BLOCKCHAIN PROJECT (CS-540)

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT,

INDIAN INSTITUTE OF TECHNOLOGY, ROPAR (IIT ROPAR)

Submitted by:

Pritam Mahajan	(2023AIM1012)
Abdul Razique	(2023CSM1001)
Ashish Kohli	(2023CSM1002)
Vaibhav Barsaiyan	(2023CSM1019)
Pradeep Dalal	(2023AIM1008)

1) **Abstract**:-

Blockchain-based cryptocurrencies are severely limited in transaction throughput and latency due to the need to seek consensus among all peers of the network. A promising solution to this issue is payment channels, which allow unlimited numbers of atomic and trust-free payments between two peers without exhausting the resources of the blockchain. A linked payment channel network enables payments between two peers without direct channels through a series of intermediate nodes that forward and charge for the transactions. However, the charging strategies of intermediate nodes vary with different payment channel networks. Existing works do not yet have a complete routing algorithm to provide the most economical path for users in a multi-charge payment channel network. In this work, we propose MPCN-RP, a general routing protocol for payment channel networks with multiple charges. Our extensive experimental results on both simulated and real payment channel networks show that MPCN-RP significantly outperforms the baseline algorithms in terms of time and fees.

2) **Introduction**:-

Blockchain networks like Bitcoin and Ethereum face challenges regarding transaction throughput and latency due to the consensus mechanism. This mechanism requires agreement among all peers in the network, leading to slower transaction processing times and limited scalability.

The Payment channel and Linked multiple channel network. MPCN-RP (Multi-charge Payment Channel Network Routing Protocol) is a proposed routing protocol designed to address the shortcomings of existing algorithms. It aims to find the most cost-effective path for transactions in networks with multiple charging nodes. The protocol likely considers factors such as node fees, network topology, and transaction requirements to determine the optimal route.

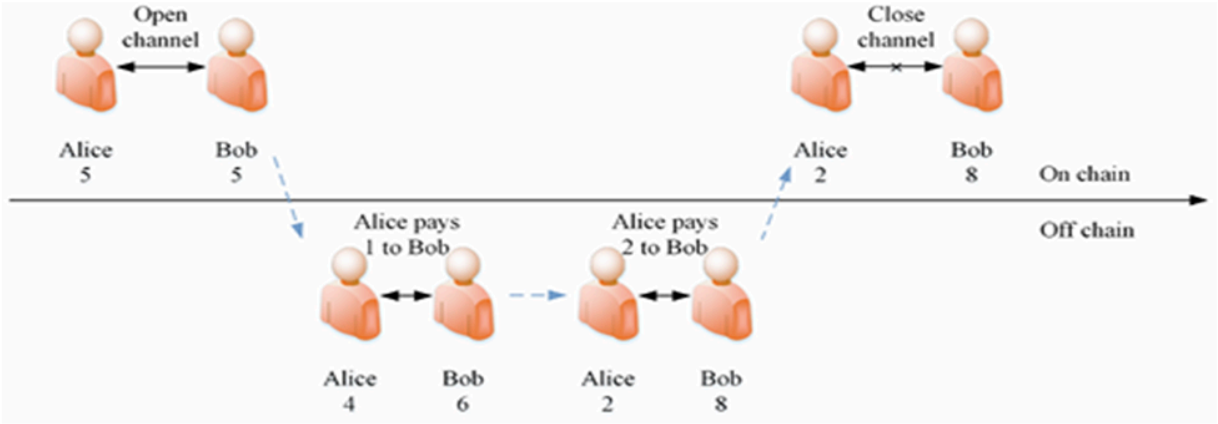
2.1) Payment channel network

Payment channel network also known as state channels, enable off-chain transactions between two or more participants. These participants can conduct multiple transactions the Layer 1 blockchain for each operation.

Benefits: Payment channels reduce transaction costs and increase throughput by enabling instant and low-fee transactions. Example : The Lightning Network is a payment channel network.

Payment channels are the foundation of off-chain expansion solutions, but they are not enough as a user cannot create a channel with everyone he wants to trade. To address this problem, Lightning Network introduced Hashed Time-Lock Contract (HTLC), which connects two nodes that are not directly connected through a series of end-to-end payment channels. HTLC allows the sender and the recipient to handle the payments through a series of intermediate nodes (INs) that charge fees for forwarding the money. A network containing all peer nodes and payment channels is called a Payment Channel Network (PCN).

Payment channels are the foundation of off-chain expansion solutions, but they are not enough as a user cannot create a channel with everyone he wants to trade. To address this problem, Lightning Network introduced Hashed Time-Lock Contract (HTLC), which connects two nodes that are not directly connected through a series of end-to-end payment channels. HTLC allows the sender and the recipient to handle the payments through a series of intermediate nodes (INs) that charge fees for forwarding the money. A network containing all peer nodes and payment channels is called a Payment Channel Network (PCN).



