Throughput vs. Latency - Similar to above, Figure 8 shows that regardless of the latency of the central link, filler traffic has no effect on the throughput of the pre-existing traffic.

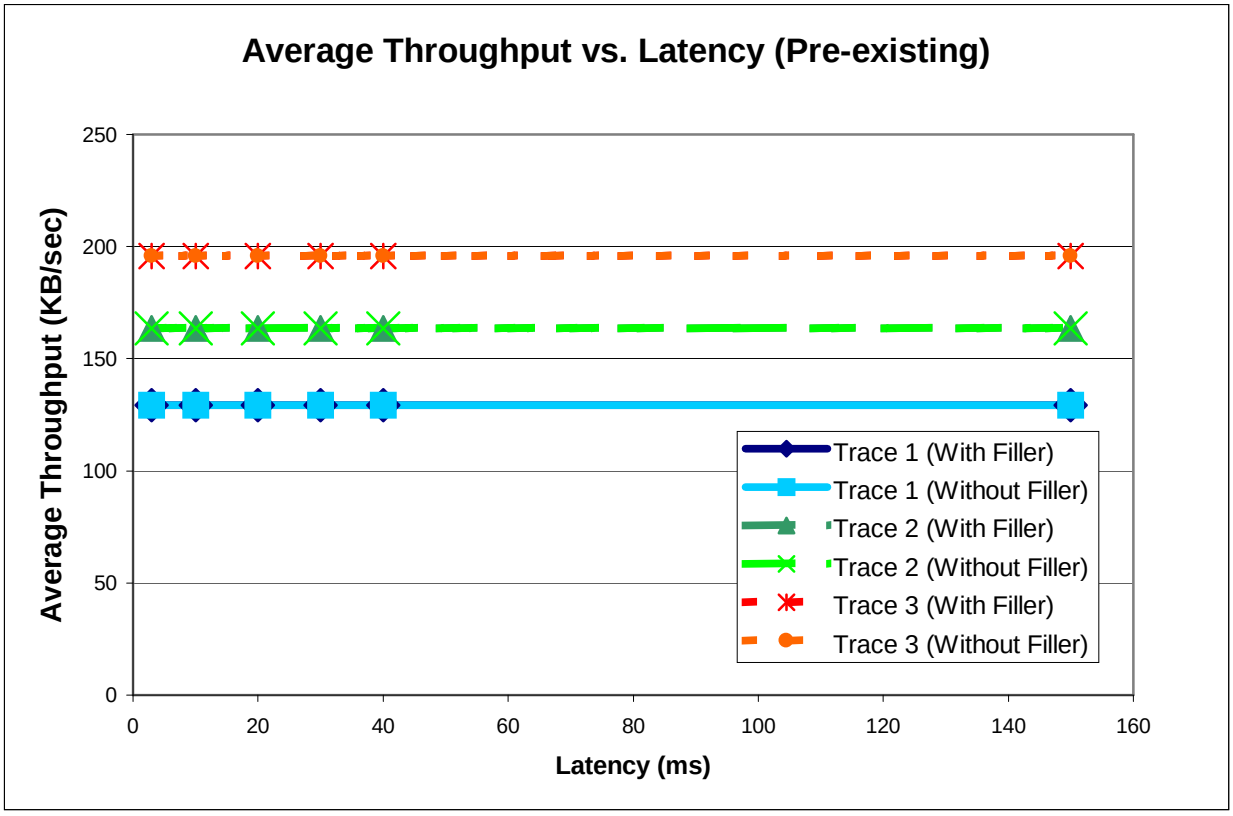

Figure 8. Average Throughput vs. Latency (Pre-existing)

Dropped Packets vs. Latency - None of the experiments involving latency resulted in any dropped packets (of pre-existing data). This is because the pre-existing buffer was set high enough to account for the small delay the caused by the filler buffer without having to lose packets. Specifically, the pre-existing buffer was set to 100 packets. The average packet size for pre-existing traffic is about 269 bytes, making the buffer likely to be around 26.25KB.

Figures 4-8 show specific examples of how pre-existing traffic reacts to the presence of filler traffic. Reactions range from nothing at all, to a deterioration of behavior by several

percent. For example, the throughput of the pre-existing traffic is totally unaffected by the existence of filler traffic. On the other hand, both average delay and percent of dropped packets were affected, but mostly on a networks which are very close to starvation. The latency of the central link had little to do with the unobtrusiveness of filler traffic, as the average delay increase was constant regardless of latency and there were no dropped packets whatsoever. However, bandwidth played a larger role in determining filler traffic unobtrusiveness - varying bandwidth changed the amount that the filler traffic affected the average delay and percent of dropped packets of the pre-existing traffic. When the bandwidth and latency were low (producing a low BDP network), the effects of filler traffic were relatively high. Otherwise, the effects were unremarkable. Thus, filler traffic generally remains unobtrusive.

Filler Traffic Performance

The last section determined that filler traffic could be used without upsetting pre-existing traffic. Now it must be shown that filler traffic is not only unobtrusive, but can successfully accomplish data transfers. To study this, the statistics of the packet dynamics of the filler traffic must be considered. While the throughput of the filler traffic is very important in determining its usefulness, it is not the only factor to be examined. Also, the percent of dropped packets shows how much work must be wasted in resending lost packets. Finally, the average delay is important because the length of the delay affects what can be used as filler traffic – the higher the delay, the more time-insensitive the filler traffic must be. Thus, this section details how the performance of filler traffic is related to the link parameters bandwidth and latency. The figures are modeled identically to those above, in colors and patterns, but these figures only concern themselves with the statistics of the filler traffic.

Average Delay vs. Bandwidth – Filler traffic behaves very similarly to varying bandwidth as pre-existing traffic. As the bandwidth increases, packets are sent faster. Therefore, packets do not have to wait as long in the queue before being sent. Figure 9 shows that as bandwidth increases, the average delay decreases, due to the extra available bandwidth for filler traffic. However, once 6 Mbps is reached for the bandwidth, the average delay ceases decreasing. This is because there is enough bandwidth to handle both filler and pre-existing traffic quickly, and so the delay due to the latency of the central link has become a large component of the total average delay. Thus more bandwidth may provide more throughput (see below), but it does not decrease the delay of the filler traffic. Even at the worst, the delay on the filler traffic is approximately 3 times the delay of the pre-existing traffic (with no filler present) at the same link bandwidth.

