Insimulator

Release 0.1

Jul 04, 2021

Contents:

This Python package was designed to empirically study the transaction fees and privacy provisions of Bitcoin's Lightning Network (LN). The simulator relies only on the publicly available data of network structure and capacities, and generates transactions under assumptions that we validated based on information spread by blog posts of LN node owners.

CHAPTER 1

What's in it for me?

We think that our simulator can be of interest mainly for two types of people: LN node owners and researchers. Hence, the simulator can answer the following questions of interest for these people:

1.1 i.) LN node owners, routers:

- What is the optimal fee I could charge for transactions going through my node in order to maximise my routing profits?
- What is my expected income from routing with respect to certain parameters (topology, traffic, transacted amounts)?
- How various parameters (topology, traffic, transacted amounts) affect the profitability of my nodes?

1.2 ii.) Researchers:

- What is the optimal fee nodes can charge? How far is it (if at all) from on-chain fees?
- What is the path length distribution of transactions on the LN graph? How much anonymity do they provide?
- How profitable is it to run a router node? Who are the most profitable ones?
- Is everyone altruistic on the LN transaction fee market?
- How various parameters (topology, traffic, transacted amounts) affect the profitability of each node?

Our research paper **"A Cryptoeconomic Traffic Analysis of Bitcoin's Lightning Network"** is available on arXiv.

1.2.1 Getting started

By executing the steps below you can set up `lnsimulator` for your environment in a few minutes.

Requirements

- UNIX or macOS environment
  - For macOS users: you need to have wget (brew install wget)
- This package was developed in Python 3.5 (conda environment) but it works with Python 3.6 and 3.7 as well.

Install

After cloning the repository from GitHub you can install the simulator with `pip`.

```
git clone https://github.com/ferencberes/LNTrafficSimulator.git
cd LNTrafficSimulator
pip install .
```

Data

By providing daily LN snapshots as input **(you can bring and use your own!)**, our simulator models the flow of daily transactions.

i.) Download

You can download the data files, that we used in our research by executing the following command:

a.) Linux

```
bash ./scripts/download_data.sh
ls ln_data
```

b.) macOS

```
sh ./scripts/download_data.sh
ls ln_data
```

You can also download the compressed data file with this link.

ii.) Content

The *download_data.sh* script downloads 4 data files into the *ln_data* folder with the following content:

First example

Execute the following code to see whether your configuration was successful.

```
from lnsimulator.ln_utils import preprocess_json_file
import lnsimulator.simulator.transaction_simulator as ts

data_dir = "..." # path to the ln_data folder that contains the downloaded data
amount = 60000
count = 7000
epsilon = 0.8
drop_disabled = True
drop_low_cap = True
with_depletion = True
find_alternative_paths = False

print("# 1. Load LN graph data")
directed_edges = preprocess_json_file("%s/sample.json" % data_dir)

print("\n# 2. Load meta data")
node_meta = pd.read_csv("%s/1ml_meta_data.csv" % data_dir)
providers = list(node_meta["pub_key"])

print("\n# 3. Simulation")
simulator = ts.TransactionSimulator(directed_edges, providers, amount, count, drop_
↪disabled=drop_disabled, drop_low_cap=drop_low_cap, eps=epsilon, with_depletion=with_
↪depletion)
transactions = simulator.transactions
_, _, all_router_fees, _ = simulator.simulate(weight="total_fee", with_node_
↪removals=find_alternative_paths, max_threads=1)

print(all_router_fees.head())
print("Done")
```

If your configuration works then you should proceed to the detailed *documentation* of the LN traffic simulator.

1.2.2 Basic Documentation

In the steps below we suppose that you have already installed `lnsimulator` and downloaded the related LN data. If it is not the case then should follow the instructions in the *Getting Started* section first.

Prepare LN data

In order to run the simulation you need to provide LN snapshots as well as information about merchants nodes.

LN snapshots

The network structure is fed to `lnsimulator` in the form of LN snapshots. Raw JSON files as well as preprocessed payment channel data can be used as input.

a.) Load data from JSON file

```
from lnsimulator.ln_utils import preprocess_json_file

data_dir = "..." # path to the ln_data folder that contains the downloaded data
directed_edges = preprocess_json_file("%s/sample.json" % data_dir)
```

b.) Load preprocessed LN snapshots

In this case you should select data related to a given daily snapshot

```
import pandas as pd
snapshot_id = 0
snapshots = pd.read_csv("%s/ln_edges.csv" % data_dir)
directed_edges = snapshots[snapshots["snapshot_id"]==snapshot_id]
```

Note that the preprocessed data format is identical to the output of the `preprocess_json_file` function.