3) Methodology:-

Cloth simulator:-

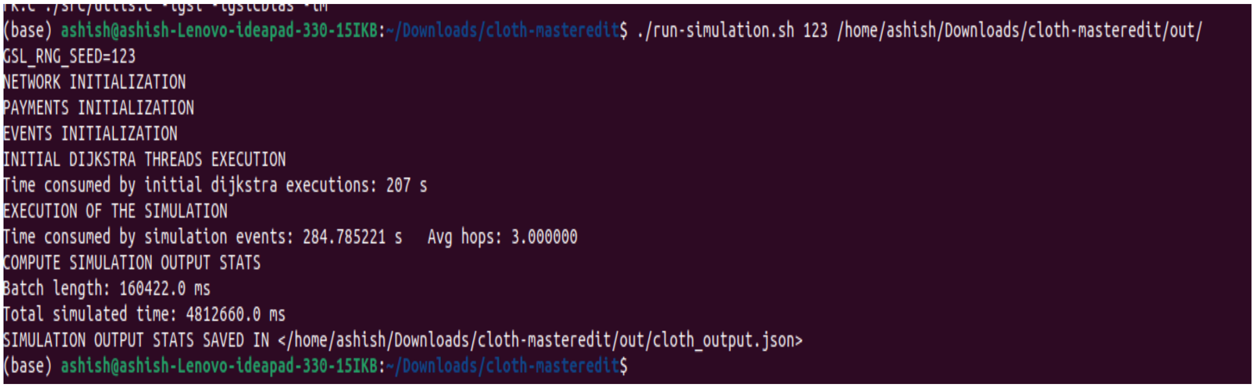
We have used a cloth simulator for the payment channel network for implementing and modifying the paper. Cloth simulator is majorly written in C language and python scripting is used for compiling, running and writing or printing the outputs. For different tasks, the simulator manages different files like cloth.c, event.c…

Cloth.c is the main file which first of all take input and store it in parameter structure after it calls the network.c file to build the network, further cloth.c file paymet.c file to list all the payments in payments structure sorted by starting time, then it calls the event file to build the events like FINDPATH, SENDPAYMENT, FORWARDPAYMENT, RECEIVEPAYMENT, etc., after that the simulation starts and for FINDPATH event uses dijkstra to find the path, calculation of dijkstra uses heap for and this heap is maintained by heap.c

Run Simulator:-

Step 1:- make build

Step2 :- /run-simulation.sh 1243/home/ashish/Downloads/cloth-master/out/



```
rk.c ./src/utils.c -lgsl -lgslcblas -lm
(base) ashish@ashish-Lenovo-ideapad-330-15IKB:~/Downloads/cloth-masteredit$ ./run-simulation.sh 123 /home/ashish/Downloads/cloth-masteredit/out/
GSL_RNG_SEED=123
NETWORK INITIALIZATION
PAYMENTS INITIALIZATION
EVENTS INITIALIZATION
INITIAL DIJKSTRA THREADS EXECUTION
Time consumed by initial dijkstra executions: 207 s
EXECUTION OF THE SIMULATION
Time consumed by simulation events: 284.785221 s   Avg hops: 3.000000
COMPUTE SIMULATION OUTPUT STATS
Batch length: 160422.0 ms
Total simulated time: 4812660.0 ms
SIMULATION OUTPUT STATS SAVED IN </home/ashish/Downloads/cloth-masteredit/out/cloth_output.json>
(base) ashish@ashish-Lenovo-ideapad-330-15IKB:~/Downloads/cloth-masteredit$
```

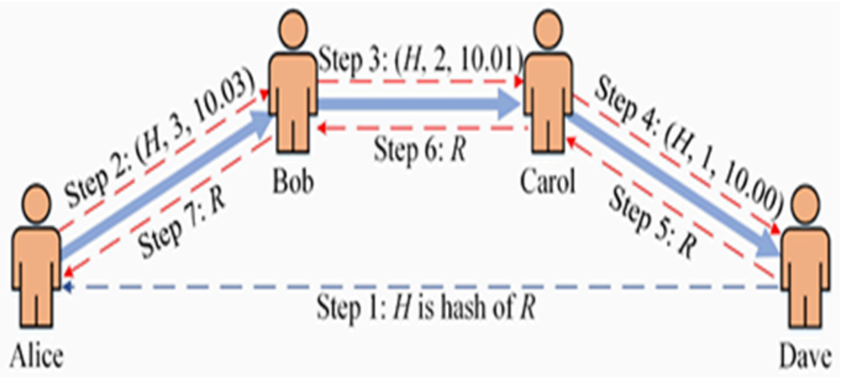
4) Implementing Paper:-

Algorithm Design of MPCN-RP:-

1. Payment Request
2. Path Finding
3. Payment Execution
4. Update

A. SYSTEM MODEL:-

1. Payment Channel Network
2. Hashed Time Lock Contract



<u>HTLC(Hash Time Locked Contracts) :-</u>

These are a kind of smart contract that is utilized in payment channel networks such as the Lightning Network. HTLCs make it possible for parties who might not completely trust one another to trade securely and without risk. This is how it operates:

1. <u>Hash Locking:-</u> Using a distinct cryptographic hash that they create, the sender locks the cash. Only by disclosing the pre-image,the original data that creates the hash and hence can the receiver make a claim for the money.

2. <u>Time Locking:-</u> The amount of time allotted to claim the monies is also limited. The sender may recover the payments if the receiver does not claim them within the allotted period. HTLCs guarantee that the money is returned to the sender within a predetermined amount of time, or the transaction is properly completed. HTLCs also ensure that either the transaction is completed successfully, or the funds are returned to the sender after a specified time, providing security and reliability in off-chain transactions.

