Universal Payment Channels

Jehan Tremback
jehan.tremback@gmail.com

November 2015
v0.4

Abstract

Payment channels are a type of private ledger, backed by assets held by a bank, payment processor, or blockchain. In this paper, we use payment channels to specify a payment network which allows for instant transfers of conventional currencies, cryptocurrencies, and any other kind of asset which can be owned without physical possession.

The network can, in many circumstances, convert one asset type into another without using exchanges. For example, Dollars to Dogecoins, or Bitcoins to grain futures. Payments are made directly between nodes in the network, without involving the entities or blockchains backing it. This greatly reduces load on these entities and blockchains, allowing for massive scalability.

The heart of this network is the Reactive Payment Routing (RPR) protocol, based on the packet routing protocols that power the internet. RPR allows the network to automatically find the cheapest route for any payment. Because of this, RPR is also able to find the best exchange rate between any two asset types used on the network.

UPC's payment channels also can process Turing-complete "smart conditions", making the network extensible and allowing developers to create a new class of payment channel applications.

Introduction

This paper concerns a payment network called Universal Payment Channels, or UPC. UPC can handle transfers of any type of conventional or crypto currency, as well as physical or virtual goods, as long as these goods can be considered owned without physical possesion and kept in escrow. Furthermore, UPC routes payments across stores of value, with the network automatically arriving at the market exchange rate.

The vast majority of UPC payments are made without interfacing with any third party ledger, such as a bank or blockchain. This means that there can be an almost unlimited amount of payments that do not put any strain on bank servers, or add anything to the blockchain. The only time that the bank or the blockchain is involved is when a network participant wants to take money out of the network, or put money into it.

This makes it very easy for application developers to interface with UPC. Instead of interfacing with a bank or payment processor, the only thing that an application needs to do is to send cryptographically signed messages on behalf of the user. For example, there is software in development using UPC to allow internet backbone routers to pay one another per-packet, many times each second. These payments can be sent or received in dollars, Dogecoins, or any other currency. Even though the volume of payments created by this type of software is oceanic, the banks and blockchains backing up the network are rarely contacted.

UPC consists of a series of "channels" between network participants. A channel is a private ledger arrangement between any two parties and a blockchain, a bank, or some other kind of institution. It allows the parties to exchange payment trustlessly, by sending updated ledger balances to one another (not to the bank or blockchain). At any time, each of the participants can be confident that they will be able to retrieve every cent or Satoshi that they are owed.

UPC's key innovation is Reactive Payment Routing (RPR), a routing protocol inspired by the protocols that power the internet. UPC uses RPR to route payments across interconnected channels, finding the cheapest path.

The Internet is a *packet-switched network*. Any computer on the Internet can reach any other computer, and the network grows organically and routes around damage. Routing protocols ensure that each packet takes the fastest path through the network.

UPC is a *payment-switched network*. Any network participant can pay any other, and UPC grows organically and routes around damage. UPC ensures that each payment takes the cheapest path through the network.

Related work

Payment channels are, at root, based on the idea of commercial credit. One of the main purposes of commercial credit is to allow two parties to consolidate a large number of smaller payments into one larger payment which is made periodically. Clearinghouses extend this instrument by placing funds in escrow so that two parties can exchange a large number of transactions without having to trust each other.

Some of the first formalization of the role of a clearinghouse into the concept of a payment channel occurred in the Bitcoin community with Mike Hearn's work on micropayment channels[2][3], Alex Aakselrod's work on chained micropayment channels[4] and C. J. Plooy's system Amiko Pay[5]. Further innovation appeared with Poon and Dryja's Lightning Network[7] and Decker and Wattenhofer's Duplex Channels[6]. All of these protocols are designed for Bitcoin, whose limited scripting capabilities demand complicated specifications. Interledger[1] is the only protocol we know of specifying a multihop payment channel system generalized across stores of value. It is a bit more complex than the protocol in this paper, but includes a formal specification. Zackary Hess's work in Flying Fox[11] deserves special mention, because his channel specification forms the basis of the one in this paper.

None of the above work includes the two key innovations in this paper:

Turing-complete **Smart Conditions** (“smart contracts” are a similar concept), and the Reactive Payment Routing (RPR) protocol, which is necessary for an actual working payment network. RPR is partially based on Ad hoc On-Demand Distance Vector Routing (AODV)[8], however it is simpler and provides full network anonymity for all nodes participating in a payment route.

