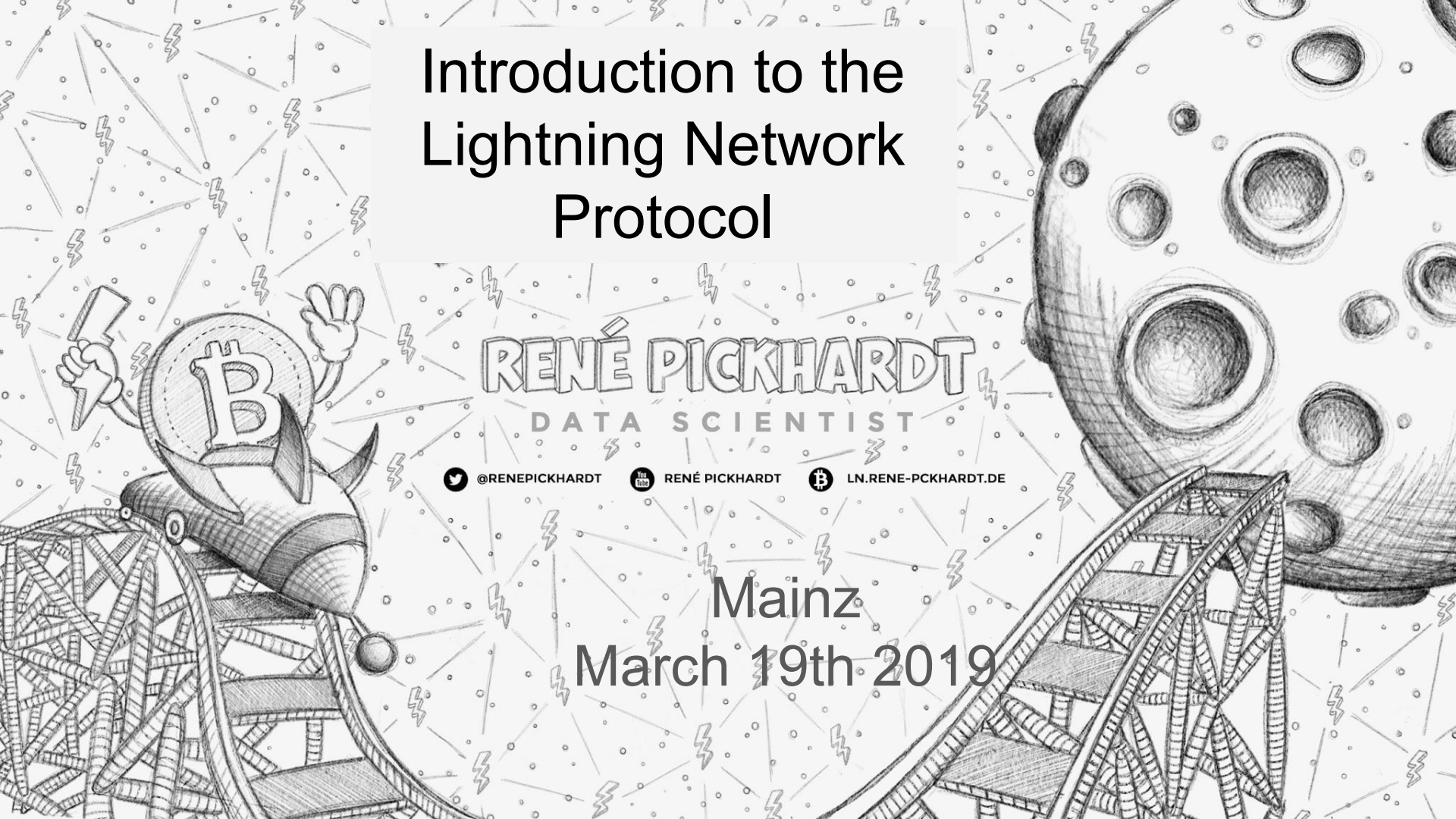
# Introduction to the Lightning Network Protocol



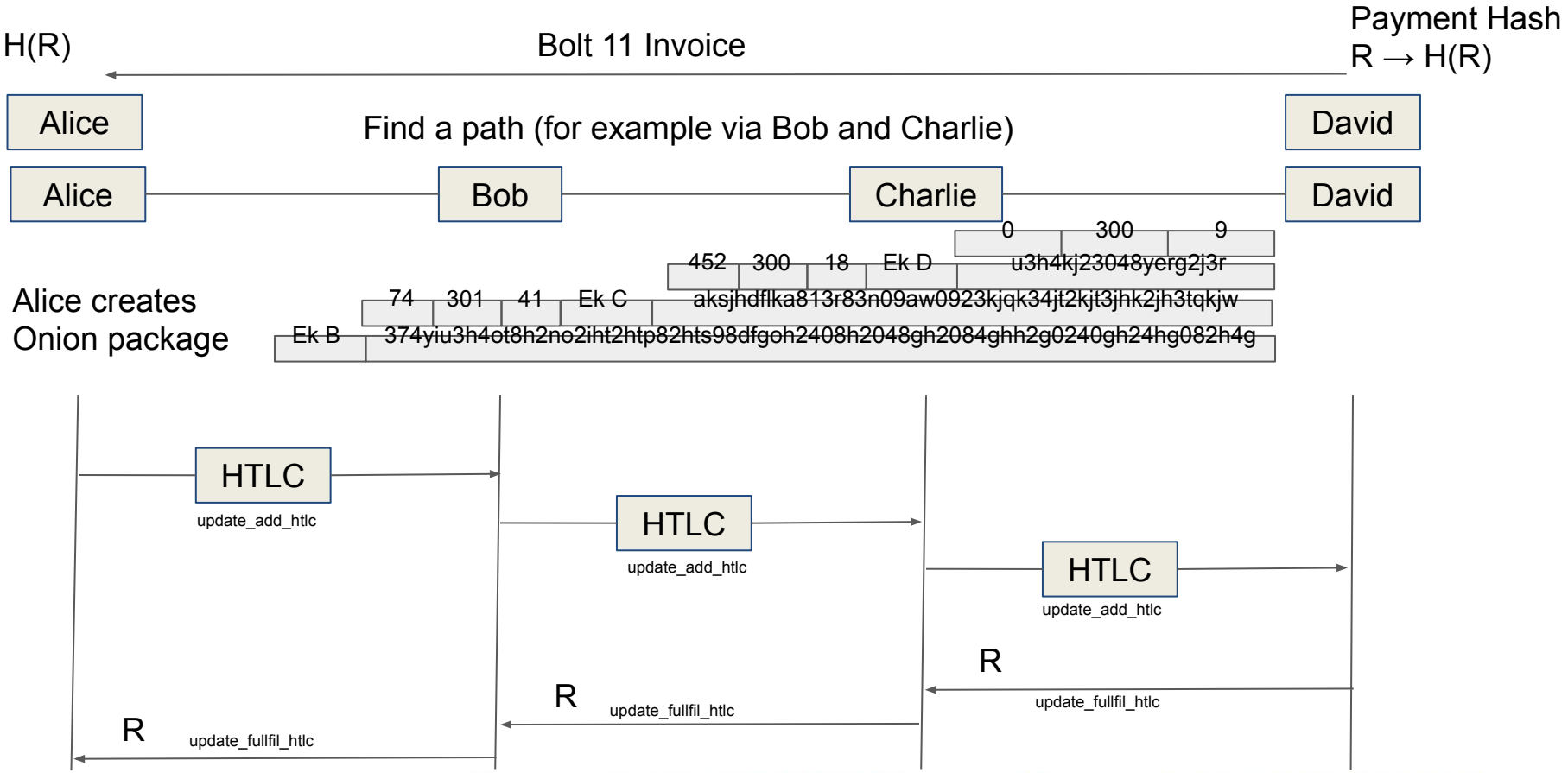

Mainz

March 19th 2019

# Disclaimer

- Please note the copyright notice at the end of the slide deck
- The pdf version - which you probably read has some weird formatting problems.
  - https://docs.google.com/presentation/d/1-eyceLlSmcLpbPJLzj6_CnVYQdo1AUP3y5XD716U-Lg has the source files without formatting problems and with animations
- This slidedeck is crowd funded (see last slide) if you wish to learn more or contribute
- I do not take any guarantee that the information in this slide deck are 100% correct.
  - Sometimes I made simplifications: which I point out
  - Also I could just have misunderstood something or just made a mistake

# After this talk you will hopefully understand this slide!



# Outline: Understand the Lightning Network

1. Construction of Payment channels (RSMCs)
2. The payment process (based on invoices)
3. Trustless routing of payments via a network of channels (HTLCs)
4. Source based onion routing (SPHINX mix Format)
5. Transport Layer (Noise XK - Noise Protocol Framework)
6. Messaging / peer protocol
7. Gossip Protocol
8. Advanced Topics
    - Future enhancements to the BOLTs
    - Privacy considerations
    - Security breaches
        - By (poor) Implementations
        - By Services

# Bitcoin P2P e-cash paper

**Satoshi Nakamoto** satoshi at vistomail.com
*Fri Oct 31 14:10:00 EDT 2008*

- Previous message: Fw: SHA-3 lounge
- **Messages sorted by:** [ date ] [ thread ] [ subject ] [ author ]

---

I've been working on a new electronic cash system that's fully peer-to-peer, with no trusted third party.

The paper is available at:
http://www.bitcoin.org/bitcoin.pdf

The main properties:
 Double-spending is prevented with a peer-to-peer network.
 No mint or other trusted parties.
 Participants can be anonymous.
 New coins are made from Hashcash style proof-of-work.
 The proof-of-work for new coin generation also powers the
    network to prevent double-spending.

Bitcoin: A Peer-to-Peer Electronic Cash System

**Bitcoin: A Peer-to-Peer Electronic Cash System**

Satoshi Nakamoto
satoshin@gmx.com
www.bitcoin.org

**Abstract.** A purely peer-to-peer version of electronic cash would allow online payments to be sent directly from one party to another without going through a financial institution. Digital signatures provide part of the solution, but the main benefits are lost if a trusted third party is still required to prevent double-spending. We propose a solution to the double-spending problem using a peer-to-peer network. The network timestamps transactions by hashing them into an ongoing chain of hash-based proof-of-work, forming a record that cannot be changed without redoing the proof-of-work. The longest chain not only serves as proof of the sequence of events witnessed, but proof that it came from the largest pool of CPU power. As long as a majority of CPU power is controlled by nodes that are not cooperating to attack the network, they'll generate the longest chain and outpace attackers. The network itself requires minimal structure. Messages are broadcast on a best effort basis, and nodes can leave and rejoin the network at will, accepting the longest proof-of-work chain as proof of what happened while they were gone.

## 1. Introduction

Commerce on the Internet has come to rely almost exclusively on financial institutions serving as trusted third parties to process electronic payments. While the system works well enough for most transactions, it still suffers from the inherent weaknesses of the trust based model. Completely non-reversible transactions are not really possible, since financial institutions cannot avoid mediating disputes. The cost of mediation increases transaction costs, limiting the minimum practical transaction size and cutting off the possibility for small casual transactions, and there is a broader cost in the loss of ability to make non-reversible payments for non-reversible services. With the possibility of reversal, the need for trust spreads. Merchants must be wary of their customers, hassling them for more information than they would otherwise need. A certain percentage of fraud is accepted as unavoidable. These costs and payment uncertainties can be avoided in person by using physical currency, but no mechanism exists to make payments over a communications channel without a trusted party.

# Bitcoin P2P e-cash paper

**James A. Donald** jamesd at echeque.com
*Sun Nov 2 18:46:23 EST 2008*

- Previous message: Who cares about side-channel attacks?
- Next message: Secrets and cell phones.
- **Messages sorted by:** [ date ] [ thread ] [ subject ] [ author ]

---

Satoshi Nakamoto wrote:
> *I've been working on a new electronic cash system that's fully*
> *peer-to-peer, with no trusted third party.*
>
> *The paper is available at:*
> *http://www.bitcoin.org/bitcoin.pdf*

We very, very much need such a system, but the way I understand your proposal, it does not seem to scale to the required size.

For transferable proof of work tokens to have value, they must have monetary value. To have monetary value, they must be transferred within a very large network - for example a file trading network akin to bittorrent.

To detect and reject a double spending event in a timely manner, one must have most past transactions of the coins in the transaction, which, naively implemented, requires each peer to have most past transactions, or most past transactions that occurred recently. If hundreds of millions of people are doing transactions, that is a lot of bandwidth - each must know all, or a substantial part thereof.

## Bitcoin: A Peer-to-Peer Electronic Cash System

Satoshi Nakamoto
satoshin@gmx.com
www.bitcoin.org

**Abstract.** A purely peer-to-peer version of electronic cash would allow online payments to be sent directly from one party to another without going through a financial institution. Digital signatures provide part of the solution, but the main benefits are lost if a trusted third party is still required to prevent double-spending. We propose a solution to the double-spending problem using a peer-to-peer network. The network timestamps transactions by hashing them into an ongoing chain of hash-based proof-of-work, forming a record that cannot be changed without redoing the proof-of-work. The longest chain not only serves as proof of the sequence of events witnessed, but proof that it came from the largest pool of CPU power. As long as a majority of CPU power is controlled by nodes that are not cooperating to attack the network, they'll generate the longest chain and outpace attackers. The network itself requires minimal structure. Messages are broadcast on a best effort basis, and nodes can leave and rejoin the network at will, accepting the longest proof-of-work chain as proof of what happened while they were gone.

### 1. Introduction

Commerce on the Internet has come to rely almost exclusively on financial institutions serving as trusted third parties to process electronic payments. While the system works well enough for most transactions, it still suffers from the inherent weaknesses of the trust based model. Completely non-reversible transactions are not really possible, since financial institutions cannot avoid mediating disputes. The cost of mediation increases transaction costs, limiting the minimum practical transaction size and cutting off the possibility for small casual transactions, and there is a broader cost in the loss of ability to make non-reversible payments for non-reversible services. With the possibility of reversal, the need for trust spreads. Merchants must be wary of their customers, hassling them for more information than they would otherwise need. A certain percentage of fraud is accepted as unavoidable. These costs and payment uncertainties can be avoided in person by using physical currency, but no mechanism exists to make payments over a communications channel without a trusted party.

# Why Lightning network?

- Transform Bitcoin to an electronic cash system
  - Allow arbitrary amount of transactions
  - Get currency / cash property instead of store of value
- Make payments more anonymous
  - Don't store all payments on chain
- Allow for instant peer to peer payments
  - No need to wait for block confirmations
- In other words to scale Bitcoin
- And of course because…
  - we can … (:

# 3 Building blocks of the Lightning Network

1. Duplex Micropayment Channels
   - Revocable Sequence Maturity Contracts (RSMCs)
   - Penalty based invalidation mechanism
2. Value transfer in a network of payment channels
   - Comparing the payment process in Bitcoin with the Lightning Network
   - Hashed Time Lock Contracts (HTLCs)
   - Onion Routing based on the SPHINX Mix Format
3. Communication protocol
   - Transport layer with strong encryption
   - The peer protocol
   - Gossip protocol to create a P2P network

# Construction of Payment channels (RSMCs)

(Chapter 1)

# Purpose of a trustless bidirectional payment channel

- Sending payments between two partners
  - Instantly
    - Like updating a balance sheet
    - Only bound by network traffic over the internet
  - No direct involvement of the blockchain (as long as everyone plays by the rules)
- No need for any partner to trust the other side
  - The bitcoin network (and the blockchain) needs to be trusted
  - The bitcoin network will settle conflicts if they arise
- No fees or overhead when sending payments within the channel
- Payments can go in either direction back and forth
- Scale usage of Bitcoin
  - transfer of value can be achieved without hitting the blockchain

→ How can this technically be possible?

# A comparison with HTTP

- HTTP is a Request Response Protocol by design
  - You (the client / Browser) can make a request
  - The server gives you a response
- According to the protocol a server cannot initiate communication with a client

- How can a website (e.g. Twitter) give us a notification that some new data is there?
  - By abusing the protocol
  - aka: Server push, long polling,
  - Took us about 10 years to figure that out

# How does a HTTP server push work?

- You load a web page
- Javascript (AJAX) initiates an HTTP request
- Server looks if he has some information for the client
- If yes → response
- If no → defer answering to that request to some time in the future
  - If new information for the client is available
  - If client opens a new page and can't receive that response anymore
  - Just as a heartbeat mechanism to ask the client to make a new request
- From an end user perspective the server has initiated a conversation
  - The end user is usually not aware of the fact that an AJAX Request is outstanding

More details at: https://en.wikipedia.org/wiki/Push_technology

# Payment channels are constructed by deferring the publication or broadcast of some TXs to the future

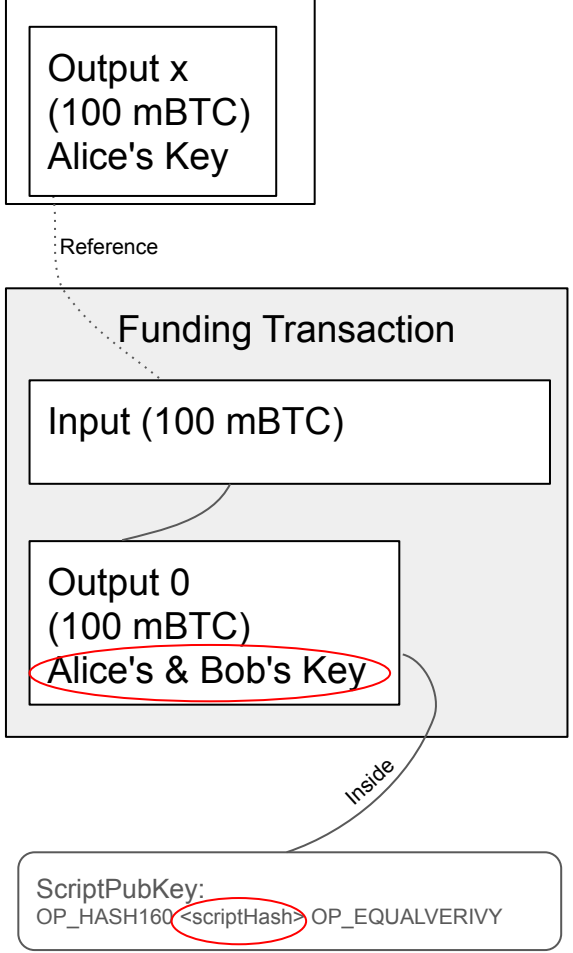
- Bitcoin is not a request response protocol
  - Rather a broadcast / gossip protocol
- What exactly is being deferred to the future?
  - Transactions spending outputs are held back from being broadcasted
  - Later Transactions are purposely tried to be double spent
    - Obviously only one of the transactions can be actually spent
    - Lightning is about making sure it is the correct one
  - We look at this in the next couple slides
- Payment channels are to Bitcoin what HTTP Server Push is to HTTP
  - Just a non obvious use of the Bitcoin protocol
  - It took us a little while to figure this feature of Bitcoin out (as for HTTP)

# A payment channel is a 2-2 multisig Address together with some (smart?) contract

- A 2-2 multisig Address means that the output script of a transaction requires 2 keys in order to be able to spend the transaction
- It is a form of Pay to Script Hash
  - scriptPubKey (OUTPUT): OP_HASH160 <scriptHash> OP_EQUAL
  - scriptSig (INPUT): ...<signatures>... <serialized script>
- Case of a m-of-n multi-signature
  - The OUTPUT script (scriptPubKey) as above
    - OP_HASH160 <scriptHash> OP_EQUAL
  - The INPUT Script to spend the scriptPubKey looks like this:
    - scriptSig: 0 <sig1>...<serialized script>
    - Script: OP_m <pubKey1> … OP_n OP_CHECKMULTISIG

# The secret smart contract (RSMC)

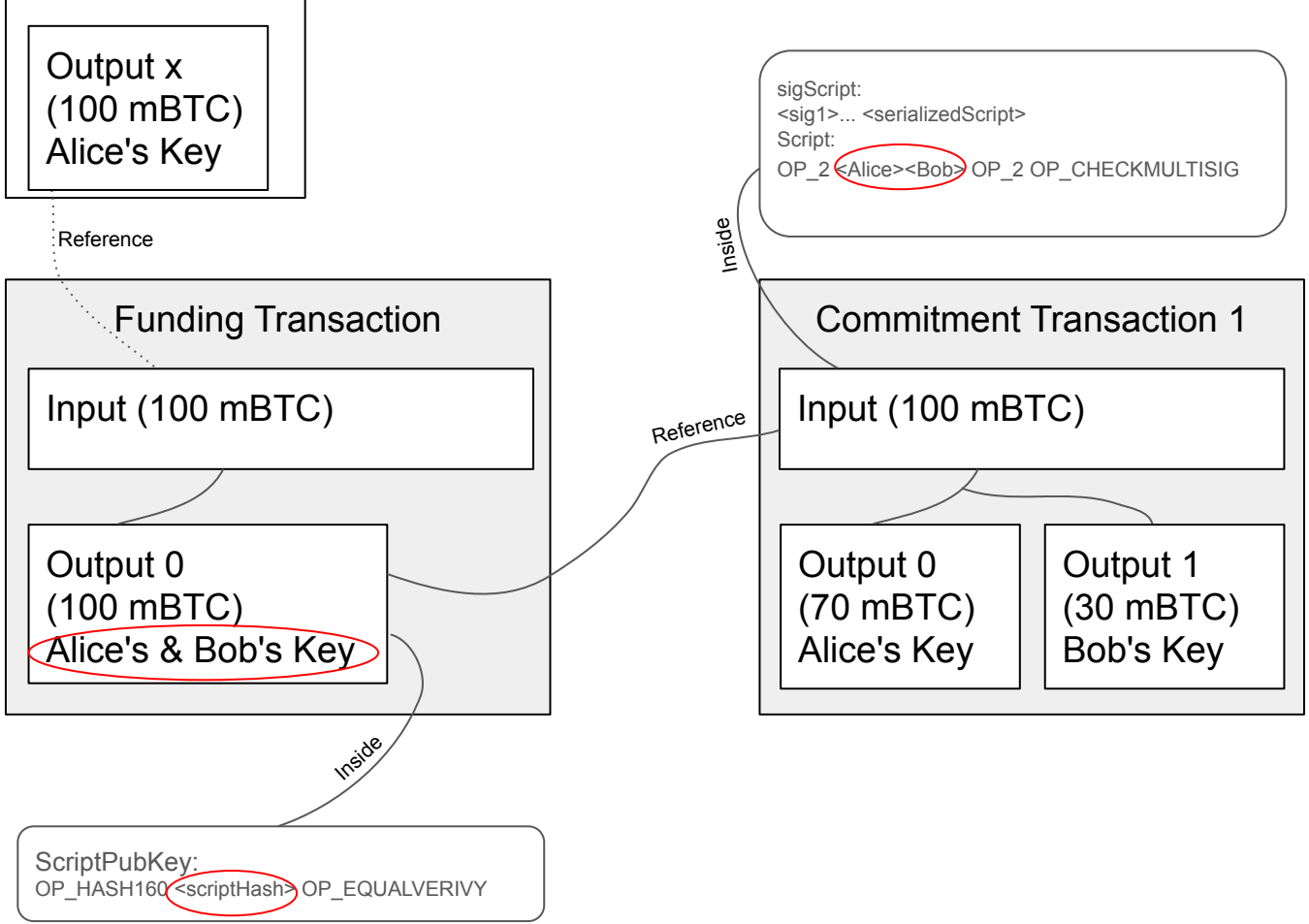
- The 2-2 multisig wallet has some funds
  - Seen on chain
- Who owns these funds?
  - Channel partners do not share the 2 private keys
  - Depends on how the funds are being spent
- The Transaction spending from the 2-2 Wallet
  - Will be kept secret (if possible)
  - Will encode the balance
  - Is called a commitment transaction (has to be signed BEFORE publishing funding tx)
- Technical details
  - Will be heavily attempted to be double spent (which by the bitcoin protocol is impossible)
  - Secures funds with timelocks
  - Allows for conditional hashed time locked outputs (used in routing)
- Revocable sequence maturity contract



