CS486 Network Security - Project Specification Fall 2020
End-to-End Detection of Network Compression

Due: November 5th, 2020 at 11:59PM

1 Project Outcomes

1. Implement two network applications (client/server and standalone) that detect network compression by end-hosts.
2. Verify and validate your compression detection applications.

2 Project Overview

For this project, you will implement two network applications to detect if network compression is present on a network path, and if found, to locate the compression link. The first application you are to implement is a client/server application. The second application is a standalone application that works in an uncooperative environment. The standalone application detects network compression without requiring special software to be installed on the other host. This is inspired by the work, End-to-End Detection of Compression of Traffic Flows by Intermediaries, which is recommended that you read in detail up to Section V.

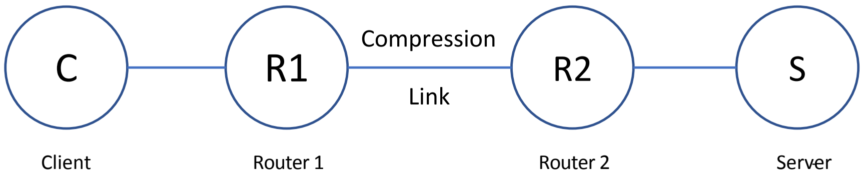

Figure 1: A simple topology between two nodes with a compression link on the path.

3 Components

3.1 Compression Detection Client/Server Application

You will implement the network compression detection *only in the cooperative environment*. In summary, your network application is a client/server application where the sender sends two sets of n UDP packets back-to-back (called packet train). The receiver records the arrival time between the first and last packet in the train (packets 0 and $n-1$).[1] The first packet train consists of all packets of size ℓ bytes in payload, filled with all 0's, while the second packet train contains a random sequence of bits. You can generate a random sequence of bits using `/dev/urandom`. If the difference in arrival time between the first and last packets of the two trains ($\Delta t_H - \Delta t_L$) is more than a fixed threshold $\tau = 100\ ms$, the client application reports `Compression detected!`, whereas when the time difference is less than τ, there was probably no compression link on the path and the client application should display `No compression was detected.`

Every UDP packet in the packet train should have the following characteristics (Figure 2):

1. Don't Fragment flag in IP header set to 1
2. Source Port: 9876
3. Destination Port[2]: 8765

[1] If tail packets are lost, the application records the arrival time of the last packet received in the train instead.
[2] That requires the server to listen to this port to capture incoming packets.

4. Packet ID: the first 16 bits of the payload are reserved for a unique packet ID for each UDP packet in the train, starting with packet ID 0.

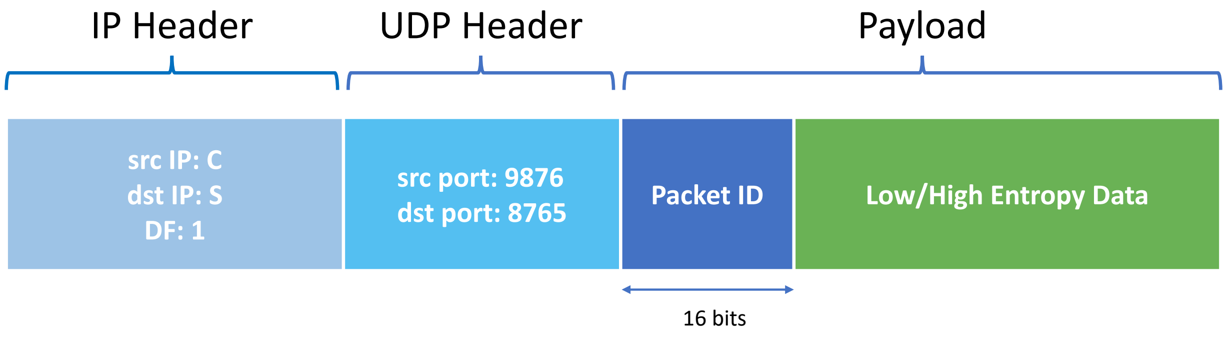

Figure 2: UDP Packet Structure.