1 Basic channel

A channel is created when two parties create and sign an **Opening Transaction** to put money from both parties in escrow with an institution such as a bank, or lock up coins on a blockchain. The money is put into escrow with the understanding that at some point in the future it will be transferred back to the parties upon receipt of an **Update Transaction** signed by both parties. This **Update Transaction** has a special number on it called a “nonce”, and specifies two additional actions to take before transferring the funds back to the parties:

1. Adjust the amounts that both parties have in escrow, lowering one amount and raising the other by the same amount. This effectively transfers funds from one party to the other.
2. Wait for a certain **Hold Period** before releasing the funds to back to the parties. If someone gives you another **Update Transaction** signed by both parties, and this transaction’s nonce is higher, throw the current **Update Transaction** out and use the new one, restarting the **Hold Period**.

Alice and Bob exchange **Update Transactions** back and forth, changing the amount of funds to be transferred to make payments to one another. Each time they make a new **Update Transaction**, they increment the nonce. To accept a payment, both of them sign it.

- If Bob disappears, Alice can post the last signed **Update Transaction** and collect the money owed her without Bob’s involvement, after the hold period is over.
- If Bob tries to cheat by posting an old **Update Transaction** where he’s doing better than he is currently, Alice can post the latest **Update Transaction**, which will override the old one. As long as Alice checks whether Bob has posted an old **Update Transaction** at least once every **Hold Period**, she can post the latest **Update Transaction** and stop him from cheating.

1.1 Opening Transaction

A channel is opened with an **Opening Transaction**. The **Opening Transaction** serves to identify the channel and the parties, and place the money in escrow. This could be sent to a bank, or supplied to a smart contract on a blockchain.

Opening Transaction:

- **Party 1:** Public key or other signature verification information for one of the participants.
- **Party 2:** Public key or other signature verification information for the other participant.
- **Amount 1:** The amount of money that **Party 1** has placed in the channel
- **Amount 2:** The amount of money that **Party 2** has placed in the channel

Signature 1: **Party 1**'s signature on **Opening Transaction**

Signature 2: **Party 2**'s signature on **Opening Transaction**

1.2 Update Transaction

Update Transactions are sent back and forth between **Party 1** and **Party 2** and serves to transfer the money. Only the last of these **Update Transactions** should be posted to the bank or blockchain. When an **Update Transaction** is posted, it always results in the closure of the channel (whether it is honored, or a higher-nonced **Update Transaction** is honored).

Update Transaction:

- **Nonce:** A number which is incremented with each new **Update Transaction**.
- **Net Transfer Amount:** The amount of money to transfer from **Party 1** to **Party 2** (can be negative).
- **Hold Period:** An amount of time (or number of blocks) to wait before closing the channel and transferring funds, after a **Update Transaction** has been published

Signature 1: **Party 1**'s signature on **Update Transaction**

Signature 2: **Party 2**'s signature on **Update Transaction**

1.2.1 Making payments

To make payments to one another, Alice and Bob pass signed **Update Transactions** back and forth.

If Alice wants to pay Bob, she adjusts the **Net Transfer Amount**, signs the **Update Transaction**, then passes it to Bob. None of this involves the **Update Transaction** being shown to anyone else, and can happen instantly. Alice and Bob can do this as many times as they want.

To actually claim the funds, either party posts the latest **Update Trans-**

action. After **Hold Period** is over, the channel closes: the **Net Transfer Amount** is subtracted from **Amount 1** and added to **Amount 2**, and the amounts are transferred back to the accounts of the participants.

This means that if Alice disappears or becomes uncooperative, Bob can still get his money out by posting the last valid **Update Transaction** he has and waiting for the **Hold Period** to end.

1.2.2 Stopping cheaters

If a **Update Transaction** with a higher **Nonce** is published before the **Hold Period** ends, it overrides the older **Update Transaction**.

If Bob tries to cheat by publishing an old **Update Transaction** where he has a higher amount than he does currently, Alice can simply publish the newer **Update Transaction**, which will have a higher **Nonce**.

Another option is to punish Bob by transferring all his funds to Alice if she is able to post a higher-nonced **Update Transaction** after him. This makes a cheating attempt riskier and may be beneficial in some situations.

2 Smart Conditions

Smart Conditions are pieces of Turing-complete code that are evaluated by the bank or blockchain during the **Hold Period**. They can return either true or false when supplied with a piece of data, which is referred to as a **Fulfillment**. They have an associated **Conditional Transfer Amount**, which is added to the channel's **Net Transfer Amount** if the **Smart Condition** returns true.

Update Transactions can have a list of **Smart Conditions**. A **Smart Condition** consists of the **Conditional Transfer Amount** and the **Function** which takes an argument and returns either true or false. The **Function** does not have any side effects, it is a pure function.

Update Transaction:

Nonce: A number which is incremented with each new **Update Transaction**.

