INPUT | OUTPUT

Babel Fees via Limited Liabilities

Authors : Polina Vinogradova, James Chapman, Andre Knispel, and Orestis Melkonian

Presenter : Polina Vinogradova

The Problem

- We want to be able to **predict** what changes a transaction makes to the ledger (or its parts) to which it is applied
  - This ability to predict is colloquially referred to as **ledger determinism**

- **Under what conditions** can we make this prediction correctly?

Motivating Example

Selling a Sword

Ledger
Offer1 : 1 Swords for 5 Coins paid to PK1
…
PK2 has 20 Coins
…

Transaction1 :

PK buys Sword
in Offer1

Ledger
…
PK1 has 5 Coins
PK2 has 15 Coins
PK2 has 1 Sword

Motivating Example

Selling a Sword

Ledger
Offer1 : 1 Swords for 5 Coins paid to PK1
…
PK2 has 20 Coins
…

Transaction1 :

PK1 changes Sword price in Offer1

Ledger
Offer1 : 1 Swords for 10 Coins paid to PK1
…
PK2 has 20 Coins
…

Transaction2 :

PK buys Sword in Offer1

Ledger
…
PK1 has 10 Coins
PK2 has 10 Coins
PK2 has 1 Sword

PK1 and PK both care which transaction gets processed **first**!

The Problem

Why is this prediction hard?

- It should be **easy**, though - blockchains rely on deterministic computation!
  - ie. given a ledger, and a block (or transaction), it is possible to compute the new ledger that results from applying the block/transaction

- In practice, however, a user **cannot predict what ledger** their transaction/block will be applied to

- There are a number of reasons for this :
  - unpredictable propagation of transactions over the network, rollbacks, delays in getting the most current ledger, malicious actors, etc.

Our approach

Make it formal, and forget the state (sort of)

- Consider ledgers as **state transition systems**, with valid transactions (or blocks) as the only transitions
- Construct a **formal specification** of this view of ledgers
- Ask "what if we **did not need the ledger state** to which a transaction is being applied to determine what changes it will make?"
- Formulate this in the language of our specification, and use mathematical tools to look for the answer

Ledger (Specification) L

- Specifies what it means for something to be a ledger
- Captures the structure shared by most ledgers (eg. Cardano, Bitcoin, Ethereum, Tezos, Zilliqa)
- Is a tool for comparing different ledgers across a formally stated property, eg. determinism

$\mathtt{State}_L : \mathtt{Type}$

ledger state type

$\mathtt{Tx}_L : \mathtt{Type}$

transaction type

$\mathtt{initState}_L : \mathtt{State}$

initial state

$\mathtt{updateState}_L : \mathtt{Tx}_L \to \mathtt{State}_L \to \mathtt{State}_{L\perp}$

output new state with a given transaction applied,
or throw an error

Valid States

From here on, we only talk about valid states

- A ledger state $\texttt{s} \in \texttt{State}_L$ is valid whenever there exists a trace $\texttt{lstx} \in [\texttt{Tx}_L]$ from the initial state to $\texttt{s}$, such that

$$\texttt{foldl updateState}_L\ \texttt{initState}_L\ \texttt{lstx} = \texttt{s}$$

- We denote the set of all valid states (ie. the dependent pairs of a state and a trace proving its validity), plus the error state, by

$$\texttt{ValSt}_{L\perp}$$

- We use shorthand $(\texttt{s lstx})$ for $\texttt{foldl updateState}_L\ \texttt{s lstx}$

Order-Determinism (OD)

Transaction commutativity with errors

$$\forall\ lstx \in [Tx_L],\ lstx' \in \text{Permutation } lstx,$$
$$(initState_L\ lstx) \neq \perp \neq (initState_L\ lstx')$$

$$\Rightarrow (initState_L\ lstx) = (initState_L\ lstx')$$

- Given any list of transactions, any permutation of this list, when applied to the initial state, will result in the same state as applying the original list - unless applying either list produces an error

Examples

We define two ledgers *L*, *K*

$\texttt{Tx}_L = \texttt{State}_L = \texttt{Tx}_K = \texttt{State}_K = \texttt{Bool}$

transactions and states are booleans

$\texttt{initState}_L = 0 = \texttt{initState}_K$

initial states are both 0

Update in L : `(s 0) = 0, (s 1) = 1`

Update in K : `(s tx) = s XOR tx`

Examples

How are these different?

- Transactions **commute** :
  - `(a XOR b) XOR c = (a XOR c) XOR b`

L

The **change** a transaction makes to the state is independent of the state

- (0 means no change, 1 means flip the bit)

- Transactions **do not commute** :
  - `((s 0) 1) = 1 ≠ 0 = (s 1) 0`

K

The **change** a transaction makes to the state is independent of the state

- changes every state to state specified in the transaction

The **actual output state** is also independent of the input state

Theory of Changes

How do we classify changes in these ledgers?

```
Diff = [Tx]
```

the type of changes for *L*

```
applyDiff s txs = (s txs)
```

apply the change set

```
extend
```

```
zero
```

- Difficult to define change set type while remaining **agnostic** of the underlying data structure
- **But**, for ledgers, every permissible set of changes corresponds exactly to a sequence of valid transactions, and update is the function that applies those changes
- We omit some details here

Theory of Changes

How do we classify changes in these ledgers?

```
takeDer ∈ [Tx] → DerType

evalDer ∈
     DerType → State → Diff → Diff
```

Constraint : When neither side is ⊥,

```
((applyDiff ds s) txs) =

   applyDiff (evalDer (takeDer
   txs) s ds) (s txs)
```

- The constraint says that
  - applying the `Diff` **first**, then applying a change set `txs` , is the same as
  - applying `txs`, then applying the differentiated `Diff` term

Theory of Changes

What do we want to say using this theory?

```
Diff = [Tx]
```

the type of changes for *L*

```
applyDiff s txs = (s txs)
```

apply the change set

```
applyDiff s txs = (s txs)
```

apply the change set

Update in *K* : $\forall$ tx $\in$ Tx_K, s $\in$ $State_K$,

(s tx) = s XOR tx

- Difficult to define change set type while remaining **agnostic** of the underlying data structure
- **But**, for ledgers, every permissible set of changes corresponds exactly to a sequence of valid transactions, and update is the function that applies those changes
- We omit some details here

