Improving Network Bandwidth: 
Using Overlay Nodes to Help Transfer Data

Brian Curtin
Yang Richard Yang, Advisor

CPSC490b Senior Project
Yale University Department of Computer Science

May 5, 2003

Abstract. This project implemented a new file transfer protocol in which the source sends data to the destination along more than one route. The goal is a system whose overhead costs are offset by the performance gains of using additional bandwidth.

One connection in each file transfer is based on TCP/Reno; it sends the majority of packets. Additional connections use nodes in an overlay network to forward data along different routes. These connections are based on TCP/Nice. They send far fewer packets, so as to avoid interfering with other traffic in the network. Whenever a connection’s window is not full, it asks the source to allocate a block of data, which it then sends along its prescribed route. Based on packet acknowledgements, the connections adjust their window sizes for congestion control and obtain more data to send.

Evaluation of a Java implementation suggests that the overhead costs to allocate and acknowledge packets for multiple connections is high. Since transmission rates are so fast, any additional delay has a large impact on performance. Experimental trials suggest a deeper problem, if it can be called that. Current networks, including the Internet, are already very fast and reliable. Forcing packets to travel along different routes substantially increases the chances of loss and reordering, which both cause the source connections to reduce their window sizes, thus limiting bandwidth. This “problem” is a major reason why this implementation did not improve transfer bandwidth. Still, further work on this protocol may prove beneficial.
1 Introduction and Design

Data transfers in the Internet today mostly occur along a single path from source to destination. Changes to network topology, such as a link between two nodes malfunctioning, do make routers find alternative paths to a destination. However, routers do not use these alternative paths during a file transfer. The idea behind this paper is that traffic flows could use a small portion of the additional bandwidth in these alternative paths, with the goal of improving throughput.

In order to use more than one route in the network, data must be split up, sent along the different routes, and then reassembled at the destination. Of course, currently in the Internet, data is already split up into smaller packets that get sent (along one route) and reassembled at the destination. Our protocol adds a step of redirection at the source: many packets still are sent “directly” to the destination, but some packets are sent to other nodes responsible for forwarding these packets to the destination. In this way, some packets necessarily use different routes to arrive at the destination. By having more packets out in the network at a given time, all headed for one destination, overall throughput potentially will be increased.

An overly simple implementation of this idea would send an equal number of packets using each of N connections. Bandwidth for the transfer would increase N-fold, and we believe this gives an unequal share of network bandwidth to the transfer. Therefore, we decided to have two types of connections. One is “regular” in the sense that it sends the same number of packets as if it were not being helped. The other is “helpful” in that it sends far fewer packets so as to avoid interfering — too much — with other connections on the network. Deciding how many packets each connection should send falls into the realms of flow control and congestion control.

TCP connections employ a sliding window for flow control. The size of this window determines how many packets can be outstanding in the network. Every connection used by the source in our protocol is based on a version of TCP; each has its own window of outstanding packets. These windows are coordinated by the source so that no packet is being sent by two connections at one time.

TCP/Reno [1] is widely used as a congestion control scheme for TCP connections in the current Internet. TCP/Reno increases its window size first in a “slow start” phase,
during which time the window grows exponentially fast. When the size of the window reaches a certain threshold, any further window size increases occur in a “congestion avoidance” phase, slowing the growth of the window to a linear expansion. If the connection detects that a packet has not arrived at the destination, the window size is decreased, reducing the number of outstanding packets in the network that belong to the transfer. TCP/Reno has been shown to be aggressive in its use of network bandwidth, yet fair in its ability to share bandwidth with other connections. The “regular” connection in our protocol will be based on TCP/Reno.

Other versions of TCP have been proposed with differing congestion control schemes. TCP/Vegas [2, 3] is one such scheme that increases or decreases its window size based on changes to network congestion detected by measuring packets’ round-trip time (RTT). Since TCP/Vegas detects congestion on every packet, it is more dynamic, and can consequently avoid a situation where the network is congested because the sender itself is sending too many packets, which can happen in TCP/Reno.

Proponents of TCP/Vegas claim that this version yields significantly better throughput than TCP/Reno. However, TCP/Vegas is not widely used in the current Internet because it would have to work alongside TCP/Reno. Experiments with these versions coexisting [4] show that TCP/Reno unfairly uses much more network bandwidth than TCP/Vegas due to its aggressive algorithm for increasing window size. The conclusion is that TCP/Vegas yields too much bandwidth to be a useful scheme for transferring data on its own. But this submissiveness is just what we need for the “helpful” connections in our protocol.

