Ripple Inter Server Protocol built on a fee system and single-hop consensus

Johan Nygren, @bipedaljoe

In 2004 Ryan Fugger invented the money system Ripple, where each person was a bank. Since each person is a bank, it is possible to organize Ripple in a truly distributed way, computationally. Consensus is only needed at the user-to-user level, i.e., only ever between two people. To enforce agreements at the multi-hop level, a fee system can be used. The fee system enforces signal propagation at each hop, preventing stuck half–finished payments. The fees compensate users who get stuck with a half-finished payment and they penalize the person who is causing the payment to not proceed. The fees are collected from the payment by users in a decentralized way where any cheating only impacts your own trust relationships.

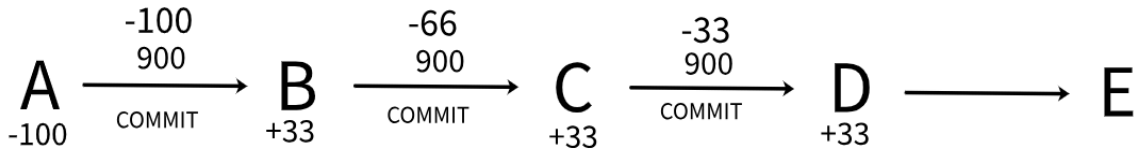
The fee system

Fees can enforce signal propagation when they can apply a net negative balance on the person who is causing the signal to not propagate. Fees work in a straightforward way when propagation is at “edge nodes”, i.e., when a signal originates from an “edge node” (buyer or seller) and nodes that have propagated the signal then remove themselves from the payment so the next node is thus the new “edge node”. To enforce propagation at intermediaries, fees instead need to be balanced in an asymmetric way so that they are stronger on one side of the intermediary and weaker on the other.

The fee system uses two signals that are propagated by “edge nodes”: “buyer cancels” and “seller finalizes”, and it uses one signal that is propagated by intermediaries: “seal payment”.

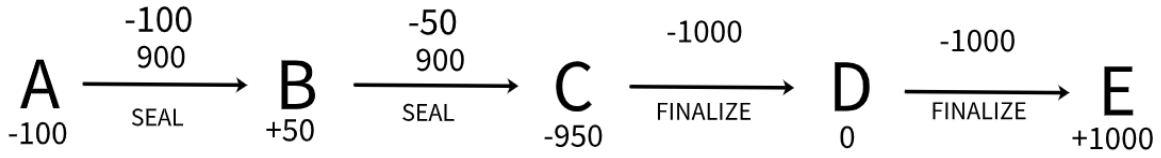
“Buyer cancels” signal

“Buyer cancels” is used to commit to the multi-hop payment with the “commit” signal. The “commit” signal is itself not enforced by fees, instead it relies on “buyer cancel” to avoid the payment getting stuck. If the “commit” signal fails to reach the seller because it gets stuck at an intermediary, the fees will start to be collected and the buyer will be the one paying for them. The buyer therefore has an incentive to cancel the payment. When the buyer cancels (and remove themselves from the payment), the fees start to act on the next user in line (B in the illustration below), as they become the new “edge node”. The new “edge node” (B) then has an incentive to propagate the cancel signal to C who then becomes the new “edge node”, etc.



“Seller finalizes” signal

“Seller finalizes” is used to finish the payment, and it relies on the fee system as the incentive to enforce propagation of the decision. The fees are being collected by users between the person who does “finalize” and the buyer. While the buyer is the one paying the fees to the intermediaries, the person currently doing “finalize” is paying the buyer. As fees are paid out, the amount that the intermediary can finalize from the next hop is reduced. Thus the fees mean they pay more out than they receive in.



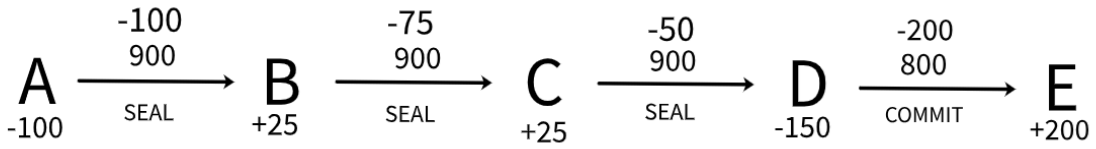
In the illustration above, note that if C were to propagate “finalize” at the moment shown in the illustration, they would only be able to finalize 950 (900 + the 50 they already finalized as fees), and thus end up with a net negative balance of -50. B would likewise only be able to finalize 950 but as their outgoing balance with C ended up 950 as well, they have a net neutral balance. And A would finalize 1000 in full, which in the situation above is also what E received, thus A was not impacted financially.

