Consumer
Person or entity interested in purchasing items from vendors.

Distributor
The distributor is a combination of smart contract and a managed service. Although it is not a 100% blockchain entity, it does not have the ability to cause any losses to the consumer or the vendors. It uses so-called 'off-chain' contracts, the mechanisms popularised by Bitcoin projects Lightning network and Duplex Micropayment Channels (though in Bitcoin it won't be possible until Segregated Witness is deployed and with it transaction malleability will be fixed).
There could be many distributors and they can compete. They'd probably earn some commission depending on the volume that they facilitate**.**

Payment channels - from Bitcoin to Ethereum
Micropayment channels described in [1] and [2], rely on the so-called MultiSig outputs in Bitcoin - those requiring more than one signature to be spent. Off-chain smart contracts are based on passing around half-signed and fully-signed transactions spending MultiSig outputs. This, in turn, requires access to the private key.
in Ethereum, an external transaction (i.e. a transaction not originated from a contract) is always signed by one private key. To mimic the Bitcoin off-chain smart contract functionality in Ethereum (not because there are no smart contracts, but because they are not scalable enough yet), one could try to emulate MultiSig transactions as invocations of a multiSig smart contract. Taking as an example a multiSig wallet https://github.com/ethereum/dapp-bin/blob/master/wallet/wallet.sol, one could create a transaction calling ‘confirm’ method on the wallet, sign it with the private key, and send it off the chain to another party. When another party wants to settle the channel, it relays the signed ‘confirm’ call, which will then appear on the blockchain. There is a small problem with that. In order to construct a transaction in Ethereum, one has to specify the value of “nonce” of the sending account, which is ever-incrementing counter attached to every account. Therefore, the transaction that has been created, signed, and given to another party, will have some “nonce”. The sender can then rescind at any time by creating another valid transaction which has the same nonce, effectively invalidating the one she had previously created and signed. This problem does not arise in Bitcoin MultiSig because half-signed MultiSig transaction will be ignored and will not change the state of the outputs (accounts).
Solution to this could be a ‘synthetic’ MultiSig in Ethereum, whereby the sender uses its private key to sign some piece of data, for example, split of the deposit between sender and the receiver. The data and the signature can be given to the recipient and at any moment presented to the smart contract. The smart contract can verify the validity of an ECDSA signature thanks to one of the pre-compiled contracts ECRECOVER.
The main disadvantage of such synthetic MultiSig is that it required the access to the private key from outside of a ‘standard’ Ethereum client, from the application (Dapp) itself. It might sound like not a big deal, but this disadvantage has a potential of closing many avenues of adoption. If the access to the private key is not required, then the application can be built on top of any Ethereum client APIs. Hardware wallets are designed to protect the private key, and one cannot realistically expect the manufacturers of such wallets to embed application-specific logic in them.
Unfortunately, it seems to be impossible to implement payment/state channels without at least some parties having to use the synthetic MultiSig, and therefore accessing their private keys outside of the standard Ethereum client. An obvious choice of such party is a distributor. In the description below, the distributor will use its private key to produce payment receipts for the consumer.

Solution - Merkle tree chequebook
The main idea is for the sender to pre-create a large Merkle tree of payments (each payment is a number of units to be payed + random salt). For example, one can generate payments for 1, 2, 3, ... , 1024 units, add salts to them, organize them into the Merkle tree and calculate the hash root. One can

think of them as leaves in a chequebook, with the difference that if multiple cheques are cashed in, the one with the highest amount supersedes those with the lower amounts.
To open such a payment channel, the payer creates transaction on the channel contract, depositing some ether, specifying the intended payee (also the only person who is allowed to settle the channel), the hash root of the Merkle tree chequebook, and the denomination of the unit (for example, 0.1 ether, so in our example, the channel will be capable of paying 0.1, 0.2, 0.3, … up to 102.4 ether). When the payer needs to add payment to the channel, she calculates the total settlement amount, finds it the Merkle tree, and reveals the Merkle proof (the path to the number+salt with some siblings). This proof could be verified by the smart contract to belong to the tree, because the root is known in advance. Since only the payee can cash in cheques, they can be revealed publicly, which obviously simplifies the infrastructure (no secure transmission is required).