Eventually, we chose TCP/Nice [5], a variant of TCP/Vegas. The authors suggest that TCP/Nice works well for background transfers, which is similar to our need. They claim that their algorithm “interferes little with foreground flows, [and] reaps a large fraction of spare network bandwidth.” Like TCP/Vegas, TCP/Nice monitors the RTTs of each packet in a window. However, TCP/Nice adds, “first, a more sensitive congestion detector; second, multiplicative reduction in response to increasing round trip times; and third, the ability to reduce the congestion window below one.” These additions are intended to reduce interference with other flows more aggressively than TCP/Vegas does.

Our protocol uses both TCP/Reno and TCP/Nice for the regular connection and helpful connections, respectively. During a transfer, the source employs exactly one
TCP/Reno connection, allocating to it as many packets as its window can hold. These packets travel along a direct route to the destination. The source also employs zero or more TCP/Nice connections. Again, the source allocates to each connection as many packets as the individual connection is supposed to send. The TCP/Nice connections do not send their traffic directly to the destination; rather, they pick nodes in an overlay network to achieve different routes, as mentioned above. First the TCP/Nice connections tell their overlay nodes where they should forward the data (i.e., the transfer destination and port). The overlay nodes then accept only traffic from this connection and simply forward packets to the specified destination. When a TCP/Nice connection closes, its overlay node becomes free to forward other connections’ data.

The destination, in our protocol, acknowledges correctly received packets by sending an ACK packet directly to the source, no matter which connection sent the data. The source is responsible for telling the responsible connection that one of its packets was acknowledged. Because multiple connections are involved in the transfer, both selective ACKs and cumulative ACKs are sent. The details are outlined in section 2. Because our protocol increases the chance that two adjacent blocks of data will travel along two different routes, there is a greater likelihood in our protocol that packets will be received out-of-order. This implies that selective ACKs are more appropriate. Furthermore, the TCP/Nice connections rely on RTT calculations to change their window size, and selectively acknowledging each packet more easily facilitates this.

2 Implementation

We should be clear that although this implementation strives to model our protocol accurately, it is lacking in some respects. Shortcuts were made that simplified the implementation, and allowed us to place greater focus on the transfer mechanics. These are areas that would be improved in a complete implementation. They are addressed in more detail later; look for sentences that begin “Ideally.”

Code to implement our protocol for the source, overlay nodes, and destination was written in Java. Anyone familiar with Java knows that the language compromises performance for features like garbage collection. This compounds the overhead of execution in user space. One illustrative example of this overhead is that the thread handling packet acknowledgements must wait to be context-switched in, which can be an arbitrary delay. This additional delay increases packet RTTs, which affect TCP/Nice
congestion control measures, as well as threaten unnecessary timeout expiration. Ideally, the protocol would be implemented in kernel space in a lower-level programming language to minimize overhead. In the test results in section 3, we tried to reduce the effect of overhead costs. Nevertheless, they should be considered.

Figure 1 gives an overview of the involved classes and their interaction, given one TCP/Reno connection and two TCP/Nice connections. Dashed lines with arrows indicate data transfer over a network. Solid lines indicate classes constructed by the other. Dotted lines indicate messages sent between classes.

The **Source** class is the main entry point for the source. When starting the program, the one available command-line parameter is the maximum number of overlay nodes a file transfer should use. If the parameter is omitted, the default value is zero, and file transfers will not use overlay nodes.

The Source class is responsible for starting file transfers and maintaining a list of overlay nodes. Ideally, this list should be constantly updated to find the best nodes in the overlay network. Then, when a transfer wanted to use an overlay node, it would be given the best available one by the source. In this implementation, however, the user adds overlay nodes to the list by typing `add_overlay <hostname>`. When a transfer asks
for a node, the source allocates the first available one that it finds. To remove an overlay node from the source list, the user types `remove_overlay <hostname>`. The user can see all currently stored nodes by typing `list_overlays`. The command `quit` exits the source.

To start a file transfer, the user types `send <filename> <hostname>`. The source then creates a new `SendFile` thread object that is immediately run.

SendFile coordinates the various connections by maintaining a single buffer (an array of 262,144 bytes) and a collection of markers that allocate blocks of data to the connections. At startup, SendFile creates both a `FileInputStream` object to read from the file and a new `DatagramSocket` to send and receive packets. The timeout of this socket is set to ten seconds.