Net Transfer Amount: The amount of money to transfer from **Party 1** to **Party 2** (can be negative).

Hold Period: An amount of time (or number of blocks) to wait before closing the channel and transferring funds, after an **Update Transaction** has been posted.

- **Conditions:**
  - **1:**
    - **Conditional Transfer Amount:** Add this to the channel's **Net Transfer Amount** if **Function** returns true.
    - **Function(argument):** Takes an argument and returns either true or false
  - **2:** ...

Pieces of data called **Fulfillments** can be posted during the **Hold Period**. These act to "fulfill" the "conditions". **Fulfillments** only need to be signed by one of the channel participants.

Fulfillment:

- **Condition:** Which condition does this fulfill?
- **Argument:** Data with which to evaluate the **Smart Condition**.

When a **Fulfillment** is posted, it is evaluated by the corresponding **Smart Condition**. If the **Smart Condition** evaluates to true, the **Conditional Transfer Amount** is added to the channel's **Net Transfer Amount**, and the **Smart Condition** is removed from the list.

2.1 Gas

Notice that one channel participant could send the other a **Smart Condition** that resulted in an infinite loop or other excessive use of resources. Blockchain-based smart contract systems like Ethereum[9] or Tendermint[10] use a concept of "gas" where each step of code execution costs a small amount. Execution aborts if there is insufficient gas. Such a gas scheme could be specified here, but we believe that it is an implementation detail, and outside of the scope of this specification.

Whatever the gas scheme used, nodes could be required to pay gas upon posting a **Fulfillment**. This way incentives are aligned so that the party who would like a certain condition evaluated pays for it.

2.2 Using Smart Conditions

Smart Conditions can be used to implement complex logic over channels to give them enhanced capabilities. Here is how Alice and Bob would implement a "hashlock" **Smart Condition**. Specifically, Alice wants to guarantee that she will transfer 32 coins to Bob if he can supply a string (referred to as a **Payment Secret**) that hashes to "59A1904325CCB". Alice is **Party 1** and Bob is **Party 2**.

Alice to Bob

Update Transaction:

- **Nonce:** 12
- **Net Transfer Amount:** -34
- **Hold Period:** 8
- **Conditions:**
  - **1:**
    - **Conditional Transfer Amount:** 32
    - **Function(secret):**

```
if sha3(secret) is equal to
"59A1904325CCB", return true;
else return false
```

Signature 1: Alice's signature on **Update Transaction**

2.2.1 Closing the channel

If Bob wants to close the channel at this point, he posts the **Update Transaction**, along with his and Alice's signatures. To fulfill the **Smart Condition** and have the **Conditional Transfer Amount** added to the channel's **Net Transfer Amount**, Bob must post a **Fulfillment** that causes the **Smart Condition** to return true during the **Hold Period**. Note that the **Fulfillment** only needs to be signed by the party posting it.

*Along with the above **Update Transaction**, Bob posts*

Fulfillment:

- **Condition:** 1
- **Argument:** "theSecret"

Signature: Bob's signature on **Fulfillment**

2.2.2 Fulfilling the condition without closing the channel

Of course, most of the time Bob doesn't want to close the channel right away. Bob can now prove that he could unlock the money if he wanted, so Alice might as well adjust the channel's **Net Transfer Amount** as specified by the **Smart Condition**. Bob sends the **Payment Secret** to Alice, who adjusts the channel's **Net Transfer Amount**, increments the **Nonce**, removes condition 1, and signs a new **Update Transaction**.

Bob to Alice

Fulfillment:

> **Condition:** 1
>
> **Argument:** "theSecret"

Signature: Bob's signature on **Fulfillment**

Both sign

Update Transaction:

> **Nonce:** 13
>
> **Net Transfer Amount:** -2
>
> **Hold Period:** 8

Signature 1: Alice's signature

Signature 2: Bob's signature

2.2.3 Canceling the condition

Similarly, Bob can inform Alice that he will never be able to provide the secret. In this case there is no reason for them to keep passing a condition that will never be fulfilled back and forth. Bob simply makes a new **Update Transaction**, without the **Smart Condition**.

Both sign

Update Transaction:

  Nonce: 13

  Net Transfer Amount: -34

  Hold Period: 8

Signature 1: Alice's signature on **Update Transaction**

Signature 2: Bob's signature on **Update Transaction**

3 Multihop payments

With the help of a hashlock condition, it is possible to trustlessly route payments across multiple hops. Let's say that Alice would like to transfer some funds to Charlie, but she does not have a channel open with him. If she has a channel with Bob, and Bob has a channel with Charlie, the funds can be transferred.

