Omnet Router Specification

November 20, 2018

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)

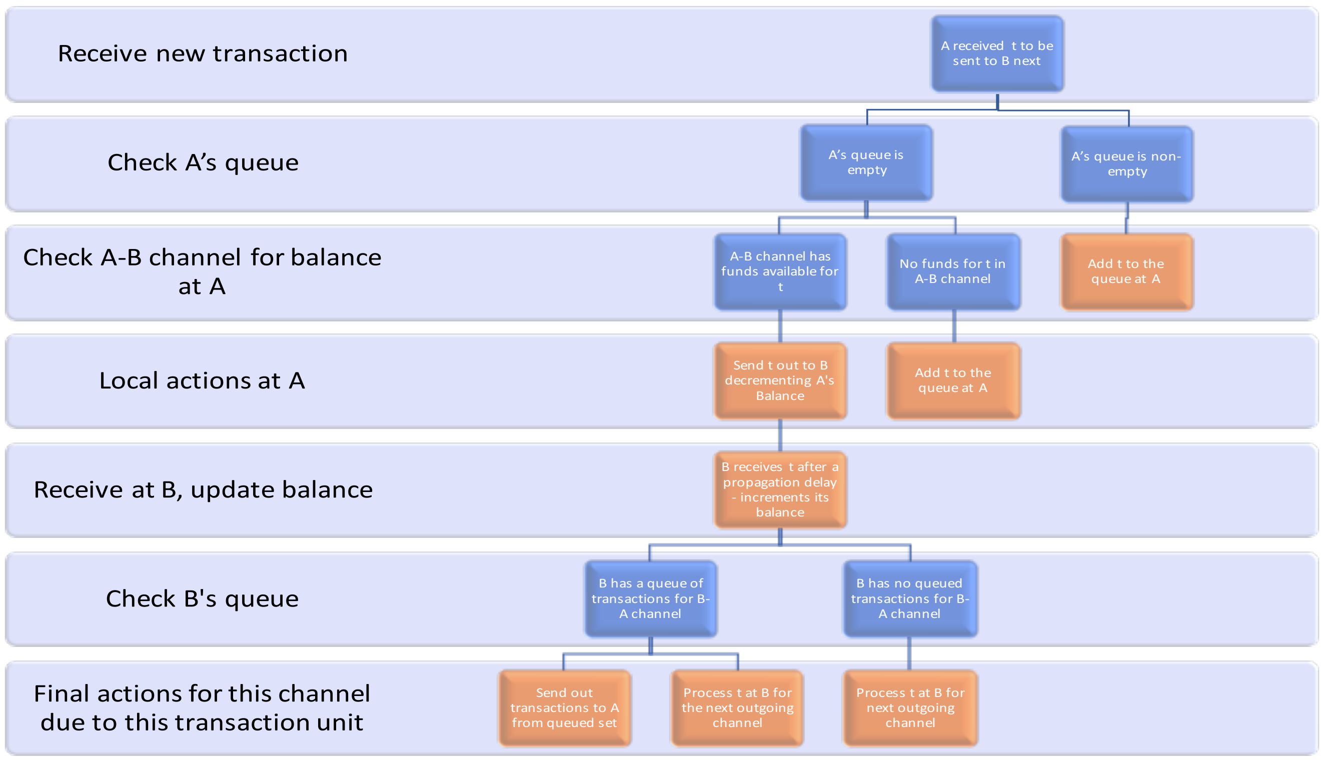
Receive new transaction
A received t to be sent to B next
Check A's queue
A's queue is empty
A's queue is non-empty
Check A-B channel for balance at A
A-B channel has funds available for t
No funds for t in A-B channel
Add t to the queue at A
Local actions at A
Send t out to B decrementing A's Balance
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.