This implementation uses its own format for packets, even though they are sent using the sockets as `DatagramPacket` objects. The `Packet` class defines an array of up to 1037 bytes. There is a fixed, 13-byte header and up to 1024 bytes of payload data. The header is broken down as follows:

<table>
<thead>
<tr>
<th>Version (4 bits)</th>
<th>ACK</th>
<th>SYN</th>
<th>FIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Port Number (16 bits)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Destination Port Number (16 bits)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequence Number (32 bits)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Acknowledgement Number (32 bits)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Since the `DatagramPacket` object carries port information, the four bytes used on port numbers can technically be left out. But looking ahead to an implementation without Java, these fields will be necessary.

There are four “types” of packets sent in this implementation:

- **Data.** No flags set.
  Sent from source to destination. The Sequence Number for these packets is the byte offset in the transferred file. The Acknowledgement Number is zero.
• **Acknowledgement.** ACK bit set.
  Sent from destination to source. The Acknowledgement Number is the sequence number of the packet being acknowledged (the selective ACK). The Sequence Number is the sequence number of the next expected packet (the cumulative ACK). There is no data payload.

• **Handshake.** SYN bit set.
  The Sequence Number and Acknowledgement Number are both zero. TCP/Reno handshake packets are sent from the source to the destination; TCP/Nice handshake packets are sent from the source to the overlay node. A handshake packet is sent back to the source (without data) to confirm the handshake. The returned packet has port information that the source must use when sending data packets. Details on the data payload are given below.

• **Teardown.** FIN bit set.
  The Sequence Number and Acknowledgement Number are both zero. There is no data payload. Teardown packets are sent from the source to the destination and overlay nodes once all bytes have been acknowledged. A teardown packet is echoed back upon reception.

  SendFile starts a **BufferData** thread that continuously reads data from the input stream and copies it into the buffer. If the buffer is full, the thread waits until packets are acknowledged and space clears, at which point the thread is notified. Once the entire file has been read in, the **FileInputStream** closes and the thread ends.

  SendFile next creates a new **RenoConnection**. RenoConnection, like **NiceConnection**, is derived from the abstract **Connection** class. Upon initialization, RenoConnection sets its window size to one and the threshold between the slow start and congestion control phases to 64.

  SendFile then tells RenoConnection to handshake with the destination. It sends a handshake packet to the destination on port 4949, and then tells the SendFile socket to listen for a response. The data payload for these packets is the name of the file to be transferred. This is simply to give the destination something to name the file it writes. If the response timeout expires, RenoConnection sends another handshake packet, up to three total. If the third fails to be echoed, then an error is passed to SendFile, which ends.

  The **Destination** class runs a loop that listens on port 4949 for handshake packets. When one is received, a new **ReceiveFile** object is started. ReceiveFile initializes a buffer of 262,144 bytes for any out-of-order packets. It creates a socket to receive data, and
sends a handshake packet back to the source. ReceiveFile then goes into a loop, waiting to receive packets.

When RenoConnection receives the handshake packet response from the destination, it tells SendFile the new port number that the destination is using for this file transfer; future data packets will send to the destination on this port number.

After the handshake, SendFile asks the Source for overlay nodes that the NiceConnections can use. For each non-null node address that is returned, a new NiceConnection is created (with an initial window size of one, like RenoConnection), and told to handshake. A NiceConnection handshake is very similar to a RenoConnection handshake, but one difference is that overlay nodes listen on port 4950. Since overlay nodes only accept one connection at a time in this implementation, all data packets sent to the overlay nodes will also be sent to port 4950. Another difference is that the data payload of these handshake packets is much more important: the first four bytes give the address of the destination that the overlay node will forward to, and the next two bytes give the destination’s port number. Like the TCP/Reno handshake, a packet is sent back to the source by the overlay node (without data) to confirm the handshake.

When the connections have finished handshaking, SendFile lengthens the timeout period for the socket to 60 seconds and then starts a Listener thread. The Listener thread begins a loop that receives acknowledgement packets. More details are given below.