First, Alice sends a **Payment Secret** to Charlie:

Alice to Charlie

Payment Secret: "theSecret"

Then, Alice sends a hashlocked payment to Bob:

Alice to Bob

Update Transaction:

  Nonce: 13

  Net Transfer Amount: -2

  Hold Period: 8

  Conditions:

    1:

      Conditional Transfer Amount: -101

      Function(secret):

```
if sha3(secret) is equal to
"73B88F8C24EAA", return true;
else return false
```

Signature 1: Alice's signature on **Update Transaction**

Notice that Alice has sent Bob 101 coins instead of 100, as Bob charges her a 1% fee for routing payments.

Now, Bob sends the payment along to Charlie (Bob is **Party 1** and Charlie is **Party 2** in their channel):

Bob to Charlie

Update Transaction:

- **Nonce:** 42
- **Net Transfer Amount:** 56
- **Hold Period:** 10
- **Conditions:**
  - **1:**
    - **Conditional Transfer Amount:** 100
    - **Function(secret):**

      ```
      if sha3(secret) is equal to
      "73B88F8C24EAA", return true;
      else return false
      ```

Signature 1: Bob's signature on **Update Transaction**

To claim the payment, Charlie can post this **Update Transaction**, along with the **Payment Secret** that Alice sent him.

Charlie posts

Update Transaction:

- **Nonce:** 42
- **Net Transfer Amount:** 56
- **Hold Period:** 10
- **Conditions:**
  - **1:**
    - **Conditional Transfer Amount:** 100
    - **Function(secret):**

      ```
      if sha3(secret) is equal to
      "73B88F8C24EAA", return true;
      else return false
      ```

Signature 1: Bob's signature on **Update Transaction**

Signature 2: Charlie's signature on **Update Transaction**

Charlie also posts

Fulfillment:

  Condition: 1

  Argument: “theSecret”

Signature: Charlie’s signature on **Fulfillment**

Once Charlie has posted the **Update Transaction**, Bob is able to see the **Payment Secret** and unlock his hashlocked funds from Alice. In this way, a network of nodes are able to exchange payment trustlessly with one another. One interesting aspect of this system is that while Alice and Bob both need to have channels open with the same blockchain or bank, and Bob and Charlie need to have channels open with the same blockchain or bank, Alice and Charlie do not. As long as the banks involved banks are willing to hold money in escrow and honor **Update Transactions** for their customers, and the blockchains involved are able handle the smart contract logic to do the same, a payment network can be created that spans banks and blockchains.

4 Multihop payments across currencies

In the multihop payment example above, it is not necessary for Alice and Bob’s channel to be with the same bank or blockchain as Bob and Charlie’s channel. It’s actually not even necessary for the channels to be denominated using the same store of value.

If Alice wants to send Charlie some Euros, and Charlie and Bob have a Euro channel open, it can be done. Alice needs to know how many dollars she needs to send Bob to have him send Charlie the right number of Euros (this can also be calculated from Bob’s exchange rate and fee). Alice sends the hashlocked dollars to Bob, and Bob sends hashlocked Euros to Charlie. If Charlie is happy with the number of Euros he will receive (which may depend on whether he receives a market exchange rate and intermediaries charge a fair fee), he reveals the **Payment Secret** to Bob as usual.

This can also be used to connect two parties transacting in the same currency, across hops of another currency. Let’s say that Alice wants to send dollars to Doris, but her only connection to Doris is through Bart and Conrad, who have a channel open on the Dogecoin blockchain. If Alice knows Bart’s fee and exchange rate, and Conrad’s fee and exchange rate, she can calculate whether it would be worth it to send Doris payment across Bart and Conrad’s Dogecoin channel. This technique could be very powerful for providing payment connectivity between separate groups of people using non-crypto currency channels, as it will probably be a lot quicker to open a channel on a blockchain vs with a bank. Enterprising individuals can identify parts of the network lacking connectivity and supply it, earning transaction fees for their efforts.

5 Routing multihop payments

If you're going to have a multihop payment network, you need some way to route payments. How does Alice know that Bob is the best person to go through to reach Charlie? Perhaps Benjamin also has channels open with Alice and Charlie but he charges a lower fee. There needs to be some way to find the lowest-priced route to a payment's destination. This problem is very similar to the problem of routing packets on the internet, so we will look at some possible solutions from that domain.

There are two main categories of ad-hoc routing protocols— proactive and reactive. Proactive protocols work by exchanging messages to build up routing tables listing the next hop for each address on the network. When a node receives a packet, it is immediately able to forward it along to the best peer to get it closer to its destination. However, every node needs to have an entry in its routing tables for every other node. On a large network, this becomes infeasible.

