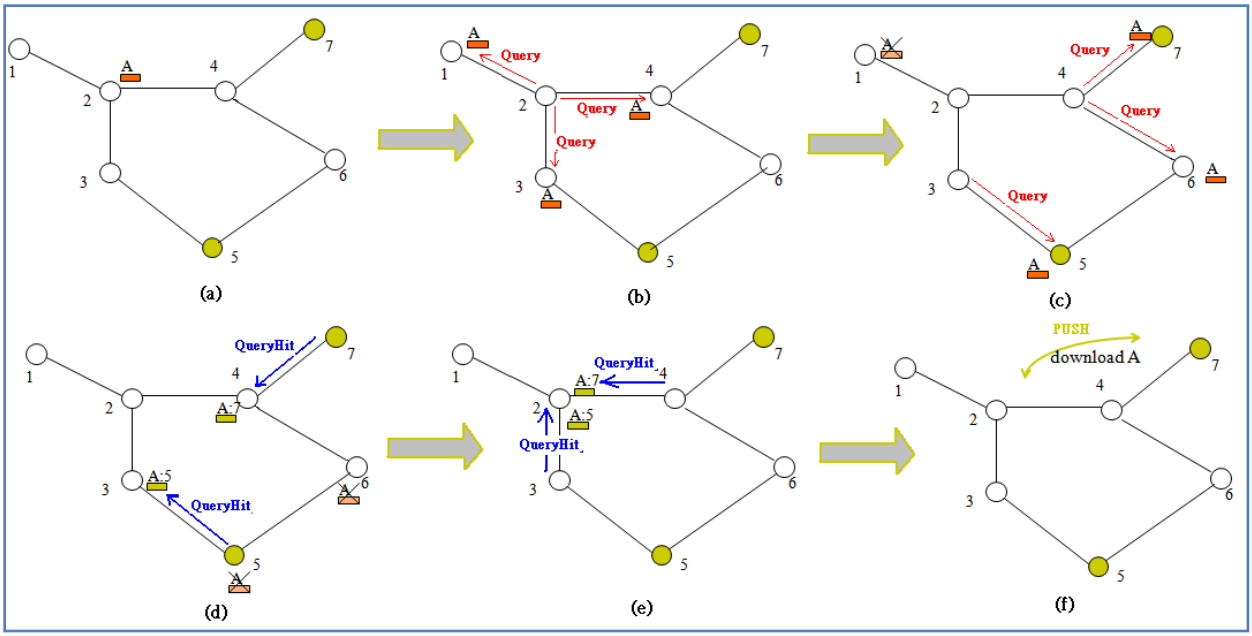

Figure 2. Query, QueryHit and Push routing in Gnutella protocol

2.2.5 Project Scope

The goal of the project is to build a very simplified OPNET model of a Gnutella v0.4 network that simulates Ping and Pong message passing between servents. Different scenarios of small Gnutella networks with varying numbers of servents and different topologies will be built and tested for robustness. We will vary parameters such as number of servent nodes, number of failed/disconnected nodes, network topology, and TTL initial values to see their effect on the robustness and traffic of the Gnutella network.

It must be emphasized that the real Gnutella network is very large and complex, and the model we are implementing is only an extremely simplified model. Note that we are not including the Query, QueryHit and Push descriptors in our model, or the actual protocol (HTTP) used for file transfer/download.

3. IMPLEMENTATION

A servent model is implemented with a specified packet format, link model, node model and a process model to test the robustness of the P2P network.

3.1 Packet Format

We created a packet format with many variables such as original packet id, message id, time to live, hops and sender id, each variable contained 32 bits. The process model will use these variables to allow the communication between two or more nodes; we'll discuss more on the process model later on the report. The packet format created is shown in Fig. 3.

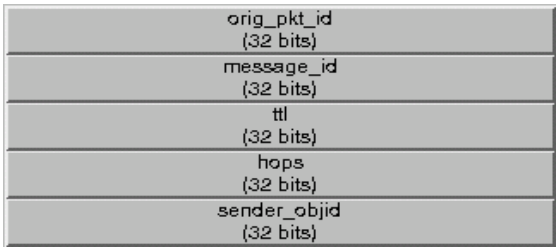

Figure 3. The packet format

3.2 Link Model

The link model is created to support the point to point duplex and packet format we created. Some of the attributed we changed are: error correction model *ecc model* to *ecc_zero_err*, *error model* to *NONE*, point to point propagation delay model *propdel model* to *dpt_propdel*, point to point transmission delay *txdel model* to *dpt_txdel* and declare external files. These setting are set to allow us to analysis error correction with *ecc_zero_err* model and allowed us to take link delay statistics. The link model is shown in Fig. 4.

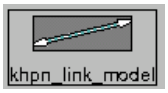

Figure 4. Duplex link model

3.3 Node Model

The node model consisted of a simple source process which created ping message at a constant rate. Also we can change the Inter-arrival-time (IAT) of the ping message if we wish. The default IAT is set to one second, we did not change IAT for this project because we figured the graph for default IAT is easier to analysis.

There are three point-to-point transmitters and receivers which support the packet format we created. There is a process *proc* in the center of the node model, this process model is created to support the flow of the packets received and transmitted. We will explain more on how the process model works later on the report.

Notice the packet stream have to be connected in ordered fashion or else it won't work, this was done by analyzed and followed the order specified in the code from the process model. We connected the packet streams from receiver 0 *pr_0* to receiver *pr_2* to process model *proc*, then source src to proc and finally *proc* to the transmitters *pt_0* to *pt_2* respectively. Lastly the transmitters and the receivers are connected with logical Tx/Rx (transmitter and receiver) association lines. We check these connections are what we expected by verifying the connectivity from the process model *proc,* indeed we obtained the connected we wanted.

In the node interface we set node type to fixed node and in node statistics we promoted the packet count in and packet count out, these data can be collected and analyzed. Each node model itself will represent a servent (peer/server) itself. The Node model we created is shown in Fig. 5.

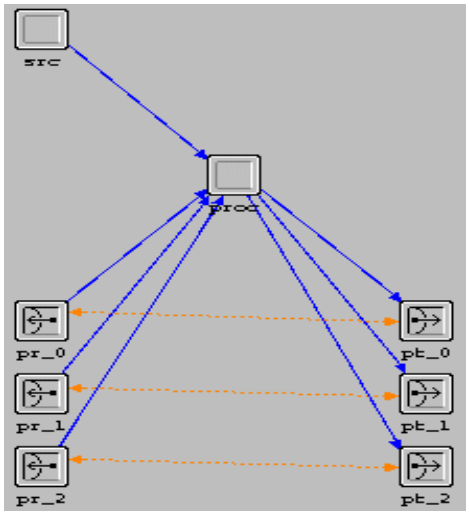

Figure 5: The node model

3.4 Process Model

We have three state in this process model, the *init* state, *idle* state, and *procRCV* state. The init and procRCV state is in forced state which means from init to idle state and procRCV to idle state required no condition to change state. In the idle state, when there is no action idle state will loops to itself with *default*. When the source *SRC_ARRVL* is arrived then the ping message is generated *xmt_src_ping* then it loops back to the idle state. Upon the arriving of packet detected by the receiver RCVR_ARRVL idle state jumps to procRCV state. The procRCV state handlethe flow of the ping/pong message. It replied with the pong message and forward ping/pong if time to live is greater than one. The global variable is declared in the State Variable. Noticed more coding are in the Header Block and Function Block. The local statistic is set to collect *pack count* in and *packet count out*. The process model is shown in Fig. 6. We'll provide the code for our process model in the appendix.