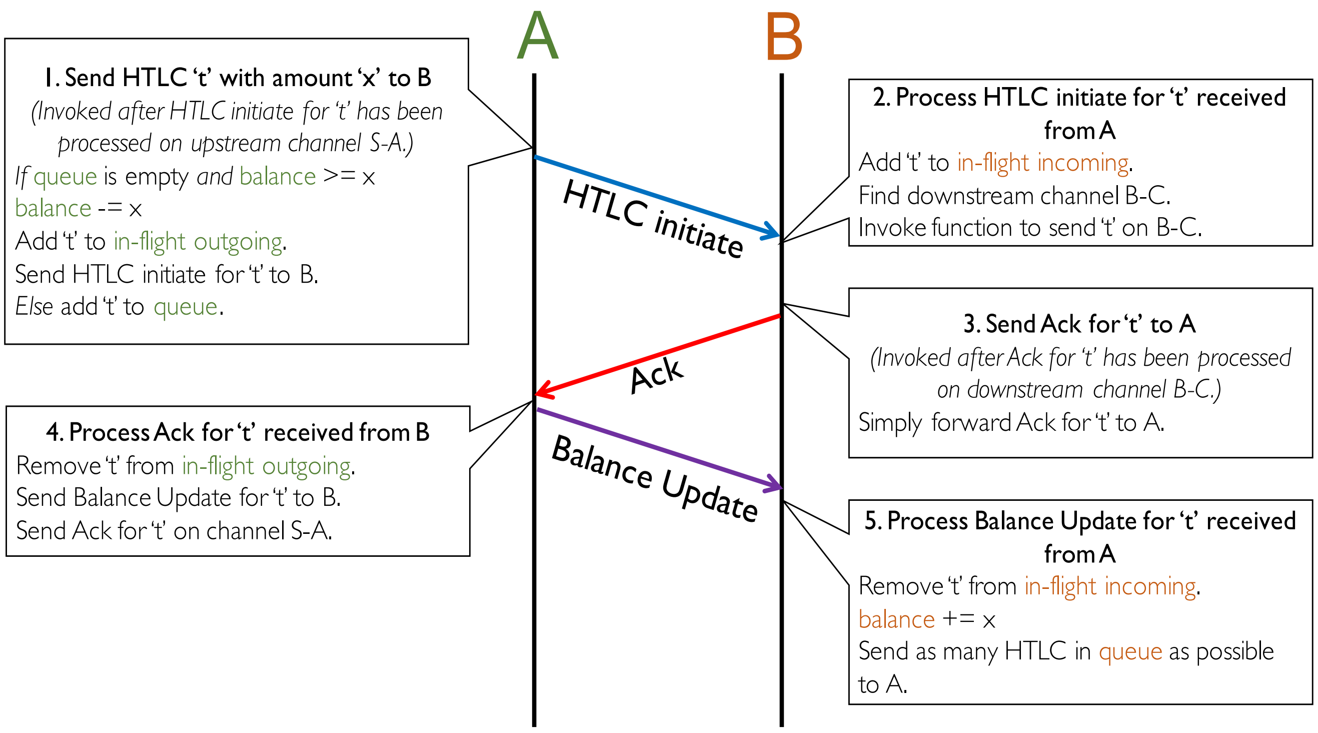
A
B
1. Send HTLC 't' with amount 'x' to B
(Invoked after HTLC initiate for 't' has been processed on upstream channel S-A.)
If queue is empty and balance >= x
balance -= x
Add 't' to in-flight outgoing.
Send HTLC initiate for 't' to B.
Else add 't' to queue.
HTLC initiate
2. Process HTLC initiate for 't' received from A
Add 't' to in-flight incoming.
Find downstream channel B-C.
Invoke function to send 't' on B-C.
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.

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.

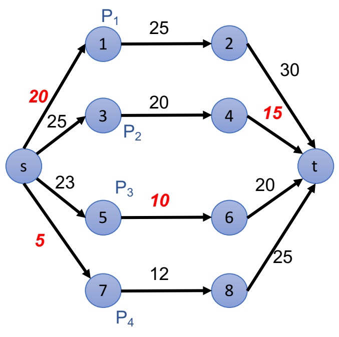

Figure 3: Numbers on edges denote the balance in that direction on the payment channel

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.