

1)

- 1) The IP address used by the source is 209.87.56.144 and the source port number is 52724.
- 2) The IP address gaia.cs.umass.edu is 128.119.245.12 and it is sending and receiving TCP segments from port number 80.
- 3) The IP address used by the source is 209.87.56.144 and the source port number is 52724

We will refer to the trace file tcp-ethereal-trace for the rest of the questions.

Figure 1:

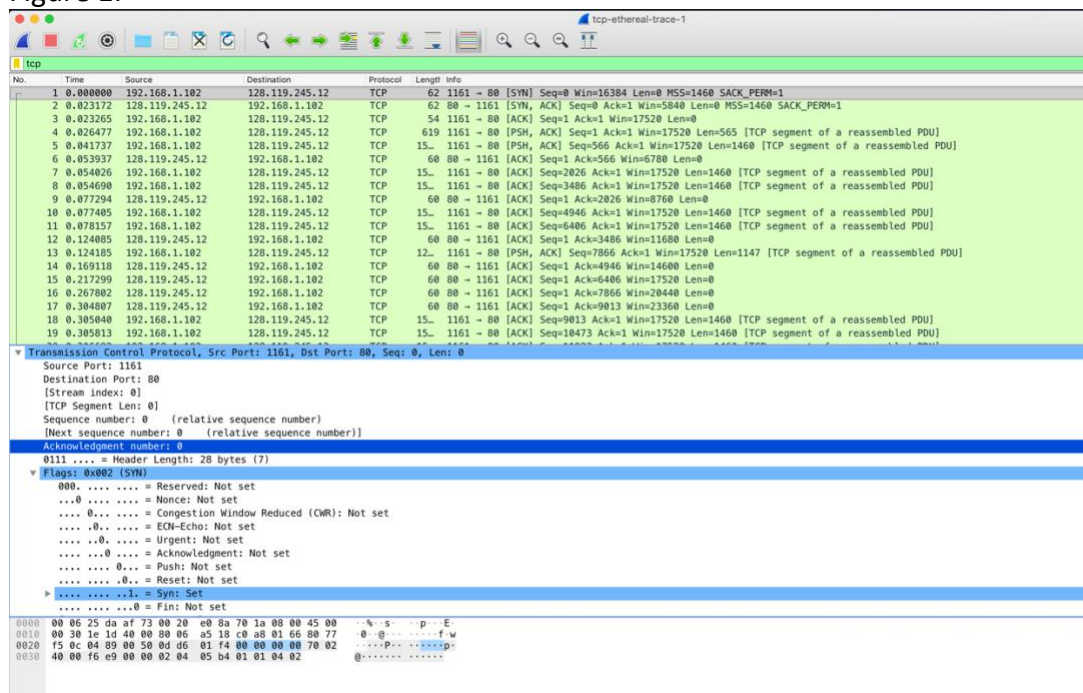
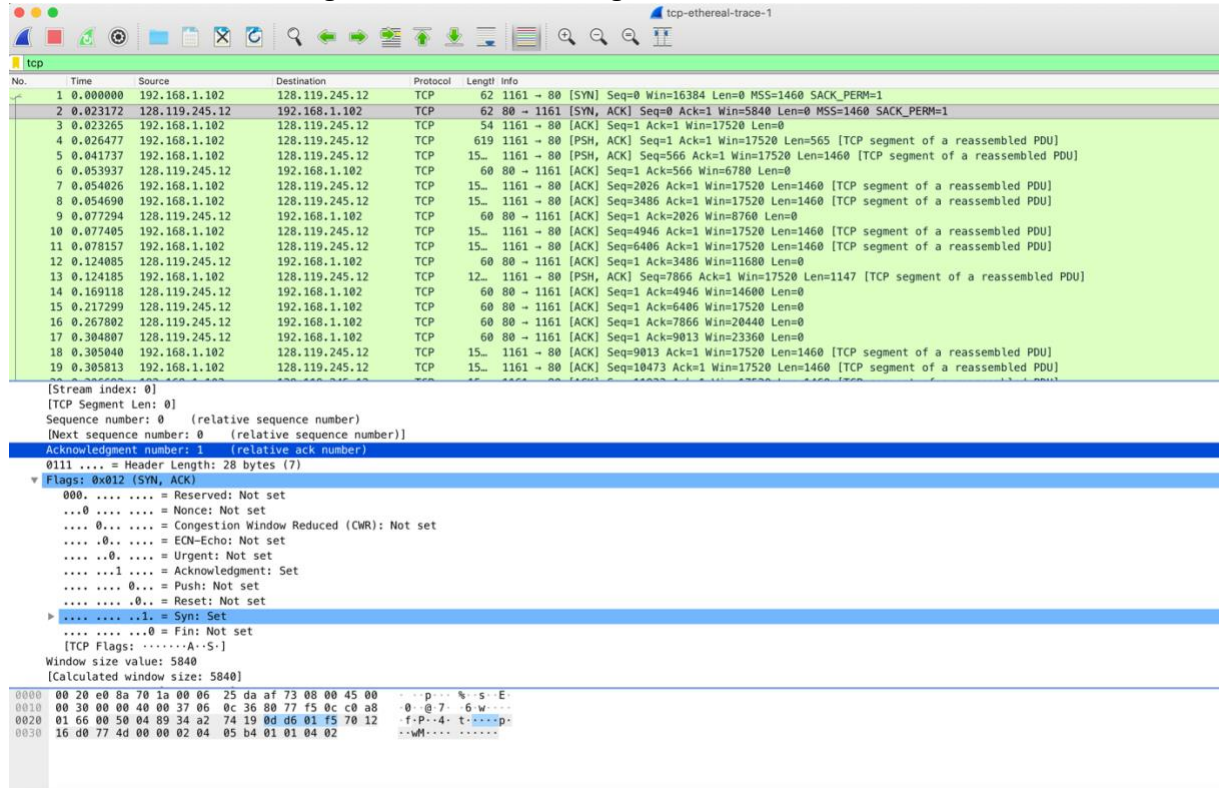