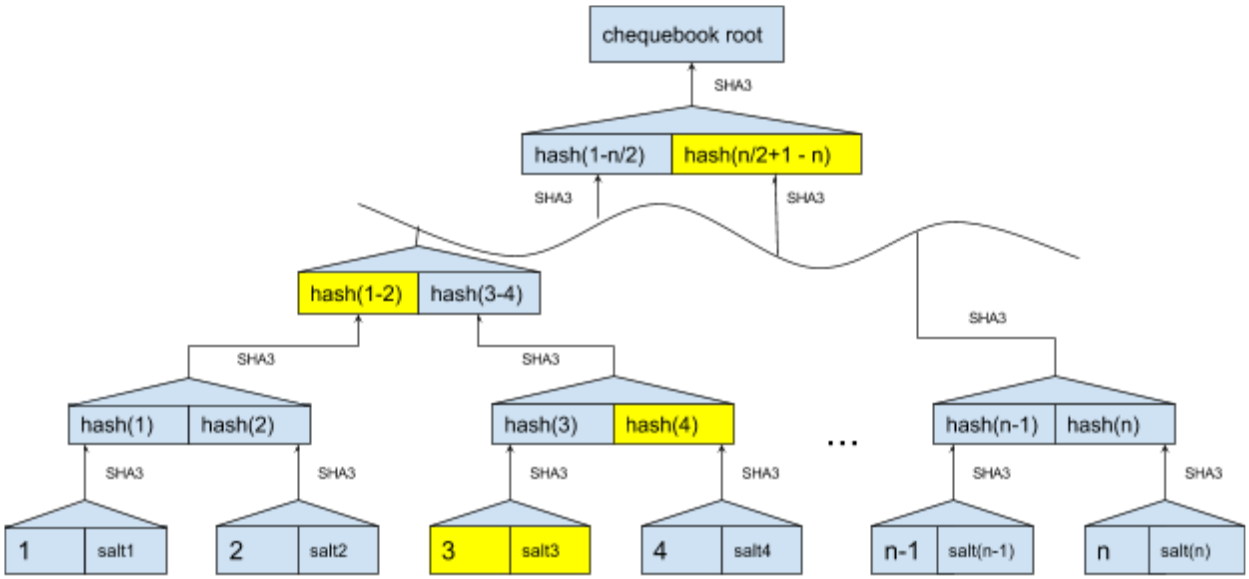

Figure 1: Construction of a Merkle tree chequebook

Figure 1 shows how a Merkle tree cheque book is constructed. All salt values are generated by a secure random number generator, since they need to be unpredictable. Highlighted are pieces of information that together constitute a cheque with value 3 - essentially this is a Merkle proof, allowing anybody (including a smart contract) to verify that the cheque was in the book at the moment of creation, and therefore is a commitment of the payer.

Payment receipts

In the payment channel described above, a single cheque pays for multiple items. However, the cheque does not contain any information about those items. This makes it is difficult or impossible for the payer to later settle any disputes if the payee did not honour its part of the off-chain smart contract. Therefore, the payer needs to extract a receipt for each item that she has paid for. Such a receipt has to:

- be non-repudiable (in our case it would be signed by the payee's private key, which can be verified with help of ECRECOVER pre-compiled contract),
- contain the amount paid for the item
- contain an unique identifier which is linked to the item (in our case, this identifier has to be present in the issued item that is eventually committed to the blockchain).

What should happen first - the payment (i.e. revealing of the Merkle proof of the cheque to the payee), or the receipt (i.e. creating, signing, and revealing the receipt to the payer)? If the payment happens first, we need to handle the situation when the receipt is not given. If the receipt is given first, we need to handle the situation when the payment has not been made.

Payment is made, but receipt is not given

If the payer had made a bulk payment for multiple items and she is missing multiple receipts, there is no way for her to prove (to the smart contract, our ultimate judge) that she has purchased that many items and that the price for each item was that much. The payee can always fabricate a receipt for just one item with the price equal to the total of the bulk payment that she has made. It can even be a receipt for an item that the payer never intended to purchase. That means that the smart contract judge should not in general give any weight to the receipts presented by the payee (because they can be easily fabricated), unless they are approved (or unchallenged) by the payer.

However, if the payer has kept all the receipts but is only missing one, she can present them to the smart contract and get the disputed amount reduced to the amount of that missing receipt. That amount would need to be refunded. To ensure that there is always at most one receipt is missing, the payer adopts the following strategy: only make the next payment if the receipt for the previous one is given.

But what if the payer does not submit all the receipts that she received, to get a bigger amount refunded? To prevent this, the payee needs to have the ability to present the receipts that the payer wants to withhold. To trust these receipts the smart contract needs to give the payer a chance to challenge them, and if they are left unchallenged, they are considered to be genuine.

