ENSC 427

Spring 2009

Implementing the BitTorrent Protocol in OPNET with a Leech Blocking Algorithm

Group 7

Pavel Bloch	pbloch@sfu.ca
Gondang Prabowo Yudo	gpy1@sfu.ca
Amir Kamyabnejad	aka39@sfu.ca

ABSTRACT

The concept of peer-to-peer (P2P) file sharing has been around for a decade, it started with the invention of Napster in 1999. Since then the popularity of the concept has grown considerably. The advantage of P2P networks is that they require minimum (possibly none) server bandwidth. The clients themselves are the servers. Currently there are two types of P2P file sharing protocols that dominate the P2P file sharing network, the Gnutella protocol (LimeWire, ShareAza) and BitTorrent (Utorrent,Vuze,BitTorrent). For our project, we will take a look at the BitTorrent protocol more closely by constructing the network and simulating the mechanism in OPNET. Furthermore we will also implement a leech blocking algorithm in the network to increase the throughput to the peers.

TABLE OF CONTENTS

1.0 INTRODUCTION

File sharing using peer-to-peer (P2P) networks began in 1999 with the invention of Napster. Since then the popularity of P2P network has grown exponentially. Some even say in the future, P2P network might replace the standard client-server network. For client server networks, all file requests from different clients go to a dedicated server. The server then processes each request and sends the correct file to each of the requesting clients. The problem with this network is with an increasing number of clients the server runs a risk of getting overloaded and thus slowing down the speed of downloads. On the other hand, in a P2P network the client itself is the server. This implies that every client in the network is able to prepare, request, and transmit any type of data over the network. The omission of a dedicated server in P2P networks greatly reduces operational cost. Another advantage of P2P is the bandwidth of data transmission, as more clients connect the probability of successful connection increases exponentially. Currently there are 2 types of P2P protocols that dominate the internet: the Gnutella protocol (KaZaa, LimeWire) and the more recent BitTorrent protocol (BitTorrent, Azureus).

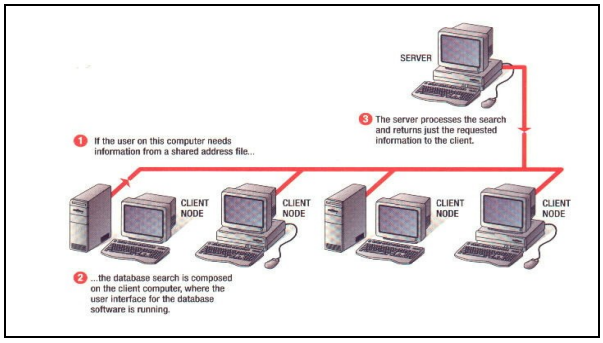


Figure 1.1 - Client Server Network

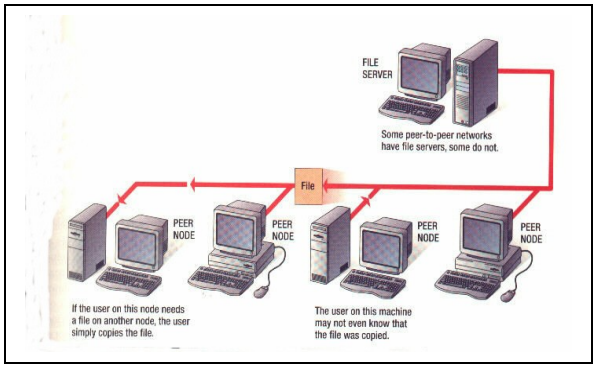


Figure 1.2 - Peer to Peer Network

As avid users of the BitTorrent file sharing protocol, we thought it was more interesting to make a project about BitTorrent. For our project we have 2 goals, first construct and simulate the BitTorrent file sharing protocol in OPNET and secondly implement a leech blocking algorithm in the router to improve packets goodput. The next section will dive into more detail about the BitTorrent file sharing protocol and the leech blocking algorithm.

2.0 THEORY

2.1 BitTorrent Protocol

The BitTorrent protocol and application was invented by Bram Cohen in 2001. Since then the number of existing BitTorrent applications has increased considerably. The protocol defines two types of clients, **seeds** and **peers**. Seeds are clients that hold the full files and peers are the clients that download files as well as share the incomplete downloaded files with other peers. Diagram 3.1.1 below shows the seeds and the peers.

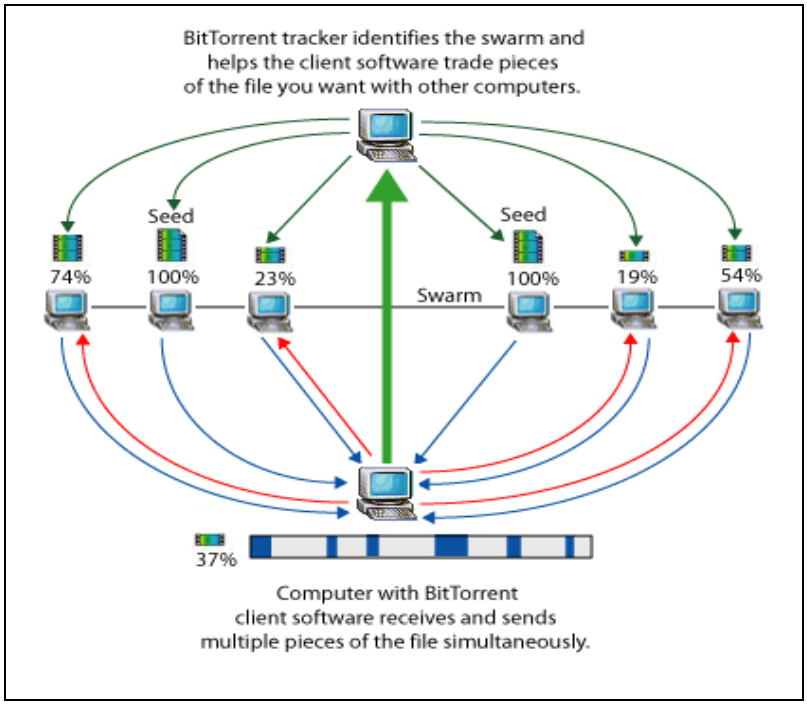


Figure 2.1.1. *Peers and Seeds in a BitTorrent Network*

Before a peer can start downloading, it must acquire a **.torrent file** (eg. Data torrent). The .torrent file contains the Metadata about the BitTorrent file as well as the **tracker** information. A Metadata is a special file that contains information about the tracker. A tracker is a dedicated computer that oversees and coordinates the file distribution. For the file transport between peers, the BitTorrent protocol uses a slightly modified **HTTP** file transfer protocol. A normal HTTP only sends request to a single TCP socket while the BitTorrent HTTP sends many small data requests over different TCP sockets.

2.2 Leech Blocking Algorithm

In the BitTorrent protocol a leech is defined as a peer that has a poor share ratio (below 0.5). The problem with leeching is that they slow down the network speed. To alleviate this problem we decided to design an algorithm **inside the seed and peer nodes** to block the leeches. To simplify the problem we decided to give leeches a share ratio of 0, which means they only request data but do not share at all.

The general idea is that the seeder node detects all incoming packets and measures the share ratio (ratio between uploaded data and downloaded data) for every node connected to the seeder. Instead of emulating the share ratio we will check if the packet destination address is the leecher node and if this is true the seed or peer node will either suspend or block the BitTorrent data transmission to that particular node. The underlying FTP traffic would not be affected by this algorithm.

2. 3 Leech Detection Methods

There are a variety of methods to detect the leech in the seeds and peers nodes. Each method has its own advantages and disadvantages. The four predominant methods are: Crawlers, Network Flow, Port Blocking, and Packet Filtering. A brief description of each method as well as its advantage /disadvantage will be discussed in this section, as the related work is the subject of our interest. Table 2.3.1 below shows an overview of the different detection methods. We chose the packet filtering method for our model because it is simpler to implement in OPNET.

	Description	Advantage	Disadvantage
Crawlers	*Crawlers join the P2P network like a client, learns system structure, IP addresses etc.*	*Can identify all IPs in P2P network with high accuracy.*	*High resource usage (CPU & memory).* *Detects and block only one specific P2P network*
Network Flow	*Network Flow characterizes network activity with several parameters (Host distribution, bandwidth, traffic pattern, topology, and connection duration).*	*Works for all P2P networks, changing P2P network won't affect this method's success.*	*Complicated.* *Cannot block specific P2P protocol while enabling another.*
Port Blocking	*Port Blocking Blocks certain port used by P2P networks. HTTP servers use port 80, FTP server use port 23.*	*Simple and fast.*	*P2P network use dynamic port allocation. High Probability to block other connection*
Packet Filtering	*Inspect each packet and compare contents (Header) to known patterns.*	*Fast, simpler than crawlers and network flow.* *Low probability of blocking other connections.*	*Work only for single P2P network* *Uses only specific parts of the protocol.*

Table 2.3.1. Packet Detection Method

3.0 OPNET IMPLEMENTATION

In this section we will discuss the OPNET implementation of our project.

3.0 Purpose

The main purpose of our project is to improve BitTorrent performance by increasing the goodput (in bits/sec) of the model. Here, we define goodput as good BitTorrent throughput. That is, goodput consists of BitTorrent reply packets (BitTorrent data packets) that are destined for the BitTorrent peers in our model. We will achieve our main goal in three steps.

First, we will run our baseline BitTorrent implementation with no modifications. Then, we will cut off (block) BitTorrent reply packets destined for the leech nodes, and analyze the results. Finally, we will modify the blocking of leech nodes by routing BitTorrent reply packets to the peer nodes instead. This way, overall throughput in the system will remain the same (as well as channel utilization) but the goodput will increase considerably.

Our second goal is to decrease the channel utilization of certain specific links in our model. We will demonstrate the effect leech blocking has on the utilization of the leech<->router links. The leech blocking is not dynamic or timed, so this will serve as a simple demonstration of channel utilization monitoring.

Finally, we will run another test using a more realistic traffic pattern distribution and analyze any differences in results. For the first goals, our traffic will be governed by the Poisson distribution with a mean packet interarrival time of 1 sec. The last test will be run with a Pareto distribution with a distribution of mean 1 second and shape 1.