After the Listener thread is started, the RenoConnection and NiceConnections are started. Ideally, the RenoConnection could start sending packets as soon as it had finished its handshake. But the SendFile socket still must listen for responses to the NiceConnection handshakes, preventing it from listening for ACK packets. In the worst case, every NiceConnection handshake will time out — up to thirty seconds for each desired connection — before the RenoConnection can start. To solve this problem, another port could be added for handshaking and tearowns, separate from data transfer. Alternatively, intelligence could be added to the Listener thread so that it passed handshake responses to the appropriate connection. This would address another issue, which is that this implementation has no ability to change NiceConnections. That is, if one NiceConnection is not working well, it cannot be closed and replaced by a new NiceConnection, as it would need to handshake before sending data.
When a RenoConnection or NiceConnection begins its main thread, it enters a loop that makes sure the connection’s window is full. First, it checks to see if its counter of allocated packets is smaller than the current window size. If it is not, the execution waits, to be notified later once the number of allocated packets has gone down (because of an acknowledgement) or the window size has increased (because of congestion control). If the window is not full, the connection asks SendFile to allocate a block of data to the connection.

A brief aside on the **BufferMarker** class. SendFile uses BufferMarker objects to keep track of buffered bytes that have been allocated to a connection. In a sense, they correspond to packets. Whenever a block of data is allocated, a new BufferMarker object is created. It stores the first byte number of the allocated block of data, and the length of the block. It also stores a pointer to the Connection object responsible for this block, and the system time when this block was sent.

SendFile keeps track of these allocated blocks in a vector of BufferMarkers, ordered by starting byte number. So when SendFile is told to allocate a block of data, it first searches the existing BufferMarkers for any gaps that have not been allocated to a connection. If no gap is found, SendFile checks for buffered bytes after the range of BufferMarkers. If a gap is found, or if there are unallocated bytes after the BufferMarkers, SendFile allocates these bytes, up to the maximum packet length of 1024. A new BufferMarker is made and added to the vector, the requesting connection’s counter of allocated packets is incremented, and the BufferMarker is returned. If no BufferMarker could be allocated, null is returned.

The Connection object gets the new BufferMarker and immediately sends a packet if the marker is non-null. If the marker is null, the Connection waits 100ms before again asking SendFile to allocate data. Since a null BufferMarker is returned when no data is available, the hope is that in the time spent waiting, the buffer will add new data.

When sending a packet, the Connection copies the bytes pointed to by the BufferMarker into a new Packet object. The BufferMarker’s timestamp is set to the current time, and the packet is sent in a DatagramPacket object. Each Connection has a timeout timer that simply calls timeout code after a given amount of time. If this timer is not running when the packet is sent, the Connection starts it.
When an overlay node receives a DatagramPacket, it parses out the Packet object. It changes the source and destination port numbers in the header, and sends a new DatagramPacket to the destination.

When ReceiveFile receives a DatagramPacket object from either the source or an overlay node, it parses the Packet object and checks to see if it is a teardown packet. If it is, it echoes the teardown, closes the socket and closes the file. If the packet is of data, it calculates the difference between the sequence number and the number of bytes written to the file. If they are not the same, then the packet is out-of-order, so ReceiveFile buffers the data at the calculated offset. ReceiveFile also creates a BufferMarker (the same as above, except without a Connection pointer or a timestamp) and adds it to a vector. If the packet is in-order, its data is written to the file. ReceiveFile shifts the buffer by the packet length, and then checks the vector of BufferMarkers. If previously out-of-order packets are now at the front of the buffer, they are written, and the buffer is shifted again.

Whenever a packet is received, either in-order or out-of-order, an acknowledgement packet is sent to the source. As mentioned above, both a selective ACK (the first byte of the acknowledged packet) and a cumulative ACK (the destination’s next expected byte) are sent in the acknowledgement packet. When the Listener thread running on the source receives this packet, it compares the two values. If the cumulative ACK is greater than the selective ACK then the packet arrived in-order. The acknowledged bytes are removed from the buffer. Then, the vector of BufferMarkers is searched, looking for and removing those that point to blocks lower than the cumulative ACK. For each of these, the associated connection’s number of allocated packets is decremented, its timeout timer is restarted, and its congestion control code is called. If, however, the cumulative ACK is less than the selective ACK, then the packet was out-of-order. The method still calls its connection’s congestion control code. However, the timeout timer is not restarted, and a counter of duplicate ACKs is incremented. If the counter reaches three (it is reset whenever the cumulative ACK is greater), then a triple duplicate ACK has been received, and appropriate congestion control code is called in the Connection responsible for the next byte expected by the destination. This code resends the expected packet immediately and adjusts the window size.

