Omnet Router Specification

January 6, 2019

1 Transaction Unit

Each transaction unit is a small amount of payment that is to be sent from one end-point to another. The route for a transaction unit is pre-specified by the sending end-point. Thus, each transaction unit contains 6 fields: amount, time sent, sender, receiver, route, priority class. A transaction unit should never have an amount that is larger than a pre-defined *Maximum Transaction Unit.* The route should have the sender and the receiver as the first and last entries. For starters, we don't assume deadlines on the completion of a transaction unit. As and when that is added, that can be added as a field. **Vibhaa:** Add deadlines/priority explanation

2 Flow of Transaction Units - Simplistic Model

2.1 Overview

The high level flow per transaction unit is as follows. Let's consider a transaction arriving at router A and meant to be sent out on the $A-B$ channel. When a transaction unit is received at A whose next hop is B, we first check if there is already a queue of transaction units waiting to be sent out to B. If so, we add this new transaction unit (let's say t) to it. If not, we check if there were sufficient funds available to send out t. If not, we add t to the queue and are done for this transaction unit. If there are funds available, we update the balances (decrement) on A's end for the $A-B$ channel. When B receives the same transaction unit, it increments its balance on the $A-B$ channel. Note that this naturally introduces a delay in the balance update because of "propagation delay" of the transaction unit. If this balance increment frees up any funds for further transaction units currently queued at B, they should be sent out on the $B-A$ channel.
Once B has completed the updates for the $A-B$ channel for the original transaction unit t, it looks at the next hop and repeats the procedure on the outgoing channel at B.
This is illustrated in the following flowchart.

2.2 Message Types

We only have one message corresponding to a transaction unit.

- Amount
- Time sent
- Original Sender
- Receiver
- Route
- Priority Class

2.3 State Per Router

This necessiates the following states at a router and the ability to update them

- Router state (Map to identify channels): key is neighboring router id (typically public key) and value is a unique channel index or id. The idea is that you would use this to identify the channel id when you have the route of a packet in terms of hops/routers it traverses. Once you have this id, you can use it to index into a table that contains every channel's state (described below).
- Per channel state:
  - Balance for oneself and for the other party
  - Queue of transaction units (in the above example, it would be a queue at A of transaction units it wanted to send to B but didn't have funds for)

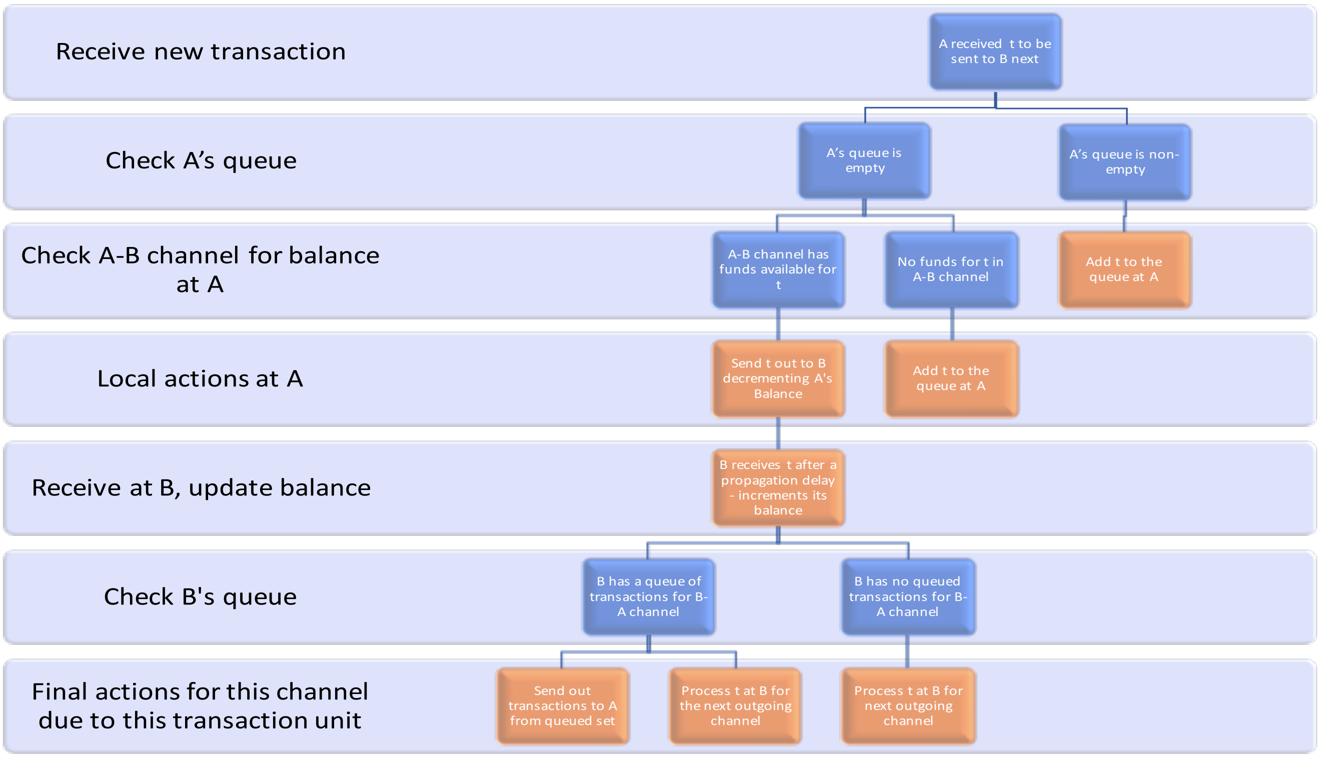
Receive new transaction
A received t to be sent to B next
Check A’s queue
A’s queue is empty
A’s queue is non-empty
Check A-B channel for balance at A
A-B channel has funds available for t
No funds for t in A-B channel
Add t to the queue at A
Local actions at A
Send t out to B decrementing A’s Balance
Add t to the queue at A
Receive at B, update balance
B receives t after a propagation delay - increments its balance
Check B's queue
B has a queue of transactions for B-A channel
B has no queued transactions for B-A channel
Final actions for this channel due to this transaction unit
Send out transactions to A from queued set
Process t at B for the next outgoing channel
Process t at B for next outgoing channel