Figure 1: SYN flag set and sequence number of SYN segment is 0.

- 4) The sequence number of the SYN segment is 0. The SYN flag is set to 1; this indicates that the segment is a SYN segment.

- 5) The sequence number of the SYNACK segment in response to the SYN segment is 0. The value of the acknowledgement field is 1. Gaia.cs.umass.edu determines the value of the acknowledgement field by adding one to the initial sequence number of the SYN segment sent from the source. The SYN and acknowledgement flags are both set to 1, this identifies this segment as a SYNACK segment



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.1.102	128.119.245.12	TCP	62	1161 → 80 [SYN] Seq=0 Win=16384 Len=0 MSS=1460 SACK_PERM=1
2	0.023172	128.119.245.12	192.168.1.102	TCP	62	80 → 1161 [SYN, ACK] Seq=0 Ack=1 Win=5840 Len=0 MSS=1460 SACK_PERM=1
3	0.023265	192.168.1.102	128.119.245.12	TCP	54	1161 → 80 [ACK] Seq=1 Ack=1 Win=17520 Len=0
4	0.026477	192.168.1.102	128.119.245.12	TCP	619	1161 → 80 [PSH, ACK] Seq=1 Ack=1 Win=17520 Len=565 [TCP segment of a reassembled PDU]
5	0.041737	192.168.1.102	128.119.245.12	TCP	15	1161 → 80 [PSH, ACK] Seq=566 Ack=1 Win=17520 Len=1460 [TCP segment of a reassembled PDU]
6	0.053937	128.119.245.12	192.168.1.102	TCP	60	80 → 1161 [ACK] Seq=1 Ack=566 Win=6780 Len=0
7	0.054026	192.168.1.102	128.119.245.12	TCP	15	1161 → 80 [ACK] Seq=2026 Ack=1 Win=17520 Len=1460 [TCP segment of a reassembled PDU]
8	0.054690	192.168.1.102	128.119.245.12	TCP	15	1161 → 80 [ACK] Seq=3486 Ack=1 Win=17520 Len=1460 [TCP segment of a reassembled PDU]
9	0.077294	128.119.245.12	192.168.1.102	TCP	60	80 → 1161 [ACK] Seq=1 Ack=2026 Win=8760 Len=0
10	0.077405	192.168.1.102	128.119.245.12	TCP	15	1161 → 80 [ACK] Seq=4946 Ack=1 Win=17520 Len=1460 [TCP segment of a reassembled PDU]
11	0.078157	192.168.1.102	128.119.245.12	TCP	15	1161 → 80 [ACK] Seq=6406 Ack=1 Win=17520 Len=1460 [TCP segment of a reassembled PDU]
12	0.124085	128.119.245.12	192.168.1.102	TCP	60	80 → 1161 [ACK] Seq=1 Ack=3486 Win=11680 Len=0
13	0.124185	192.168.1.102	128.119.245.12	TCP	12	1161 → 80 [PSH, ACK] Seq=7866 Ack=1 Win=17520 Len=1147 [TCP segment of a reassembled PDU]
14	0.169118	128.119.245.12	192.168.1.102	TCP	60	80 → 1161 [ACK] Seq=1 Ack=4946 Win=14600 Len=0
15	0.217299	128.119.245.12	192.168.1.102	TCP	60	80 → 1161 [ACK] Seq=1 Ack=6406 Win=17520 Len=0
16	0.267802	128.119.245.12	192.168.1.102	TCP	60	80 → 1161 [ACK] Seq=1 Ack=7866 Win=20440 Len=0
17	0.304807	128.119.245.12	192.168.1.102	TCP	60	80 → 1161 [ACK] Seq=1 Ack=9813 Win=23360 Len=0
18	0.305840	192.168.1.102	128.119.245.12	TCP	15	1161 → 80 [ACK] Seq=9013 Ack=1 Win=17520 Len=1460 [TCP segment of a reassembled PDU]
19	0.305813	192.168.1.102	128.119.245.12	TCP	15	1161 → 80 [ACK] Seq=10473 Ack=1 Win=17520 Len=1460 [TCP segment of a reassembled PDU]