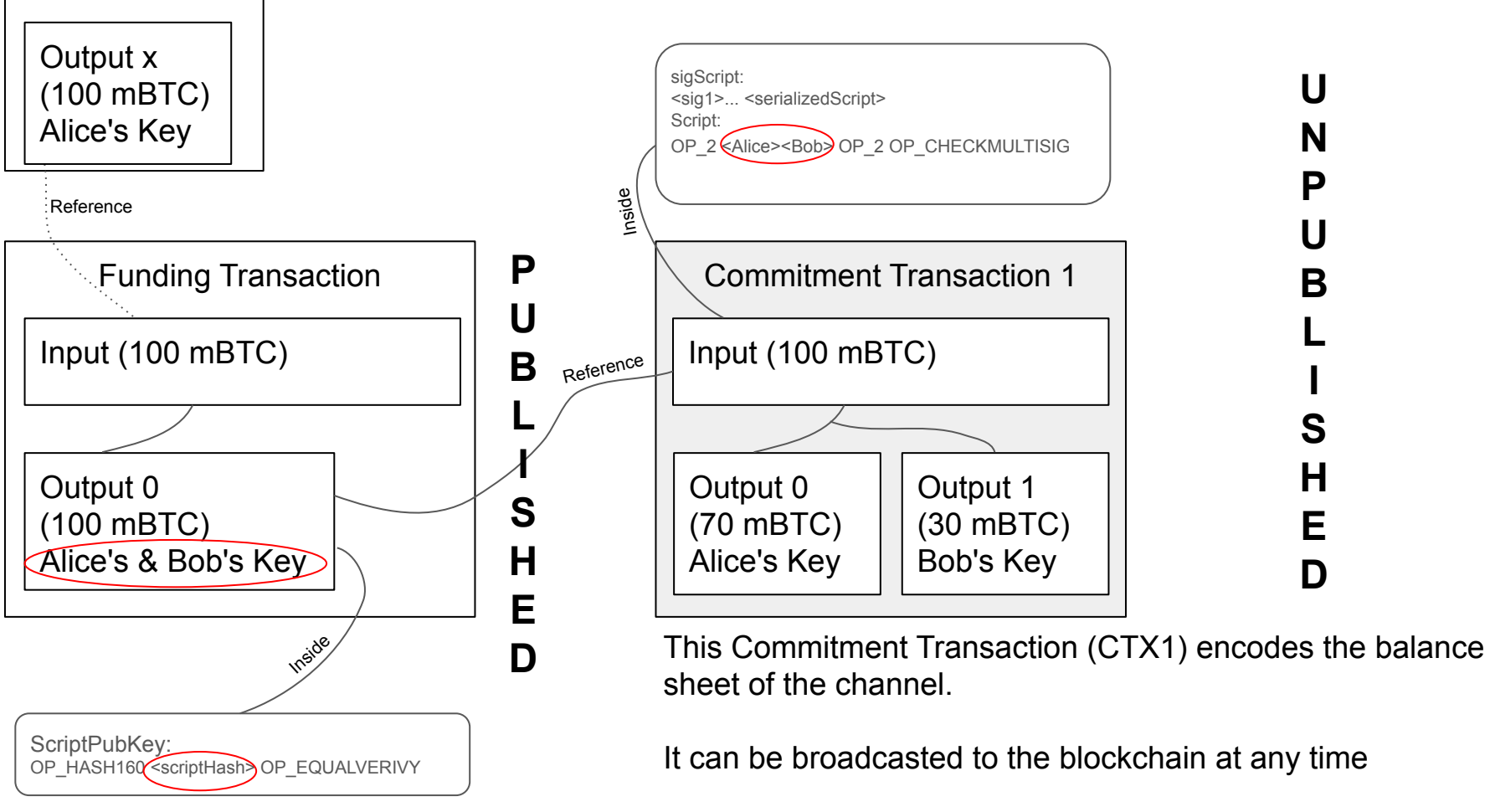

**UNPUBLISHED**

**Legend for colors:**

- Broadcasted and mined
- Not boradcasted
- Should be broadcasted
- Should not be broadcasted
- Cannot by minded
  (Consumed outputs are already spent)




UNPUBLISHED

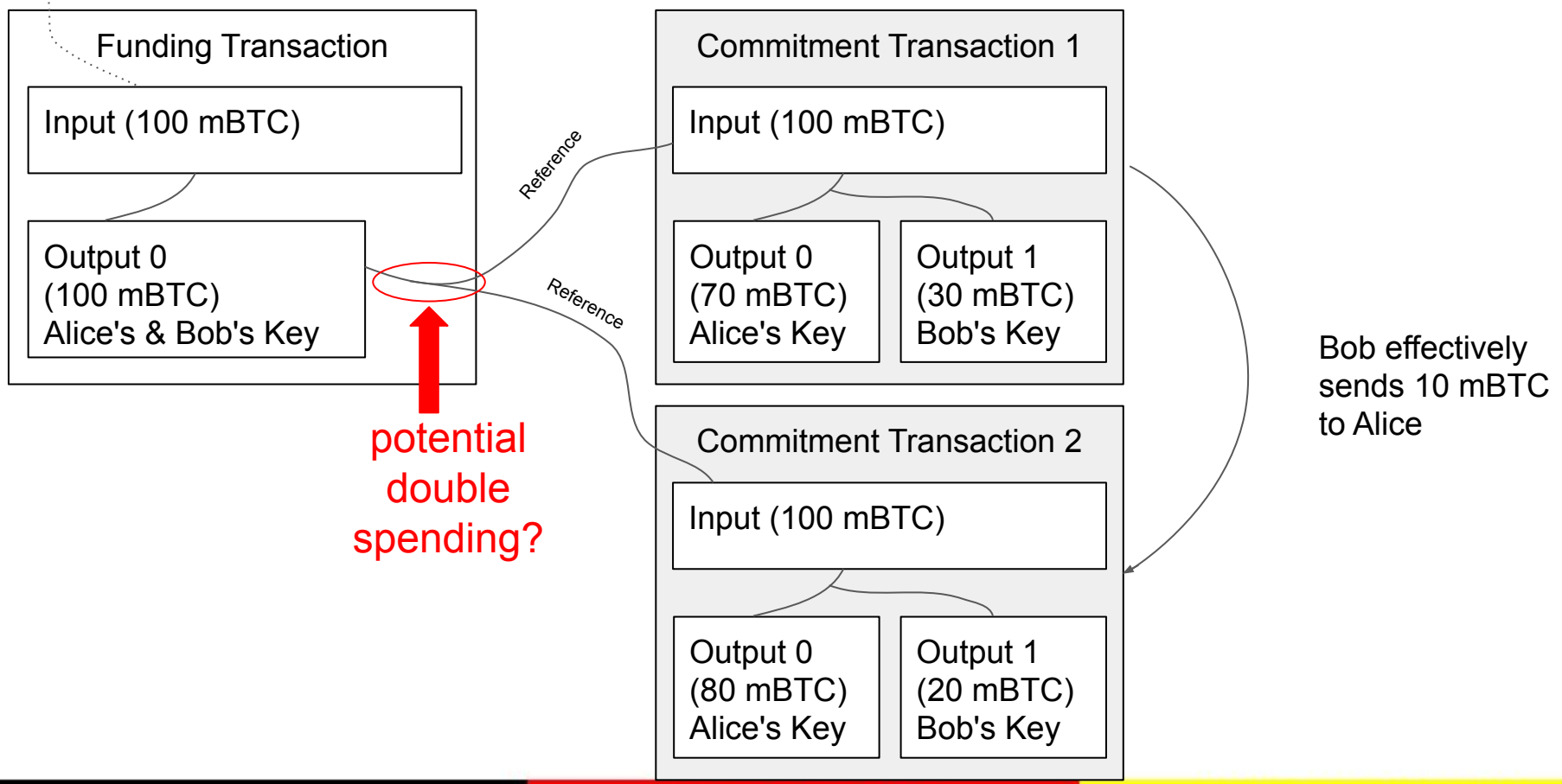



This Commitment Transaction (CTX1) encodes the balance sheet of the channel.

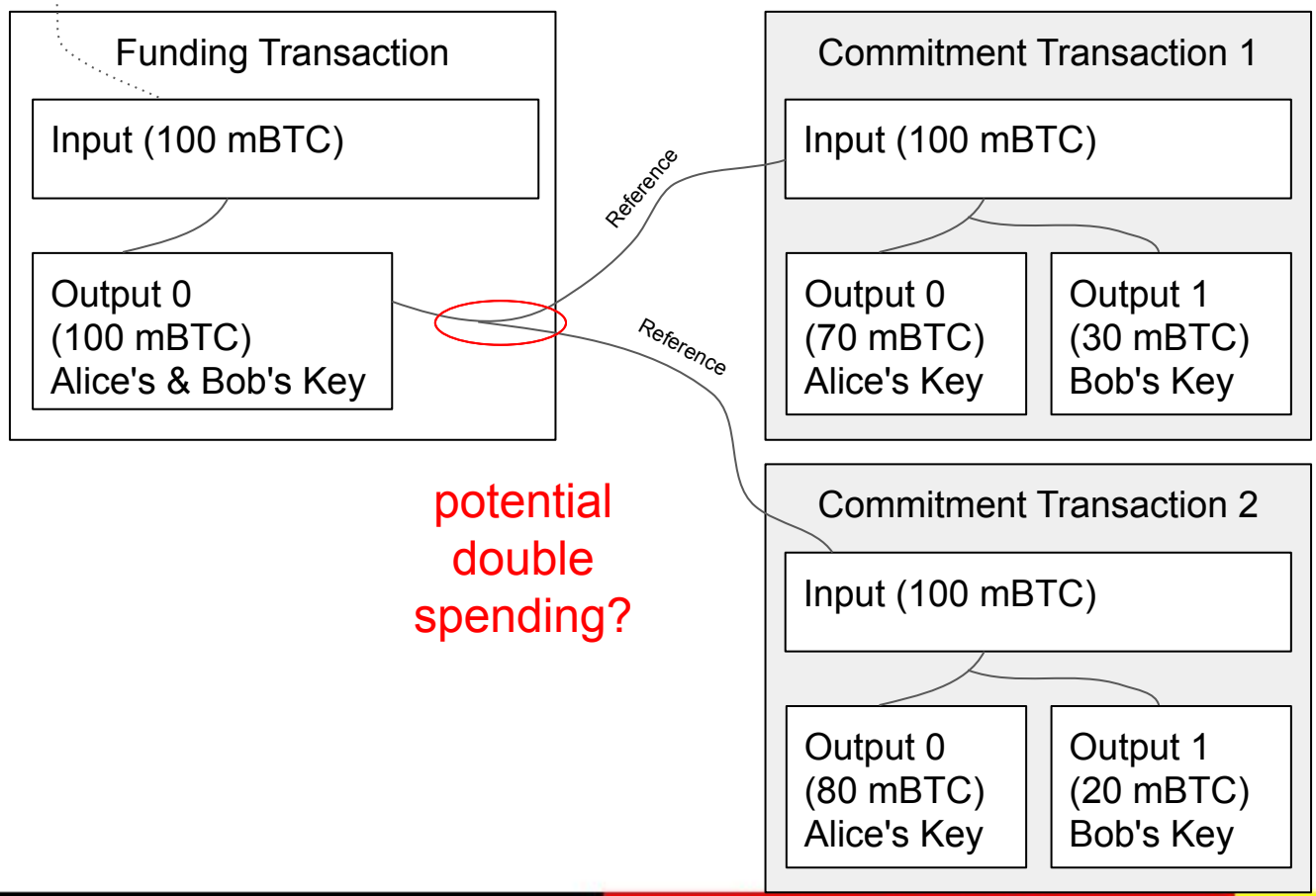
It can be broadcasted to the blockchain at any time

Funding Tx must use segWit outputs to prevent malleability

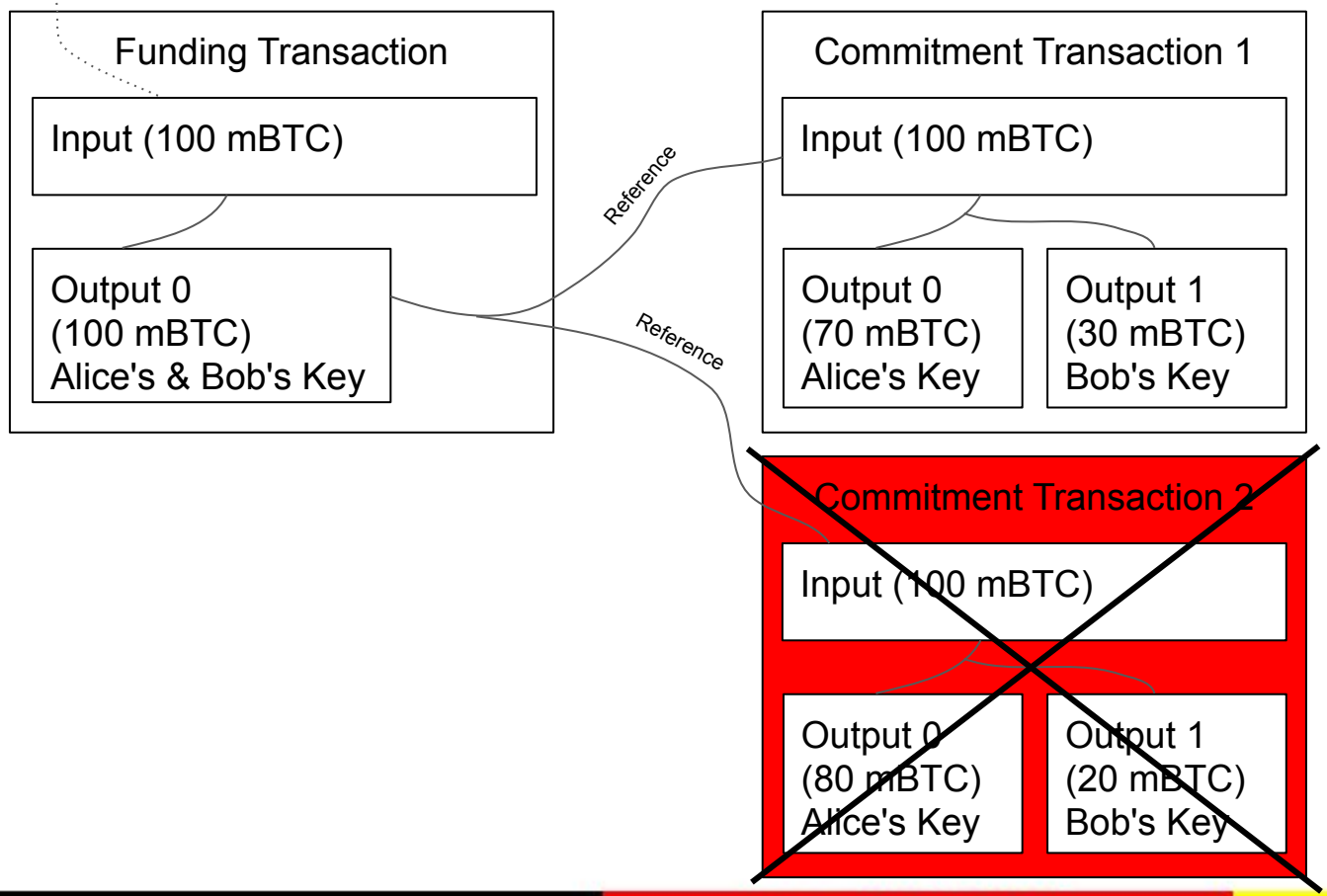
# With CTX2 Bob updates the channel Balance



# Anybody seeing a problem with this?

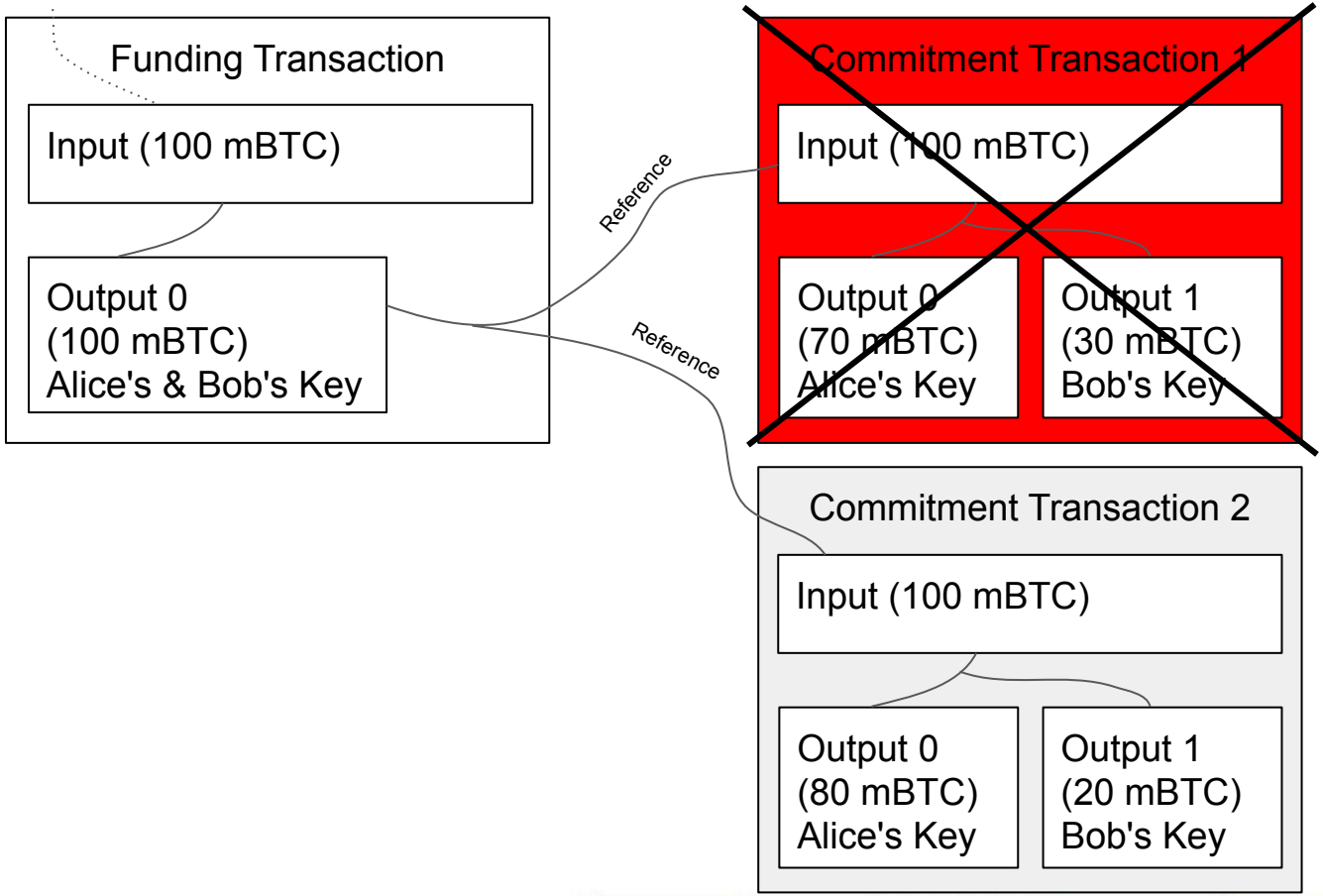


# Bob broadcasts CTX1 which is successfully mined




Bob just effectively stole 10 mBTC from Alice

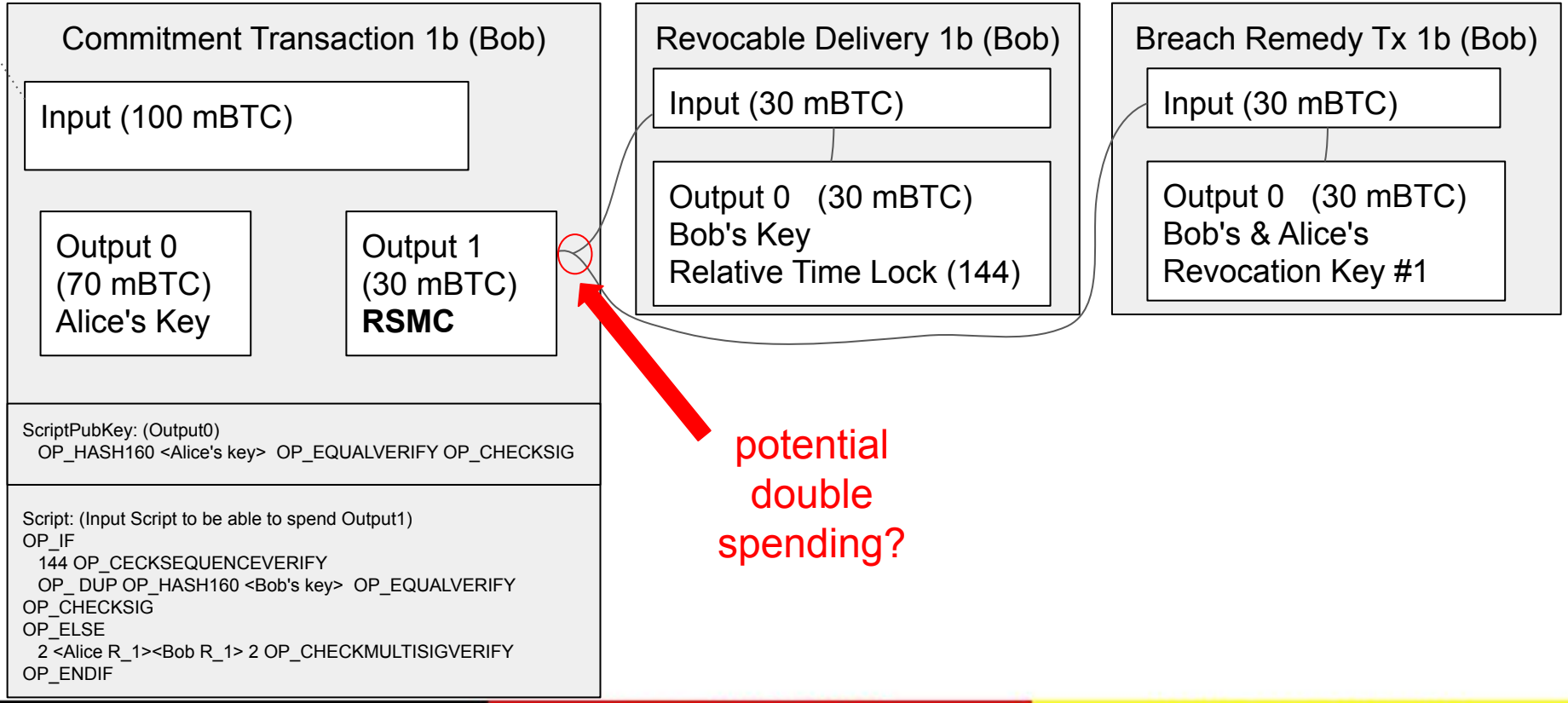
# Goal: Make old CTXs somehow unpublishable




- Modify the output of the CTX
- Fraudulent publishing will punish the publisher
- For example by giving the silent party the opportunity to claim all outputs of the CTX

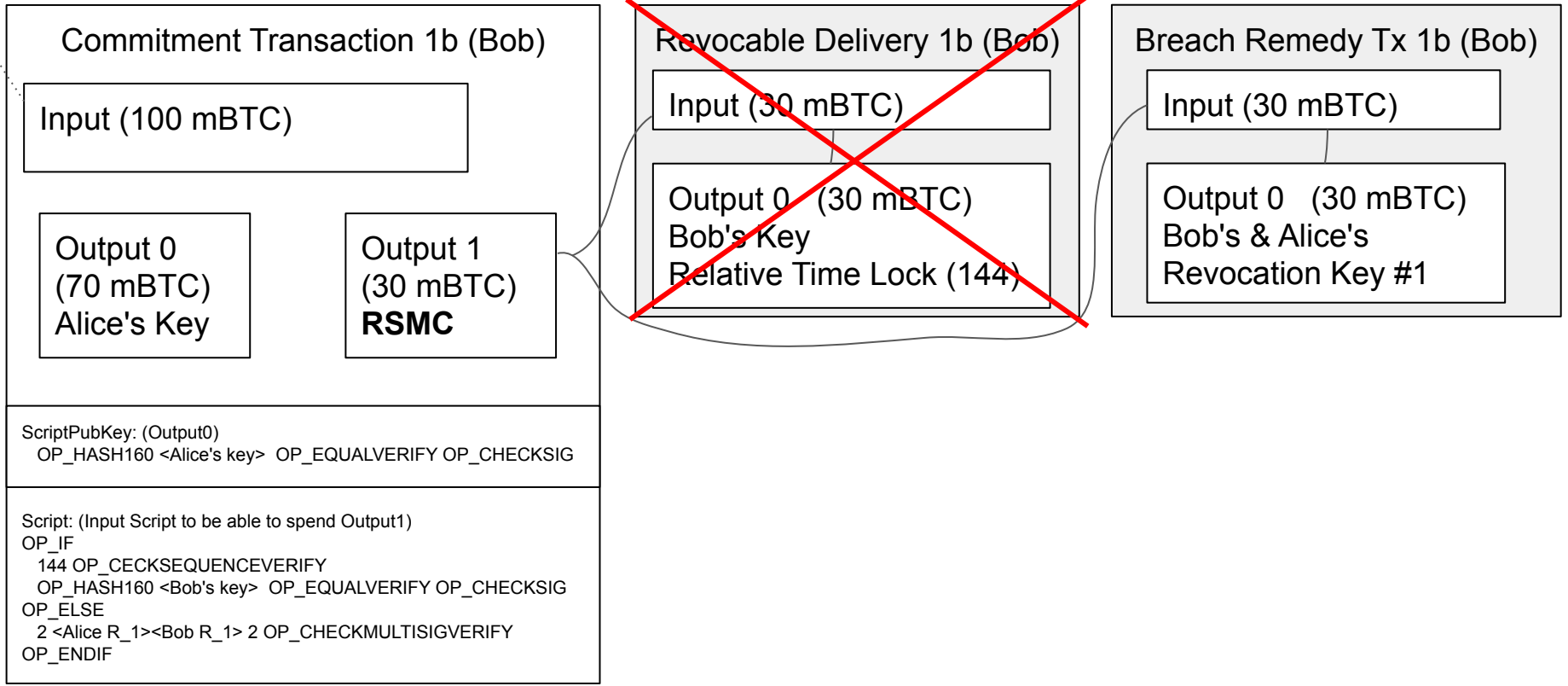
# We modify the output scripts of CTXs that it can be spent by either one of the two Transactions