To challenge a receipt presented by the payee, the payer needs to present its own version of the 'same' receipt, and that version will prevail. But what does the 'same' mean? In order to match them up, all the receipts need to have unique IDs. Moreover, these IDs need to be issued by the payer, one by one, for a specific item. For example, if the payer purchased 5 items, it should have given (revealed) 5 IDs to the payee, and before she proceeds to make the 6th payment, she needs to have received 5 receipts, one for each of those IDs. For the smart contract to be sure that these IDs are issued by the payer (and therefore constitute her commitment), they need to be somehow pre-committed to the blockchain. A very simple way of doing it is to organise them into a hash chain, and commit the root, as shown on the Figure 2:

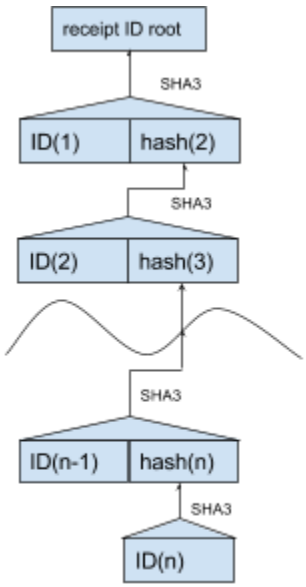

Figure 2: Construction of a receipt ID hash chain

The receipt ID root is provided by the payer (together with the chequebook root and the deposit) upon the creation of the payment channel. Only the IDs that are contained in this hash chain are considered valid IDs for the purpose of arbitration and refunds. After each payment, the payer reveals the next ID to the payee (and everyone else, so no secure transmission is required). The payee incorporates this ID into the receipt, together with the price and the reference to the item that is purchased (in our case, `itemId`). The receipt is then signed by the payee's private key and revealed to the payer (again, an

unsecure transmission could be used). If the payer follows the strategy of never revealing the next ID without getting the receipt for the previous one, the payee will have at most one valid ID.
From the point of view of the payee, even if she is following the protocol honestly, there is always a chance that the very last payment by the payer will be refunded if the payer pretends to never have received a receipt. That means that the item for this last payment cannot be issued without a possibility of a small loss to the payee. To obviate this, we will require the payer to always send a small extra overpayment after each receipt, as a confirmation of getting the previous receipt. This overpayment can (and will) be refunded when the payment channel is closed, but its presence prevents the previous payment to be reverted if the payee is honest. This trick allows the payee to issue the item at any time after the overpayment is made, instead of waiting for the next payment (which may never happen).

Receipt is given, but payment is not made
Imagine the payee produces the receipt first, reveals it to the payer, but then the payer does not pay (i.e. does not reveal the next cheque). In this situation, the payer will not get the item issued. However, now that the payer has one receipt too many, with the total amount greater than what she had paid. Knowing about such a possibility, the smart contract (judge) cannot trust that any receipt presented by the payer actually corresponds to a payment. That precludes the ability of effective arbitrage. For example, assume the payer has 6 receipts, for the amounts 1,2,3,3,2,1, and paid 11 (1+2+3+3+2) in total. If it is known that only 5 receipts were paid for, it is still not possible to be sure whether the total amount should be 11, or, let's say, 9 (1+2+3+2+1). It depends on which 5 out of 6 receipts the payer decides to present.
Solution to this problem is to enforce that the exact order in which receipts were given is known. For example, if we know that the receipts were given in this order: 1,2,3,3,2,1, then it is clear that 5 out of 6 receipts should have the amount 11, and not 9. The hash chain of receipt IDs shown in the Figure 2 can be used for that. The payee uses receipt IDs from the hash chain when revealed by the payer, in exactly the same order as they appear in the hash chain.
This approach (first receipt, then payment) also eliminates the problem of creating receipt for the wrong item. The payer will not reveal the next cheque if the receipt she received is for the item she did not want. Because receipt IDs cannot be reused, it follows that in the case where the payer does not pay after getting a receipt, the payment channel needs to be completely settled and reset (via transactions on the blockchain).
Comparison of two approaches ("payment first, receipt after" and "receipt first, payment after"), the second one - "receipt first, payment after" leads to a simpler design, because it does not require overpayments, and prevents the payer from paying for the wrong item.