[Stream index: 0]
[TCP Segment Len: 0]
Sequence number: 0 (relative sequence number)
[Next sequence number: 0 (relative sequence number)]
Acknowledgment number: 0 (relative ack number)
0111 = Header Length: 28 bytes (7)
▼ Flags: 0x012 (SYN, ACK)
000. = Reserved: Not set
...0. = Nonce: Not set
....0... = Congestion Window Reduced (CWR): Not set
....0.. = ECN-Echo: Not set
....0. = Urgent: Not set
....1... = Acknowledgment: Set
....0... = Push: Not set
....0.. = Reset: Not set
►1. = Syn: Set
....0... = Fin: Not set
[TCP Flags:A..S..]
Window size value: 5840
[Calculated window size: 5840]
0000 00 20 e0 8a 70 1a 00 06 25 da af 73 08 00 45 00 ...p...%s...E...
0010 00 30 00 00 40 00 37 06 0c 36 80 77 f5 0c c0 a8 ...@...6w...
0020 01 66 00 50 04 89 34 a2 74 19 8d d6 01 f5 70 12 ...fP...t...p...
0030 16 d0 77 4d 00 00 02 04 05 b4 01 01 04 02 ...wM....

Figure 2: SYN flag and ACK flag of the SYNACK segment is set.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.1.102	128.119.245.12	TCP	62	1161 → 80 [SYN] Seq=0 Win=16384 Len=0 MSS=1460 SACK_PERM=1
2	0.023172	128.119.245.12	192.168.1.102	TCP	62	80 → 1161 [SYN, ACK] Seq=0 Ack=1 Win=5840 Len=0 MSS=1460 SACK_PERM=1
3	0.023265	192.168.1.102	128.119.245.12	TCP	54	1161 → 80 [ACK] Seq=1 Ack=1 Win=17520 Len=0
4	0.026477	192.168.1.102	128.119.245.12	TCP	619	1161 → 80 [PSH, ACK] Seq=1 Ack=1 Win=17520 Len=565 [TCP segment of a reassembled PDU]
5	0.041737	192.168.1.102	128.119.245.12	TCP	15	1161 → 80 [PSH, ACK] Seq=566 Ack=1 Win=17520 Len=1460 [TCP segment of a reassembled PDU]
6	0.053937	128.119.245.12	192.168.1.102	TCP	60	80 → 1161 [ACK] Seq=1 Ack=566 Win=6780 Len=0
7	0.054026	192.168.1.102	128.119.245.12	TCP	15	1161 → 80 [ACK] Seq=2026 Ack=1 Win=17520 Len=1460 [TCP segment of a reassembled PDU]
8	0.054690	192.168.1.102	128.119.245.12	TCP	15	1161 → 80 [ACK] Seq=3486 Ack=1 Win=17520 Len=1460 [TCP segment of a reassembled PDU]
9	0.077294	128.119.245.12	192.168.1.102	TCP	60	80 → 1161 [ACK] Seq=1 Ack=2026 Win=8760 Len=0
10	0.077405	192.168.1.102	128.119.245.12	TCP	15	1161 → 80 [ACK] Seq=4946 Ack=1 Win=17520 Len=1460 [TCP segment of a reassembled PDU]
11	0.078157	192.168.1.102	128.119.245.12	TCP	15	1161 → 80 [ACK] Seq=6406 Ack=1 Win=17520 Len=1460 [TCP segment of a reassembled PDU]
12	0.124085	128.119.245.12	192.168.1.102	TCP	60	80 → 1161 [ACK] Seq=1 Ack=3486 Win=11680 Len=0