Reference



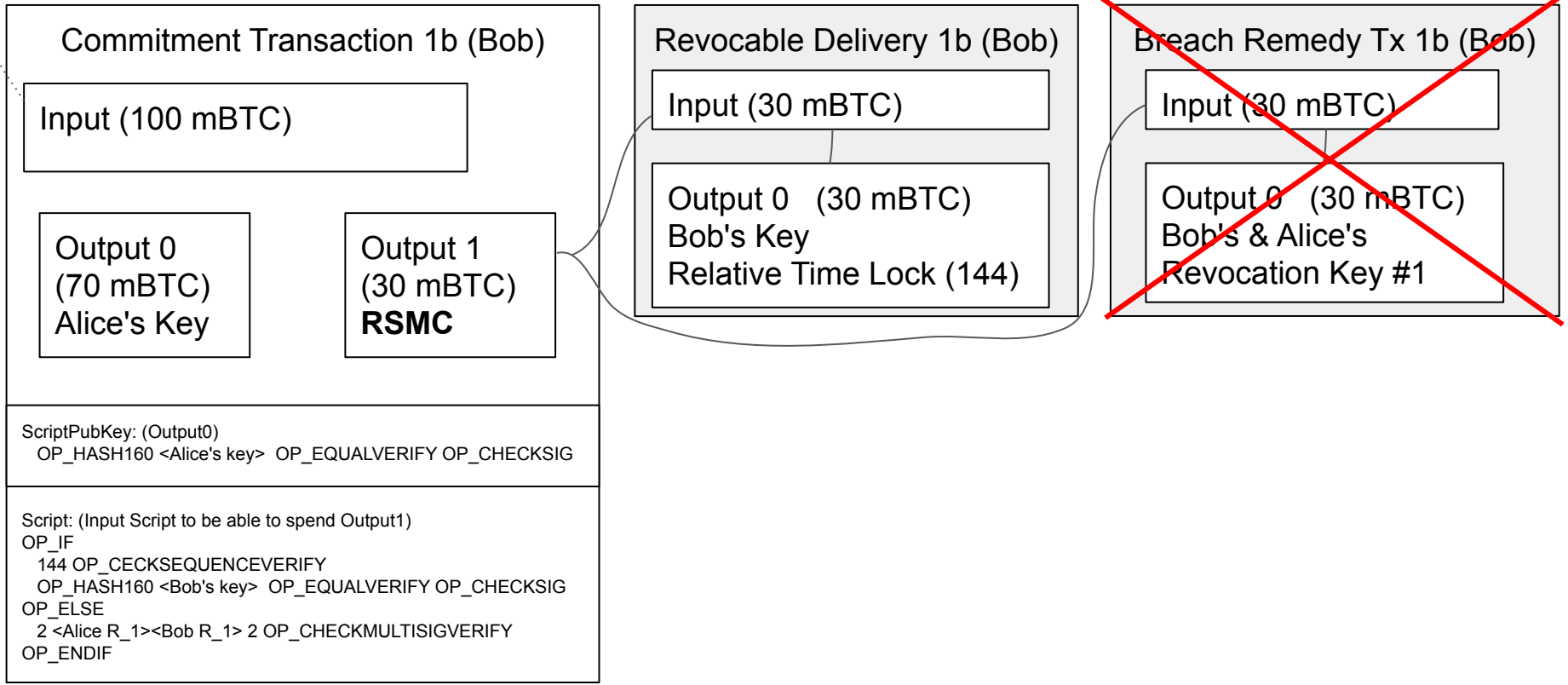
# Within 144 Blocks after CTX1b is mined only Alice can spend Output1 (if she has Bob's Revocation Key)

Reference

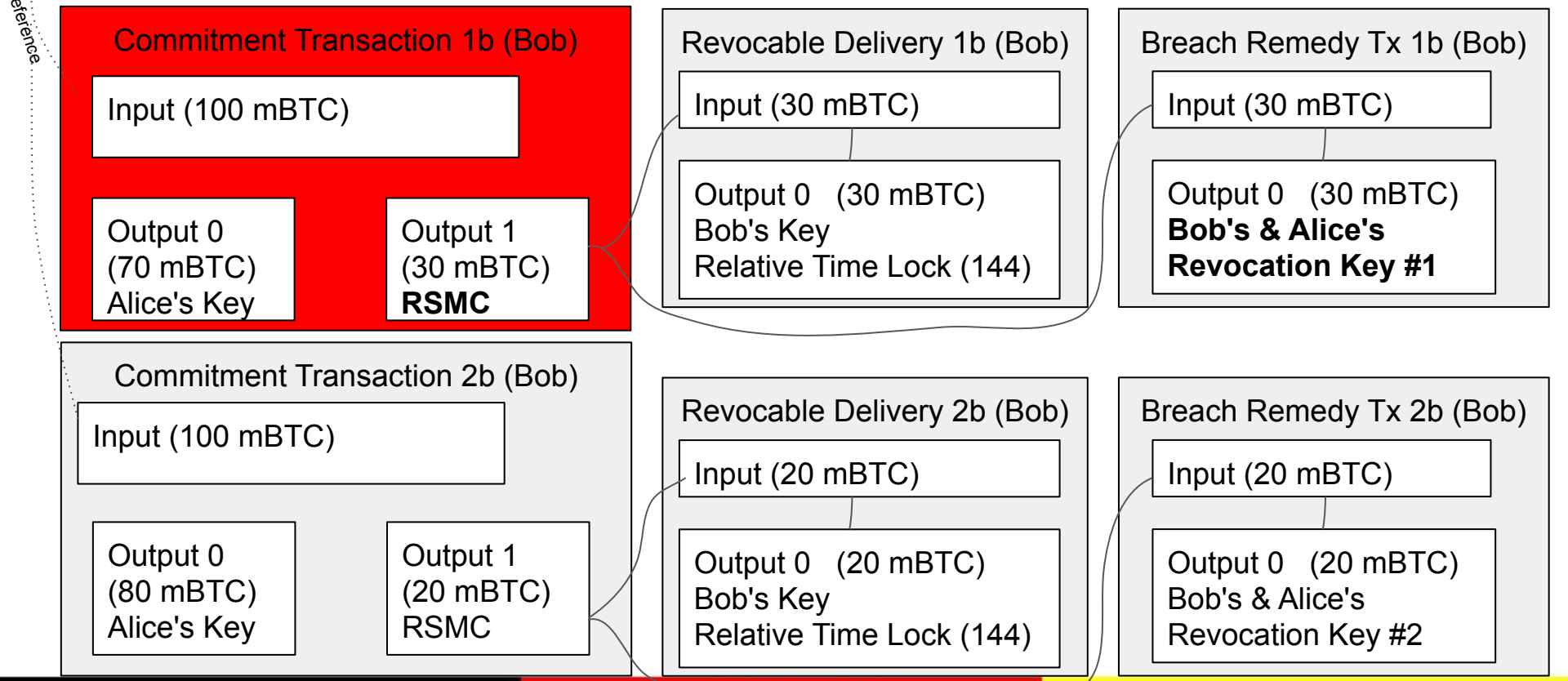


# After 144 Blocks Bob can redeem his 10 mBTC & Alice can't spent BRTX1b without Bob's Key

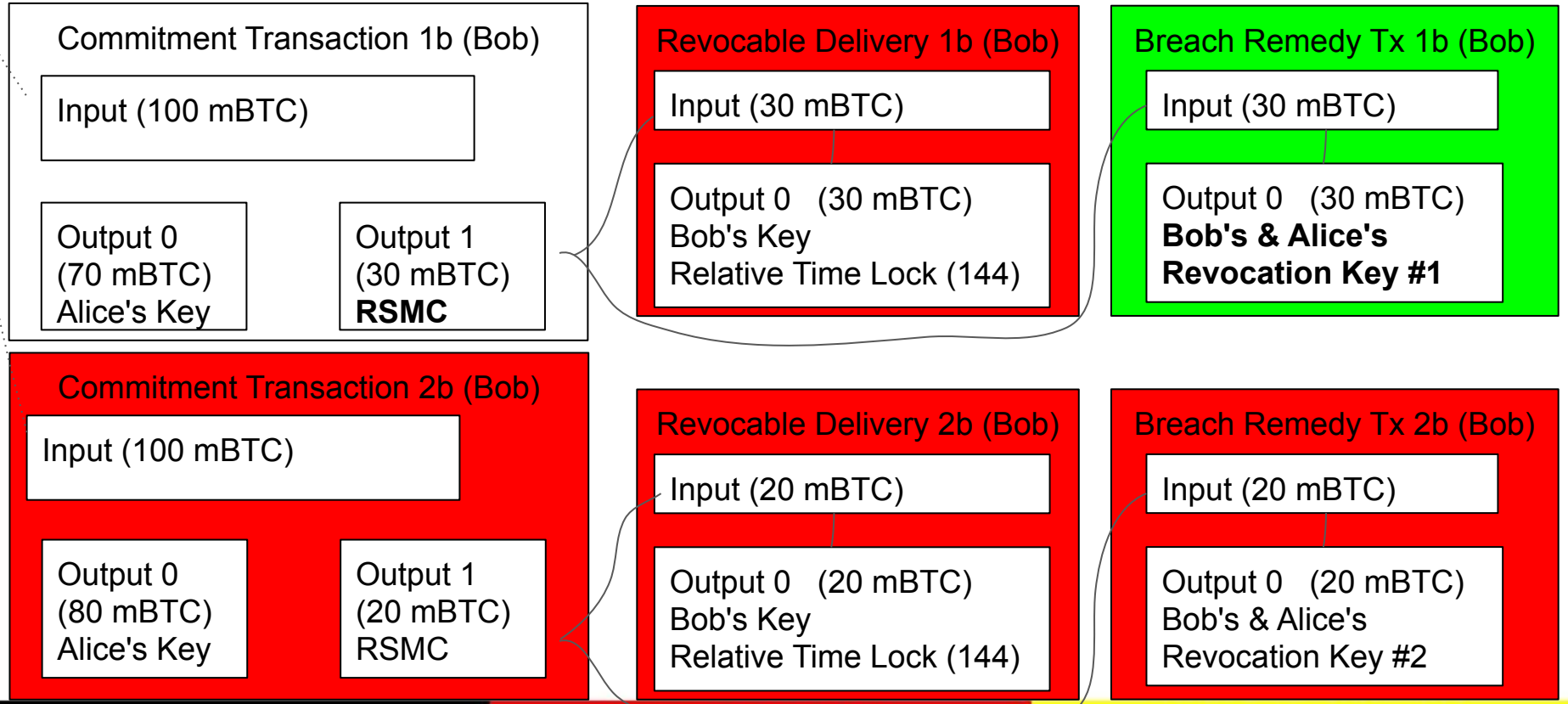
Reference



# A new Commitment Transaction is only signed if the other party reveals their previous revocation Key



# Assume CTX1b is published and mined. Alice should immediately publish BRTX1b to punish Bob's breach



# Some remarks on presented simplifications

- In order to be able to ascribe blame both sides get their own CTXs
  - Commitment Tx 1b, 2b, 3b,... (for Bob)
  - Commitment Tx 1a, 2a, 3a,... (for Alice)
- Transactions for every channel update
  - The Revocation Deliver TXs and  the Breach Remedy TXs are adapted accordingly
- For every Update of the CTXs new Revocation Keys have to be created
  - Otherwise revealing the key once would not help for future updates
    - Future Breach Remedy Transactions could be spent directly
  - They are created from a Hierarchical Deterministic Wallet
- The Commitment Transactions will have even more outputs to support HTLCs
  - HTLC outputs will be needed for routing third party payments through one's channel
- Reading suggestion for a payment channel construction with less Overhead
  - eltoo: A Simple Layer2 Protocol for Bitcoin
    - Christian Decker, Rusty Russell (Blockstream)
    - Olaoluwa Osuntokun (Lightning Labs)
    - Summary at https://www.rene-pickhardt.de/thoughts-about-eltoo-another-protocol-for-payment-channel-management-in-the-lightning-network/

# Some remarks on dust outputs

- In Bitcoin outputs MUST have a certain amount otherwise
  - the fees will be higher to spend the output than the amount attached to it
  - the blockchain can be spammed
- Dust limit depends on the script and is a few hundred satoshis.
  - C.f.: https://bitcoin.stackexchange.com/questions/10986/what-is-meant-by-bitcoin-dust
- If outputs in the commitment transaction are below the dust limit
  - they will be omitted
  - added to the fees
- While lightning supports tiniest payments those amounts cannot always be enforced on chain
- Making tiny payments / routing tiny amounts might make you lose those
  - Lighting implementations can be configured to not accept payments under a certain threshold

# Standard to_local witness script for commitment txs

```
OP_IF
    # Penalty transaction
    <revocationpubkey>
OP_ELSE
    `to_self_delay`
    OP_CSV
    OP_DROP
    <local_delayedpubkey>
OP_ENDIF
OP_CHECKSIG
```

# Summary of what we have achieved now

- Bidirectional payment channel is opened with one on chain transaction
- Channel can stay open for arbitrary time
- The newest Commitment Transaction encodes the balance of a channel
- Old Commitment Transactions are invalidated to prevent broadcasting them
  - Achieved by sharing private keys to effectively allow the other side to spend the output
- The balance sheet can instantly be updated
  - Effectively allowing each party to send funds to the other party
  - Both parties need to collaborate for this to happen
  - Blockchain doesn't need to get involved
  - No fees for updating the balance within the payment channel

Where is the (Lightning) Network of payment channels?

# The Payment Process on the Lightning Network

(Chapter 2 / BOLT 11)

# Comparing the payment process of on chain Bitcoin and off chain Lightning Network payments

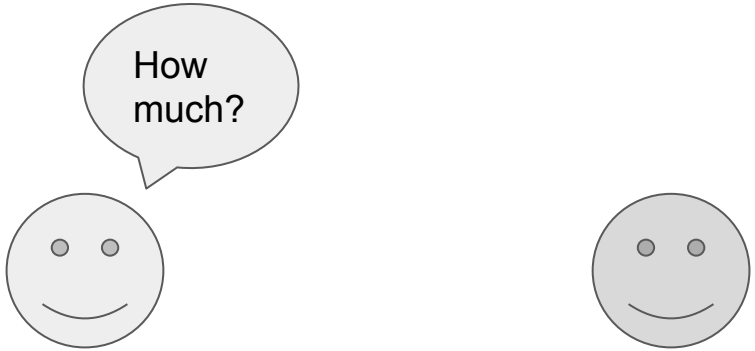
1. Bitcoin address is like a bank account number
2. Bitcoins can be sent to that address
3. Payment requests in Bitcoin exist to have a smoother User experience
4. Lightning cannot send money to another lightning node without invoice
   - unless you use dirty low level routing tricks (https://www.youtube.com/watch?v=Dwl-0cY6KkU)
   - Spontaneous payment extension by lnd in progress
5. In Lightning the recipient or payee first has to issue an invoice
6. The payer in lightning pays the invoice in return for a secret preimage
   - You could probably call it a receipt.

What does it have to do with routing and the fact that lightning is a network?

Also look at: https://www.youtube.com/watch?v=Ol12GrAy8yk

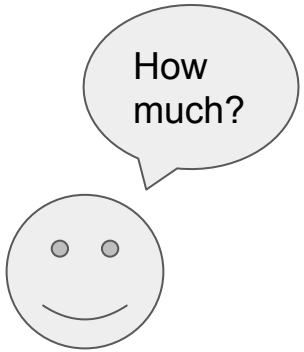
# Direct physical Payment with cash

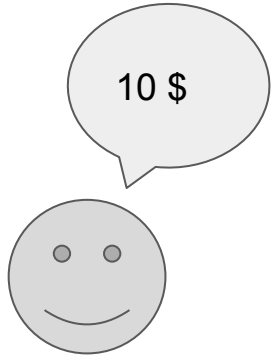
1. Ask for the price



# Direct physical Payment with cash

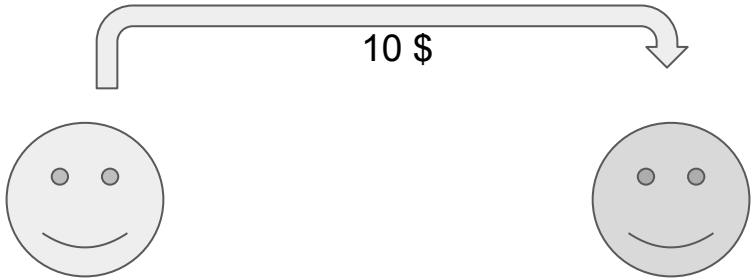
1. Ask for the price



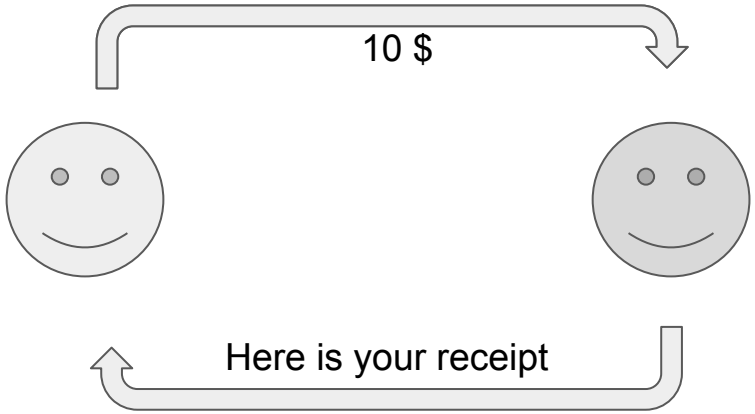
# Direct physical Payment with cash

1. Ask for the price
2. Give the money



# Direct physical Payment with cash

1. Ask for the price
2. Give the money
3. Get the receipt

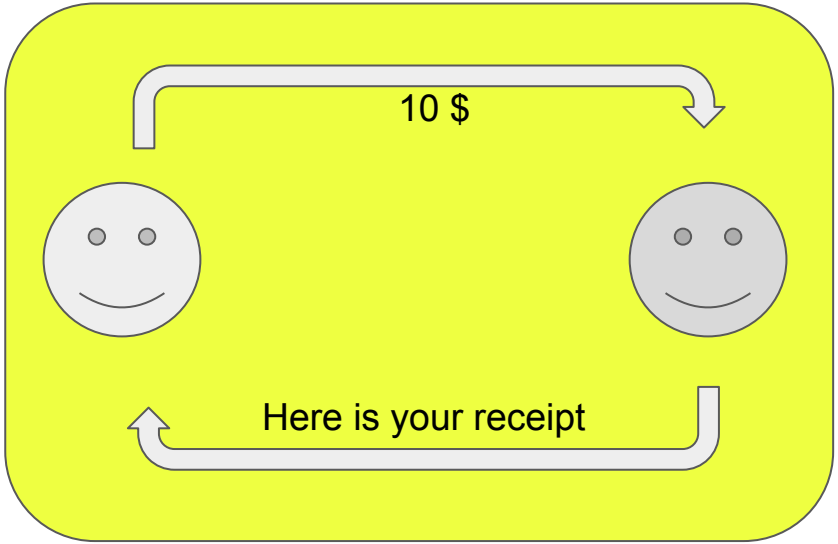


# Direct physical Payment with cash

1. Ask for the price
2. Give the money
3. Get the receipt

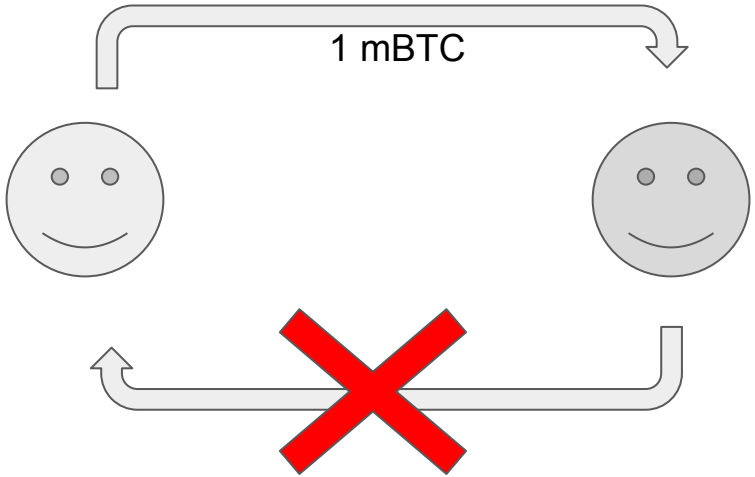
Step two and three are atomic.