B. Network Model

- Fixed fee:

$$F^{fix} = \sum_{i=1}^{n} f_i^{fix}.$$

- Time value fee:

$$F^{tv} = \sum_{i=1}^{n} f_i^{tv}.$$

- Proportional fee:

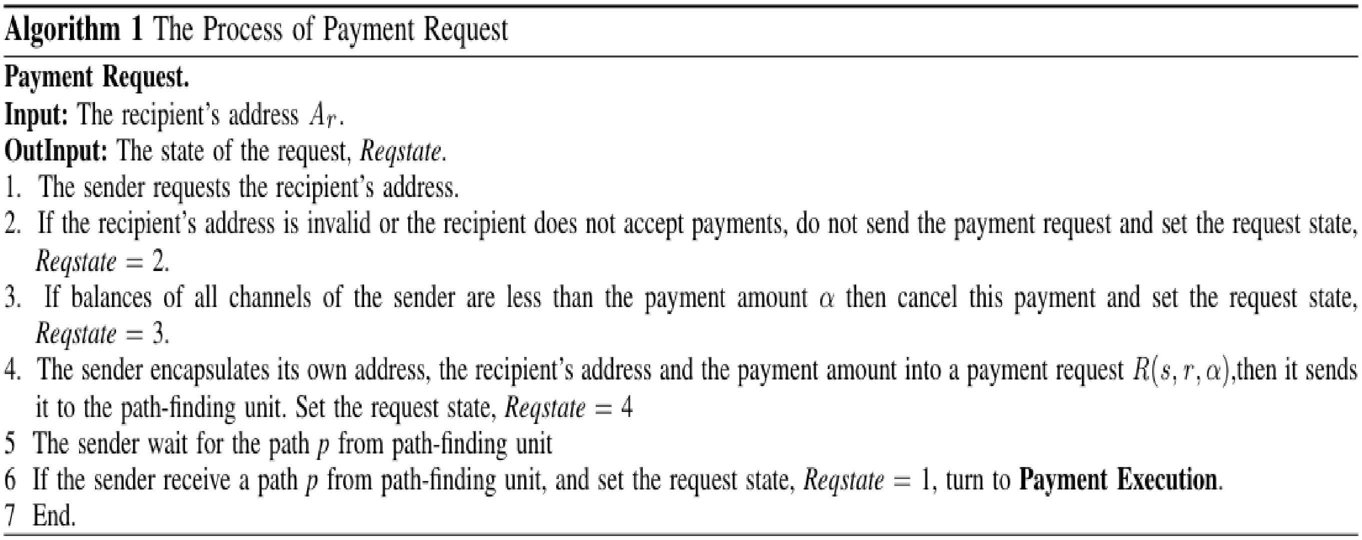
$$F^{prop} = \sum_{i=1}^{n} f_i^{prop}.$$

- Imbalance fee:

$$F^{im} = \sum_{i=1}^{n} f_i^{im},$$

- Total fee:-

$$F = F^{fix} + F^{prop} + F^{tv} + F^{im}.$$



Algorithm 1 The Process of Payment Request

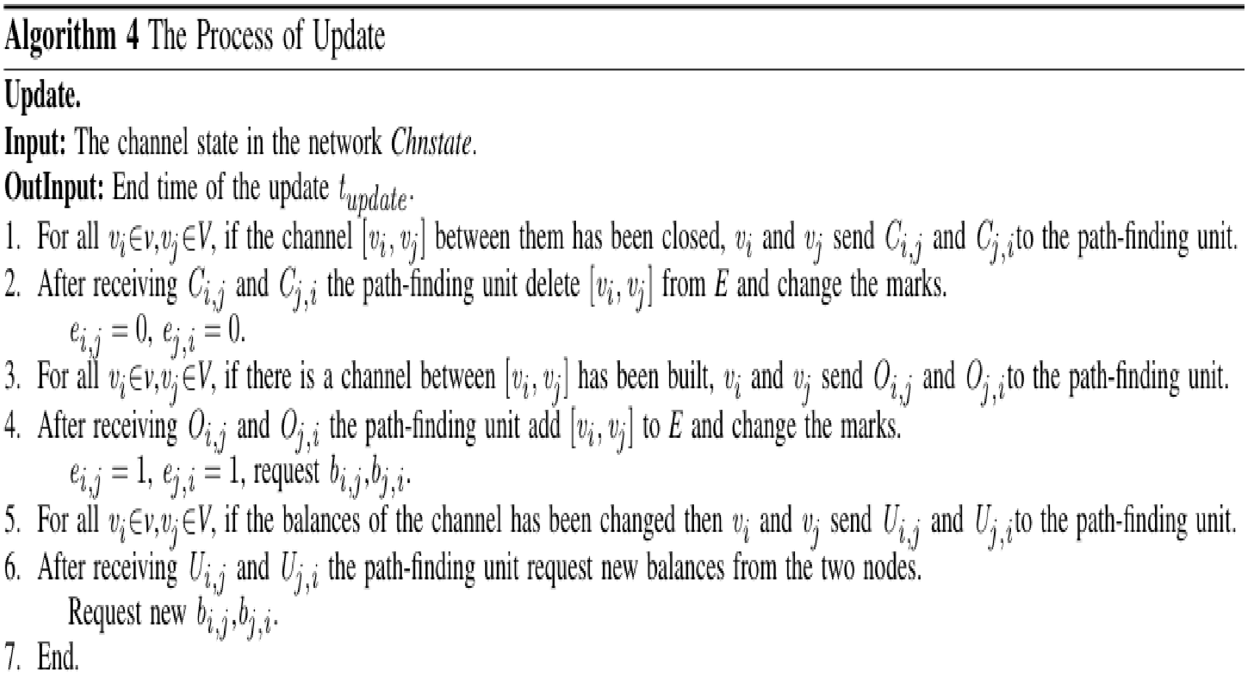
```
Payment Request.
Input: The recipient's address $A_r$.
OutInput: The state of the request, Reqstate.
1. The sender requests the recipient's address.
2. If the recipient's address is invalid or the recipient does not accept payments, do not send the payment request and set the request state, $Reqstate = 2$.
3. If balances of all channels of the sender are less than the payment amount $\alpha$ then cancel this payment and set the request state, $Reqstate = 3$.
4. The sender encapsulates its own address, the recipient's address and the payment amount into a payment request $R(s, r, \alpha)$,then it sends it to the path-finding unit. Set the request state, $Reqstate = 4$
5 The sender wait for the path $p$ from path-finding unit
6 If the sender receive a path $p$ from path-finding unit, and set the request state, $Reqstate = 1$, turn to Payment Execution.
7 End.
```

