Spring 5-2013

High Performance Network Communication between High Frequency Application Servers and Android Tablets

Brandon L. Wolfe
University of Southern Mississippi

The University of Southern Mississippi

High Performance Network Communication between High Frequency Application Servers and Android Tablets

by

Brandon Wolfe

A Thesis
Submitted to the Honors College
of The University of Southern Mississippi
in Partial Fulfillment
of the Requirements for the Degree of
Bachelor of Science
in the Department of Computer Science

March 2013

Table of Contents

List of Tables

List of Figures

Chapter 1 - Introduction

Section 1.1: A Trend Toward Mobile Devices

Many people today own some sort of mobile device with capabilities comparable to standard computers. These devices, termed “smart”, come with a variety of hardware and software capabilities. For hardware, the devices can be divided into two primary categories, tablets and smartphones. Tablets have larger screens and often more powerful hardware than smartphones; smartphones still have many of the features found in tablets, but also have the portability, functionality, and smaller screen of the cellular phone. The software that runs on these devices is as diverse as the various types of devices. Phones and tablets house a mobile operating system (OS), which is similar to the operating systems on standard computers, such as Windows or Mac OS. Two of the more popular operating systems for mobile devices are Android OS, developed by Google, and iOS, developed by Apple. Like a standard OS, the phones and tablets can be outfitted with various applications to suit the needs of the user. As powerful hardware becomes available in smaller form factors, the power of a standard computer will be brought to mobile devices, which in turn will lead to the development of more powerful applications for mobile devices.

Section 1.2: Challenges of Mobile Devices

Although mobile devices are rapidly improving, there are still many barriers that do not allow mobile devices to perform with the same potential as standard computers. Software capabilities are dependent on hardware capabilities, and mobile devices are significantly less powerful than the standard computer. Processors (CPUs) are much more powerful for standard computers than for mobile devices, and attempting to add a faster processor to a phone can add many complications.

CPUs for mobile devices are difficult to scale or develop. In part, this has to do with mobile devices have a limited power source, which will need to be used efficiently by the CPU. Also, mobile devices have limited means of dealing with the heat generated by more powerful CPUs, since large fans in mobile devices are not practical. All these things considered, these processors must be powerful enough for applications that are computationally intensive, while also being efficiently managed so that they do not overheat and ruin the devices.

Another hurdle for mobile devices is mobile networking. Since most devices do not have or need an option for connecting to a network via a cable, the devices rely solely on wireless networks. If the device is not near a wireless access point, such as a home wireless router, the device will use a mobile network from a phone company. These mobile networks can be much less reliable than standard Wi-Fi, as the nearest data tower may be miles away or under heavy load from any large number of devices. Even in the best case scenario, these mobile networks are often not as fast or reliable as current local area networks.

Section 1.3: Current Mobile Networking Standards

Mobile networks and devices typically use the same protocols as standard computers for common tasks such as data transmission and device-to-device communication. Data transmission, which could also be termed as “data transport”, is categorized under layer 4, the transport layer, of the Open Systems Interconnection model (OSI). The OSI model is outlined in RFC 1122 (Braden 1989). Two primary protocols used at the transport layer are Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). These protocols have differing applications on mobile devices due to their contrasting methods of transmission.

UDP is a very basic protocol with no error checking performed when data is received. Although UDP may not be reliable for data transfer, its minimal packet structure is excellent for operations requiring minimal interaction. It is often used for DNS queries (Vixie 1999) and for streaming data, including Voice over Internet Protocol (VoIP) and video (Sat, Batu, Wah 2006). UDP is a good choice for sending a large amount of data that does not need to be fully correct. VoIP and video streaming are good candidates for UDP-based protocols, because perfect data is usually unnecessary for the content to be satisfactory.

TCP differs drastically from UDP, as it is a very reliable protocol with several methods to ensure that the data is sent and received correctly. A TCP packet is much more complex due to the reliable scheming of TCP. TCP establishes a connection by using a three-way handshake and continues to check the correctness of transmission throughout the data transaction. However, its error checking can be a problem when data is being sent over a network of bad quality, as the constant handshaking and resending will require a lot of time to correct the bad packets.

Section 1.4: Improvement on Current Mobile Networks

While technological improvements may continue to improve mobile networks, we can attempt to improve speed and integrity of data transfer over current networks by using a protocol that will perform well, even with possible negative conditions of the mobile network. We will use NORM protocol, (NACK Oriented Reliable Multicast), a protocol developed by the Naval Research Lab, and test its performance and reliability against TCP, a standard protocol.

Section 1.5: Research Statement

This thesis will develop methods for comparing NORM protocol in unicast mode to TCP by simulating high performance network communication between Android tablets and high frequency application servers. Since mobile network conditions vary based on service provider or tower distance, we will test over a wide range of network conditions such as high delay or high loss, as well as high chances of packet corruption or duplication. We want to maximize speed of data delivery without sacrificing the integrity of the data. The methods used will be generically applicable to multiple operating system platforms and will be cross-platform compatible.

Chapter 2 - Related Works

Section 2.1: Reliable Protocols Background

There has been much work describing the foundations of reliable network protocols. Reliable protocols are not limited to our scope of testing, that is, unicast protocols. There has also been much work done with reliable multicasting protocols that is relevant to mobile networks. It is also important to note that some multicast protocols also support unicast operation. NORM protocol, although developed as a reliable multicast protocol, can also be used in a unicast setting.

Mankin et al. (1998) discussed two main issues in forming a reliable multicast protocol. First, the protocol would not necessarily be applicable to many different applications. Different applications have different requirements for receiving the messages being sent. While this may be fairly easy to see, the protocol would also need to adapt in its specific application to shrinking or growing of the receiving group. Second, the protocol will have to adequately work with congestion from large receiving groups.

Whetton et. al (2001) also discussed similar standards in the development of a multicast protocol. Like Mankin et al. (1998), the paper discusses both the challenges of congestion control and scalability of the protocol. It also covers security of the protocol. The protocol will need to protect the data being sent from being eavesdropped on as well as ensuring that the data is being sent to the correct receiver. The paper also covers some advantages of building a generic protocol in stages or modules so that can be modified to fit a specific application. These advantages include reduced complexity, verification, and debugging time; easier upgrades to the protocol as new techniques become available; common diagnostics can diagnose issues the protocol may have; modules can be developed independently; and a modular protocol makes deriving a specifically applied protocol easier. In general, the paper describes the need for the protocol to be simple, widely applicable, and high performing.

Section 2.2: Features of a Reliable Protocol

Some possible features of a reliable protocol include either acknowledgement (ACK) negative acknowledge (NACK), forward error correction codes (FEC), and automatic request for retransmission (ARQ). The following papers discuss the various features used in the development of a reliable protocol.

Luby et al. (2002) promotes the use of forward error correction over automatic request for retransmission for two primary reasons. First, the server transmitting would have to deal with potential many requests for a resend, making ARQ impractical for large groups needing a very low error threshold on the data being sent. Second, in some network scenarios, ARQ can only be efficiently implemented with a data carousel method, first described by Acharya, Franklin, and Zdonik (1995) as breaking data down into chunks, then cycle through sending the data until all receivers have received each packet. This method, however, limits performance if a receiver needs to get a packet resent, as it must wait until another batch of information is sent to get the packet it needs. According to the paper, FEC generally can overcome both losses in data and bit-level corruption by allowing the receiver to correct most of the issues in the data stream being sent without having to ask the server for a resend of the data. This method could help free up server resources and balance the error checking workload between the server and the client.

Some modifications to TCP may allow it to perform better in a mobile environment, according to Kim et al. (2012). They propose using algebraic expressions to describe the packets being sent, and when the information is received, the information can be reassembled by solving the expressions and determining if any information was lost or corrupted. Furthermore, they have modified TCP to send fewer acknowledgements (ACKs) if the packets are “seen,” which they describe as a relation to the number of consecutive packets received. The early results were excellent, showing a clear improvement in comparison to standard TCP in bandwidth tests, even on public networks. However, they are clear to state that more testing will need to be done to determine the effects of the modifications in scenarios with a very large number of devices using the protocol.

Section 2.3: Android

Ravindranath et al. propose using internal sensor data to optimize networking protocols to perform efficiently based on the current situation of the phone (2011). Some sensors include GPS, the accelerometer, the gyro, and the compass. By adjusting parameters of a network such as data transfer rate or access point association, overall throughput using protocols such as TCP and UDP can be improved.

