Finding Ideal Conditions for Arbitrary File Transfers

Hui Dai Michael Lee Steve McGhee

Computer Science 240B - Winter 2003
Department of Computer Science
University of California, Santa Barbara
{huidai, kirbysdl, stevem}@cs.ucsb.edu

1 Introduction

The traditional File Transfer Protocol (FTP) is used by countless applications to transfer files from one machine to another. Successors to the protocol have been developed to improve security (*scp*, *sftp*) or to minimize complexity (*tftp*[2]), but the performance of the protocol has not been improved since the initial design. There have been proposed protocols that improve performance for specific filetypes (VDP[4]), but nothing for arbitrary files.

We studied the performance of the base of the file transfer: the transmission of content using TCP. We investigated its performance on various file sizes, believing it would be inefficient when dealing with small files. We investigated using UDP to transfer files, using varying datagram sizes, hoping to find when UDP was superior to TCP. There has been no previous research on where exactly UDP may surpass TCP, so we believe that our findings will be beneficial to the community.

2 Motivation

This study was initially intended to discover an ideal transfer protocol to be used with storage clusters. Since the file transfers might not be intended to traverse the commodity Internet, TCP's congestion control could potentially be an unnecessary overhead. If so, UDP might provide a faster solution.

The intention was to develop an adaptive file transfer protocol that would choose between TCP and UDP depending on network conditions, file size, and other parameters. This protocol would have a faster average transmission rate than standard TCP-based protocols.

We eventually decided to develop a series of micro-benchmarks to determine where exactly UDP might surpass TCP. Once this was determined, the file transfer protocol would be developed based on those statistics.

We believe that if a faster file transmission scheme can be developed, it will be useful for any high-performance system. If we could develop a scheme for transferring arbitrary files at a faster rate than simple TCP transmission, it would be a valuable contribution to the high-performance computing community.

3 Related Work

"Size-based Adaptive Bandwidth Allocation: Optimizing the Average QoS for Elastic Flows,"[3] by Yang and Veciana, deals with the issue of perceived bandwidth for greatly varying transfers. In this paper the authors state

that an important metric for benchmarking file transfers isn't overall bandwidth consumed on a network link, but rather the average time it takes for file transfers to finish. In order to improve this time, which they term Bit Transmission Delay, they find that it is important to let smaller transfers finish first. While this has interesting ramifications in the context of our project, namely that some requests should be handled first, our main goal for this project was to compare TCP and UDP, and this paper ultimitely wasn't very relevant in that regard. However, the conclusions from this paper could be implemented in later versions of an adaptive file transfer protocol.

"Fair Bandwidth Sharing among Adaptive and Non-Adaptive Flows in the Internet,"[1] by Anjum and Tassiulas devise an algorithm to prevent congestion at gateways in the Internet. They identify two different flows: *adaptive* and *non-adaptive* ones. Adaptive flows adapt to congestion in the network, while non-adaptive ones do not. These types of flows directly corrospond to TCP and UDP data flows. The authors suggest that any time UDP and TCP flow together, UDP will dominate. This was an important discovery for the design of our system. Because we were aware of this, we made sure that we never attempted to transfer files using both TCP and UDP at the same time, ensuring fairness. We also introduce a test below that verifies this finding. In this test we determine that when UDP and TCP are both used, TCP's congestion control always causes it submit to UDP.

4 Proposed Solution

4.1 Overview

We designed and implemented a suite of tests to be run on networks to determine their breakpoints in TCP/UDP performance. These tools include multi-protocol clients, multi-protocol servers, and test scripts to manage the testing. The servers and clients were implemented in C and the test scripts were written in Perl. Here we describe two client/server combinations. The first pair provides the framework for a working adaptive transfer prorotocol, in which the client sends a request to the server, and the server chooses the best transport protocol with which to return the requested file. The second pair of applications is used for testing networks. The client sends a request to the server, specifying the test requirements, including the data size and desired transport protocol. The server then returns the requested data.

The server is implemented using threads as opposed to an event driven model. The reasons for choosing this design model are that threaded model is easier to implement, and we consider the overhead of swapping among threads to be insignificant compared with the general round trip time associated with each file request, and since all threads are so adversely affected, it should not hurt inter-protocol comparisons. The thread structures are located in an array that is bounded at runtime. The default value we chose for the size of this array is 32.

4.2 Protocol Simulator

The first server listens on TCP port for all connection requests, passes the request to a decider function that determines at the time of its invocation, what is the protocol to use. The decider uses the following statistics that it gathers for this determination:

- total number of bytes requested
- total number of files requested
- number of those files sent in UDP
- number of those bytes
- number of files sent in TCP

- number of those bytes
- number of datagrams sent
- number of datagrams acknowledged
- and number of bytes acknowledged to have been received via udp.

The acknowledgements are gathered from clients on a separated UDP socket. The sending of the file, once the protocol has been determined, is trivial. At the end of each transmission, each spawned thread updates the global statistics.