Figure 1: flow of transactions at a router

2.4 Event Handlers for a Single Router

This follows from the flowchart above.

- Receipt of a transaction t:
  - Increment balance for self by t.amount on the incoming channel and decrement other party’s balance
  - Check if any transactions were queued for the incoming channel (in the opposite direction) that can now be sent out because of the increment in balance
  - Send as many as you can in priority order on that incoming channel
- Process transaction t:
  - Inspect next hop for t, map it to right outgoing channel id using the channel - channel id map per router
  - Check if there’s already transactions queued for that channel in the per channel queue
  - If there’s already queue just add t to the queue for the outgoing channel
  - If there isn’t a queue, check if we have sufficient funds for t. If not, again, queue t
- Send transaction t:
  - Decrement balance for outgoing channel and increment other party’s balance
  - Send message on the outgoing channel (assume some propagation delay after which it will be received there)

3 Flow of HTLCs - With Receiver Acknowledgements

3.1 Overview

Sending out HTLCs The high level flow per HTLC is as follows. Let’s consider a HTLC arriving at router A and meant to be sent out on the $A - B$ channel. When a HTLC is received at A whose next hop is B, we first check if there is already a queue of HTLCs waiting to be sent out to B. If so, we add this new HTLC (let’s say t) to it. If not, we check if there were sufficient funds available to send out t. If not, we add t to the queue and are done for this HTLC for now. If there are funds available, we update the balances (decrement) on A’s end for the $A - B$ channel. So far, this is similar to the previous simplistic model.

However, while this means that the HTLC has been attempted on this channel and is going out from A to B, it hasn't been cleared yet. So, we add t to a set of the inflight outgoing HTLCs which represents the set of HTLCs that have been attempted but not cleared yet. When B receives the same HTLC, it adds t to a set of inflight incoming HTLCs (that have been received but haven't been cleared yet). Note that this naturally introduces a delay in adding t to the HTLCs at B. This is because of "propagation delay" of the HTLC. During this time, the total capacity of the channel is a little less than the original capacity because only one node knows about the HTLC t (roughly similar to before). Once B has completed this, it looks at the next hop and repeats the procedure on the outgoing channel at B. B does nothing further on the $A-B$ channel for this HTLC until an *ack* is received.

Responding to Successful Acks Now, once the HTLC has been received at the destination node, this node will relay a successful *ack* backwards along the route of the transaction which will effectively clear the transactions from the "inflight HTLC" queues on individual routers for each channel. For instance, when B receives a successful ack for some HTLC t from a neighbor node C, it must first remove t from the set of inflight outgone HTLCs. It doesn't need to update any balance information since the balance was already decremented. B sends a balance update to C to tell C that it has reliably decremented the balance. (C can now remove t from its incoming pending HTLCs and increment its balance accordingly). B sends the ack to A which sent the HTLC in the first place to B. Once it has received a balance update from A for transaction t, B clears t from the set of inflight incoming HTLCs and increments its view of the $A - B$ balance accordingly.