Threads

Determinism and ledger structure

$\texttt{T : Type}$

the type of changes for *L*

$\texttt{proj}_{\texttt{T}} : \texttt{State}_L \to \texttt{T}$

projection function (with right inverse)

$\texttt{updateT}_L : \texttt{Tx}_L \to \texttt{State}_L \to \texttt{T} \to \texttt{T}$

update thread in a given state

$$(\texttt{s tx}) \neq \perp \wedge \texttt{proj}_{\texttt{T}}\ \texttt{s} = \texttt{t}$$

$$\Rightarrow \texttt{updateT}_L\ \texttt{tx s t} \neq \perp$$

- Threads represent **components** of the ledger
- $\texttt{updateT}_L$ function coincides with the ledger update function when both succeed, but **thread update failure implies ledger update failure**
- Examples :
    1. UTxO set
        - persistent
    2. Single UTxO entry
        - non-persistent
    3. Set of accounts
    4. Single account
    5. Time (eg. slot number)

Threads

State-independent threads and smart contracts

$\forall$ s s′,
updateT_L tx s t $\neq \perp$,
updateT_L tx s′ t $\neq \perp$

$\Rightarrow$

updateT_L tx s t = updateT_L tx s′ t

eg.

T = (TxIn, TxOut)
updateT_L tx s t =
♦ when t consumed by tx
$\perp$ when tx cannot spend it

Hand-wavy example :

- This is how we want smart contract evaluation to behave
- In the UTxO model, smart contracts are stateless, so a thread would only point to a specific UTxO entry locked by a contract, and return ♦ when it is not present on the ledger
- The update function simply removes the UTxO

Summary : The Solution

We propose the Babel Fees mechanism that lets users submit transactions that function as exchange offers of the type $\neq \perp$

“I offer **X amount of token Y** to anyone who is willing to supply the **Ada** required to cover the fee of this transaction”

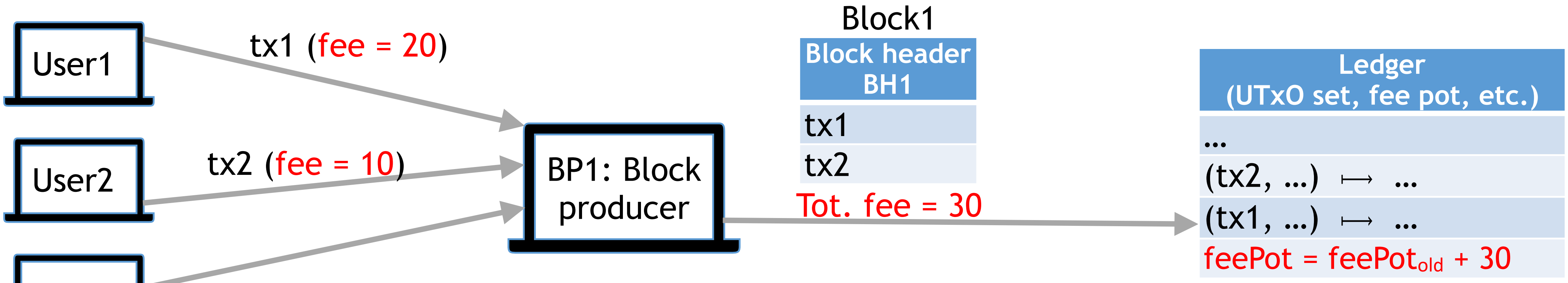
Anyone interested in the offer may submit a transaction accepting it

Together, these transactions form a valid batch, and can be added to the ledger

- Neither can be added by itself

The result is that, effectively, the user is paying fees in token Y instead of Ada

Cardano's fee mechanism review

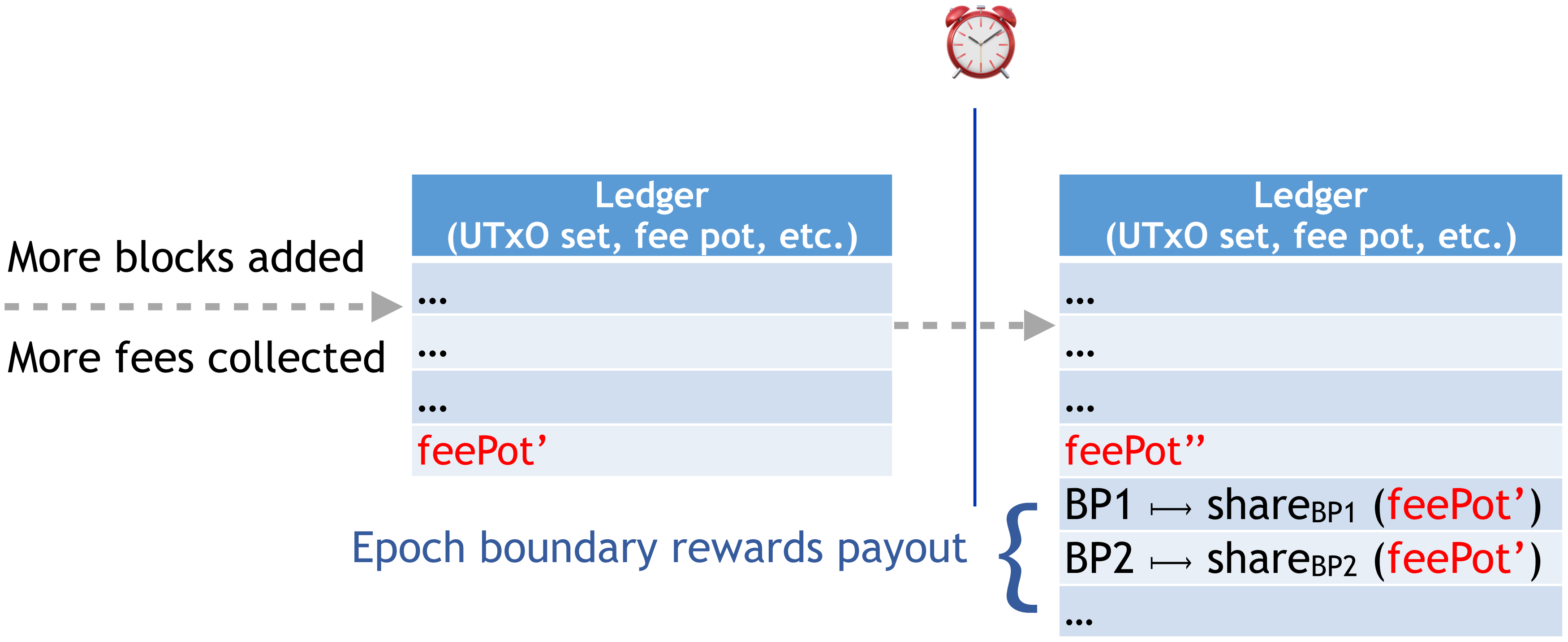


- Transaction fee is calculated based on size, smart contract execution time, etc.
- Fees go towards rewarding block producers