Merchants

We provided the list of LN merchants that we used in our experiments. This merchant information was collected in early 2019.

```
import pandas as pd
node_meta = pd.read_csv("%s/1ml_meta_data.csv" % data_dir)
providers = list(node_meta["pub_key"])
```

Configuration

First we give you the list of main parameters. **By the word "transaction" we refer to LN payments.**

You can initialize the traffic simulator by providing the network structure, merchants information and the former parameters.

```
import lnsimulator.simulator.transaction_simulator as ts

amount = 60000
count = 7000
epsilon = 0.8
drop_disabled = True
drop_low_cap = True
with_depletion = True

simulator = ts.TransactionSimulator(directed_edges, providers, amount, count, drop_
↪disabled=drop_disabled, drop_low_cap=drop_low_cap, eps=epsilon, with_depletion=with_
↪depletion)
```

Estimating daily income and traffic

i.) Transactions

The simulator samples a set of random transactions (during the initialization) that will be used for the estimation of daily node traffic and routing income. You can access the sampled transactions in the form of a `pandas.DataFrame`.

```
transactions = simulator.transactions
print(transactions.head())
print(transactions.shape)
```

ii.) Run simulation

In this step the simulator searches for cheapest payment paths from transaction senders to its receivers. Channel capacity changes are well maintained during the simulation.

```
cheapest_paths, _, all_router_fees, _ = simulator.simulate(weight="total_fee", with_
↪node_removals=False)
print(all_router_fees.head())
```

iii.) Results

After payment simulation you can export the results as well as calculate traffic and income statistics for LN nodes.

```
output_dir = "test"
total_income, total_fee = simulator.export(output_dir)
```

In order to get stable daily LN node statistics, we recommend to run the simulator for multiple times over several consecutive snapshots. **Node statistics in each output file below are restricted to a single traffic simulator experiment!** You can find these file in the `output_dir` folder.

a.) lengths_distrib.csv

Distribution of payment path length for the sampled transactions. Due to the source routing nature of LN, we assumed that transactions are executed on the cheapest path between the sender and the recipient.

Note: the length is marked -1 if the payment failed (there was no available path for routing)

Note: the sum of transactions in the second column could be less then the predefined number of payments to simulate. The difference is the number of randomly sampled loop transactions with identical sender and recipient node.

b.) router_incomes.csv

Contains statistics on nodes that forwarded payments in the simulation. We refer to these nodes as **routers**.

c.) source_fees.csv

Contains statistics on payment initiator nodes (senders).

Useful function calls

There are alternative ways to interact with the simulator object beside exporting the results (with the `simulator.export(output_dir)` function). Please follow the examples below.

Top nodes with highest daily income

You can search for the identity of these nodes on 1ml.com.

```
total_income.sort_values("fee", ascending=False).set_index("node").head(5)
```

Top nodes with highest daily traffic

```
total_income.sort_values("num_trans", ascending=False).set_index("node").head(5)
```

Payment path length distribution

Note: the length is marked -1 if the payment failed (there was no available path for routing)

```
cheapest_paths["length"].value_counts()
```

Payment succes ratio

```
(cheapest_paths["length"] > -1).value_counts() / len(cheapest_paths)
```

Payment cost statistics

```
cheapest_paths["original_cost"].describe()
```

Most frequent payment receivers

```
simulator.transactions["target"].value_counts().head(5)
```

Number of unique payment senders and receivers

```
simulator.transactions["source"].nunique(), simulator.transactions["target"].nunique()
```

1.2.3 Advanced Documentation

In the steps below we show you some specific use cases of our LN traffic simulator.

We suppose that you have already

- installed `lnsimulator` and downloaded the related LN data. If it is not the case then should follow the instructions in the *Getting Started* section first.
- understood the data preparation steps needed for payment simulation. We refer you to the *Basic Documentation* if you missed this step.

Preparation

Quickly execute all steps (e.g. imports, loading channel and merchant data) needed before advanced experiments

```
from lnsimulator.ln_utils import preprocess_json_file
import lnsimulator.simulator.transaction_simulator as ts

data_dir = "../ln_data/" # path to the ln_data folder that contains the downloaded␣
↪data
directed_edges = preprocess_json_file("%s/sample.json" % data_dir)

import pandas as pd
node_meta = pd.read_csv("%s/1ml_meta_data.csv" % data_dir)
providers = list(node_meta["pub_key"])

# the number of simulated payments and the payment value is fixed in this notebook

amount = 60000
count = 7000
```

Parameters explained

Here is the list of main parameters. **By the word "transaction" we refer to LN payments.**

The following examples will help you to understand the effect of each parameter. As `amount` and `count` are very straightforward parameters we will start with how to set merchant ratio for payment receivers.