13	0.124185	192.168.1.102	128.119.245.12	TCP	12	1161 → 80 [PSH, ACK] Seq=7866 Ack=1 Win=17520 Len=1147 [TCP segment of a reassembled PDU]
14	0.169118	128.119.245.12	192.168.1.102	TCP	60	80 → 1161 [ACK] Seq=1 Ack=4946 Win=14600 Len=0
15	0.217299	128.119.245.12	192.168.1.102	TCP	60	80 → 1161 [ACK] Seq=1 Ack=6406 Win=17520 Len=0
16	0.267802	128.119.245.12	192.168.1.102	TCP	60	80 → 1161 [ACK] Seq=1 Ack=7866 Win=20440 Len=0
17	0.304807	128.119.245.12	192.168.1.102	TCP	60	80 → 1161 [ACK] Seq=1 Ack=9013 Win=23360 Len=0
18	0.305040	192.168.1.102	128.119.245.12	TCP	15	1161 → 80 [ACK] Seq=9013 Ack=1 Win=17520 Len=1460 [TCP segment of a reassembled PDU]
19	0.305813	192.168.1.102	128.119.245.12	TCP	15	1161 → 80 [ACK] Seq=10473 Ack=1 Win=17520 Len=1460 [TCP segment of a reassembled PDU]

▶ Frame 11: 1514 bytes on wire (12112 bits), 1514 bytes captured (12112 bits) on interface 0
 ▶ Ethernet II, Src: Actionte_8a:70:1a (00:20:e0:8a:70:1a), Dst: LinksysG_daf:73 (00:06:25:daf:73)
 ▶ Internet Protocol Version 4, Src: 192.168.1.102, Dst: 128.119.245.12
 ▶ Transmission Control Protocol, Src Port: 1161, Dst Port: 80, Seq: 6406, Ack: 1, Len: 1460
 Source Port: 1161
 Destination Port: 80
 [Stream index: 0]
 [TCP Segment Len: 1460]
 Sequence number: 6406 (relative sequence number)
 [Next sequence number: 7866 (relative sequence number)]
 Acknowledgment number: 1 (relative ack number)
 0101 ... = Header Length: 20 bytes (5)
 ▶ Flags: 0x010 (ACK)
 Window size value: 17520
 [Calculated window size: 17520]
 [Window size scaling factor: -2 (no window scaling used)]
 Checksum: 0x9583 [unverified]
 [Checksum Status: Unverified]
 Urgent pointer: 0
 ▶ [SEQ/ACK analysis]
 0020 f5 0c 04 89 00 50 d6 1a fa 34 a2 74 1a 50 10P...4..t..P
 0030 44 70 95 83 00 20 55 6e 69 74 65 64 20 53 74 Dp....U nited St
 0040 61 74 65 73 20 63 6f 70 79 72 69 67 68 74 00 0a ates cop yright
 0050 ef 6a 70 ef 77 70 6c ef 77 70 74 60 60 73 70 77 ..a..fa r thie u

Figure 3: First 6 segments

- 6) The sequence number of the segment containing the HTTP POST command is 1
- 7) The first six segments are no.4,5,7,8,10,11 in the trace.

Segment 1 sequence number: 1
 Segment 2 sequence number: 566
 Segment 3 sequence number: 2026
 Segment 4 sequence number: 3486
 Segment 5 sequence number: 4946
 Segment 6 sequence number: 6406