(At least ideally they should be atomic)



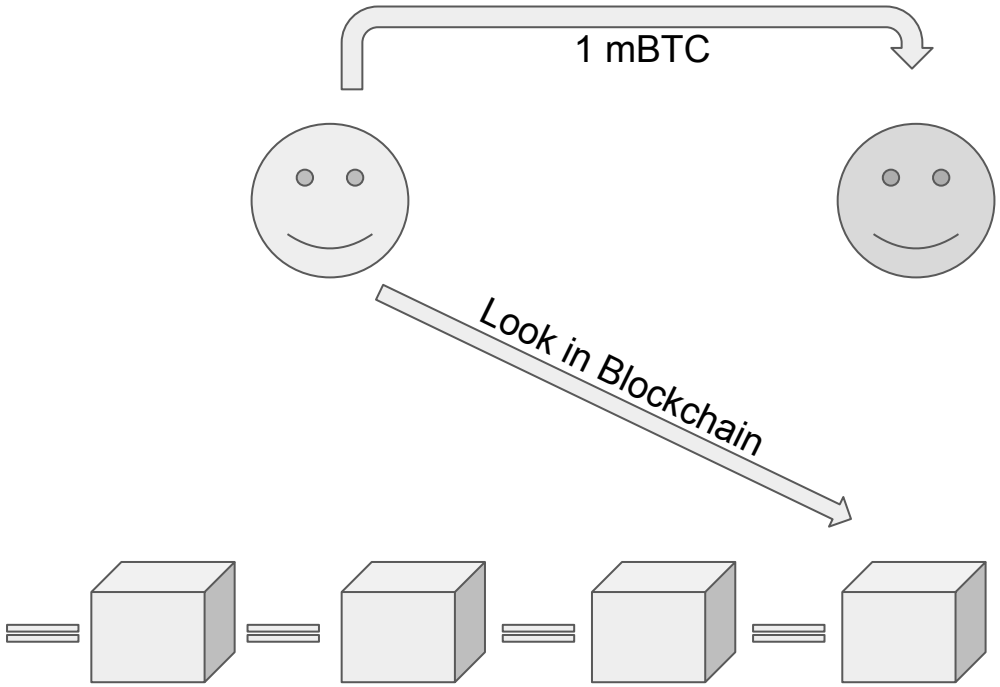
# Direct on Chain Payment with Bitcoin

1. Ask for the price
2. Broadcast transaction



# Direct on Chain Payment with Bitcoin

1. Ask for the price
2. Broadcast transaction
3. See within blockchain



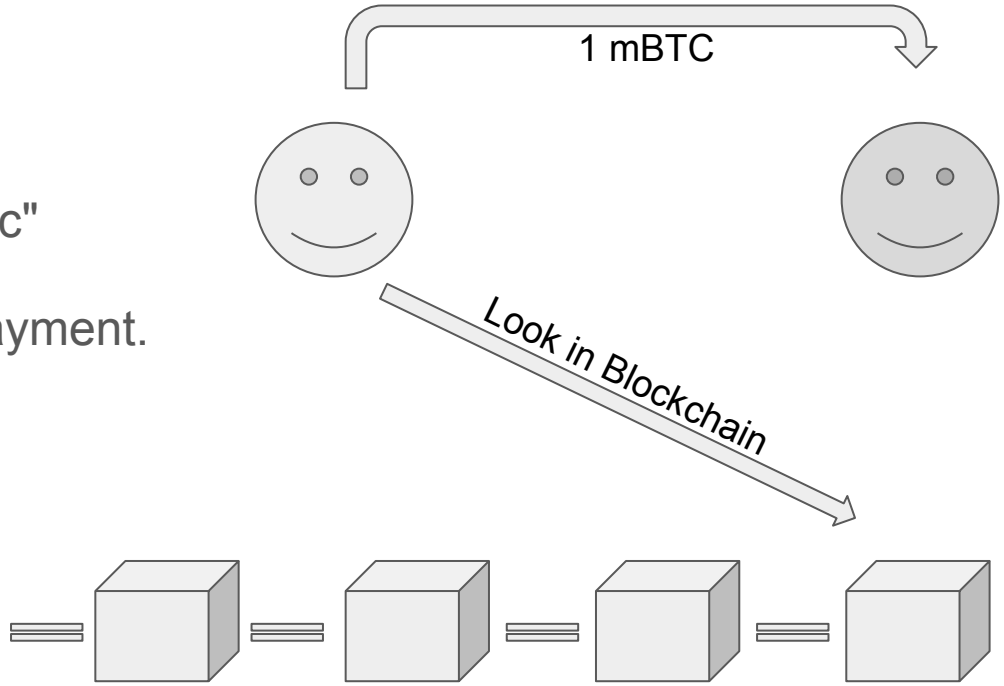
# Direct on Chain Payment with Bitcoin

1. Ask for the price
2. Broadcast transaction
3. See within blockchain

Payment is still somewhat "atomic"

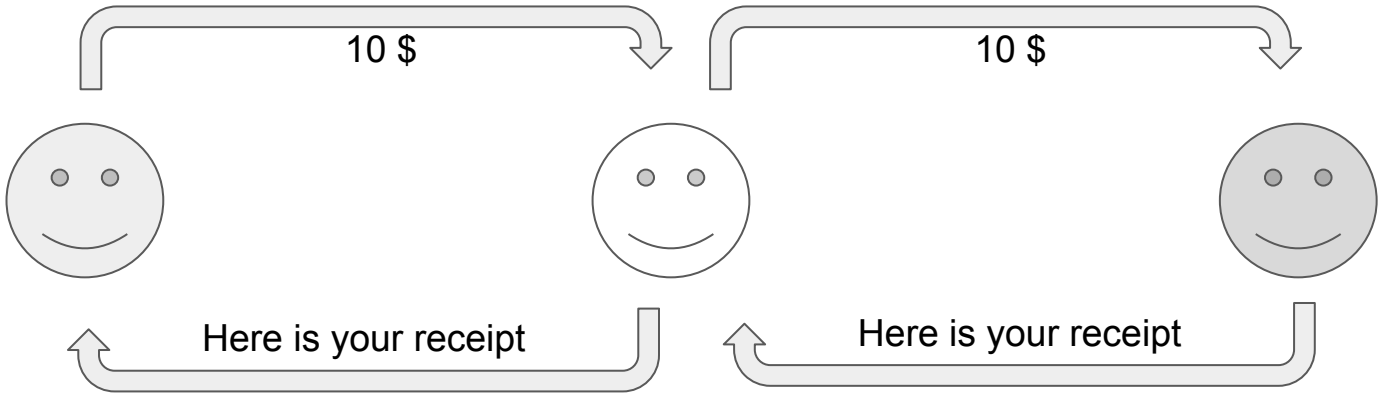
payee must have received the payment.

Payment can't be interrupted



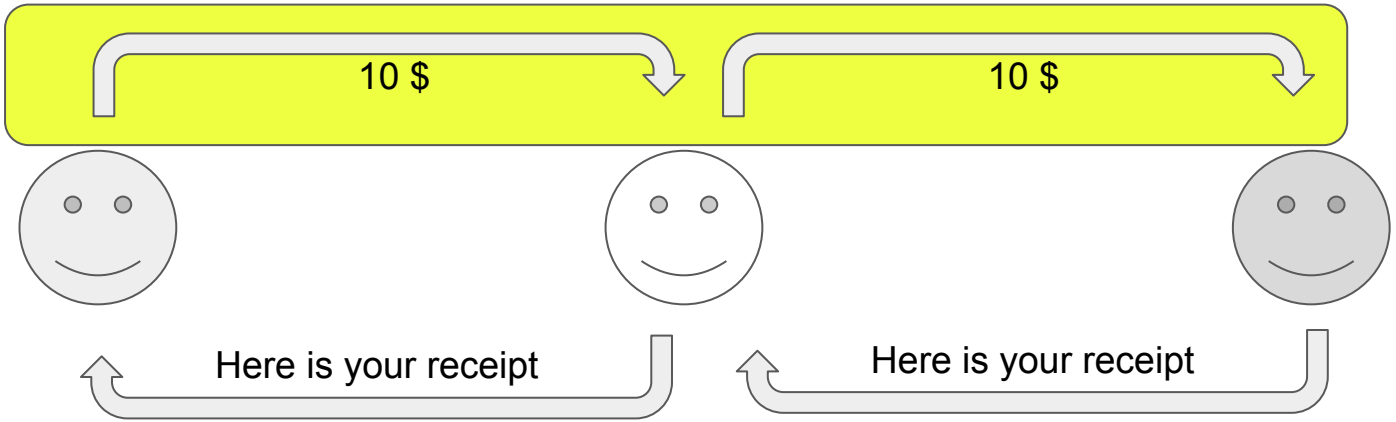
# Indirect physical payment with cash.

1. Assume there is a third party (e.g. Bank, another person) that should forward your payment



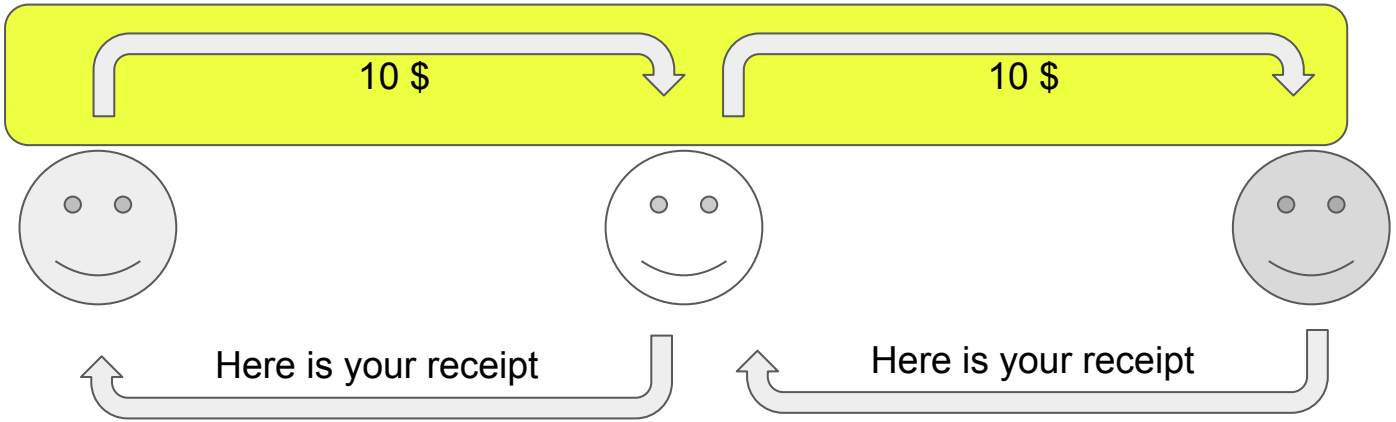
# Indirect physical payment with cash.

1. Assume there is a third party (e.g. Bank, another person) that should forward your payment
2. Should be atomic



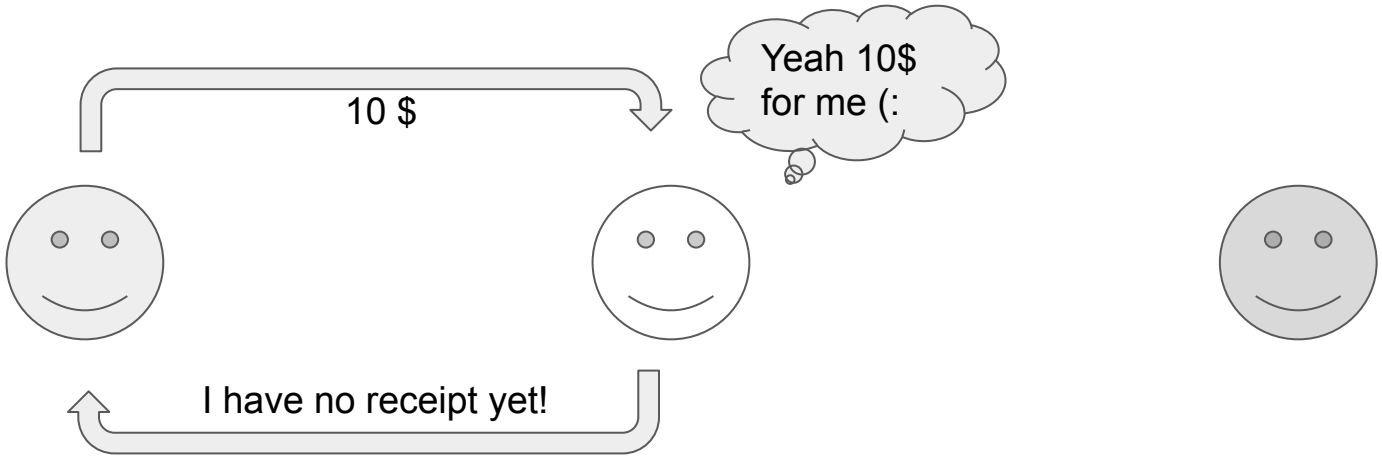
# Indirect physical payment with cash.

1. Assume there is a third party (e.g. Bank, another person) that should forward your payment
2. Should be atomic
   - Can it though?



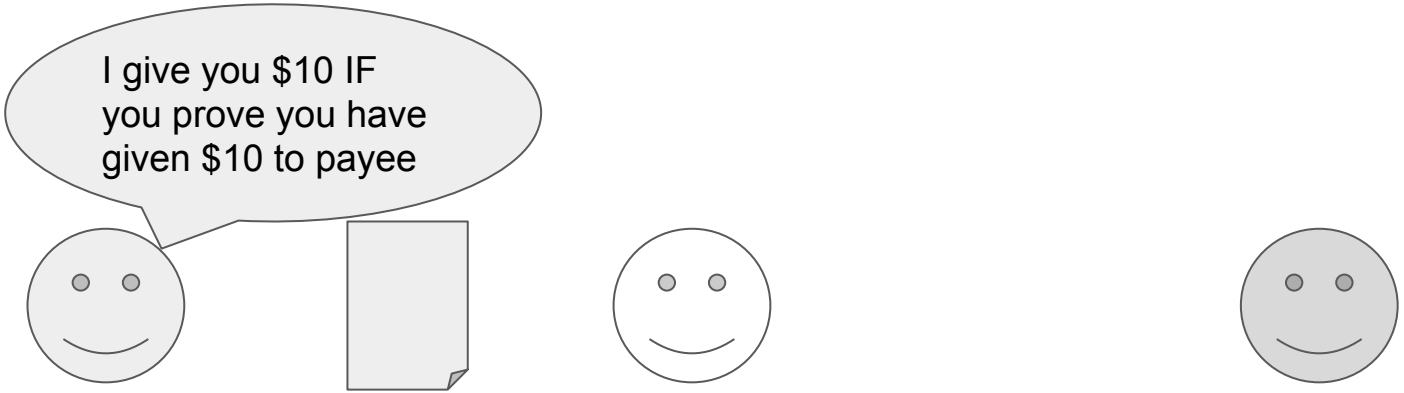
# Indirect physical payment with cash.

1. Assume there is a third party (e.g. Bank, another person) that should forward your payment
2. Should be atomic
   - otherwise



# Indirect physical payment with cash.

1. Assume there is a third party (e.g. Bank, another person) that should forward your payment
2. Should be atomic
3. Make a contract!



# Indirect physical payment with cash.

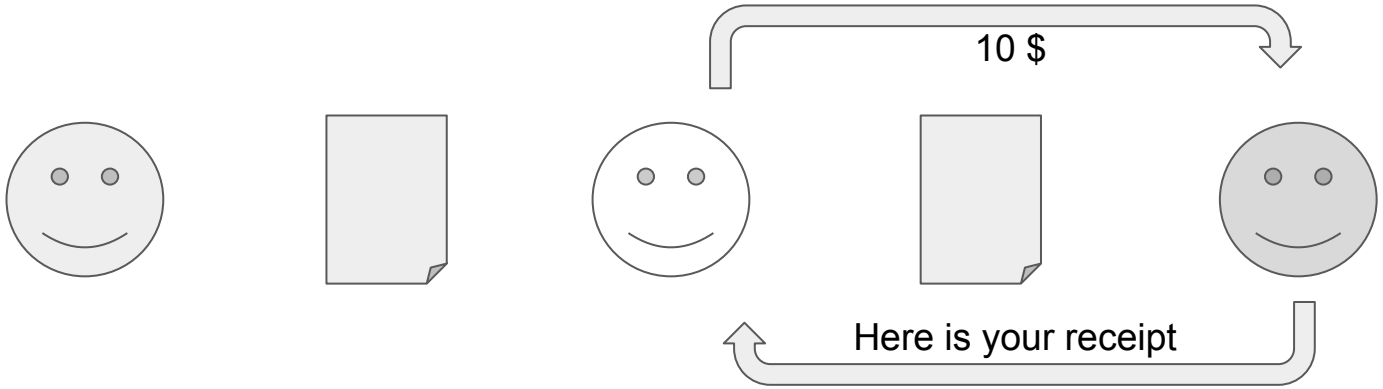
1. Assume there is a third party (e.g. Bank, another person) that should forward your payment
2. Should be atomic
3. Make ~~a~~ contracts!

# Indirect physical payment with cash (+ service fee)

1. Assume there is a third party (e.g. Bank, another person) that should forward your payment
2. Should be atomic
3. Make ~~a~~ contracts!

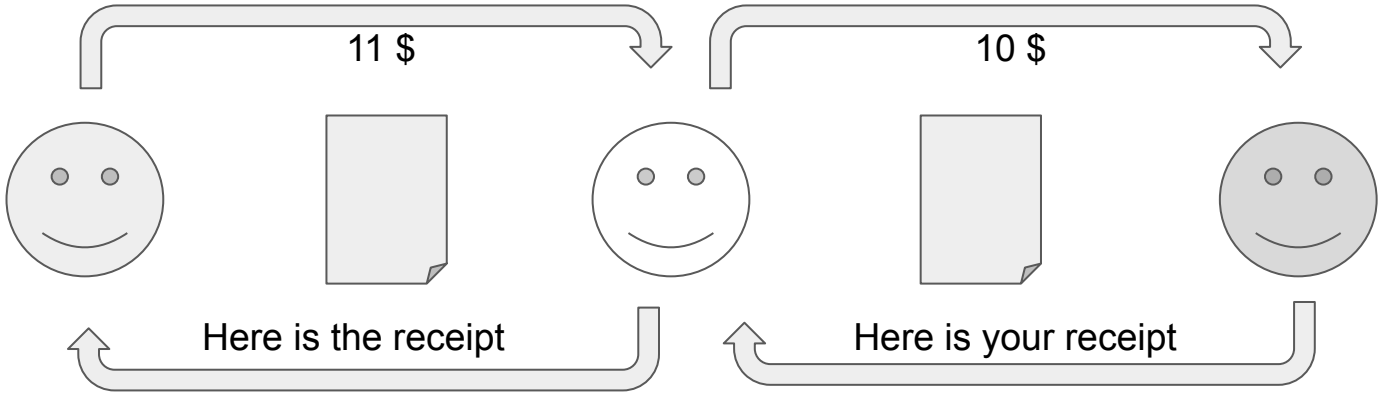
# Indirect physical payment with cash (+ service fee)

1. Assume there is a third party (e.g. Bank, another person) that should forward your payment
2. Should be atomic
3. Make ~~a~~ contracts!



# Indirect physical payment with cash (+ service fee)

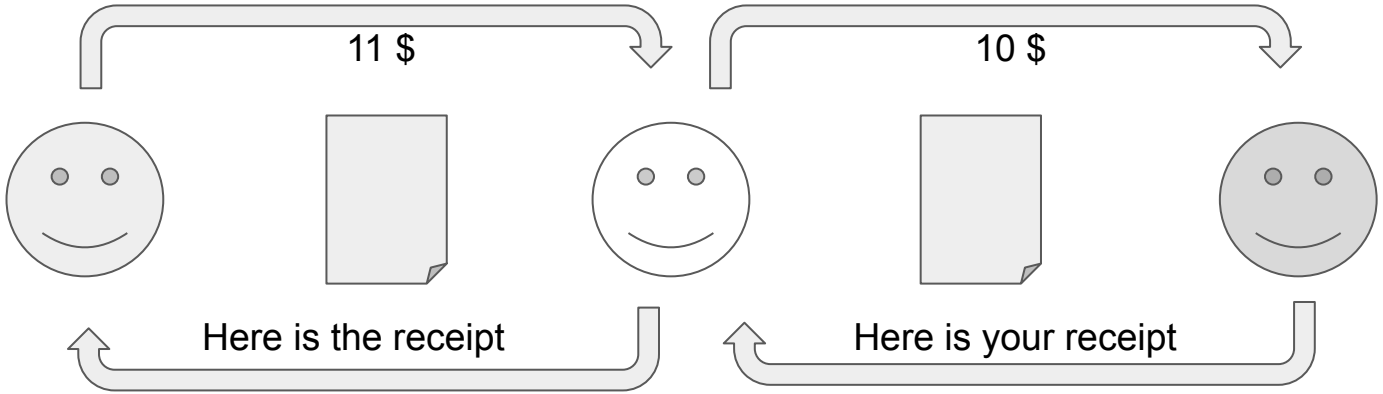
1. Assume there is a third party (e.g. Bank, another person) that should forward your payment
2. Should be atomic
3. Make ~~a~~ contracts!



# Huge risks in the physical world

1. Badly written contract
2. Forgery with the receipt
3. Court case more time consuming than losing the money

→ Programmable smart contracts in Lightning (Hashed Time Locked Contracts)



# Hashed time locked Contracts

1. It is just a regular bitcoin transaction with a special script inside
2. A conditional Payment
3. The transaction is locked for a certain time
    - Which it takes to forward the payment and get the receipt
4. The receipt is a secret random (in best case unique) value called preimage
5. The Hash of the preimage is called the payment hash
    - Can be seen as an identifier for this particular payment
6. It's like the legal contract from the cash setting but without the disadvantages
    - Cheaply enforceable by publishing the contract (bitcoin transaction) to the bitcoin network
    - Much cheaper than the court system
    - Not possible to have a misunderstanding

# Hashed time lock contracts in payment channels

- Payment channels enable the direct atomic payment between two neighbors that share a channel
- Nodes build a network and payment in the network resembles the situation of indirect payments with cash
- Hashed Time Locked Contracts or HTLCs solve this problem
- HTLCs are just another output in the commitment transaction
  - They can be enforced on chain if channel fails
  - They can settle off chain if the preimage is provided

- Before explaining details of routing: How is the payment hash sent over?

# BOLT 11 invoices - following the prefix b32 encoded

- Prefix with amount (e.g: lnbc13u)
- Timestamp
- P - payment hash
- D - description of the payment
- N - pub key of recipient
- H - SHA-256 commitment of purpose
- X - expiry
- C - min_final_cltv_expiry
- F - fallback address
- R - routing info (one or more fields)
  - Pubkey, short_channel_id, fee_base_msat, fee_proportional_millionths, cltv_expiry_delta
- signature

# Invoices should be kept secret (privacy breaches)

- Amount
- Time (window of a potential payment)
- Description
- Fallback address
  - Could be a lie but an expensive one if used
- R field
  - Can reveal hidden channels / nodes
    - In particular short_channel_id points to the on chain funding tx as it contains
      - Height
      - Index
      - Output
  - Can reveal channel balances (field is used for routing suggestions)
- Recipient pubkey (with signature) verifies (!) the authenticity of all that data

# Trustless routing of payments through a network of payment channels

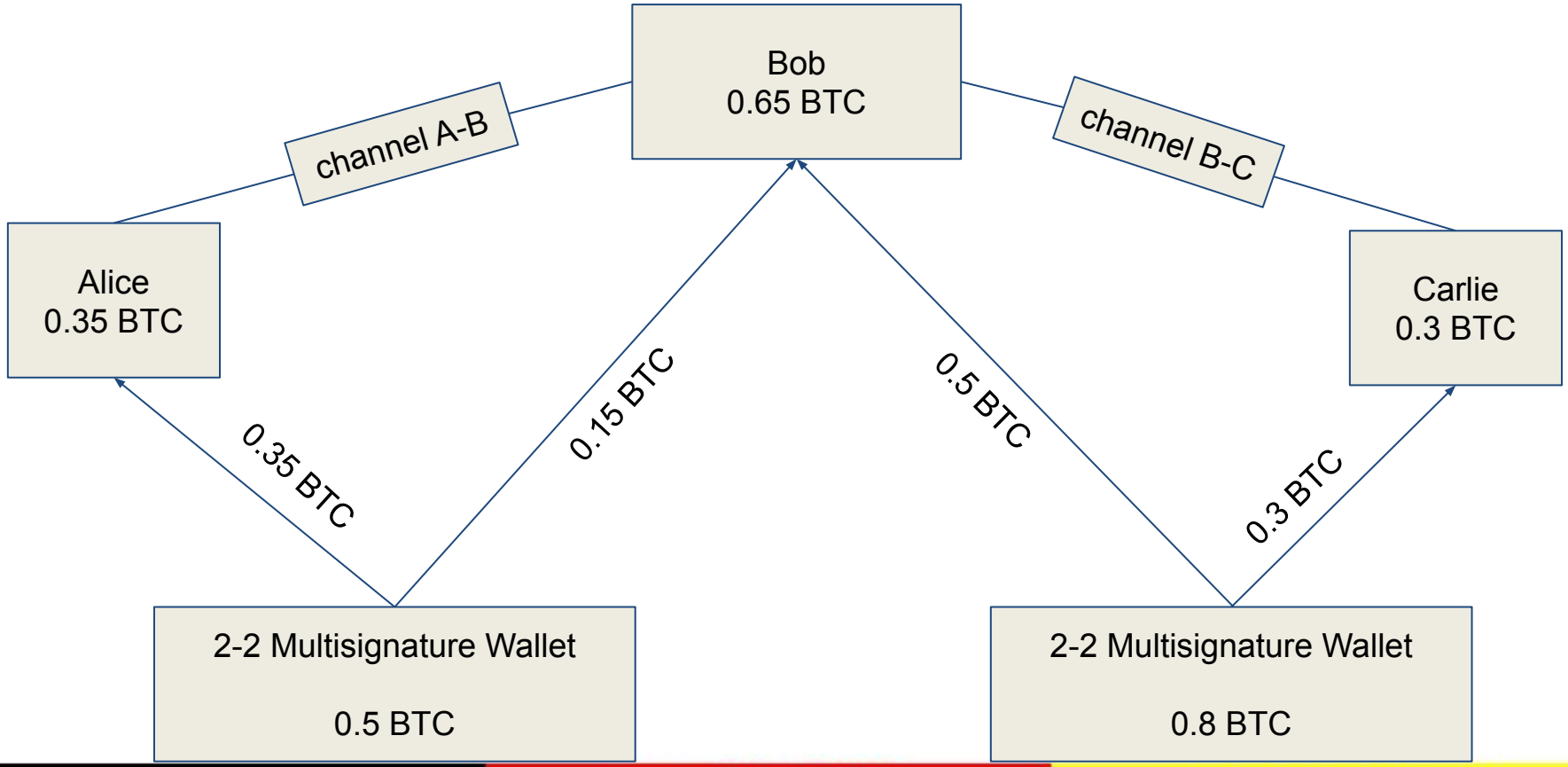
(Chapter 3 / BOLT 03 )