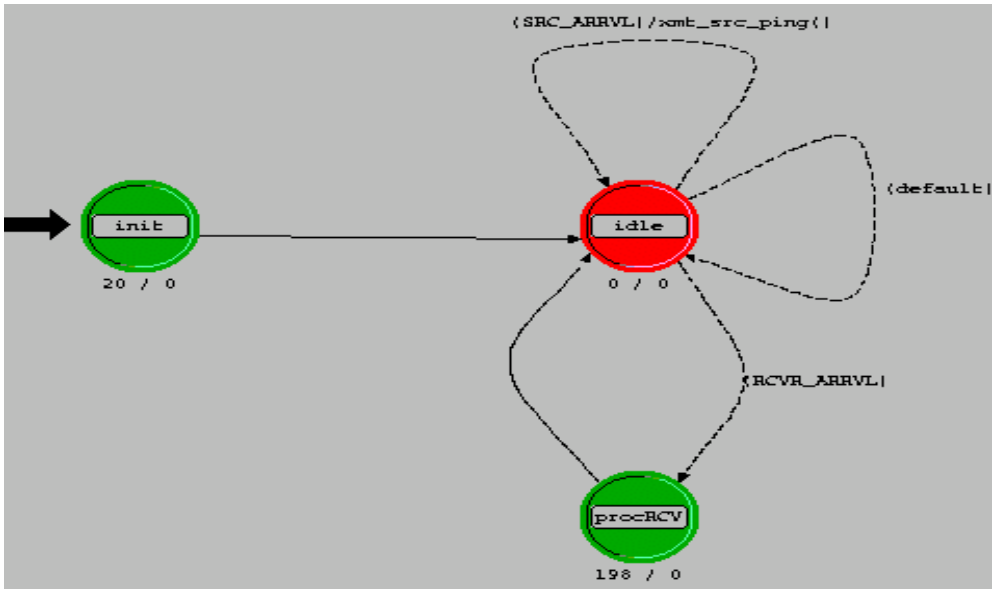

Figure 6: The process model

3.5 Network models

In this project, there were a total of 7 network models that were implemented to test the different aspects of the network's robustness. Each network servent is implemented using the node model described in section 2.3, and the link model of section 2.2. Since the node model has 3 receiver and 3 transmitters, each of the servent can only be connected to 3 other servents. So in this way, we were somewhat limited in our network model implementations. The 7 network model implementations are explained in table 7.

Table 7. Network models of different scenarios and their descriptions

Scenario	Description
5 servent ring	This is a simple ring topology with 5 servents. Each Servent is connected to the servents immediately to its left and right.
5 servent ring with 1 servent faulty	This is the same 5 servent ring topology as above, except nodes one is failed. This scenario is compared with the previous one to see the effects of failed servents in a ring topology.
28 servent ring fully interconnected	This is a more complex ring topology with 28 servents, and each servent is connected with 3 other servents that's is the maximum number of connection a servent is allowed due to our implementation design with 3 transmitters and 3 receivers.
28 servent ring fully interconnected with 10 faulty	This is the same 28 servent fully connected ring topology as above, except that 10 nodes are purposely failed. This scenario is compared with the previous one to see the effects of failing many nodes in a larger network.
Random network	This is more randomly connected network with each servent randomly connected to 1, 2, or 3 other servents in the network. Different servents inside this random network is compared with each other to see whether each node inside the topology will receive different number of packets.
Line network with TTL=2	A straight line network with 7 servents is implemented. In this network, the TTL(time to live) parameter of each Ping and Pong packets is reduced to only 2, so each packet can be passed on from servent to servent only twice.
Line network with TTL=5	This is the same network as the one above, except that in this scenario, the TTL(time to live) is increased to 5. This scenario is compared with the previous one to see the effects of changing the TTL of packets sent by the servents.

Figures 7-12 shows the network models of all the scenarios described in Table 7.