Segment 1
 Send time = 0.026477
 ACK received time = 0.053937
 RTT = 0.02746

Segment 2
 Send time = 0.041737
 ACK received time = 0.077294
 RTT = 0.035557

Segment 3

Send time = 0.054026

ACK received time = 0.124085

RTT = 0.070059

Segment 4

Send time = 0.054690

ACK received time = 0.169118

RTT = 0.11443

Segment 5

Send time = 0.077405

ACK received time = 0.217299

RTT = 0.13989

Segment 6

Send time = 0.078157

ACK received time = 0.267802

RTT = 0.18964

$\text{EstimatedRTT} = 0.875 * \text{EstimatedRTT} + 0.125 * \text{SampleRTT}$

After segment 1 ACK

$\text{EstimatedRTT} = 0.02746$

After segment 2 ACK

$\text{EstimatedRTT} = 0.875 * 0.02746 + 0.125 * 0.035557 = 0.0285$

After segment 3 ACK

$\text{EstimatedRTT} = 0.875 * 0.0285 + 0.125 * 0.070059 = 0.0337$

After segment 4 ACK

$\text{EstimatedRTT} = 0.875 * 0.0337 + 0.125 * 0.11443 = 0.0438$

After segment 5 ACK

$\text{EstimatedRTT} = 0.875 * 0.0438 + 0.125 * 0.13989 = 0.0558$

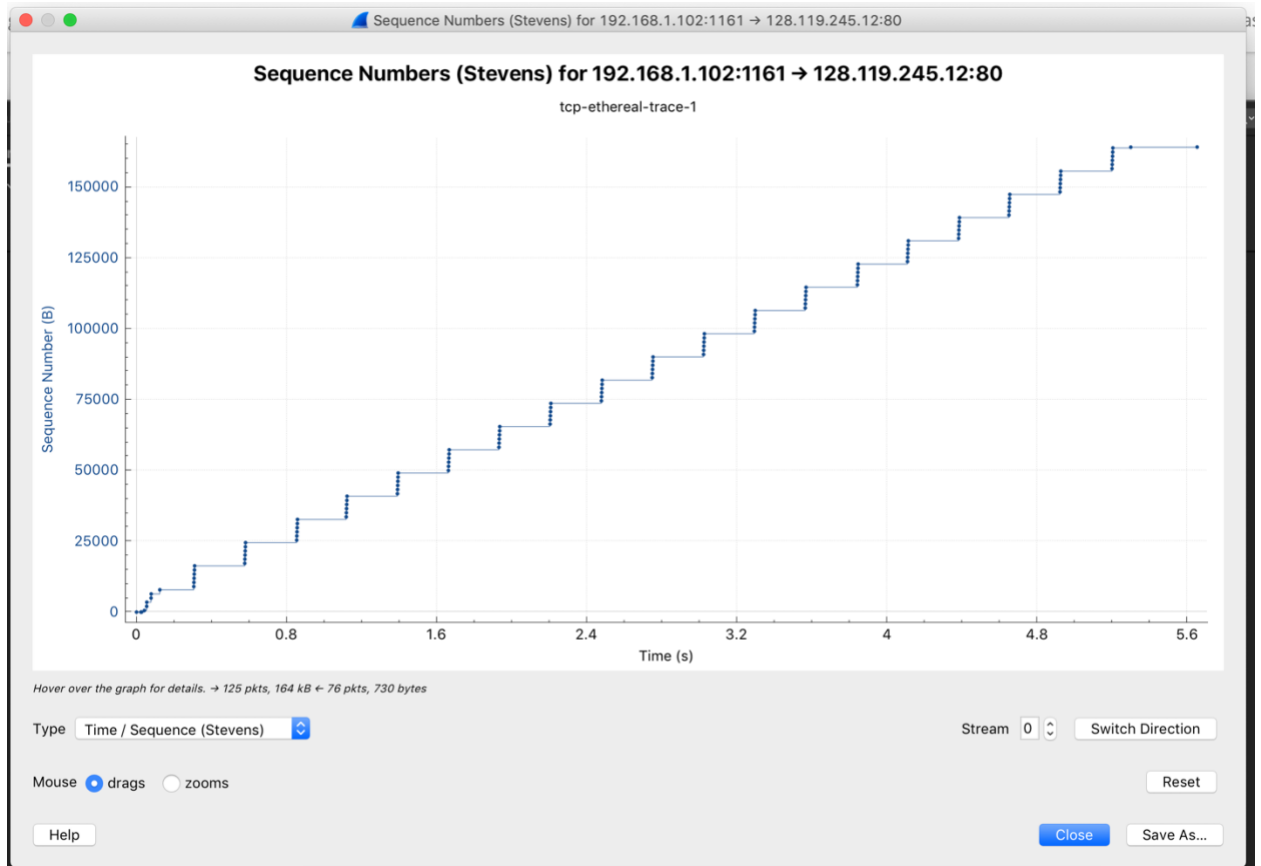
After segment 6 ACK

$$\text{EstimatedRTT} = 0.875 * 0.0558 + 0.125 * 0.18964 = 0.0725$$

- 8) Length of the first TCP segment (HTTP POST command) = 565 bytes
Length of segment 2,3,4,5,6 = 1460 bytes
- 9) The minimum amount of available buffer space advertised at the receiving end is 5840 bytes. The receiving end's window grows until it reaches a maximum size of 62780 bytes. The lack of buffer space never throttles the sender.
- 10) There are no re-transmitted segments in the trace file. In order to answer this question, I checked the sequence numbers of the TCP segments in the trace file. The sequence numbers are increasing. If there are any retransmitted segments, its sequence number will not be consistent with segments that are sent for the first time (its sequence number will be smaller).
- 11) The receiver will typically acknowledge 1460 bytes for segments it receives with the exception of the first segment (HTTP POST command segment) where it will acknowledge 566 bytes. There are cases where the receiver is acking every other segment. This happens with segment 80 where the receiver acknowledges $2920 = 1460 * 2$ bytes. The receiver also acknowledges every other segment on segment 87, 88
- 12) The throughput for the TCP connection is the ratio between the total amount of data and the total transmission time. We can compute the total amount of data transmitted by subtracting the sequence number of the first TCP segment from the acknowledged sequence number of the final ack. Thus, the total amount of data transmitted will be $164091 - 1 = 164090$ bytes.

The total transmission time can be computed by subtracting the time instant of the first TCP segment from the last segment. Thus, we get $5.455830 - 0.026477 = 5.4294$ seconds.