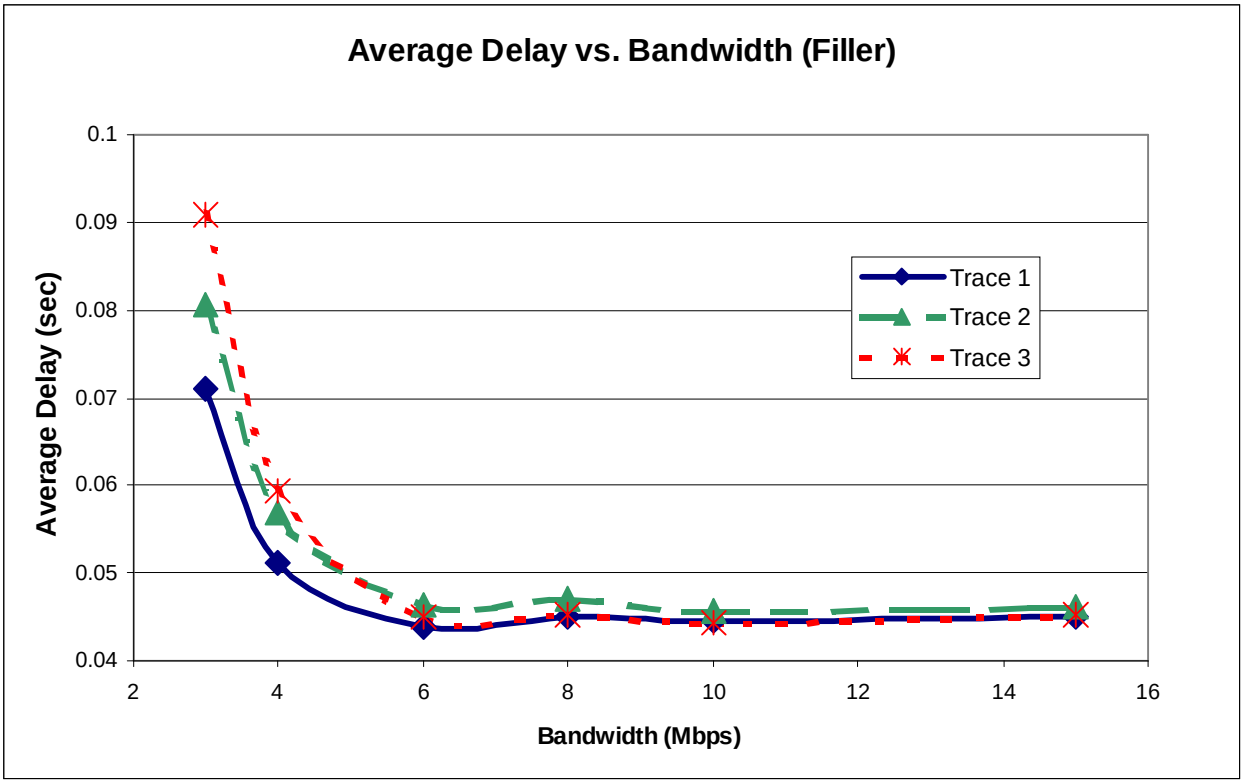

Figure 9. Average Delay vs. Bandwidth (Filler)

Dropped Packets vs. Bandwidth – The number of dropped packets decrease steadily as the bandwidth increases. This corresponds to the delay of the packets decreasing. With a large delay, the filler buffer becomes full – this is when packets are forced to be dropped. As the delay decreases (as explained above), packets move through the buffer faster, and therefore less are dropped. It should be noted in Figure 10 that the as the bandwidth approaches and exceeds 10 Mbps, the rate of decrease in the percent of dropped packets lessens. The original Harvard trace used a 10 Mbps central link, therefore at no point does the pre-existing traffic require more than 10 Mbps. Thus, as the bandwidth nears this point, there are fewer and fewer packets dropped due to bursts of pre-existing traffic, and a certain percent is caused by packets dropped by the congestion window overshooting. Regardless, the percent of dropped packets is very low – never surpassing the percent of dropped pre-existing packets by more than 0.2.

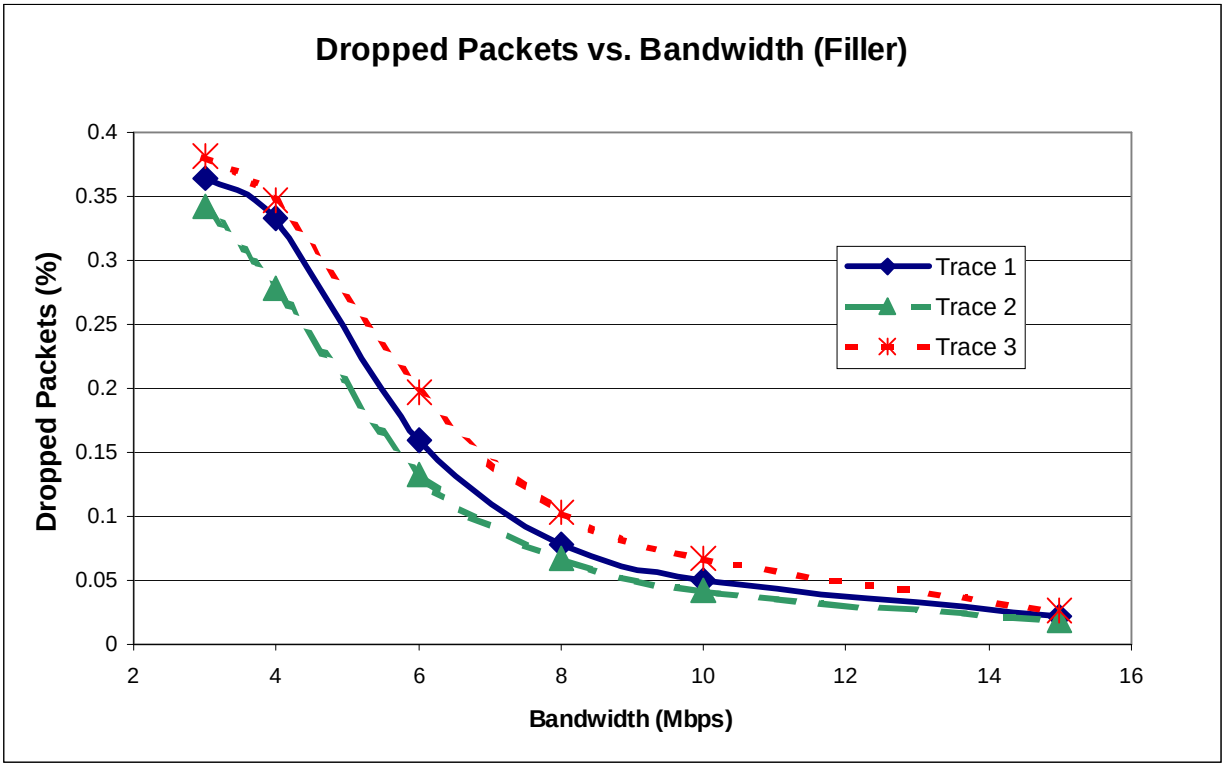

Figure 10. Dropped Packets vs. Bandwidth (Filler)

Throughput vs. Bandwidth – Throughput is the average number of bytes that cross the central link in a second. Similarly, bandwidth is the number of bytes that can cross the link per second. The throughput of the filler traffic should ideally be a percentage of the bandwidth of the link minus the throughput of the pre-existing traffic. It is only a percentage because TCP works by gradually increasing the sending rate until a packet is dropped. At this point, TCP cuts the sending speed and begins again. The throughput was simply a constant multiple (seen in experiments to be around 82%) of the difference between the link bandwidth and the throughput of the pre-existing traffic, which is a constant. As bandwidth increases, throughput increases, at a linear rate. As shown by the trendlines, the slope of the lines are approximately 125. When KB/sec is converted to Mbps, the slope becomes about 1. This indicates that as bandwidth increases, almost all of it is put towards an increase in the throughput of the filler traffic.

