Networks — CSCI 125, Fall 2014
Instructors: Goodney
Lab 3
Due: 2:00PM on Thursday, October 9

Name: Bruce Yan (byan@hmc.edu), Angela Zhou (azhou@hmc.edu)
DETER ID: hmc125ah, hmc125-aj

Problem 1 - Group Project: Fast, Reliable File Transfer:
`scp`. The 'secure copy program' `scp` is a standard tool on modern UNIX-like machines. It is used to copy files between machines, securely and reliably. However, as we will see, it does not always provide good throughput.

- We created an experiment in DETER with two computers together with a 100Mb/s link and 50ms initial delay. Below is the .ns file we used.

```
# lab3-part1.ns

# This is a simple ns script. Comments start with #.
set ns [new Simulator]
source tb_compat.tcl

# Define the 4 nodes in the topology
set nodeA [$ns node]
set nodeB [$ns node]

# Define the link and the LAN that connect the nodes
# This sets the link speed between nodeA and nodeB
# There is a 25ms round-trip delay
set link0 [$ns duplex-link $nodeB $nodeA 100Mb 50ms DropTail]

# Set the OS to Ubuntu1204-64-STD (modern)
tb-set-node-os $nodeA FBSD9-64-STD
tb-set-node-os $nodeB FBSD9-64-STD

# Enable routing (static) between all the nodes
$ns rtproto Static

# Instruct the simulator to start
# Go!
$ns run
```

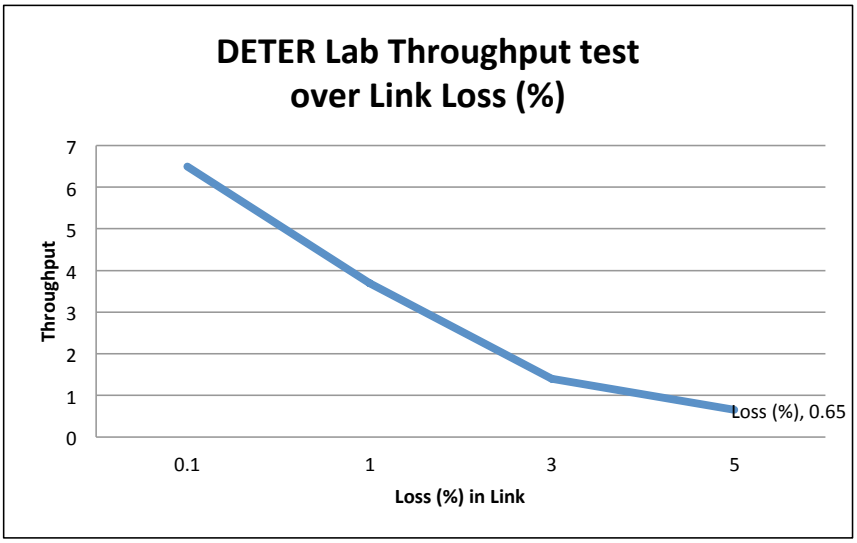
- At the beginning, we did a `ping` between the two nodes and we received the following delay:

```
64 bytes from 10.1.1.2: icmp_seq=27 ttl=64 time=100.386 ms
64 bytes from 10.1.1.2: icmp_seq=28 ttl=64 time=100.420 ms
64 bytes from 10.1.1.2: icmp_seq=29 ttl=64 time=100.324 ms
64 bytes from 10.1.1.2: icmp_seq=30 ttl=64 time=100.365 ms
```

- Once the experiment started, we used the website to set the delay on the link to zero. The RTT delay between the nodes are now:

```
64 bytes from 10.1.1.2: icmp_seq=31 ttl=64 time=0.324 ms
64 bytes from 10.1.1.2: icmp_seq=32 ttl=64 time=0.358 ms
64 bytes from 10.1.1.2: icmp_seq=33 ttl=64 time=0.387 ms
64 bytes from 10.1.1.2: icmp_seq=34 ttl=64 time=0.302 ms
```