Notes on scalability of the payment channels
Using the payment channels described above, with Merkle tree chequebook and receipt ID hash chain, the distributor will act as a payee, and multiple consumer will act as payers.
Each consumer performs its own set up (generates chequebook and receipt ID chain, and commits them together with the deposit), so there is 1 on-chain transaction per customer.
Once the channels are set up and open, the distributor can perform an intermediate settlement on multiple channels (across multiple consumers) in a single transaction (as long as it does not go over the gas limit). In order to reduce the amount of tied-up funds, the distributor might want to perform such settlements quite regularly, so the number of transactions will be proportional to the frequency of such settlements, and to much lesser extent - to the number of consumers operating simultaneously.

Vendor channel
It is established between each pair of interacting vendor and distributor. Payments flow from the distributor to the vendors, and sold items flow back. As we agreed that the distributor will have to have access it is private keys, she can produce and sign payments of any amounts directly, without the need for a Merkle tree chequebook. On the other hand, the vendor, as well as the consumer cannot be

considered as a ‘blockchain specialist’, and cannot be required to use her private key in a non-standard way.
Using the same idea as in the receipt ID hash chain, the vendor can pre-fabricate the items, place them in a hash chain, and submit the root with the setup transaction. Since the vendor may in general sell multiple types of items, and they can be purchased independently of each other, each item type must have its own hash chain. In order to further economize on the blockchain storage, all these chains can be merged into a single Merkle tree, so that only one root needs to be submitted, as shown on the Figure 3.