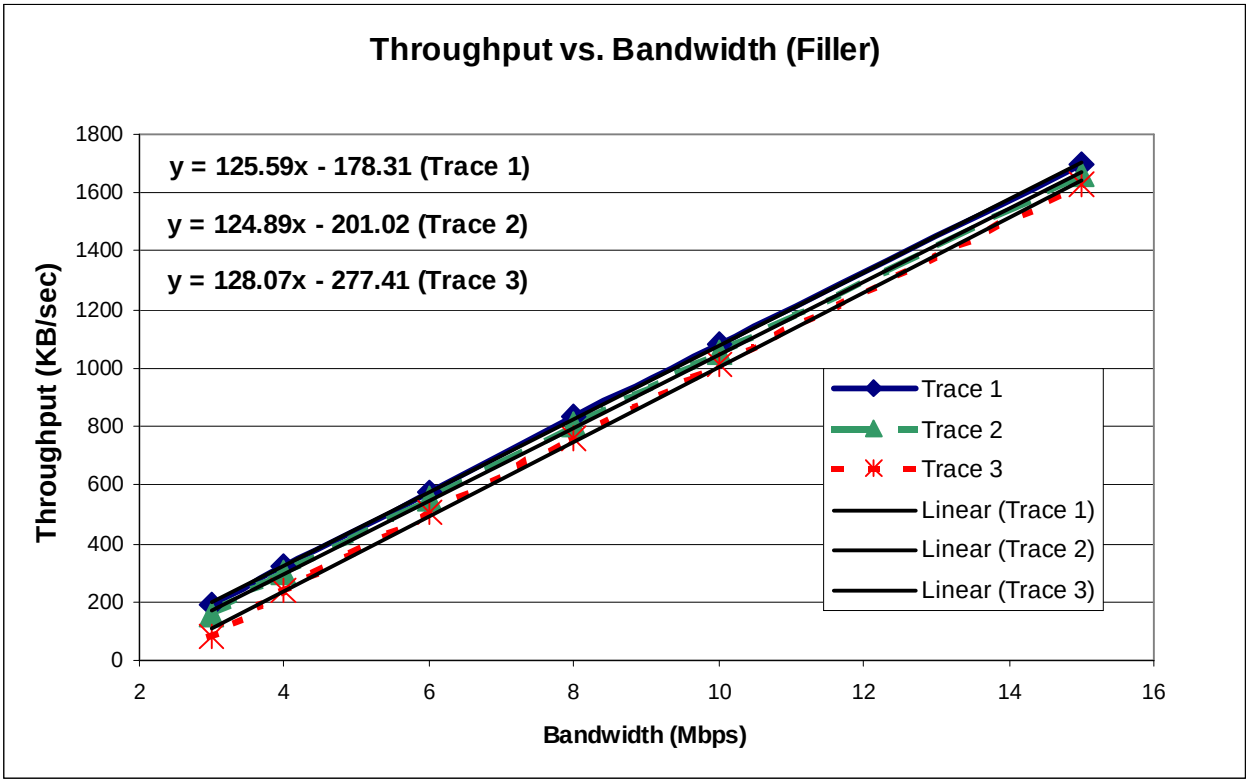

Figure 11. Throughput vs. Bandwidth (Filler)

Average Delay vs. Latency – The delay is a measure of the time a packet waits in the buffer before being sent, plus the amount of time it takes to send the packet across the central link. Since the filler buffer size is linear in terms of the link latency, average delay increases as the sum of two linear terms. Thus the slope of the line is 1 (as shown in Figure 12). The smallest two data points, as seen in Figure 12 are not at slope 1. This is due to using 16 KB as the minimum filler buffer. Because all filler buffers which would be smaller than 16 KB (i.e. at a latency of 3 ms) are raised to 16 KB, both 3 ms and 10 ms both used the same filler buffer, resulting in the increase in delay to be due solely to the increase in latency, which is still linear, but with a lesser slope.

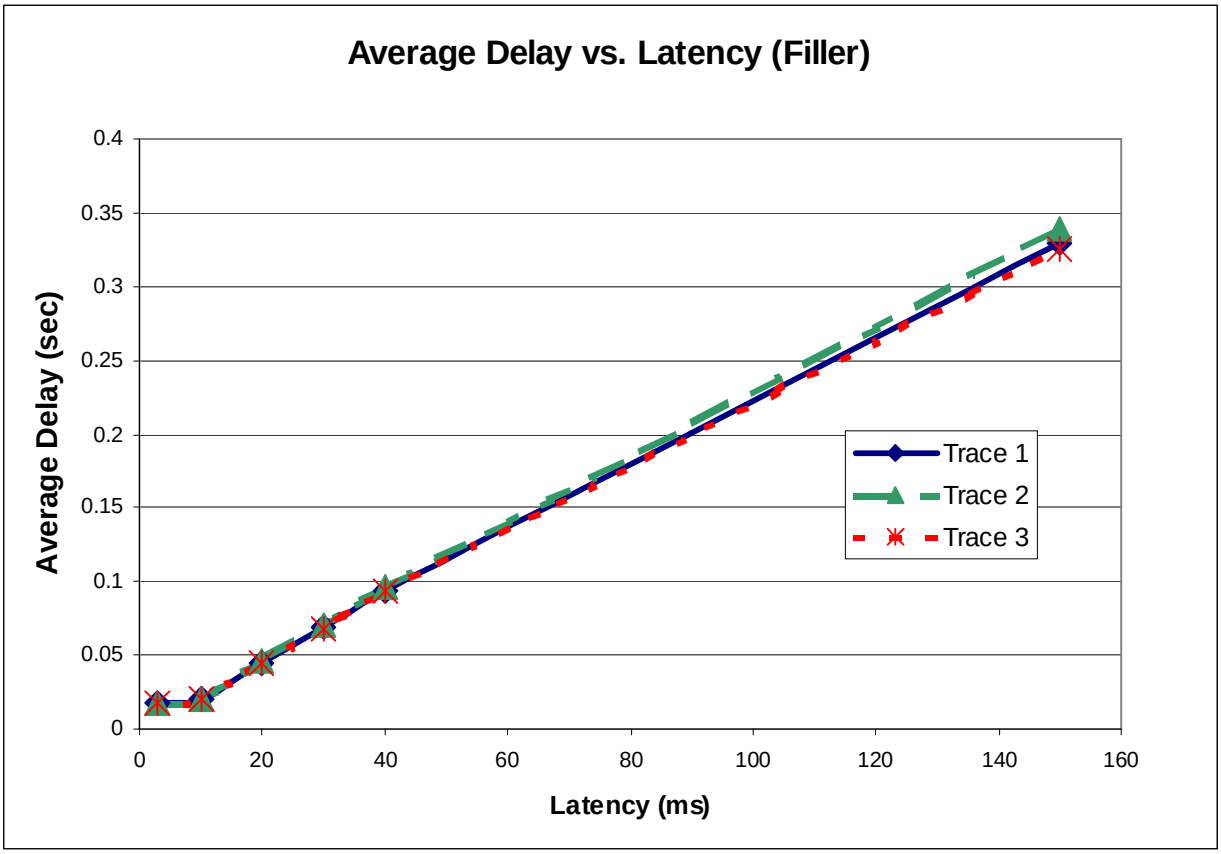

Figure 12. Average Delay vs. Latency (Filler)