Section 2.4: NORM Protocol Background

Forward error correction (FEC) can be combined with negative acknowledgement (NACK) to create an efficient, reliable protocol, according to Adamson et al. (2009). This paper discusses how NORM protocol is oriented around a negative acknowledgement response system for the receivers and could be capable of sending bulk data reliably over Internet Protocol (IP). NORM protocol also includes a congestion control scheme to fairly share available network bandwidth with other protocols, specifically TCP; it also incorporates FEC repair into its implementation to promote balanced error checking between server and client. This protocol allows for three types of bulk data to be sent both reliably and efficiently: data stored in the computer’s static memory, files stored on the hard disk drive of the computer, and continuous data streams from server to clients. Each of these file types would have a unique data type when being sent over the protocol as to allow the receiver to properly and completely allocate storage space in its hard disk or static memory for the files being sent. The protocol does not, however, provide much identification for its data in the header of the packets being sent. This information can be determined from status messages passed between the server and the clients.

Chapter 3 - Methodology

Section 3.1: Network Parameters

There will be four negative parameters that will be induced on network packets for simulating mobile networks: delay, where each packet is delayed for a specified amount of time before sending; loss, where a specified percentage of packets are dropped before sending; corruption, where a specified percentage of each packet is corrupted; and duplicate, where each packet has a specified chance to be resent. For each test of a specified parameter, a file will be transferred 50 times to ensure accurate results.

Section 3.2: Tools

Subsection 3.2.1: TCP

The OpenBSD tool `netcat` will be used for transferring a file through TCP. Due to its lightweight but versatile functionality, `netcat` will require very little configuration or tweaking to use before it can be used for sending files. For use on Android, `netcat` will be installed through a BusyBox installer that will be downloaded from the Google Play store.

Subsection 3.2.2: NORM Protocol

NORM protocol will be built from the most current source files and dependencies. The resulting C library and its Java Native Interface (JNI) will be implemented as a native library in a Java program that will ultimately be used in testing. For use on Android, the library will be built to support ARM architecture processors, a standard type of processor used in mobile devices. There should be no difference in the JNI for the standard and mobile versions of the NORM protocol testing application.

Subsection 3.2.3: Traffic Control (`tc`)

We will use the Linux traffic control application `tc` for inducing the various negative network parameters. This application can be used for both incoming and outgoing traffic on any network interface as well as for any and all types of traffic. The `tc` option of `netem` will be used to add packet delay or corruption. It can also be used to add a predetermined chance of packet loss or duplication in transmission of the data. The `tc` utility will apply negative networking parameters over all subnets used for the transfer of the file in the tests. This completeness will allow data to be delayed, corrupted, lost, or duplicated in transfer. The same effect will be applied to the protocol messages (ACKs or NACKs) that will be sent back to the server that is sending the data. However, special networking conditions and hardware will have to be used for the Android tablet setup, which will be discussed in Subsection 4.2.

Subsection 3.2.4: Android Debug Bridge (`adb`)

Google provides `adb` as an application to communicate with Android devices over USB. The `adb` tool will be beneficial in checking our result data and starting more tests, both TCP and NORM protocol tests, on the Android tablet. Despite our unstable networking environment, we will have unaffected communication to the Android tablet through use of `adb` command options such as `push`, `pull`, and `shell`. The use of `adb` in Android testing will be described in greater detail in Subsection 5.2.

Section 3.3: Application Development

NORM protocol can be compiled into a standalone program for demonstration purposes; however, it does not adequately suit the needs of the tests. Our custom program will allow for accurate testing and more precise control over specific NORM protocol parameters. The structure used in creating the custom NORM protocol Java classes will also be applicable to both standard desktop computers and Android devices. Since we will be using `netcat` as the application to test the effects of negative network conditions on TCP, there will be no need to write a specialized application for our tests.

Section 3.4: Hardware and Network Setup

Subsection 3.4.1: Standard Application Setup

Network setup for standard desktop testing will include three server computers running Ubuntu Server 12.04. Two of these servers will be used as the testing machines, one as a sender and one as a receiver. The third server will act as a router between the two testing servers, as well as a command server for scripting the tests. Since the `tc` utility will affect all types of traffic on an interface, the two testing servers will utilize two Network Interface Cards (NICs). The server will use three, as it must also act as a router for both computers.

For scripting purposes, all three servers will be on one subnet, `192.168.3.0/24`, so that the controlling server may send commands through SSH to both testing servers without having `tc` affect the SSH connection. Each individual testing server will also be on its own subnet, either `192.168.2.0/24` or `192.168.1.0/24,` which will allow for routing through the control server to take place. The control server will enable the Ubuntu IPv4 `ip_forwarding` option for routing between the two test server subnets.

The networking setup for the standard application will allow file transfer rates up to a maximum of gigabit speed (1 Gbps). Thus, the file to be transferred from sender to receiver must be of non-trivial size; that is to say, the file must be large enough so that it ensures the transfer time is above the absolute minimum (non-trivial) time that each respective protocol can take to transfer a file. This will be determined with preliminary testing that has no negative network parameters applied.

Subsection 3.4.2: Android Testing Setup

The Android tablet will be a Samsung Galaxy Tab 10.1 (model GT-P7510), running a stock Android 4.0.4 ROM (Ice Cream Sandwich). Due to the nature of BusyBox, the tablet will be modified to allow root access to applications. This will allow `netcat` to be used in our testing, as `netcat` is the only application from the BusyBox suite that we will need.

The Android tablet will be connected to a Linksys router that is broadcasting an IEEE 802.11n wireless networking signal on channel 6. The maximum data transfer rate over this wireless standard will be 300 megabits per second (Mbps) over this wireless standard. The Android tablet will only be used as a receiver in our tests, as we feel this represents a more accurate portrayal of how mobile devices are used in data transfer. The sender will be a server running Ubuntu 12.04, and it will be connected to the Linksys router over Ethernet. Ethernet has the potential to transmit the data from our server at speeds up to 1 Gbps. However, the effective data rate of sending the file to the Android tablet will only be 300 Mbps, due to our eventual relay of the information over an IEEE 802.11n wireless signal.

Applying `netem` to a networking interface will only affect outgoing data. This is a feature of `tc` that we will need to work around. To account for the desired networking conditions for both incoming and outgoing data, as described in Subsection 1 and Subsection 2.3, we will create an Intermediate Functional Block (IFB) that will allow the incoming data to be adjusted by our networking parameters. This will allow us to do our testing without having to install `tc` on the Android tablet and will allow for easy application of negative network parameters without affecting vital communication with the tablet.

Like the standard application test, the size of the file that will be transferred to the Android tablet will be determined by the absolute minimum (non-trivial) time each protocol will take to transfer a file of some size. We will determine the size of this file in preliminary testing.

Since both input and output data will be manipulated during the testing process, communication with the Android tablet will be done over USB using `adb`. This will allow us to call `netcat` and restart our NORM protocol testing application through the `adb` shell, and will also allow us to check the integrity of the received file. This will be described in Subsection 5.2.

Section 3.5: Data Collection

Subsection 3.5.1: Standard Application Test

Using the three server setup described in Subsection 4.1, our routing server will start a set of tests by sending a command to the sending and receiving servers through

SSH, then start the sending and receiving applications of NORM protocol or TCP. Since this command will be sent on the `192.168.3.0/24` subnet, we will not have to worry about the command not being received by the two data transfer servers. Once the data transfer servers have started running the command, the routing server will wait on each data transfer server to finish the file transfer. After the transfer has been completed, the routing server will write the time that the transfer has taken to a file. It will also test whether the transfer has passed or failed, and to what degree the transfer has done so, by using an MD5 hash comparison between the original file and the received file. The router server will ensure that the transfer has taken place a total of 50 times. Afterwards, the routing server will increment the current negative networking parameter and start a new set of tests. This process will continue until each negative networking parameter has reached the maximum possible value at which NORM protocol or TCP can function.

Subsection 3.5.2: Mobile Application Test

The mobile application tests will be run in a similar fashion to the standard application test, with one notable exception being communication with the Android tablet (the receiving server) over `adb`. The network instability from the negative network parameters makes sending a command over SSH potentially impossible, due to the Android tablet has only one NIC. The `adb` utility will substitute for SSH in our mobile application, as `adb` will allow full control over the networking interfaces on the sending server. This also has the added benefit of having no side effects in starting additional sets of tests.