- The average delay appears to be 0.3345 ms. Using `iperf -c nodeb -u -p 5001 -b 200M -i 10 -t 60` to test the bandwidth, we have an average of 95.6 Mbits/sec over 6 runs. The Bandwidth delay product is 3.19 Kbits.
- Explain briefly what the above commands do. `dd if=/dev/urandom of=data.bin bs=1m count=200`. `dd` is standard linux utility that copies information from stdin to stdout. Here, `if` is where we are reading the `/dev/urandom` file from and we are reading it out to `of` file called `data.bin`. `bs`, which is the block size of both input and output files, is set to 1 megabyte. `count` means copy only 200 input blocks.
- We had to change the ownership on both sides by using the following command: `sudo chown hmc125ah:HMCCS125 /mnt/data.bin` and `sudo chown hmc125ah:HMCCS125 /mnt`. We used the command `sudo scp data.bin hmc125ah@nodea:/mnt` to copy from nodeB to nodeA.
- What transfer throughput do you get? Based on 6 trials, we get a transfer throughput of 11.1MB/s using `scp`.
- If we increased the delay from 0ms to 25ms on DETER website, we saw an immediate increase in transfer time. The average throughput dropped to 900KB/s. Increasing the delay from 25ms to 75ms on DETER website, the average throughput dropped to be about 300KB/s. Further increasing the delay from 75ms to 125ms, the throughput dropped to be about 186KB/s. `scp` transfer performance seems to degrade quite rapidly. We tried a variety of `scp` parameters and they did not seem to have any effect.
- It does appear that `scp` has some limitations when transferring file over ssh. We are guessing that the TCP window size cannot be altered.
- We graphed the result of the throughput transfer vs loss, which can be seen below. It does not seem too extreme since TCP has to check every packet every time it transfers something. We believe the reason this happens is because every time data is filled in the TCP window, TCP has the property of auto-checking the integrity of all the packets before sending it off. If there's significant loss, then the transfer speed will be severely impacted.



- Per Verizon Enterprise Solutions (from Google), the average round-trip transatlantic latency between NY and London is approximately 90ms. It is also stated that roundtrip latency within North America is 45ms. Let's just add 30ms of latency for the distance between London and Switzerland. This total roundtrip is: 165ms.

  We would guess the CERN physicist will NOT be asking us for data considering the huge latency. We could potentially help him, but it would just be slow, considering our CERN physicist friend may need a lot of data. If we were to help him, we may need another method to transfer files, such as using distributed file systems or Cloud solutions.

Problem 2 - File Transfer Utility:
As we've learned above, TCP, while totally reliable and robust, doesn't always give us good throughput. In this section you'll design a IP based file-transfer utility. The design and implementation of the utility is up to your group, however it must full-fill only three requirements: it must use IP (so it can be routed), it must transfer the file reliably (with no errors) and it must be implemented with a command-line interface similar to `scp`.

The link speed between the sender and receiver must be **100Mbps** and the test file size must be at least **1GBytes**. You should emulate the delay and the loss rate of the link using the delay node. You should test your system under various different conditions. However two settings that you must expose your system for the assignment are:

- The Delay (RTT) of 10ms with the Loss rate of 1%
- The Delay (RTT) of 200ms with the Loss rate of 20%

Describe, in detail, the concept(s) behind your file transfer utility, results, and the analysis in the document that must be submitted on October 9th by 2pm. Also, on October 9th and 14th we will have presentations from each group. The presentations will give details on how you solved the problems, problems you encountered and the results you were able to obtain. Some hints:

- There are several closed and open source projects out there that do this. They typically use UDP at their base. You may use them as inspiration, however the end work must be your own. You must use UDP and only UDP for the transport of the file data. However, you may use TCP for control or metadata, but all of the file data, including retransmissions, must use UDP.
- You can implement the program in any language you'd like (C, C++, Java, Python, Ruby, etc.) as long as it works on the DETER nodes, and your submission comes with clear, concise instructions on how to build and test your program.
- Think about why `scp` has issues. Identify these weaknesses as inspiration for solving the problem. Think about selective re-transmission, parallel flows, forward error correction...
- Learn how to use the following UNIX tools, they are your friends: `tcpdump, tcpreplay, nc (aka netcat), nmap, netstat, iperf`
- There will be a prize for the team that achieves the highest throughput on the **200ms, 20%** test case!
- You may need to learn about network programming on UNIX using sockets, and/or `libpcap`. See:
  http://beej.us/guide/bgnet/
  http://www.prasannatech.net/2008/07/socket-programming-tutorial.html
  http://gnosis.cx/publish/programming/sockets2.html
- To measure the throughput achieved, the `time` utility will be used. Familiarize yourself with this utility and understand how to use it to measure the total throughput achieved by your FTP utility.