In reactive protocols, nodes request a route from the network when they need to send packets to a new destination. This means that it is not necessary for every node to store information on every destination, and it is not necessary to update every node on the network when a connection changes. Of course, the downside is that the initial route discovery process adds some unavoidable latency when sending to new destinations.

For most payments, a few hundred milliseconds to establish a route is not a huge deal. Needing to store a routing table entry for every address in the network is far worse. For this reason we'll use a variation of Ad hoc On-Demand Distance Vector Routing (AODV) [8], a reactive routing protocol.

In AODV, when nodes need to send a packet to a destination they have never sent a packet to, they send out a **Route Request Message**, which is flooded through the network (with some optimizations). When the destination receives this message, it sends a **Route Reply Message**. Intermediary nodes along the path cache the next hops for the source and the destination, thereby storing only the routing information they are likely to need often.

5.1 Reactive Payment Routing

Since our nodes are presumed to already have connectivity, we can skip the **Route Request Message**. Our protocol has only one type of message, which we'll call the **Routing Message**. A node's neighbors are those nodes that it has payment channels open with.

When a node (which we'll refer to as the source) wishes to send a multihop payment, it first sends a **Payment Initialization** to the destination of the payment.

Source to destination

Payment Initialization:

Secret: Payment secret.

Amount: Amount of payment.

The destination then constructs a **Routing Message**. The routing message includes the hash of the payment secret, and the amount of the payment. It sends the **Routing Message** to all of its neighbors who have enough in their channels to cover the payment (if Dolores is trying to receive $100, she won't send the **Routing Message** to Clark, who only has $20 in his side of the channel).

Destination to neighbors

Routing Message:

Hash: Hash of payment secret.

Amount: Amount of payment.

When a node receives a **Routing Message**, the node makes a new **Routing Table Entry**.

Saved in routing table

Routing Table Entry:

Hash: Hash of payment secret.

Amount: Amount of payment.

Neighbor: The neighbor that the **Routing Message** came from.

The node also sends the **Routing Message** along to neighbors with enough to cover the payment, but not before adjusting the **Amount**. To adjust the **Amount**, the node adds the fee that it would like to recieve for routing the payment. Also, if the node is sending the **Routing Message** to a neighbor with whom it has a channel open in a different currency, the **Amount** is converted to that currency.

Routing Message:

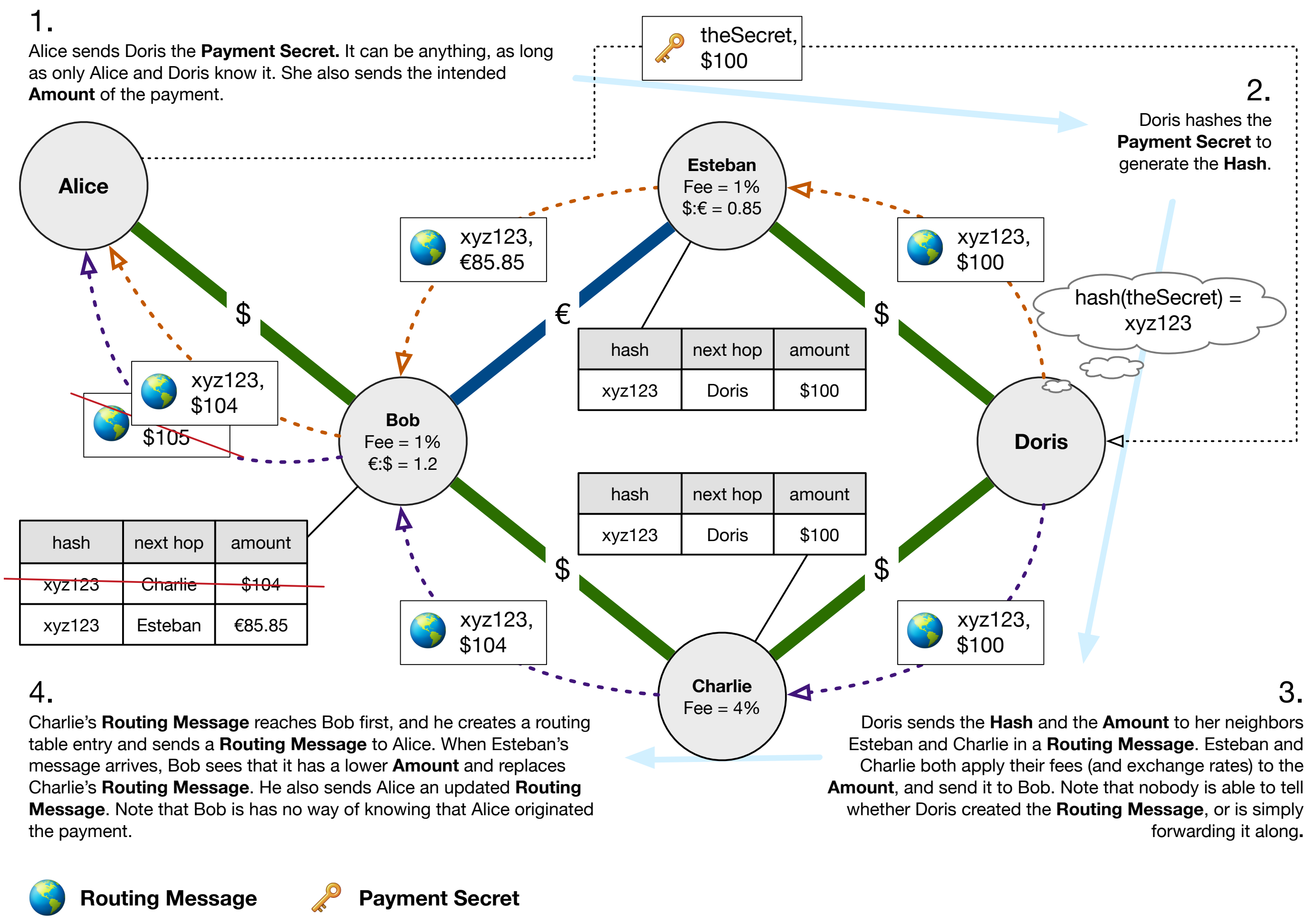
Hash: Hash of payment secret.