- Blocks have a limited capacity (eg. max size)
- Producing blocks takes resources (memory, CPU, equipment costs)
- Producers can choose what transactions to include to maximize their rewards for block production

- Fees collected from transactions in a block go into a fee pot

Cardano’s fee mechanism review



These shares are in Ada.

Dividing up non-Ada fee rewards would require a **very complicated ledger change!**

Note that **every block producer in an epoch gets a share** of the fee pot containing fees from every block produced that epoch

Related Work

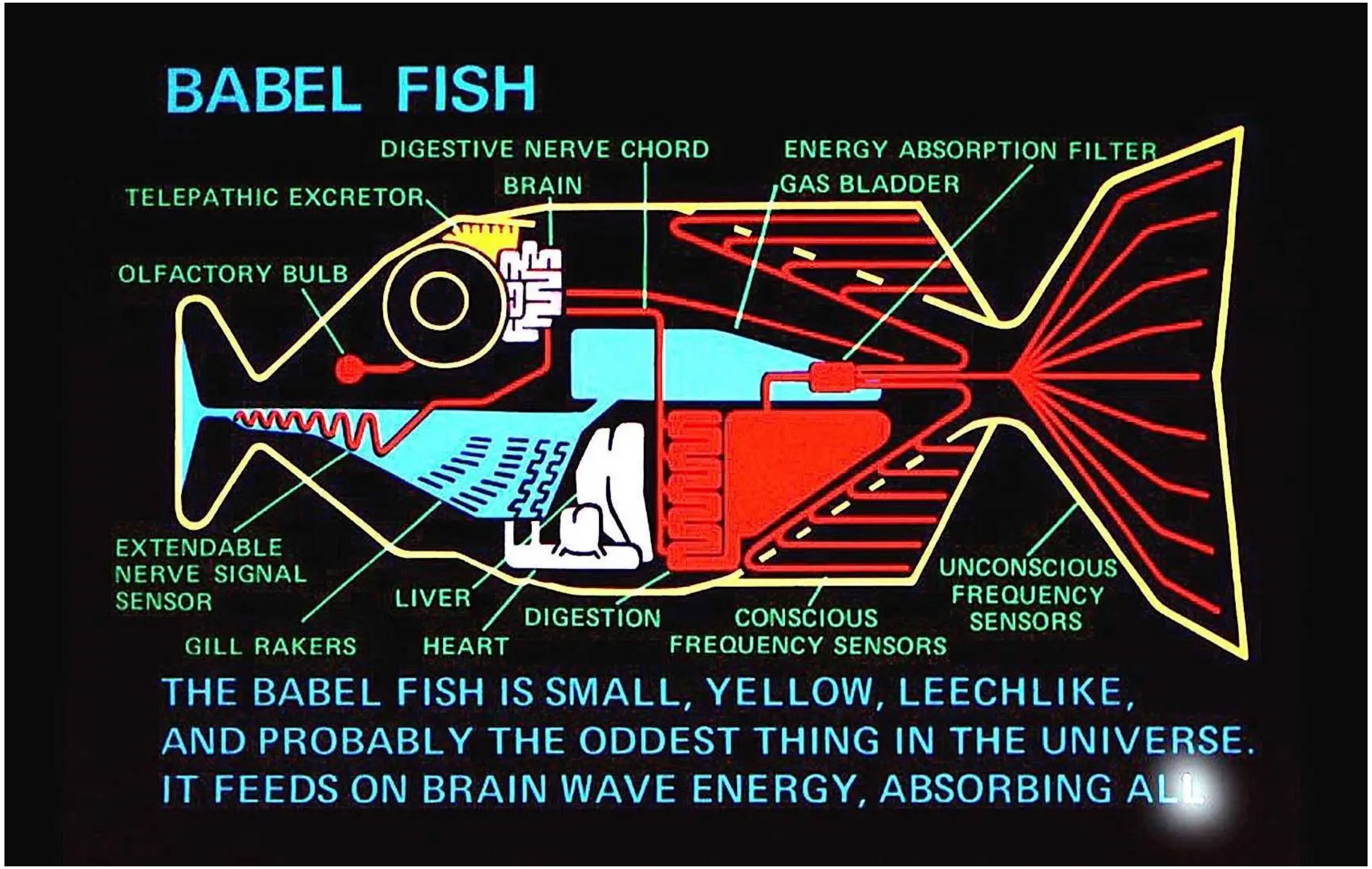
Previous work

Related Work

Previous work

- Douglas, Adams : The Hitchhiker's Guide to the Galaxy, Del Rey 1979
  - Inventor of the Babel fish

The Babel fees mechanism lets us translate fees from one currency to another!



Related Work

Existing solutions

Ethereum’s solution is to use the **Gas Station Network**

- Complicated structure built on top of Ethereum
- Requires additional infrastructure
- Requires changes to deployed contracts

Stellar’s solution is to use a **DEX** (distributed exchange)

- DEXs are complicated and susceptible to certain attacks (eg. front-running, sandwich)

Algorand’s solution is to use **meta-transactions** (incomplete transactions)

- Requires back-and-forth off-chain communication

We propose the Babel Fees mechanism as our solution

Why Babel Fees?

1. **Minimal complexity of changes to the platform**
   - minimal changes to transaction construction and processing
   - minimal changes to validity or effects of existing transactions
   - not a ledger-implemented DEX
   - simplicity of changes allows for **maintaining existing ledger security guarantees**

Why Babel Fees?

