Omnet Router Specification

October 22, 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

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.
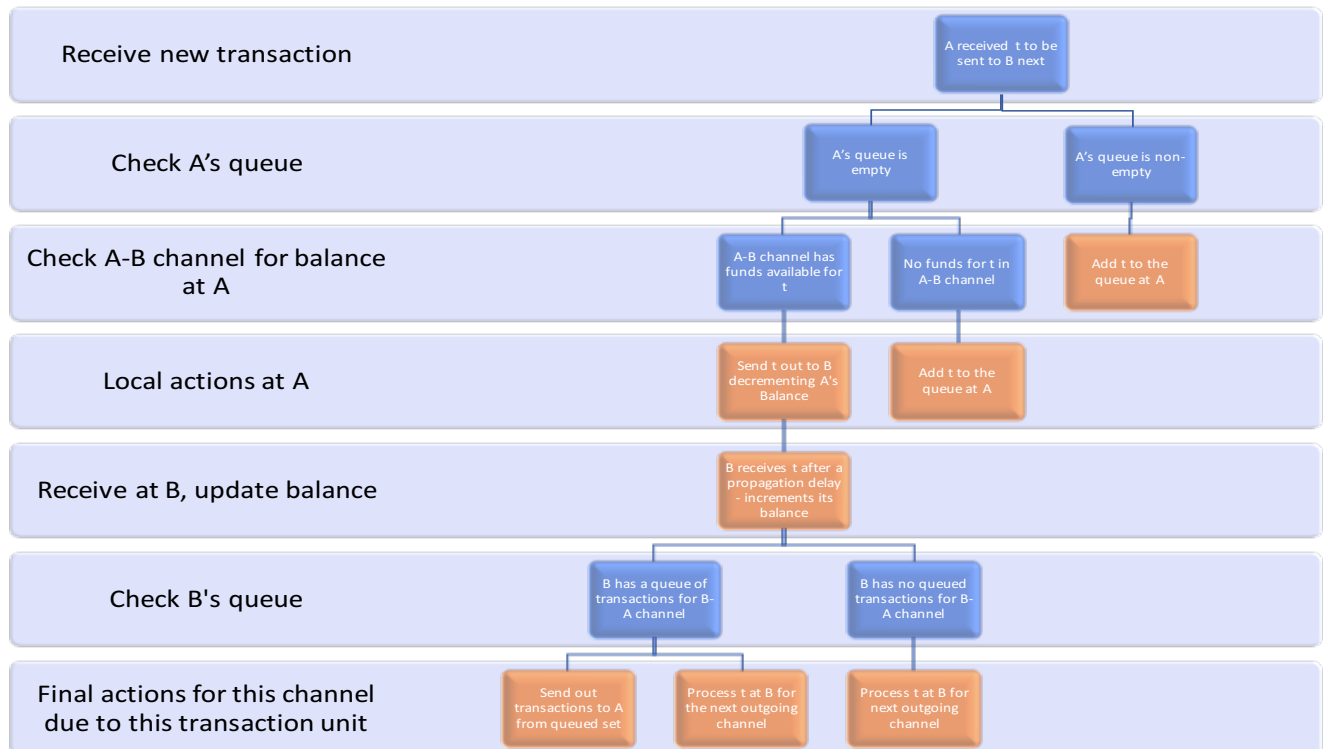
This is illustrated in the following flowchart.

Figure 1: flow of transactions at a router

3 Flow of HTLCs - With Receiver Acknowledgements

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. This means that the HTLC has been attempted on this channel and is going out from A to B. So, we add t to a set of the inflight outgone 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. Once B has completed this, it looks at the next hop and repeats the procedure on the outgoing channel at B.

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.

4 Message Types

- 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)

- 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
  - Status (Success or Failure) - just in case we want to simulate failures and propagate them upwards
  - Secret - Dummy field for now (might never change)

5   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)
  - 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

6   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
  - 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
    * Send "HTLC ack" to upstream router
  - Failure Ack:
    * Remove t from outgoing inflight transaction set and 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
  - 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