In addition to changing their window size, Connections update the length of the timeout timer every time a packet is received. A packet’s RTT is calculated by
subtracting the BufferMarker’s timestamp from the current time. This value is folded into estimates of RTT and RTT deviation. For both RenoConnections and NiceConnections, the length of time the next timer will wait before timeout is \( RTT + 4 \times RTT\text{deviation} \).

After the last byte has been acknowledged, SendFile shuts down. It sends teardown packets to the overlay nodes and the destination, similar to the way it handshakes with these hosts. SendFile then closes the DatagramSocket and the Connection objects (i.e., breaks out of their main threads). Finally, SendFile frees the overlay nodes allocated to it by the Source. The transfer is complete.

3 Experimentation

A simple measure of the efficacy of our protocol is to time file transfers. If our protocol is useful, adding NiceConnections should increase the rate of the transfer, when compared to the speed of a single RenoConnection.

Initial tests were performed within the local area network on the Yale campus. The source was a Unix computer; the destination was a Windows XP computer. Both machines were running on Pentium IV processors. Twenty pings between the two ranged from 0.772 to 1.930ms, with an average of 1.119ms.

Transfers involved files of four different sizes: 128KB, 256KB, 512KB and 1024KB (1KB = 1024 bytes). First, files were transferred using one RenoConnection and no NiceConnections. This was to get a sense of the baseline transfer rate. Then one NiceConnection was added, using a Unix computer in the same cluster as the source. The source to overlay node ping average was 0.149ms. Then a second NiceConnection was added, using another Unix computer in the same cluster. The source to overlay node ping average was 0.187ms. For each combination of file size and connection configuration, ten trials were run.

The results of these tests were mixed. A table of the transfer bandwidths (in kilobytes per second) is shown in Table 1. A graph of the data is shown below.
Interpreting these results, it seems that a single connection over a LAN is already a very good and consistent option. In fact, a number of the transfer trials with one RenoConnection worked in one pass (i.e., no packets needed to be resent), achieving very high bandwidth. However, bandwidth dropped significantly as soon as a packet was lost; note the wide discrepancy between minimum and maximum values for transfers using one RenoConnection. This is the cost of having to resend data and decrease the window size.

Adding helper connections to the transfers offered some encouragement. One extra NiceConnection boosted bandwidth for the 128KB and 1024KB files, and kept

<table>
<thead>
<tr>
<th></th>
<th>128KB</th>
<th>256KB</th>
<th>512KB</th>
<th>1024KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Reno</td>
<td>198.05</td>
<td>214.08</td>
<td>187.97</td>
<td>189.55</td>
</tr>
<tr>
<td></td>
<td>101.27</td>
<td>141.83</td>
<td>88.09</td>
<td>93.40</td>
</tr>
<tr>
<td></td>
<td>369.94</td>
<td>318.01</td>
<td>310.49</td>
<td>275.71</td>
</tr>
<tr>
<td>1 Reno, 1 Nice</td>
<td>247.30</td>
<td>213.39</td>
<td>97.77</td>
<td>211.59</td>
</tr>
<tr>
<td></td>
<td>146.12</td>
<td>162.13</td>
<td>88.46</td>
<td>182.27</td>
</tr>
<tr>
<td></td>
<td>389.06</td>
<td>306.22</td>
<td>112.70</td>
<td>241.22</td>
</tr>
<tr>
<td>1 Reno, 2 Nice</td>
<td>148.68</td>
<td>179.11</td>
<td>103.22</td>
<td>190.40</td>
</tr>
<tr>
<td></td>
<td>74.38</td>
<td>118.79</td>
<td>73.66</td>
<td>107.43</td>
</tr>
<tr>
<td></td>
<td>301.89</td>
<td>244.27</td>
<td>208.05</td>
<td>249.21</td>
</tr>
</tbody>
</table>

Table 1: Bandwidth (in KB per second) for transfers of different file sizes and connection combinations

Interpreting these results, it seems that a single connection over a LAN is already a very good and consistent option. In fact, a number of the transfer trials with one RenoConnection worked in one pass (i.e., no packets needed to be resent), achieving very high bandwidth. However, bandwidth dropped significantly as soon as a packet was lost; note the wide discrepancy between minimum and maximum values for transfers using one RenoConnection. This is the cost of having to resend data and decrease the window size.