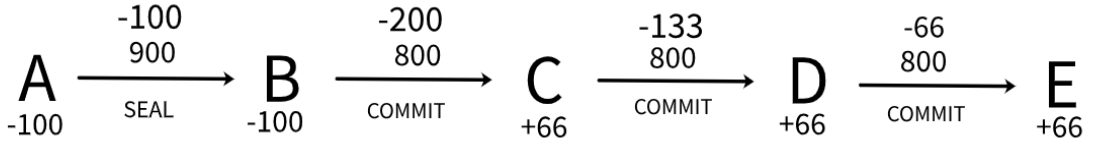
“Seal payment” signal, “asymmetric fees”

To avoid the attack where the buyer and seller issue both “buyer cancels” and “seller finalizes” at the same time, the buyer must revoke their right to cancel and inform every intermediary about doing so prior to “finalize”. This is the role of the “seal payment” signal. Contrary to “buyer cancels” and “seller finalizes” the “seal payment” is not propagated by an “edge node” (every person who propagates it stays in the payment), it is propagated by an intermediary. Thus for fees to act on the intermediary, the fees must be asymmetric. “Seal payment” propagates from the buyer and towards the seller, thus the fees must be higher on the seller side than on the buyer side. Thus we end up with two separate fee systems, a stronger one set up during “commit” that acts on “buyer cancels” and “seal payment”, and a weaker one during “seller finalizes” (that also allows the stronger one to act as enforcement during “seal payment”).

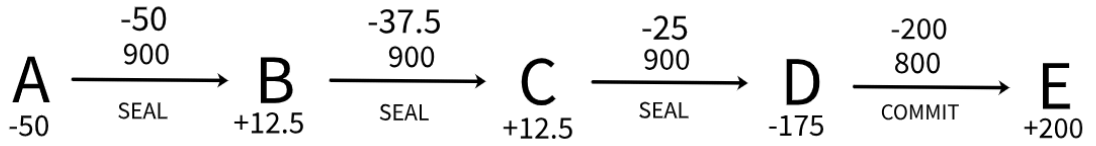
The weaker fee system set up during “seal payment” is what incentivizes “seller finalizes”. The buyer is the one paying the fees for it. This means that when an intermediary does not propagate the “seal payment” signal, the fees will also penalize the buyer even though the buyer is innocent. This is necessary. It is possible to guarantee the buyer never pays more than the attacker, but we cannot fully remove the penalty on the buyer.

To guarantee the attacker during “seal payment” always pays more than the buyer (unless they are the friend of the buyer and then they pay the same amount), the first fee system (the one that acts on the seller side of the intermediary during “seal payment”) has to operate at twice as fast as the second one (the one on the buyer side of the intermediary during “seal payment”). The two illustrations below demonstrate this effect when the attacker is a few hops away from the buyer as well as when they are a direct friend of the buyer.





The penalty on the buyer can then be further reduced, by cancelling part of what would have been collected as fee. In the illustration below, on the buyer side half the amount that would have been claimed as fee was instead cancelled. This results in a net negative balance for fees paid of -50 for the buyer and -175 for the attacker.



Decentralized (per-user) enforcement of fees

Fees can be collected in a fully decentralized way, as any cheating impacts only the next hop. Thus if you cheat you only hurt your own friends - there is no transitivity of trust problem. Users simply have a config parameter `FEE_RATE`, and keep track of how much fees should have been collected in total at a point in time (this parameter can be synchronized for the users involved in a payment if each user accepts a lower and an upper bound, such as from `0.5*FEE_RATE` to `1.5*FEE_RATE`). Users always accept `PAYMENT_FEE` commands from their friends, but they only propagate them up to the amount they calculate should have been collected up to that point. Users periodically run a `collectFees()` routine and grab a random amount of fees they calculated are available. Users that always pick max fees disproportionately hurt their own friends.

User-to-user consensus

A person-to-person Ripple requires that consensus at the person-to-person level is guaranteed. The issue with person-to-person decision making over the internet is the "two general problem", that it is impossible to be certain about if an acknowledgement was delivered. Thus it follows that agreement is impossible. The solution to the two-general problem is to agree on a single general. The ideal way to do so, is to take turns being the general.

Lockstep, a single-hop consensus mechanism

People in a trustline-relationship agree to take turns to be "general" who gets to say which transaction should happen next. They coordinate this with the use of a counter, and agree that one person will validate when *counter mod 2* is 0 and the other when *counter mod 2* is 1. The person who is not the "general" at a turn can propose transactions to the "general". Each person stores the instruction they last validated in permanent storage until they receive the next rounds validated transaction (thus continuity is guaranteed.) In permanent storage you thus maintain a turn bit (0 or 1) for *counter mod turnbit*, and a turn counter for same operation, and the last validated instruction (an instruction being a command with arguments). The "state transition function" in Lockstep includes incrementing the turn counter and setting the last validated instruction, as well as the state changes that the instruction performed, and is "atomic", all-or-nothing. Thus, two accounts are in perfect agreement over every decision that they make.

Integrating Lockstep with the role as an intermediary

With single-hop guaranteed, two-hop at an intermediary is guaranteed since it is on a single machine. But it still has to be organized practically, on that machine. A simple approach is that the command handler returns an "inter-lockstep" callback. This callback is executed only if the Lockstep state transition succeeded. Since such callback could fail, it is ideally combined with "failsafe" routines. For example, a callback that queues a transaction could fail because the queue is full. A failsafe routine can derive the same decision from the permanently stored data, and run periodically to catch any failures.

Single-hop consensus and fee system allow multi-hop agreements

Lockstep solves the two general problem and guarantees that any signal that propagates over multiple hops has been agreed on by every previous hop. The fee system guarantees that signal propagation over multiple hops will either succeed, or fees are paid out to every person who is a victim of the stuck payment (paid by the person who caused the payment to get stuck). When combined they make multi-hop agreements possible.