Failure Acks (Can wait on this) Now, once a given node has decided to fail an HTLC, it can relay a failure *ack* backwards along the route of the transaction which will effectively clear the transactions from the "inflight HTLC" queues on individual routers for each channel all the way up. For instance, when B receives a failure ack for some HTLC t from a neighbor node C, it must first remove t from the set of inflight outgone HTLCs. It also needs to update its balance information since the balance was already decremented. B sends the ack to A which sent the HTLC in the first place to B. B also clears t from the set of inflight incoming HTLCs since it can't complete.
This is illustrated in the following flowchart.

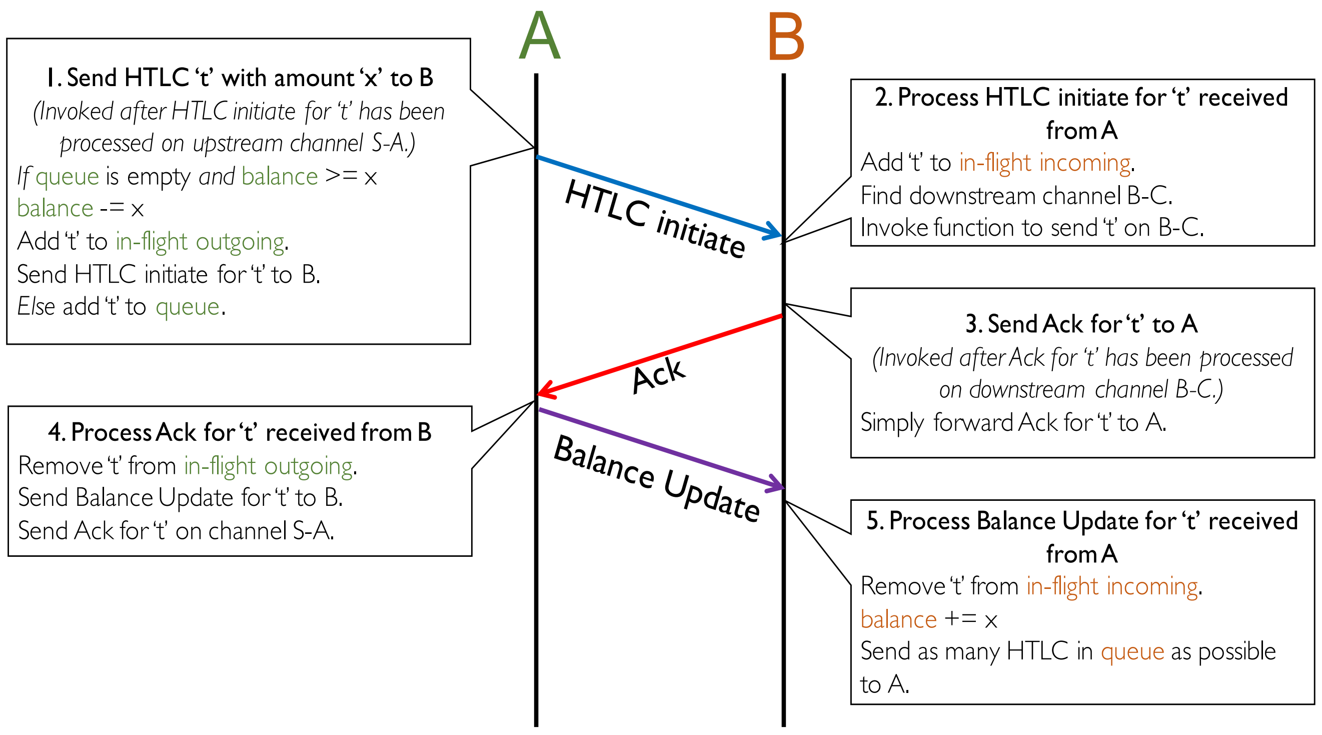
A
B
1. Send HTLC 't' with amount 'x' to B
(Invoked after HTLC initiate for 't' has been processed on upstream channel S-A.)
If queue is empty and balance >= x
balance -= x
Add 't' to in-flight outgoing.
Send HTLC initiate for 't' to B.
Else add 't' to queue.
2. Process HTLC initiate for 't' received from A
Add 't' to in-flight incoming.
Find downstream channel B-C.
Invoke function to send 't' on B-C.
HTLC initiate
3. Send Ack for 't' to A
(Invoked after Ack for 't' has been processed on downstream channel B-C.)
Simply forward Ack for 't' to A.
Ack
4. Process Ack for 't' received from B
Remove 't' from in-flight outgoing.
Send Balance Update for 't' to B.
Send Ack for 't' on channel S-A.
Balance Update
5. Process Balance Update for 't' received from A
Remove 't' from in-flight incoming.
balance += x
Send as many HTLC in queue as possible to A.