3.1 Our Model Assumptions

Several assumptions were made in our implementation. First, we defined leeches as peers that do not reply to BitTorrent requests. Second, we chose to not implement any transmission control, but instead created a free-for-all type of packet generation at each node. This allowed for a much simpler design. As well, no tracker or metadata files are created. The address to each node is fixed and known, so our design is static in nature. The packet generation follows a Poisson distribution (a Pareto distribution will later be tested). Because there is no transmission control (replying to requests), in the leech routing scenario (will be discussed later), we chose to simply redirect any BT reply packet going to the leech node to see node.

3.1 Scenarios

The network topology created to implement BitTorrent on OPNET is shown in Figure 3.1.1, below. The network consists of an FTP server, a central router, and six nodes at various arbitrary distances from the router. It should be noted that each node generates a combination of BitTorrent and FTP traffic. For example, seed nodes reply to BitTorrent requests and also request and FTP data.

Three duplicate scenarios with three node process models have been created for our project. The difference between the three scenarios is the process model used by the nodes (seeds, peers, and leeches all share the same process model). The first scenario is our baseline. In the baseline, a simplified version of the BitTorrent protocol is implemented and run. The second scenario is the leech blocking case. In this scenario, BitTorrent requests coming from the leech nodes "go unanswered" by the seeds and peers. The actual implementation will be described in the process model section. Finally, the third scenario is the leech routing case. Here, the "blocked" BitTorrent reply packets destined to the leech nodes are instead routed to the peer nodes. This is our simple solution to mimic the network controlling the bandwidth allocated to the leeches.

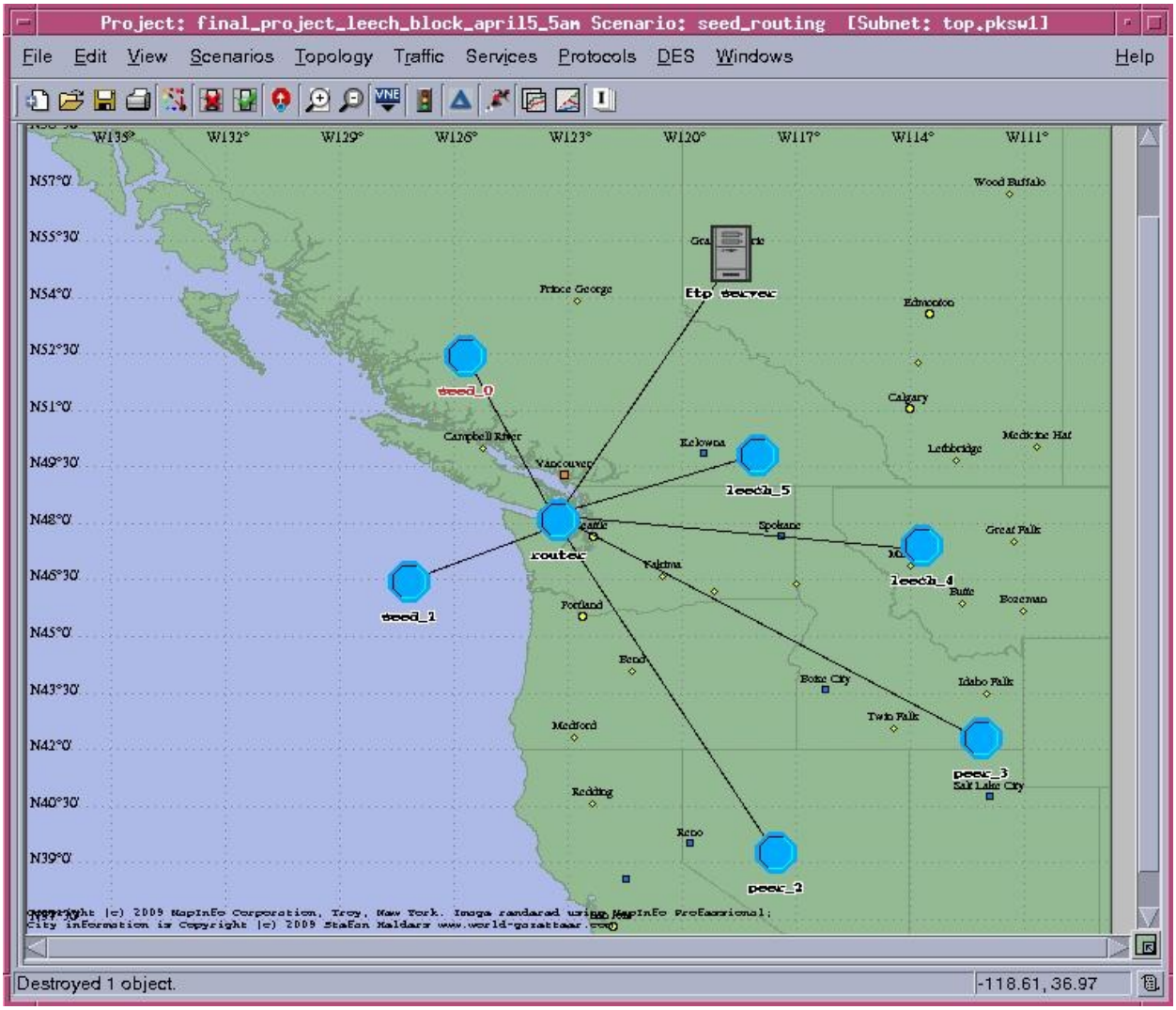


Figure 3.1.1. *BitTorrent network topology in OPNET*

3.2 Packet Formats

Four different types of packet formats are used for our BitTorrent network: *bt_req_packet*, *ftp_req_packet*, and *bt_reply_packet* and *ftp_reply_packet*. Each packet format serves a different

purpose. The *bt_req_packet* format emulates the BitTorrent Packet request sent from the peers and leeches to the seeds or other peers. This packet format consists of a 2-bit protocol field, a 4-bit source address field, and a 4-bit destination address field. The protocol field is used by the router to distinguish between the different types of packets. A '00' (0) is an ftp_request, a '01' (1) is a bt_request, a '10' (2) is a bt_reply, and a '11'(3) is an ftp_reply. The actual size of this packet is 24 bytes and is set at the node model. The form is shown in Figure 3.2.1 below.

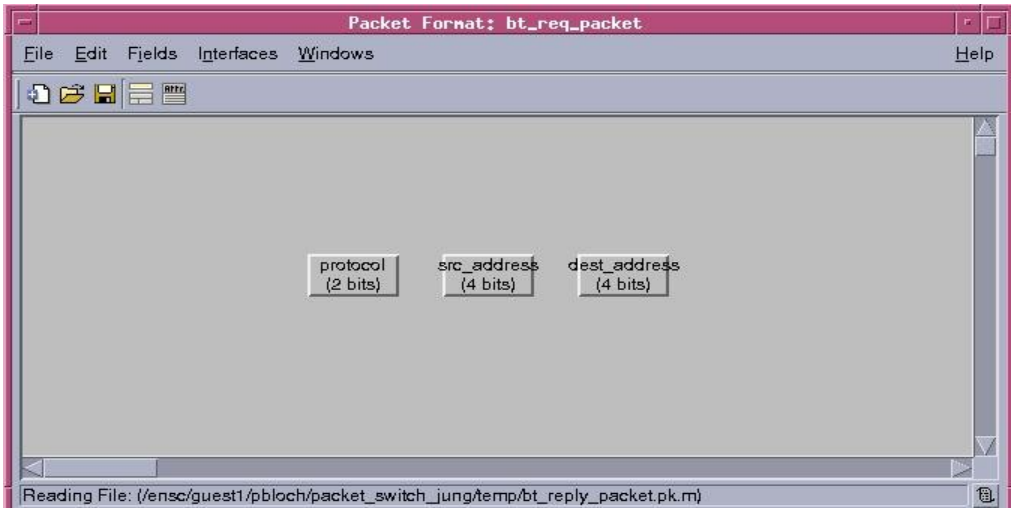


***Figure 3.2.1**. bt_req_packet.*

The *bt_reply_packet* format is used for BitTorrent packet replies to peers and leeches. Seeds do not receive BitTorrent replies. This packet format consists of a 2-bit protocol field, a 4-bit source address, a 4-bit destination address, and a 32-bit data field. This data field is for appearances only and does not hold actual data. The actual packet size is 1024 bytes (set by us at the node model). This packet format is shown in Figure 3.2.3, below.

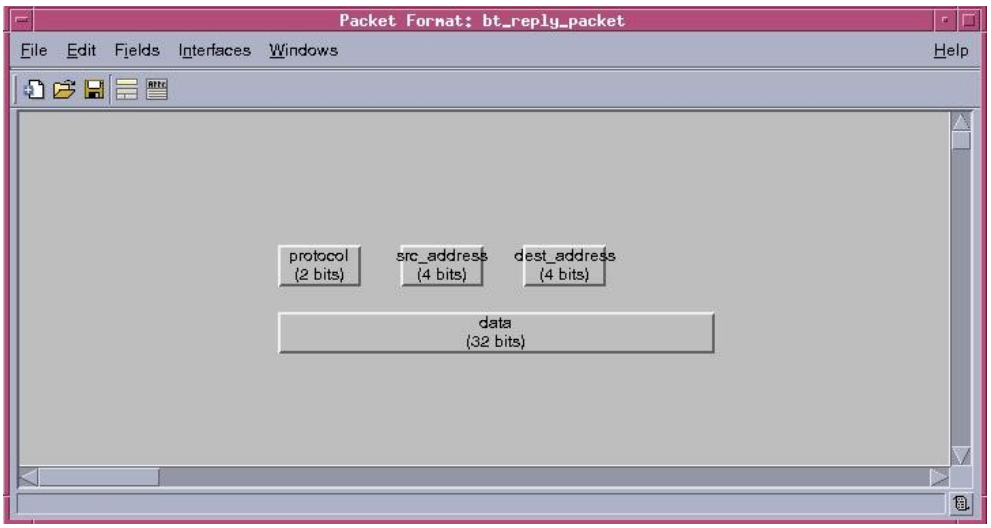


***Figure 3.2.2**. bt_reply_packet*

The *ftp_req_packet* format is used by all nodes to request FTP data from the FTP server. This packet format consists of a 2-bit protocol field, a 4-bit source address, and a 4-bit destination. The actual size of this packet is 24 bytes and is set by us at the node model. This packet format is shown in Figure 3.2.2, below.