The code for our Python implementation can be found below. We have listed our `server.py`, `client.py`, and associated helper functions such as `convertIndex.py`, `TCPhelpers_client.py`

1 client.py

```
import socket
import TCPhelpers_client
import sys
import os
import convertIndex
import ast
'''
    The client send a file to the server.
'''
BLOCK_SIZE = 512
METADATA_SIZE = 5
INDEX_SIZE = 8
buf = BLOCK_SIZE - INDEX_SIZE

# system arguments: python programName serverHostName portNumber fileName
serverHostName = sys.argv[1]
portNumber = int(sys.argv[2])
fileName = sys.argv[3]

# Initialize connection with host using TCP
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect((serverHostName, portNumber))
metaData = TCPhelpers_client.generateMetaData(fileName, buf)

######################################################################
# TCP Metadata Transfer
######################################################################
# +++ Confirmation 1: established TCP connection. +++ #
message = client_socket.recv(BLOCK_SIZE)
print message
if 'initialized' not in message:
    print 'Having error initializing connection.'

# +++ Confirmation 2: complete metadata transfer. +++ #
needMetadata = 1

while needMetadata:
    client_socket.send(str(metaData))
    needMetadata = int(client_socket.recv(BLOCK_SIZE))
    print 'needMetadata is ', needMetadata
######################################################################
```

```
#####################################################################
# UDP file transfer
#####################################################################
udp_client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
# udp_client_socket.connect(("localhost", 5006))
# Comment 1: We don't want the above line because unlike TCP, which needs to maintain a
    solid connection, we don't need to do that with UDP. UDP is meant to be used as
    opening a connection and sending stuff over, then closing the connection when done.
    Don't need to keep it open
# to create a binary file use: dd if=/dev/random of=file2.bin bs=1m count=200

host = sys.argv[1]
port = 5006
address = (host,port)
# buf = 512


filename = sys.argv[3]

udp_client_socket.sendto(filename, address)

# Open a file with the name of the transmitted file to read
f = open(filename, 'r') # read binary

# Read an initial amount data to buffer
data = f.read(buf)

# index calculations - this is tacked to the front of data
size = os.path.getsize(filename)
numBlocks = size / BLOCK_SIZE
if size % BLOCK_SIZE != 0:
    numBlocks += 1

# TODO: Attach a number to each
# instead of sending 512, we send 508 so 4 bytes will be the index #
index = 0

while(data):
    index += 1
    if(udp_client_socket.sendto(convertIndex.convertIndexToStr(index, INDEX_SIZE) +
        data,address)):
        #print "Sending:",filename, "... index:", convertIndex.convertIndexToStr(index,
            INDEX_SIZE)
        data = f.read(buf)


#####################################################################

#####################################################################
# Retransfering lost packets
```

```
######################################################################


waitingToComplete = 1
listString = client_socket.recv(BLOCK_SIZE)
print 'Before waitingToComplete : ', listString
while waitingToComplete:

    # print 'received list: ', listString
    # Test if the transfer is actually complete
    if 'waitingToComplete' in listString:
        print 'Transfer complete, no more lost packets.'
        waitingToComplete = 0
        break;

    # Parse list of lost packets being transfered
    # listString = listString.replace("\'", "")
    # listString = listString[2:-1]
    # lostPackets = listString.split(',')
    print listString
    lostPackets = ast.literal_eval(listString)
    # if '' in lostPackets:
    #     lostPackets.remove('')
    if lostPackets:
        print 'Lost packets: ', lostPackets
        for index in lostPackets:
            # index = index.strip(' ')
            # index = int(index)
            f.seek(buf * (index-1))
            data = f.read(buf)
            if (udp_client_socket.sendto(convertIndex.convertIndexToStr(index, INDEX_SIZE)
                + data,address)):
                print "Resending:",filename, "... index:",
                    convertIndex.convertIndexToStr(index, INDEX_SIZE)
    listString = client_socket.recv(BLOCK_SIZE)


######################################################################



######################################################################
# Cleaning up
######################################################################

udp_client_socket.close()
f.close()

client_socket.close()
print "Terminating."
```