Dropped Packets vs. Latency – Figure 13 shows that there is a rapid decrease in the percent of dropped packets as latency increases. This is because of the granularity of the in congestion window size. For example, when the latency is 3ms, the BDP is approximately 3.5 KB. With the window so small, TCP congestion control is overshooting very often, each time causing dropped packets. However, once the latency rises higher, the congestion window is much bigger. This results in less packets because there is more time in between each time the congestion window is overshot. Thus, the dropped packets are dropped due to normal TCP functionality, and are not caused by the reduced priority of the filler traffic.

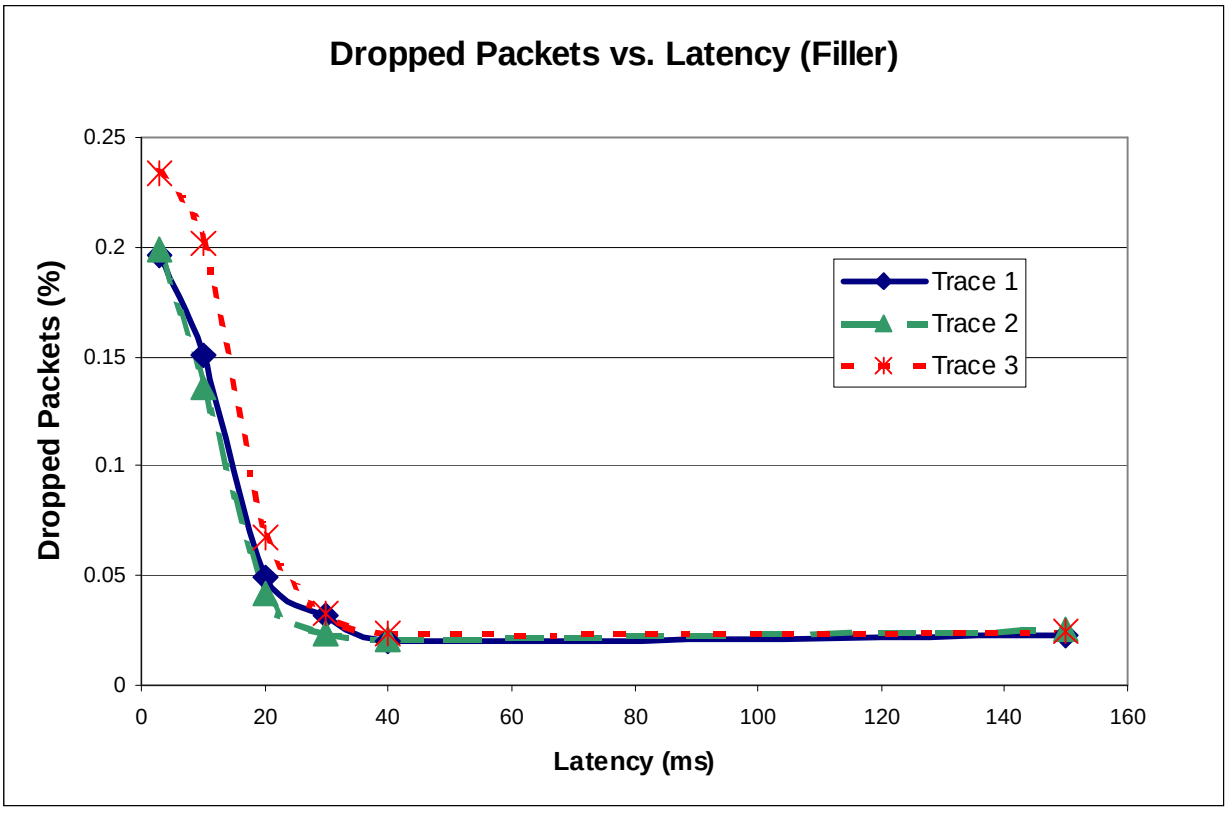

Figure 13. Dropped Packets vs. Latency (Filler)

Throughput vs. Latency – Throughput is relatively unaffected by latency, as seen in Figure 14. There is only less than a 4%, 5%, and 6.5% difference, respectively, between the highest value and lowest value for each trace. Latency has a fairly low affect on throughput because in our experiments an increased latency does not change how much data can flow across the line, only how long it takes. If the filler buffer remained the same from one experiment to another, as the latency increased, it would eventually become high enough to cause the buffer to fill up and packets to be dropped. However, since the filler buffer is calculated based on latency, it increases at the same rate as latency does, and can therefore provide enough space to keep an ever increasing number of packets from being dropped. In the base case, with trace 2 (this will be examined more later), the throughput is about 1050 KB/sec, or 8.6 Mbps.