Algorithm 2 The Process of Path-Finding

```
Path-finding.
Input: Addresses of the sender $A_s$ and the recipient $A_r$.
OutInput: The path $p$.
1.  Set the recipient as the starting node $v_s$ and the sender as the destination node $v_r$.
2.  Initialize the distance from $v_s$, and the hop number of all nodes. If $[v_i,v_s]\in E$, $dis_i = 0$, $hop_i = 0$. Else, $dis_i = +\infty$, $hop_i = 0$
3.  Initialize the set $VIS$ of nodes which the shortest path has been found and the set $UVIS$ of nodes which the shortest path has not been found.
        $VIS = \phi$, $UVIS = V$
4.  Move the node with minimum $dis$ $v_m$ from $UVIS$ to $VIS$.
5.  For every node $v_i\in VIS$, if $\exists v_u$, $[v_i,v_u]\in E$ and $dis_u > dis_i+f_u$, then
        update $dis_u=dis_i+f_u$.
        update the hop number and previous node of node $v_u$, $hop_u++$, $prep_u = v_i$.
7   If $VIS! = V$ and $v_s \notin VIS$, then return to step 4.
8   Initialize the path $p$ and current node of the path.
        $p=\{v_s\}$, $v_{cur}=v_s$.
9   Set the previous node of the current node as the current node, $v_{cur}=pre_{cur}$, add $pre_{cur}$ to p.
10  If $pre_i! = v_r$ and $pre_i! = null$, then send $p$ to the sender. Else, return to Step 9.
11  End.
```

Algorithm 3 The Process of Payment Execution

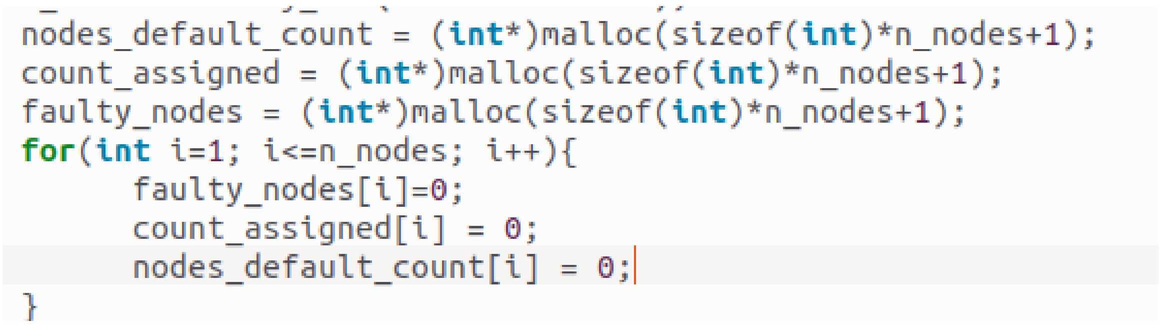
```
Payment Execution.
Input: Addresses of the sender $A_s$ and the recipient $A_r$.
OutInput: The state of the payment $Paystate$.
1.  Confirm the address of the recipient and set the current node $v_{cur} = v_r$. Set the state of the payment, $Paystate = 0$.
2.  Calculate the fee $f_{cur-1}$ that will be charged from the previous node to the current node.
3.  If the current node publishes H(R) to the previous node before the end of the time lock.
        $b_{cur-1,cur}$-$\alpha$-$f_{cur-1,cur}$, $b_{cur,cur-1}$+$\alpha$+$f_{cur-1,cur}$. Set the previous node of the current node as the current node, $v_{cur}=pre_{cur}$.
    Return to Step 2.
4.  Else, the payment fails. Set the state of the payment, $Paystate = 2$. Go to Step 6.
5.  Set the state of the payment, $Paystate = 1$.
6.  End.
```