Chapter 4 - Results

Table 4.1: Mobile Application Tests (NORM)

NORM				
Delay		Corrupt		
Time (ms)	Average Time (s)	Bit Error (%)	Average Time (s)	Hash Failures
0	5.51745098039	0	3.95590196078	0
100	9.91262745098	5	5.80001960784	0
200	7.88515686275	10	6.99933333333	0
300	11.9940588235	15	7.88166666667	0
400	18.9297254902	20	9.03368627451	1
500	24.5085294118	25	15.9407647059	3
600	24.5085294118	30	9.84874509804	1
700	33.3969215686	35	11.8245490196	2
800	41.2074509804	40	13.7780196078	1
900	102.226784314	45	14.3498431373	1
1000	50.9480392157	50	12.5831176471	0
		55	17.8075686275	6
		60	21.2394901961	8
		65	22.4758235294	6
		70	28.0779215686	4

Loss

Lost (%)	Average Time (s)
0	6.53033333333
5	6.43411764706
10	6.43652941176
15	4.57978431373
20	4.28117647059
25	4.91407843137
30	6.95341176471
35	4.35778431373
40	6.76731372549
45	7.44543137255
50	9.26296078431
55	14.6744901961
60	25.2856078431
65	46.2160588235

Duplicate

Duplication (%)	Average Time (s)
0	7.31511538462
5	9.28798039216
10	6.01470588235
15	4.46331372549
20	5.01425490196
25	4.48125490196
30	5.76664705882
35	4.62949019608
40	5.20174509804
45	5.57305882353
50	5.96805882353
55	7.07080392157
60	6.67152941176
65	6.36596078431
70	5.93976470588
75	5.41790196078
80	5.93323529412
85	6.09535294118
90	7.64815686275
95	9.77847058824
100	5.62143137255

Table 4.2: Mobile Application Tests (TCP)

TCP

Delay

Time (ms)	Average Time (s)
0	3.6483134123
100	13.5893421632
200	17.7790240966
300	65.9406932547
400	84.1365083594
500	100.389171958
600	107.957106311
700	106.160027692
800	115.746439622
900	129.484409814
1000	146.278960563

Corrupt*

Bit Error (%)	Average Time (s)	Hash Failures
0	18.004144609	0
0.02	3.76858170674	10
0.04	3.92708722903	20
0.06	3.89096484276	19
0.08	5.71938019532	29
0.10	5.68725178333	36
0.12	4.98322460285	34
0.14	4.73774617452	32
0.16	6.78956252795	47
0.18	33.345048313	40
0.20	4.45599959905	38

** Note: the mobile networking TCP tests with corruption added were the only tests that had substantial MD5 hash mismatches at such low parameter values. The number of this failures is out of a total of 50 tests run. This is discussed in Section 4.2.1*

Loss

Lost (%)	Average Time (s)
0	3.64469026052
3	7.99275776056
6	16.877409394
9	28.62879782
12	46.9222646768
15	116.227699477
18	172.141429974
21	216.299349899

Duplicate

Duplication (%)	Average Time (s)	Hash Failures
0	3.31747461741	0
5	64.0667704252	1
10	68.8530153036	1
15	65.5055791964	1
20	4.27197780517	0
25	57.7942578426	1
30	69.8206650935	1
35	5.33594928796	0
40	122.36681294	2
45	72.3756605386	1
50	201.07562536	3
55	204.146972583	3
60	206.713097196	3
65	272.580928321	4
70	136.777654194	3
75	207.030634605	5
80	242.210385125	6
85	136.310462314	3
90	165.782522078	5
95	104.548839982	3
100	135.903081082	4

Table 4.3: Standard Application Tests (NORM)

NORM			
Delay		Corrupt	
Time (ms)	Average Time (s)	Bit Error (%)	Average Time (s)
0	2.47973492813	0	1.88131186485
100	9.36356542507	5	42.0565103245
200	16.5655429268	10	58.3856527471
300	20.3668978739	15	60.0905677605
400	27.4813037157	20	77.6660950374
500	35.4199774122	25	64.9751196432
600	45.477275548	30	58.124481616
700	53.7405391741	35	62.4063548088
800	62.3772495174	40	63.5909676552
900	72.6577617788	45	52.5729706812
1000	81.5713464372	50	46.6915375519
		55	39.9192517812
		60	49.0697722286
		65	59.5309140587
		70	62.6408917236

Loss

Lost (%)	Average Time (s)
0	1.92759618282
5	3.95006384373
10	6.83402519226
15	8.4805047369
20	9.47769023895
25	10.1636055613
30	10.9191163206
35	11.8861831141
40	12.9423074865
45	14.1469590855
50	15.7298204851
55	17.9545792341
60	21.9061140394
65	25.6975946522

Duplicate

Duplication (%)	Average Time (s)
0	1.85281494617
5	1.861946311
10	1.85997519493
15	1.86466278076
20	1.85745548725
25	1.85210840225
30	1.84542104244
35	1.83923261166
40	1.83478843212
45	1.85446674824
50	1.85530232906
55	1.85196208477
60	2.06933405399
65	2.21036903381
70	2.05975035191
75	2.12028933525
80	2.1051688385
85	2.17454503536
90	2.22607143402
95	2.34869624138
100	2.24043930054

Table 4.3: Standard Application Tests (TCP)

TCP			
Delay		Corrupt	
Time (ms)	Average Time (s)	Bit Error (%)	Average Time (s)
0	1.34365602568	0	1.38025881946
100	11.2355766606	1	6.90010064603
200	21.2103670168	2	9.23556391716
300	31.5235369015	3	18.1375432301
400	41.6850223064	4	27.2168324184
500	52.4113567924	5	36.7255697775
600	62.4966660261	6	55.7620031309
700	72.4579406691	7	83.8314852667
800	83.7973110104	8	145.54889338
900	92.5111307859	9	281.087950997
1000	103.393945699	10	678.923252306

Loss

Lost (%)	Average Time (s)
0	1.35586238384
1	6.61872742176
2	15.5671227837
3	26.063038435
4	43.2312340021
5	59.7318134451
6	104.390770512
7	206.487164478
8	213.308822217
9	513.300556979
10	1425.69833303

Duplicate

Duplication (%)	Average Time (s)
0	1.13802670479
5	1.07672141075
10	1.14420938015
15	1.21635595322
20	1.30722589493
25	1.34143207073
30	1.36685935497
35	1.2743807888
40	1.30588029861
45	1.24786633492
50	1.28089592934
55	1.31366980553
60	1.3246570158
65	1.31123177528
70	1.30690385342
75	1.25710924625
80	1.3101513195
85	1.26166550159
90	1.26257130623
95	1.26071363926
100	1.25227894306

Section 4.1: Parameters in Tables

Each parameter chosen in the tests represents some negative value added to the network transmission to make the network less reliable. The delay parameter adds a delay of the specified time to each packet sent in the transmission. The corrupt parameter adds bit errors to the packets in the frame based on the ratio of correct packets to incorrect packets. For example, a corrupt parameter of 0.1% will cause 1 out of 1000 packets to contain a bit error.

The loss parameter has a similar parameter effect as the corrupt parameter, except that instead of having a bit error, the packet will simply be lost. For example, a 0.1% loss parameter value will cause 1 out of 1000 packets to be lost in transmission. Finally, the duplication parameter is specified like the loss and corrupt parameters. If the parameter is given a value of 0.1%, then 1 out of every 1000 packets will be duplicated (resent) in the transmission. Unlike loss or corruption, each packet will be guaranteed to be sent with the duplication parameter added; the protocol being used for the transfer will have to only ignore the duplicate information.

Section 4.2: File Sizes

Due to the different maximum network speeds that the data transfer would take place over, we decided to use a smaller file for the mobile networking tests. The mobile networking tests will be using an 8 megabyte (MB) file and the standard networking tests will be using a 64 MB file. These files were chosen based on the time taken to transfer them in comparison to smaller files being used for the tests. At 8 MB and 64 MB, for mobile and standard networks respectively, the time taken to transfer these files is greater than files of any smaller size. We wanted to transfer a file that was large enough in size to take longer than the minimum, non-trivial amount of time NORM protocol and TCP could take to transfer any file.

Section 4.3: Data Validation

In order to check the integrity of the file, we compared the MD5 hash of the received file with an expected hash from the file on the sending server. If the hash was not an exact match, then the data was unreliable, and the protocol that was used had failed. Generally, the packet corruption and the packet loss tests, given a high loss percentage, were the most susceptible for transfer failure. There were generally few failures in all tests, with the notable exception of the mobile network TCP test with corruption added.

Subsection 4.3.1: Mobile Network TCP Corruption Test:

The mobile networking TCP tests with corruption added to the packets had a near impossible chance to achieve consistent reliable data reception. Original parameters for this test were more similar to those used in the standard network TCP tests with corruption added, but due to the extreme lack of reliable data reception at those parameters, we greatly lowered the parameter values for the mobile networking tests. Theories on the reason for this massive unreliability will be discussed in Chapter 5.

Subsection 4.3.2: Side Effects of MD5 Hash Failures

Some parameters, including the NORM protocol mobile mobile tests with corruption added and the TCP mobile network tests with duplication added, show sets of tests with lower values behaving unlike previous or later sets of tests. We first thought that these values were outliers in our data. However, a closer inspection showed that these very high or very low transfer times were simply how the protocol would behave in a situation where the transfer did not fully complete. TCP often would take a very long time to recognize that the transfer was failed, while NORM protocol usually would have the failure take place very quickly. The effects of these failures are represented very well visually, with steep increases or decreases in the graph.