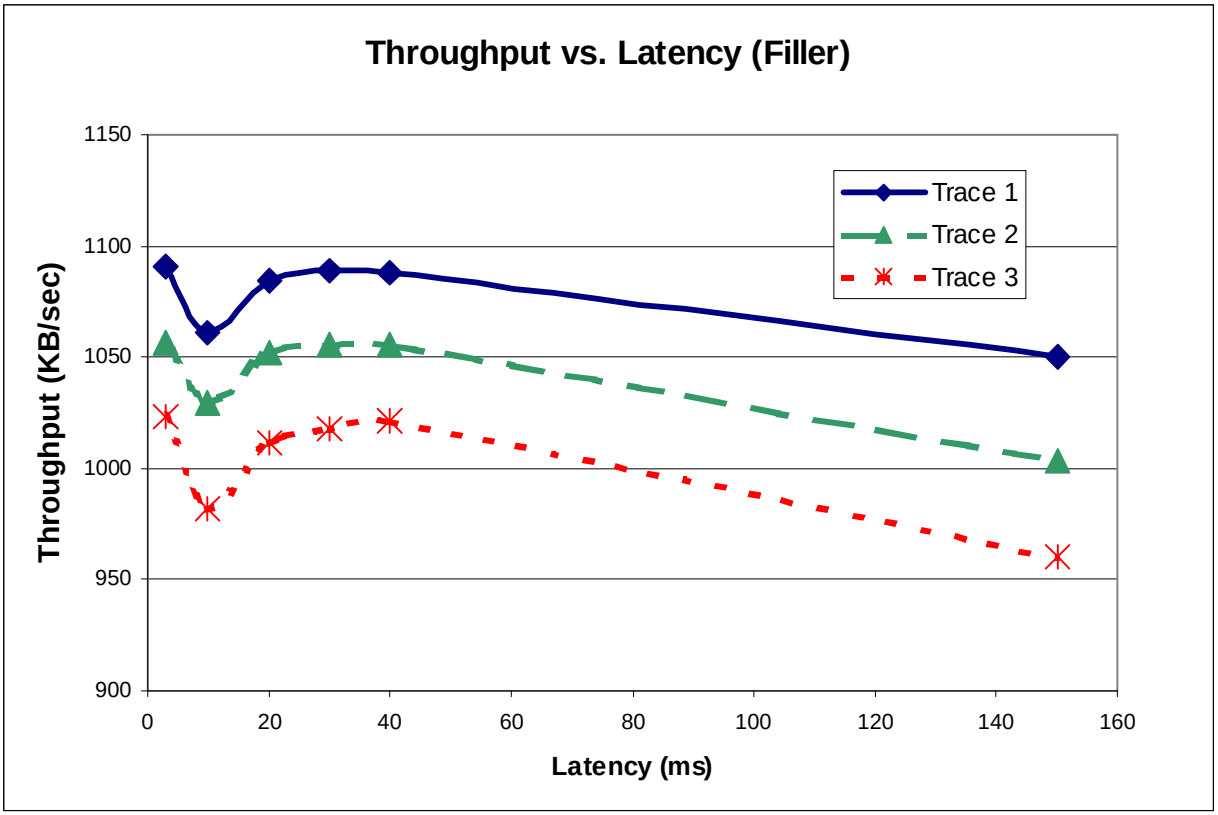

Figure 14. Throughput vs. Latency (Filler)

To act as a basis of comparison, an experiment was conducted involving only filler traffic. First, the throughput of the pre-existing traffic of trace 2 was found (about 1.34 Mbps). This was subtracted from the bandwidth of the central link in the base case, leaving 8.66 Mbps. 8.66 Mbps was then used as the link bandwidth in the experiment without pre-existing traffic. The result was about 8.32 Mbps of throughput for the filler traffic, or about 96% utilization. This is compared to the throughput of the filler with the pre-existing traffic present. In Figure 14 it is shown that the filler traffic had a throughput of 8.60 Mbps. After adding in the pre-existing traffic, the combined throughput is 1.31 Mbps. The total percent of utilization is Without the bursts of the pre-existing traffic, it can be seen that filler traffic utilizes most of the available bandwidth. Therefore, most of the unutilized bandwidth in the rest other experiments is due to the bursts of the pre-existing traffic. It will be interesting to compare some of the charts above (with filler an pre-existing traffics) with some new charts created from experiments with only filler traffic, after subtracting out the bandwidth that would have been needed by the pre-existing traffic.

Figures 9 through 14 show that filler traffic does perform useful data transfer. In many cases the filler traffic performed nearly as well as the pre-existing traffic. With the bandwidth set high enough that the network is not on the verge of starvation, the filler traffic's performance is excellent. Specifically, the average delay of filler packets is very low. Further, once the latency is around that of the base case (20ms) or higher, the percent of dropped packets diminishes while the throughput increases greatly. Even in the experiments with a relatively low latency or bandwidth, the filler traffic managed to provide useful data transmission, while, as shown above, remaining unobtrusive to the pre-existing traffic. Thus filler traffic can be very useful.

Conclusion

To assess the ability to use filler traffic on a network, two things must be considered. First, by nature, filler traffic should not affect the pre-existing traffic of a network. In light of this, its unobtrusiveness, or how much it does affect the pre-existing traffic, should be studied. Additionally, filler traffic must perform useful transmission of data, otherwise it serves no purpose. Thus, its performance must be studied. These two factors, filler unobtrusiveness and performance, were studied via a variety of experiments. Each experiment either tested filler's impact on pre-existing traffic or the usefulness of the filler traffic itself.

The filler traffic was found to have little affect on pre-existing traffic. The pre-existing traffic appears to behave very similarly in the presence of filler traffic as it did without. Average delay only increased slightly, while the percent of dropped packets and throughput remained virtually unchanged. Thus the filler traffic behaves fairly unobtrusively.

Additionally, the filler traffic managed to accomplish useful data transmission. Between pre-existing and filler traffic together, over 95% of the bandwidth was utilized.. Average delays and percent of dropped packets remained low, often no higher than the values of the pre-existing traffic. Even when they were higher, neither delays nor the number of packets dropped increased to an unusable level.

All of the experiments conducted involved networks with medium to high BDP. Therefore, it has been determined that filler traffic is a feasible option on these networks. However, as BDP diminishes, so does filler traffic's value. Because of this, future work is being done to study filler traffic on a network which has a low BDP, such as a bank of modems (see Appendix).

Appendix

A. Modem Traces

The experiments we conducted show that varying bandwidth and latency did not cause filler traffic to greatly affect pre-existing traffic. However, all of these experiments were conducted on relatively high BDP networks. Ongoing work is being carried out to determine the unobtrusiveness and performance of filler traffic on low BDP networks. Specifically, a set of experiments is being set up to test a modem bank connected to the network.

The data was gathered by recording incoming and outgoing packet information of a bank of modems at an Internet Service Provider. Packet information was gathered using WinDump [11] starting on Wednesday, January 12th, 2001 at 1:37pm. The traces contain one hour's worth of packet information. By configuring WinDump using command line parameters, only traffic involving a modem on the sending or receiving end was included in the output file. All modems are 56k modems, which send data at a maximum of 33.6 Kbps and receive data at 53.3 Kbps. There are 74 modems connected, as a group, to the Internet through a T1 (1.54 Mbps) link. Once the trace file is parsed and turned in to an ns-ready file, it will contain a filler and pre-existing source/destination for each modem, and one more on the other side of the central T1 link representing the Internet.

Even though pre-existing traffic is at a higher priority than filler traffic, if a filler packet is currently being sent when a pre-existing packet is queued, it will not be sent until the filler packet is finished sending. Because of this, filler traffic could create a substantial delay on low BDP networks due to the length of time it takes to send one complete packet. Figure 15 shows a possible negative affect of filler traffic on pre-existing traffic. Two green pre-existing packets must wait while the black filler packet is being sent for a relatively long time.