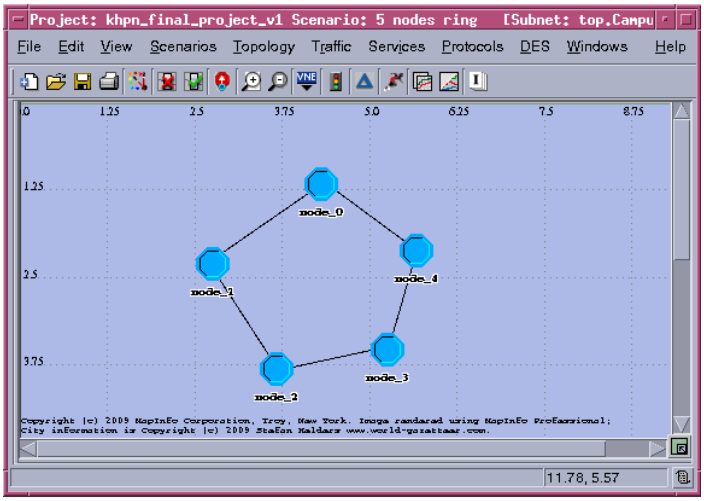

Figure 7. Scenario 1 - 5 Servents ring

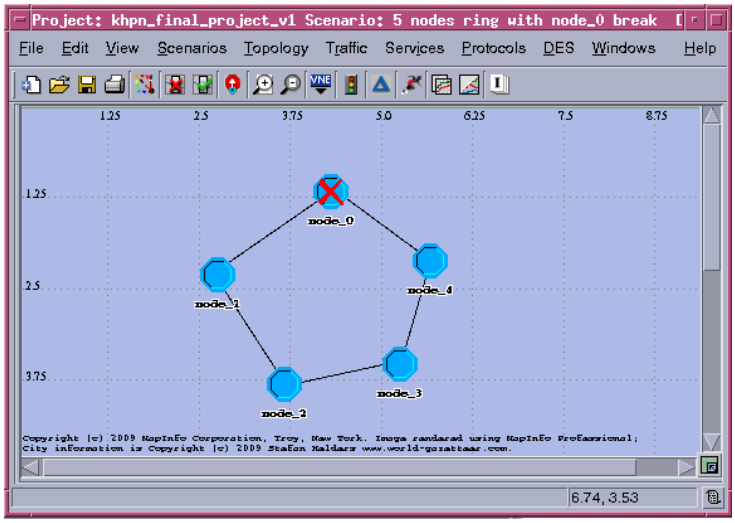

Figure 8. Scenario 2 - 5 servents ring with node_0 failed

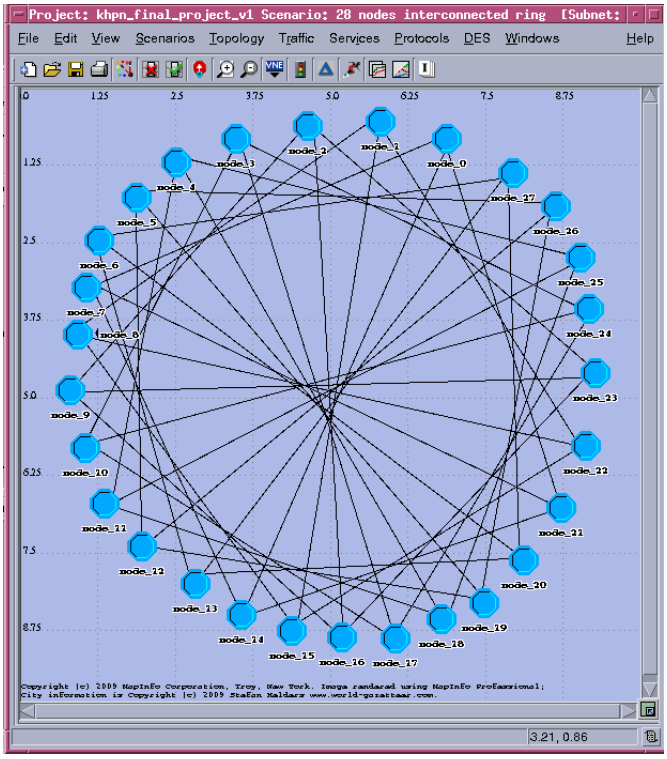

Figure 9. Scenario 3 - 28 servents interconnected ring

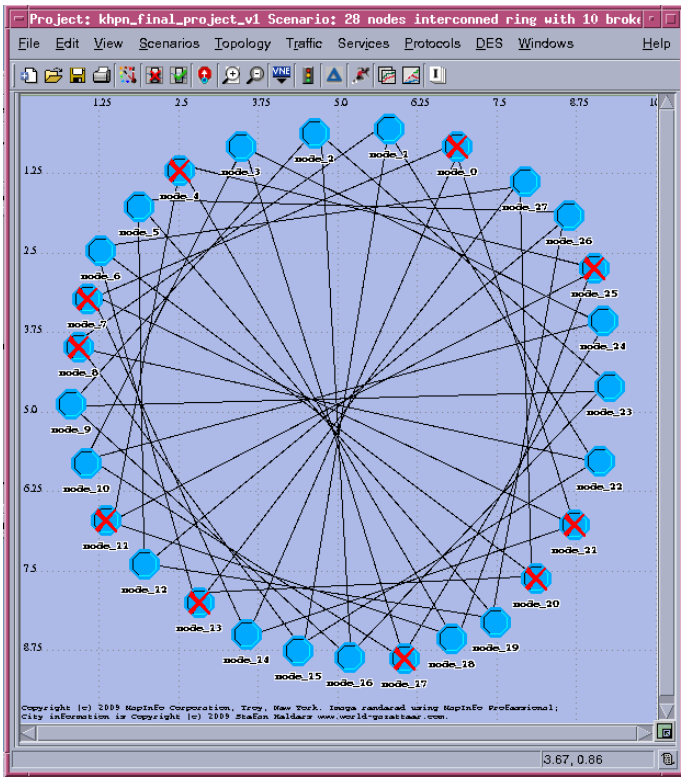

Figure 9. Scenario 4 - 28 servents interconnected ring with failed nodes

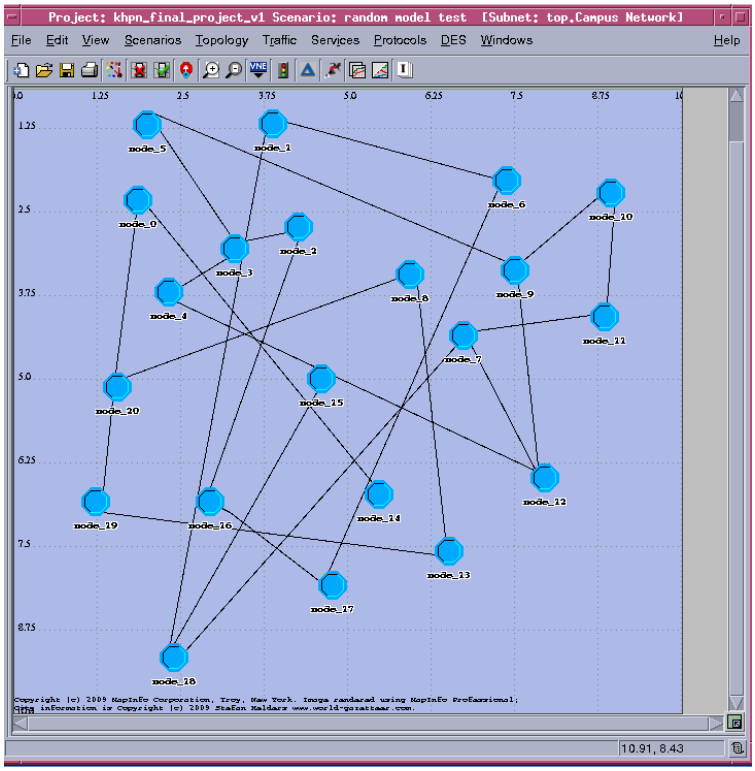

Figure 10. Scenario 5 - randomly connected servents

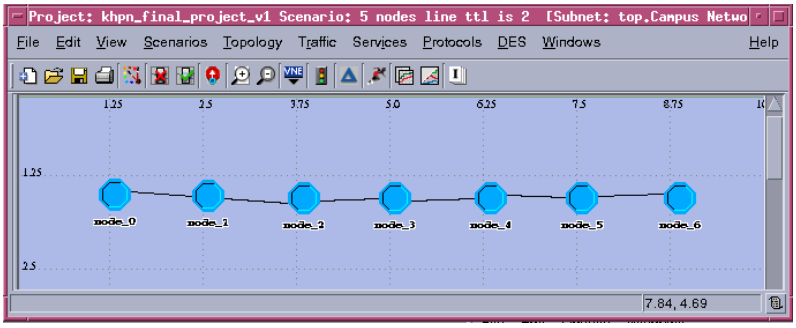

Figure 11. Scenario 6 and 7 - line topology with different TTL

Noticed that scenario 6 and 7 have the same picture, but the node models used are different. The node model for scenario 6 is khpn_node_model_ttl this node model used TTL = 2 whereas, the node model for scenario 7 is khpn_node_model this node model used TTL = 5.

4. SIMULATION and RESULTS

4.1 Running the simulation

All simulations were run for 200 seconds, and ping messages were generated 5 seconds after simulation begins. In our simulation, we only used the inter-arrival time (IAT) of 1 second at the node source. Each scenario is modeled in campus network, the average overlaid statistic of packet count in is taking for analysis in the graphs below. The simulation for each scenario takes about a minute to simulate depending on server traffic.

4.2 Node Results

In all simulations, we plotted the number of packets received (packet count in) by the node (servent) against the simulation time. Because the IAT is only 1 second, the graphs show as a linear line increasing over time. This shows that the packets are continuously being received as time progress.

In Fig. 12 shows the result of scenarios 1 and 2, namely, the 5 node ring and the 5 node ring with a failed servent at node_0. The result is collect for node_1 only, since all the non-broken nodes have the same characteristics, we only need to analysis one node to show the robustness of the network. As expected, the 5-node ring that does not have a broken node received more packets than the 5-node ring with a broken node, and the number of packets received by the latter is roughly half the amount received by the former. This is easy to see why, because in such a ring topology, each node should receive ping and pong messages from both the left and right, but when a node is broken, the path in one way is broken and the node can only receive packets travelling one way rather than both ways.

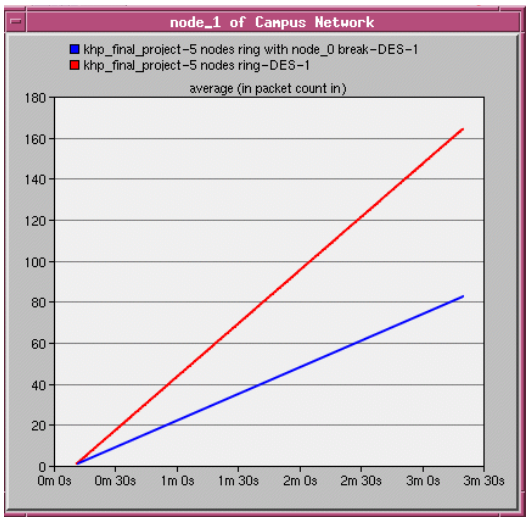

Figure 12.Packets received by the 5 ring topology node 1

In Fig. 13 Shows the result of scenarios 3 and 4, namely, the 28-node completely connected network and the same network with 10 broken nodes. The 28 nodes interconnected ring topology (red line) has higher packet count in compared with 28 node interconnected ring topology with 10 broken nodes (blue line) as expected. Although there is many broken nodes to the network node_1 still able to connect to other servents thus, the network for P2P is very strong and is hard to break “all” the nodes to the network. This greatly enhanced the file transfer compared with a server based protocol; if the server node is broken than no other user can access the network until the server is running again.

