CS765 Assignment 3
Building a layer-2 Dapp on top of Blockchain

Sanchit Jindal (200020120), Sarthak Mittal (200050129), Virendra Kabra (200050157)

Spring 2023

Contents

1 Overview

In this assignment, we build a Decentralized application (Dapp) over a peer-to-peer decentralized network. We simulate the Ethereum network protocol that allows users to create and run smart contracts over the network. The simulation of transactions is done using joint accounts over a user network graph where transactions cause a decrement of the balance of the sender and an increment of the balance of the receiver in their joint account, and this propagates throughout the path that is determined for the transaction. The transaction's validity depends on the balance availability across all the joint accounts on the path.

2 Implementation

2.1 Contract (Solidity)

- For each node, an array of edges is maintained. The `Edge` structure is defined to hold the id of the adjoining node, and balances in the joint account.
- Functions `registerUser`, `createAcc` and `closeAccount` are defined as described in the problem statement. `sendAmount` is used to send a specified amount across an edge. The caller (client) determines a path (which is a compute-intensive task) and calls this function for each edge in that path.
- A helper function, `getEdges`, returns an adjacency matrix `arr` filled with balances in joint accounts - `arr[i][j]` is the balance of node `i` in the joint account between `i` and `j` (-1 if there is no joint account). This is used to fetch the graph state.

2.2 Client (Python)

- Functions similar to the above are used to register users and create/close accounts.
- We used `networkx` module to create a connected graph following the power-law degree distribution.
- The initial joint balance is sampled from an exponential distribution with a mean of 10. The least integral value ($\geq$ sample) divisible by two is used and divided equally between the two nodes.
- `sendAmount` is used to transfer `amount` between two users. It first finds some shortest path using BFS on the fetched graph (if it exists). This can be tweaked to get the shortest path with sufficient funds along the edges. Having found the path, `sendAmount` (from the contract) is called for edges in the path.
- In the required simulation, the graph is only modified with `sendAmount` after the initial account setup. So, we maintain the graph state in the client as well, to avoid fetching the adjacency matrix each time.

3 Observations

- We observe that the algorithm choosing a path with sufficient funds gives a success fraction close to 1.
- In our experiments, the network is connected (ensured in accordance with the problem statement). Further, it has around 500 edges (joint accounts), each starting with a total balance sampled from an exponential distribution with a mean of 10.
- Since the payments are with `amount=1` and between random nodes, we can expect the existence of such paths. This is confirmed in the plots.
- The intra-plot fraction remains almost constant. This can be attributed to the randomized picking of nodes for payment. On average, we don't expect a concentration of funds with a single node (and scarcity for others).

4 Results

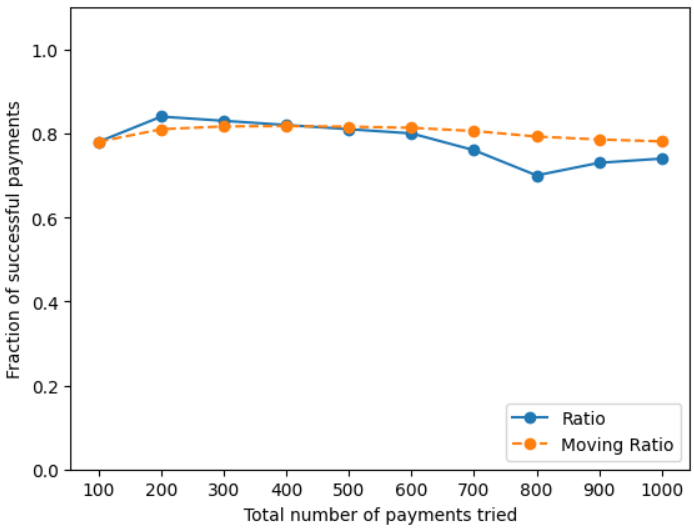

Figure 1: Fraction of successful payments with some shortest path

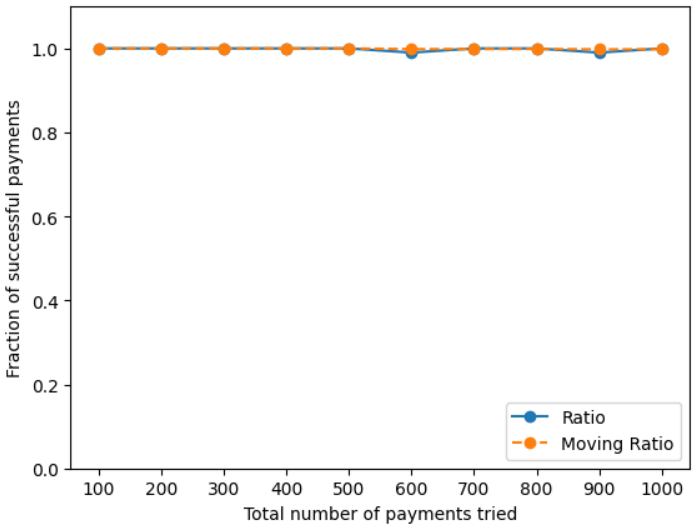

Figure 2: Fraction of successful payments with some shortest path having sufficient funds