Merchant ratio for payment receivers

The number of unique receivers is the highest when **receivers sampled uniformly at random** (`epsilon=0.0`) while you have the chance to send payments **only to merchants** (`epsilon=1.0`). In most of our experiments we sample merchant receivers with high probability (`epsilon=0.8`) but we also select random receivers as well with small probability.

```
only_merchant_receivers = ts.TransactionSimulator(directed_edges, providers, amount,␣
↪count, epsilon=1.0)
many_merchant_receivers = ts.TransactionSimulator(directed_edges, providers, amount,␣
↪count, epsilon=0.8)
uniform_receivers = ts.TransactionSimulator(directed_edges, providers, amount, count,␣
↪epsilon=0.0)
print(only_merchant_receivers.transactions["target"].nunique())
print(many_merchant_receivers.transactions["target"].nunique())
print(uniform_receivers.transactions["target"].nunique())
```

Control channel exclusion with `drop_disabled` and `drop_low_cap`

Channels can be temporarily disabled for a given snapshot while active for others. If you want to **enable disabled channels** in your experiments then use `drop_disabled=False`. But

```
default_sim = ts.TransactionSimulator(directed_edges, providers, amount, count, drop_
↪disabled=True, drop_low_cap=True)
with_disabled_sim = ts.TransactionSimulator(directed_edges, providers, amount, count,␣
↪drop_disabled=False, drop_low_cap=True)
print(default_sim.edges.shape)
print(with_disabled_sim.edges.shape)
```

A payment can only be forwarded on a given channel if the channel capacity is at least the value of the payment (`drop_low_cap=True`). But in the simulation you have the possibility to disabled this condition (`drop_low_cap=False`).

```
with_lowcap_sim = ts.TransactionSimulator(directed_edges, providers, amount, count,␣
↪drop_disabled=False, drop_low_cap=False)
print(with_lowcap_sim.edges.shape)
```

Updating node balances with payments

Individual balances of LN nodes is a private data but `lnsimulator` can **keep track of capacity imbalances** (`with_depletion=True`) as payments are executed on the fly. After distributing capacities randomly between related channel endpoints (initialization step), our simulator can monitor whether a node has enough outbound capacity on a given channel to forward the upcoming payment with respect to the payment value. This feature has several advantages:

- ability to detect node capacity depletions in case of heavy one-way traffic
- better understanding of payment failures

In case you disable this feature (`with_depletion=False`) payments can pass a channel in a fixed direction infinitely many times as long as the payment value is at most the channel capacity.

In the next example we observe the payment failure ratio with respect to the `with_depletio` parameter.

```
sim_with_dep = ts.TransactionSimulator(directed_edges, providers, amount, count, with_
↪depletion=True)
_, _, _, _ = sim_with_dep.simulate(weight="total_fee")


sim_wout_dep = ts.TransactionSimulator(directed_edges, providers, amount, count, with_
↪depletion=False)
_, _, _, _ = sim_wout_dep.simulate(weight="total_fee")
```

Transaction success rates are lower if capacity depletion is enabled (`with_depletion=True`). This indicates that **there are channels with heavy one-way traffic**.

```
print("Succes rate with depletions:", sim_with_dep.transactions["success"].mean())
print("Succes rate without depletions:", sim_wout_dep.transactions["success"].mean())
```

Advanced simulation features

In the past experiment after initializing your simulator the `simulate()` function executed cheapest path routing by default without modifying the available channel data. Now let's see some additional use cases.

```
sim = ts.TransactionSimulator(directed_edges, providers, amount, count)
```

Routing algorithm

For now you can choose between two routing algorithms by setting the `weight` parameter

- **cheapest path** routing (`weight="total_fee"` - DEFAULT SETTING)
- **shortest path** routing (`weight=None`)

```
shortest_paths, _, _, _ = sim.simulate(weight=None)
cheapest_paths, _, _, _ = sim.simulate(weight="total_fee")
```

Filter out payments that could not be routed (they are denoted with `length==-1`)

Then observe the average path length for the simulated payments

```
print(shortest_paths[shortest_paths["length"]>0]["length"].mean())
print(cheapest_paths[cheapest_paths["length"]>0]["length"].mean())
```

Node removal

You can observe the effects of node removals as well by providing a list of LN node public keys. In this case every channel adjacent to the given nodes will be removed during payment simulation.