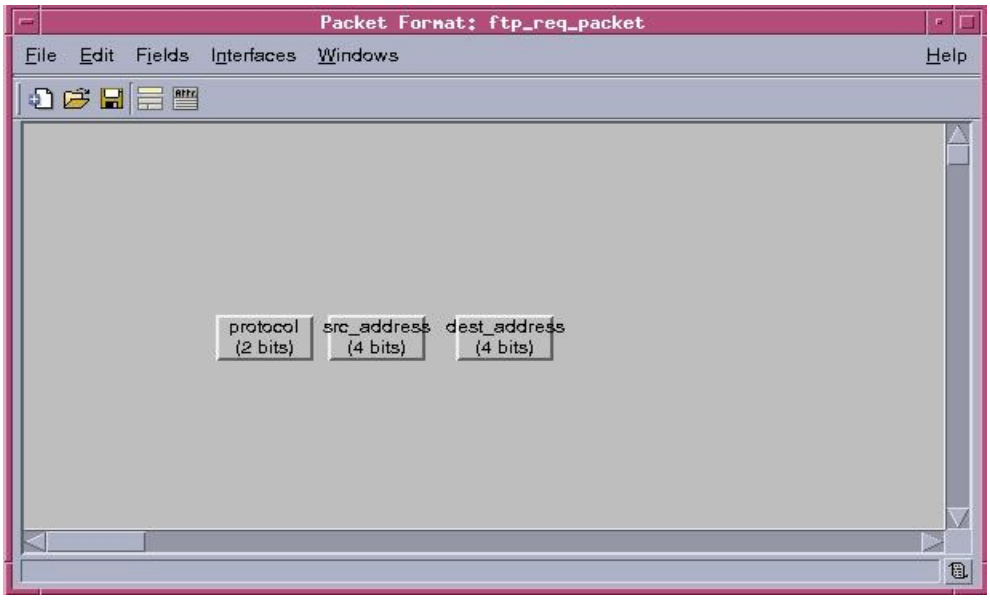


Figure 3.2.3. *ftp_req_packet*

The *ftp_reply_packet* format is used by the FTP serve to reply to seeds, peers and leeches with FTP data. This packet format consists of a 2-bit protocol field, a 4-bit source address, a 4-bit destination address, and a 32-bit data field. This data field is for appearances only and does not hold actual data. The actual packet size is 1024 bytes (set by us at node model). This packet format is shown in Figure 3.2.3, below.

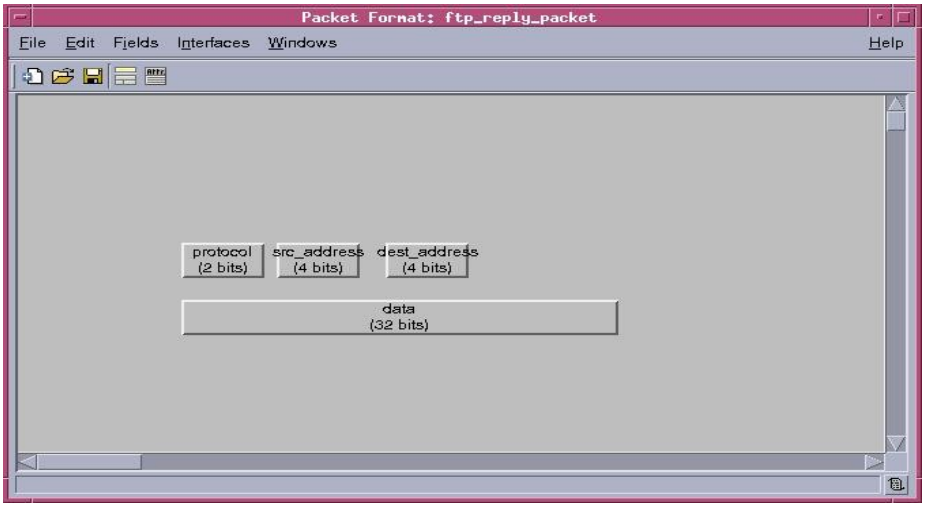


Figure 3.2.4. *ftp_reply_packet*

3.3 Node Models

Four types of node models were designed for our project, one for every type of node: *seeder node*, *peer node*, *leecher node,* and *ftp server node*. The seeder nodes generate ftp_request and bt_reply packets. Peer nodes generate ftp_request, bt_request, and bt_reply packets. The leecher nodes generate only ftp_request and bt_request packets. And lastly the ftp server only generates ftp_reply packets.

In addition, we adjusted the rate of BitTorrent packet creation (packet interrival time) at packet source module attributes in the node models. The bt_reply_packet generation rate for seeder nodes is greater than the bt_reply_packet generation rate for peer nodes.

The seeder node model that was made in OPNET consists of a processor, receiver, transmitter, as well as two simple sources that generate ftp_req and bt_reply packets. This topology is shown in Figure 3.3.1, below.

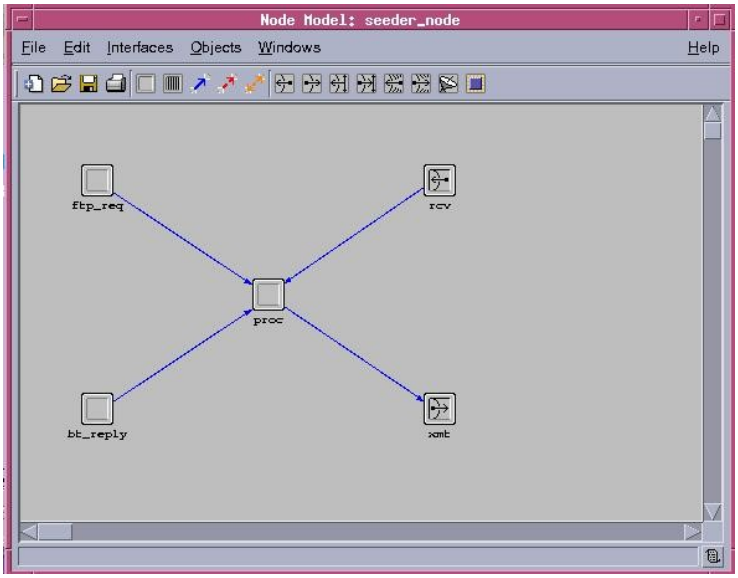


Figure 3.3.1. *Seeder node model.*

The leecher node model is created in the same manner as the seeder model. However, the leech does not reply to BitTorrent requests (in other words, leechers do not BitTorrent data) but only requests for BitTorrent data instead. This model is shown in Figure 3.2.2 where the leecher model uses the bt_req instead of the bt_reply simple sources.

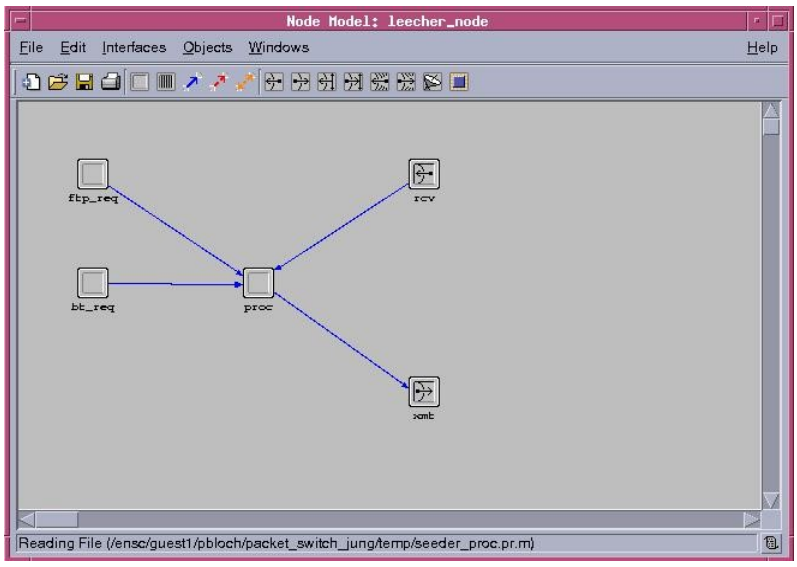


Figure 3.3.2. *Leecher node model.*

The peer node model contains three simple sources. Peers request and reply for BitTorrent data as well as request FTP data. This model is shown in Figure 3.2.3.

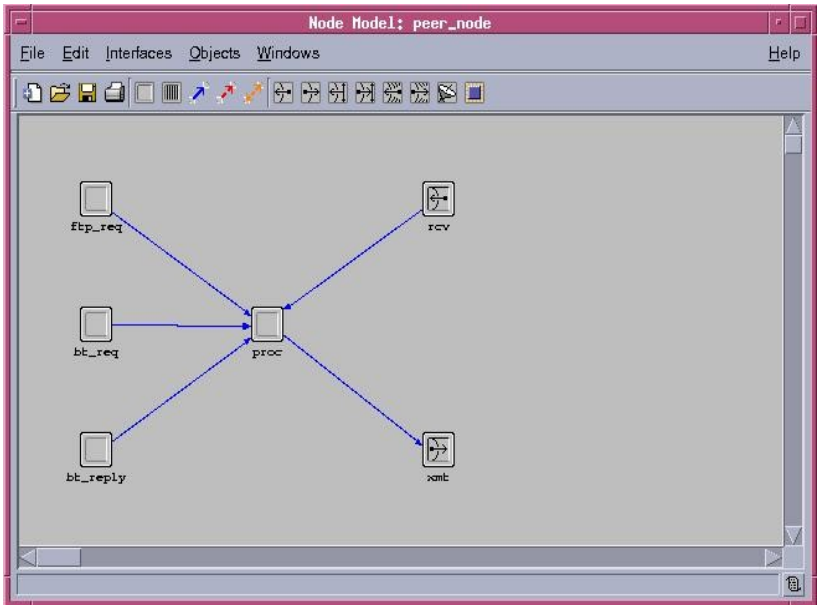


Figure 3.3.3. *Peer node model*

Our custom FTP server consists of a processor, receiver, transmitter and a simple source generating *ftp_reply_packet* packets. This topology is shown in Figure 3.2.4, below.