# Purpose of a trustless network of payment channels

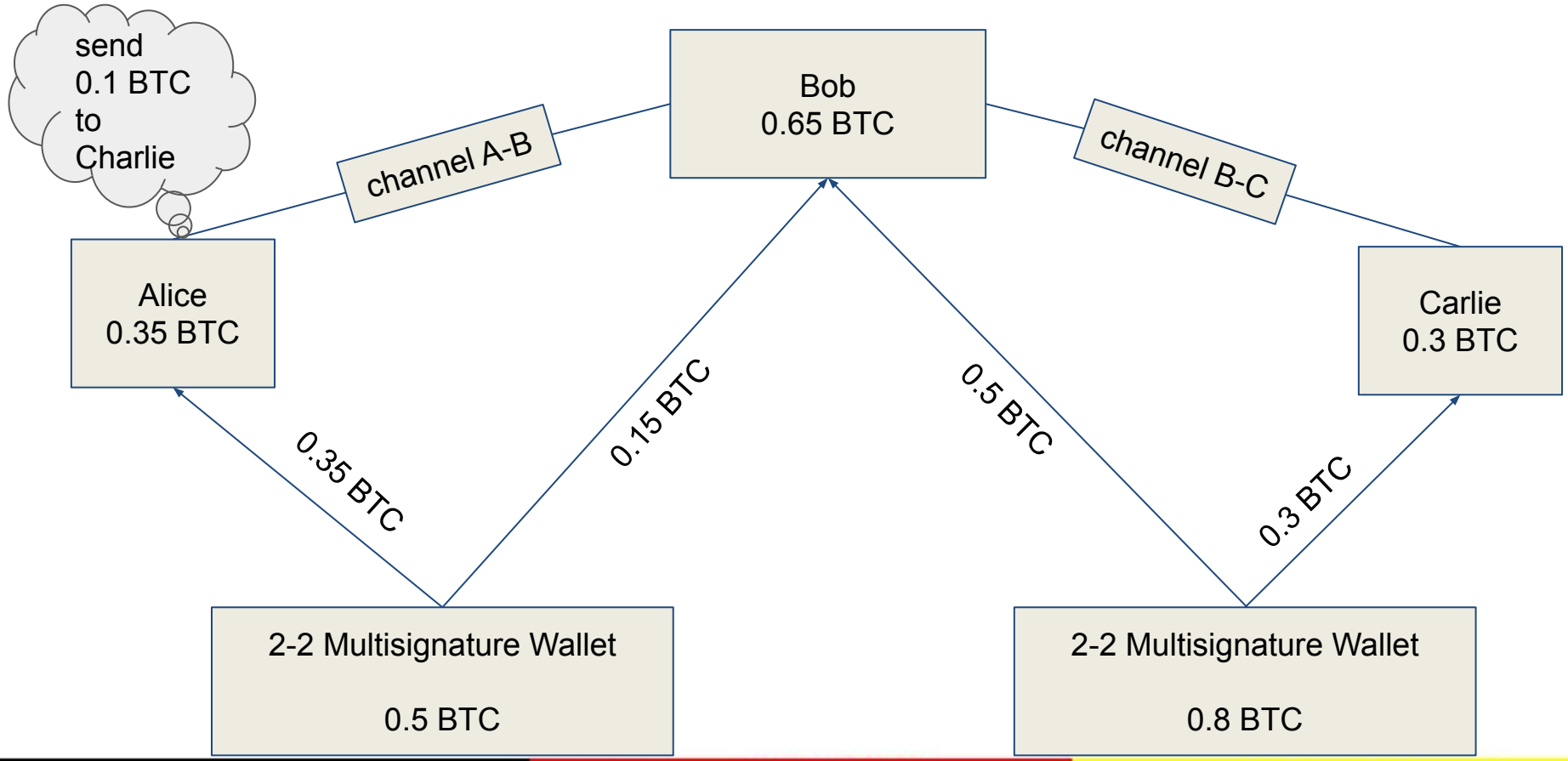
- Sending payments between any two partners
  - With proper behaviour almost Instantaneously
    - Only bound by network traffic over the internet
  - No direct involvement of the blockchain (as long as everyone plays by the rules)
- No need for any partner to trust intermediary nodes
  - The bitcoin network (and the blockchain) needs to be trusted
  - The bitcoin network will settle conflicts if they arise
- Increasing privacy in comparison to Bitcoin transactions
  - Unlike in Bitcoin not every payment made is visible to everyone
- Scale usage of Bitcoin

→ How can this technically be possible?

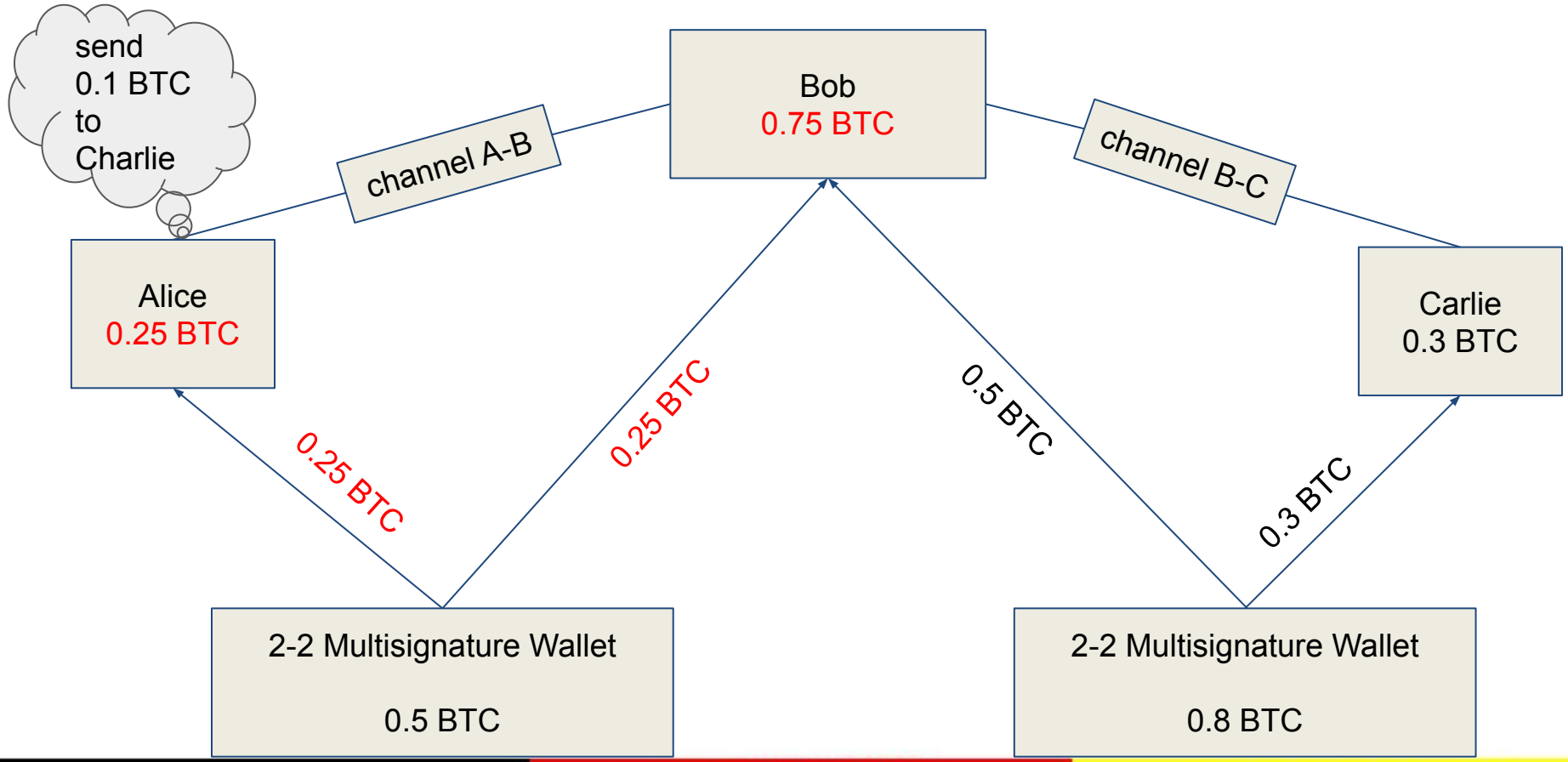
# A trusting (Lightning) Network of payment channels



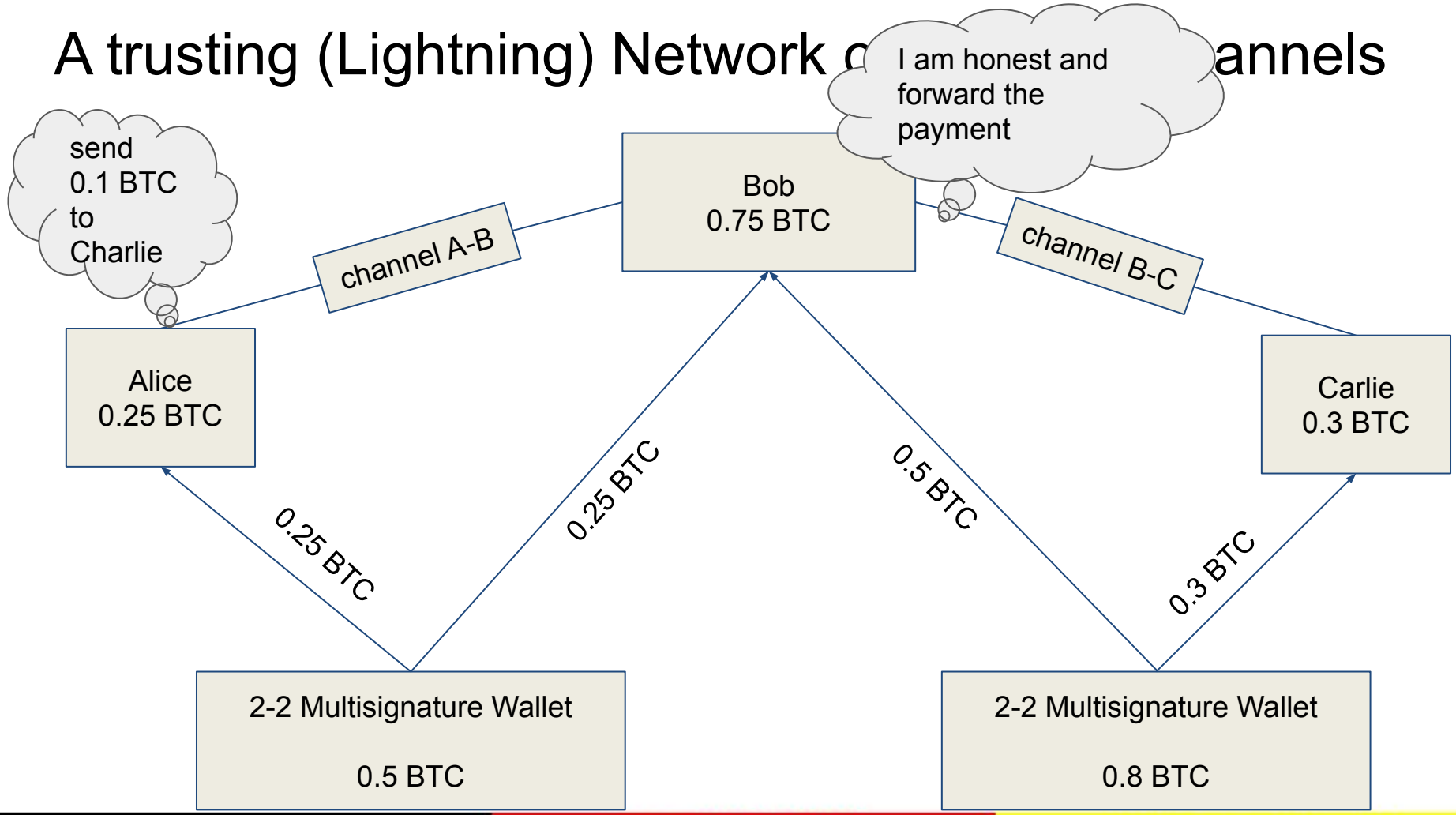
# A trusting (Lightning) Network of payment channels



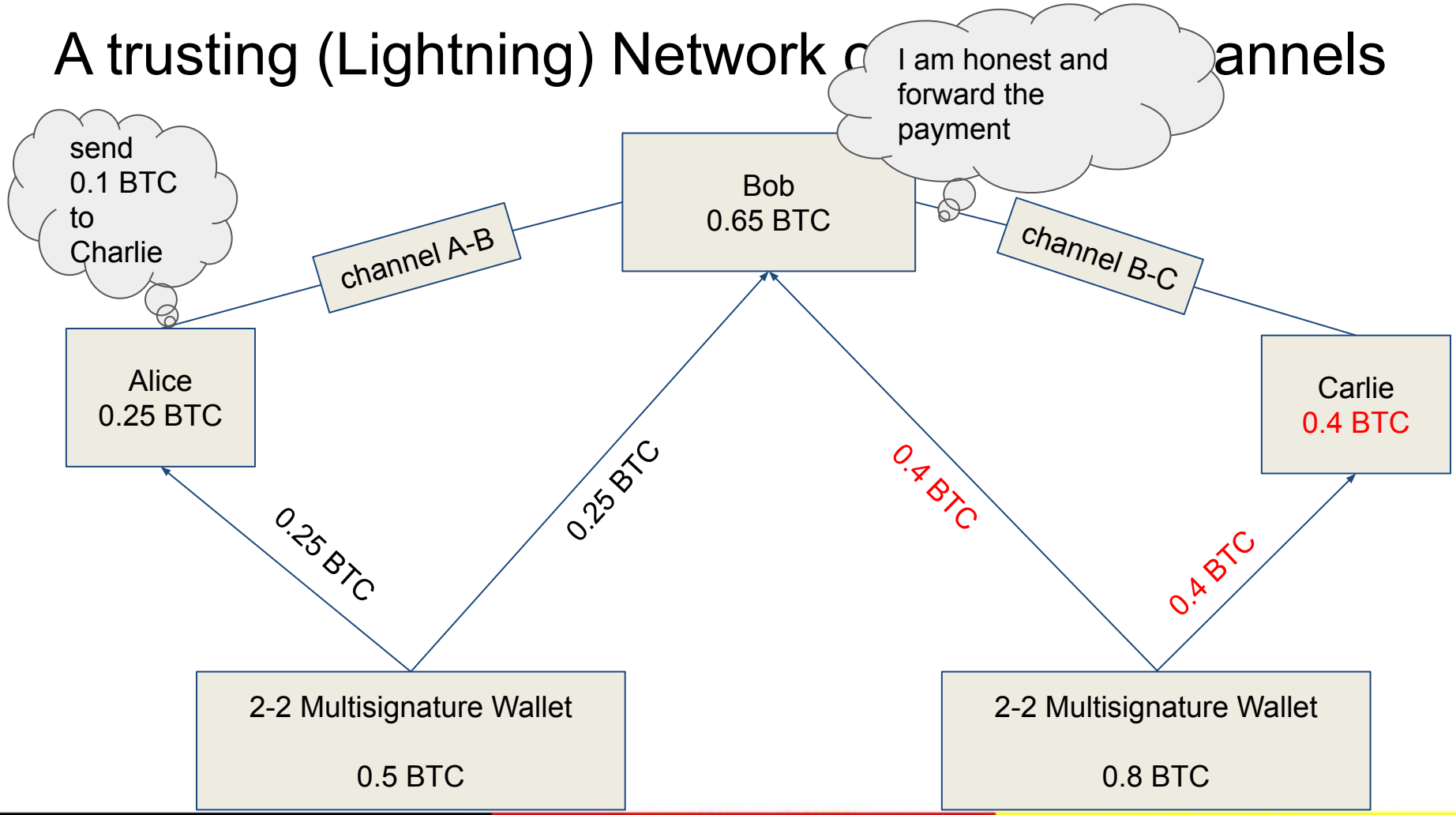
# A trusting (Lightning) Network of payment channels



# A trusting (Lightning) Network c annels



# A trusting (Lightning) Network c annels



# Alice needed to trust Bob to forward the payment

- Both payment channels needed to negotiate new commitment transactions for both sides
- Could we change the output of the CTXs so that routing works trustless?
- Idea (aka Hashed Time Locked Contract - HTLC):
  - Add another conditional output to the Commitment Transactions
  - It can only be spent by the recipient
    - within a certain time frame
    - if the recipient can provide the preimage of some hash
  - After the timeframe the sender can reclaim the funds

# Offered HTLC outputs

```
# To remote node with revocation key
OP_DUP OP_HASH160 <RIPEMD160(SHA256(revocationpubkey))> OP_EQUAL
OP_IF
    OP_CHECKSIG
```

# Offered HTLC outputs

```
# To remote node with revocation key
OP_DUP OP_HASH160 <RIPEMD160(SHA256(revocationpubkey))> OP_EQUAL
OP_IF
    OP_CHECKSIG
OP_ELSE
    <remote_htlcpubkey> OP_SWAP OP_SIZE 32 OP_EQUAL
    OP_NOTIF
        # To local node via HTLC-timeout transaction (timelocked).
        OP_DROP 2 OP_SWAP <local_htlcpubkey> 2 OP_CHECKMULTISIG
```

# Offered HTLC outputs

```
# To remote node with revocation key
OP_DUP OP_HASH160 <RIPEMD160(SHA256(revocationpubkey))> OP_EQUAL
OP_IF
    OP_CHECKSIG
OP_ELSE
    <remote_htlcpubkey> OP_SWAP OP_SIZE 32 OP_EQUAL
    OP_NOTIF
        # To local node via HTLC-timeout transaction (timelocked).
        OP_DROP 2 OP_SWAP <local_htlcpubkey> 2 OP_CHECKMULTISIG
    OP_ELSE
        # To remote node with preimage.
        OP_HASH160 <RIPEMD160(payment_hash)> OP_EQUALVERIFY
        OP_CHECKSIG
    OP_ENDIF
OP_ENDIF
```

# Received HTLC outputs for incomming HTLCs

```
# To remote node with revocation key
OP_DUP OP_HASH160 <RIPEMD160(SHA256(revocationpubkey))> OP_EQUAL
OP_IF
    OP_CHECKSIG
OP_ELSE
    <remote_htlcpubkey> OP_SWAP OP_SIZE 32 OP_EQUAL
    OP_IF
        # To local node via HTLC-success transaction.
        OP_HASH160 <RIPEMD160(payment_hash)> OP_EQUALVERIFY
        2 OP_SWAP <local_htlcpubkey> 2 OP_CHECKMULTISIG
    OP_ELSE
        # To remote node after timeout.
        OP_DROP <cltv_expiry> OP_CHECKLOCKTIMEVERIFY OP_DROP
        OP_CHECKSIG
    OP_ENDIF
OP_ENDIF
```

# HTLC-Timeout & HTLC Success Transactions

```
OP_IF
    # Penalty transaction
    <revocationpubkey>
OP_ELSE
    `to_self_delay`
    OP_CSV
    OP_DROP
    <local_delayedpubkey>
OP_ENDIF
OP_CHECKSIG
```

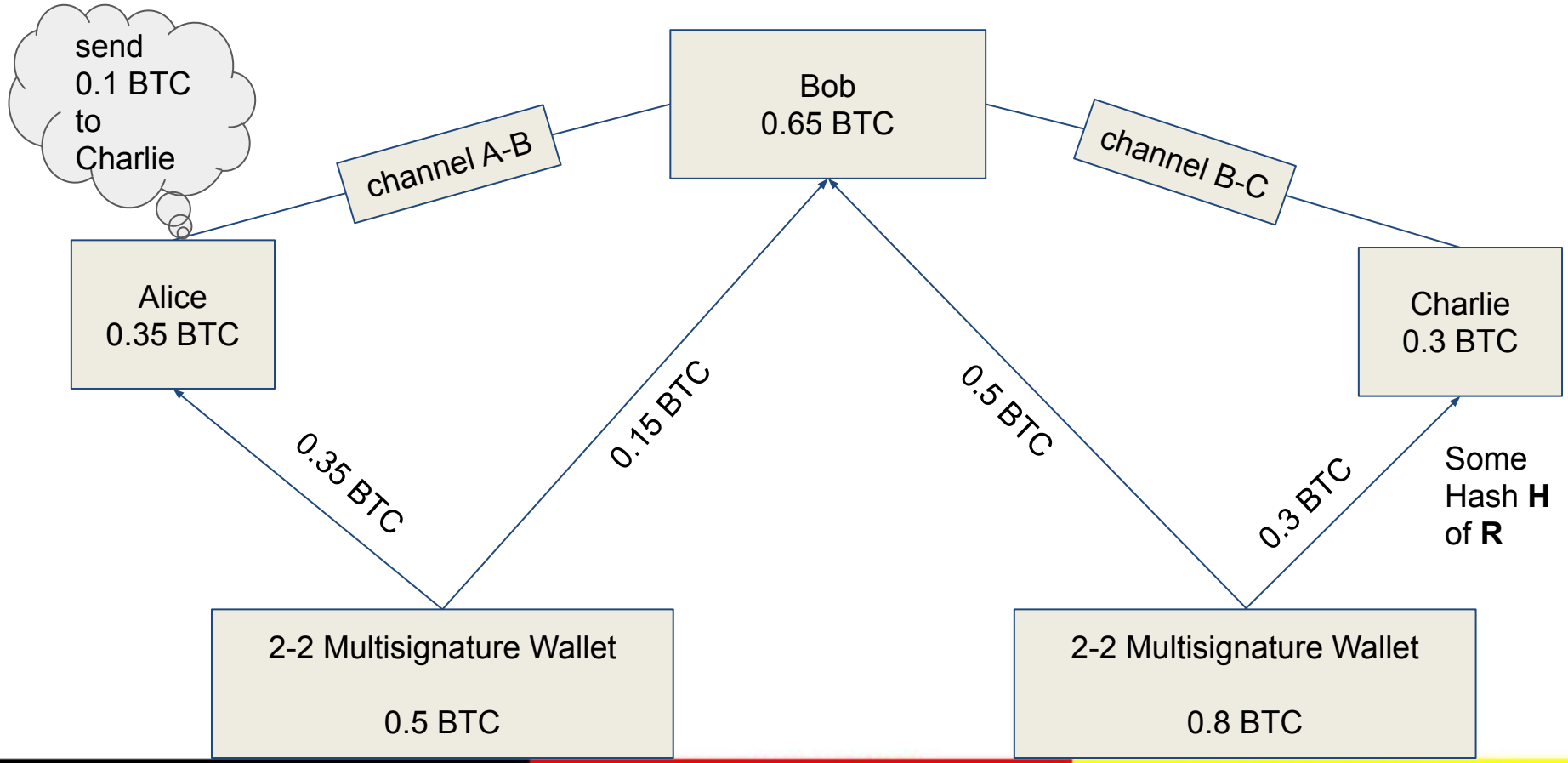
# Key derivation (BOLT 03) for increased privacy

- Each commitment tx uses a unique set of keys
  - localpubkey
  - remotepubkey
- HTLCs use (based on per_commitment_point)
  - local_delaydpubkey
  - revocationpubkey
- Reason ability to share as little information as possible with watching service
  - per_commitment_secret
  - revocation_base_point
  - delayed_payment_basepoint
- Changing localpubkey / remotepubkey prevents guessing of commitment tx id
- Regular / unilateral close should not be seen by the watching service
  - As well as the funding tx which also should not have been identified as such