The average throughput for the TCP connection will be $164090/5.4294 = 30.2$ KB/s



- 13) TCP slow-start only lasts for the first 0.1 seconds. Afterwards, the TCP session is consistently in the congestion avoidance state. The sender appears to send segments in groups of 6. This is not caused by flow control because the receiver has a window greater than 6. Thus, the receiver probably has a limitation on the number of packets it receives.



14) TCP slow start seems to occur for the first 5.6-5.7 seconds, after which we reach the congestion avoidance phase. The sender seems to send segments in groups of 1-2 segments. Similar to 13, this does not seem to be caused by flow control because the receiver has a much larger window.

2) source code:

```
from collections import defaultdict
import math

class Graph:

    def __init__(self):
        self.nodes = set()
        self.edges = defaultdict(list)
        self.distances = {}
```

```

def add_node(self,value):
    self.nodes.add(value)

def add_edge(self, from_node, to_node, distance):
    self.edges[from_node].append(to_node)
    self.edges[to_node].append(from_node)
    self.distances[(from_node, to_node)] = distance
    self.distances[(to_node, from_node)] = distance

first_node = None
Q = None

# Specify graph properties by taking an input file from the user.
input_file_name = input("Enter input file name: ")

with open(input_file_name, "r") as input_file:
    num_lines = sum(1 for line in input_file)

#constructGraph will determine the network properties by reading the input file
def constructGraph():

    i = 0
    graph = Graph()

    with open(input_file_name, "r") as input_file:

        while i < num_lines:

            #Read the first line in the input file.
            if i == 0:
                line1 = input_file.readline()
                line1_arr = line1.split()

                num_nodes = int(line1_arr[0])

```



```

num_links = int(line1_arr[1])

i += 1

#Read the second line in the input file. (list of node names)
if i == 1:
    line2 = input_file.readline()
    node_names = line2.split()

    for node_name in node_names:
        graph.add_node(node_name)

    global Q
    Q = node_names

    global first_node
    first_node = node_names[0]

    i += 1
    continue

#For the rest of the lines, connect edges to nodes given the input.
line = input_file.readline()
line_arr = line.split()
graph.add_edge(line_arr[0], line_arr[1], line_arr[2])

i += 1

#The distance from each node to itself is 0
for node in graph.nodes:
    graph.distances[(node,node)] = "0"

return graph

def ls_routing(graph, src_node, outputfile):