1. **Minimal complexity of changes to the platform**
2. **Minimal overhead for users**
   - no costly design, implementation, upgrade, or execution of contracts
   - no intermediate transaction construction steps or meta-transactions
   - no off-chain coordination required for transaction construction
   - offer is specified and accepted in the same block
   - no additional costs are associated with making or accepting offers as compared to primary-fee transactions

Why Babel Fees?

1. **Minimal complexity of changes to the platform**
2. **Minimal overhead for users**
3. **Versatility**
    - exchanges are not enforced by a fixed algorithm, giving users power to configure their selection process any time
    - low commitment : the users are not forced to commit to, or prepay for accepting any future fee offers until they submit the accepting transaction
    - can be used for smart contract fee coverage and spot swaps, as well as tightly batched transactions

Multi-Asset Representation

All on-chain assets are represented via **token bundles**, which are heterogeneous collections of primary and user-defined asset tokens, eg.

- **Identifier** is a unique identifier for each type of token

- Negative Ada quantity makes this bundle a liability

Identifier		Quantity
Swords	↦	1
GameCoin	↦	20
Ada	↦	-4

Now for the big reveal!

Limited	Non-persistent
Liabilities	Debt

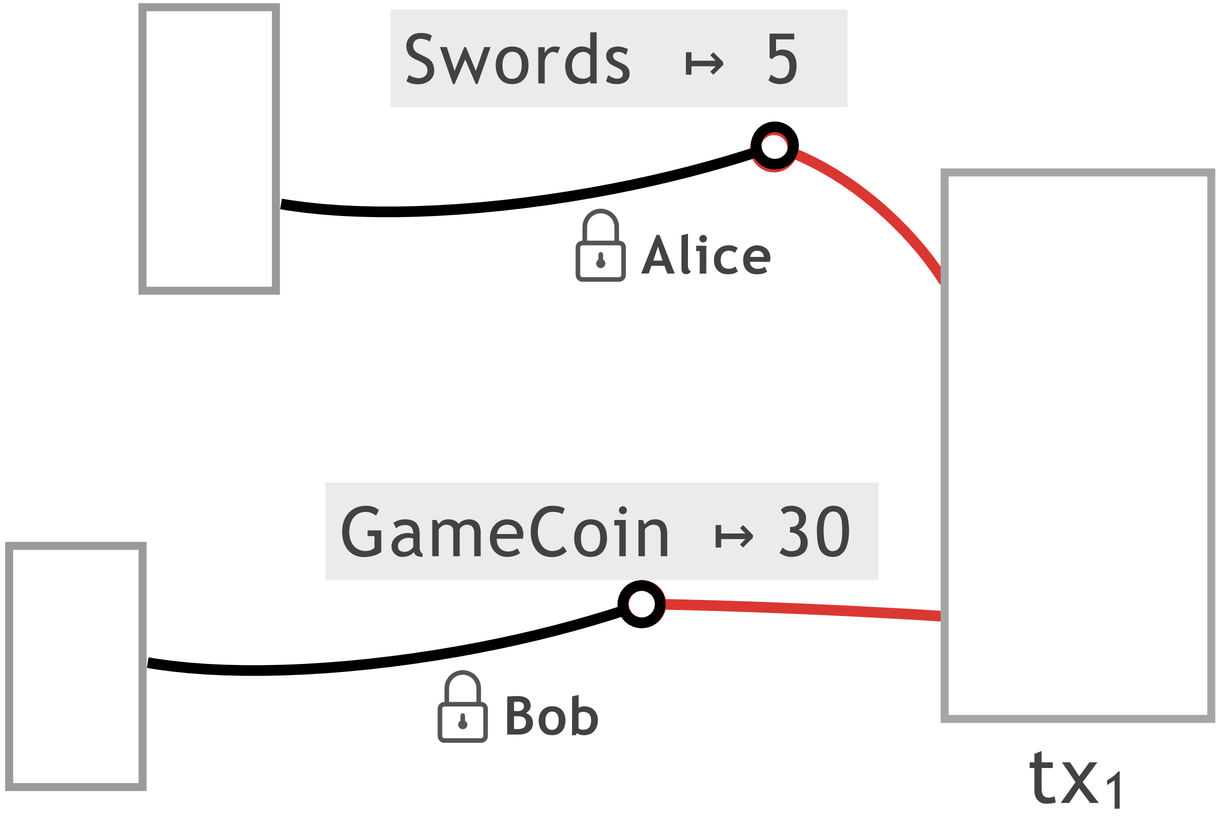
Batch Validity

Debt on the ledger must be resolved inside a batch

Batching is a way of ensuring that all liabilities are resolved in a valid ledger

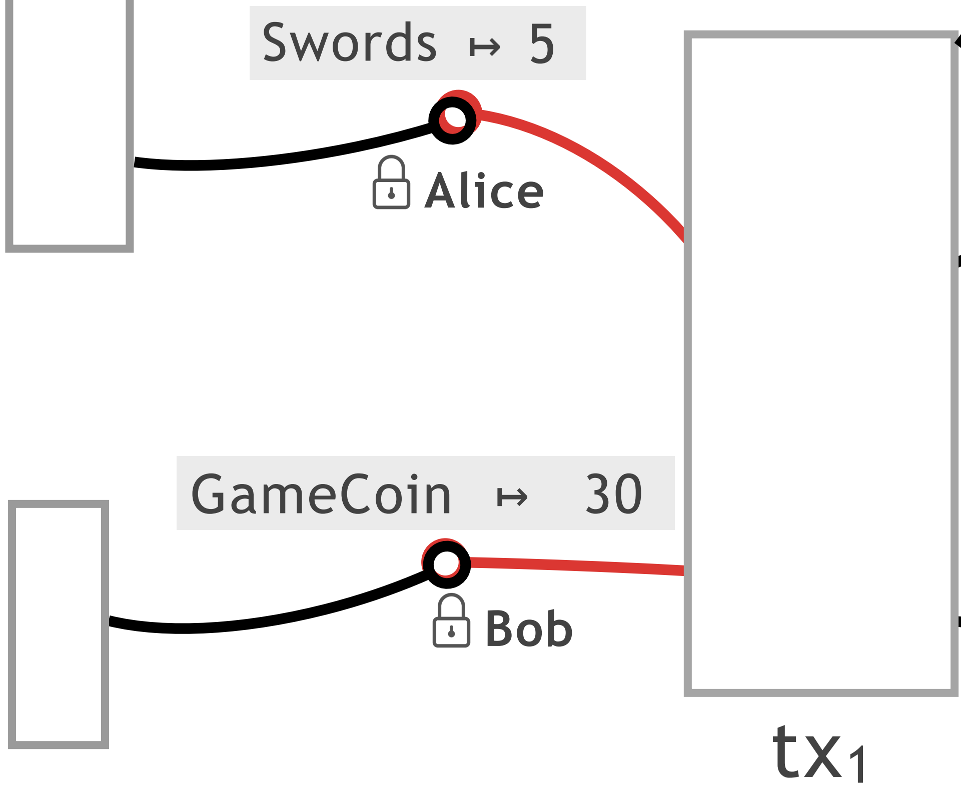
- Enforced by a change in the block validation rules, which now validate transactions in batches, rather than atomically
- Any list of transactions can form a batch so long as
  - each transaction in the batch is **conditionally valid**, i.e. valid but may contain liabilities
  - all liability outputs created inside a batch are consumed inside the same batch
- A natural batching strategy is checking that **each block forms a batch**