We can compare Fig. 12 with Fig. 13 we noticed Fig 13 generally have more packets count in. This is because there are 3 connections to node_1 in the 28 nodes scenario compared with 2 connections to 5 nodes scenario.

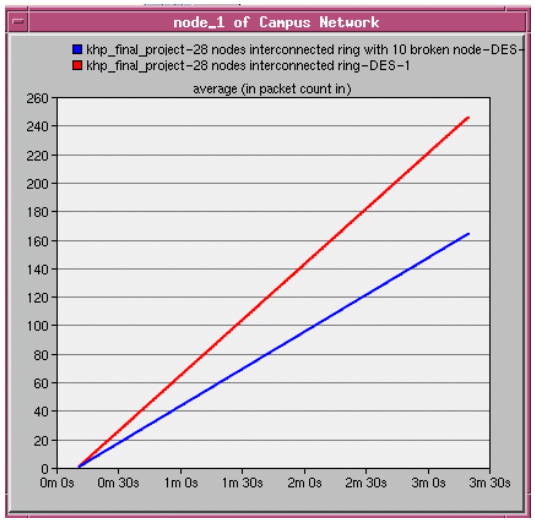

Figure 13: Packet received by 28 nodes topology in node_1

Fig. 14 shows the result of scenario 5, which is a “random” network topology. Note that this scenario is only a very crude model of a realistic random topology, created with reference to. This graph shows the packet received at three different nodes (node_1, node_3 and node_15). As expected, node 3, which has the most connections with other nodes, received the most number of packets. Node 15, which only has 1 other node connected to it, received the least number of packets, while node 1 received the medium amount of packets. Compared with figure 13, we see that node 3(red line in figure 14) in the random model received the same number of packets as the 28 node topology(red line in figure 13). This shows that the network topology does not really affect the number of packets received at each node when the inter-arrival time is very fast. Rather, what matters is how many other nodes are connected to the node in question. Also, we can see that for even though failing nodes will decrease the number of packets received by the node in question, to truly disconnect it from the rest of the network one must sever every connection it has with the rest of the network.

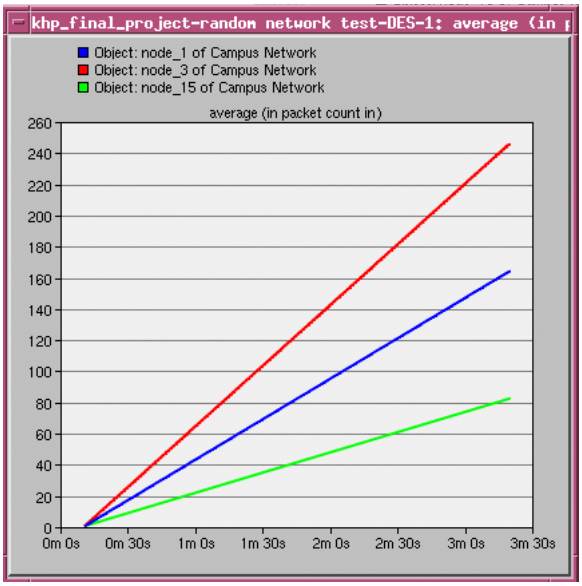

Figure 14. Random topology, packet received at different nodes in the network

Fig. 15 Shows the result of scenario 6 and 7, namely, the line topology with TTL=2 and TTL=5. Initially, one may find the result very surprising, as the two different TTL values yields the same result. One would expect that TTL=5 should give a higher number of packet received, as the Ping and Pong messages would have a more hops to propagate before they are discarded, thus each node would receive more packets, especially in a simple line topology such as this, where packets can only travel in one path. However, since we are using inter-arrival time (IAT) of only 1 second, it could be possible that the packets do not have time to “die out” before the next packet is sent out 1 second later, so TTL does not really affect the packet received by each node. In other words, each non-edge node in the topology is still receiving packets from both sides at a constant rate despite the lower TTL, and thus the result is saturated, and the graphs are overlapped in Fig. 15.

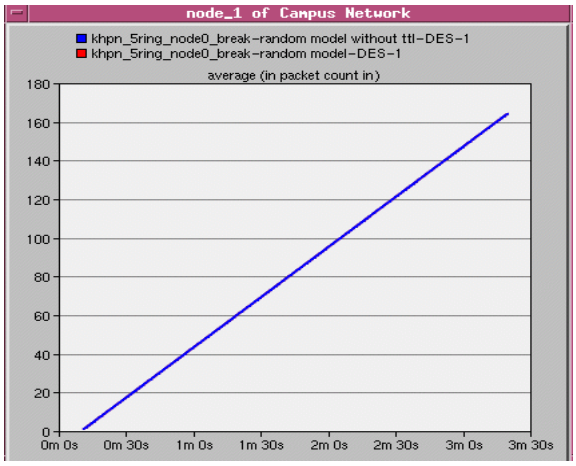

Figure 15. Line topology with different TTL values

5. CONCLUSION and Discussion

5.1 Conclusions

In this project, we implemented a very simplified Gnutella P2P network using the OPNET modeler. The node model we constructed only simulates the Ping and Pong descriptor message passing in Gnutella servents, but omitted the Query and QueryHit descriptor that the real Gnutella protocol uses. Using our node model, we built network models with varying number of nodes, in different topologies. From our resulting graphs, we have found that the Gnutella protocol and network structure is very robust indeed. Although the number of packets received at the node in question will decrease if other nodes close to this node are failed, to truly disconnect it from the rest of the network one must ever every connection it has with the rest of the network.

Results also show that when the inter-arrival time of the Ping and Pong generation is very fast, the network topology does not really affect the number of packets received at a node. Rather, what affects the number of packets received is how many other nodes that the node in question is connected to. Of course, this would not necessarily be true in a real world Gnutella system, since the inter-arrival time of each servent will be different, and depending on where in the topology that the node in question is connected to, there will be a difference in the number of packets received.

Furthermore, it must be noted that at this time in our study, the effects of TTL on the number of packets received is inconclusive. As mentioned earlier, our investigation on TTL yielded the same number of packets received no matter what value we set our TTL to be. This is due to the fact that our model has a very short inter-arrival time of only 1 second, therefore, the packets being received at the node in question does not have time to “die out”, thus saturating the result. Theoretically, however, the TTL should affect the number of packets received at each node, because a larger TTL will give each packet more range to travel, thus having more chance to reach its destination.

5.2 Project Difficulties

The difficulty of this project resides mostly in the implementation of the process model and the node model. Due to the massive number of steps required to build a working simulation of our Gnutella network and the complexity of the OPNET tool and the C code, errors are easily made which results in a non-compiling model, or a model that does not yield the correct results. Due to these reasons, a great deal of time was spent on debugging to make a working model. More problems arose even after a successful working model had been built. For example, because of the complexity of our model, and the large number of inter-related state variables, functional blocks and other C codes, changing one small thing in the node model will make the whole model not compile, and debugging will take a long time. Initially, to demonstrate the effect that TTL has on the robustness of the Gnutella network, we intended to modify our existing node model to have no source. This way, we would be able to make a network in which some nodes are able to generate Pings and Pongs, while the other nodes are not, but simply pass them along in the network. Such a network will better demonstrate the effect TTL will have on robustness of the system. However, when we tried to modify our existing node model to have no source, it would not compile, and we could not debug the problem. Despite these difficulties, the project is non trivial, because we are able to see that the topology does not really affect the packets received at each node. Instead, how many connections are at each node is what really affects the number of packets received at the node. Also, we were able to see that it would be really difficult to really fail a decentralized network such as Gnutella.