Amount: $(payment + fee) * exchange\_rate$

The **Routing Message** can be thought of as asking one implicit question: "How much would someone have to send you for you to send me **Amount**?" By adjusting the amount, a node is informing the network how much it charges to transfer money, and consequently, how good the route that it is on is. The **Routing Table Entry** makes sure the node routes the actual payment correctly, if it is on the winning route.

If a node receives a **Routing Message** with the same **Payment Secret** hash again, it will compare the **Amount** of the new **Routing Message** with the **Amount** that it has stored in its **Routing Table**. If the **Amount** of the new **Routing Message** is lower than what is in the **Routing Table**, it will update the **Routing Table** and send out a new **Routing Message**.

The **Routing Messages** propagate until they reach the source of the payment. At this point, the source can continue to wait, because it may receive another **Routing Message** with a lower **Amount**. If the source is satisfied with the **Amount**, it then uses the **Payment Secret** hash to make a hashlocked payment to the neighbor that sent it the routing message. This neighbor then checks their routing table and makes a payment to their corresponding neighbor. The hashlocked payments continue until they reach the destination, at which point it unlocks the payment with the secret, letting the rest of the chain unlock their payments as well (as explained in "Multiphop Channels" above).



1.
Alice sends Doris the **Payment Secret.** It can be anything, as long as only Alice and Doris know it. She also sends the intended **Amount** of the payment.

2.
Doris hashes the **Payment Secret** to generate the **Hash**.

3.
Doris sends the **Hash** and the **Amount** to her neighbors Esteban and Charlie in a **Routing Message**. Esteban and Charlie both apply their fees (and exchange rates) to the **Amount**, and send it to Bob. Note that nobody is able to tell whether Doris created the **Routing Message**, or is simply forwarding it along.

4.
Charlie's **Routing Message** reaches Bob first, and he creates a routing table entry and sends a **Routing Message** to Alice. When Esteban's message arrives, Bob sees that it has a lower **Amount** and replaces Charlie's **Routing Message**. He also sends Alice an updated **Routing Message**. Note that Bob is has no way of knowing that Alice originated the payment.