Adding helper connections to the transfers offered some encouragement. One extra NiceConnection boosted bandwidth for the 128KB and 1024KB files, and kept
bandwidth the same for the 256KB file. The 512KB bandwidth, however, dropped considerably. It is conceivable that network traffic was higher during those trials; trials were run close together in this series of tests. Further testing may prove that this set of transfers was merely unlucky.

Adding a second NiceConnection was typically not beneficial. For the 128KB, 256KB and 512KB files, the bandwidth of one RenoConnection and two NiceConnections was worse than that of just one RenoConnection. The 1024KB transfer had a very slight improvement, yet was worse than one RenoConnection and one NiceConnection. It seems that the overhead cost of two extra connections exceeds any improvement in performance.

It is tempting to blame these underwhelming results on the fact that LAN transfers already perform incredibly well. A single connection is so fast and so accurate that it is very hard to improve upon. Indeed, sending packets along different routes greatly increases the chance that packets will be received at the destination out of order, or will be dropped completely. These factors combined with the overhead of extra connections — allocating packets, thread switching, etc. — to degrade the transfer performance. In a LAN, any performance gains from extra bandwidth are drowned out.

Therefore, further testing was done using computers not locally connected. In these experiments, the source was the Windows XP machine used earlier as the destination, located on the Yale campus. The same overlay nodes as before were used. This time, the destination was a commercial web server, running Windows 2000 Server. Though it was also based in New Haven, traffic between these computers had to travel over the broader Internet. The average ping time between the source and the destination was 18ms — substantially larger than the 1.119ms from the first series of tests. Pings from the source to the overlay nodes averaged 0.913ms and 1.137ms. The same files were again sent ten times each.

The results of these tests were discouraging. A table of the transfer bandwidths (again in kilobytes per second) is shown in Table 2. A graph of the data is shown below.
The only time the protocol did better than the baseline was the 128KB file transfer, with one added NiceConnection. All other connection combinations performed worse than the single RenoConnection.

While testing, we noticed that a surprisingly large number of transfers using one RenoConnection completed in one pass. This is the same phenomenon that was seen in the LAN tests. During these transfers, the source transmitted every packet only once, since none was dropped and the few that arrived out of order were soon followed by the missing packets. This implies that the network is already as good as it is going to get,

<table>
<thead>
<tr>
<th></th>
<th>128KB</th>
<th>256KB</th>
<th>512KB</th>
<th>1024KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Reno</td>
<td>130.47</td>
<td>155.31</td>
<td>159.46</td>
<td>168.74</td>
</tr>
<tr>
<td></td>
<td>101.11</td>
<td>130.08</td>
<td>138.27</td>
<td>159.45</td>
</tr>
<tr>
<td></td>
<td>163.89</td>
<td>170.67</td>
<td>173.38</td>
<td>173.82</td>
</tr>
<tr>
<td>1 Reno, 1 Nice</td>
<td>142.73</td>
<td>148.96</td>
<td>153.84</td>
<td>157.42</td>
</tr>
<tr>
<td></td>
<td>117.00</td>
<td>128.97</td>
<td>121.36</td>
<td>139.74</td>
</tr>
<tr>
<td></td>
<td>157.64</td>
<td>163.89</td>
<td>164.63</td>
<td>168.92</td>
</tr>
<tr>
<td>1 Reno, 2 Nice</td>
<td>128.40</td>
<td>131.06</td>
<td>135.96</td>
<td>139.14</td>
</tr>
<tr>
<td></td>
<td>93.09</td>
<td>103.69</td>
<td>108.50</td>
<td>107.79</td>
</tr>
<tr>
<td></td>
<td>160.80</td>
<td>159.01</td>
<td>160.65</td>
<td>166.78</td>
</tr>
</tbody>
</table>

Table 2: Bandwidth (in KB per second) for transfers of different file sizes and connection combinations
because, to equal this performance, packets taking different routes must not only arrive, but also they must arrive when the receiver expects them. This is obviously difficult to achieve, and helps explain why bandwidth declined as helper connections were added.

There are some silver linings. The maximum bandwidths of transfers using NiceConnections are very close to the maximum bandwidths when using only a RenoConnection. Perhaps if overhead can be reduced, these maximum bandwidths can surpass those of single connection transfers. Also, multiple-connection transfers along less reliable routes may yet provide relatively better bandwidth, as the performance of these transfers won’t be compared to those that never lose packets.

Still, the results of testing this particular implementation of the protocol indicate that using multiple connections to transfer data is not as good as, and certainly not better than, using just one.