Section 4.4: Test Parameter Limits

The limits on the tests were determined in preliminary testing by keeping records when each protocol could either no longer complete the transfer, or when the data received in an entire set of tests was wrong. If a protocol would no longer receive correct data, there would be no need to continue testing further. Furthermore, if a connection between sender and receiver could not be established to begin a data transfer, as was the case with high loss percentage cases, then testing would not be able to yield results.

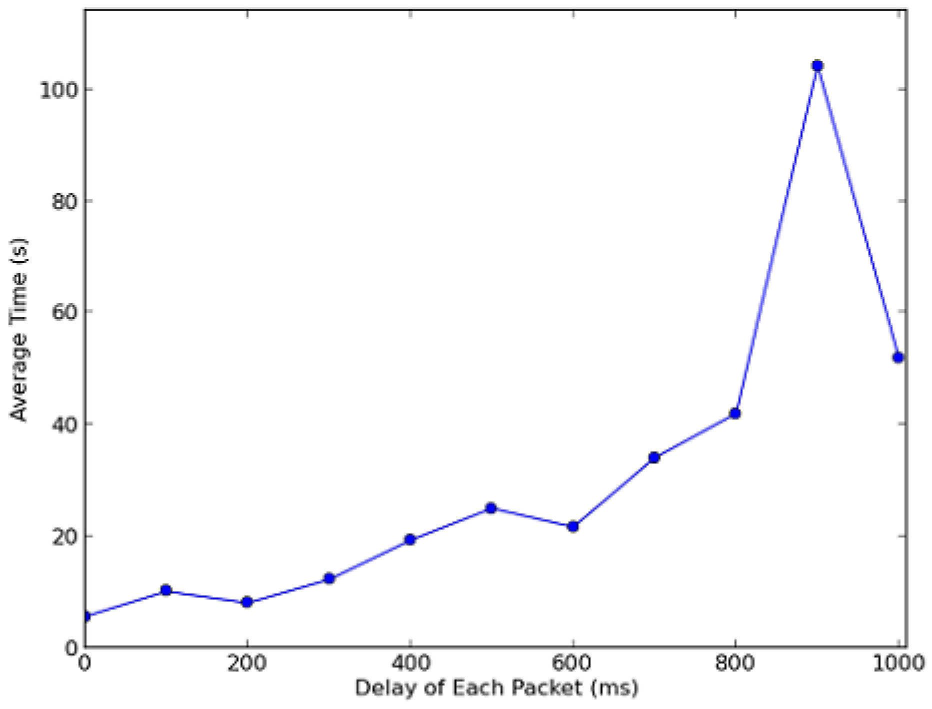

Fig. 4.1 - NORM Protocol Mobile Network Delay Test

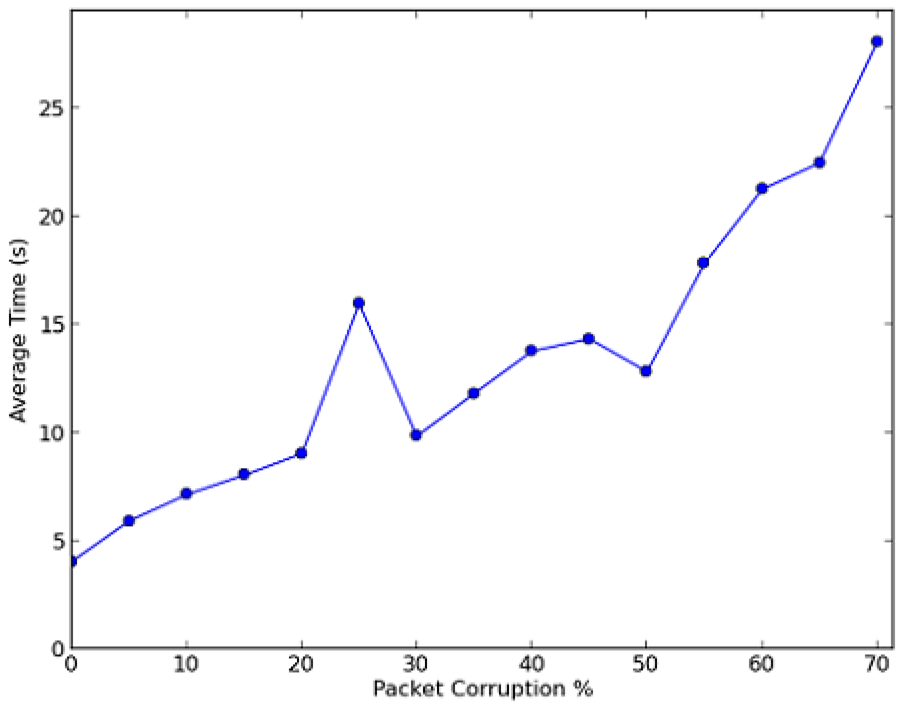