Example : Transactions with Liabilities



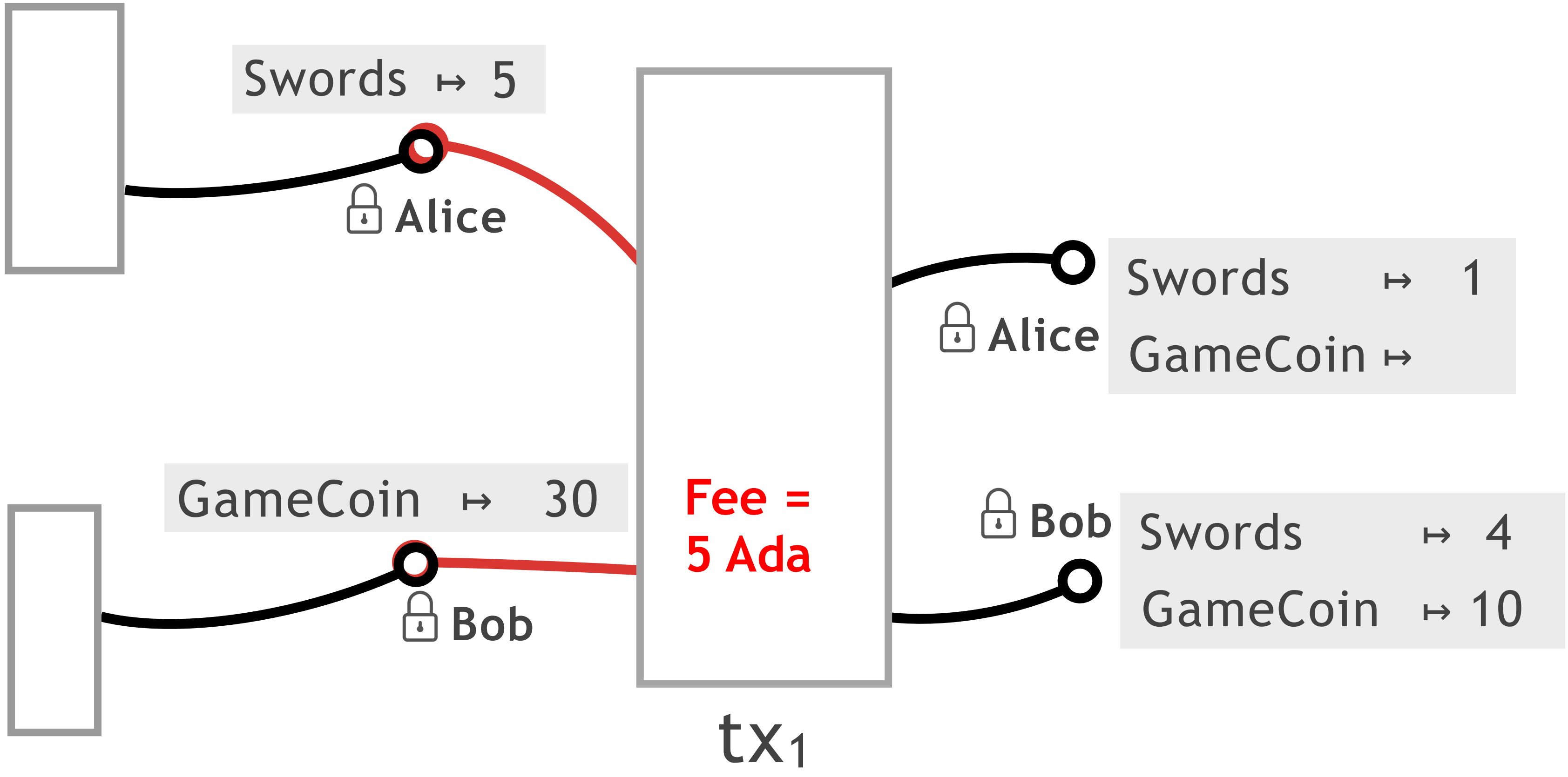
- Alice has 5 swords and wants to sell some of them to get more of the game's currency
- She saw advertised in the game's marketplace that Bob wants to buy some for 5 GameCoin per sword
- They construct a transaction

