Network Programming Laboratory

3 June 2024

Exercise 1

TCP flows can be uniquely identified by the canonical four-tuple `<srcip, srcprt, dstip, dstprt>`. However, TCP traffic is inherently bi-directional; hence packets travelling in the reverse direction should be tagged with the same identifier. In C++, the type `tcpflowid_t` type can be defined as:

```
struct tcpflowid_t {
    uint32_t srcip;
    uint16_t srcprt;
    uint32_t dstip;
    uint16_t dstprt;
};
```

The C++ `unordered_map` container implements a collision hash table. In order to use a key of type `tcpflowid_t` it is then necessary to specialize the template classes `std::hash` and `std::equal_to`. This can be done by implementing both the *call* and *bool* operators in the following code:

```
namespace std {
    template <>
    struct hash<tcpflowid_t> {
        size_t operator()(const tcpflowid_t& f) const {
            //
            // Implement the hash function here
            //
        };
    };

    template<>
    struct equal_to<tcpflowid_t>  {
        bool operator()(const tcpflowid_t& lhs, const tcpflowid_t& rhs) const {
            //
            // Implement the bool operator here
            //
        }
    };
}
```

The file `20240603.hpp` contains the definition of `tcpflowid_t` as well as the skeleton for the above classes specialization.

1. Fill in the call `std::hash<tcpflowid_t>::operator()` to produce a *symmetric* hash function, i.e. a function that returns the same value if applied to flow identifiers like `tcpflowid_t(x,y,z,w)` and `tcplowid_t(z,w,x,y)`. *Hint: using the XOR logical operator may help a lot here...*
2. Fill in the `bool std::hash<tcpflowid_t>::operator()` to compare two `tcpflowid_t` objects. Again, remember that `tcpflowid_t(x,y,z,w)` and `tcpflowid_t(z,w,x,y)` identifies the same TCP flow.
3. Compile and run the test program `one.cpp` to check the correctness of the above implementations.

Exercise 2

TCP is a connection oriented protocol, hence any connection must be established before sending/receiving data. The following connection establishment process is known as *three-way handshake*:

- the client sends a segment with the `SYN` flag set to 1;
- the server responds with a segment in which both the `SYN` flag and the `ACK` flag are set to 1;
- finally the client concludes the handshake by sending a segment with the `ACK` flag set to 1.

Conversely, TCP connections can be either *abruptly* or *gracefully* closed. In the *abrupt* scenario, it is sufficient for one of the two peers to send a segment with the `RST` flag set to 1 to teardown the connection. In the *graceful* scenario, instead:

- one of the two peers (say P_1) sends a segment with the `FIN` flag set to 1;
- the other peer (say P_2) responds with a segment with the `ACK` flag set to 1 (*half close*), and then continues sending segments (if any);
- when P_2 has no more data to transmit, it also sends a segment with the `FIN` flag set to 1;
- P_1 then acknowledges receiving the `FIN` segment by responding with a segment with the `ACK` flag set to 1.

In this exercise, it is requested to implement a simple *TCP connection tracker* to detect full TCP connections, from their initial setup to their closing. To process of exactly tracking a TCP connection can be very tedious and it involves checking sequence/ack numbers, detecting retransmissions and so on. However, since our goal is to track full connections only, the whole process can be greatly simplified as it follows.

- The TCP connection is set as open once the very first segment carrying a `SYN` flag (but not the `ACK` flag) is found.
- The TCP connection is set as closed once the first segment carrying an `ACK` flag after two segments carrying a `FIN` flag is found.

The above mechanism is clearly approximated as it does not check sequence numbers and does not account for retransmissions, out-of-order segments, etc.. However, for our purposes, the algorithm provides reasonable estimates of the traffic volume exchanged by the two peers, of the completion time of connections and their throughput.

1. Based on the above described mechanism, write and application that reads a pcap file trace and extracts the *completed* TCP connections. For each of them, report:
   - the number of packets exchanged between the two TCP peers
   - the number of bytes (TCP payload data only) exchanged between the two TCP peers
   - the completion time of the connection
   - the connection throughput
2. (Extra) Extend the tracker app to perform TCP connection tracking on live capture captured from a physical network device.