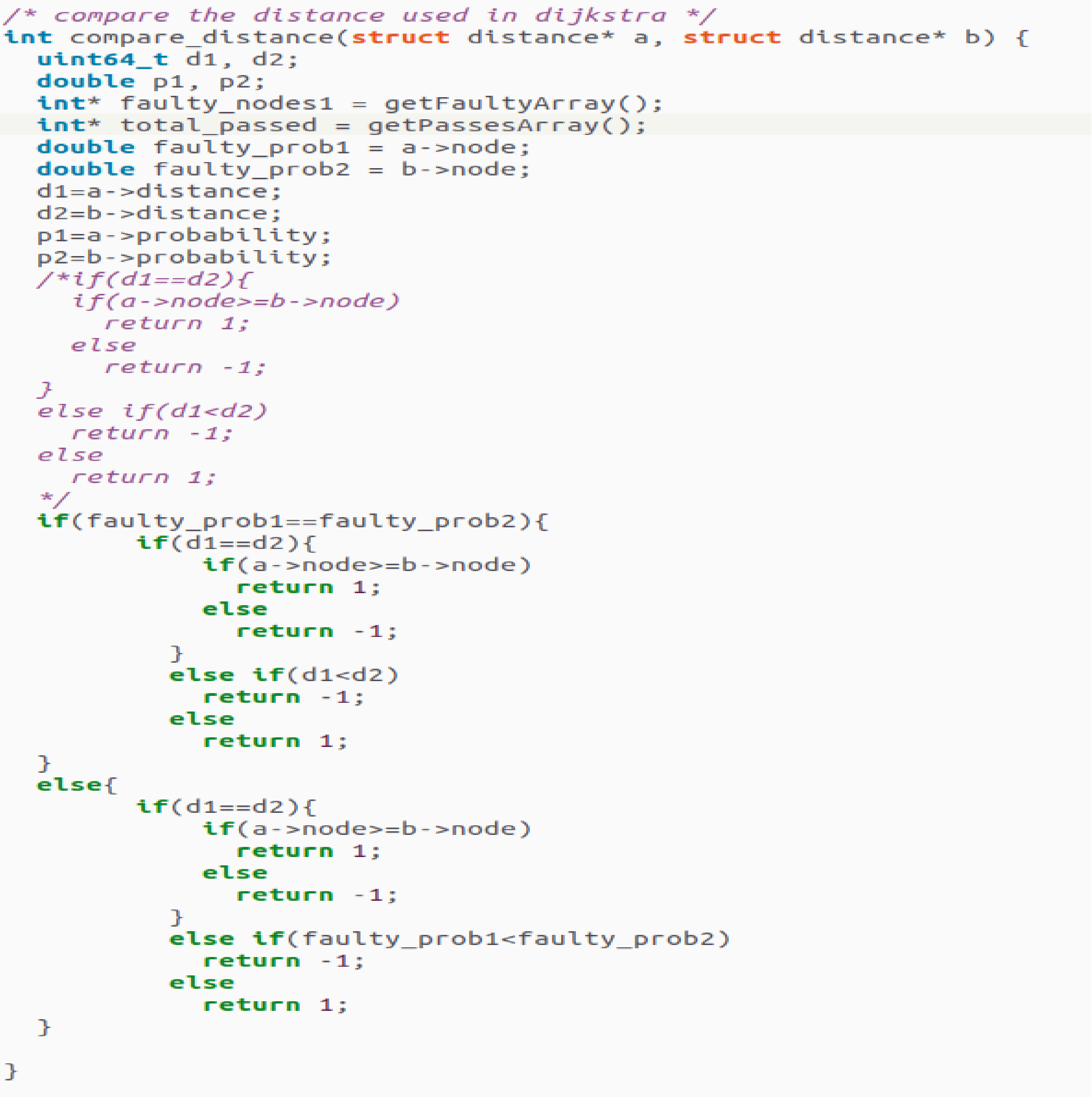
Algorithm 4 The Process of Update

```
Update.
Input: The channel state in the network Chnstate.
OutInput: End time of the update $t_{update}$.
1. For all $v_i \in v, v_j \in V$, if the channel $[v_i, v_j]$ between them has been closed, $v_i$ and $v_j$ send $C_{i,j}$ and $C_{j,i}$to the path-finding unit.
2. After receiving $C_{i,j}$ and $C_{j,i}$ the path-finding unit delete $[v_i, v_j]$ from $E$ and change the marks.
       $e_{i,j} = 0,\ e_{j,i} = 0$.
3. For all $v_i \in v, v_j \in V$, if there is a channel between $[v_i, v_j]$ has been built, $v_i$ and $v_j$ send $O_{i,j}$ and $O_{j,i}$to the path-finding unit.
4. After receiving $O_{i,j}$ and $O_{j,i}$ the path-finding unit add $[v_i, v_j]$ to $E$ and change the marks.
       $e_{i,j} = 1,\ e_{j,i} = 1$, request $b_{i,j}, b_{j,i}$.
5. For all $v_i \in v, v_j \in V$, if the balances of the channel has been changed then $v_i$ and $v_j$ send $U_{i,j}$ and $U_{j,i}$to the path-finding unit.
6. After receiving $U_{i,j}$ and $U_{j,i}$ the path-finding unit request new balances from the two nodes.
       Request new $b_{i,j}, b_{j,i}$.
7. End.
```