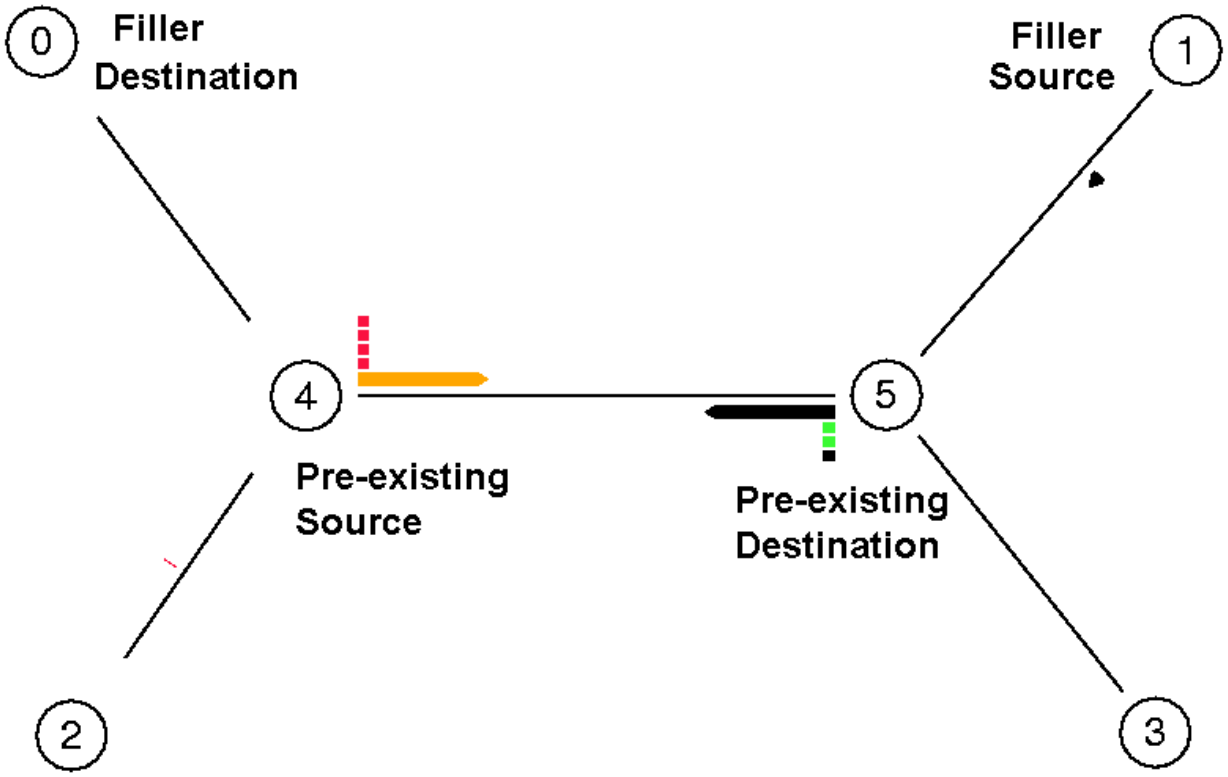

Figure 15. Effects of Low BDP networks

B. Program Source

```
// Adam Feldman
// NS-2 Output Analyzer – Project.cc

#include "jkstring.h"
#include "jklist.h"
#include <iostream.h>
#include <fstream.h>
#include <math.h>

#define TRUE 1
#define FALSE 0

#define NODE1 4
#define NODE2 5
#define ACK_SIZE 54
#define TCP_SIZE 1000.0
#define BREAKDOWN_SIZE 10000
#define BREAKDOWN_MULT  1000
#define MAX_PACKET_SIZE 50000

struct trace {
  char eventType;
  float time;
  int startNode;
  int endNode;
  string packetType;
  int packetSize;
  string flags;
  int flowIdentifier;
  float packetSource;
  float packetDestination;
  int sequenceNumber;
  int packetIdentifier;
};

// Reads in a line from the specified file stream and returns it as a trace
trace GLine(bool flag);

// Function that converts a numerical string into an int
int String2Int(string stringVal);

// Global Variables
ifstream inFile;

int main(int argc, char *argv[]) {
  // Variable List
  ofstream out;
  ofstream charts("data.txt", ios::app);  // Open file stream to append to file
  trace tempTrace;
  list<trace> traffic;
  int flow = 0, c = 0;
  int breakdown[4][BREAKDOWN_SIZE];
  list<int> BWBreakdown[4];
  double delay = 0;
  double avgBW[4], avgDelay[4], avgReg[4], ackCompRatio[4], regDelay[4], ackDelay[4];
  int numberReg[4], numberAcks[4];
  double total[4], totalBytes[4], received[4], receivedBytes[4], dropped[4], droppedBytes[4];
  double firstTime[4], lastTime[4], max[4], min[4];
  bool found = FALSE, flag = FALSE;
  int seqNum = 0;
```

```
// Set all elements of all arrays to 0
for(int i = 0; i < 4; i++) {
  avgBW[i] = avgDelay[i] = avgReg[i] = ackCompRatio[i] =
    regDelay[i] = ackDelay[i] = total[i] = totalBytes[i] = received[i] =
    receivedBytes[i] = dropped[i] = droppedBytes[i] = firstTime[i] = lastTime[i] =
    max[i] = min[i] = 0;
  numberReg[i] = numberAcks[i] = 0;
  for(int j = 0; j < BREAKDOWN_SIZE; j++)
    breakdown[i][j] = 0;

}
// Inform the correct usage and then exit if incorrect
if(argc != 5) {
  cerr << "Usage:  program.exe <input> <output> <parameter> <value>" << endl;
  cerr << "   or:  program.exe x <output> <parameter> <value>" << endl;
  cerr << "Second usage takes input from the stdin stream." << endl;
  return 1;
}
// Open the input and output file streams
if(*argv[1] != 'x') {
  flag = TRUE;
  inFile.open(argv[1]);
}
out.open(argv[2]);

// Output chart parameter and value for Excel
charts << argv[3] << ", " << argv[4] << ", ";

// Get the first line of the input file, and convert to a trace (tempTrace)
tempTrace = GLine(flag);

// For each line (event) in the input file, convert to a trace (tempTrace)
for(;tempTrace.eventType != 'X'; tempTrace = GLine(flag)) {
  // It is assumed that each event in the input file deals solely with the two middle nodes
  // Set flow to the correct value (0-3)
  flow = int(tempTrace.packetSource);

  // If tempTrace is type + (enque), put event in traffic list
  if(tempTrace.eventType == '+') {
    traffic += tempTrace;
  }

  // Else if tempTrace is type - (deque)
  else if(tempTrace.eventType == '-') {
    if(BWBreakdown[flow].length() - 1 != int(tempTrace.time * 10)) {
      BWBreakdown[flow] += tempTrace.packetSize;
    }
    else
      BWBreakdown[flow][BWBreakdown[flow].length() - 1] += tempTrace.packetSize;
  }

  // Else if tempTrace is type r (receive)
  else if(tempTrace.eventType == 'r') {
    // If Ack Packet, set seqNum
    if(tempTrace.packetType == "ack")
      seqNum = tempTrace.sequenceNumber;

    // Increment total delay time of total received packets & bytes
    received[flow]++;
    receivedBytes[flow] += tempTrace.packetSize;

    // Set lastTime for this flow to this event's time (will end up being the time of last event)
    lastTime[flow] = tempTrace.time;
```

```
  // Look through traffic to find event when this packet was deque'd at other middle node
  for(int i = found = 0; i < traffic.length(); i++)
    // If the event is found
    if(tempTrace.packetIdentifier == traffic[i].packetIdentifier) {
      // Set the delay to the time from deque to receive
      delay = tempTrace.time - traffic[i].time;

      // Remove event from traffic, set found, and exit loop
      traffic -= i;
      found = TRUE;
      break;
    }

  // If event is never found, output error and exit
  if(!found) {
    cerr << "EVENT RECEIVED BUT NEVER ENQUED:  " << tempTrace.packetIdentifier << endl;
    return 2;
  }

  // Keep min and max set to the minimum and maximum delays so far
  if(delay > max[flow]) {
    max[flow] = delay;
    // If delay is bigger than should be allowed, output error and exit
    if(delay > (BREAKDOWN_SIZE / double(BREAKDOWN_MULT))) {
      cerr << "DELAY GREATER THAN " << BREAKDOWN_SIZE  / double(BREAKDOWN_MULT) << ":  "
           << delay << endl;
      return 5;
    }
  }
  else if(delay < min[flow])
    min[flow] = delay;

  // Add to delay list, and increment total delay time of correct packet type (ack or reg)
  breakdown[flow][int(delay*BREAKDOWN_MULT)]++;
  if(tempTrace.packetSize == ACK_SIZE || tempTrace.packetType == "ack") {
    numberAcks[flow]++;
    ackDelay[flow] += delay;
  }
  else {
    numberReg[flow]++;
    regDelay[flow] += delay;
  }

  // If firstTime for this flow has not been set
  if(!firstTime[flow]) {
    // Set it to the time this packet dequed
    firstTime[flow] = tempTrace.time - delay;
    // Set initial max and min times
    max[flow] = min[flow] = delay;
  }
}

// Else if tempTrace is type d (drop), increment dropped and droppedBytes, remove from traffic
else if(tempTrace.eventType == 'd') {
  // Increment dropped and dropped Bytes to account for all dropped packets
  dropped[flow]++;
  droppedBytes[flow] += tempTrace.packetSize;
```

```
    // Look through traffic to find event when this packet was deque'd at other middle node
    for(int i = found = 0; i < traffic.length(); i++)
      // If the event is found
      if(tempTrace.packetIdentifier == traffic[i].packetIdentifier) {
        // Remove event from traffic, set found, and exit loop
        traffic -= i;
        found = TRUE;
        break;
      }

    // If event is never found, output error and exit
    if(!found) {
