OpenLitGen, an open source implementation of LitGEN traffic generator

Anderson Agondi dos Santos Paschoalon

No Institute Given

Resumo The abstract should summarize the contents of the paper using at least 70 and at most 150 words. It will be set in 9-point font size and be inset 1.0 cm from the right and left margins. There will be two blank lines before and after the Abstract. ...

Keywords: computational geometry, graph theory, Hamilton cycles

1 Introduction

2 Bibliographic Review

2.1 Realist Traffic Generation

Realistic traffic generators have been the subject of extensive research, as evidenced by numerous studies and papers [?, ?, ?]. Various approaches and models have been proposed to address the need for accurate traffic generation. Traditional constant-traffic generators are used to measure metrics in idealistic (and often unrealistic) scenarios. These generators are simpler to use and facilitate the extraction of features from network components. However, they fall short in mimicking the stochastic nature of real network traffic, which is essential for understanding network performance under realistic conditions.

Realistic traffic generators, on the other hand, aim to generate network traffic with stochastic characteristics that closely resemble those occurring in real scenarios. It has been demonstrated that the realistic burstiness of network traffic significantly impacts network performance, such as queue lengths, when compared to constant-traffic scenarios.

2.2 LitGen

LiTGen, as introduced in the paper "LiTGen, a lightweight traffic generator: application to P2P and mail wireless traffic" [?], is an easy-to-use and tune open-loop traffic generator. The primary goal of LiTGen is to capture and reproduce the burstiness and properties of actual network traffic.

The LitGen model is user-oriented and hierarchical, as outlined in the original paper. This architecture can be divided into multiple levels: user, session, object, and packet levels.

User Level At the highest level, we assume there are n users sending traffic. Each user undergoes an infinite succession of "session-periods"and "inter-session-periods". During a session period, the user is active, while during an inter-session period, the user is completely idle.

Session Level At this level, two random variables are defined:

- N_{session}: The number of objects sent during a session period.
- T_{is}: The duration of the inter-session period.

This layer of abstraction emulates the behavior of users, whether they are humans or machines interacting with the network.

Object Level An object represents each element transmitted during a session period. This abstraction layer emulates the transfer of various objects over a network, such as HTML pages, videos, files, etc. This level also has two random variables:

- N_{obj}: The number of IP packets that compose the object.
- IA_{obj}: The inter-arrival times among each object in a session.

Packet Level Finally, since each object consists of a certain number of IP packets, this layer models the inter-packet times within an object, using the random variable:

- IA_{pkt}: The inter-arrival times between packets within an object.

Despite its relative simplicity, LiTGen achieves remarkable results. However, the source code for LiTGen is not publicly available, limiting its accessibility and utility for the broader research community.

In this work, we aim to provide an open-source implementation of LiTGen, which we call OpenLitGen. OpenLitGen comprises two executables:

- ***litgen-model***: This tool is responsible for extracting features from actual traffic captures in pcap files, fitting the models, and storing their parameters in text files with a `.lit` extension.
- ***litgen-tg***: This tool stands for LitGen traffic generator. It reads the parameters stored in `.lit` files, instantiates the statistical models, and generates traffic using the LiTGen architecture.

By making OpenLitGen available, we aim to facilitate further research and development in realistic traffic generation, providing the community with a versatile and accurate tool for modeling network traffic.

3 Methodology and Implementation

Despite the LitGen paper's clear explanation of its architecture, certain assumptions were necessary in our implementation. These assumptions are explicitly highlighted where they occur. OpenLitGen was developed in C++ using Libtins as the primary library for pcap sniffing and traffic generation. For simulation and testing purposes, CSV files can also be used for input and output.

The following sections detail the implementation of *litgen-model* and *litgen-tg*.

3.1 litgen-model

Despite the clear explanation of its architecture in the LitGen paper, certain assumptions were necessary in our implementation. These assumptions are explicitly highlighted where they occur. OpenLitGen was developed in C++ using Libtins as the primary library for pcap sniffing and traffic generation. For simulation and testing purposes, CSV files can also be used for input and output.

The following sections detail the implementation of ***litgen-model*** and ***litgen-tg***.

3.2 litgen-model

The ***litgen-model*** is responsible for extracting features from all packets in the input pcap files. These features include interarrival time, packet size, source and destination IP addresses, source and destination ports, and TCP flags (SYN, ACK, FIN, and RST). Consistent with the original paper, we model only TCP traffic.

First, we grouped all packets by users. Since the model operates on a per-user and per-application basis, for simplicity, we treat each user and application as a separate user. This implies that each pair {port source, IP destination} is grouped as a different user. This approach is sensible since each {port source, IP destination} represents a client, and each pair {port destination, IP source} represents a server. We also extract and store all the pairs {port source, IP destination} and {port destination, IP source} as valid pairs for traffic generation.

We then sort all packets from the same user according to their arrival times. If the interval between packets is greater than 300 seconds (as suggested by the paper), it is considered an inter-session period. If the interval is smaller, it is a session. Thus, we have the user packets grouped into "session periods."

To classify packets into objects, we use the TCP flags. The first packet of a session interval is considered to belong to the first object. If a FIN or RST flag is identified, this signifies the end of the object, and the next packet will belong to the next object. In the end, we have a list of objects composed of packets.

From these sets, we can query:

- The list of inter-session periods from all users;
- The number of packets in all sessions;

– The number of packets in all objects;
– The list of inter-arrival times among all objects;
– The list of inter-arrival times between packets within all objects.

With these sets of data, we can calculate the random variables. For simplicity, we use exponential distributions to represent these random variables.

3.3 litgen-tg

With the total number of users, the five random variables (N_{session}, T_{is}, N_{obj}, IA_{obj}, IA_{pkt}), and a list of users and servers defined by IP and port pairs, we can generate synthetic traffic using the LitGen architecture.

The key steps are:

– Instantiate the random variables.
– Calculate the PDUs to be generated for each user separately.
– Keep track of the total elapsed time for each user, stopping PDU synthesis only when the timeout is exceeded, as the model is designed to continue indefinitely.
– For each user, select a random valid user IP and port.
– Continuously stack new PDUs into an array following the hierarchical model of sessions/inter-session times, objects/inter-object times, and inter-packet times.
– Finally, sort all synthesized objects by arrival time.

Below is the pseudo-code representation of the procedure used to synthesize and stack the PDUs:

```
Initialize packetVector
Initialize random distributions with different seeds

For each user:
    Reset accumulated time
    Get user's IP and port

    While session not ended:
        Generate number of objects in session
        For each object in session:
            Generate number of packets in object
            For each packet in object:
                If accumulated time exceeds timeout, end session
                Create PDU with attributes
                Add PDU to packetVector
                If not the last packet in the object:
                    Update accumulated time with inter-packet arrival time
            If session ended, break
            If not the last object in the session:
```

```
                Update accumulated time with inter-object arrival time
        Update accumulated time with inter-session duration

Sort packetVector by arrival time
```

With all the PDUs organized sequentially by arrival time, we can loop over the stacked PDUs to craft the packets and send them in sequence, waiting for the appropriate arrival time before sending the next packet.

4 Results

5 Conclusion

Referências