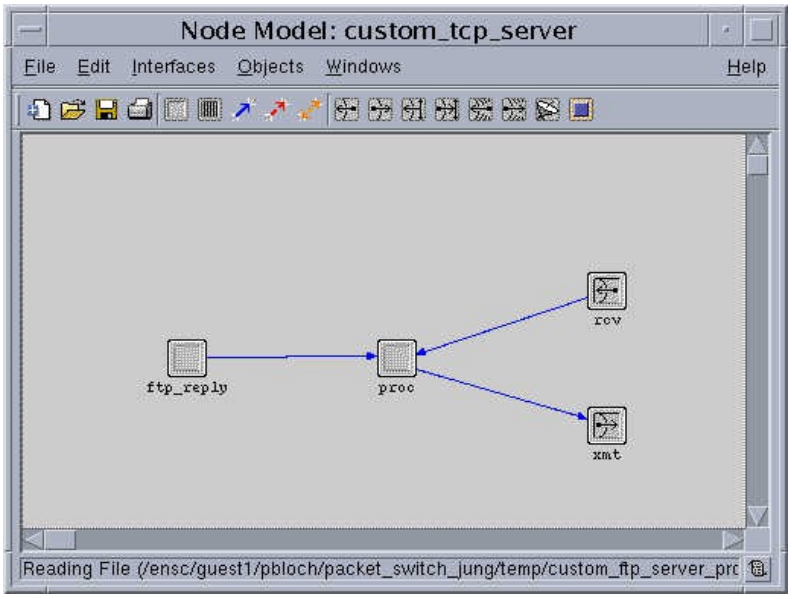


Figure 3.3.4. *Custom tcp server*.

Lastly, the node model for the hub was created. This model consisted of eight receivers and eight transmitters and a central processor called hub. This topology is shown in Figure 3.3.5, below.

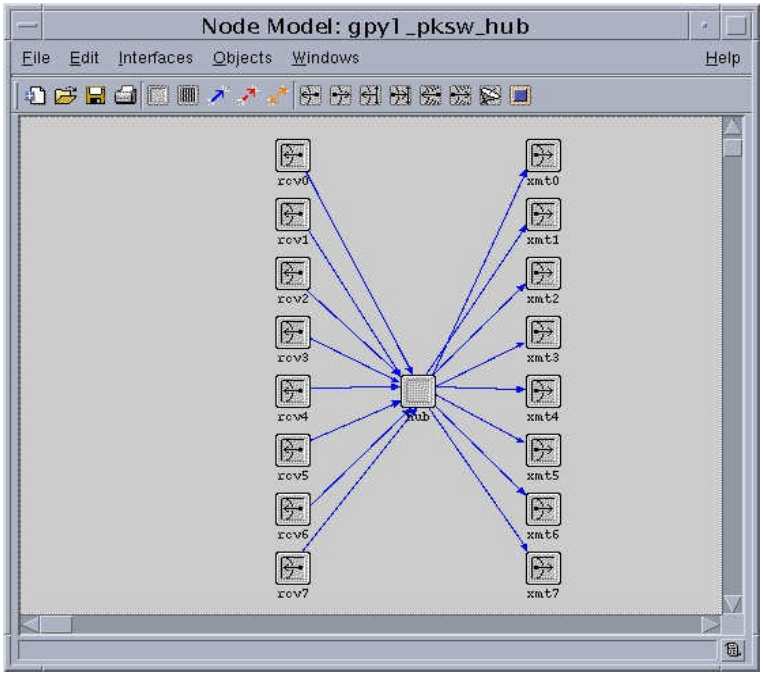


Figure 3.3.5. *Hub model.*

3.4 Process Models

As mentioned in section 3.1, we created a different process model to be used by the nodes (seeds, peers, and leeches) for each scenario tested. That makes three process models for these nodes in total. The baseline and leech routing process models are nearly identical and these will be discussed as one. The leech blocking process model warrants its own separate discussion.

In addition, we created process models for the central router, and our custom ftp server. These models do not change regardless of the scenario being tested.

3.4.1 Seeder, Peer, Leech: Baseline Scenario and Leech Routing Scenario Process Models

The seeder process model consists of two states: an initial state and idle state. The initial state only has one connection to the idle state. The idle has state has five transitions for five possible interrupts from channel streams: default, source arrival (FTP packet request), source arrival 2 (BT packet request), source arrival 3 (BT packet reply), and a packet arrival from the receiver. Each interrupt causes a separate Interrupt Service Routine /Function (ISR) to execute. This model is illustrated in Figure 3.4.1.

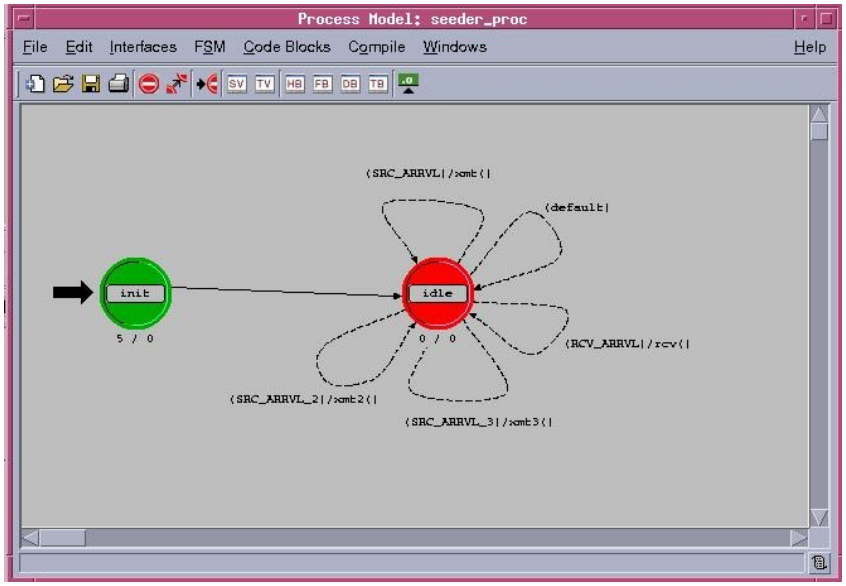


Figure 3.4.1.1 *Seeder, Peer, Leech process model.*

Three channels were created in the Header block to accommodate the three possible simple sources in the node models. The packet destination address is assigned a random value in the xmt() functions according to a uniform distribution. The State Variable block below shows the Distribution assignments for the destination addresses. The bottom two pictures show the Enter Executives of the init process. The destination addresses for each different packet consists of a uniform distribution with a lower and upper bound. Figure 3.4.4 shows the Enter Execs for the Baseline Scenario while Figure 3.4.5 shows the Enter Execs for the Leech Routing Scenario. The difference between the two Execs is the range for destination addresses made available for the BitTorrent reply packet. **In the Leech Routing scenario, BT replies only travel to the peer nodes. In the Baseline scenario, BT replies travel to leech nodes as well.**

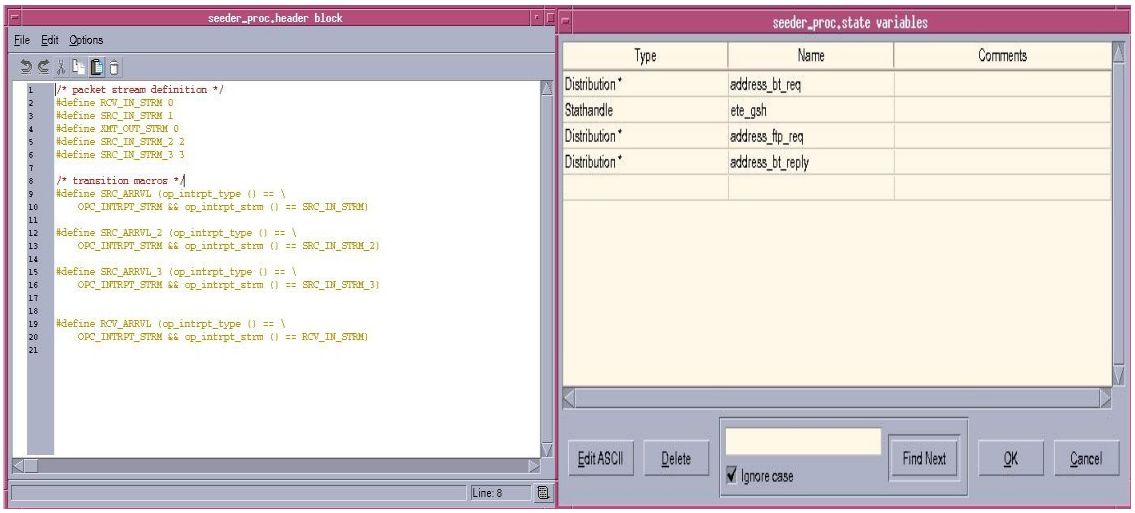


Figure 3.4.1.2 *Header Block for all scenarios* ***Figure 3.4.1.3*** *State Variables for all scenarios*

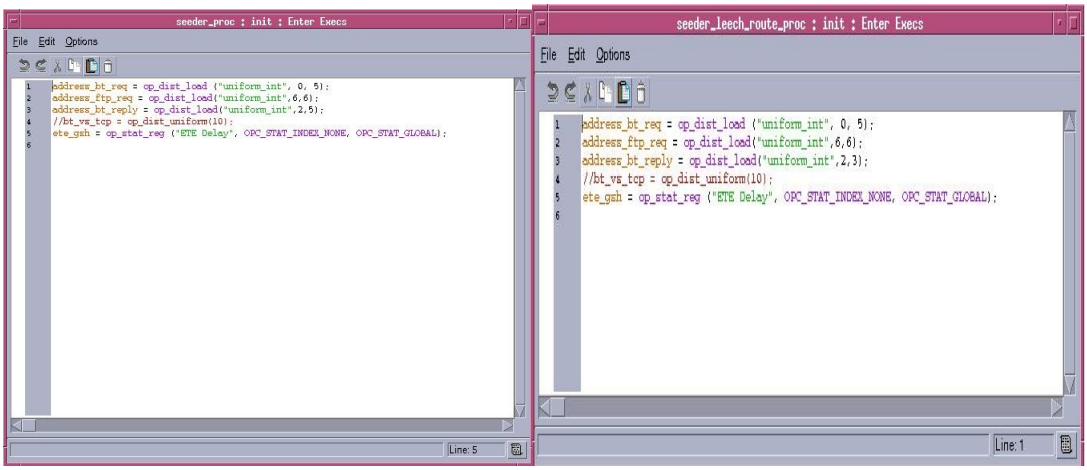


Figure 3.4.1.4 *Enter Execs for Baseline Scenario* ***Figure 3.4.1.5*** *Enter Execs for Routing Scenario*

3.4.2 Seeder, Peer, Leech: Leech Blocking Scenario Process Model

The process model for the nodes in the Leech Blocking Scenario differs from the above two scenarios in its transmitting function. In the leech blocking scenario, a condition checks for whether or not the BitTorrent reply packet is destined for the leech nodes, and if the condition is true the packet is destroyed.