5.3 Future Work and Improvements

In the future, we could improve the project by changing the inter-arrival time of the packets, therefore decreasing the rate at which Ping and Pong messages are generated, so that the results will not be saturated in hopes that we will see a difference in the number of packets received when the TTL value is changed. Also, if there were more time, we could successfully make a new node model that has no source, but simply pass on the received messages. This way will also help see the effects that TTL has on the number of packets received at each node. Furthermore, it would also give a better and more accurate simulation of the Gnutella protocol if we could include the Query, QueryHit, and Push descriptors in our future model.

Reference

[1] D. Andre, "Peer-to-Peer Networks as Content Distribution Networks," *ensc.sfu.ca*, report. Fall 2003. [Online]. Available: http://www.ensc.sfu.ca/~ljilja/ENSC835/Fall03/Projects/dufour/Report.pdf. [Accessed: Mar. 06, 2009].

[2] "BitTorrent," Mar. 10, 2008. [Online].
Available: http://wiki.limewire.org/index.php?title=User_Guide_Bittorent. [Accessed: Feb. 06, 2009]

[3] E. Elghoneimy, "Scalability and Robustness of the Gnutella protocol," *ensc.sfu.ca,* report. Spring 2006. [Online]. Available: http://www.sfu.ca/~eelghone. [Accessed: Feb. 06, 2009].

[4] T. Kelvin, "Examination of Routing Algorithms in Distributed Hash Tables (DHTs) for Peer-to-Peer (P2P) Network," *ensc.sfu.ca*, report. Spring 2008. [Online].
Available: http://www.sfu.ca/~kta18/ENSC835ProjectReport.pdf. [Accessed: Feb. 13, 2009].

[5] T. Klingberg, "Gnutella 0.6," *murdoch.edu.au*, report. June 2002. [Online]. Available: http://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html. [Accessed: Feb. 13, 2009].

[6] W. Stephanie, "How Kazaa Works," *computer.howstuffworks.com*, document. [Online].
Available: http://computer.howstuffworks.com/kazaa3.htm. [Accessed: Feb 13, 2009].

[7] A. Dufour, L. Trajkovic, "Improving Gnutella Network Performance Using Synthetic Coordinates," *ensc.sfu.ca*, report. August 2006 [Online].
Available: http://www.ensc.sfu.ca/~ljilja/papers/QShine2006_158.pdf. [Accessed: April 03, 2009].