2 server.py

```
import socket
import sys
import os
import ast
'''
       The serve receives a file from a client.
'''
BLOCK_SIZE = 512
METADATA_SIZE = 5
INDEX_SIZE = 8

portNumber = sys.argv[1]
tcp_server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
tcp_server_socket.bind(("", int(portNumber)))
tcp_server_socket.listen(5)

print "TCPServer Waiting for client on port ", portNumber

packets = []
filename = None
filesize = None
checksum = None
numPackets = None
packetSize = None


######################################################################
# TCP Metadata Transfer
######################################################################
tcp_client_socket, address = tcp_server_socket.accept()
print "I got a connection from ", address
# +++ Confirmation 1: established TCP connection. +++ #
tcp_client_socket.send("Connection initialized.")

needsMetaData = 1
array = []
while needsMetaData:
       # Hacky processing :)
       metadata = tcp_client_socket.recv(BLOCK_SIZE)
       metadata = metadata.replace("\'", "")
       metadata = metadata[1:-1]
       array = metadata.split(', ')
       needsMetaData = 0
       # +++ Confirmation 2: complete metadata transfer. +++ #
       tcp_client_socket.send(str(needsMetaData))

print 'Metadata : ', array
```

```
for data in array:
        data = data.split(':')
        if 'file size' in data[0]:
                filesize = int(data[1])
                print 'filesize: ', filesize
        if 'name' in data[0]:
                filename = data[1]
                print 'filename: ', filename
        if 'checksum' in data[0]:
                checksum = data[1]
                print 'checksum: ', checksum
        if 'number of blocks' in data[0]:
                numPackets = int(data[1])
                print 'numPackets: ', numPackets
        if 'block size' in data[0]:
                packetSize = int(data[1])
                print 'packetSize: ',packetSize

# Confirmation for client that the metadata transfer is complete
# tcp_client_socket.send('0')
# tcp_server_socket.close()
print 'Done with metadata.'
#########################################################################


#########################################################################
# UDP file transfer
#########################################################################
# Variables
host = "0.0.0.0"
port = 5006
# buf = 512 # adjustable, try higher
# BLOCK_SIZE = 512
buf = BLOCK_SIZE - INDEX_SIZE

# Connect and bind
udp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
udp_socket.bind((host, port))

# Address here is a pair "(host, port), which is used for the AF_INET address family"
address = (host,port)

# Get the filename from buffer
data,address = udp_socket.recvfrom(BLOCK_SIZE) # Tuple (name, address)
filename = data.strip()
print "Receive file start:",filename


# Open a file with the name of the transmitted file to write
```

```
# f = open(filename, 'wb') # 'wb' is for writing binary files
f = open(filename, 'w')
# index calculations - this is tacked to the front of data
size = os.path.getsize(filename)
numBlocks = size / BLOCK_SIZE
if size % BLOCK_SIZE != 0:
	numBlocks += 1

# instead of sending 512, we send 408 so 4 bytes will be the index #
# we start at -1 because the first block of the file has the file name in it
index = -1
lostPackets = []
successPackets = []
# Get the rest of the files
error = 0
resendPackets = []
try:
	while data:
		index += 1
		if (index % 10000 == 0):
			print "The index is:", index

		# Get the current index from the first few characters in data block
		current_index = data[0:INDEX_SIZE]
		if (index > 0):
			#print "The data block index is:", current_index
			successPackets.append(int(current_index))


		# remove the filename from the header

		# if (index == 0):
		data = data.replace(filename, "")

		# replace the header of the blocks.
		data = data.replace(current_index,"")
		# print data
		if (index > 0):
			current_index = int(current_index)
			f.seek(buf*(current_index-1))
			f.write(data)
			f.flush()
		udp_socket.settimeout(50) # Round-trip time

		if index > 0 and int(current_index) >= numPackets:
			break;

		data,address = udp_socket.recvfrom(BLOCK_SIZE)
		# print 'numPackets',numPackets
```

```
        print 'Done transfering data through udp first.'
except socket.timeout:
        print 'time out index: ', index
        print "File download complete! "

for x in xrange(1, numPackets):
        if x not in successPackets:
                resendPackets.append(x)
                if (x % 10000 == 0):
                        print 'xrange : ', x
print 'resend the following packets: ', resendPackets

#resendPackets = [123,125,69]
try:
        while resendPackets:
                print 'lost packets: ', resendPackets
                windowSize = 50
                arrayChunk = []
                if len(resendPackets) < windowSize:
                        arrayChunk = resendPackets
                else:
                        arrayChunk = resendPackets[:windowSize]
                # print str(arrayChunk)
                tcp_client_socket.send(str(arrayChunk))
                data,address = udp_socket.recvfrom(BLOCK_SIZE)

                while data:
                        # if filename in data:
                        #       continue
                        print 'arrayChunk: ', arrayChunk
                        current_index = data[:INDEX_SIZE]
                        index = int(current_index)
                        if index in resendPackets:
                                print "The data block index is:", current_index
                                data = data.replace(current_index,"")
                                resendPackets.remove(index)
                                if index in arrayChunk:
                                        arrayChunk.remove(index)
                                print 'lost packet after update :', resendPackets

                                # Go seek the file at the appropriate place
                                # and write data to the file
                                f.seek((index-1) * buf)
                                f.write(data)
                        # if len(resendPackets) == 0:
                        #       print 'waitingToComplete = 0'
                        #       tcp_client_socket.send('waitingToComplete = 0')
                        #       break;
                        if not arrayChunk:
                                break;
```

```
                    data,address = udp_socket.recvfrom(BLOCK_SIZE)
except socket.timeout:
        print 'time out with resendPackets, ', resendPackets
        print 'Done making shit up.'



udp_socket.close()

print "( " ,address[0], " " , address[1] , " ) received: ", filename
print "missing packets: ", resendPackets

f.close()

print "The file we have written is: ", filename
# Change file name
try:
        tcp_client_socket.send('waitingToComplete = 0')
except socket.error:
        tcp_client_socket.close()

tcp_server_socket.close()
```