5) Proposed method:-

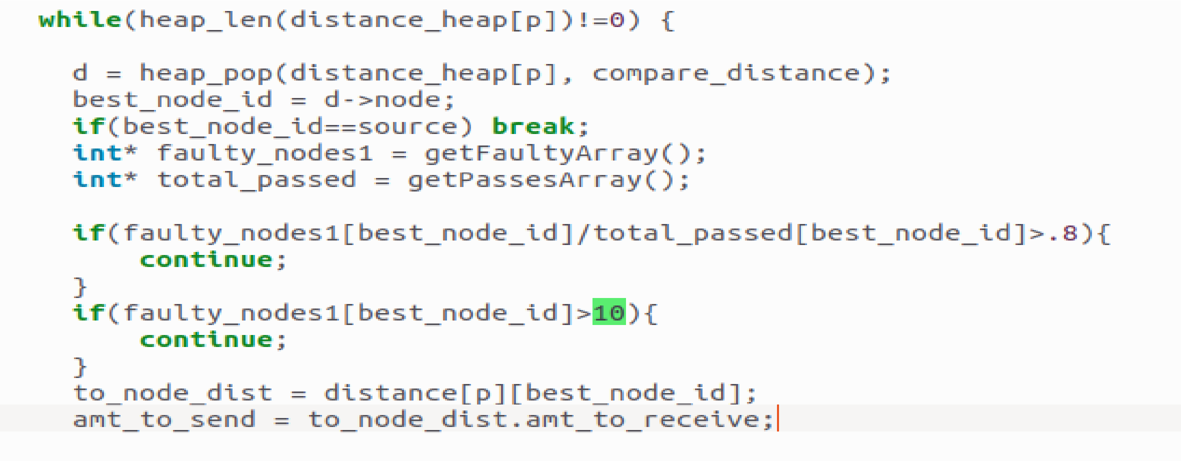


```
nodes_default_count = (int*)malloc(sizeof(int)*n_nodes+1);
count_assigned = (int*)malloc(sizeof(int)*n_nodes+1);
faulty_nodes = (int*)malloc(sizeof(int)*n_nodes+1);
for(int i=1; i<=n_nodes; i++){
      faulty_nodes[i]=0;
      count_assigned[i] = 0;
      nodes_default_count[i] = 0;
}
```



```
/* compare the distance used in dijkstra */
int compare_distance(struct distance* a, struct distance* b) {
  uint64_t d1, d2;
  double p1, p2;
  int* faulty_nodes1 = getFaultyArray();
  int* total_passed = getPassesArray();
  double faulty_prob1 = a->node;
  double faulty_prob2 = b->node;
  d1=a->distance;
  d2=b->distance;
  p1=a->probability;
  p2=b->probability;
  /*if(d1==d2){
    if(a->node>=b->node)
      return 1;
    else
      return -1;
  }
  else if(d1<d2)
    return -1;
  else
    return 1;
  */
  if(faulty_prob1==faulty_prob2){
        if(d1==d2){
            if(a->node>=b->node)
              return 1;
            else
              return -1;
          }
          else if(d1<d2)
            return -1;
          else
            return 1;
  }
  else{
        if(d1==d2){
            if(a->node>=b->node)
              return 1;
            else
              return -1;
          }
          else if(faulty_prob1<faulty_prob2)
            return -1;
          else
            return 1;
  }

}
```



```
  while(heap_len(distance_heap[p])!=0) {

    d = heap_pop(distance_heap[p], compare_distance);
    best_node_id = d->node;
    if(best_node_id==source) break;
    int* faulty_nodes1 = getFaultyArray();
    int* total_passed = getPassesArray();

    if(faulty_nodes1[best_node_id]/total_passed[best_node_id]>.8){
        continue;
    }
    if(faulty_nodes1[best_node_id]>10){
        continue;
    }
    to_node_dist = distance[p][best_node_id];
    amt_to_send = to_node_dist.amt_to_receive;
```

In our modified Dijkstra algorithm, the inclusion of faulty probability as a key factor in queue prioritization fundamentally shifts the focus from solely optimizing distance to considering the reliability of nodes. By calculating the faulty probability as the ratio of faulty transactions to total transactions, we’re effectively quantifying the trustworthiness of each node in the network. This allows the algorithm to prioritize exploring paths through nodes with lower probabilities of being faulty, thereby increasing the likelihood of selecting more reliable routes.

Furthermore, the incorporation of distance as a tiebreaker when nodes possess equal faulty probabilities ensures that the algorithm remains efficient and capable of finding optimal paths. In scenarios where nodes exhibit similar levels of reliability, prioritizing shorter distances helps maintain the algorithm's core objective of finding the shortest path from the source node to all other nodes in the network.

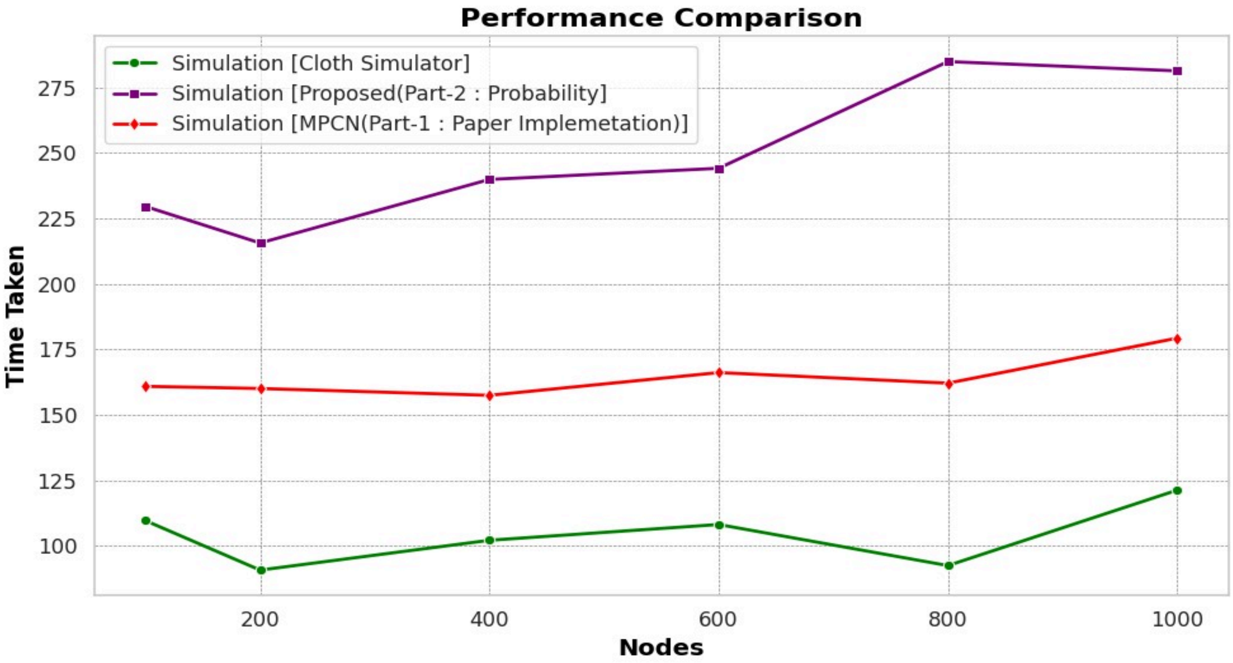
Overall, this hybrid approach combines the principles of fault tolerance and efficiency, making it well-suited for applications where both reliability and performance are critical considerations, such as in distributed systems, communication networks, or blockchain technologies.