Example : Transactions with Liabilities



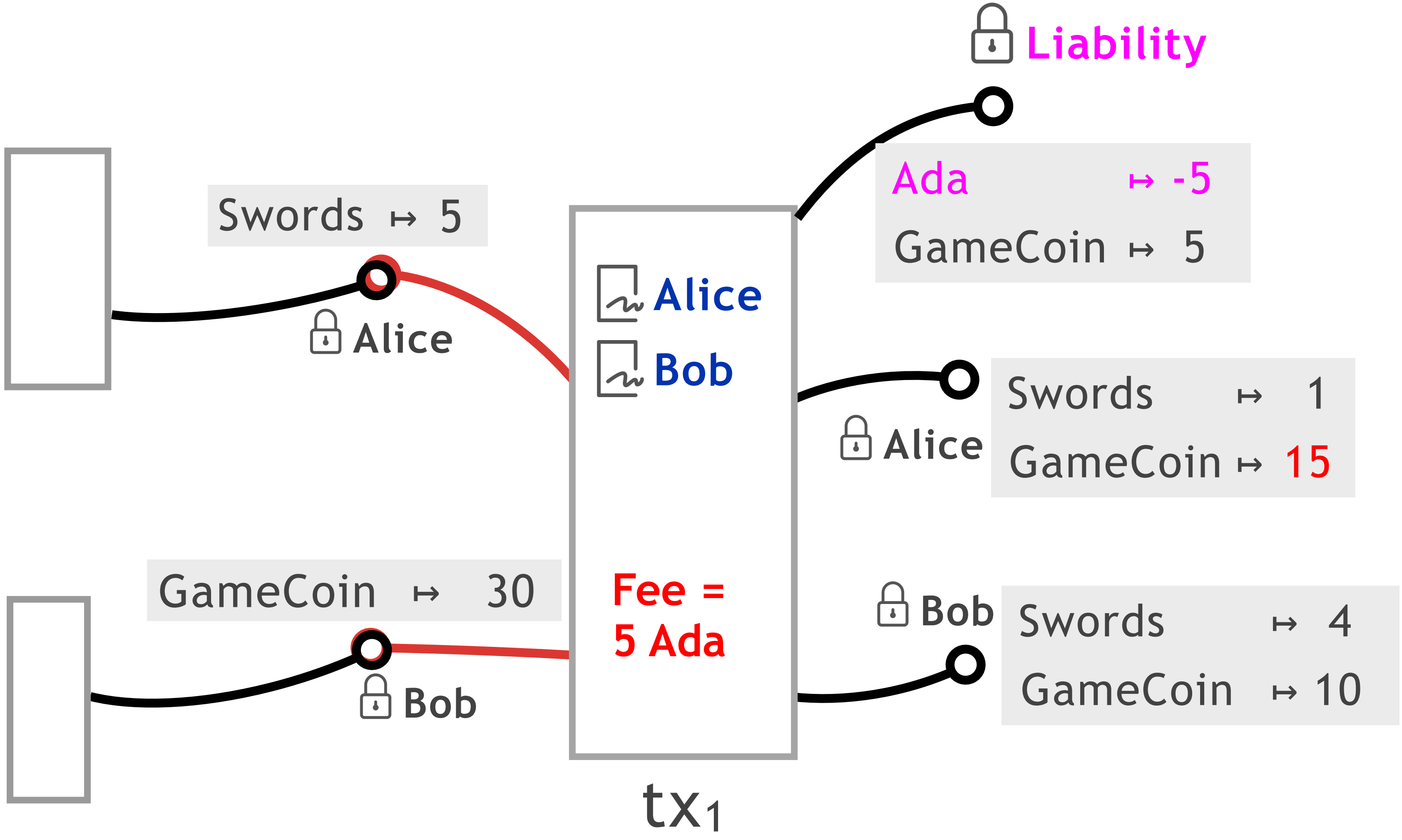
A 1-1 exchange
rate of GameCoin to Ada
is advertised by a block producer