Figure 2: flow for a given HTLC t at a given channel

3.2 Message Types

- HTLC Initiate: This is similar to the current transaction message
  - Amount
  - Time sent
  - Original Sender

  - Receiver
  - Route
  - Priority Class
  - Transaction id (needed for removing it from the set of inflight transactions)

- HTLC Ack
  - Transaction id for which the Ack is received
  - Route for the ack (reverse of the path the transaction was sent on
  - Status (Success or Failure) - just in case we want to simulate failures and propagate them upwards
  - Secret - Dummy field for now (might never change)

- Balance Update
  - Transaction id that is being completed (Assuming that the inflight transaction set at each node has the amount already, this is all we really need to update the balances correctly)
  - Local Balance (The one who is sending the message)
  - Other Party's balance (the one the message is sent to)

3.3 State Per Router

This necessiates the following states at a router and the ability to update them

- Router state (Map to identify channels): key is neighboring router id (typically public key) and value is a unique channel index or id. The idea is that you would use this to identify the channel id when you have the route of a packet in terms of hops/routers it traverses. Once you have this id, you can use it to index into a table that contains every channel's state (described below).

- Per channel state:
  - Balance for oneself ~~and for the other party~~
  - Total Channel Capacity
  - Queue of transaction units (in the above example, it would be a queue at A of transaction units it wanted to send to B but didn't have funds for)
  - Inflight transactions that were sent out on this channel - hashmap with transaction ids as keys and amount as value
  - Inflight transactions that came into this channel - hashmap with transaction ids as keys and amount as value

3.4 Event Handlers for a single Router

This follows from the flowchart above.

- Receipt of an incoming incomplete HTLC t from upstream router:
  - Add t and the amount to incoming inflight transaction set
  - Process HTLC t:
    * Inspect next hop for t, map it to right outgoing channel id using the channel - channel id map per router
    * Check if there's already transactions queued for that channel in the per channel queue
    * If there's already queue just add t to the queue for the outgoing channel
    * If there isn't a queue, check if we have sufficient funds for t. If not, again, queue t
  - Send HTLC t to downstream router:
    * Decrement balance for outgoing channel ~~don't yet increment other party's balance~~
    * Add t and the amount to outgoing inflight transaction set
    * Send "HTLC Initiate" message on the outgoing channel (assume some propagation delay after which it will be received there)

- Receive HTLC t's ack from downstream router:

  - Successful ack:
    - Remove t from outgoing inflight transaction set on (me - downstream) router channel
    - Send "balance update" message to the downstream router with your own balance decremented and the other router's incremented
    - ~~Increment your local view of the other party's balance~~
    - Send "HTLC ack" to upstream router
  - Failure Ack:
    - Remove t from outgoing inflight transaction set and (only) increment my side of the balance
    - Send "Failure HTLC ack" to upstream router
    - Remove t from incoming inflight transaction set for the corresponding channel it came in on

- Receipt of balance update
  - Increment my side of the balance by the corresponding amount ~~and decrement other party's~~.
  - Check if any transactions were queued for the incoming channel (in the opposite direction) that can now be sent out because of the increment in balance
  - Send as many as you can in priority order on that incoming channel

4 Waterfilling Algorithm

4.1 Heuristic Idea

The essence of the waterfilling algorithm is to maintain balance across a fixed set of paths by always trying to send on the path(s) with maximum bottleneck balance in the direction that you want to send. The bottleneck balance is defined as the minimum balance amongst the edges or payment channels on the path from the source to the destination. For instance, if you have the following payment channel network (Fig. 3), the bottleneck balances on P_1, P_2, P_3 and P_4 between s and t are 20, 15, 10 and 5.

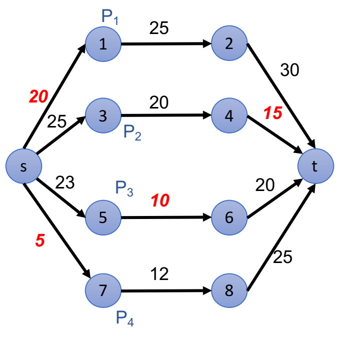

Figure 3: Numbers on edges denote the balance in that direction on the payment channel

If one had to send a payment of 18 units, the first 5 would be sent on P_1, the next 10 would be sent equally on P_1 and P_2 (with 5 on each) and the last 3 would be sent one each on paths P_1, P_2 and P_3. If the payment isn't fully completed in this process, any remaining amount is queued at the sender until the deadline on the sender for the payment expires.