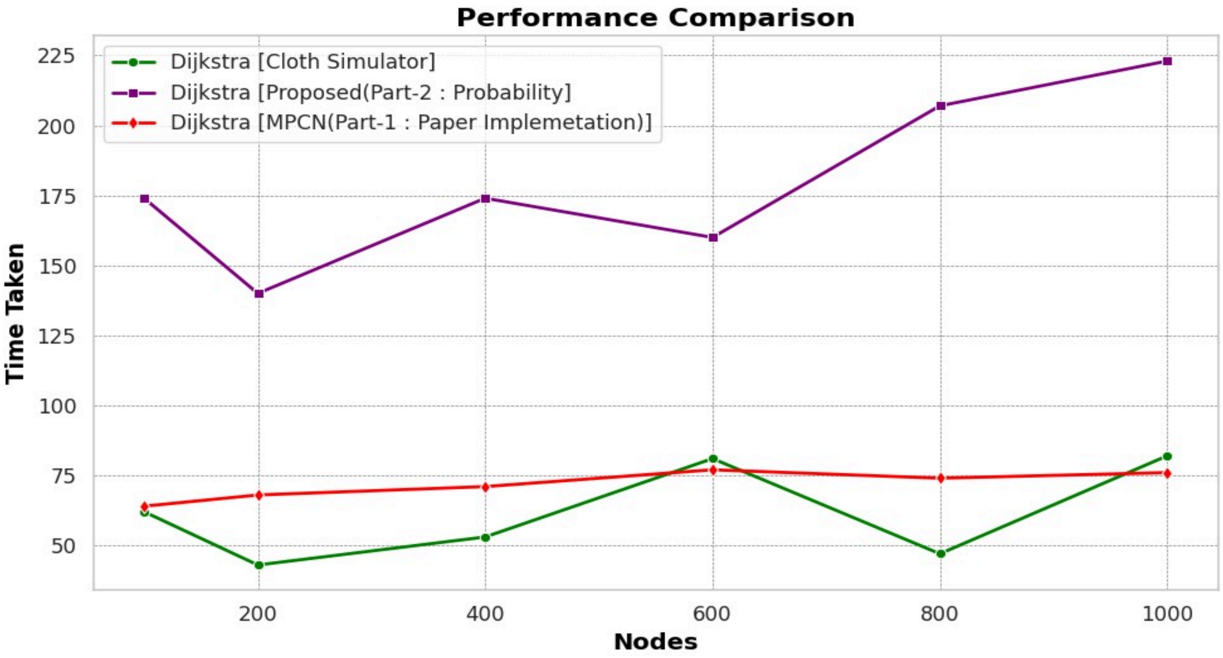
This allows the algorithm to prioritize exploring paths through nodes with lower probabilities of being faulty, thereby increasing the likelihood of selecting more reliable routes.