Example : Transactions with Liabilities



A 1-1 exchange
rate of GameCoin to Ada
is advertised by a block producer

Example : Transactions with Liabilities

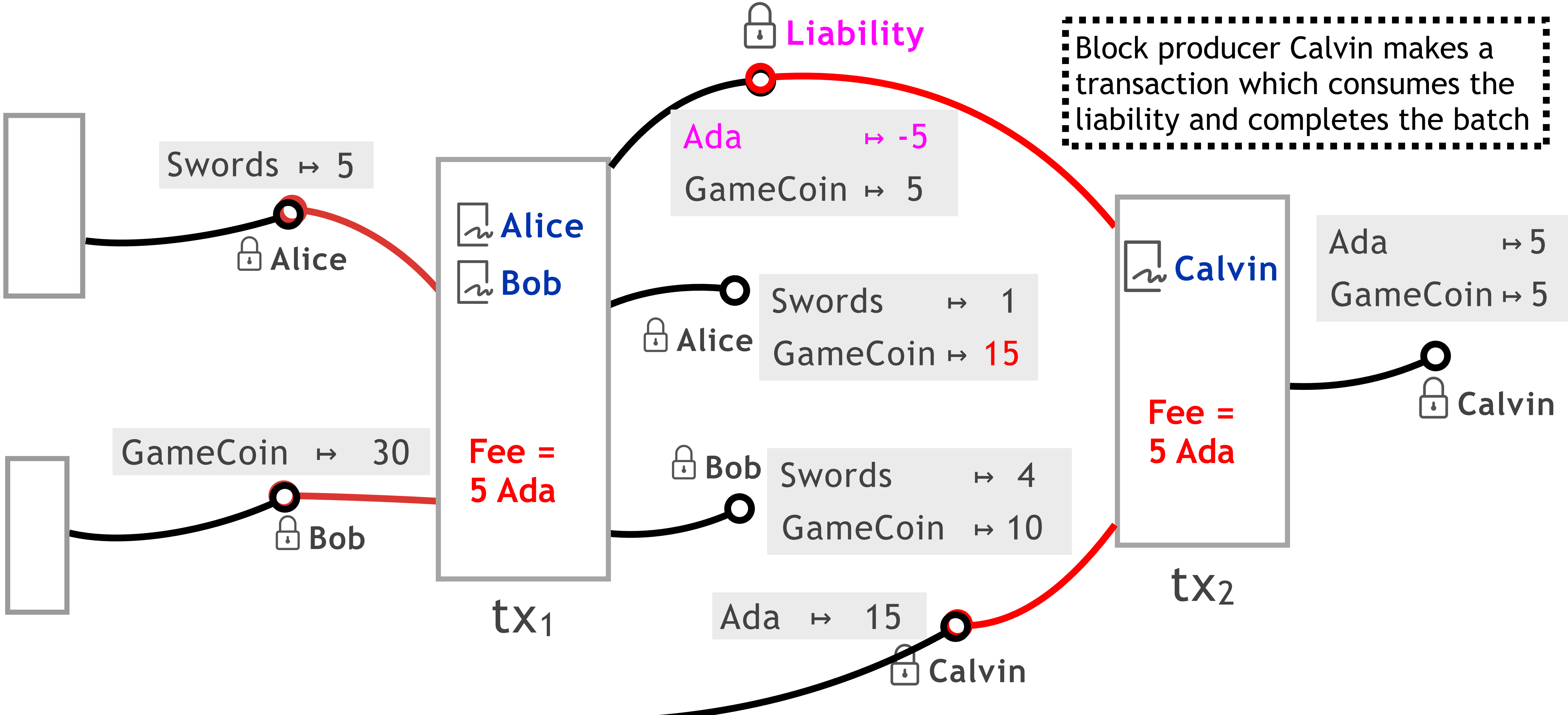


A 1-1 exchange rate of GameCoin to Ada Is advertised by a block producer

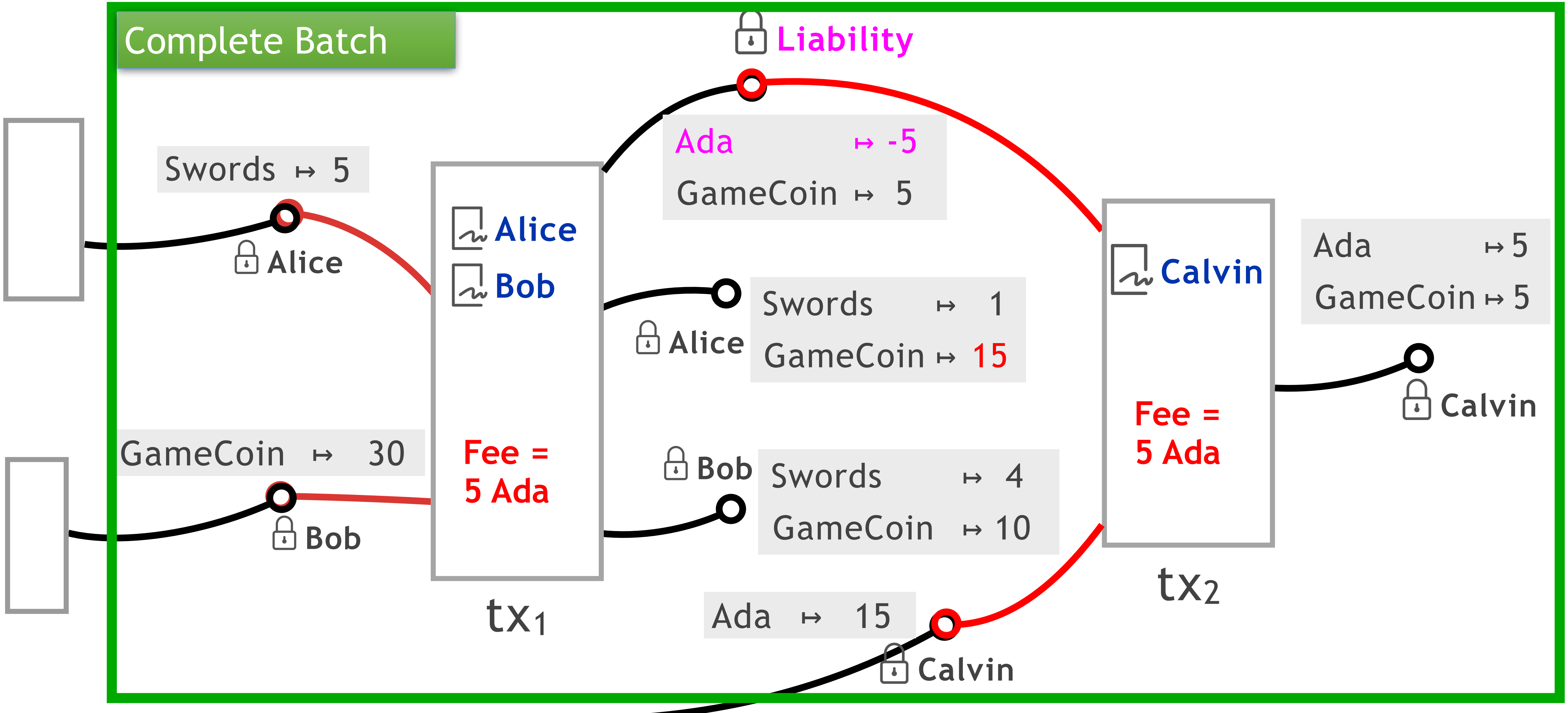
Alice uses 5 of her GameCoins as an offer to have the fee covered via a liability

This transaction cannot go on the ledger unless it is **batched** with another one, fulfilling the liability

Example : Transactions with Liabilities



Example : Transactions with Liabilities



Spot Market

A mechanism for personalized selection of Babel fee offers to maximize profit of the block producer

1. **Price discovery** : exchange rates from all sellers are published on an off-chain roster

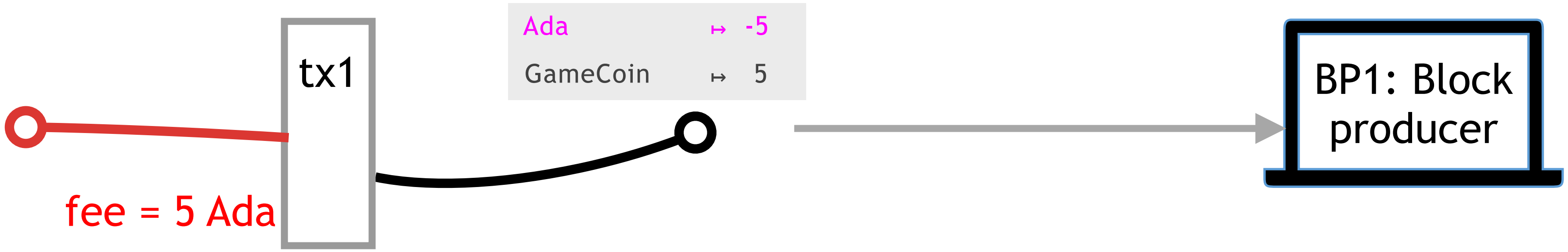
Token Type	Exchange Rate with Primary Currency
Swords	1 Sword for 50 Ada
Swords	1 Sword for 40 Ada
GameCoin	1 GameCoin 1 Ada

Spot Market

A mechanism for personalized selection of Babel fee offers to maximize profit of the block producer