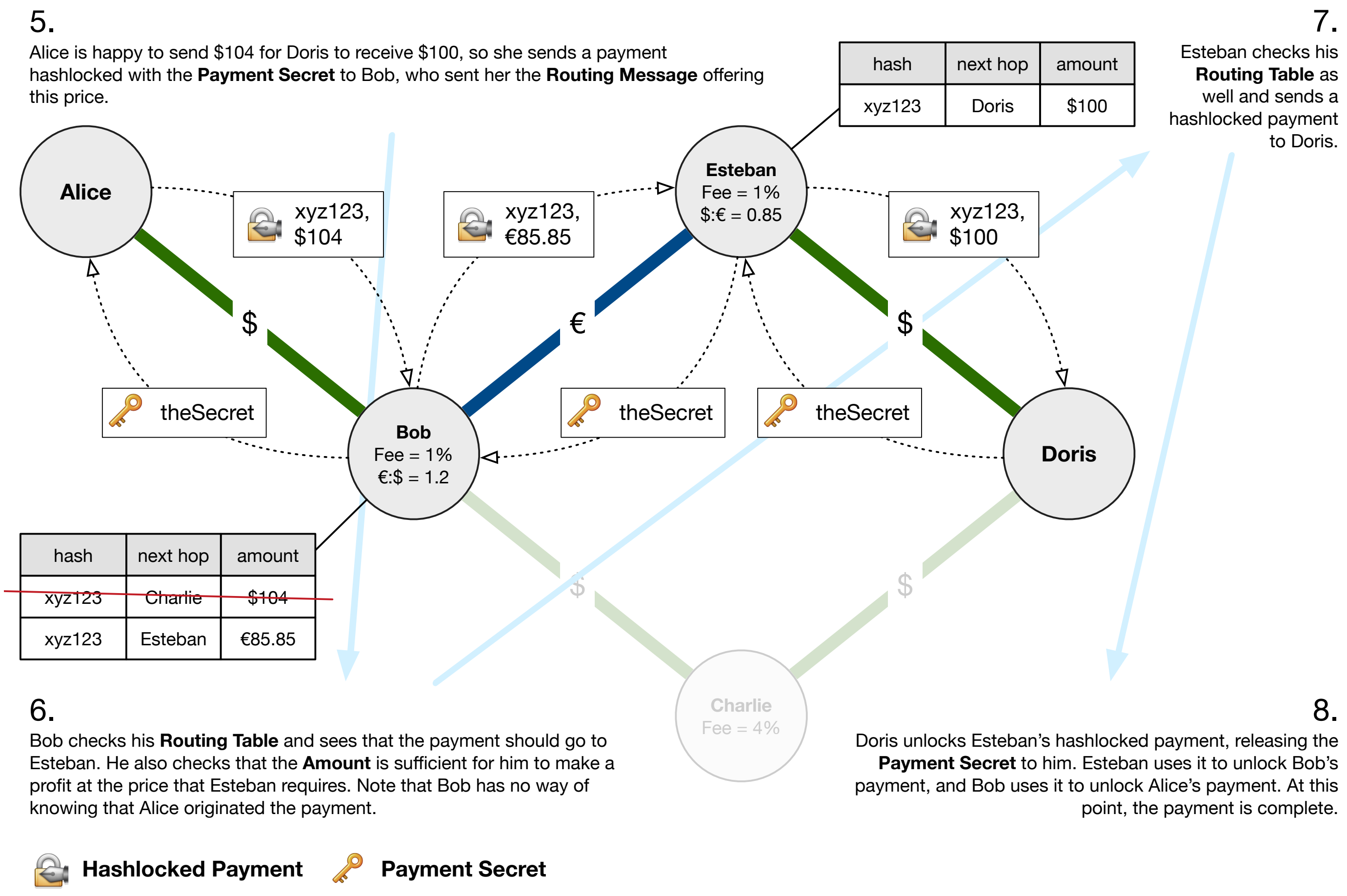
hash	next hop	amount
xyz123	Doris	$100

hash	next hop	amount
xyz123	Doris	$100

hash	next hop	amount
~~xyz123~~	~~Charlie~~	~~$104~~
xyz123	Esteban	€85.85

🌍 **Routing Message** 🔑 **Payment Secret**

Figure 1: Finding a payment route



5.
Alice is happy to send $104 for Doris to receive $100, so she sends a payment hashlocked with the **Payment Secret** to Bob, who sent her the **Routing Message** offering this price.

hash	next hop	amount
xyz123	Doris	$100

7.
Esteban checks his **Routing Table** as well and sends a hashlocked payment to Doris.

hash	next hop	amount
~~xyz123~~	~~Charlie~~	~~$104~~
xyz123	Esteban	€85.85

6.
Bob checks his **Routing Table** and sees that the payment should go to Esteban. He also checks that the **Amount** is sufficient for him to make a profit at the price that Esteban requires. Note that Bob has no way of knowing that Alice originated the payment.

8.
Doris unlocks Esteban's hashlocked payment, releasing the **Payment Secret** to him. Esteban uses it to unlock Bob's payment, and Bob uses it to unlock Alice's payment. At this point, the payment is complete.