      cerr << "EVENT DROPPED BUT NEVER ENQUED:  " << tempTrace.packetIdentifier << endl;
      return 2;
    }
  }

  // Send a * to the screen every 50,000 lines to show that the program is still running
  if(c++ == 50000) {
    cerr << '*';
    c = 0;
  }
}

// Make sure that all undropped packets are accounted for (???)
if(traffic.length())
  cerr << endl << "TOO MANY PACKETS LEFT OVER:  " << traffic.length() << endl;

// Close input file
if(flag)
  inFile.close();

// Fix the number of decimal places shown in output
out.setf(ios::fixed);

// Output inFile information
if(flag)
  out << "Input File:  " << argv[1] << endl
      << "-----------------------------" << endl << endl;

// For each flow i, from 0 to 3
for(int i = 0; i < 4; i++) {
  // Compute statistics for each flow if there was at least 1 regular packet for that flow
  if(numberReg[i]) {
    // Find total packets and bytes
    total[i] = received[i] + dropped[i];
    totalBytes[i] = receivedBytes[i] + droppedBytes[i];

    // Find overall Average Delay
    avgDelay[i] = (regDelay[i] + ackDelay[i]) / received[i];

    // If there were any ack packets
    if(numberAcks[i]) {
      // Divide ackDelay by the number of ack delays (average ack delay)
      ackDelay[i] /= numberAcks[i];
    }

    // Divide delay by the number of regular packets (average regular packet delay)
    regDelay[i] /= numberReg[i];
  }
}
```

```
// For each flow i, from 0 to 3
for(int i = 0; i < 4; i++) {
  // Output statistics for each flow if there was at least 1 regular packet for that flow
  if(received[i] > numberAcks[i]) {
    // Set ackCompRatio for this flow
    ackCompRatio[i] = ackDelay[(i ^ 1)] / regDelay[i];

    // Set precision to 7 decimal places
    out.precision(7);

    // Output the Flow number
    out << "Flow From Node " << i << " to " << (i ^ 1) << endl << endl;

    // Output Delay information
    out << "Average Reg Delay:  " << regDelay[i]     << " seconds" << endl;
    out << "Average Ack Delay:  " << ackDelay[i]     << " seconds" << endl;
    out << "Ack Comp. Ratio:    " << ackCompRatio[i] << " Ack/Reg" << endl << endl;

    out << "Minimum Delay:      " << min[i]          << " seconds" << endl;
    out << "Maximum Delay:      " << max[i]          << " seconds" << endl;
    out << "Average Delay:      " << avgDelay[i]     << " seconds" << endl << endl;

    out << "Delay Density:"       << endl;

    out.precision(5);

    // For each slot in breakdown, output the range and concentration
    for(int j = 0; j < BREAKDOWN_SIZE; j++) {
      if(breakdown[i][j] > 0) {
        out << j / double(BREAKDOWN_MULT) <<  "s to "
            << (j + 1) / double(BREAKDOWN_MULT) << "s: ";
        for(int k = 7; k-- > (int(log(breakdown[i][j]+1) / log(10)) + 1); out << " ");
        out << breakdown[i][j] << endl;
      }
    }

    // Output a blank line
    out << endl;

    // Set precision to 0 decimal places
    out.precision(0);

    // Output the total and dropped packet information
    out << "Total Packets:    " << total[i] << " (" << totalBytes[i] << " bytes)" << endl;
    out << "Dropped Packets:  " << dropped[i] << " (" << droppedBytes[i] << " bytes)";

    // Set precision to 5 decimal places
    out.precision(5);

    // If any packets were dropped, display the percentage information
    if(dropped[i])
      out << endl <<"Percent Dropped:  " << dropped[i] * 100.000 / total[i] << "% ("
                  << droppedBytes[i] * 100.000 / totalBytes[i] << "% of bytes)";

    // Output a blank line
    out << endl << endl;

    // Set precision to 2 decimal places
    out.precision(2);
```

```
      // If firstTime and lastTime are not the same, output bandwidth information
      if(lastTime[i] != firstTime[i]) {
        avgBW[i] = (totalBytes[i] - droppedBytes[i]) / 1024 / (lastTime[i] - firstTime[i]);
        out << "Average Bandwidth Across Center Nodes:" << endl
            << "      " << avgBW[i] << " Kilobytes/second" << endl;
      }

      // Output a blank line
      out << endl;

      // Output Bandwidth CDF info
      for(int j = 0; j < BWBreakdown[i].length(); j++)
        out << BWBreakdown[i][j] << "  ";

      // Output a divider between the flows
      out << endl << endl << "==================================================" << endl << endl;
    }

    // Output chart information for Excel
    charts << avgDelay[i] << ", ";
    if(dropped[i])
      charts << dropped[i] * 100.000 / total[i] << ", ";
    else
      charts << "0, ";
    charts << ackCompRatio[i] << ", " << avgBW[i];
    if(i != 3)
      charts << ", ";
  }

  // Output new Throughput info (in KB/sec)
  charts << ", " << double(seqNum) * TCP_SIZE / 1024.0 / lastTime[0];

  // Output a blank line to charts
  charts << endl;

  // Close output files
  out.close();
  charts.close();

  // Reset variables
  traffic.kill();

  // Exit smoothly
  return 0;
}

trace GLine(bool flag) {
  // Variable List
  trace tempTrace;
  char tempChar;

  if(flag) {
    inFile >> tempTrace.eventType;
    inFile >> tempTrace.time >> tempTrace.startNode >> tempTrace.endNode;
    inFile >> tempTrace.packetType;
    inFile >> tempTrace.packetSize;
    inFile >> tempTrace.flags;
    if(inFile.eof())
      tempTrace.eventType = 'X';
    inFile >> tempTrace.flowIdentifier >> tempTrace.packetSource >> tempTrace.packetDestination
           >> tempTrace.sequenceNumber >> tempTrace.packetIdentifier;
  }
```

```
  else {
    cin >> tempTrace.eventType;
    cin >> tempTrace.time >> tempTrace.startNode >> tempTrace.endNode;
    cin >> tempTrace.packetType;
    cin >> tempTrace.packetSize;
    cin >> tempTrace.flags;
    cin >> tempTrace.flowIdentifier >> tempTrace.packetSource >> tempTrace.packetDestination
        >> tempTrace.sequenceNumber >> tempTrace.packetIdentifier;

    if(tempTrace.packetSize > MAX_PACKET_SIZE) {
      tempTrace.eventType = 'X';
    }
  }

  return tempTrace;
}

int String2Int(string stringVal) {
  // Variable List
  int value = 0, modifier = 1;

  // Convert each digit from a character to an int, and multiply
  // by the modifier which signifies place in the number
  for(int i = stringVal.length() - 1; i > -1; i--, modifier *= 10)
    value += (stringVal[i] - '0') * modifier;

  return value;
}
```