3.4.3 Central Router Process Model (Any Scenario)

Furthermore a process model for the router is also created. It consists of 2 states: the st_2 and idle state. Two transitions are used in this model, the default and the *PK_ARRVL/route_pk ().* This model is shown in Figure 3.4.2, below.

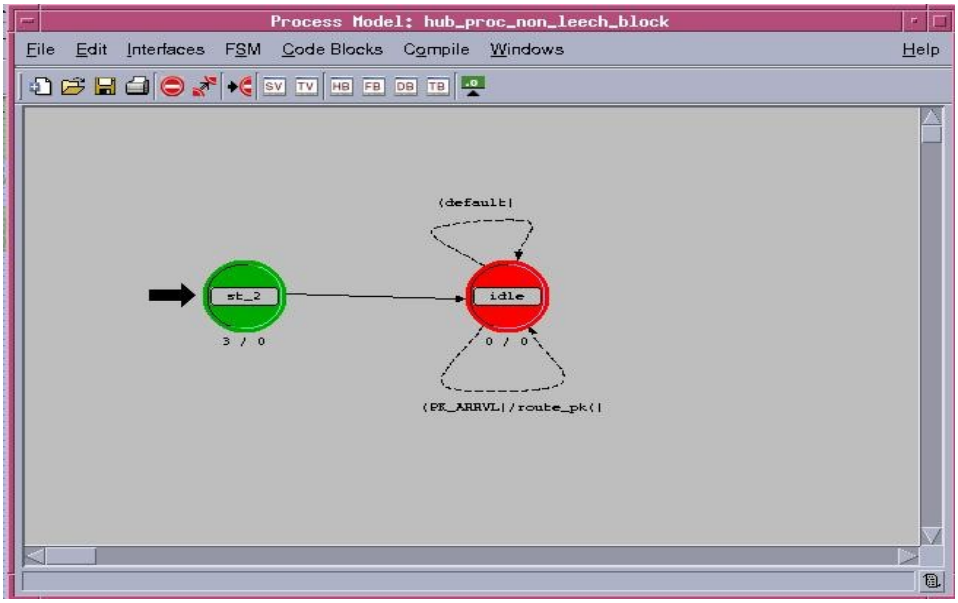
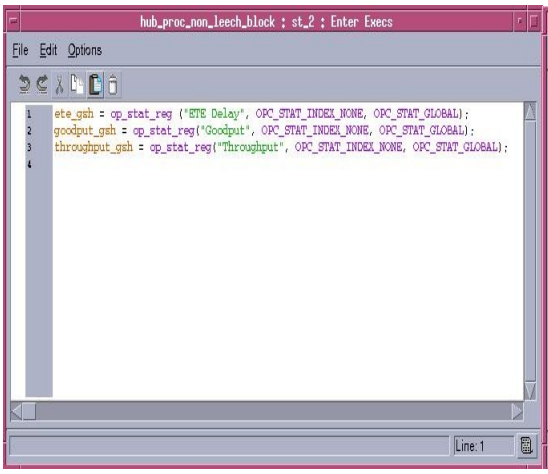


Figure 3.4.3.1 *Hub process model.*

In the Enter execs, three global statistics are created and are later collected in the simulation (ETE delay, Goodput, and Throughput). In the Header block, two global variables are created to count the number of bits the router encounters (throughput and goodput). These are used to output the updated values to the global variables using op_stat_write().



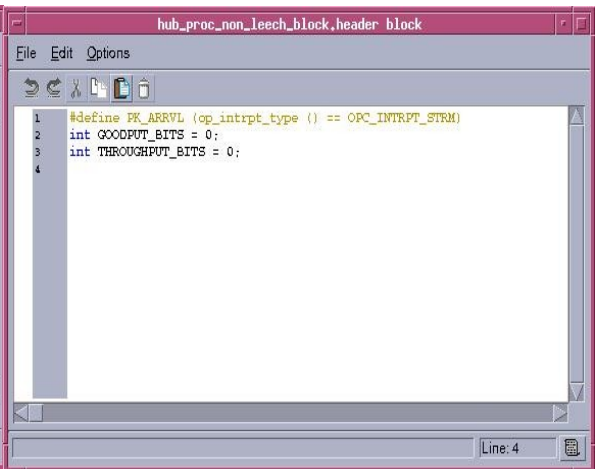


Figure 3.4.3.2 *Enter Execs of the router*

Figure 3.4.3.3 *Header block of router*

3.4.4 Custom FTP Server Process Model (Any Scenario)

Lastly, the FTP server's process model is shown below. The process model has some similarities with the seeder node but only 3 state transitions exist: the source arrival transmitter, the receive arrival receiver and the default. The model is shown below.

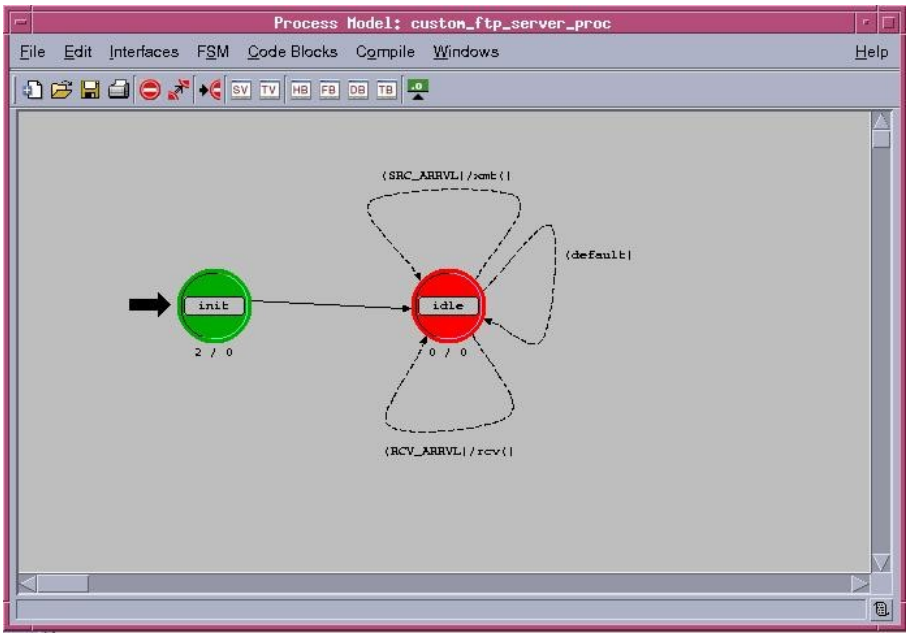


Figure 3.4.4.1 *Custom FTP server process model*

4.0 RESULTS

The three scenarios were run for 5 minutes each. The results are shown below. The figure below shows that the Leech Blocking cenario has the lowest goodput value (a global statistic collected by the central router) compared to all the scenarios. The goodput of the Leech Blocking and Baseline scenarios are in fact equal (not able to tell from graph). The Leech Routing scenario on the other hand, has the highest goodput. Therefore, our primary goal has been achieved.

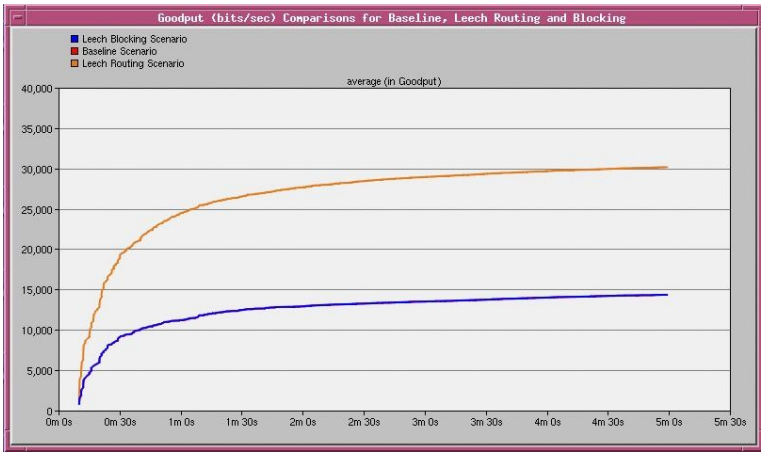


Figure 4.1 *Goodput (bits/sec) Results*

Next, figure 4.2 shows how throughput (a global statistic collected by the central router) varies from scenario to scenario. Throughput is roughly twice as large for the Leech Routing and Baseline scenarios than for the Leech Blocking scenario (Leech Routing and Baseline have same throughput but hard to tell from graph). This makes sense because many BitTorrent reply packets that are destined to the router and to the leeches get destroyed before reaching the router in the Leech Blocking scenario. This is a quick and efficient way to manage overutilization of a P2P network.