Hashlocked Payment **Payment Secret**

Figure 2: Sending a hashlocked payment along a route

Glossary

Conditional Transfer Amount An amount included in **Smart Conditions**, which is added to the channel's **Net Transfer Amount** if a **Fulfillment** is posted which causes the **Smart Condition** to return true. 5–7, 17, 18

Fulfillment A piece of data to be evaluated by a **Smart Condition**. This can be posted at any time during the **Hold Period**, and only needs to be signed by one of the parties. 5–8, 11, 17, 18

Hashlock Condition A **Smart Condition** that hashes its argument, usually a **Payment Secret** and compares the hash to a prespecified string. If it matches, the **Smart Condition** returns true and adds its **Conditional Transfer Amount** to the channel's **Net Transfer Amount**. 17

Hold Period A time period included the **Update Transaction**. The bank or blockchain must wait this amount of time before transferring any money when the channel is closed. This provides a chance for one of the parties to counteract a cheating attempt where the other party posts an old **Update Transaction**. If a newer **Update Transaction** with a higher **Nonce** is posted before the **Hold Period** is over, it will override the older one. 3, 5–7, 17, 18

Net Transfer Amount An amount included in **Update Transactions** which specifies how much money to transfer from **Party 1** to **Party 2** when the channel closes. If it is negative, funds are transferred in the other direction. 4–8, 17, 18

Nonce An integer included the **Update Transaction** which is incremented with each new **Update Transaction**. This is used by the bank or the blockchain to ascertain the ordering of **Update Transactions**. An **Update Transaction** with a higher **Nonce** will always override one with a lower **Nonce**. 5, 8, 17

Opening Transaction A message signed by both parties that creates a channel. This is posted to the bank or blockchain and serves to identify the parties and place funds in escrow. 3, 4

Payment Secret A secret shared between the source and destination of a multihop payment. It is hashed and used to create **Hashlock Conditions** between all the intermediary nodes involved in the multihop payment. The destination reveals it to the last intermediary node to claim the payment, and the last intermediary nodes reveals it to the second-to-last and so on back to the source. 7–11, 14, 17

Routing Message A message propagated between nodes as part of the routing protocol. It contains the hash of the **Payment Secret** and an **Amount**, which is added to by each node that propagates it. 12–14, 18

Routing Table A table maintained by each node that records **Routing Messages** received and forwarded by that node. It is used to route the payment corresponding to a given **Routing Messages**. 14

Smart Condition A piece of Turing-complete code included in the **Update Transaction** that is evaluated by the bank or blockchain during the **Hold Period**. It can return either true or false when supplied with a **Fulfillment**. It has an associated **Conditional Transfer Amount**, which is added to the channel's **Net Transfer Amount** if the **Smart Condition** returns true. 3, 5–8, 17, 18

Update Transaction A message signed by both parties that updates the state of a channel. This is posted to the bank or blockchain to close the channel, however an infinite number of them can be exchanged between the two parties before then. 3–11, 17, 18

Acknowledgements

Zackary Hess, for coming up with much of the definition of the **Basic Channel** as part of Flying Fox[11], a channel-based cryptocurrency and prediction market. Jae Kwon, for years of advice and guidance on the theory, implementation, and philosophy of cryptocurrency. Anke Tremback, for editing the first complete draft of this paper. Alice Townes, for editing and feedback from a legal and finance perspective.

References

[1] *A Protocol for Interledger Payments*
Stephan Thomas, Evan Schwartz
`https://interledger.org/interledger.pdf`
2015

[2] *Micropayment Channel*
Bitcoin Wiki Contributors
`https://bitcoin.org/en/developer-guide#micropayment-channel`
2014

[3] *[ANNOUNCE] Micro-payment channels implementation now in bitcoinj*
Mike Hearn
`https://bitcointalk.org/index.php?topic=244656.0`
2013

[4] *Decentralized networks for instant, off-chain payments*
Alex Akselrod
`https://en.bitcoin.it/wiki/User:Aakselrod/Draft`
2013

[5] *Amiko Pay*
C. J. Plooy

http://cornwarecjp.github.io/amiko-pay/doc/amiko_draft_2.pdf
2013

[6] *A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels*
Christian Decker, Roger Wattenhofer
http://www.tik.ee.ethz.ch/file/716b955c130e6c703fac336ea17b1670/duplex-micropayment-channels.pdf
2015

[7] *The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments*
Joseph Poon, Thaddeus Dryja
https://lightning.network/lightning-network-paper.pdf
2015

[8] *Ad-hoc On-Demand Distance Vector Routing*
Charles E. Perkins, Elizabeth M. Royer
https://www.cs.cornell.edu/people/egs/615/aodv.pdf

[9] *A Next-Generation Smart Contract and Decentralized Application Platform*
Vitalik Buterin
https://github.com/ethereum/wiki/wiki/White-Paper
2014

[10] *Tendermint: Consensus without Mining*
Jae Kwon
http://tendermint.com/docs/tendermint.pdf
2015

[11] *Flying Fox*
Zackary Hess
https://github.com/BumblebeeBat/FlyingFox
2015