Appendix

Project code

```
/* Process model C form file: khpn_process_model.pr.c */
/* Portions of this file copyright 1992-2007 by OPNET Technologies, Inc. */
/* This variable carries the header into the object file */

const char khpn_process_model_pr_c [] = "MIL_3_Tfile_Hdr_ 140A 30A opnet 7 49D9848C 49D9848C 1 payette fta1 0 0 none none 0 0 none 0 0 0 0 0 0
0 0 18a9 3
";

#include <string.h>

/* OPNET system definitions */
#include <opnet.h>

/* Header Block */
#include <stdbool.h>

/* packet stream definitions */

#define RCV_0_IN_STRM 0

#define RCV_1_IN_STRM 1

#define RCV_2_IN_STRM 2

#define SRC_IN_STRM 3  //simple_source

#define XMT_0_OUT_STRM 0

#define XMT_1_OUT_STRM 1

#define XMT_2_OUT_STRM 2

/* message ids*/

#define PING 1

#define PONG 2

/* initial ttl value (initial hops = 0) - for gen ping */

#define TTL_INIT 5

//for pong ttl = rcvhops+1


/* Packet format and size */

#define FORMAT_STR "khpn_packet_format"

#define PKSIZE 1024



/* transition macros */

#define SRC_ARRVL (op_intrpt_type () == \

OPC_INTRPT_STRM && op_intrpt_strm () == SRC_IN_STRM)

#define RCVR_ARRVL (op_intrpt_type () == OPC_INTRPT_STRM \

&& ((op_intrpt_strm () == RCV_0_IN_STRM) \

|| (op_intrpt_strm () == RCV_1_IN_STRM) \
```

```
||(op_intrpt_strm () == RCV_2_IN_STRM)) )


/* Ping and pong cache memory size */
#define CACHE_SIZE 100
/* cache struct, currently only ids are used */
typedef struct{
int id;
int hops;
Objid orig_Objid;
int sndr;
}msg_cache;
/* End of Header Block */


#if !defined (VOSD_NO_FIN)
#undef	BIN
#undef	BOUT
#define	BIN			FIN_LOCAL_FIELD(_op_last_line_passed) = __LINE__ - _op_block_origin;
#define	BOUT	BIN
#define	BINIT	FIN_LOCAL_FIELD(_op_last_line_passed) = 0; _op_block_origin = __LINE__;
#else
#define	BINIT
#endif /* #if !defined (VOSD_NO_FIN) */


/* State variable definitions */
typedef struct
	{
	/* Internal state tracking for FSM */
	FSM_SYS_STATE
	/* State Variables */
	int					pk_count_out			;
	int					pk_count_in			;
	Objid				own_id				;
	char				node_name[10]			;
	Objid				node_objid			;
	Stathandle			pk_cnt_out_stathandle		;
```

```
	Stathandle			pk_cnt_in_stathandle		;
	int				pong_ptr			;
	int				ping_ptr			;
	msg_cache			ping_msg_cache[CACHE_SIZE]	;
	msg_cache			pong_msg_cache[CACHE_SIZE]	;
	} khpn_process_model_state;

#define pk_count_out			op_sv_ptr->pk_count_out
#define pk_count_in			op_sv_ptr->pk_count_in
#define own_id				op_sv_ptr->own_id
#define node_name			op_sv_ptr->node_name
#define node_objid			op_sv_ptr->node_objid
#define pk_cnt_out_stathandle		op_sv_ptr->pk_cnt_out_stathandle
#define pk_cnt_in_stathandle		op_sv_ptr->pk_cnt_in_stathandle
#define pong_ptr			op_sv_ptr->pong_ptr
#define ping_ptr			op_sv_ptr->ping_ptr
#define ping_msg_cache			op_sv_ptr->ping_msg_cache
#define pong_msg_cache			op_sv_ptr->pong_msg_cache

/* These macro definitions will define a local variable called	*/
/* "op_sv_ptr" in each function containing a FIN statement.	*/
/* This variable points to the state variable data structure,	*/
/* and can be used from a C debugger to display their values.	*/
#undef FIN_PREAMBLE_DEC
#undef FIN_PREAMBLE_CODE
#define FIN_PREAMBLE_DEC	khpn_process_model_state *op_sv_ptr;
#define FIN_PREAMBLE_CODE	\
		op_sv_ptr = ((khpn_process_model_state *)(OP_SIM_CONTEXT_PTR->_op_mod_state_ptr));


/* Function Block */
#if !defined (VOSD_NO_FIN)
enum { _op_block_origin = __LINE__ + 2};
#endif


static void xmt_src_ping(void)
```

```
{
Packet * pkptr;
Packet * cp1_pkptr;
Packet * cp2_pkptr;
OpT_Packet_Id pkt_id;
//char * pkt_id_str;
FIN(xmt_src_ping());

/* get packet sent from source, set packet values */
pkptr = op_pk_get (SRC_IN_STRM);
pkt_id = op_pk_id (pkptr);
op_pk_nfd_set_pkid (pkptr, "orig_pkt_id", pkt_id);
op_pk_nfd_set_int32 (pkptr, "message_id", PING);
op_pk_nfd_set_int32 (pkptr, "ttl", TTL_INIT);
op_pk_nfd_set_int32 (pkptr, "hops", 0);
op_pk_nfd_set_objid (pkptr, "sender_objid", node_objid);

cp1_pkptr = op_pk_create_fmt (FORMAT_STR);
op_pk_total_size_set (cp1_pkptr, PKSIZE);
op_pk_nfd_set_pkid (cp1_pkptr, "orig_pkt_id", pkt_id);
op_pk_nfd_set_int32 (cp1_pkptr, "message_id", PING);
op_pk_nfd_set_int32 (cp1_pkptr, "ttl", TTL_INIT);
op_pk_nfd_set_int32 (cp1_pkptr, "hops", 0);
op_pk_nfd_set_objid (cp1_pkptr, "sender_objid", node_objid);

cp2_pkptr = op_pk_create_fmt (FORMAT_STR);
op_pk_total_size_set (cp2_pkptr, PKSIZE);
op_pk_nfd_set_pkid (cp2_pkptr, "orig_pkt_id", pkt_id);
op_pk_nfd_set_int32 (cp2_pkptr, "message_id", PING);
op_pk_nfd_set_int32 (cp2_pkptr, "ttl", TTL_INIT);
op_pk_nfd_set_int32 (cp2_pkptr, "hops", 0);
op_pk_nfd_set_objid (cp2_pkptr, "sender_objid", node_objid);

/* send packets to transmitters */
op_pk_send (pkptr, XMT_0_OUT_STRM);
```

```
		op_pk_send (cp1_pkptr, XMT_1_OUT_STRM);
		op_pk_send (cp2_pkptr, XMT_2_OUT_STRM);
		pk_count_out ++;
		pk_count_out ++;
		pk_count_out ++;
		op_stat_write (pk_cnt_out_stathandle, pk_count_out);
		printf("\n%s:pk_in %d,pk_out %d", node_name, pk_count_in, pk_count_out);
		printf("\n...gen ping: pkt_id %d, ttl %d, hops 0, sender %s", (int)pkt_id, TTL_INIT, node_name);
		printf("\n...gen ping: pkt_id %d, orig_pkt_id %d, ttl %d, hops 0, sender %s", (int)op_pk_id(cp1_pkptr), (int)pkt_id, TTL_INIT, node_name);
		printf("\n...gen ping: pkt_id %d, orig_pkt_id %d, ttl %d, hops 0, sender %s", (int)op_pk_id(cp2_pkptr), (int)pkt_id, TTL_INIT, node_name);

		/* Save to ping cache */
		ping_msg_cache[ping_ptr].id = (int) pkt_id;
		ping_msg_cache[ping_ptr].hops = 0;
		ping_msg_cache[ping_ptr].orig_Objid = node_objid;
		ping_msg_cache[ping_ptr].sndr = SRC_IN_STRM;

		ping_ptr ++;
		if(ping_ptr >= CACHE_SIZE){
			ping_ptr = 0;
		}
		printf("\n\t\tWrote to ping cache, ping_ptr=%d", ping_ptr);
		FOUT;
}
/* End of Function Block */

/* Undefine optional tracing in FIN/FOUT/FRET */
/* The FSM has its own tracing code and the other */
/* functions should not have any tracing.                    */
#undef FIN_TRACING
#define FIN_TRACING
#undef FOUTRET_TRACING
#define FOUTRET_TRACING
#if defined (__cplusplus)
extern "C" {
```

```
#endif

		void khpn_process_model (OP_SIM_CONTEXT_ARG_OPT);

		VosT_Obtype _op_khpn_process_model_init (int * init_block_ptr);

		void _op_khpn_process_model_diag (OP_SIM_CONTEXT_ARG_OPT);

		void _op_khpn_process_model_terminate (OP_SIM_CONTEXT_ARG_OPT);

		VosT_Address _op_khpn_process_model_alloc (VosT_Obtype, int);

		void _op_khpn_process_model_svar (void *, const char *, void **);

#if defined (__cplusplus)

} /* end of 'extern "C"' */

#endif


/* Process model interrupt handling procedure */

void

khpn_process_model (OP_SIM_CONTEXT_ARG_OPT)

		{

#if !defined (VOSD_NO_FIN)

		int _op_block_origin = 0;

#endif

		FIN_MT (khpn_process_model ());

			{

			FSM_ENTER ("khpn_process_model")

			FSM_BLOCK_SWITCH

				{

				/*---------------------------------------------------------*/

				/** state (init) enter executives **/

				FSM_STATE_ENTER_FORCED_NOLABEL (0, "init", "khpn_process_model [init enter execs]")

					FSM_PROFILE_SECTION_IN ("khpn_process_model [init enter execs]", state0_enter_exec)

					{

					int i = 0;


					/* Obtain the object id of the surrounding module.		*/

					own_id = op_id_self ();

					/* Obtain the parent object id and name. */

					node_objid = op_topo_parent (own_id);

					//op_ima_obj_attr_get (node_objid,  "name" , node_name);
```

```
/* Initialize state (global) variables */
pk_count_in = 0;
pk_count_out = 0;
pk_cnt_in_stathandle = op_stat_reg ("packet count in", OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
pk_cnt_out_stathandle = op_stat_reg ("packet count out", OPC_STAT_INDEX_NONE,
OPC_STAT_LOCAL);

for(i = 0; i < CACHE_SIZE; i++){
        ping_msg_cache[i].id = -1;
        pong_msg_cache[i].id = -1;
}
ping_ptr = 0;
pong_ptr = 0;
}
FSM_PROFILE_SECTION_OUT (state0_enter_exec)

/** state (init) exit executives **/
FSM_STATE_EXIT_FORCED (0, "init", "khpn_process_model [init exit execs]")

/** state (init) transition processing **/