4.2 Simulation Overview

Inferring the bottleneck balances is not a trivial task since nodes don't have a global view of every other node/payment channel balance. Thus, this algorithm breaks down into two phases: querying for channel balances and processing of the probes to compute the waterfilling splits. These two phases will run at slightly different time scales. The querying/probing will happen on every RTT; we send out a probe immediately after receiving the response from the previous one. We start these probes the only once we receive a payment that is to be sent out to that recipient and stop once there is no more demand/transaction to be sent to avoid unnecessary probing overhead. The computation of the splits happens as necessary when the next transaction unit to send arrives or a current set of splits complete/fail.

4.3 Message Types

- Query Channel Balance
  - Sender
  - Receiver
  - Path
  - Is this probe on the reverse path and coming back?
  - Current Hop Number (starts at 0 when the sender creates this message)
  - Empty Map of node to channel balance on the outgoing channel from that node along the path
- Modification to HTLC Initiate/ACK/Balance update:
  - Additional sequence number to identify what chunk of the original transaction this is

4.4 Processing Channel Balance Query at every node

- The sender is reponsible for creating this probe message for every path that it wants to probe and initialize the fields including the empty map
- Every intermediate node on the forward path (check by checking is the flag for reverse path is set to *false*) is responsible for (a) ensuring that it is on the path (at index denoted by hop number), (b) finding the balance on the (me - nextHop) channel and adding it to the map and (c) forwarding the probe to the next hop.
- When the probe is received at the receiver, it changes the direction of the probe by changing the path field to the reverse of the original. The hop number needs to be reset to 0 and the reverse path flag to *true.*
- Every intermediate node (along the path) on the reverse direction merely forwards the probe and doesn't need to do anything else. An added check is to verify that the current node is on the path (at the index denoted by hop number) and then forward it to the next hop (denoted by the item at the next index on the path)

4.5 New State

- State Per Sender
  - Per destination state (updated/initialized when a probe is received)
    - Destination
    - Set of paths (let's start with 2 shortest edge disjoint paths)
    - Bottleneck balance on each of those paths
  - Set of pending transactions (some of these might have been partially completed relative to the initial amount they started with) along with the last sequence number that was used for each of them. This must be updated every time a new transaction is split or an old one is coalesced due to a failure.
- State Per Receiver
  - Transaction Id of transactions received
  - Set of sequence numbers for the given transaction id that have been already received/acked successfully.

4.6 Event Handlers for a Sender

- Receipt of completed probe
  - Update the information in the per-destination table with the latest bottleneck balance for that path
  - Involves computing the minimum across all the balances on that path
- New Transaction To Send
  - If this is a brand new transaction (Sequence number - 0), first send out a probe and wait for the response.
  - Look up the corresponding set of paths for that destination and the associated bottleneck balances.
  - Compute the splits for that transaction according to the balances on each of those paths. This means, send as much as you can on the path with highest bottleneck balance until its balance falls to the second highest one. Then send equal amounts on the two highest bottleneck balance paths until their balances fall to the third and so on.

```
Computes output path_list which consists of tuples (path, amount on path)
begin
  path_list := $\emptyset$

  for $i := 1$ to $k$ do
      $B_i$ := compute_bottleneck_bal($P_i$)
      $PQ := PQ \cup (P_i, B_i)$;                         Insert balances into some heap PQ
  end

  do if size($PQ$) $> 0 \wedge$ txn_amt $> 0$
        highest_bal := PQ.poll()                 Let the path with this highest balance be $p_{max}$
        second_highest_bal := PQ.peek()
        diff_to_send := highest_bal − second_highest_bal
        amt_to_send := $min$(txn_amt/(size(path_list) + 1), diff_to_send)

        for $p$ in path_list do
            amt$_p$ := amt$_p$ + amt_to_send
            txn_amt := txn_amt − amt_to_send
        end

        path_list := path_list ∪ ($p_{max}$, amt_to_send)
    end
end
```

  - Create separate transactions with those splits that are then sent on each of the paths. Give each of them a unique sequence number incrementing from the last one used or starting from 1.
  - If the payment is not completed in the process, queue the remaining at the sender for trying later into the set or queue that is tracking pending payments.
- Received feedback from a payment that was attempted
  - Successful Ack: Do nothing. When splitting the payment, we already assumed it would go through and only queued up the remaining amount.
  - Failure Ack: Add the failed amount to the pending transaction set. If there was already something associated with this transaction, coalesce the two by just adding the failed amount to it.
  - Treat any remaining part of the parent transaction that this Ack was a part of as a new transaction and follow the above procedure.