Fig. 4.2 - NORM Protocol Mobile Network Corrupt Test

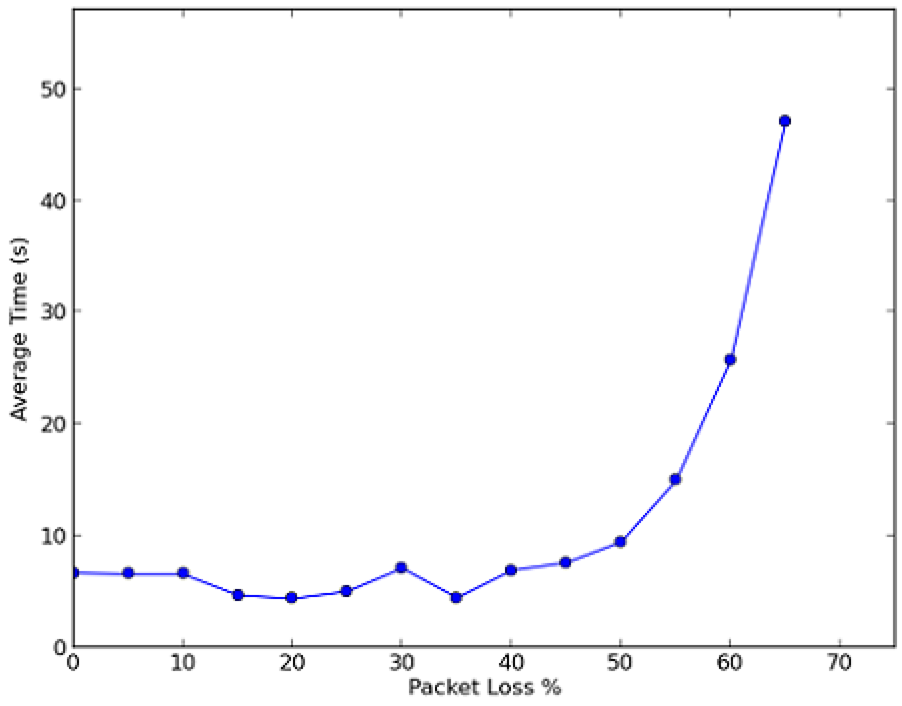

Fig. 4.3 - NORM Protocol Mobile Network Loss Test

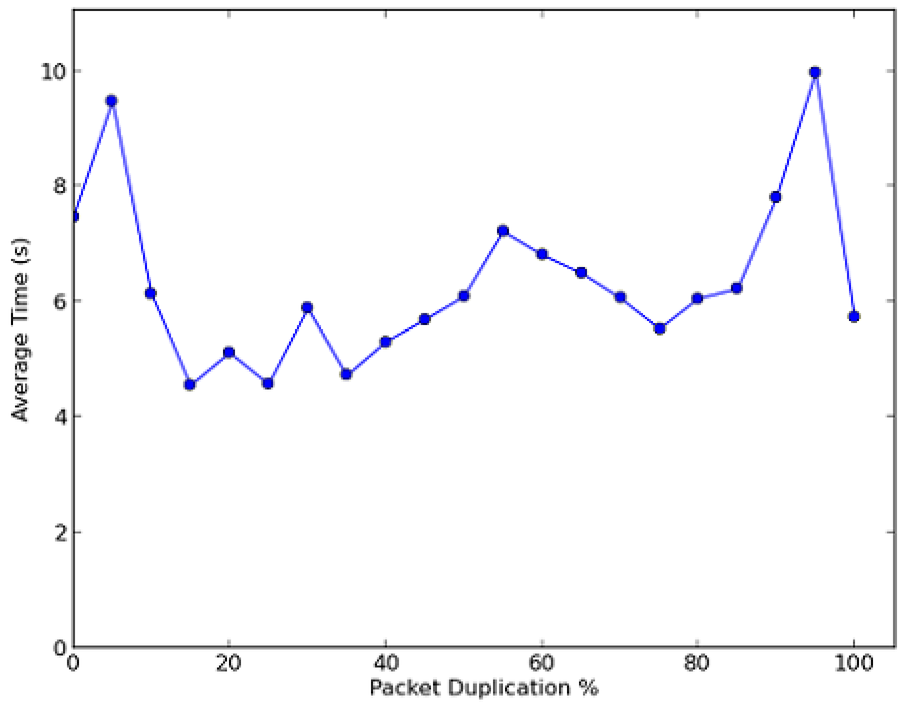

Fig. 4.4 - NORM Protocol Mobile Network Duplication Test

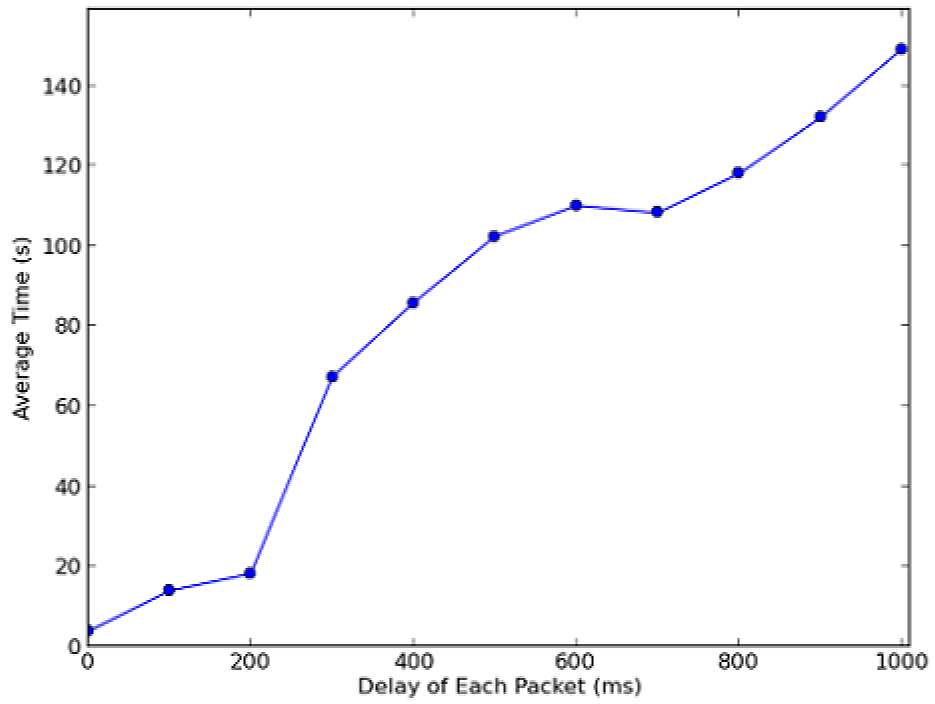

Fig. 4.5 - TCP Mobile Network Delay Test

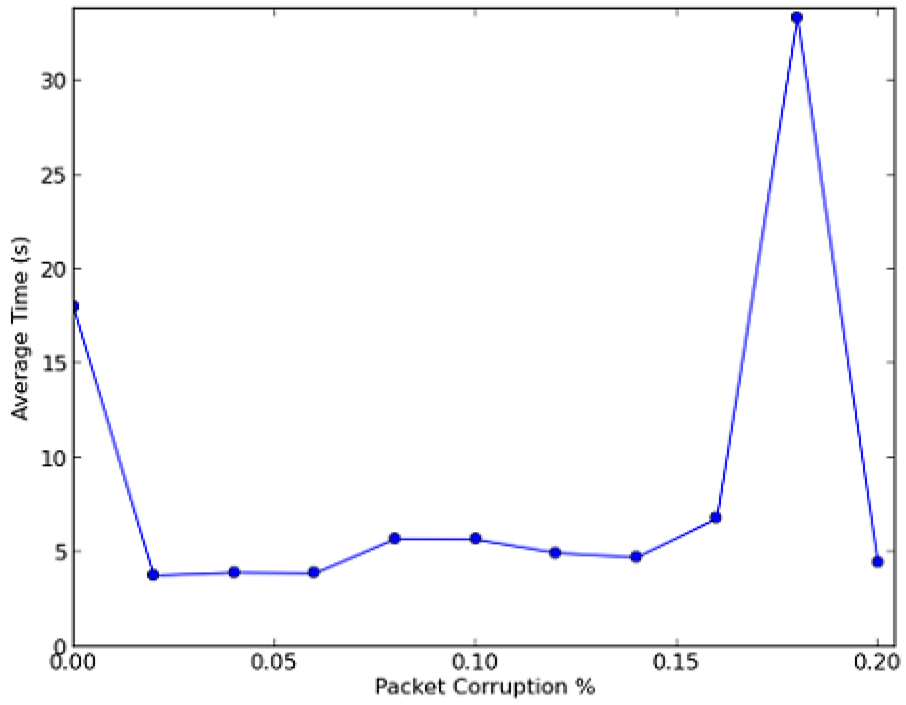

Fig. 4.6 - TCP Mobile Network Corrupt Test

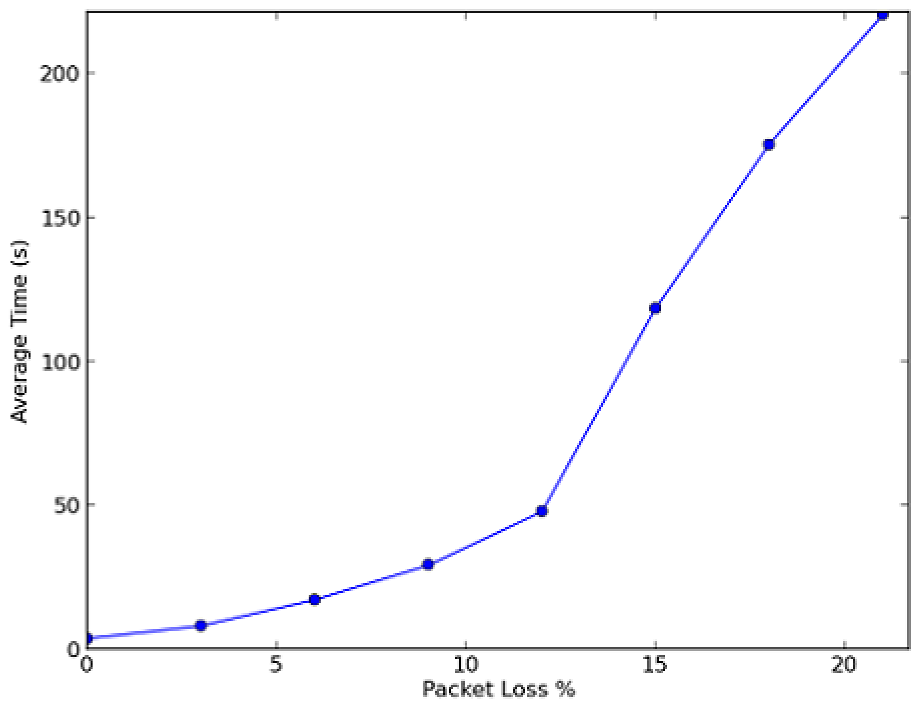

Fig. 4.7 - TCP Mobile Network Loss Test

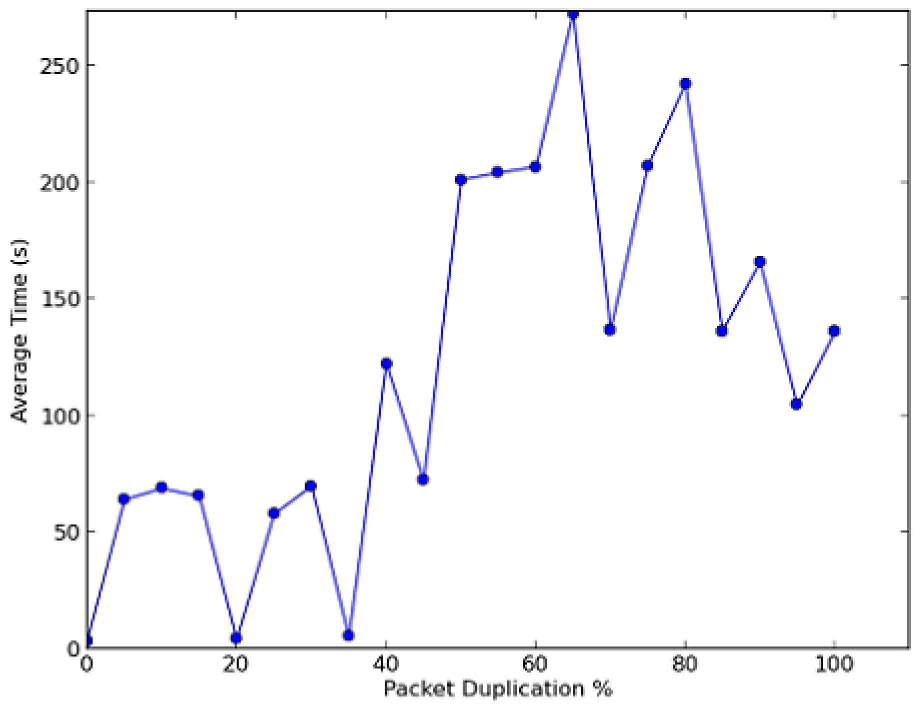

Fig. 4.8 - TCP Mobile Network Duplication Test

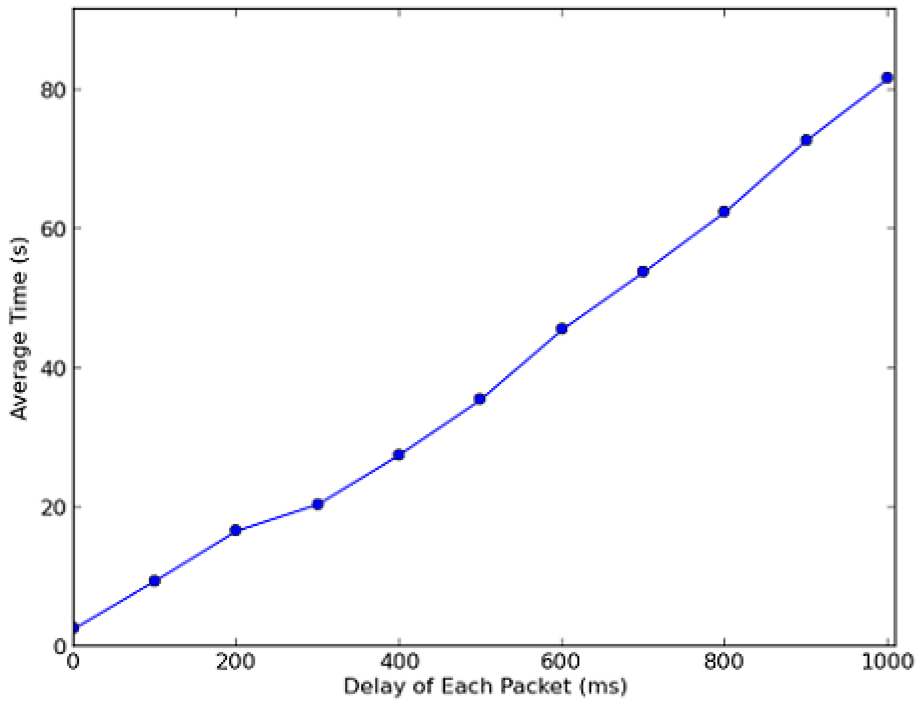

Fig. 4.9 - NORM Protocol Standard Network Delay Test

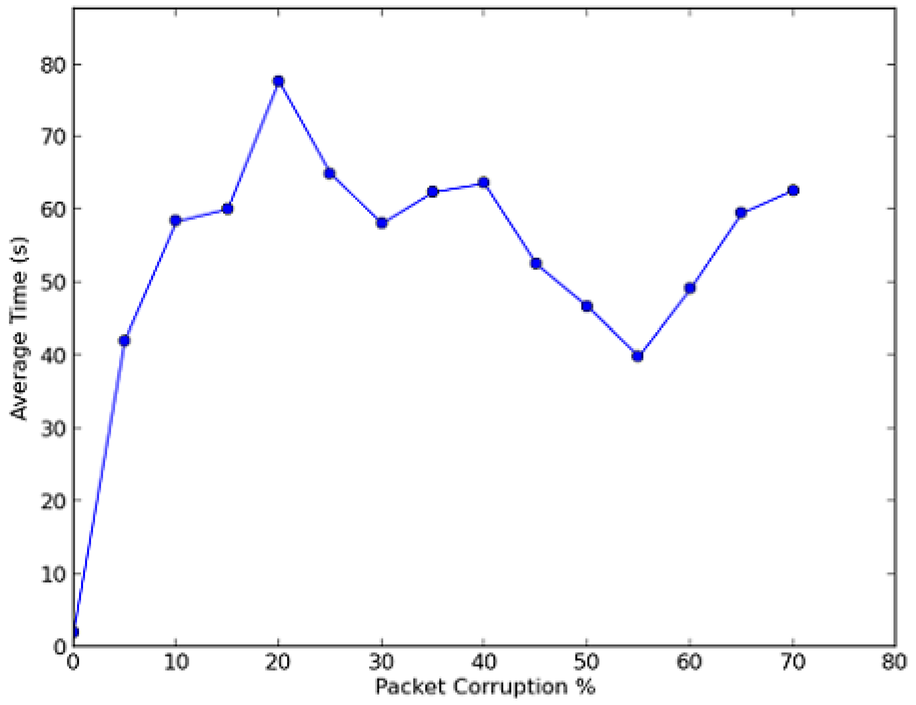