C. Program Documentation

project.cc (requires *jkstring.h* and *jklist.h*)
Compiled to '**project**' using g++ - "`g++ project.cc -w`"

Purpose: This software is used to analyize the output of ns-2 network simulator.

Usage 1: program.exe <input> <output> <parameter> <value>

<*input*>: Name of the file containing the output from NS-2, which is in this form:
+ 0.00107 4 5 udp 42 ------- 1 2.0 3.0 0 0

The length of the file is not limited, however, there should be no extraneous information present (aside from blank lines at the end). Currently, the input file should contain only information relating to the 2 center nodes (nodes 4 and 5) - any other lines will be ignored.

<*output*>: Name of the file where the output is sent. Format of output is described below.

<*parameter*> <*value*>: This is the name and value of the parameter which is being changed in this experiment. They are included on the command line for the purpose of associating the data in *data.txt* with this experiment. *data.txt* is used to easily create an Excel chart for studying the results of multiple experiments. Format of *data.txt* is described below.

Usage 2: project x <output> <parameter> <value>
The only change for this second usage is that the first parameter is an 'x' instead of a filename. This is to allow the program to take input from the standard input stream (instead of a file), so that the program will run on the output directly from NS-2 (or any other program, such as gzip), without having to first save the data to disk.

Output file: If the program is executed via usage 1, the output file first contains the name of the input file. Next is the label of which flow the data pertains to. Each of the four flows is only included if any packets traveled across that link. The first data for each flow is basic data about delays and the ack compression ratio in an easy to read format. Next is a breakdown of the delay cdf – the number of packets which had a delay in the specified time range. This is followed by dropped packet information and average bandwidth utilization. Finally, is a list of the size of data crossing the central link in 0.01 second increments.

Data.txt file: Each line in this file is comma delimited, and represents one experiment. A single line in data.txt corresponds to this format:

BW, 10, 0.00537133, 6.51442, 0, 116.857, 0, 0, 0, 0, 0.00600353, 0.039953, 0.82705, 132.335, 0.00735259, 0.00163182, 0.599299, 183.515, 101.314

The first block (BW) is the name of the changing parameter (<parameter>), while the second (10) is the value of that parameter for this experiment (<value>). The next four blocks correspond to values for the flow from node 0 to node 1 (0.00537133, 6.51442, 0, and 116.857). In order, they are delay (in seconds), percent of packets dropped, the ack compression (ratio of

regular packets divided by ack packets), and the average bandwidth (in KB/sec). The next four blocks are the same values for the flow from node 1 to node 0 (0, 0, 0, and 0, in this case). The next eight blocks are for node 2 to node 3 (0.00600353, 0.039953, 0.82705, and 132.335), then from node 3 to node 2 (0.00735259, 0.00163182, 0.599299, and 183.515). The final block is the value for calculating the throughput, which is to take the highest packet number received, multiply by the packet size, and divide by the length of the experiment.

2_Bandwidth.xls: This is an example Excel chart file which can be easily produced by using the numbers in data.txt. Simply order the rows from lowest to highest (by changing parameter), and enter all of the rows, 4 columns at a time, into the appropriate flow's cells. This will automatically cause the charts to graph the correct data. If creating a chart from one of a different type (using one from Bandwidth to fill in Latency, for example), be sure to correct the values of the parameter, which are used to plot on the x-axis. Finally, label the chart as desired, and it is complete.

D. References

[1] W. Willinger and V. Paxson. Where Mathematics Meets the Internet. *Notices of the AMS*, 45(8):961-970, Sept. 1998.

[2] A. Odlyzko. The Internet and Other Networks: Utilization Rates and Their Implications. Technical Report 99-07, DIMACS, 1999.

[3] B. Leida. A Cost Model of Internet Service Providers: Implications for Internet Telephony and Yield Management. Master's thesis, 1998.

[4] A. Odlyzko. Data Networks are Lightly Utilized, and Will Stay That Way. Technical Report 99-10, DIMACS, 1999.

[5] V. N. Padmanabhan and R. H. Katz. TCP Fast Start: A Technique For Speeding Up Web Transfers. In Proc. IEEE Globecom '98 Internet Mini-Conference, 1998.

[6] B. D. Davison and V. Liberatore. Pushing Politely: Improving Web Responsiveness One Packet at a Time. Technical Report DCS-TR-415, Department of Computer Science, Rutgers University, June 2000.

[7] NS-2 is located at: http://www.isi.edu/nsnam/ns/

[8] Traces and related information can be found at: http://www.eecs.harvard.edu/net-traces/

[9] M. Allman. A Web Server's View of the Transport Later. ACM Computer Communication Review, 30(5), October 2000.

[10] S. K. Schneyer. Survey Paper on TCP. University of Karlstad, November 1998.

[11] Information about WinDump can be found at: http://netgroup-serv.polito.it/windump/