5 Time Outs with no Splitting of Transactions

5.1 Overview

The purpose of handling timeouts is a way for the sender to cancel a previously sent transactions at time t and be guaranteed that the transaction has been successfully canceled or receive a successful ack by time $t + delta$. A new message type $timeOutMsg$ is introduced that is generated at initialization by the sender of the transaction and is sent out at the $timeSent + timeOutLimit$ for each transaction. A message type $clearStateMsg$ is introduced at every router node that periodically implements the effects of the $timeOutMsg$.

5.2 New Message Types

- timeOutMsg
  - Amount
  - Transaction Id
- clearStateMsg
  - (possibly *delta*, else no fields)

5.3 New State per Node

- set of tuples containing transaction id for time out messages and time stamp for when time out message was received at the router node as well as the previous and next nodes if applicable
  - `set<canceledTrans> canceledTransactions`
  - `canceledTrans`
    * `transactionId`
    * `simTime_t`
      of time we received timeOutMsg
    * `prevNode`
      (if not sender)
    * `nextNode`
      (if not destination)
- (Host only) set of transaction ids for transactions that we have received acks for and also have a time out limit (so we already generated a timeOutMsg and scheduled it for that transaction)
  - `set<int> successfulDoNotSendTimeOut;`

5.4 Implementation

5.4.1 Changes to initialization

- Calculate the longest path between two nodes in the graph, store as a global constant *delta*

5.4.2 Handling the timeOutMsg

- (Generation) a timeOutMsg is generated at initialization by the sender of the transaction and is scheduled to arrive at the same sender node at time $timeSent + timeOutLimit$ for each transaction
- if the timeOutMsg is at hop count 0 or the sender (first time we are seeing it), check if the transaction Id is in the set of successful transactions (successfulDoNotSendTimeOut), if so, then delete the timeOutMsg, and the transactionId from the set of successful transactions
- If not, when processing the timeout message at other routers,
  - **Radhika:** Delete transaction from the out-of-balance queue.
  - add the transaction id and current time to the set of canceled transactions (canceledTransactions) **Radhika:** You need to maintain this list at the sender as well, right? I think it is required to conclude whether or not a timed-out transaction has failed or succeeded. Please checkout the comment below.
  - if the time out message is at the destination node, delete it

5.4.3 Changes to Handling Ack Message

- check if the transaction id in the ack message is in the set of canceled transactions. If so, delete it from the set of canceled transactions.
- (Host) if the ack message is back at the original sender of the transaction, add the transaction to the set of successful transaction ids (successfulDoNotSendTimeOut)

5.4.4 Changes to Handling Transaction Messages

- when receiving a transaction message, check to see if the transaction Id is in the set of canceled transactions, if so, delete the transaction message.
- ~~when processing transactions (these include transactions in the queue, sending the transactionMsg out and decrementing the balance), check if the transaction we are trying to send out is in the set of canceled transactions, if so, delete it and don't change any balances~~

5.4.5 Handling clearStateMsg

- (Generation) a message is generated at initialization for every node (router/sender) scheduled to arrive at time currentTime+clearRate
- For all transactions in the set canceled transactions whose cancellation time was more than *delta* time ago, remove the transaction from the incoming and outgoing maps. If removed from outgoing map, increment my end of the balance on the payment channel to the next node. If removed from the incoming queue, we do *not* need to decrement the balance to the previous node. Search for the transaction in the queue of transactions waiting to be sent to the next node, and delete it if found.
- **Radhika:** At the sender, if a transaction in cancelled set, having cancellation time more than *delta* time ago, has received acks for the entire transaction amount, count it as a successful transaction. Otherwise, count it as a failed transaction.
- Delete the transaction mentioned from the set of canceled transactions.

6 Handling Time Outs with split transaction (Waterfilling)

6.1 Overview

We add the "acked amount" to the ack messages and keep track of the total amount sent on each path, and total amount acked on each path, for every transaction. A transaction is successful only if the entire transaction amount has been acked (i.e. the sum of the acked amount across all paths is the same as the total transaction amount). Otherwise, (i) if there are paths for which the amount sent is greater than the amount acked, a timeoutMsg is

sent on those paths and the transaction is added to the list of canceled transactions, (ii) if no such path exists, the transaction is simply added to the cancelled list without sending any timeOutMsg.
Vibhaa: In both cases, we need to make sure that no further "portion" of the transaction is sent out. In the sense that any logic that sends out a fresh set of HTLCs based on probes or the action of sending probes themselves needs to stop - in particular, the payments per destination counter needs to be decremented since this transaction is over now

6.2 Additional data structures

- maintain a map with the key as transaction id and index of one of k paths and the value as a two numbers, the total amount sent down that path, and the total amount we have received acknowledgements for from that path
  - `map<tuple<int, int>, AmtState> transPathToAmtState`
  - struct AmtState
    - double amtSent
    - double amtReceived

6.3 Changes to handling timeOutMessages

- Generate a timeOutMsg for each of the k paths. When the timeOutMsg arrives at time timeSent + timeOutLimit, check if the map value corresponding the transaction id and path index has the same value for amount sent and amount acknowledged. If not, forward the timeOutMsg.