```

```

print("Link state routing results: ")

dist = dict() #dist will be a dictionary of all the nodes in the graph and their distances from the source node
prev = dict() #prev will be used to construct the least costly path by backtracking each node
min_path = dict()

global Q
temp_Q = list()

for node in Q:
    temp_Q.append(node)

dist[src_node] = 0 #Distance from initial node to initial node is 0
prev[src_node] = None

while temp_Q:

    #The source node will be selected first, it's distance from itself is 0
    u = None
    for node in temp_Q:
        if node in dist:
            if u is None:
                u = node
            elif dist[node] < dist[u]:
                u = node

    temp_Q.remove(u)

    #For each node in the network, compute the shortest path to it from the source node
    for edge in graph.edges[u]:

        alt = dist[u] + int(graph.distances[(u, edge)])

        if edge not in dist:
            dist[edge] = math.inf

```

```

        if alt < dist[edge]:
            dist[edge] = alt
            prev[edge] = u

#Construct the paths for each node for the output file by backtracking with prev
for node in Q:
    temp_Q.append(node)
temp_Q.remove(src_node)

for node in temp_Q:
    curr_node = node
    min_path[curr_node] = [curr_node]

    while prev[curr_node] != None:
        if prev[curr_node] not in min_path[node]:
            min_path[node].append(prev[curr_node])
            curr_node = prev[curr_node]

for node in temp_Q:
    min_path[node] = list(reversed(min_path[node]))

#Write our results to the output file
with open(outputfile, "w") as outputfile:
    for node in temp_Q:
        outputfile.write(str(node) + ": ")
        print(str(node) + ": ", end = "")
        for edge in min_path[node]:
            outputfile.write(str(edge))
            print(str(edge), end = "")

            if edge == min_path[node][-1]:
                outputfile.write(" " + str(dist[node]))
                print(" " + str(dist[node]), end = "")
                outputfile.write("\n")
                print("\n", end = "")
                continue

```

```
outputfile.write("-")  
print("-", end = ")
```

```
def dv_routing(graph, src_node, outputfile):
```

```
    print("Distance vector routing results: ")
```

```
    dist = dict()
```

```
    prev = dict()
```

```
    global Q
```

```
    temp_Q = list()
```

```
    iterations = 0
```

```
    for node in Q:
```

```
        temp_Q.append(node)
```

```
    #Initialize the distances from the source node to each node in the network to infinity
```

```
    for node in Q:
```

```
        dist[node] = math.inf
```

```
    #Initialize the distance from the source node to itself to 0
```

```
    dist[src_node] = 0
```

```
    #Main part of the algorithm
```

```
    for i in range(0, len(graph.nodes)):
```

```
        for node in graph.nodes:
```

```
            for edge in graph.edges[node]:
```

```
                dist[edge] = min(dist[edge], dist[node] + int(graph.distances[(node, edge)]))
```

```
    iterations += 1
```

```
    #Initialize a table with all 0s
```

```
    table = []
```

```
    for i in range(0, len(graph.nodes) + 1):
```

```

table.append([])

for j in range(0, len(graph.nodes)+1):
    table[i].append("0")

#Fill in the first row and first column of the table
for row in table:
    for val in row:

        if table.index(row) == 0 and row.index(val) == 0:
            table[table.index(row)][row.index(val)] = " "

        elif table.index(row) == 0:
            for node in temp_Q:
                if temp_Q.index(node)+1 == row.index(val):
                    table[table.index(row)][row.index(val)] = node

        if table.index(row) > 0 and row.index(val) == 0:
            for node in temp_Q:
                if temp_Q.index(node) + 1 == table.index(row):
                    table[table.index(row)][row.index(val)] = node

#Fill in the rest of the table with distance values
for row in table:
    for val in row:

        for node in temp_Q:
            if table.index(row) > 0:
                if table[table.index(row)][row.index(val)] == node:
                    for i in range(1, len(graph.nodes)+1):

                        table[table.index(row)][row.index(val)+i] = graph.distances[(node, table[0][i])]

i = 0

#Print our results to system.out
print("\t\tCost to\n")