4 Future Work

Some known issues are worthy of mention. They are areas that should be reviewed in future revisions of this implementation.

A major flaw in the implementation, alluded to earlier, is the context switching of the Listener thread. Acknowledgement packets are queuing at the source while other threads send out more packets to fill the connections’ windows. Since TCP/Reno and TCP/Nice detect congestion using acknowledgement packets, late processing of these packets is harmful for two reasons. First, actual congestion problems are addressed late. Many more packets are sent than should be before a dropped packet is detected. This is somewhat acceptable, since the receiver can buffer many out-of-order packets. However, when congestion control finally does detect the problem, the window size is reduced and many previously sent (and successfully received) packets are resent unnecessarily, reducing performance. The second problem of late processing is that connections detect nonexistent congestion. The timeout timer can run, even though appropriate ACK packets have been sent. To help address this problem, the Listener thread priority can be increased, or a delay can be added to ensure that all existing ACK packets have been received. Neither solution seems ideal compared to running the protocol in kernel space with better control over thread execution. Such an implementation would improve
performance for other reasons, and should be a high priority in the evolution of this protocol.

During experimentation, a problem with having connections “own” packets was exposed. We will illustrate this by a hypothetical scenario. Say a RenoConnection sends a packet with sequence number #128000 as part of its window. This packet is accepted by the destination, and an acknowledgement is sent back. While waiting for this acknowledgement, the RenoConnection times out, reduces its window size to one, and consequently frees packet #128000. A NiceConnection now asks to allocate a packet, and SendFile gives packet #128000 to it. When the acknowledgement finally arrives, it is passed to the packet’s owner, which is now the NiceConnection. Since the BufferMarker’s timestamp was reset, any RTT calculation is invalid in addition to the fact that the wrong connection is making the calculation. Solutions to this problem will improve the robustness of the protocol.

Another potential area of improvement is the helper connection. Finding good overlay nodes is a critical aspect of this protocol. The source certainly does not want to send packets on a greatly inferior route, because this cannot improve bandwidth. Methods for selecting the best overlay nodes to use should be incorporated into the design. Also, the congestion control algorithm for the helper connections may be revisited. Perhaps TCP/Nice is too nice. Maybe pure TCP/Vegas would grab more unused bandwidth. Or maybe an entirely different mechanism would be better. Of course, fairness in the overall network is a concern. Any subsequent modification of the protocol must be mindful of its effect on the network. This paper does not address the extent to which the implementation interferes with other network traffic. Rather, it works off of the claim in [5] that TCP/Nice aggressively avoids such interference. Future tests could investigate the veracity of this claim.

5 Conclusions

This project designed and implemented a protocol that uses multiple routes to transfer data from a source to a destination. In addition to the main route that would typically be used, the source sends packets to nodes in an overlay network that forward the data to the destination. So as not to use too much bandwidth, these extra routes are subjected to aggressive congestion control that willingly yields to other flows in the network.
There are clear overhead costs in this system. First, the source must keep track of which connections have sent which packets, as well as which packets have not yet been sent. Second, by using multiple paths to the destination, the likelihood that packets will arrive out-of-order is greatly increased. In our implementation, these costs overwhelmed any benefit from using the helper routes so much that bandwidth was often decreased.

In spite of its problems, we still believe there is promise in our protocol. Improving performance at the source, in terms of allocating and acknowledging packets, will help. A better method for finding overlay nodes should reduce the arrival delay between packets of near bytes. A less sensitive congestion control scheme should favor larger window sizes, and fewer retransmissions. However, a new scheme must still consider fairness in the overall network.

Perhaps the system would benefit from more radical thinking. One suggestion would be to remove or replace triple duplicate ACK code. Packets in this system are going to arrive out-of-order. Unlike a single-connection transfer, the missing packets are likely to still be on their way. So a triple duplicate ACK does not indicate congestion the way it does in a traditional file transfer. Another suggestion would be to treat the network itself as congested, instead of one route. Adjusting all connections’ window sizes based on each acknowledgement might reduce overhead, but it probably will also reduce accuracy.

Certainly more aspects of the protocol can be rethought, and we believe there will be value in doing so. Studies suggest that networks have a significant amount of spare bandwidth. Finding a way to efficiently tap into this bandwidth is a worthwhile goal, as these extra resources can be used to improve the performance of all data transfers.
References