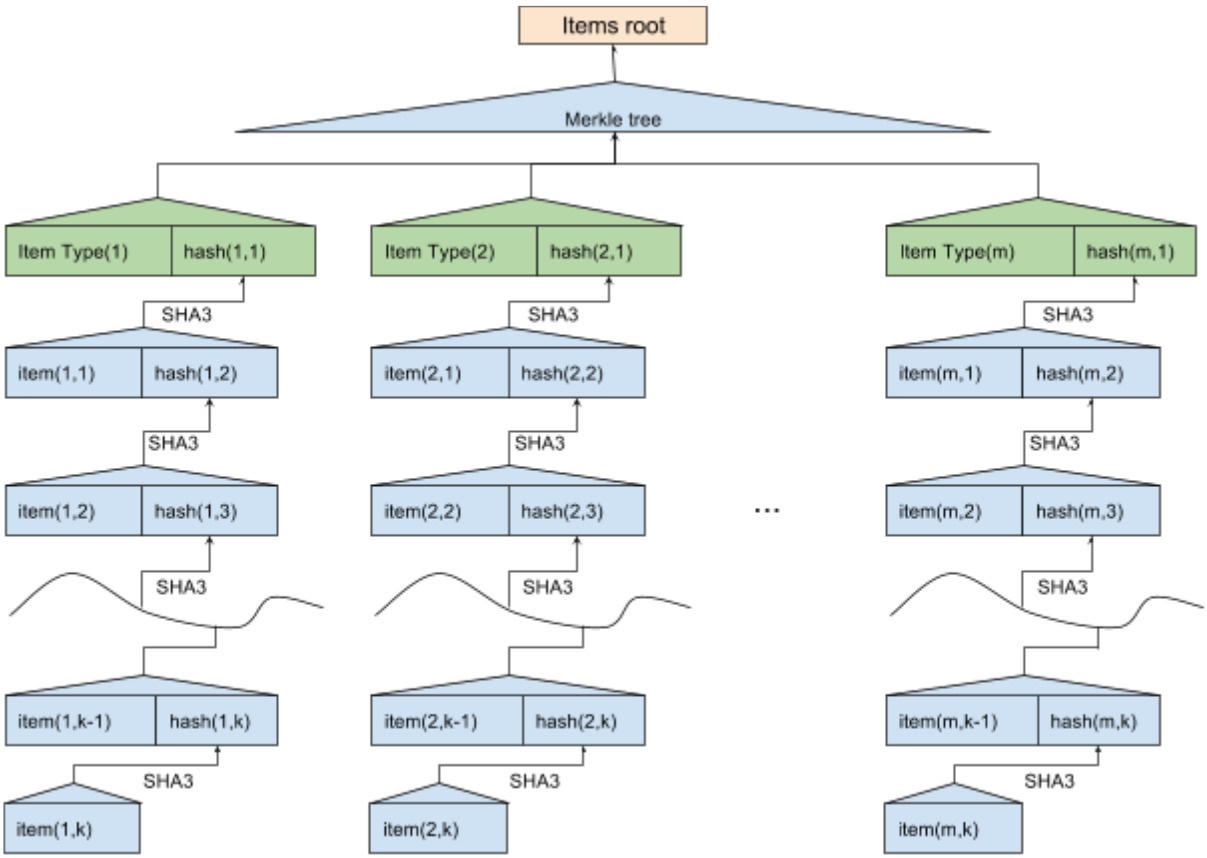

Figure 3. Item tree with hash chains

Because the items root is committed by the vendor during the setup, it can be viewed as both commitment and authorisation. Therefore, issuing individual items does not require any more transactions, but simply revealing of the parts of the Items tree and its chains. It means that issuance is instantaneous.
However, there is small problem. For the consumer, the item as a piece of data would comprise the merkle proof of the corresponding item type as well as the entire hash chain leading to the required item. If the number of items issued by the vendor is large, this would make all the items large too.
The solution is to organise items themselves, and not only the item types, into a merkle tree, as shown on the Figure 4.

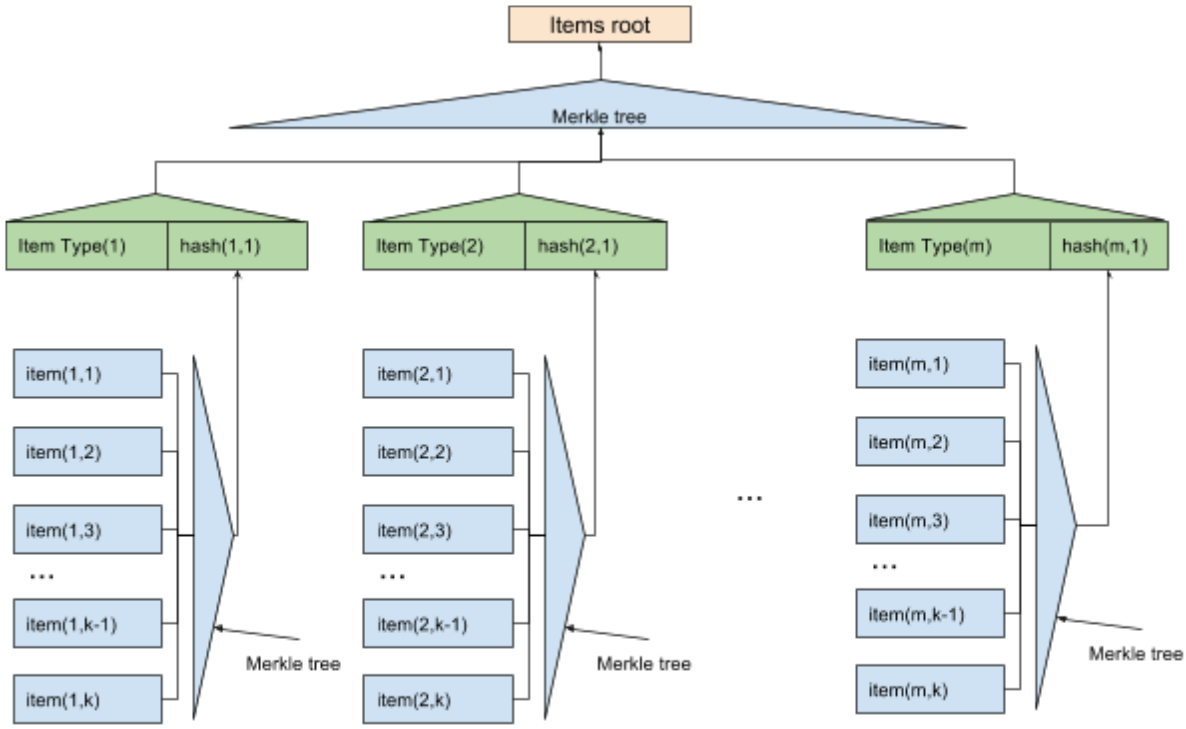

Figure 4. Items tree without hash chains

Each item leaf needs to be accompanied by a random salt before the inclusion into the Merkle tree. That allows inclusion of the item into a receipt without revealing the Merkle proof. With this approach, the size of an item would be logarithmic to the total number of pre-generated items.

Notes on scalability of items channels

One on-chain transaction from the vendor is required to set up the items channel. No more on-chain transactions are required from the vendor until the items of a certain type get depleted, and the new items root needs to be submitted. The distributor can setup the payment channels to multiple vendors in a single transaction.