1. **Price discovery**

2. **Sellers produce Babel offers (transactions with liabilities), and publish them to the network**
   - to decide what offers to make, sellers inspect the off-chain roster
   - for an offer of token *T* to be attractive to *P* % of buyers, the seller needs to choose a certain amount of that token, which depends on *P* and the minimum listed exchange rate for that token
   - liveness : if a Babel offer attracts at least one honest party, the accepted offer will be (eventually) published in the blockchain



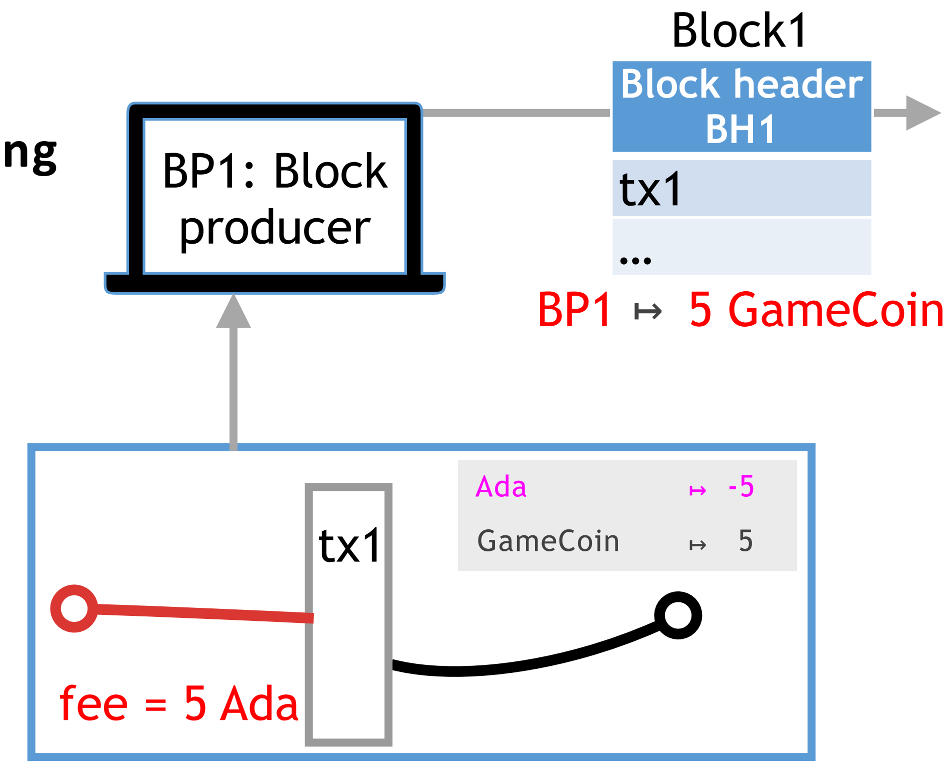
Spot Market

A mechanism for personalized selection of Babel fee offers to maximize profit of the block producer

1. **Price discovery**

2. **Sellers produce Babel offers (transactions with liabilities), and publish them to the network**

3. **A block issuer constructs a block of transactions by choosing from a set of available transactions called the mempool**
   - A rational block issuer tries to maximize the amount of primary currency earned by this block
   - We give a variation of the dynamic programming solution to the 0-1 knapsack problem to solve this problem in exponential time
   - We also give a polynomial time approximation



Other Applications of Limited Liabilities

More useful than just for fee coverage

1. **Atomic swaps**
   - broaden the idea of babel fees to offers for any kind of exchange, not just fees
   - liabilities enable us to break up the cooperative process of building a monolithic atomic swap transaction into a non-interactive two-stage process

Other Applications of Limited Liabilities

More useful than just for fee coverage

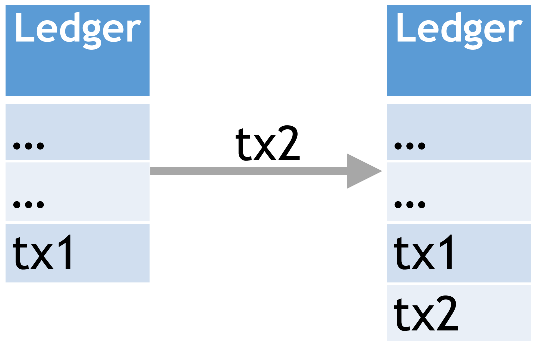
1. **Atomic swaps**
    - broaden the idea of babel fees to offers for any kind of exchange, not just fees
    - liabilities enable us to break up the cooperative process of building a monolithic atomic swap transaction into a non-interactive two-stage process

2. **Indivisibility**
    - Babel fee offers can be batched with any transaction accepting the offer
    - by changing the script or signature locking a liability output, it is possible to control what kinds of transactions are able to consume that liability
    - this allows users to form batches where transactions cannot be replaced with another transactions
    - scripts that require several transactions to run a program step to completion can make use of this

Next Steps : Develop DDoS Prevention Mechanism

Mempool transaction validation

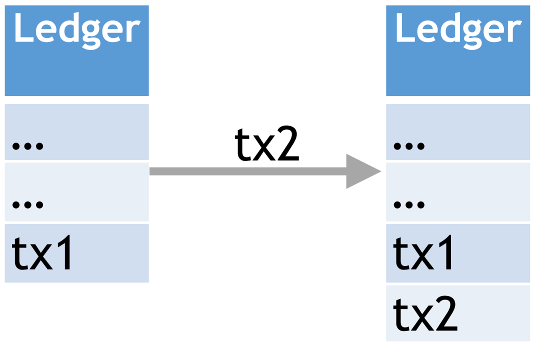
In the existing system :



Valid Valid

- Can validate transactions atomically to decide if they belong in the mempool

With Babel Fees :



Valid Conditionally valid

- Given a transaction with liabilities, cannot decide if it will ever be part of a valid batch
- **Mempool may be filled with such transactions indefinitely**

Next Steps : Develop DDoS Prevention Mechanism

Solution options

1. **Only publish transactions in completed batches instead of individually**
   - requires alternate method of publication for individual transactions with liabilities, likely off-chain

2. **Instruct relay nodes to only relay transactions to those nodes that have opted-in to accept transactions with certain liabilities**
   - requires a mechanism to maintain such lists at the relay nodes
   - may be tricky for indivisible batches

Next Steps

Follow the Process

1. Make a CIP
2. Formalize
3. Implement in the ledger, wallet, DBSync, etc.

Thank you for listening!

Questions??