Fig. 4.10 - NORM Protocol Standard Network Corrupt Test

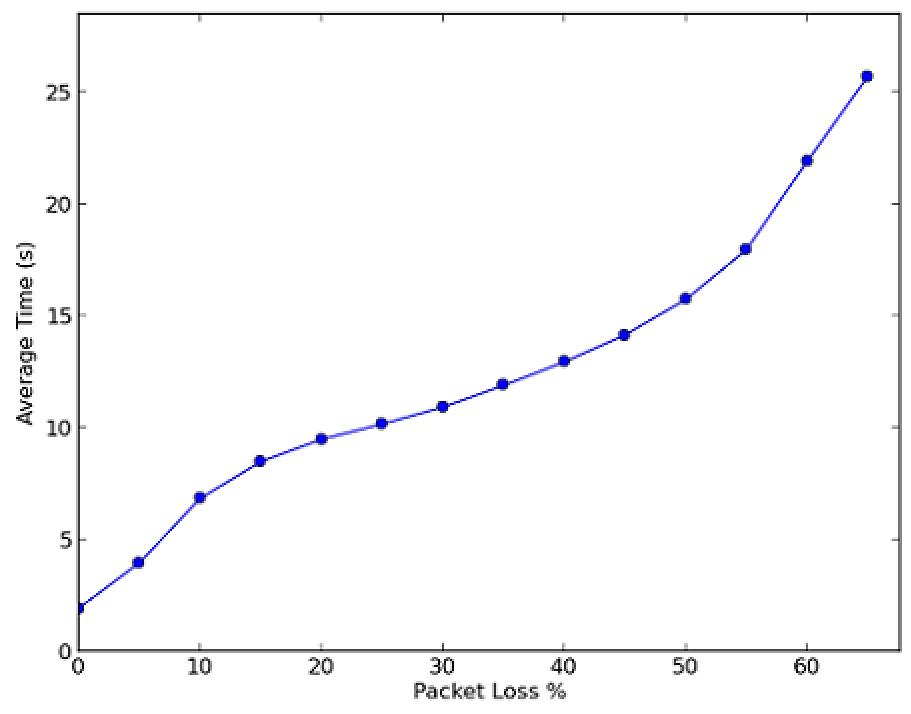

Fig. 4.11 - NORM Protocol Standard Network Loss Test

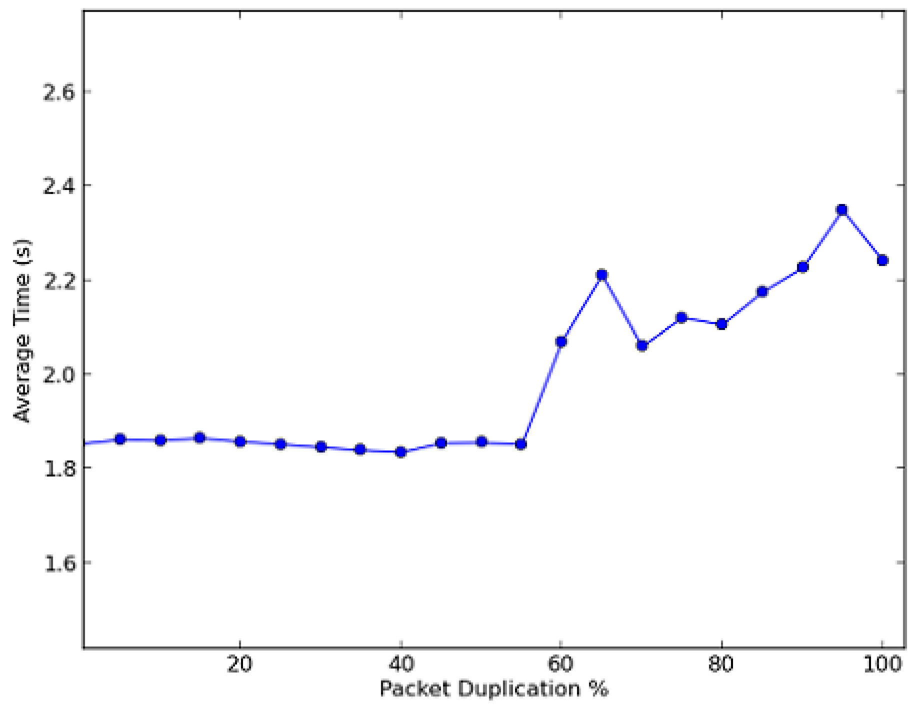

Fig. 4.12 - NORM Protocol Standard Network Duplication Test

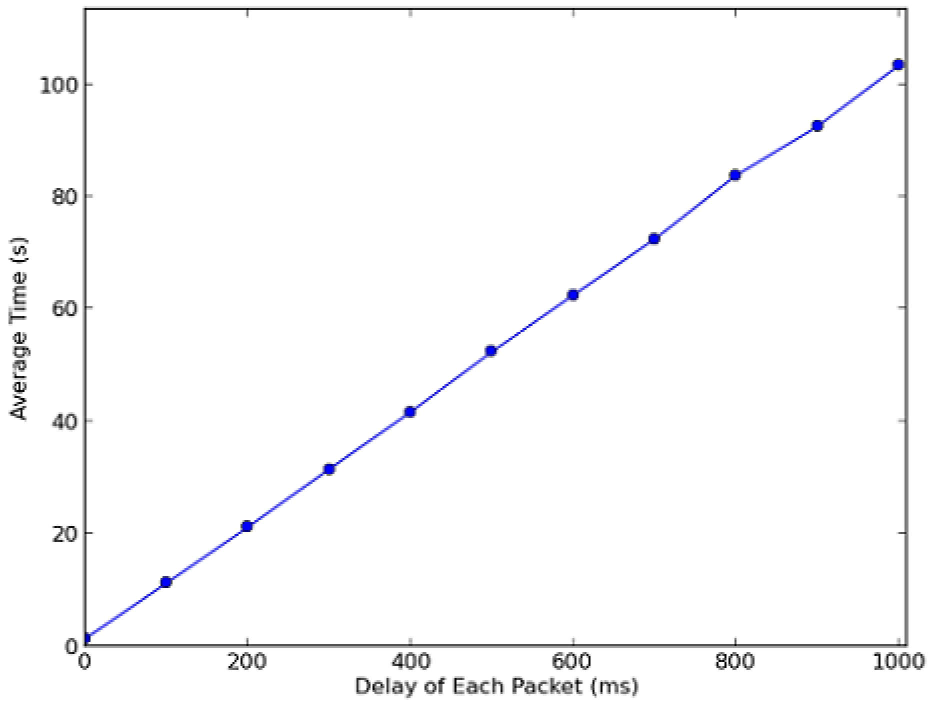

Fig. 4.13 - TCP Standard Network Delay Test

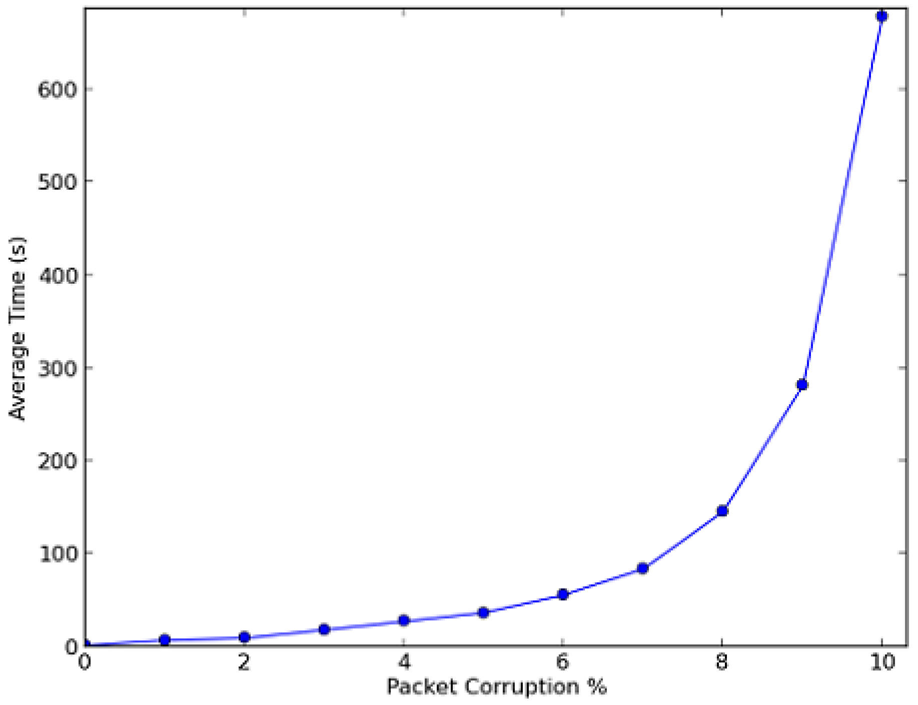

Fig. 4.14 - TCP Standard Network Corrupt Test

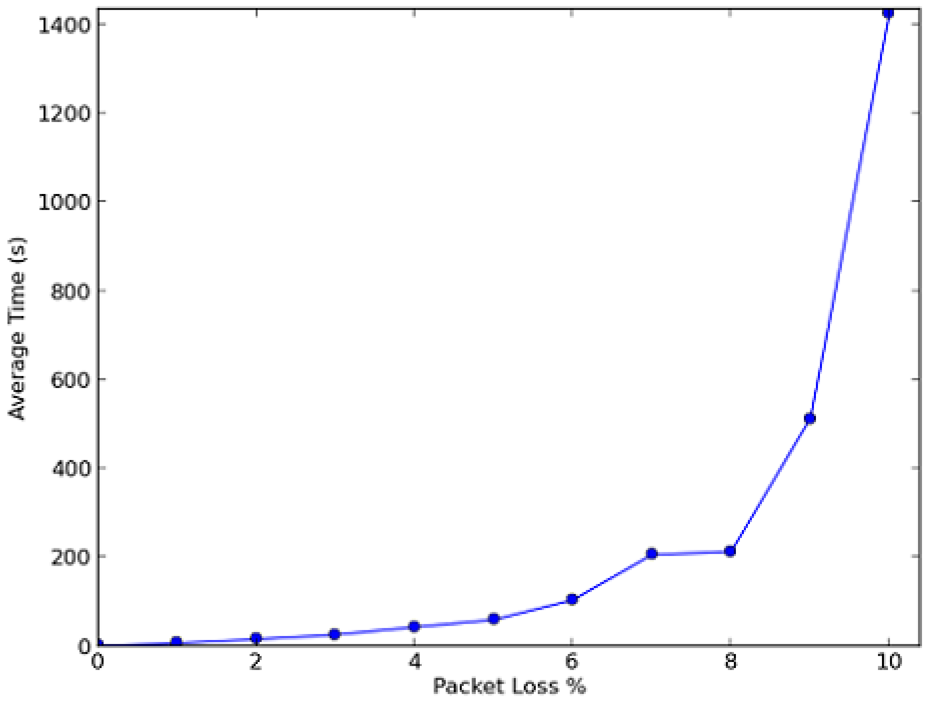

Fig. 4.15- TCP Standard Network Loss Test

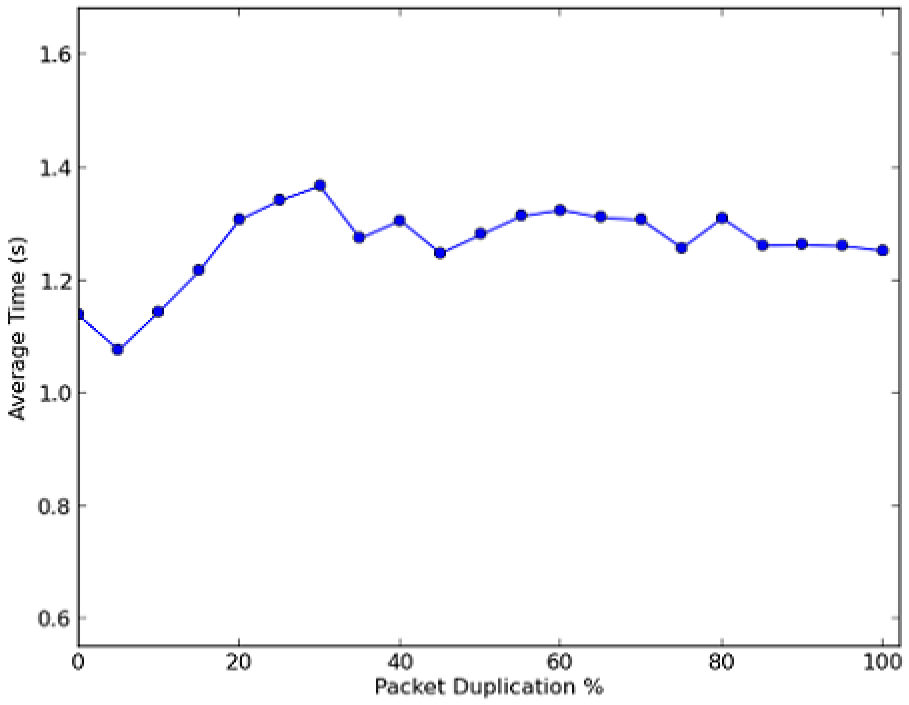

Fig. 4.16 - TCP Standard Network Duplication Test

Chapter 5: Analysis and Evaluation

Section 5.1: Effects of Varying Network Conditions

Subsection 5.1.1: NORM Protocol

The effect of delayed packets in data transmission on NORM protocol is similar across our network types, for the most part. The time taken to transfer a file grew in a linear fashion within the standard networking tests, which is entirely normal. As each packet is delayed for an increasingly longer time, the transfer time will also grow at the same rate. The graph of our standard networking test with delay (Figure 4.9) backs up this statement. The mobile networking test also behaved in a similar fashion, but with oscillation between some data points. This can be attributed to the relative unpredictability of wireless networks in comparison to a standard wired network.