```

```

for row in table:
    if i == 1:
        print("From\t"+str(row)+"\n")
    else:
        print("\t\t"+str(row)+"\n")
    i +=1
print("Number of rounds: ",iterations)

#Write our results to the output file
with open(outputfile, "w") as outputfile:
    outputfile.write("\t")
    outputfile.write("Cost to")
    outputfile.write("\n\n")
    i = 0
    for row in table:
        if i == 1:
            outputfile.write("From\t" + str(row))
            outputfile.write("\n")
        else:
            outputfile.write("\t" +str(row))
            outputfile.write("\n")
        i +=1
    outputfile.write("\n")
    outputfile.write("Number of rounds: "+str(iterations))

ls_routing(constructGraph(), first_node, "LS_output.txt")
print("\n")

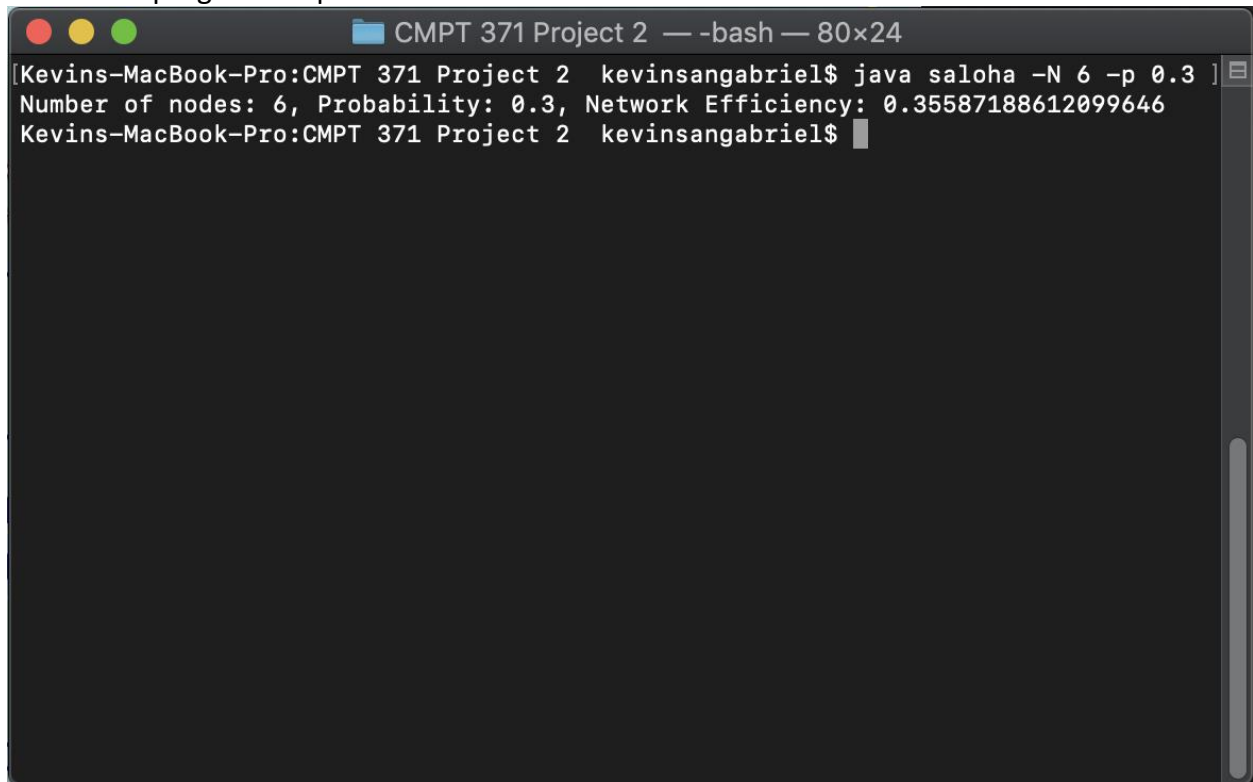
dv_routing(constructGraph(), first_node, "DV_output.txt")

```

3)

$$Np(1-p)^{N-1} = (6)(0.3)(0.7)^5 = 0.302526$$

Simulation program output results:

A screenshot of a macOS terminal window titled "CMPT 371 Project 2 — -bash — 80x24". The terminal shows the execution of a Java command: `java saloha -N 6 -p 0.3`. The output of the command is: `Number of nodes: 6, Probability: 0.3, Network Efficiency: 0.35587188612099646`. The prompt returns to the user's shell.

```
[Kevins-MacBook-Pro:CMPT 371 Project 2 kevinsangabriel$ java saloha -N 6 -p 0.3 ]
Number of nodes: 6, Probability: 0.3, Network Efficiency: 0.35587188612099646
Kevins-MacBook-Pro:CMPT 371 Project 2 kevinsangabriel$
```

The network efficiency obtained by the simulation was nearly consistent with the theoretical model. The network efficiency of the simulation was more efficient by a utilization magnitude of $0.355871 - 0.302256 = 0.053615$.