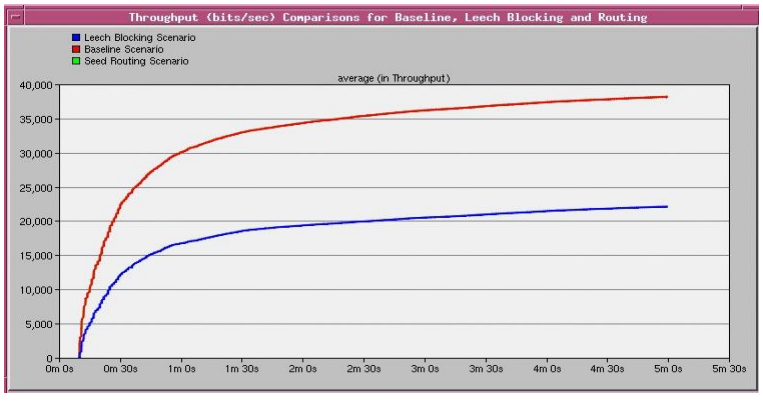


Figure 4.2 *Throughput (bits/sec) Results*

Next, Figure 4.3 shows the various ETE Delay values for the different scenarios. The smallest ETE values occur for the Leech Blocking Scenario. The next smallest ETE values correspond to the Baseline scenario, and finally the largest ETE values are found with the Leech Routing scenario. This also makes sense when

referring back to the original network topology model Figure 3.1.1. The two peer nodes are the farthest distance-wise from the central router. As a result, the packet End-To-End (ETE) Delay will be greatest for packets traveling to the peers. The Leech Routing scenario forces all BitTorrent reply packets destined for the leech nodes to travel to the peer nodes instead, thereby increase ETE considerably. The Leech Blocking scenario on the other hand, destroys a large number of packets and so decreases ETE. The baseline scenario ETE delay is then somewhere in between.

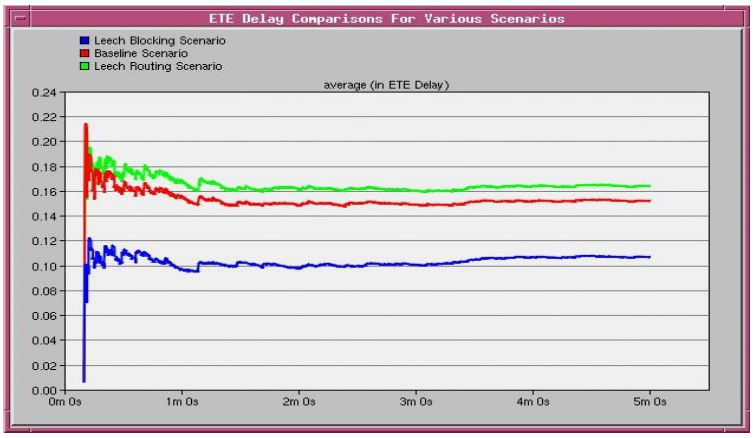


Figure 4.3 *Packet End-To-End Delay Results*

Figure 4.4 shows the individual link throughput (packets/sec) from the router to the seed node. We see that the Leech Blocking scenario decreased the throughput (because packets are destroyed at the seeds) whereas the Baseline and Leech Routing scenarios have a higher and equal throughput. Figure 4.5 shows that the peer to router throughput (bits/sec) decreases for the Leech Blocking scenario but is higher and equal for both the Baseline and Leech Routing scenarios just like the seed node. It is not shown in the figures, but the Baseline and Leech Routing scenarios produce the same results.

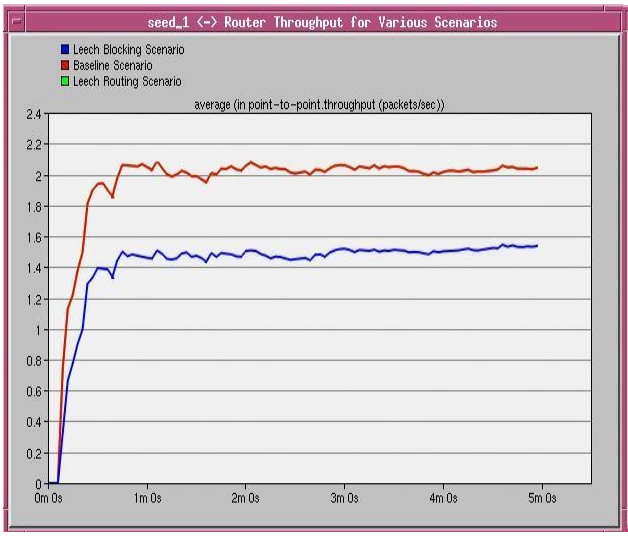


Figure 4.4 *Router to Seed Link Throughput*

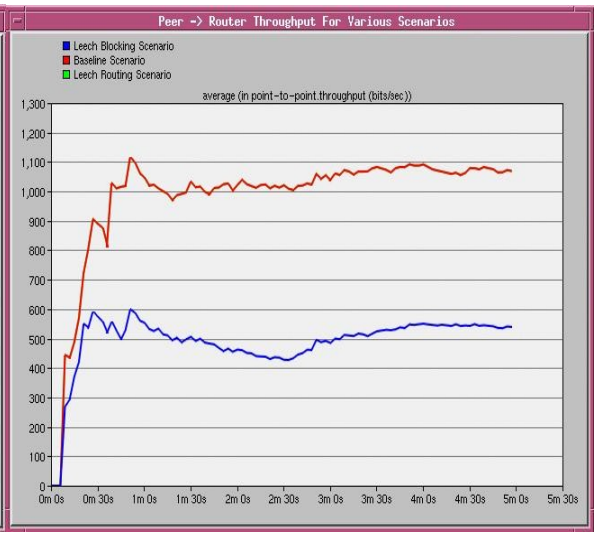


Figure 4.5 *Peer to Router Link Throughput*

Figure 4.6 shows the router to peer link throughput (bit/sec). It is not shown on the graph, but the throughput for the Leech Blocking and Baseline scenarios are equal. The Leech Blocking scenario does not change the number of BitTorrent reply packets that are sent to the peers. The throughput is greatest for the Leech Routing scenario for obvious reasons. Figure 4.7 shows the router to leech link throughput (bits/sec). Obviously, the throughput is least when the BitTorrent reply packets are destroyed or redirected to the peers before reaching the leech nodes. Therefore, throughput is lowest and essentially equal for the Leech Blocking and Leech Routing scenarios.

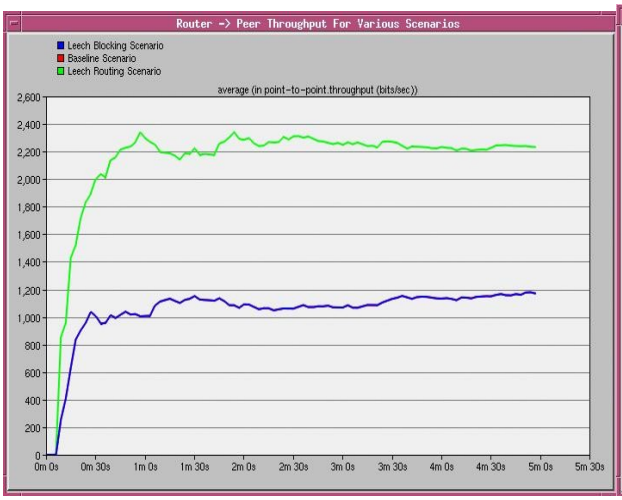


Figure 4.6 *Router to Peer Link Throughput*

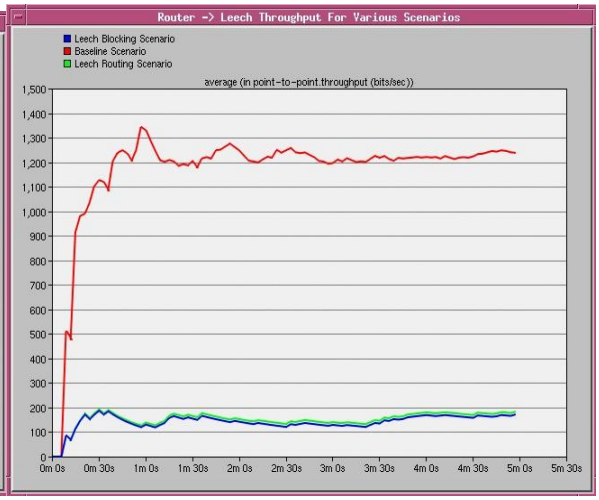


Figure 4.7 *Router to Leech Link Throughput*

Figure 4.8 shows that leech to router throughput does not change much at all for either scenario. This makes sense because it is the leech downloading that is blocked not the BitTorrent requests. Finally, figure 4.9 shows the same results as figure 4.7. Clearly, leech blocking can be used to manage channel utilization in case the network becomes over-congested.

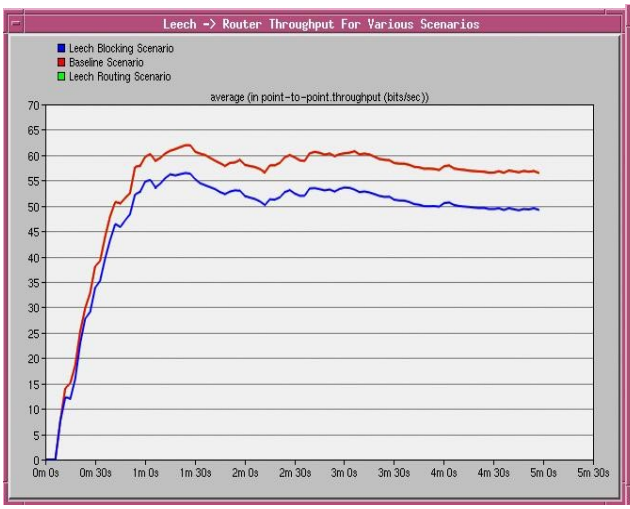


Figure 4.8 *Leech to Router Link Throughput*

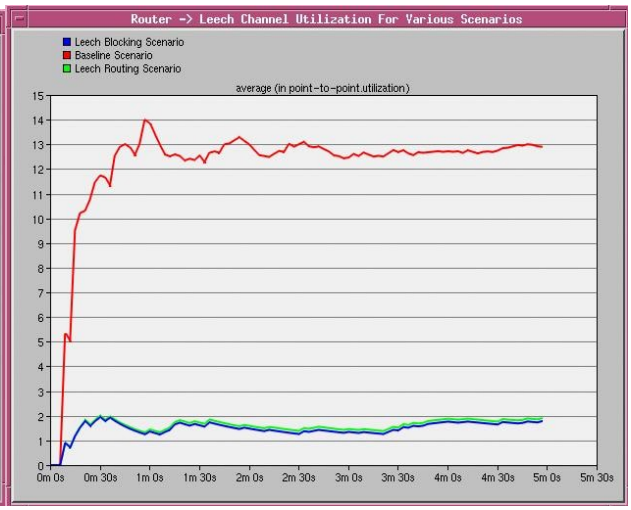


Figure 4.9 *Router to Leech Link Utilization*

The last figures show the goodput and throughput results for a Pareto(1,1) distribution. The Pareto distribution portrays the file size distribution of internet traffic using the TCP protocol more accurately. The results agree with the previous results, leech blocking decreases throughput and leech routing increases goodput.