In this example we exclude the top 5 nodes with highest routing income

```
_, _, all_router_fees, _ = sim.simulate(weight="total_fee")
print("Succes rate BEFORE exclusion:", sim.transactions["success"].mean())

top_5_stats = all_router_fees.groupby("node")["fee"].sum().sort_
↪values(ascending=False).head(5)
print(top_5_stats)
top_5_nodes = list(top_5_stats.index)
```

You can observe how the payment success rate dropped by removing 5 important routers

```
_, _, _, _ = sim.simulate(weight="total_fee", excluded=top_5_nodes)
print("Succes rate AFTER exclusion:", sim.transactions["success"].mean())
```

Node capacity reduction

You can observe the traffic changes for a node by reducing its capacity to a given fraction of its original value.

In this example we reduce capacity to 10% of the top 5 nodes with highest routing income. Then we compare their new income with the original.

```
_, _, reduced_fees, _ = sim.simulate(weight="total_fee", cap_change_nodes=top_5_nodes,
↪ capacity_fraction=0.1)
print("Succes rate AFTER capacity reduction:", sim.transactions["success"].mean())

new_stats = reduced_fees.groupby("node")["fee"].sum()
old_and_new = top_5_stats.reset_index().merge(new_stats.reset_index(), on="node", how=
↪"left", suffixes=("_old","_new"))
print(old_and_new.fillna(0.0))
```

Longer path (genetic) routing

In our paper we proposed a genetic algorithm to find cheap paths with at least a given length (`required_length` parameter). By default genetic routing is disabled (`required_length=None`).

We note that..

- using a higher value for `required_length` could increase the running time significantly.

- payment paths with length 1 (direct channels) won't be forced to longer as with zero intermediary node there is no privacy issue here
- if the genetic algorithm cannot find a path with the required length then it will return a path that is lower than this bound

In this example we will observe the path length distribution for different values of the `required_length` parameter

```
sim_for_routing = ts.TransactionSimulator(directed_edges, providers, amount, count)

min_path_l = [2,3,4]
length_distrib_map = {}

for length_value in min_path_l:
    cheapest_paths, _, _, _ = sim.simulate(weight="total_fee", required_length=length_
↪value)
    length_distrib_map[length_value] = cheapest_paths["length"].value_counts()
    print(length_value)
```

Observe the fraction of path with a given length (rows). The columns represent values of the `required_length` parameter. **The fraction of path in (3,3) and (4,4) cell of the heatmap are indeed high due to longer path routing.** We note that the row with index -1 represent the fraction of failed payments.

```
import seaborn as sns

distrib_df = pd.DataFrame(length_distrib_map)
distrib_df = distrib_df / distrib_df.sum()
sns.heatmap(distrib_df.loc[[-1,1,2,3,4]], cmap="coolwarm", annot=True)
```

Base fee optimization

In the Lightning Network data that we observed more than 60% of the nodes charged the default base fee. From a node's position in the network `lnsimulator` can estimate the base fee increment needed to achieve optimal routing by setting `with_node_removals=True` and calling `calc_optimal_base_fee` function afterwards. **For now optimal base fee search is enabled only for cheapest path routing (`weight="total_fee"`). We also recommend you apply parallelization by setting a higher value for the `max_threads` parameter.**

```
sim_fee_opt = ts.TransactionSimulator(directed_edges, providers, amount, count)
shortest_paths, alternative_paths, all_router_fees, _ = sim_fee_opt.simulate(weight=
↪"total_fee", with_node_removals=True, max_threads=2)
opt_fee_df, _ = ts.calc_optimal_base_fee(shortest_paths, alternative_paths, all_
↪router_fees)
print(opt_fee_df.head())
```

The result of `calc_optimal_base_fee` contains the following informations.

In the next step we transform the `opt_delta` column into a categorical feature that represent the increment magnitude.

```
def to_category(x):
    if x > 100:
        return 3
    elif x > 10:
        return 2
    elif x > 0:
        return 1
```

(continues on next page)

(continued from previous page)

```
    else:
        return 0

print("The magnitude distribution of base fee increments:")
print(opt_fee_df["opt_delta"].apply(to_category).value_counts())
```

1.2.4 Acknowledgements

To Antoine Le Calvez (Coinmetrics) and Altangent Labs for kindly providing us their edge stream data and daily graph snapshots. To Domokos M. Kelen and Rene Pickhardt for insightful discussions. To our reviewers, Christian Decker, Cyril Grunspan and to our anonymous reviewer for their invaluable comments. Support from Project 2018-1.2.1-NKP-00008: Exploring the Mathematical Foundations of Artificial Intelligence and the “Big Data——Momentum” grant of the Hungarian Academy of Sciences.