FSM_TRANSIT_FORCE (1, state1_enter_exec, ;, "default", "", "init", "idle", "tr_0", "khpn_process_model [init ->
idle : default / ]")
/*---------------------------------------------------------*/
/** state (idle) enter executives **/
FSM_STATE_ENTER_UNFORCED (1, "idle", state1_enter_exec, "khpn_process_model [idle enter execs]")

/** blocking after enter executives of unforced state. **/
FSM_EXIT (3,"khpn_process_model")

/** state (idle) exit executives **/
FSM_STATE_EXIT_UNFORCED (1, "idle", "khpn_process_model [idle exit execs]")

/** state (idle) transition processing **/
FSM_PROFILE_SECTION_IN ("khpn_process_model [idle trans conditions]", state1_trans_conds)
FSM_INIT_COND (SRC_ARRVL)
```

```
FSM_TEST_COND (RCVR_ARRVL)

FSM_DFLT_COND

FSM_TEST_LOGIC ("idle")

FSM_PROFILE_SECTION_OUT (state1_trans_conds)

FSM_TRANSIT_SWITCH

	{

	FSM_CASE_TRANSIT (0, 1, state1_enter_exec, xmt_src_ping();, "SRC_ARRVL", "xmt_src_ping()",
"idle", "idle", "tr_2", "khpn_process_model [idle -> idle : SRC_ARRVL / xmt_src_ping()]")

	FSM_CASE_TRANSIT (1, 2, state2_enter_exec, ;, "RCVR_ARRVL", "", "idle", "procRCV", "tr_4",
"khpn_process_model [idle -> procRCV : RCVR_ARRVL / ]")

	FSM_CASE_TRANSIT (2, 1, state1_enter_exec, ;, "default", "", "idle", "idle", "tr_1",
"khpn_process_model [idle -> idle : default / ]")

	}

	/*---------------------------------------------------------*/


/** state (procRCV) enter executives **/

FSM_STATE_ENTER_FORCED (2, "procRCV", state2_enter_exec, "khpn_process_model [procRCV enter execs]")

	FSM_PROFILE_SECTION_IN ("khpn_process_model [procRCV enter execs]", state2_enter_exec)

	{

		int message_id;

			int ttl;

			int hops;

			Objid snd_id;

			Packet *pkptr;

			OpT_Packet_Id pkt_id;

			OpT_Packet_Id orig_pkt_id;

			char * sender_name="test";

			int i;

			bool duplicate_packet; //flag to indicate duplicate
ping,pong

			int input_rcvr;

			int output_xmtr;

			int output_xmtr2;


			/* get packet sent from receiver, read its values */

			input_rcvr = op_intrpt_strm ();

			pkptr = op_pk_get (input_rcvr);
```

```
pkt_id = op_pk_id (pkptr);
op_pk_nfd_get_int32 (pkptr, "message_id",
&message_id);
op_pk_nfd_get_int32 (pkptr, "ttl", &ttl);
op_pk_nfd_get_int32 (pkptr, "hops", &hops);
op_pk_nfd_get_objid (pkptr, "sender_objid",
&snd_id);
//op_ima_obj_attr_get (snd_id,  "name" ,
sender_name);
op_pk_nfd_get_pkid (pkptr, "orig_pkt_id",
&orig_pkt_id);


pk_count_in ++;
op_stat_write (pk_cnt_in_stathandle,
pk_count_in);
printf("\n%s rcv:pk_in %d,pk_out %d",
node_name, pk_count_in, pk_count_out);
printf("\n\t...received from rcvr %d: msg_id %d,
pkt_id %d, orig_pkt_id %d, ttl %d, hops %d, sender %s", input_rcvr, message_id, (int)pkt_id, (int)orig_pkt_id, ttl, hops, sender_name);



if(message_id == 1){
        //Ping,reply with pong, save, forward
to others

        //generate pong and reply
        Packet * pong_pkptr;
        pong_pkptr = op_pk_create_fmt
(FORMAT_STR);
        op_pk_total_size_set (pong_pkptr,
PKSIZE);
        op_pk_nfd_set_pkid (pong_pkptr,
"orig_pkt_id", orig_pkt_id);
        op_pk_nfd_set_int32 (pong_pkptr,
"message_id", PONG);
        op_pk_nfd_set_int32 (pong_pkptr,
"ttl", hops+1); //TTL=HOPS+1
        op_pk_nfd_set_int32 (pong_pkptr,
"hops", 0);
        op_pk_nfd_set_objid (pong_pkptr,
"sender_objid", node_objid);

        switch(input_rcvr){
```

```
                                                                case RCV_0_IN_STRM:
            output_xmtr = XMT_0_OUT_STRM;
                                                                        break;
                                                                case RCV_1_IN_STRM:
            output_xmtr = XMT_1_OUT_STRM;
                                                                        break;
                                                                case RCV_2_IN_STRM:
            output_xmtr = XMT_2_OUT_STRM;
                                                                        break;
                                                                default:
            printf("\nWarningInvalid input rcvr! Sending to XMT_0_OUT_STRM");
            output_xmtr = XMT_0_OUT_STRM;
                                                        }
                                                        op_pk_send (pong_pkptr,
output_xmtr);
                                                        printf("\n\t...gen pong to xmtr %d:
pkt_id %d, orig_pkt_id %d, ttl %d, hops 0, sender %s", output_xmtr, (int)op_pk_id(pong_pkptr), (int)pkt_id, hops+1, node_name);
                                                        pk_count_out ++;

                                                        /* detect if this is a duplicate */
                                                        duplicate_packet = false;
                                                        for(i = 0; i < CACHE_SIZE; i++){
                                                                if(
            ping_msg_cache[i].id == (int) orig_pkt_id){
            duplicate_packet = true;
                                                                        }
                                                        }

                                                        if(duplicate_packet){
                                                                //duplicate ping
                                                                op_pk_destroy (pkptr);
                                                                printf("\n\t....destroy
duplicate ping packet");
                                                        }
```

```
                                                                    else{
                                                                        /* Save to ping cache */
        ping_msg_cache[ping_ptr].id = (int) orig_pkt_id;
        ping_msg_cache[ping_ptr].hops = hops;
        ping_msg_cache[ping_ptr].orig_Objid = snd_id;
        ping_msg_cache[ping_ptr].sndr = input_rcvr;

                                                                        ping_ptr ++;
                                                                        if(ping_ptr >=
CACHE_SIZE){
                                                                            ping_ptr = 0;
                                                                            }
                                                                        printf("\n\t\tWrote ping
to ping cache, ping_ptr=%d", ping_ptr);

                                                                        /* forward ping to others
if ttl ok*/
                                                                        if(ttl > 1){
        switch(input_rcvr){
        case RCV_0_IN_STRM:
        output_xmtr = XMT_1_OUT_STRM;
        output_xmtr2 = XMT_2_OUT_STRM;
        break;
        case RCV_1_IN_STRM:
        output_xmtr = XMT_0_OUT_STRM;
        output_xmtr2 = XMT_2_OUT_STRM;
        break;
        case RCV_2_IN_STRM:
        output_xmtr = XMT_0_OUT_STRM;
```

```
                output_xmtr2 = XMT_1_OUT_STRM;

                break;

                default:

                printf("\nWarningInvalid input rcvr! Sending to XMT_1_OUT_STRM,XMT_2_OUT_STRM");

                output_xmtr = XMT_1_OUT_STRM;

                output_xmtr2 = XMT_2_OUT_STRM;
                                                                                                                                                }

                op_pk_nfd_set_int32 (pkptr, "ttl", ttl-1);

                op_pk_nfd_set_int32 (pkptr, "hops", hops+1);
                                                                                                                                                op_pk_send
(pkptr, output_xmtr);

               printf("\n\t...fwd ping to xmtr %d: orig_pkt_id %d, ttl %d, hops %d, sender %s", output_xmtr, (int)orig_pkt_id, ttl-1, hops+1, sender_name);

                pk_count_out ++;