Figure 4.10 *Goodput (bit/sec) Results*

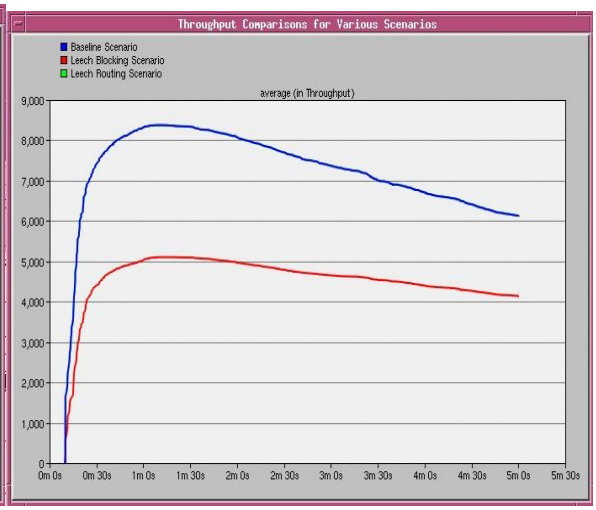


Figure 4.11 Throughput(bits/sec) Results

5.0 DISCUSSION and CONCLUSION

The goals for our project were to implement the BitTorrent peer-to-peer protocol in OPNET and implement a Leech Blocking algorithm in the seeder node to increase the goodput in the network. Based on the results the implementation of the leech blocking and routing algorithms do increase the throughput to the peer nodes **two-fold**. The major stumbling block in our project is the implementation of the Leech Blocking algorithm due to the complexity of the problem. At first, we implemented the leech blocking in the central router. However, upon simulating the model, we noticed that the overall throughput going to the router do not change because the router will still be receiving all the packets (whether or not the router will later destroy them). Instead, we had to change the model and implement the leech blocking at the seeds and peers. We were also required to create new packet formats and alter the process model of the nodes in the BitTorrent-like network. In general, setting up the network model to behave accordingly, accept multiple simple sources at the node modules, and coding the statistics collection at the router, and debugging proved fairly consuming.

We can expand and improve our BitTorrent-like network by:

1. Making the Leech Blocking algorithm more dynamic by permitting the leech to connect again after a certain time and keep track of their behaviour. If they continue leeching for a specified number of times the router would cut the channel indefinitely. Furthermore by expanding the number of nodes, we can gather more accurate results for our network thus enabling us to determine the maximum number of “leeches” allowed on a network before it starts to have a detrimental effect.

2. Improving the scalability by expanding the BitTorrent network. Or in other words make the router node able to communicate to other routers. The Leech Blocking algorithm must be modified to support the expansion.

3. Add a more realistic transmission control to our client nodes. For our project in order to simplify the OPNET implementation, the destination address for the packets being sent are created by random uniform distribution instead of the source address of the actual incoming packet request. Adding a more realistic transmission would mean adding mechanisms to extract destination address from the incoming packet request and attaching it into the outgoing packet reply.

6.0 REFERENCES

[1] E. Elghoneimy, “ENSC 835 – Spring 2006,” [Online]. Available: http://www.sfu.ca/~eelghone/ [Accessed: Feb. 10. 2009].

[2] K. Eger, “Simulation of BitTorrent Peer-to-Peer (P2P) Networks in ns-2,” [Online]. Available: http://www.tu-harburg.de/et6/research/bittorrentsim/index.html [Accessed: Feb. 10, 2009].

[3] E. Chen and C. Ng, “Comparative Analysis of Wireless Routing Algorithms in ns-2,” [Online]. Available: http://www.ensc.sfu.ca/~ljilja/ENSC835/Spring06/Projects/chen_ng/Report.pdf [Accessed: Feb 10, 2009].

[4] N. Liogkas, R. Nelson, E. Kohler, and L. Zhang, “Exploiting BitTorrent For Fun (But Not Profit),” [Online]. Available: http://iptps06.cs.ucsb.edu/papers/Liogkas-BitTorrent06.pdf [Accessed: Feb 11, 2009].

[5] L. Xu, K. Harfoush, and I. Rheel, “Binary Increase Congestion Control for Fast, Long Distance Networks,” [Online]. Available: http://www4.ncsu.edu/~rhee/export/bitcp.pdf [Accesed: Feb 11, 2009].

[6] Cohen, Bram “The BitTorrent Protocol Specification,” [Online]. Available: http://www.bittorrent.org/beps/bep_0003.html [Accessed: Mar 30, 2009].

[7] Image, “Client-server,” [Online]. Available: http://www.geocities.com/doyetech/clientsev.jpg

[8] Image, “peer-to-peer,” [Online]. Available: http://www.geocities.com/doyetech/peer2.jpg

[9] Image, “bit-torrent,” [Online]. Available: http://www.pcmech.com/wp-content/uploads/2007/08/bittorrent.gif