6.4 Changes to handling acks

- When an ack is received at the sender, get the transaction id and path index from the ack, and update the map transPathToAmtState

7 Simpler SilentWhispers

7.1 Overview

Landmark Routing is a reasonable proxy since SilentWhispers builds on top of it. The idea is that there are a set of landmarks that are responsible for routing in that the sender sends transactions to the landmark which then sends transactions to the destination. There are multiple landmarks and the question is how should a transaction be split across the paths that use each landmark. SilentWhispers does this last part in a privacy preserving manner, but we are going to assume that we can relay this balance information and do the minimum computation without the secret sharing overhead.

7.2 Additional data structures/information

- Set of landmarks (global information and likely to be routers) - a few well connected nodes - to be picked ahead of time based on graph connectivity
- Table at Landmarks of source $\rightarrow$ landmark balance information while it is sending a probe to find thelandmark $\rightarrow$ destination minimum balance
- Shortest Paths to and from landmarks (per end host/source and destination) computed via BFS (**Vibhaa:** Should this be computed once in the beginning or periodically with preference towards links with credit? Nothing about the paper seems to suggest that this is done in any "smart" fashion. This should not be changed too frequently otherwise the minimum will be computed for one path and the transaction sent on a different one)

7.3 New Message Types

- ProbeToLandmarkMsg
  - Sender
  - Landmark
  - Destination
  - Path to Landmark

  - Current Hop Number (starts at 0 when the sender creates this message)
  - Empty Map of node to channel balance on the outgoing channel from that node along the path

- ProbeFromLandmarkMsg
  - Original Sender
  - Landmark
  - Destination
  - Path from Landmark to Destination
  - Is the probe on the reverse path? (if so, just need to forward it back)
  - Current Hop Number (starts at 0 when the landmark creates this message)
  - Empty Map of node to channel balance on the outgoing channel from that node along the path

- MinimumInfoMsg
  - Original Sender
  - Landmark sending this info back to the above sender
  - Destination for whom this minumum was computed
  - Minimum balance on path from above sender via above landmark to above destination
  - Path from landmark to sender
  - Current Hop Number (Starts at 0 when landmark creates this message)

7.4 Implementation

7.4.1 Changes to handling transactions at the sender

- First sender sends a "probe" to every landmark along the shortest path to that landmark
- Sender gets a minimum from each landmark corresponding to the minimum on the path through that landmark to the destination
- Sender waits for all probes to come back and then computes splits for the transaction such that total transaction amount is met, but individual path minimum balances aren't violated
- Sender sends out transaction according to splits on all these paths
- Actual splits computed somewhat randomly and then reassign amounts that exceed minimum and so on until and unless there is no way to achieve the transaction (See `https://github.com/vibhaa/lightning_routing/blob/7660d88a8808019dd1d4590cad44d10112851aeb/speedy/src/treeembedding/credit/partioner/Partitioner.java#L23`)

7.4.2 Handling ProbeToLandmarkMsg

- Initiated at the sender - to be sent to a particular landmark - with the goal of probing balance to the destination
- Intermediate nodes append their balance information onto it
- When landmark receives it, it computes minimum balance on the forward path and stores it in a table of source - (route, min balance) information
- the landmark finds the shortest path to the destination. It creates and sends out a new probe to the destination

7.4.3 Handling ProbeFromLandmarkMsg

- Initiated at the landmark once probe from sender has been received
- Intermediate nodes append their balance information
- When the destination receives a "probe from landmark" it sends it back to the landmark
- Intermediate nodes merely forward on the way back
- Landmark computes the minimum when it gets the probe back and sends the minimum back in a new message

7.4.4 Handling MinimumMsg

- Initiated at the landmark after it recieves probeFromLandmarkMsg back
- Landmark computes the minimum of all the balances on the received probe and the minimum stored in the source - (route, min balance) table
- Landmark Embeds this information and the reverse of route from the same table and sends message to sender
- Intermediate nodes forward this all the way to the sender
- Sender aggregates information from all probes and computes transaction splits

7.5 Questions/Comments

- No change to timeouts, though the delay with this setup might be a lot higher
- How to compute shortest paths? Should this just be stored at sender and landmark (this seems simplest to reason about)?
- Acks are received for every individual path and now the transaction is successful only if all paths returned a successful ack. **Vibhaa:** Check if this is easy to modify/is already somewhat implemented for sequencing for WF.
- If we are assuming small transactions of indivisible size, should we just send it on one path with the maximum bottleneck balance or are we splitting transactions actually?
- One option is to reuse the ProbeToLandmarkMsg for the minimum information also, but that would mean holding onto the message for a while, which seems not ideal.