Typical process of purchase of an item

1. Vendors sets up items channel as described above, i.e. pre-generates the items of all items types she wants to sell, and submits the Items root to the smart contract. No more on-chain transactions from the vendor.
2. Consumer sets up the payment channel as described above, i.e. pre-generates Merkle tree cheque book and receipt ID hash chain.
3. Consumer interacts with the items type smart contract via a suitable interface, selects an item type and obtains `itemTypeId`. This identifier also refers to the price and other properties of the item.
4. Consumer takes the next `receiptId` from its receipt ID hash chain and reveals this `receiptId` to the distributor. Consumer also sends the selected `itemTypeId`.
5. Distributor checks that the `receiptId` is the next item in the consumer's hash chain and asks the vendor for the `itemId` belonging to the selected type.
6. Vendor looks up an `itemId` from its items tree, which has not been used yet, and sends it back to the distributor (without revealing the corresponding random salt and the Merkle proof for this `itemId`).
7. Distributor generates a receipt, binding together `itemTypeId`, price to be paid for the item, `itemId` provided by the vendor, and the `receiptId` provided by the consumer. Distributor signs this receipt with its private key and gives it to the consumer.

8. Consumer checks that the `itemTypeId`, price and the `receiptId` are all correct, and the signature of the distributor verifies. Consumer selects from its Merkle tree chequebook the cheque covering the total amount spent so far, and reveals to the distributor its salt and its Merkle proof.
9. Distributor considers the payment successful, and generates its own payment for its payment channel with the vendor. Distributor signs its payment and sends it to the vendor.
10. Vendor, having received the payment from the distributor, reveals the salt and the Merkle proof for the `itemId` that has been provided earlier. Now the item is issued, and the receipt produced by the distributor earlier, binds it with the payment made by the consumer.
11. Distributor hands the salt and the merkle proof for the item to the consumer.

Arbitration and refunds

Note that the last two steps in the purchase process (revealing the Merkle proof and the salt for the item) and not followed by any conditional actions or confirmations. What if one or both of these steps are not performed? First of all, it is easy to prove that something has been revealed, but it is not possible to prove that something has NOT been revealed.
If the consumer wants to claim that an item that she purchased has not been revealed to her, she would need to present the receipt that satisfies all of the following conditions (they are verified by the smart contract on the submission):

- Receipt ID is one of the IDs in the hash chain of the consumer's payment channel.
- Total payment on that payment channel is equal or greater than the cumulative amount of all the receipts in the hash chain up to the receipt ID in question, inclusive (it means that the consumer will also have to present all the receipts in the hash chain up to the one in question). It could be that the consumer got the receipt but did not pay for it yet - in this case she cannot make a claim.
- Receipt contains the `itemId` and and the receipts are properly signed by the distributor.

A claim is a transaction sent to the arbitration smart contract. There will be a time limit on when the claim can be made, so that the distributor's funds are not tied up indefinitely.
Once a claim is submitted, the distributor has a certain time (for example, 24 hours), to challenge the claim, by submitting the salt and the Merkle proof of the `itemId` in question. If the submitted challenge is valid, the claim is rejected and no refunds are given.
If no valid challenge is submitted within the time limit, the consumer can perform the refund by calling the arbitration contract again.
If we require the consumer to never send any further payment until all the previous items are issued (she received salt and the merkle proof), the scope of arbitration can be reduced to only the very last item purchased by that customer. This reduction of scope in turn reduces the amount of tied-up funds that distributor has to keep in the contract in case of arbitration. More specifically, distributor will have to keep (will not be able to withdraw) for each customer the amount equal to the price of the last purchased item plus the arbitration allowance.
Because the challenge of the claim cost gas, it needs to be covered by the consumer in the case when the challenge was successful. Therefore, the claim can only be accepted if it comes with a fee covering the cost of challenge. If the claim is successful, the fee is refunded to the consumer.
If the claim is successful, the consumer gets back not just her refund by a small amount (arbitration allowance) that is designed to make it costly to run 'scam' distributors.

Arbitration between distributor and vendor

What if the distributor is prepared to honour her obligations, but is prevented from doing so by the vendor? Distributor needs to have a similar recourse to arbitration against the rights vendor. To make this arbitration possible, the payments produced by the distributor, need to embed all the `itemId`s that the payments are made for. This is easy to do since the distributor can sign arbitrary messages with her private key.

Communication between distributors, rights holders and customers

Initially, distributor can run an HTTP server, responding to the requests of both rights holders and the consumers. In more advanced design, some meta-protocol features of Ethereum's devp2p (or whisper) could be used (requires feasibility check).

References

1. Christian Decker, Roger Wattenhoffer. A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels. 17th International Symposium, SSS 2015, Edmonton, AB, Canada, August 18-21, 2015
2. Joseph Poon, Thaddeus Dryja. The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments. 2015-2016.