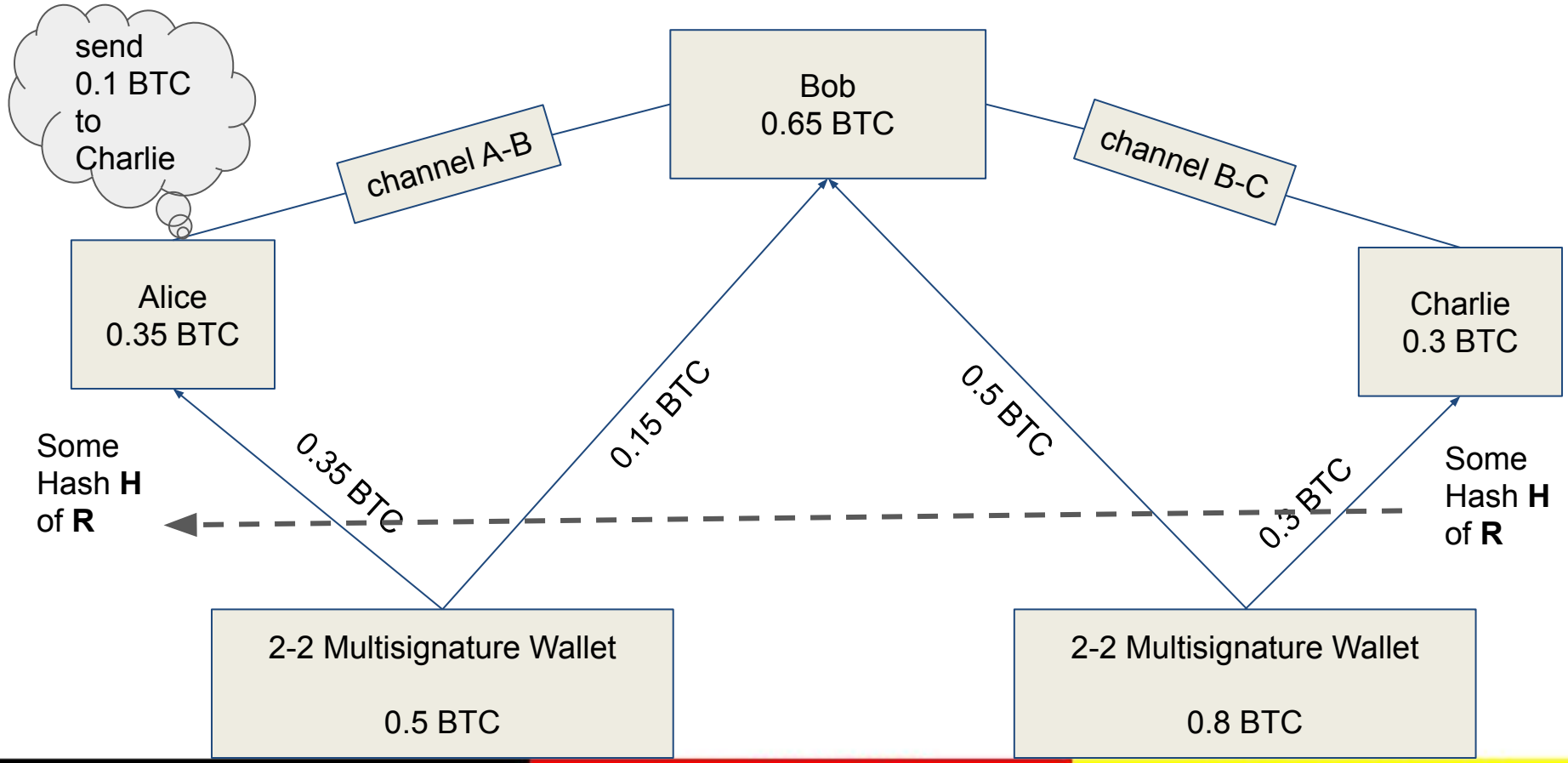
# Some properties of the key derivation

- Deterministic way to generate secretes
- In particular past secrets can be computed from current one
- No need to store all previous revocation secrets
- Revocation pubkeys are blinded
  - revocation_basepoint
  - remotes per_commitment_point
- Technical details described in BOLT 03
  - https://github.com/lightningnetwork/lightning-rfc/blob/master/03-transactions.md#key-derivation

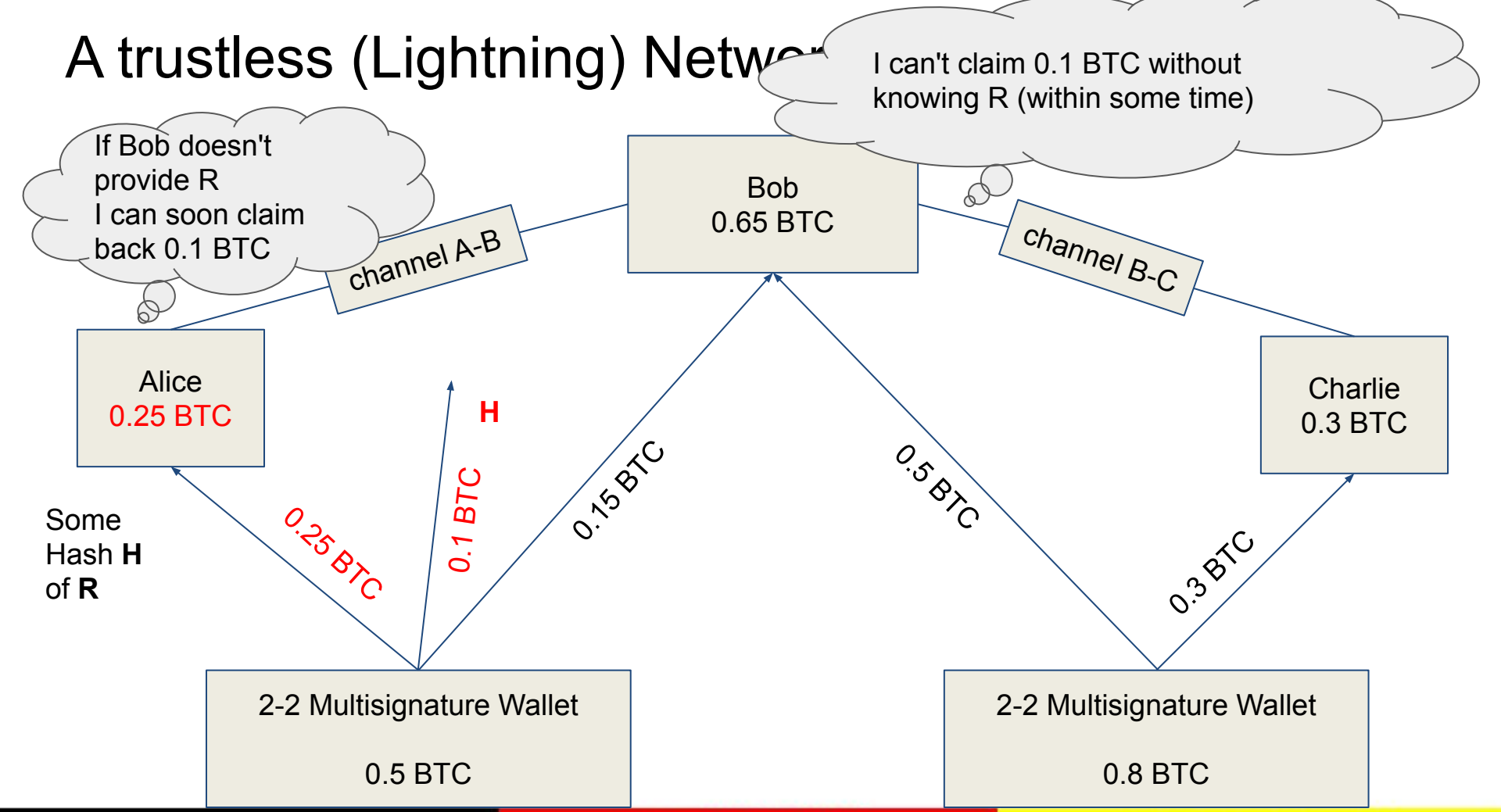
# A trustless (Lightning) Network of payment channels



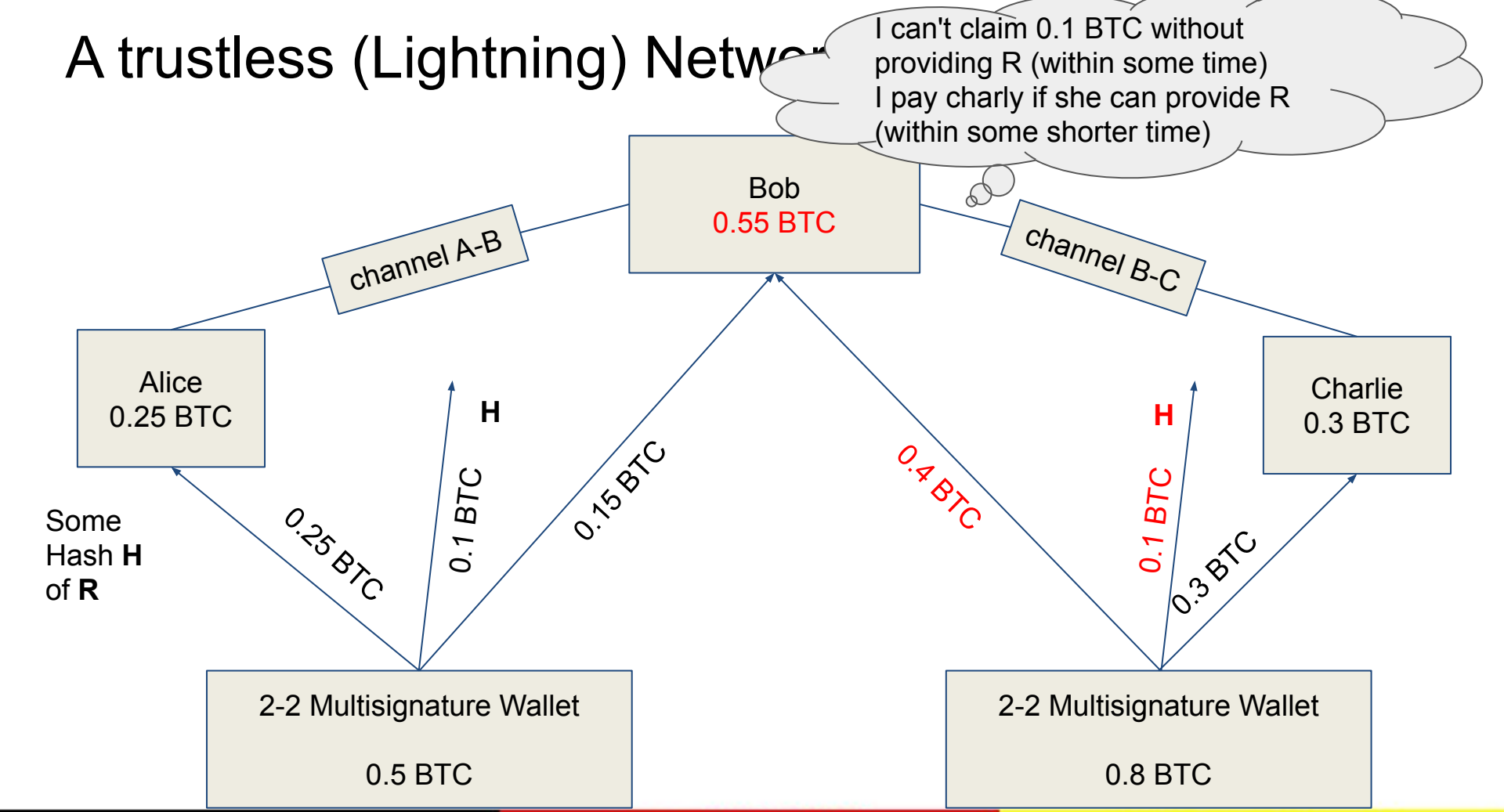
# A trustless (Lightning) Network of payment channels



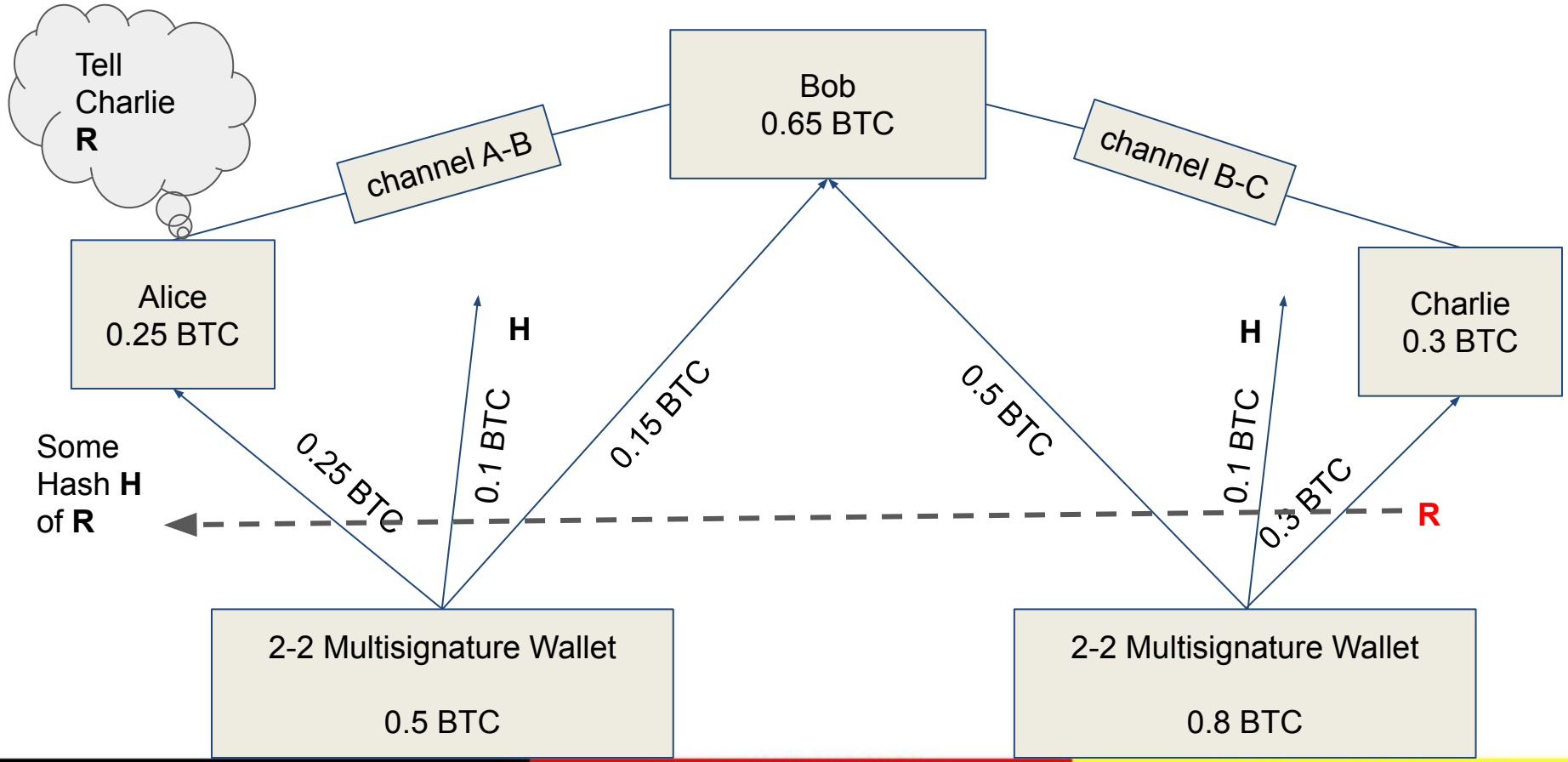
# A trustless (Lightning) Netwo



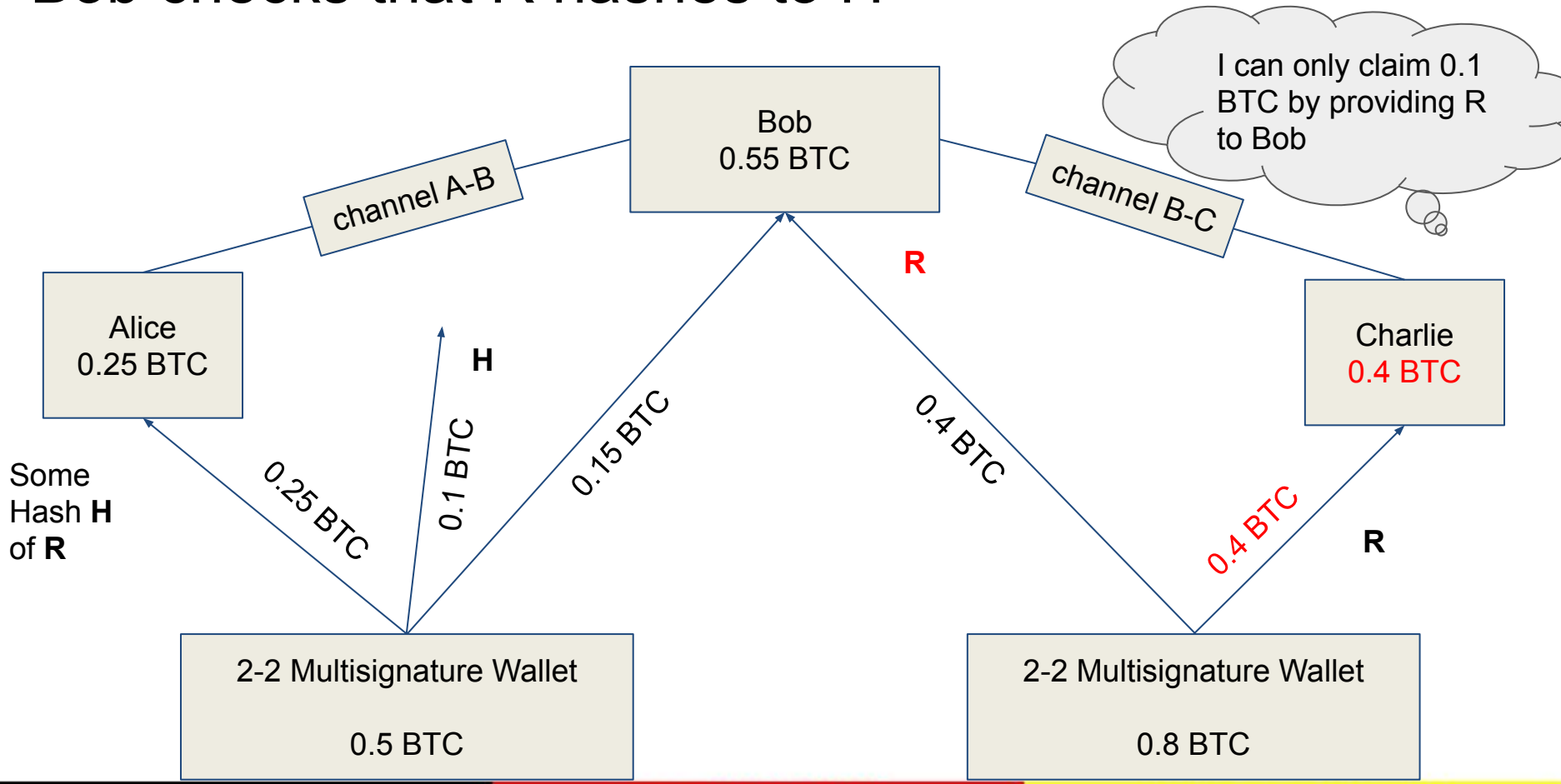
# A trustless (Lightning) Netwo



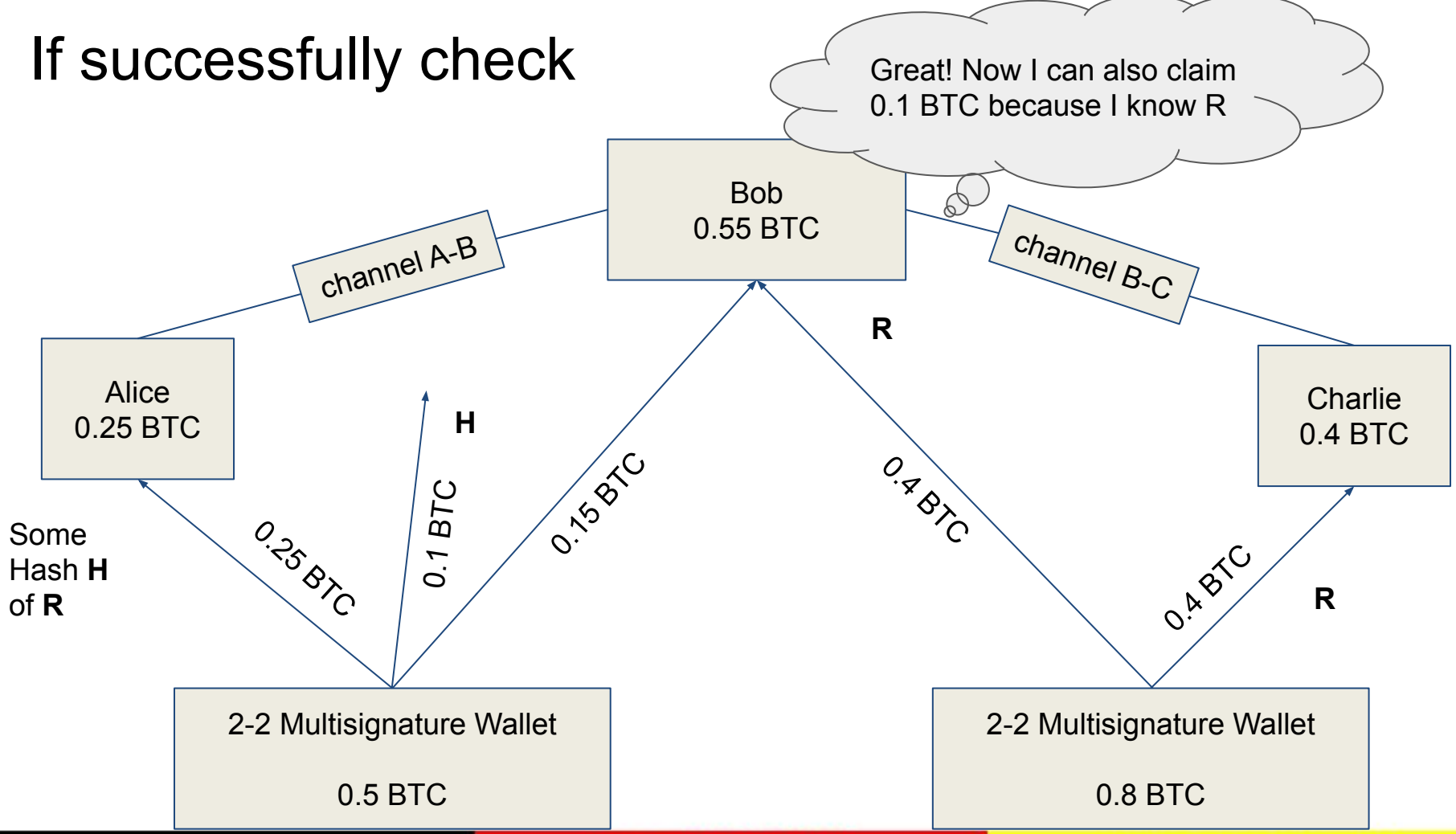
# A trustless (Lightning) Network of payment channels