Your client/server application consists of three phases:

1. **Pre-Probing Phase TCP Connection**

   The client application initiates a TCP connection to the server. Upon a successful connection, the client then passes all the configuration file's contents to the server (see Section 4). The server will use this information in later phases to detect whether network compression is present or not. The TCP connection is released upon the successful transmission of the contents of the configuration file.

2. **Probing Phase**

   In this phase, the sender sends the UDP packets. After sending n UDP packets with low entropy data, the client must wait Inter-Measurement Time (γ) seconds before sending the packet train consists of n UDP packets with high entropy data. This is to ensure that the two probing phases will not interfere with one another.

3. **Post-Probing Phase TCP Connection**

   After all packets are arrived to the server, the server performs the necessary calculations required to detect the presence of network compression on the path between the server and the client. After the probing phase, the client application initiates another TCP connection. Once the connection is established, the server sends its findings to the client.

3.2 (2) Compression Detection Standalone Application

You are also to implement a standalone application that detects network compression, without requiring any cooperation from the server, other than responding to standard network events. This application also locates the link where the compression is applied.

Using a raw socket implementation, your program sends a single *head* SYN packet. The SYN packet is immediately followed by a train of n UDP packets, and a single *tail* SYN packet. The SYN packets are sent to two different ports (x and y) that are not among well-known ports, so they are expected to be inactive or closed ports that no application is listening to. Sending SYN packets to closed ports should trigger RST packets from the server. Your application only needs to record two RST packets' arrival time and may ignore any ICMP messages generated by the UDP packets. The difference between the arrival time of the RST packets is then used to determine whether network compression was detected or not. If either of the RST packets is lost (you need a timer to infer loss) or the server not responding with an RST packet, your application will output `Failed to detect due to insufficient information.` and terminates.

The UDP packets are constructed exactly the same way as described in Section 3.1, with one exception: their TTL field in the IP header is set manually to a fixed value for all UDP packets in the packet train. The user provides this information (see Section 4). Figure 3 illustrates the sequence of packets the standalone application generates.

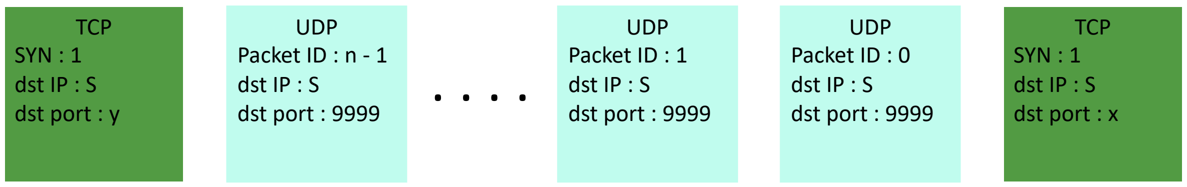

Figure 3: The illustration of the packet train generated by the standalone compression detection application.

4 Program Interface

You are to implement a Linux-based command-line program that takes one command-line argument: the *configuration file*.[3] Below are examples of how one would execute your applications in Linux:

```
#On the client system
./compdetect_client myconfig.json
```

```
#On the server system
./compdetect_server
```

```
#On any system (the standalone program)
./compdetect myconfig.json
```

4.1 Configuration File

Your applications must use a configuration file to obtain the values for the parameters listed in this section. You may add other parameters as needed to this list. Depending on which of your three applications are reading from the configuration file, some parameters may be irrelevant; thus, they should be ignored. For instance, the server application only some parameters such as "Port Number for TCP (Pre-/Post- Probing Phases)" and ignores the rest. Or, only the standalone application uses the TTL value provided in the configuration file, so it is ignored by both client and server applications.

1. The Server's IP Address
2. Source Port Number for UDP
3. Destination Port Number for UDP
4. Destination Port Number for TCP Head SYN, x
5. Destination Port Number for TCP Tail SYN, y
6. Port Number for TCP (Pre-/Post- Probing Phases)
7. The Size of the UDP Payload in the UDP Packet Train, ℓ (*default value: 1000B*)
8. Inter-Measurement Time, γ (*default value: 15 seconds*)
9. The Number of UDP Packets in the UDP Packet Train, n (*default value: 6000*)
10. TTL for the UDP Packets (*default value: 255*)

Good luck!

[3]You may choose any configuration file format as you wish. Some popular formats include but not limited to JSON, INI, and XML.