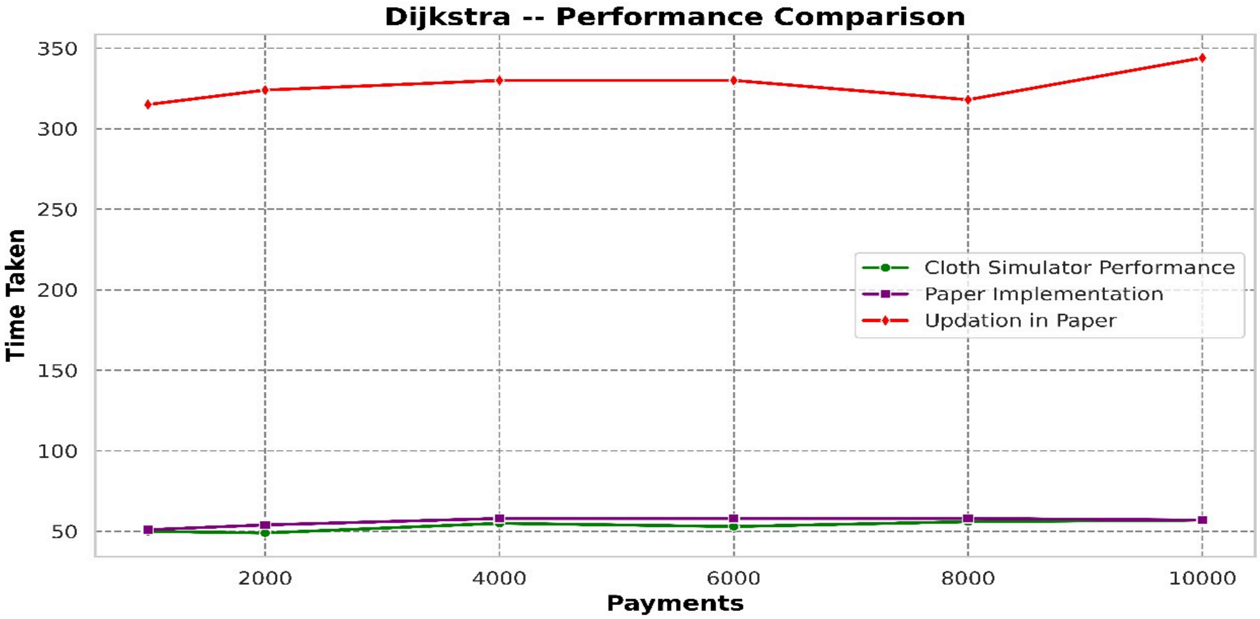
Furthermore, the incorporation of distance as a tiebreaker when nodes possess equal faulty probabilities ensures that the algorithm remains efficient and capable of finding optimal paths. In scenarios where nodes exhibit similar levels of reliability, prioritizing shorter distances helps maintain the algorithm's core objective of finding the shortest path from the source node to all other nodes in the network.

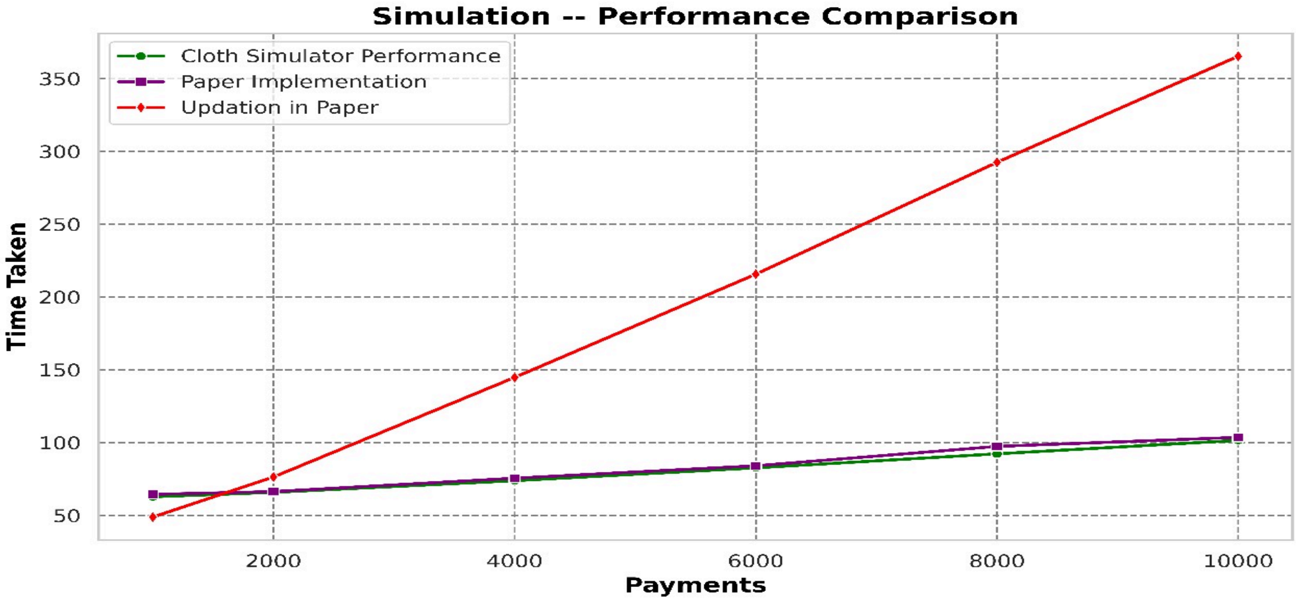
Overall, this hybrid approach combines the principles of fault tolerance and efficiency, making it well-suited for applications where both reliability and performance are critical considerations, such as in distributed systems, communication networks, or blockchain technologies.

6) Evaluation:-









Conclusion :-

Throughout this study, we have meticulously maintained an array to track the ratio of defaulted payment attempts to the total number of forwarding opportunities for each node. This meticulous tracking aims to efficiently allocate nodes for forwarding payments, ultimately enhancing the payment delivery ratio.

The graphical analyses presented above demonstrate that the implementation of MPCN, as outlined in the research paper, performs comparably to the existing method utilizing Dijkstra's algorithm. This similarity arises primarily due to the sole update in fee calculation, which does not significantly impact the simulation timing of Dijkstra's algorithm.

We have conducted a comprehensive analysis of both the paper implementation and the proposed method across various parameters, including the number of nodes and payments. The graphs depict that the proposed method exhibits slightly longer processing times compared to the other two approaches. This is attributed to the maintenance of an array storing default ratios for each node and the consequent frequent access to this array.

In conclusion, while the proposed method may require slightly more processing time due to the additional array management, its potential to enhance payment delivery ratio warrants further exploration and optimization.