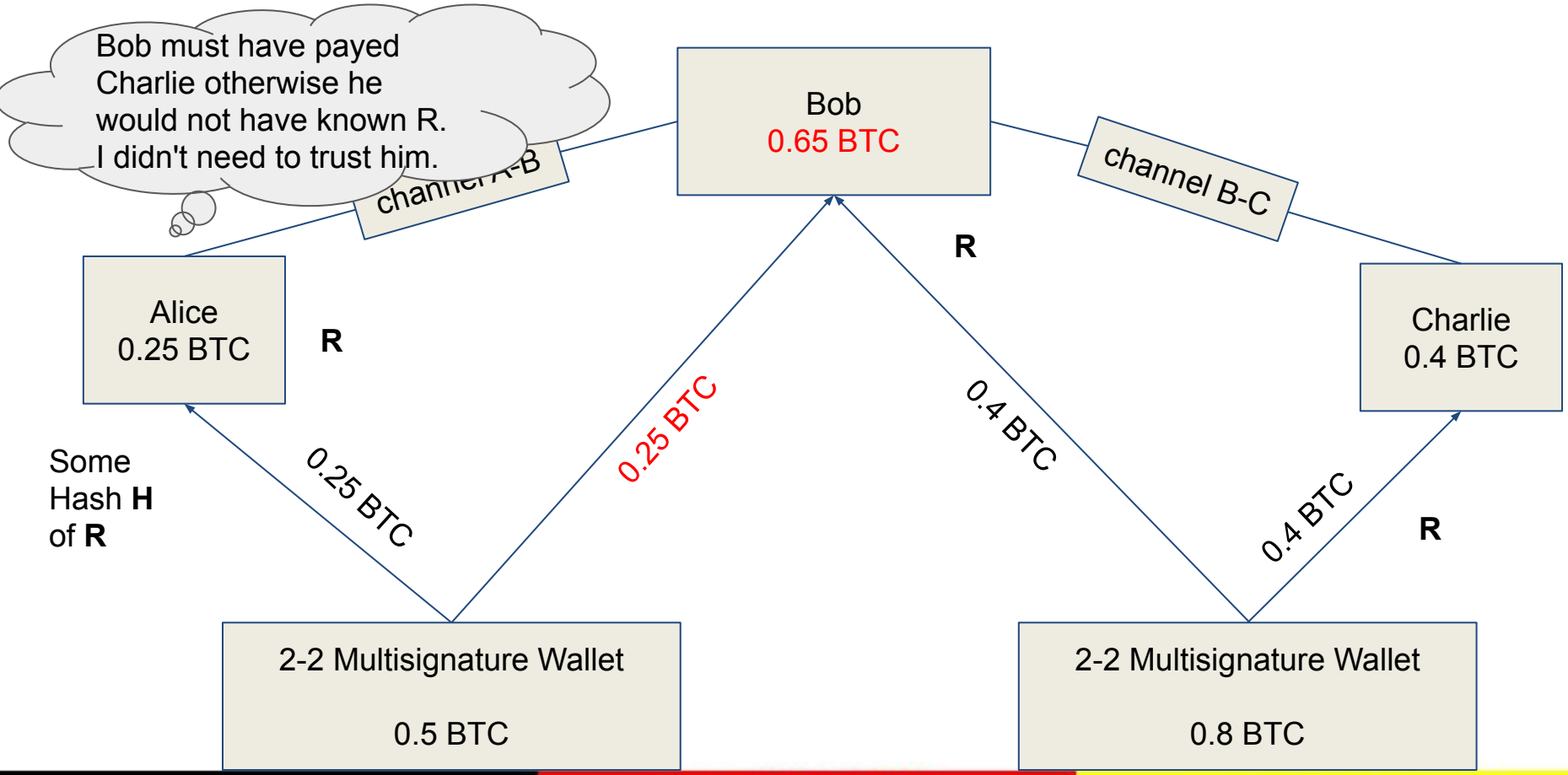
# Bob checks that R hashes to H



# If successfully check



# Bob claims to settle htlc after releasing R



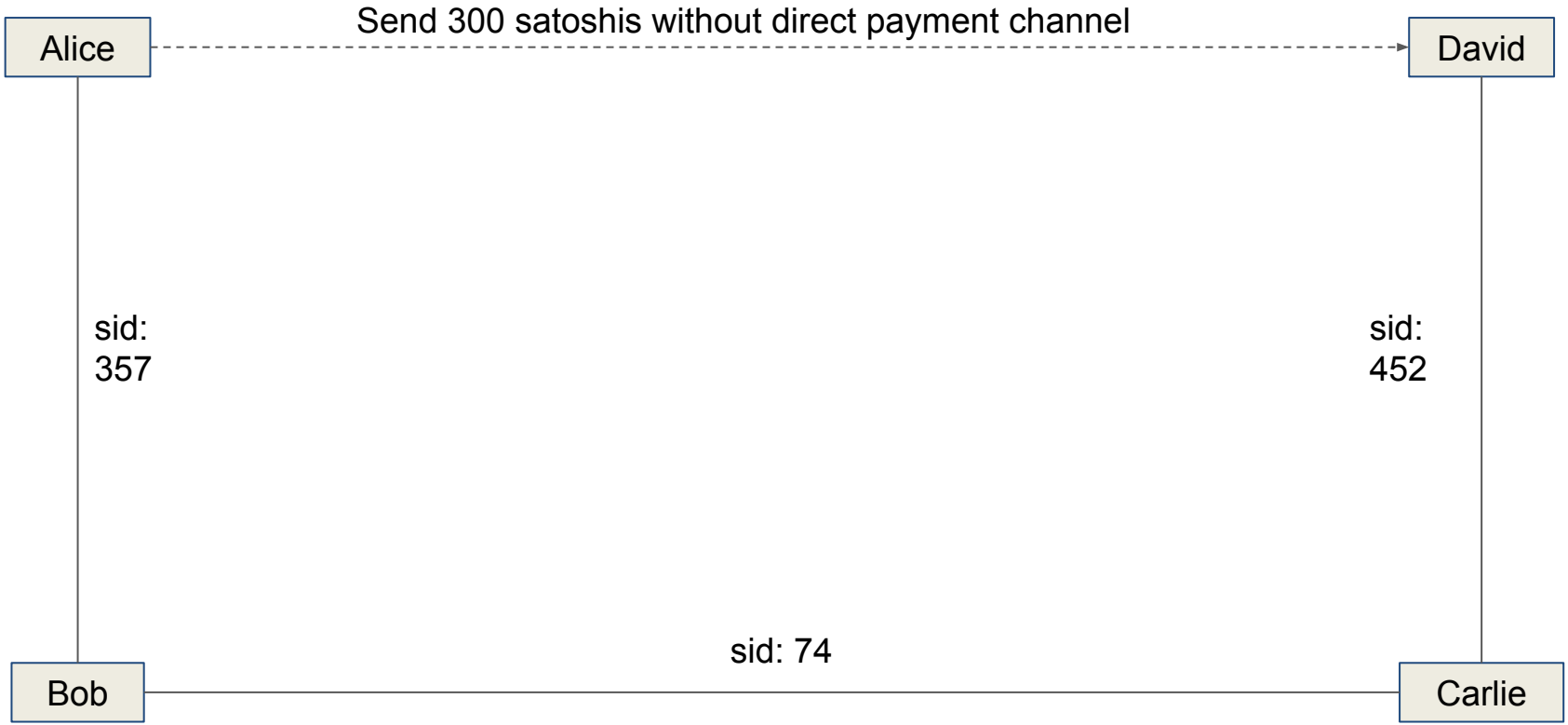
# Source based Onion Routing - Sphinx Mix format Setting up htlcs

(Chapter 4 / BOLT 04)

# Why using the SPHINX mix Format?

- Payer wants to be able to send money without being exposed
- Payee doesn't want to be exposed
- Routing nodes have to be able to send back an error message
- Routing nodes should only know as little information about the payment as possible
  - Payment hash (will stop with payment decorrelation)
  - An upper bound for the amount (will stop with AMP)
  - Know incoming channel
  - Know outgoing channel
- Routing nodes want to be able to verify the authenticity of the onion
  - HMAC checks at every hop
- Research community will tell you
  - It's very compact - uses only little data
- Better than slides: https://www.youtube.com/watch?v=toarjBSPFqI

# Assuming enough capacity on all channels

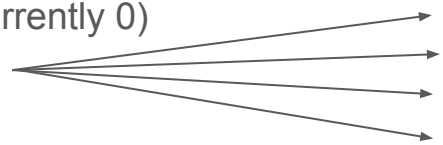


# Create onions starting from David (no payment hash)

- Header
  - Version Byte
  - 33 Byte compressed secp256k1 pubkey
    - Not the sender Pubkey (node_id / static key)
    - But an ephemeral pubkey from the sender for David!
- Payload
  - 1300 Hops_data
  - 20 x 65 Bytes
    - 1: realm (currently 0)
    - 32 per_hop
    - 32: HMAC
    - … filler
- HMAC
  - 32 Byte to verify the integrity

# Per_hop data includes the amount and route info

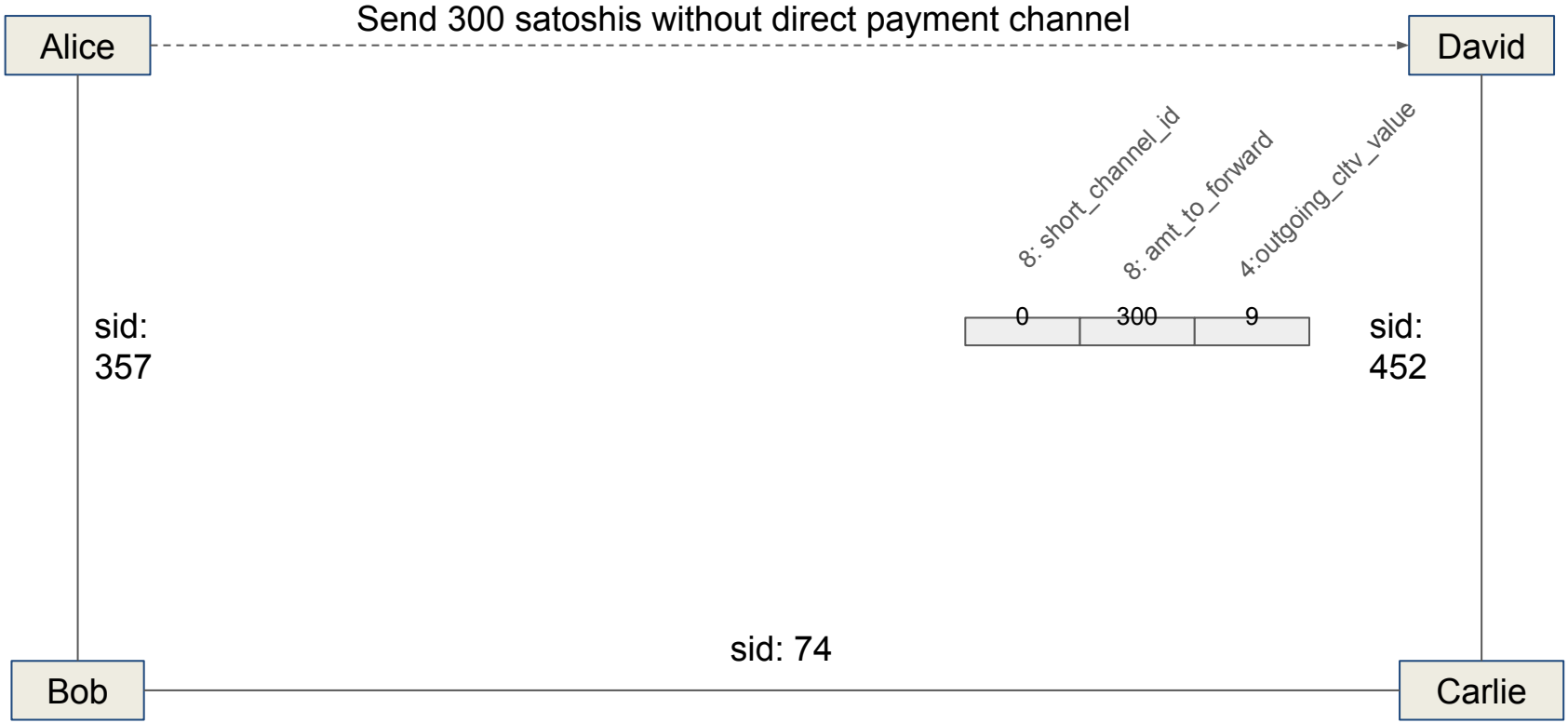
- Header
  - Version Byte
  - 33 Byte compressed secp256k1 pubkey
    - Not the sender Pubkey (node_id / static key)
    - But a pubkey from the sender for David!
- Payload
  - 1300 Hops_data
  - 20 x 65 Bytes
    - 1: realm (currently 0)
    - 32 per_hop
    - 32: HMAC
    - … filler



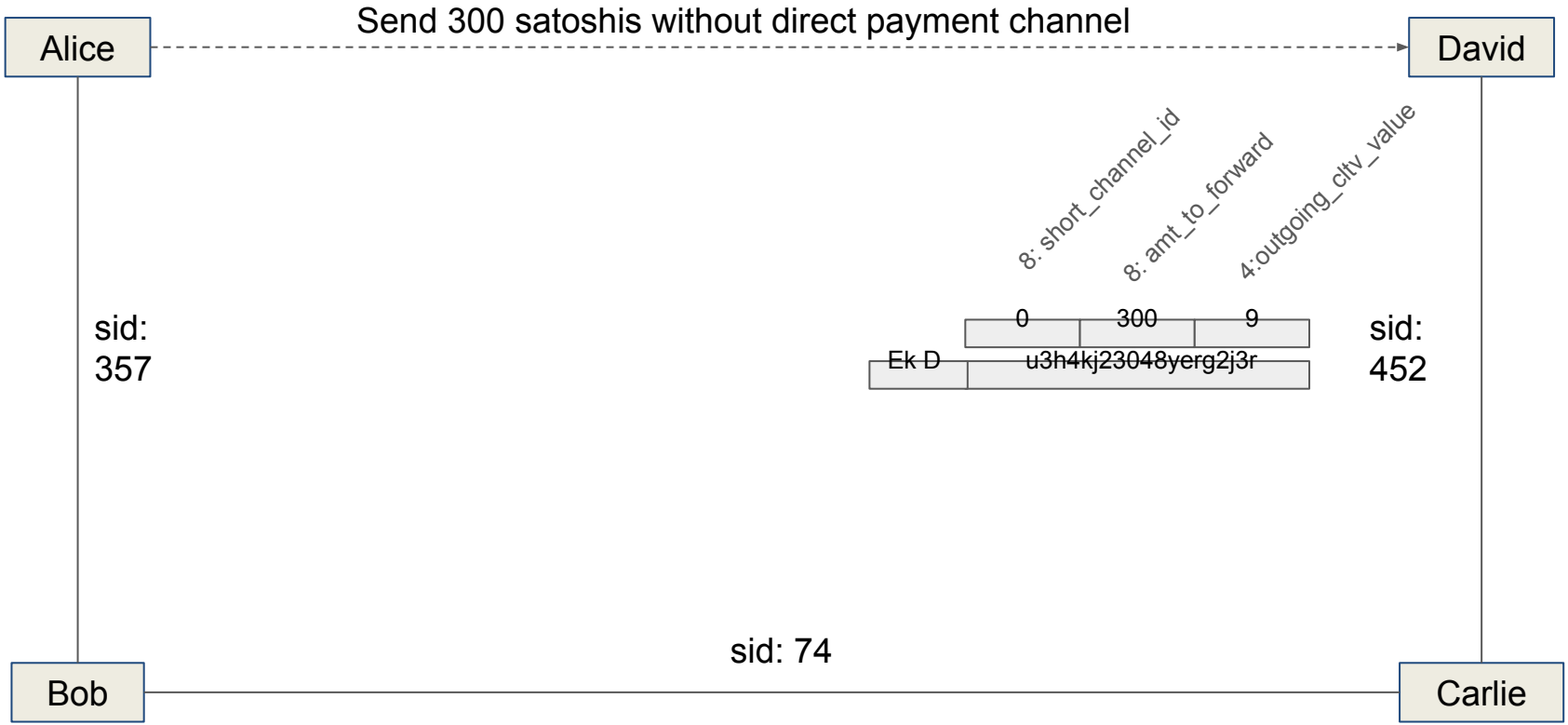
- 8: short_channel_id
- 8: amt_to_forward
- 4: outgoing_cltv_value
- 12: padding (for backwards compatibility)

- HMAC
  - 32 Byte to verify the integrity

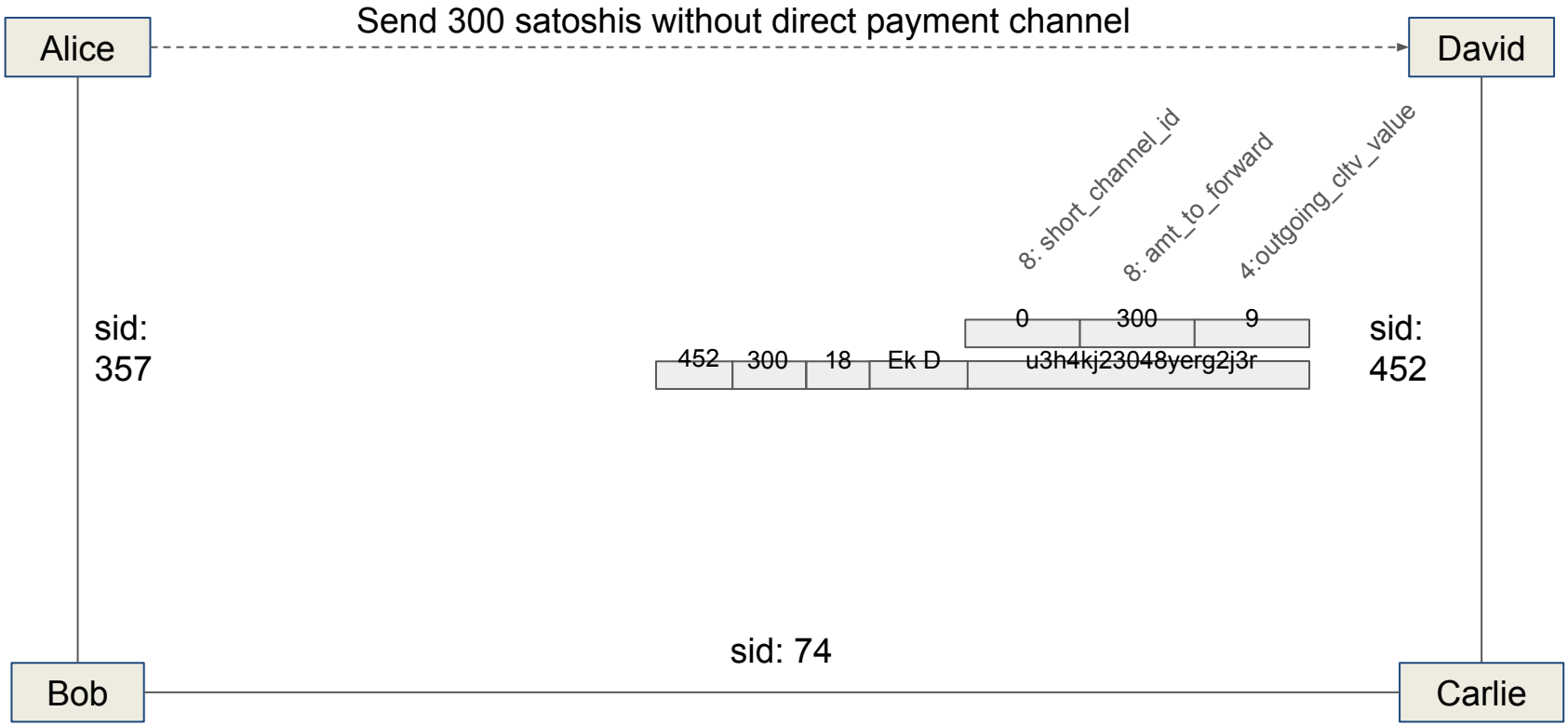
# Per_hop payload for david (simplified!)



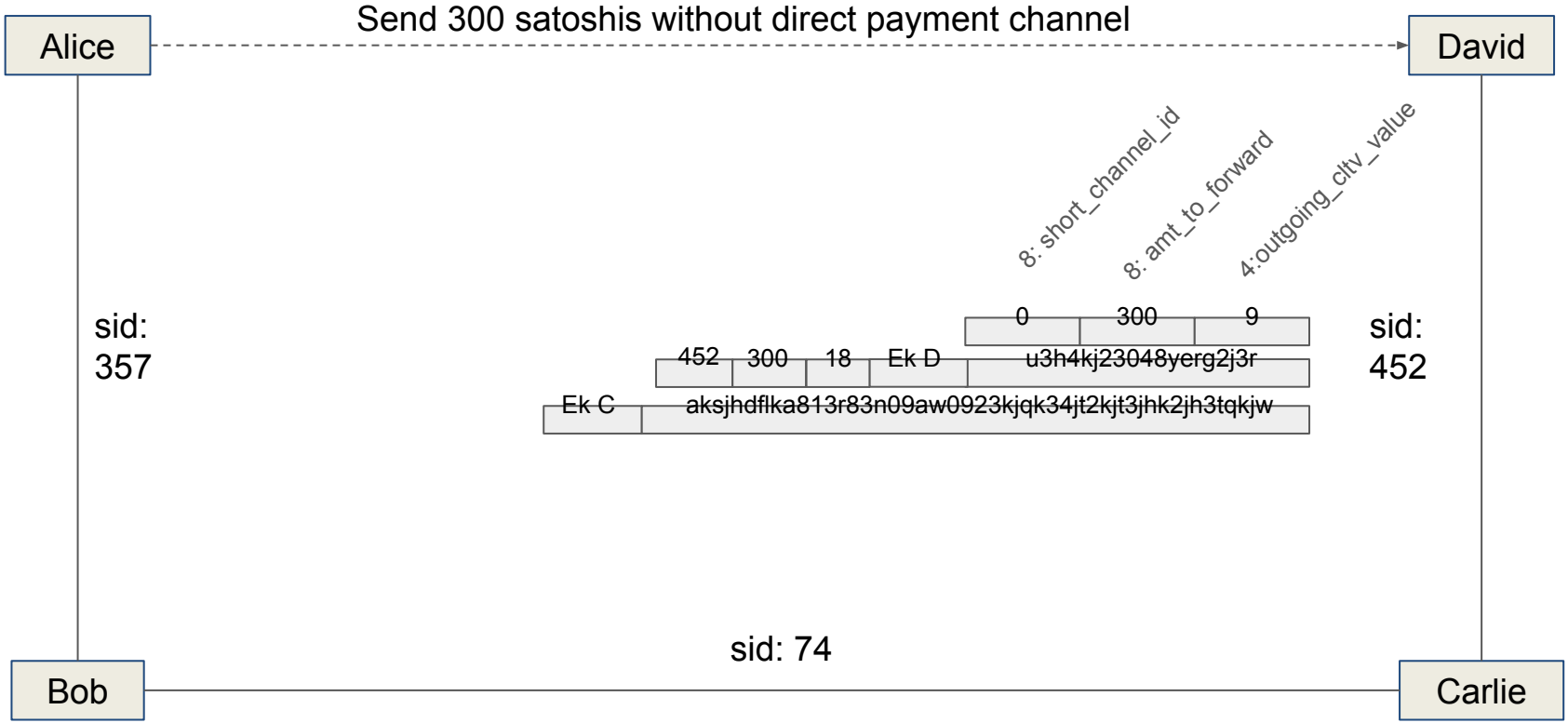
# Onion for David (without HMAC and filler)



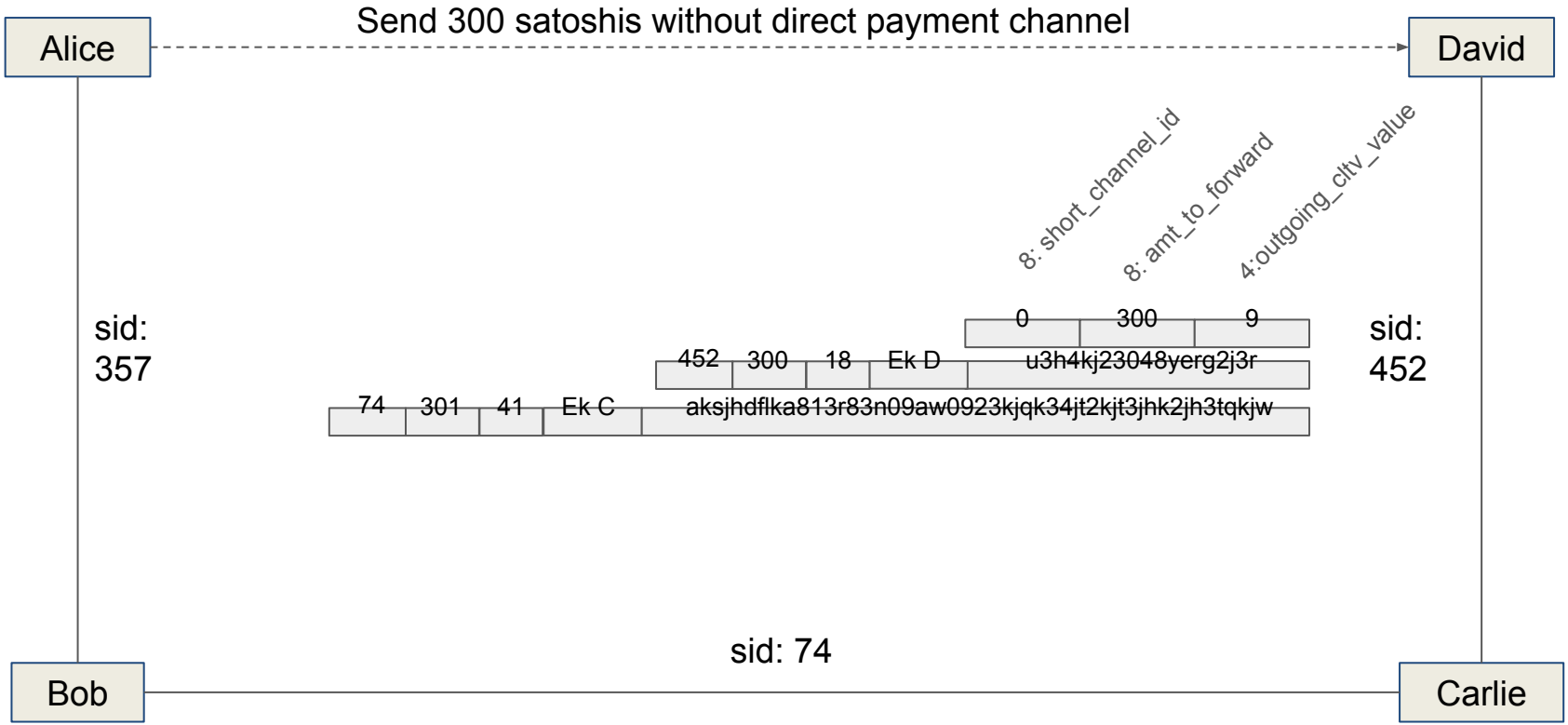
# Payload for Charlie (Simplified)