The results of tests (Figures 4.6 and 4.10) with packet corruption show distinct differences between the standard and mobile networks. The standard network tests had a very large jump in transfer time as soon as corruption was induced; however, the time taken in mobile network tests instead gradually increased. Also, while data gathered from the mobile networking test grew in a mostly linear fashion, the data from the standard network tests had no real pattern of growth. In fact, the standard tests show a dip in transfer times as the corruption percentage increases. However, it is important to note that both networks show an increase in transfer time as the limit for our tests is approached. Mobile networks also show a much higher chance of a failed transfer after 50% corruption.

Perhaps at first glance, the duplication graphs of both the standard (Figure 4.12) and mobile (Figure 4.4) network tests show a strange flux in the graph, especially at the end. However, the transfer times show that duplication does not heavily impact NORM protocol on either standard or mobile networks. The range of transfer times over the entire set of tests is very small, less than 1 second. The range of times in mobile networking tests is larger, approximately 5 seconds, but this can still be explained by the relative unpredictability of wireless networks.

The most interesting data in all of the NORM protocol tests is the loss graph (Figure 4.11). The standard network graph shows a moderate, linear rate of data growth; however, the mobile network test shows almost no change between data points until high loss percentages occur. Although a partial explanation for this may be due to the difference in file sizes used in the transfer, this can only apply to low values of loss percentages. One possible theory is that the Android operating system kernel has been somehow optimized for dealing with higher amounts of data loss even when the protocols used in data transfer are not native to Android.

Subsection 5.1.2: TCP

There was a small difference between the standard (Figure 4.9) and mobile networking (Figure 4.5) tests under the effects of packet delay. The standard network data grew perfectly linear, as expected. Conversely, the Android tests show a sharp increase in transfer time, approximately 45 seconds, at the 300 millisecond mark. Furthermore, the Android tests show a small decrease in transfer time at the 700 millisecond mark. The decrease, like with some of the NORM protocol tests, can be traced to the relative instability of wireless networking in comparison to wired networks.

However, the large increase in transfer time cannot be explained by the same reasoning. There must be a more obscure reason to explain this, such as certain Android operating system kernel settings.

As mentioned in Chapter 4, the mobile networking tests showed that TCP could not handle any amount of corruption (Figure 4.6). This is contrary to the results of the standard network tests, where it was possible to have a possible ratio of 10% bad packets to 90% good packets in a frame. With the mobile networking tests, however, having 99.98% good packets was not enough to ensure that the data would always arrive reliably. As noted in Table 4.2, this scenario contained the only instances of MD5 hash matching failure.

TCP behaved very erratically under the effects of packet duplication (Figures 4.8, 4.16). Average times in consecutive sets of tests show very little relationship. This is due to situations where TCP attempted to complete a transfer but eventually could not. Test times in these situations would possibly be longer than 3000 seconds. Removing these large times would produce a graph similar to that of NORM protocol, but since NORM protocol did not have any failures in the mobile network tests with duplication added, we decided to include these large times in the results since they show the possible effects on TCP of duplication in mobile networks.

Despite the failings with packet corruption, tests with packet loss in mobile networks (Figure 4.7) showed an interesting phenomenon, very similar to what happened with NORM protocol. First, the standard networking tests with packet loss added could not function past 10% loss; the mobile networking tests with packet loss could function at 20%. Second, although data from the mobile networking tests initially grow faster compared to standard networking tests data, the standard networking tests show a massive spike in transfer times at its maximum limit of 10% loss. Mobile networking tests show a much less severe jump in transfer times leading to the 20% loss limit. The similarity of results between NORM protocol and TCP mobile networking tests, as compared to their respective standard networking tests, implies that the Android operating system may be optimizing network transfers that have high loss.

Section 5.2: Overall Protocol Comparison

TCP had a substantial margin of advantage when networking conditions were optimal, beating out NORM protocol in all of our tests with no negative effects added. TCP also performed strongly against NORM protocol in the standard networking tests with duplication added. However, NORM protocol was much more effective at completing data transfers in networks with loss or corruption, regardless of whether the network was a standard wired network or a mobile wireless network. NORM protocol performed especially well in comparison to TCP when packet loss was greater than 5%. NORM protocol also had a much larger limit of loss than TCP, with tests on both the mobile and standard networks showing NORM protocol being able to complete transfers even when packet loss reached 65%.

NORM protocol also has an advantage over TCP in networks with delay of 200 milliseconds or more. This is most likely due to NORM protocol, like its name states, using negative acknowledgements (NACKs) to respond to lack of data. Since NORM protocol does not acknowledge (ACK) each time as it receives data, there is significantly less communication that has to be resolved during the transfer.

Section 5.3: Practical Applications of NORM Protocol

NORM protocol seems to be an excellent fit for the world of mobile networking. If a mobile device is in an area where mobile coverage is weak and data is lost in transmission due to low signal, applications using NORM protocol for data transmission will be minimally affected in comparison to applications using TCP for data transmission. Also, as mobile network speeds have increased, applications that stream data, such as VoIP or video streaming services, are in very high demand. However, in scenarios with high loss or delay, these applications must sacrifice quality to ensure that the data stream will continue. NORM protocol can help these applications continue to have a high quality stream even on networks with high loss or high delay. An example of using NORM protocol for video on a high-loss network can be found at http://cs.itd.nrl.navy.mil/work/noviss/demo.php.

Section 5.4: Additional Research Possibilities

One possible area to extend from this research would be using NORM protocol and TCP for streaming live data, such as music or video. Although we will not be able to record quantitative measurements, we will be able to add other protocols that can be used for live streaming, such as Real Time Messaging Protocol (RTMP) or coded TCP (Kim 2012). Since live video feeds are in very high demand, determining an optimal protocol, such as one with low overhead or easy compression of video data, for use in streaming video would be a practical topic. This topic could also extend into mobile devices and networks, such as battery consumption by various networking protocols used in streaming video and if a live video stream causes poor network performance in other applications.

References

Acharya, S., Franklin, M. and S. Zdonik, "Dissemination - Based Data Delivery Using Broadcast Disks", IEEE Personal Communications, pp.50-60, Dec 1995.

Adamson, B., Bormann, C., Handley, M., Macker, J., "NACK-Oriented Reliable Multicast (NORM) Transport Protocol," Internet Engineering Task Force (IETF) RFC 5740, November 2009.

Braden, R. "RFC 1122." Requirements for Internet Hosts—Communication Layers (1989).

Kim, MinJi, et al. "Network Coded TCP (CTCP)." (2012).

Luby, M., Vicisano, L., Gemmell, J., Rizzo, L., Handley, M., and J. Crowcroft, "The Use of Forward Error Correction (FEC) in Reliable Multicast", RFC 3453, December 2002

Mankin, A., Romanov, A., Bradner, S., and V. Paxson, "IETF Criteria for Evaluating Reliable Multicast Transport and Application Protocols", June 1998

Ravindranath, Lenin, et al. "Improving wireless network performance using sensor hints." USENIX NSDI. 2011.

Sat, Batu, and Benjamin W. Wah. "Analysis and evaluation of the Skype and Google-Talk VoIP systems." Multimedia and Expo, 2006 IEEE International Conference on. IEEE, 2006.

Vixie, Paul. "Extension mechanisms for DNS (EDNS0)." (1999).

Whetten, B., Vicisano, L., Kermode, R., Handley, M., Floyd, S., and M. Luby, "Reliable Multicast Transport Building Blocks for One-to-Many Bulk-Data Transfer", RFC 3048, January 2001