8 SpeedyMurmurs

8.1 Overview

At the heart of SpeedyMurmurs is an embedding based approach to routing payments. What this means is that every node is assigned a coordinate which is then used to compute relative distances between nodes and so on. This is made privacy preserving, which we can ignore. **Vibhaa:** shortcuts??. A combination of these distances and the presence of funds is used to determine what routes to use for every transaction.

8.2 Global State for Coordinates

Coordinate generation starts at the landmarks and then visits neighbors and their neighbors and so on. At each point, one generates an empty coordinate list for the landmark or the root and then adds a new dimension for the next level of children. In this new dimension, you start with 1 for the first child and go on until all children at this level have been assigned coordinates. Then you process all of their neighbors and so on. (See Figure 1 of `https://arxiv.org/pdf/1709.05748.pdf`). This process is repeated for every one of the L landmarks generating a separate coordinate for every node according to the tree corresponding to that landmark.
It is ideal if these coordinates can be maintained as global state that is updated via a trigger from any node when a link's balance becomes 0 or when a link's balance changes from 0 to some non-zero value alone. **Vibhaa:** What are the exact details here?

8.3 New Message Types

- RouteDiscoverMsg
  - Sender
  - Landmark i
  - Destination
  - Split amount c_i for landmark i
  - Route so far
  - Failure

8.4 Additional Data structures

- List of L coordinates for every node which denotes the coordinate in the i^{th} landmark's tree (Part of the global state)
- (Per node) Per payment channel "guaranteedAvailableCredit" denoting what is available after considering amount reserved for transactions that we are currently discovering routes for. Initialized to the balance on a given node's end for every payment channel.

8.5 Implementation

8.5.1 Changes to Handling Transaction

- When a new transaction has been received at a sender, split into L splits for L landmarks at random.
- For each landmark i and the split c_i, initiate a routeDiscoverMsg and send it to the neighbor that is closest to the sender (based on coordinate distance) **Vibhaa:** not sure how to account for "shortcuts" and has atleast c_i balance on the corresponding payment channel.
- When a successful routeDiscoverMsg has been received for landmark i, send out an HTLC for the path through that landmark for amount c_i
- If a failure was received instead, just track this part of the payment as incomplete.
- Keep track of how much of the transaction has been completed so far and retry (**Vibhaa:** not sure if this is needed or we declare failure at the first instance of an unsuccessful routeDiscoverMsg)

8.5.2 Handling RouteDiscoverMsg

- Initiated at the sender - one per landmark with a singleton route (sender alone), false Failure, and the corresponding split
- If failure is false,
  - find the closest neighbor based on coordinate distances and as long as the guaranteedAvailableCredit on the payment channel between current hop and neighbor is atleast c_i
  - If such a neighbor is found,
    - append this neighbor to the route and forward RouteDiscoverMsg onto the next router
    - Reserve funds c_i on the payment channel between current node by decrementing the guaranteedAvailableCredit with this neighbor by c_i.
  - If no such neighbor found, set Failure to true, reverse the route and send it back to the sender
- If failure is true,
  - increment guaranteedAvailableCredit by c_i on the payment channel between current node and the nextHop in the forward direction (to restore funds that were reserved)
  - forward the message backwards (in the opposite direction as dictated by the route) all the way to sender

8.5.3 Changes to Handling Acks

Once an Ack has been received at a certain node for the payment channel to a neighboring node, if the guaranteedAvailableCredit at the receiving node's end for that payment channel is 0, initiate a recomputation of global coordinate state.

8.5.4 Changes to Handling BalanceUpdate Messages

Do the regular processing for HTLCs by adding the amount to inflight outgoing and then waiting for Acks before updating balances and so on. But, also increment the guaranteedAvailableCredit at the receiving node's end for a given payment channel since it reflects the clearing out of payment channel state between two nodes. If a given node's guaranteedAvailableCredit for a payment channel goes up from 0 to non-zero, initiate a recomputation of global coordinate state.

9 Backpressure