APPENDIX

```
//#########################################################

// Seed/Peer/Leech Node Process Model State Variables

//#########################################################

Distribution *      address_bt_req

Stathandle          ete_gsh

Distribution*       address_ftp_req

Distribution *      address_bt_reply


//#########################################################

// Seed/Peer/Leech Node Process Model Header Block

//#########################################################

/* packet stream definition */

#define RCV_IN_STRM 0

#define SRC_IN_STRM 1

#define XMT_OUT_STRM 0

#define SRC_IN_STRM_2 2

#define SRC_IN_STRM_3 3


/* transition macros */

#define SRC_ARRVL (op_intrpt_type () == \

        OPC_INTRPT_STRM && op_intrpt_strm () == SRC_IN_STRM)

#define SRC_ARRVL_2 (op_intrpt_type () == \

        OPC_INTRPT_STRM && op_intrpt_strm () == SRC_IN_STRM_2)

#define SRC_ARRVL_3 (op_intrpt_type () == \
```

```
	OPC_INTRPT_STRM && op_intrpt_strm () == SRC_IN_STRM_3)
#define RCV_ARRVL (op_intrpt_type () == \
	OPC_INTRPT_STRM && op_intrpt_strm () == RCV_IN_STRM)

//############################################################
// Seed/Peer/Leech  Node Process Model init process: Enter Execs
//############################################################
address_bt_req = op_dist_load ("uniform_int", 0, 5);
address_ftp_req = op_dist_load("uniform_int",6,6);
address_bt_reply = op_dist_load("uniform_int",2,5);
ete_gsh = op_stat_reg ("ETE Delay", OPC_STAT_INDEX_NONE, OPC_STAT_GLOBAL);

//############################################################
// Seed/Peer/Leech  Node Process Model init process: Function block (baseline scenario)
//############################################################
static void xmt(void)
	{
	Packet * pkptr;
	FIN(xmt());
	pkptr = op_pk_get (SRC_IN_STRM);
	int protocol;
	op_pk_nfd_set_int32(pkptr, "src_address", op_id_self());
	op_pk_nfd_get_int32(pkptr,"protocol",&protocol);

	if (protocol==0){
```

```
    printf("sending protocol 0");
        op_pk_nfd_set_int32(pkptr,"dest_address",(int)op_dist_outcome(address_ftp_req));
        }
    else if (protocol==1){
        printf("sending protocol 1");
        op_pk_nfd_set_int32(pkptr,"dest_address",(int)op_dist_outcome(address_bt_req));
        }
    else if (protocol==2){
        printf("sending protocol 2");
        op_pk_nfd_set_int32(pkptr,"dest_address",(int)op_dist_outcome(address_bt_reply));
        }
    op_pk_send (pkptr, XMT_OUT_STRM);
    FOUT;
    }


static void xmt2(void)
    {
    Packet * pkptr;
    FIN(xmt2());
    pkptr = op_pk_get (SRC_IN_STRM_2);
    int protocol;
    op_pk_nfd_set_int32(pkptr, "src_address", op_id_self());
    op_pk_nfd_get_int32(pkptr,"protocol",&protocol);
    if (protocol==0){
    printf("sending protocol 0");
```

```
		op_pk_nfd_set_int32(pkptr,"dest_address",(int)op_dist_outcome(address_ftp_req));
		}
	else if (protocol==1){
		printf("sending protocol 1");
		op_pk_nfd_set_int32(pkptr,"dest_address",(int)op_dist_outcome(address_bt_req));
		}
	 else if (protocol==2){
		printf("sending protocol 2");
		op_pk_nfd_set_int32(pkptr,"dest_address",(int)op_dist_outcome(address_bt_reply));
		}
	op_pk_send (pkptr, XMT_OUT_STRM);
	FOUT;
	}

static void xmt3(void)
	{
	Packet * pkptr;
	FIN(xmt3());
	pkptr = op_pk_get (SRC_IN_STRM_3);
	int protocol;

	op_pk_nfd_set_int32(pkptr, "src_address", op_id_self());
	op_pk_nfd_get_int32(pkptr,"protocol",&protocol);

	if (protocol==0){
```

```
    printf("sending protocol 0");
            op_pk_nfd_set_int32(pkptr,"dest_address",(int)op_dist_outcome(address_ftp_req));
            }
        else if (protocol==1){
            printf("sending protocol 1");
            op_pk_nfd_set_int32(pkptr,"dest_address",(int)op_dist_outcome(address_bt_req));
            }
       else if (protocol==2){
            printf("sending protocol 2");
            op_pk_nfd_set_int32(pkptr,"dest_address",(int)op_dist_outcome(address_bt_reply));
            }
        op_pk_send (pkptr, XMT_OUT_STRM);
        FOUT;
        }


static void rcv(void)
        {
        Packet * pkptr;
        double ete_delay;
        FIN(rcv());
        pkptr = op_pk_get (RCV_IN_STRM);
        ete_delay = op_sim_time () -
                        op_pk_creation_time_get (pkptr);
        op_stat_write (ete_gsh, ete_delay);
        op_pk_destroy (pkptr);
```

```
	FOUT;
	}
//#########################################################
// Seed/Peer/Leech  Node Process Model init process: Function block (Leech blocking scenario)
//#########################################################
static void xmt(void)
	{
	Packet * pkptr;
	FIN(xmt());
	pkptr = op_pk_get (SRC_IN_STRM);
	int protocol;
	int dest_addr;

	op_pk_nfd_set_int32(pkptr, "src_address", op_id_self());
	op_pk_nfd_get_int32(pkptr,"protocol",&protocol);
	if (protocol==0){
		printf("sending protocol 0");
		op_pk_nfd_set_int32(pkptr,"dest_address",(int)op_dist_outcome(address_ftp_req));
		}
	else if (protocol==1){
		printf("sending protocol 1");
		op_pk_nfd_set_int32(pkptr,"dest_address",(int)op_dist_outcome(address_bt_req));
		}
	  else if (protocol==2){
		op_pk_nfd_set_int32(pkptr,"dest_address",(int)op_dist_outcome(address_bt_reply));
```

```
		}
	op_pk_nfd_get_int32(pkptr,"dest_address",&dest_addr);
	if (dest_addr==4)   // packe destined to leech location, must be bt_reply
		{
		op_pk_destroy (pkptr);
		}
	else if (dest_addr ==5)
		{
		op_pk_destroy (pkptr);
		}
	else
		{
		op_pk_send (pkptr, XMT_OUT_STRM);
		}
FOUT;
}


static void xmt2(void)
	{
	Packet * pkptr;
	FIN(xmt2());
	pkptr = op_pk_get (SRC_IN_STRM_2);
	int protocol;
	int dest_addr;
```

```
op_pk_nfd_set_int32(pkptr, "src_address", op_id_self());
op_pk_nfd_get_int32(pkptr,"protocol",&protocol);
if (protocol==0){
        op_pk_nfd_set_int32(pkptr,"dest_address",(int)op_dist_outcome(address_ftp_req));
        }
else if (protocol==1){
        op_pk_nfd_set_int32(pkptr,"dest_address",(int)op_dist_outcome(address_bt_req));
        }
 else if (protocol==2){
        op_pk_nfd_set_int32(pkptr,"dest_address",(int)op_dist_outcome(address_bt_reply));
        }
op_pk_nfd_get_int32(pkptr,"dest_address",&dest_addr);
if (dest_addr==4)   // packe destined to leech location, must be bt_reply
        {
        op_pk_destroy (pkptr);
        }
else if (dest_addr ==5)
        {
        op_pk_destroy (pkptr);
        }
else
        {
        op_pk_send (pkptr, XMT_OUT_STRM);
        }
FOUT;
```

```
}


static void xmt3(void)
        {
        Packet * pkptr;
        FIN(xmt3());
        pkptr = op_pk_get (SRC_IN_STRM_3);
        int protocol;
        int dest_addr;


        op_pk_nfd_set_int32(pkptr, "src_address", op_id_self());
        op_pk_nfd_get_int32(pkptr,"protocol",&protocol);
        if (protocol==0){
                op_pk_nfd_set_int32(pkptr,"dest_address",(int)op_dist_outcome(address_ftp_req));
                }
        else if (protocol==1){
                op_pk_nfd_set_int32(pkptr,"dest_address",(int)op_dist_outcome(address_bt_req));
                }
   else if (protocol==2){
                op_pk_nfd_set_int32(pkptr,"dest_address",(int)op_dist_outcome(address_bt_reply));
        }
        op_pk_nfd_get_int32(pkptr,"dest_address",&dest_addr);
        if (dest_addr==4)   // packe destined to leech location, must be bt_reply
                {
                op_pk_destroy (pkptr);
```

```
		}
	else if (dest_addr ==5)
		{
		op_pk_destroy (pkptr);
		}
	else
		{
		op_pk_send (pkptr, XMT_OUT_STRM);
		}
FOUT;
}
static void rcv(void)
	{
	Packet * pkptr;
	double ete_delay;
	FIN(rcv());
	pkptr = op_pk_get (RCV_IN_STRM);
	ete_delay = op_sim_time () -
			op_pk_creation_time_get (pkptr);
	op_stat_write (ete_gsh, ete_delay);
	op_pk_destroy (pkptr);
	FOUT;
	}
```

```
//#########################################################
// Router Node Process Model State Variables
//#########################################################
Stathandle      ete_gsh
Stathandle      goodput_gsh
Stathandle      throughput_gsh

//#########################################################
// Router Node Process Model Header Block
//#########################################################
#define PK_ARRVL (op_intrpt_type () == OPC_INTRPT_STRM)
int GOODPUT_BITS = 0;
int THROUGHPUT_BITS = 0;

//#########################################################
// Router Node Process Model init process: Enter Execs
//#########################################################
ete_gsh = op_stat_reg ("ETE Delay", OPC_STAT_INDEX_NONE, OPC_STAT_GLOBAL);
goodput_gsh = op_stat_reg("Goodput", OPC_STAT_INDEX_NONE, OPC_STAT_GLOBAL);
throughput_gsh = op_stat_reg("Throughput", OPC_STAT_INDEX_NONE, OPC_STAT_GLOBAL);

//#########################################################
// Router Node Process Model Function Block
//#########################################################
static void route_pk(void)
```

```
{
int dest_address;
int node_id;
int protocol;
double GOODPUT;
double THROUGHPUT;
double ete_delay;
Packet* pktptr;

FIN(route_pk());
pktptr = op_pk_get(op_intrpt_strm ());
op_pk_nfd_get_int32 (pktptr, "dest_address", &dest_address);
op_pk_nfd_get_int32 (pktptr, "protocol", &protocol);
op_pk_nfd_get_int32 (pktptr, "src_address", &node_id);

if (protocol ==2)  //if bt reply packet
        {
        if (dest_address ==4)  //leech node
                {
                        THROUGHPUT_BITS = THROUGHPUT_BITS + 1024*8;
                   //increment by size of bt reply packet
                        THROUGHPUT = THROUGHPUT_BITS/op_sim_time ();
                        op_stat_write (throughput_gsh, THROUGHPUT);

                }
        else if (dest_address ==5)  //leech node
```

```
                {
                        THROUGHPUT_BITS = THROUGHPUT_BITS + 1024*8;
                   //increment by size of bt reply packet
                        THROUGHPUT = THROUGHPUT_BITS/op_sim_time ();
                        op_stat_write (throughput_gsh, THROUGHPUT);
                }

        else  //bt reply packet destined for non leech node
                {
                        THROUGHPUT_BITS = THROUGHPUT_BITS + 1024*8;
                   //increment by size of bt reply packet
                        THROUGHPUT = THROUGHPUT_BITS/op_sim_time ();
                        op_stat_write (throughput_gsh, THROUGHPUT);
                      //increment by size of bt reply packet
                       GOODPUT_BITS = GOODPUT_BITS + 1024*8;
                        GOODPUT = GOODPUT_BITS/op_sim_time ();
                        op_stat_write (goodput_gsh, GOODPUT);
                }
        op_pk_send (pktptr, dest_address);
        }
else // not a bt reply packet
        {
        op_pk_send (pktptr, dest_address);

        //increment throughput(bits/sec) in router based on type of packets passing through
```

```
		//bt request and ftp request are 24 bytes. ftp reply packet is 1024 bytes long.
		if (protocol == 3) //ftp reply packet, increment by 1024 bytes
			{
				//increment by size of FTP reply packet
				THROUGHPUT_BITS = THROUGHPUT_BITS + 1024*8;
				THROUGHPUT = THROUGHPUT_BITS/op_sim_time ();
				op_stat_write (throughput_gsh, THROUGHPUT);

			}
		else  //it's a bt or ftp request. increment throughput by 24 bytes
			{
				  //increment by size of bt reply packet
				THROUGHPUT_BITS = THROUGHPUT_BITS + 24*8;
				THROUGHPUT = THROUGHPUT_BITS/op_sim_time ();
				op_stat_write (throughput_gsh, THROUGHPUT);
			}
		}
FOUT;
}

//############################################################
// FTP server Node Process Model State Variables
//############################################################
Distribution*	address_bt_req
Stathandle	 ete_gsh
Distribution*	address_ftp_reply
```

```
Distribution*	address_bt_reply

//#########################################################
// FTP server Node Process Model Header Block
//#########################################################
/* packet stream definition */
#define RCV_IN_STRM 0
#define SRC_IN_STRM 1
#define XMT_OUT_STRM 0

/* transition macros */
#define SRC_ARRVL (op_intrpt_type () == \
	OPC_INTRPT_STRM && op_intrpt_strm () == SRC_IN_STRM)
#define RCV_ARRVL (op_intrpt_type () == \
	OPC_INTRPT_STRM && op_intrpt_strm () == RCV_IN_STRM)

//#########################################################
// FTP server Node Process init process: Enter Execs
//#########################################################
address_ftp_reply = op_dist_load("uniform_int",0,5);
ete_gsh = op_stat_reg ("ETE Delay", OPC_STAT_INDEX_NONE, OPC_STAT_GLOBAL);

//#########################################################
// FTP server Node Process Function Block
//#########################################################
```

```
static void xmt(void)
        {
        Packet * pkptr;
        FIN(xmt());
        pkptr = op_pk_get (SRC_IN_STRM);
        op_pk_nfd_set_int32(pkptr,"dest_address",(int)op_dist_outcome(address_ftp_reply));
        op_pk_send (pkptr, XMT_OUT_STRM);
        FOUT;
        }


static void rcv(void)
        {
        Packet * pkptr;
        double ete_delay;
        FIN(rcv());
        pkptr = op_pk_get (RCV_IN_STRM);
        ete_delay = op_sim_time () -
                        op_pk_creation_time_get (pkptr);
        op_stat_write (ete_gsh, ete_delay);
        op_pk_destroy (pkptr);
        FOUT;
        }
```