                                                                                                                                                Packet *
cp1_pkptr;
                                                                                                                                                cp1_pkptr =
op_pk_create_fmt (FORMAT_STR);

                op_pk_total_size_set (cp1_pkptr, PKSIZE);

                op_pk_nfd_set_pkid (cp1_pkptr, "orig_pkt_id", orig_pkt_id);

                op_pk_nfd_set_int32 (cp1_pkptr, "message_id", PING);

                op_pk_nfd_set_int32 (cp1_pkptr, "ttl", ttl-1);

                op_pk_nfd_set_int32 (cp1_pkptr, "hops", hops+1);

                op_pk_nfd_set_objid (cp1_pkptr, "sender_objid", snd_id);
                                                                                                                                                op_pk_send
(cp1_pkptr, output_xmtr2);

                printf("\n\t...gen ping to xmtr %d: orig_pkt_id %d, ttl %d, hops %d, sender %s", output_xmtr2, (int)orig_pkt_id, ttl-1, hops+1,
sender_name);

                pk_count_out ++;
```

```
                    }
                }
            }
            else{
                //Pong, fwd , save

                /* detect if this is a
duplicate */
                duplicate_packet = false;
                for(i = 0; i < CACHE_SIZE;
i++){
                    if(
                    pong_msg_cache[i].id == (int) orig_pkt_id){
                    duplicate_packet = true;
                    }
                }

                if(duplicate_packet){
                    //duplicate
pong
                    op_pk_destroy (pkptr);
                    printf("\n\t....destroy duplicate pong packet");
                }
                else{
                    /* Save to
pong cache */
                    pong_msg_cache[pong_ptr].id = (int) orig_pkt_id;
                    pong_msg_cache[pong_ptr].hops = hops;
                    pong_msg_cache[pong_ptr].orig_Objid = snd_id;
                    pong_msg_cache[pong_ptr].sndr = input_rcvr;

                    pong_ptr ++;
                    if(pong_ptr
>= CACHE_SIZE){
                    pong_ptr = 0;
```

```
}

printf("\n\t\tWrote pong to pong cache, pong_ptr=%d", pong_ptr);

/* forward pong to others if ttl ok*/
if(ttl > 1){

switch(input_rcvr){

case RCV_0_IN_STRM:

output_xmtr = XMT_1_OUT_STRM;

output_xmtr2 = XMT_2_OUT_STRM;

break;

case RCV_1_IN_STRM:

output_xmtr = XMT_0_OUT_STRM;

output_xmtr2 = XMT_2_OUT_STRM;

break;

case RCV_2_IN_STRM:

output_xmtr = XMT_0_OUT_STRM;

output_xmtr2 = XMT_1_OUT_STRM;

break;

default:

printf("\nWarningInvalid input rcvr! Sending to XMT_1_OUT_STRM,XMT_2_OUT_STRM");

output_xmtr = XMT_1_OUT_STRM;

output_xmtr2 = XMT_2_OUT_STRM;

}

op_pk_nfd_set_int32 (pkptr, "ttl", ttl-1);

op_pk_nfd_set_int32 (pkptr, "hops", hops+1);
```

```
op_pk_send (pkptr, output_xmtr);

printf("\n\t...fwd pong to xmtr %d: orig_pkt_id %d, ttl %d, hops %d, sender %s", output_xmtr, (int)orig_pkt_id, ttl-1, hops+1, sender_name);

pk_count_out ++;

Packet * cp2_pkptr;

cp2_pkptr = op_pk_create_fmt (FORMAT_STR);

op_pk_total_size_set (cp2_pkptr, PKSIZE);

op_pk_nfd_set_pkid (cp2_pkptr, "orig_pkt_id", orig_pkt_id);

op_pk_nfd_set_int32 (cp2_pkptr, "message_id", PONG);

op_pk_nfd_set_int32 (cp2_pkptr, "ttl", ttl-1);

op_pk_nfd_set_int32 (cp2_pkptr, "hops", hops+1);

op_pk_nfd_set_objid (cp2_pkptr, "sender_objid", snd_id);

op_pk_send (cp2_pkptr, output_xmtr2);

printf("\n\t...gen pong to xmtr %d: orig_pkt_id %d, ttl %d, hops %d, sender %s", output_xmtr2, (int)orig_pkt_id, ttl-1, hops+1, sender_name);

pk_count_out ++;

}

}

}

op_stat_write (pk_cnt_out_stathandle, pk_count_out);

}

FSM_PROFILE_SECTION_OUT (state2_enter_exec)


/** state (procRCV) exit executives **/

FSM_STATE_EXIT_FORCED (2, "procRCV", "khpn_process_model [procRCV exit execs]")


/** state (procRCV) transition processing **/

FSM_TRANSIT_FORCE (1, state1_enter_exec, ;, "default", "", "procRCV", "idle", "tr_3", "khpn_process_model [procRCV -> idle : default / ]")
```

```
                                        /*---------------------------------------------------------*/

                                }

                        FSM_EXIT (0,"khpn_process_model")

                        }

        }


void

_op_khpn_process_model_diag (OP_SIM_CONTEXT_ARG_OPT)

        {

        /* No Diagnostic Block */

        }

void

_op_khpn_process_model_terminate (OP_SIM_CONTEXT_ARG_OPT)

        {


        FIN_MT (_op_khpn_process_model_terminate ())


        /* No Termination Block */

        Vos_Poolmem_Dealloc (op_sv_ptr);

        FOUT

        }

/* Undefine shortcuts to state variables to avoid */

/* syntax error in direct access to fields of */

/* local variable prs_ptr in _op_khpn_process_model_svar function. */

#undef pk_count_out

#undef pk_count_in

#undef own_id

#undef node_name

#undef node_objid

#undef pk_cnt_out_stathandle

#undef pk_cnt_in_stathandle

#undef pong_ptr

#undef ping_ptr

#undef ping_msg_cache

#undef pong_msg_cache
```

```
#undef FIN_PREAMBLE_DEC

#undef FIN_PREAMBLE_CODE

#define FIN_PREAMBLE_DEC

#define FIN_PREAMBLE_CODE

VosT_Obtype

_op_khpn_process_model_init (int * init_block_ptr)

		{

		VosT_Obtype obtype = OPC_NIL;

		FIN_MT (_op_khpn_process_model_init (init_block_ptr))


		obtype = Vos_Define_Object_Prstate ("proc state vars (khpn_process_model)",

			sizeof (khpn_process_model_state));

		*init_block_ptr = 0;

		FRET (obtype)

		}

VosT_Address

_op_khpn_process_model_alloc (VosT_Obtype obtype, int init_block)

		{

#if !defined (VOSD_NO_FIN)

		int _op_block_origin = 0;

#endif

		khpn_process_model_state * ptr;

		FIN_MT (_op_khpn_process_model_alloc (obtype))

		ptr = (khpn_process_model_state *)Vos_Alloc_Object (obtype);

		if (ptr != OPC_NIL)

			{

			ptr->_op_current_block = init_block;

#if defined (OPD_ALLOW_ODB)

			ptr->_op_current_state = "khpn_process_model [init enter execs]";

#endif

			}

		FRET ((VosT_Address)ptr)

		}

void

_op_khpn_process_model_svar (void * gen_ptr, const char * var_name, void ** var_p_ptr)
```

```
{
khpn_process_model_state                    *prs_ptr;

FIN_MT (_op_khpn_process_model_svar (gen_ptr, var_name, var_p_ptr))

if (var_name == OPC_NIL)
        {
        *var_p_ptr = (void *)OPC_NIL;
        FOUT
        }
prs_ptr = (khpn_process_model_state *)gen_ptr;
if (strcmp ("pk_count_out" , var_name) == 0)
        {
        *var_p_ptr = (void *) (&prs_ptr->pk_count_out);
        FOUT
        }
if (strcmp ("pk_count_in" , var_name) == 0)
        {
        *var_p_ptr = (void *) (&prs_ptr->pk_count_in);
        FOUT
        }
if (strcmp ("own_id" , var_name) == 0)
        {
        *var_p_ptr = (void *) (&prs_ptr->own_id);
        FOUT
        }
if (strcmp ("node_name" , var_name) == 0)
        {
        *var_p_ptr = (void *) (prs_ptr->node_name);
        FOUT
        }
if (strcmp ("node_objid" , var_name) == 0)
        {
        *var_p_ptr = (void *) (&prs_ptr->node_objid);
        FOUT
```

```
        }
if (strcmp ("pk_cnt_out_stathandle" , var_name) == 0)
        {
        *var_p_ptr = (void *) (&prs_ptr->pk_cnt_out_stathandle);
        FOUT
        }
if (strcmp ("pk_cnt_in_stathandle" , var_name) == 0)
        {
        *var_p_ptr = (void *) (&prs_ptr->pk_cnt_in_stathandle);
        FOUT
        }
if (strcmp ("pong_ptr" , var_name) == 0)
        {
        *var_p_ptr = (void *) (&prs_ptr->pong_ptr);
        FOUT
        }
if (strcmp ("ping_ptr" , var_name) == 0)
        {
        *var_p_ptr = (void *) (&prs_ptr->ping_ptr);
        FOUT
        }
if (strcmp ("ping_msg_cache" , var_name) == 0)
        {
        *var_p_ptr = (void *) (prs_ptr->ping_msg_cache);
        FOUT
        }
if (strcmp ("pong_msg_cache" , var_name) == 0)
        {
        *var_p_ptr = (void *) (prs_ptr->pong_msg_cache);
        FOUT
        }
*var_p_ptr = (void *)OPC_NIL;
FOUT
}
```