The first client depends on the server to choose the transport protocol, thereby simiulating the client of an adaptive protocol-switching server. Since it is also meant for simulating the requests of many clients, it was implemented as a multithreaded application. Each thread handles a portion of the desired number of requests. After it receives the requested information, it sends the bytes received and packets received to the server in a UDP packet. UDP was chosen as the transport because we did not want the statistics reports to

4.3 Network Testing Suite

The second server, being only a test server, is much simpler in design than the first implementation. It listens for requests and fulfills those requests according to the requests protocol served.

The second client is meant to connect to the second server and receive a constant "stream" of information. It also keeps track of the network performance in terms of bytes received per second, and dropped packets (in the case of UDP). For testing TCP performance, the client connects to the server on its listening TCP port, sends a GET request, receives the relevant data on the same TCP connection, closes the connection, and repeats the process by opening a new TCP connection. For testing UDP performance, the client connects to the server on the TCP port, sends a single request, and closes the connection. As part of the request, it tells the server the port on which it will listen. The server then sends the requested data the specified number of times to the client.

We chose to only use TCP to connect in order to ensure that connections are seen by the server. We only used a single TCP request for multiple UDP transfers because performing TCP operations for each file transfer would nullify the advantages of using UDP. For realism, we used a different TCP connection for each TCP file transfer. Timings for UDP start before the first read from the socket, and end after the last read. Any packet of length less than the specified datagram size is considered an end of file. If the file size is a multiple of the datagram size, the server is expected to send a datagram of size zero. Timings for TCP start before connecting the socket and end after the closing the socket.

The tests are launched by Perl scripts. The scripts are designed to call the client with varying parameters. We chose to vary the datagram size and filesize in the first round of tests. This allowed us to develop graphs showing how the various datagram sizes resulted in different data rates for each file type. The scripts also allowed us to duplicate the tests several times, normalizing the values.

5 Evaluation Methods / Tests

We measured the performance of our system in two schemes, by the number of datagrams, and by the bytes that the client received, in contrast with how many bytes and datagrams that the client should have received. The parameters on which we varied the tests were number of systems, number of threads per system, number of files per thread, file size, and datagram size, with which we used a 0 to indicate a TCP request. On these parameters we varied, ran three times per test, and averaged the results. With these results we generated the graphs for test one.

Number of systems varied between 1 and 4, since each of these systems also ran a varying number of threads to simulate connections, we did not think it necessary to test more than 4 systems. The datagram size varied between 50 and 60000 bytes, for near complete coverage of valid datagram sizes. This test was meant to determine where the ideal UDP datagram size is for any given file size. The second test that we devised addes clients onto existing clients, using the same protocol. With this test, we expected to see the exact diference bertween running different cleints Vs. single client. The third test we ran adds clients onto currently connected clients, using the same protocol of currently connected client threads, and tries to examine the amount of datagrams dropped.

6 Experiment Results

Figure 1 shows data for a single UDP host accessing small files (between 50 and 51200 bytes). As expected, UDP performs much better, and TCP only begins to approach the performance of UDP for the larger request sizes because it passes its slow-start mode. In Figure 2 we see TCP approach closer to UDP performance since bigger files were being requested. Note that some UDP plots have fewer data point because certain tests showed extreme data loss, and those were discarded as unusable.

As one can see from data gathered for simultaneous multiple UDP clients (Figure 3) and single UDP connections (Figure 2), the multiple UDP requests interfere with one another, causing drops in datagrams to increase for both, leading one to believe that UDP is unstable in cases where there are multiple clients.

TCP, as seen in Figure 4, was able to adjust the bandwidth usage of each of the clients until a stable maximum was reached. Although the ’throughput’ for each of the many UDP clients is high in Figure 3, the data loss rate is also quite high. This is evident in Figure 7, where the data loss rate is also graphed as two UDP client connections disrupt the previously established TCP connection: the TCP suffers the congestion control, but was able to right itself back to a similar level of performance; the UDP connections were able to maintain relatively higher throughput, but do so at what seemed to be 30 to 100 percent data loss. A similar trend can also be seen in Figure 6: in this case, the UDP connections were interrupted by TCP connections: after several moments of fluctuations and uncertainties, the two TCP connections were able to stablize while the UDP connection was not.

7 Conclusions

We conclude that TCP is almost always superior to UDP. While transferring small files using UDP during periods of low network traffic will outperform TCP, the guaranteed delivery and reordering properties of TCP make it more desirable in all situations. This is especially true given that our tests were done on a local area network. On a production environment, clients would be expected to connect through a less reliable network and reliability would be of even greater importance.

Possible work for the future includes developing a specific protocol which would use UDP for different conditions including small transfers during periods of very low traffic, clients which are on the same local network as the server, clients whose requests are deemed low priority, etc.

References

[1] F.M. Anjum and L. Tassiulas. Fair bandwidth sharing among adaptive and non-adaptive flows in the internet. In *INFOCOM ’99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies*, March 1999.

[2] N. Chiappa, B. Baldwin, and D. Clark. The tftp protocol (revision 2), July 1992.

[3] S.C. Yang and G. de Veciana. Size-based adaptive bandwidth allocation: Optimizing the average qos for elastic flows. In *Proceedings of IEEE INFOCOM'02*, June 2002.

[4] Chen Z., Tan S., Campbell R., and Li Y. Real time video and audio in the world wide web, 1995.