3 convertIndex.py

[language=Python]

```
import sys

# index must be integer
def convertIndexToStr( index, bytes ):
    indexStr = str(index)
    numDigit = len(indexStr)
    if numDigit > bytes:
        return '-1'
    else:
        filler = '0' * (bytes - numDigit)
        return filler + indexStr

# print 'convertIndexToStr( 500, 8) : ', convertIndexToStr( 500, 8)
# print 'convertIndexToStr( 10, 1) : ', convertIndexToStr( 10, 1)
```

4 TCPhelpers_client.py

[language=Python]

```
import os
import hashlib

def generateMetaData(inputfile, blockSize):
        results = []
        size = os.path.getsize(inputfile)
        numBlocks = size / blockSize
        if size % blockSize != 0:
                numBlocks += 1
        totalCheckSum = md5_for_file(inputfile, blockSize)

        results.append('name:'+inputfile)
        results.append('file size:'+str(size))
        results.append('number of blocks:'+str(numBlocks))
        results.append('block size:'+str(blockSize))
        results.append('checksum:'+totalCheckSum)
        return results


def md5_for_file(filename, block_size):
        f = open(filename, 'r')
        md5 = hashlib.md5()
        data = f.read(block_size)
        while data:
                md5.update(data)
                data = f.read(block_size)
        f.close()
        return md5.hexdigest()

# filename = raw_input("file name: ")
# result = generateMetaData(filename, 512)
# print result
# print result[1][0], result[1][1]
```