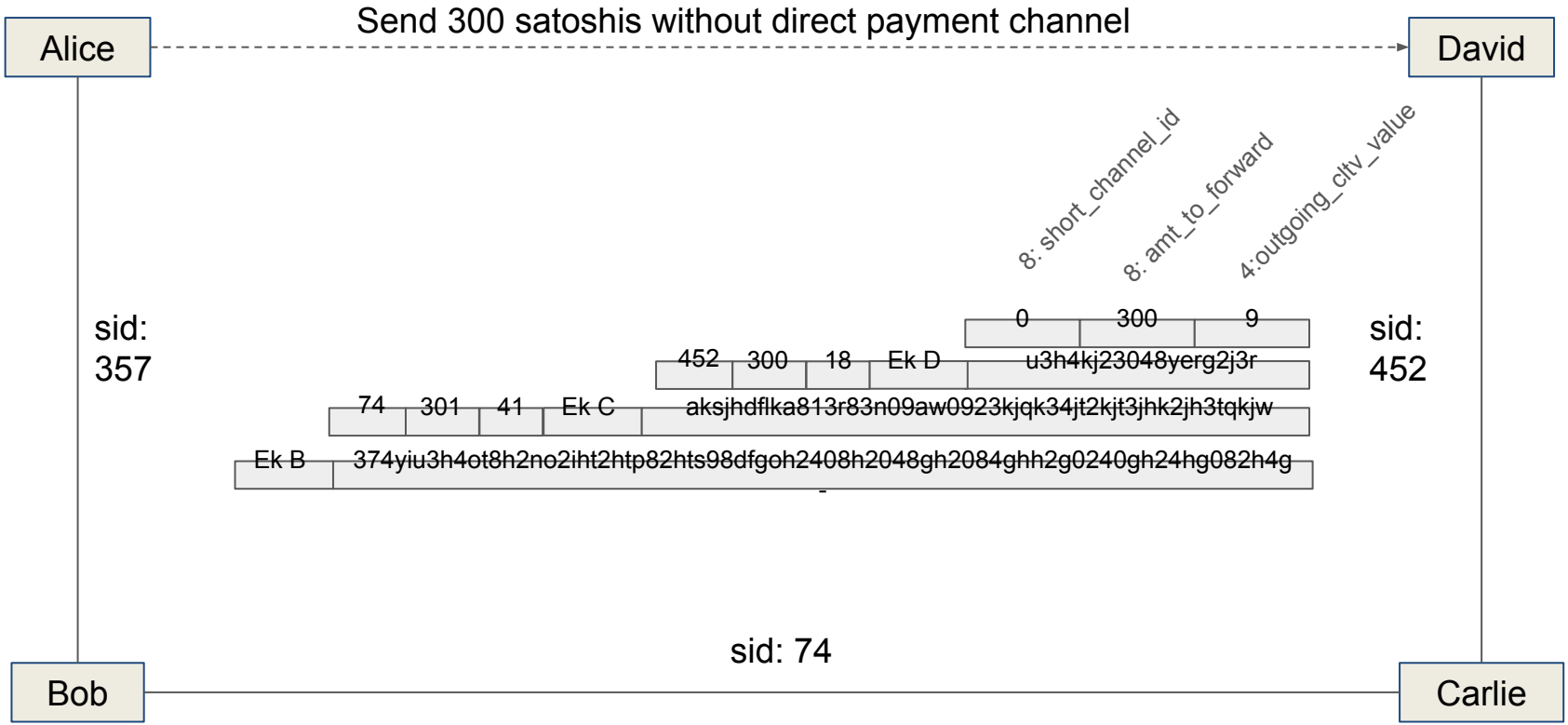
# Onion for Charly (without HMAC and filler)



# Payload for Bob (simplified)



# Onion for Bob without HMAC and filler



# Some notes on the presented simplifications

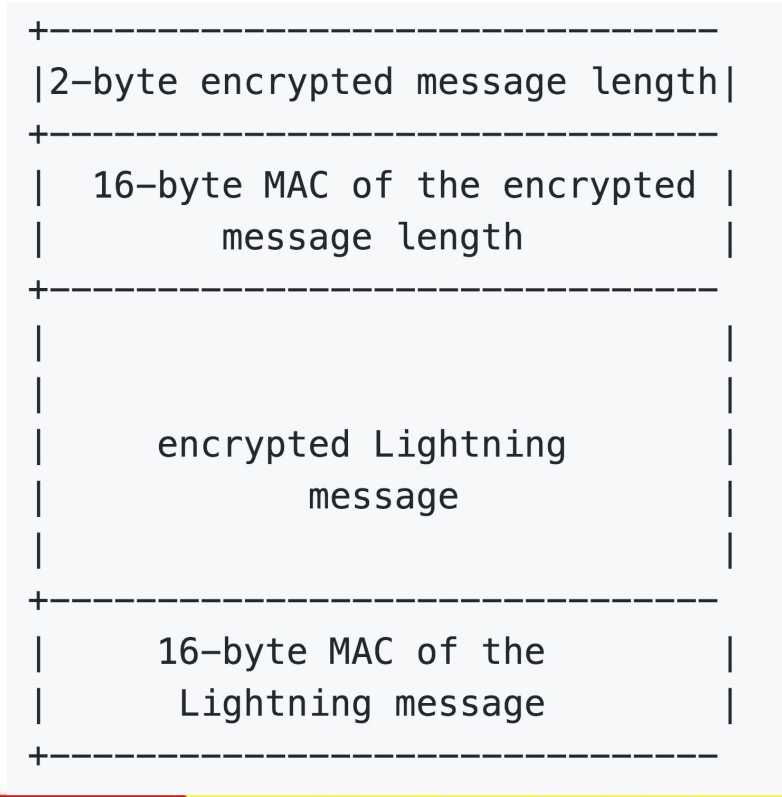
- The onions themselfs are the payload of the update_add_htlc message
  - Described later in the peer protocol
- The message contains the payment hash
- The message offers an htlc with an actual amount
  - Usually nodes offer the amount that they are supposed to forward
- Alice constructs the onion and uses ephemeral keys for every hop
  - Onions are encrypted with a Diffie-Helmann shared secret between
    - Hops ephemeral key (generated by Alice)
    - Hops node_id (static key)
- Onions are always 1366 bytes in length
  - Even if it is the last hop
  - The onions are padded with junk data
    - Padding process left out but described in BOLT 04
  - This prevents a routing node to guess its position in the route by the length of the onion

# Transport Layer - The Noise Protocol Framework

(Chapter 5 / BOLT 08)

# Communication is encrypted and authenticated

- Communication is done by messages between peers
- Messages are sent in a session
  - Encrypted
  - And authenticated with a 16 Byte MAC
- Messages are prepended with a header
  - The header contains the message length
  - The header is encrypted
  - The header is also authenticated with a MAC
- MAC makes messages safe against malicious interference
- Encryption makes messages private



# Lightning uses the Noise Protocol Framework

- 2 phases
  - Handshake phase
    - Exchange of DH public keys, hashing, creating a session key
  - Transport phase
    - Shared key will be used to send encrypted messages
- Noise Protocol Framework can create 12 fundamental handshake patterns
  - Sender
    - N - No static key for initiator
    - K - static for initiator known to responder
    - X - Static key for initiator xmitted (transmitted) to responder
    - I - Static key for initiator immediately transmitted to responder
  - Responder
    - N - No static key for Transponder
    - K - Static key for responder known to initiator
    - X - Static key for responder xmitted to initiator

# Lightning Network uses the Noise XK pattern

- X - Static key for initiator xmitted (transmitted) to responder
- K - Static key for responder known to initiator

XK:

< -- s

…

--> e, es (act 1: send ephemeral key to responder)

<-- e, ee (act 2 send responders ephemeral key to sender)

--> s, se (act 3 authenticate sender)

# Payload security properties achieved by XK

- **Sender authentication *resistant* to key-compromise impersonation (KCI)**.
  - The sender authentication is based on an ephemeral-static DH (`"es"` or `"se"`) between the
    - sender's static key pair and
    - the recipient's ephemeral key pair.
  - Assuming the corresponding private keys are secure, this authentication cannot be forged.
- **Encryption to a known recipient, strong forward secrecy.**
  - This payload is encrypted based on
    - an ephemeral-ephemeral DH as well as
    - an ephemeral-static DH with the recipient's static key pair.
  - This payload cannot be decrypted assuming
    - the ephemeral private keys are secure,
    - and the recipient is not being actively impersonated by an attacker that has stolen its static private key

# Identity hiding properties achieved by XK

- **Initiator**
  - Encrypted with forward secrecy to an authenticated party
- **Responder**
  - Not transmitted,
  - Passive attacker can check candidates for the responder's private key
    - and determine whether the candidate is correct.
  - An attacker could also replay a previously-recorded message to a new responder
    - determine whether the two responders are the "same"
      - (i.e. are using the same static key pair)
    - Does the recipient accept the replay message?
- **Warning! Identities may be exposed through other means including:**
  - Payload fields
  - Traffic analysis
  - Metadata such as IP addresses

# Key setup for the Transport layer

- Each Lightning Network node is identified by a static node_id (pubkey)
  - Curvepoint on secp256k1 curve which is also used in bitcoin
  - Private key is derived from a mnemonic seed
- The Transport layer uses ephemeral session keys before actual data transfer
  - Authenticated key agreement handshake
  - Based on the Noise Protocol Framework
    - Whatsapp, WireGuard, I2P
    - http://noiseprotocol.org/noise.html
- Message exchange phase
  - Authenticated encryption with associated data (AEAD) cyphertexts
  - Key rotation after a key is used 1000 times
    - I.e. every 500 messages
    - Each messages goes through 2 separate but identical encryption processes

# Authenticated Key Agreement Handshake

```
Noise_XK(s, rs):
   <- s
   ...
   -> e, es
   <- e, ee
   -> s, se
```

- Following the noise framework e and s are public keys
  - s is the node_id (also called the static key)
  - e is an ephemeral key used to for setting up the cryptographic session
- ee, es, se, ss are a notation for a Diffie Hellman Key exchange in which
  - The first letter is the key from the local side
  - The second letter is the key from the remote side
  - Attention!
    - Can be confusing when sending messages from responder to initiator

# Processing of handshake data

- Handshake data including keying materials is hashed into a session-wide "handshake-digest" h
- h is never transmitted during the handshake
- h is used as a MAC in the AEAD messages
- If a MAC check fails during the handshake process the connection is immediately terminated
- Privacy Risk:
  - Man in the middle could disturb operations by making handshakes fail all the time
  - User might be tricked to upgrade the node
    - Potentially like the electrum phishing hack from a malicious source

# Handshake State

- Each side maintains these variables
  - Chaining key: ck
  - Handshake key: h
  - Ephemeral key: e
  - Static keypair: s (ls for local, rs for remote) usually the node_id is taken
- Functions to be used
  - ECDH(k,rk) with k is a privatekey and rk is a curvepoint.
    - Elliptic curve Diffie Hellman
    - Return is the SHA256 of the DER-compressed format of the generated point
  - HKDF(salt, ikm)
    - HMAC-based Extract-and-Expand Key Derivation Function (HKDF)
    - https://tools.ietf.org/html/rfc5869
  - ChaCha20-Poly1305 (IETF Variant)
    - encrypt(k, n, ad, plaintext)
    - decrypt(k,n, ad, ciphertext)

# Noise protocol instantiation

- Noise Protocol Framework needs three cryptographic functions
  - Hashfunction: SHA-256
  - Elliptic curve: secp256k1
  - AEAD: ChaChaPoly-1305 as the composion of ChaCha20 and Poly1305
    - Specified in https://tools.ietf.org/html/rfc7539
- Official Name:
  - Noise_XK_secp256k1_ChaChaPoly_SHA256
- Hash of official name is hashed to a digest to initialize the starting handshake state:
  - h = 0x2640f52eebcd9e882958951c794250eedb28002c05d7dc2ea0f195406042caf1
- ck = h
- h = sha( h || "lightning" )

# Noise protocol setup: Act 1  --> e, es (50 bytes)

- e = genKey()
- h = SHA-256(h || e.pub.serialize())
- es = ECDH(e.priv, rs)
- ck,tmp_k1 = HKDF(ck, es)
- c = encrypt(tmp_k1,0,h,"")
- h = SHA-256(h || c)

Send m =0 || e.pub.serialize() || c

# Noise protocol setup: Act 1 --> e, es (50 bytes)

- e = genKey()
- h = SHA-256(h || e.pub.serialize())
- es = ECDH(e.priv, rs)
- ck,tmp_k1 = HKDF(ck, es)
- c = encrypt(tmp_k1,0,h,"")
- h = SHA-256(h || c)

Send m =0 || e.pub.serialize() || c

- v, re, c = read(m)
- h = SHA-256(h || re.serialize())
- es = ECDH(s.priv, re)
- ck, tmp_k1 = HKDF(ck, es)
- p = decrypt(tmp_k1, 0, h, c)
- h = SHA-256(h || c)

Important! MAC check in decrypt. This allows us to know the authenticity of the ephemeral key e.

# Remember senders e and receivers re are the same

- e = genKey()
- h = SHA-256(h || e.pub.serialize())
- es = ECDH(e.priv, rs)
- ck,tmp_k1 = HKDF(ck, es)
- c = encrypt(tmp_k1,0,h,"")
- h = SHA-256(h || c)

Send m =0 || e.pub.serialize() || c

- v, re, c = read(m)
- h = SHA-256(h || re.serialize())
- es = ECDH(s.priv, re)
- ck, tmp_k1 = HKDF(ck, es)
- p = decrypt(tmp_k1, 0, h, c)
- h = SHA-256(h || c)

Important! MAC check in decrypt. This allows us to know the authenticity of the ephemeral key e.

# That is exactly the DH-Key exchange

- e = genKey()
- h = SHA-256(h || e.pub.serialize())
- es = ECDH(e.priv, rs)
- ck,tmp_k1 = HKDF(ck, es)
- c = encrypt(tmp_k1,0,h,"")
- h = SHA-256(h || c)

Send m =0 || e.pub.serialize() || c

- v, re, c = read(m)
- h = SHA-256(h || re.serialize())
- es = ECDH(s.priv, re)
- ck, tmp_k1 = HKDF(ck, es)
- p = decrypt(tmp_k1, 0, h, c)
- h = SHA-256(h || c)

Important! MAC check in decrypt. This allows us to know the authenticity of the ephemeral key e.

# Noise protocol setup: Act 2  <-- e, ee (50 bytes)

- e = genKey()
  - H = SHA-256(h || e.pub.serialize())
- ee = ECDH(e.priv, re)
- ck,tmp_k2 = HKDF(ck, ee)
- c = encrypt(tmp_k2,0,h,"")
- h = SHA-256(h || c)

Send m =0 || e.pub.serialize() || c

# Noise protocol setup: Act 2  <-- e, ee (50 bytes)

- v, re, c = read(m)
- h = SHA-256(h || re.serialize())
- ee = ECDH(e.priv, re)
- ck, tmp_k2 = HKDF(ck, ee)
- p = decrypt(tmp_k2, 0, h, c)
- h = SHA-256(h || c)

Important! MAC check in decrypt. This allows us to know the authenticity of the ephemeral key of the responder e.

- e = genKey()
  - H = SHA-256(h || e.pub.serialize())
- ee = ECDH(e.priv, re)
- ck,tmp_k2 = HKDF(ck, ee)
- c = encrypt(tmp_k2,0,h,"")
- h = SHA-256(h || c)

Send m =0 || e.pub.serialize() || c

# Pay attention to the notation of e, re and ee

- v, re, c = read(m)
- h = SHA-256(h || re.serialize())
- ee = ECDH(e.priv, re)
- ck, tmp_k2 = HKDF(ck, ee)
- p = decrypt(tmp_k2, 0, h, c)
- h = SHA-256(h || c)

Important! MAC check in decrypt. This allows us to know the authenticity of the ephemeral key of the responder e.

- e = genKey()
  - H = SHA-256(h || e.pub.serialize())
- ee = ECDH(e.priv, re)
- ck,tmp_k2 = HKDF(ck, ee)
- c = encrypt(tmp_k2,0,h,"")
- h = SHA-256(h || c)

Send m =0 || e.pub.serialize() || c

# Noise protocol setup: Act 3  --> s, se (66 bytes)

- c = encrypt(tmp_k2, 1, h, s.pub.serialize())
- h = SHA-256(h || c)
- se = ECDH(s.priv, re)
- ck,tmp_k3 = HKDF(ck, se)
- t = encrypt(temp_k3, 0, h, "")
- sk,rk = HKDF(ck, "")
- c = encrypt(tmp_k1, 0, h, "")
- rn = 0, sn = 0

Send m =0 || c || t

# Noise protocol setup: Act 3 --> s, se (66 bytes)

- c = encrypt(tmp_k2, 1, h, s.pub.serialize())
- h = SHA-256(h || c)
- se = ECDH(s.priv, re)
- ck,tmp_k3 = HKDF(ck, se)
- t = encrypt(temp_k3, 0, h, "")
- sk,rk = HKDF(ck, "")
- c = encrypt(tmp_k1, 0, h, "")
- rn = 0, sn = 0

Send m =0 || c || t

- v, c, t = read(m)
- rs = decrypt(tmp_k2, 1, h, c)
- h = SHA-256(h || c)
- se = ECDH(e.priv, rs)
- ck, tmp_k3 = HKDF(ck, se)
- p = decrypt(tmp_k3, 0, h, t)
- rk, sk = HKDF(ck, se)
- rn = 0, sn = 0

# Encrypting Messages

- Message m with sendingkey sk and nonce sn
- l = len(m)
- Serialize l into 2 byte big-endian integer
- Encrypt l (using ChaChaPoly-1305, sn, sk) to obtain HMAC lc
  - sn = sn + 1
- Encrypt m in the same way to receive ciphertext c
  - sn = sn + 1
- Send lc || c

# Decrypting Messages

- Read exactly 18 bytes header as lc
- Decrypt lc (using ChaCha20-Poly1305, rn, rk)
  - Obtain size l
  - Check HMAC
  - Increment nonce rn
- Read exactly 16 + l bytes as c
- Decrypt c (using ChaCha20-Poly1305, rn, rk) to obtain plaintext p
  - Increase nonce rn
-

# Lightning Message Key Rotation

- Keys should be rotated regular basis
- Every message uses a key twice
- Keys are rotated after being used 1000 times
  - So every 500 messages
- k is the sk or rk
  - ck from the end of ACT 3
  - ck', k' = HKDF(ck, k)
  - Reset the nonce for k to n = 0
  - k = k'
  - ck = ck'
- Using ping and pong messages helps rotating keys more quickly

# Messaging / Peer Protocol

(Chapter 6 / BOLT 01 / BOLT 02)

# Purpose of the Peer Protocol

- Establish communication between two peers that are connected
- Assumes a working authenticated and encrypted  Transport Layer
  - (on top of a single TCP socket)
- How will two nodes be able to maintain a RSMC (and HTLCs)?
- How will onion packages be sent between peers?
- Channel Management and operation of the channel
  - Channel establishment
  - Channel close
  - Normal Operation
    - Sending, accepting and forwarding payments
    - Updating fees
  - Message Retransmission (as communication transports are unreliable)

# Format of Lightning messages

- 2 byte type field (big endian)
- Variable length payload
  - Max: 65535 Byte
- Format of the payload confirms to the type
- Even typed messages MUST be specified and MUST NOT be sent otherwise
- It's ok to be odd
- Logical groups of types
  - 0 - 31 Setup & Control
  - 32-127 Channel establishment and closure
  - 128-255 channel operation including setup and settlement of htlcs
  - 256-511 gossip protocol

# The init message

- First message after setup of the Transport layer
- Resent after reconnection
- MUST be the first lightning message
- Shares information about supported / deprecated features
  - See BOLT 09
- Features allow for upgrading the protocol
  - Local features are relevant for the channel
  - Global features are relevant for the network
- Type 16:
- Data
  - 2: gflen
  - gflen: globalfeatures
  - 2: lflen
  - lflen: localfeatures

# The error message

- Usually not exposed to userland
- Type 17:
- Data
  - 32: channel_id
    - Before funding_created it is the temporary_channel_id
  - 2: len
  - len: data
  - lflen: localfeatures

# The ping and pong messages

- Type 18: (ping)
- Data:
  - 2: num_pong_bytes
  - 2: byteslen
  - byteslen: ignored
- Type 19: (pong)
- Data:
  - 2: byteslen
  - byteslen: ignored

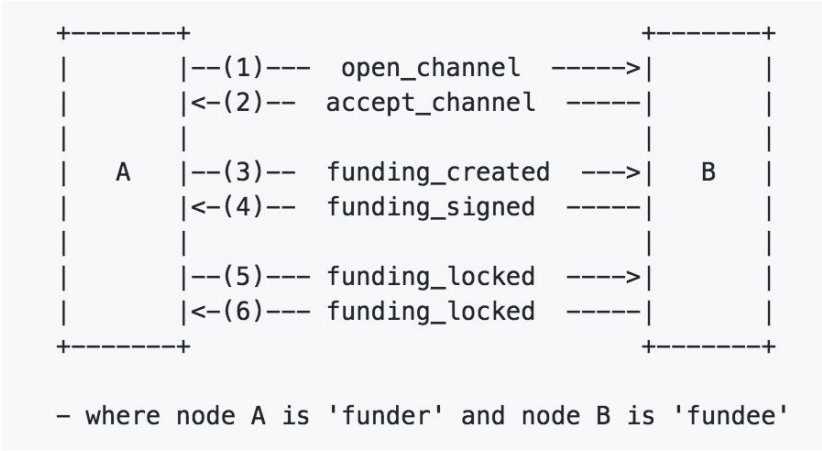
- Nodes MUST NOT use sensitive data in ping & pong messages
- Synthetic traffic to fake activity to defend against packet & timing analysis
- Leads to frequent key rotation of the transport layer (every 500 messages)

# Establishing a channel

(Chapter 6a / BOLT 02)

# Establishing a channel

- Initializing and authenticating a peer connection
- 5 messages
  - open_channel
    - Suggests a channel with certain parameters
  - accept_channel
    - Aggrees to the channel
  - funding_created
    - Sends its signature for first commitment tx
  - funding_signed
    - Sends 2nd sig for commitment tx
  - funding_locked
    - Sends next_per_commitment_point (if funding tx has enough confirmations)



# The open_channel message (type 32) ---->

- **32: chain_hash**
- 32: temporary_channel_id
- 8: **funding_satoshis**
- **8: push_msat**
- 8: dust_limit_satoshis
- 8: max_htlc_value_in_flight_msat
- 8: htlc_minimum_msat
- 2: to_self_delay
- 2: max_accepted_htlcs
- 33: funding_pubkey
- Basepoints (as in BOLT 03 for privacy reason with watchtowers)
  - 33: Revocation
  - 33: Payment
  - 33: Delayed_payment
  - 33: Htlc
  - 33: first_per _commitment
- **1: channel_flags**, 2: shutdown_len, shutdown_scriptpubkeyv

# The accept_channel message (type 33)  <----

- 32: temporary_channel_id
- 8: dust_limit_satoshis
- 8: max_htlc_value_in_flight_msat
- 8: channel_reserve_satoshis
- 8: htlc_minimum_msat
- **4: minimum_depth**
- 2: to_self_delay
- 2: max_accepted_htlcs
- 33: funding_pubkey
- Basepoints (as in BOLT 03 for privacy reason with watchtowers)
  - 33: Revocation
  - 33: Payment
  - 33: Delayed_payment
  - 33: Htlc
  - 33: first_per _commitment
- 2: shutdown_len, shutdown_scriptpubkey

# The funding_created message (type 34) ---->

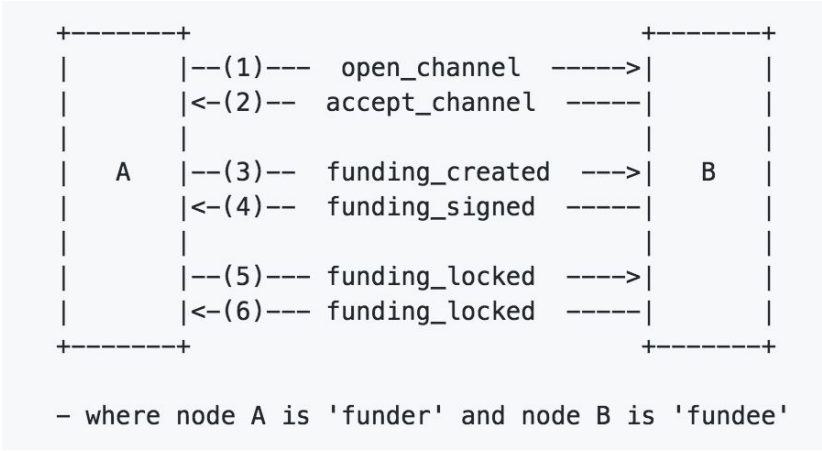
- 32: temporary_channel_id
  - MUST be same as in open channel message
- 32: funding_txid
  - Tx has to be non malleable and MUST NOT be broadcasted at that time
- 2: funding_output_index
- 64: signature
  - Valid signature using the funding_pubkey for the initial commitment tx
  - If incorrect channel MUST be failed

# The funding_signed message  <----

- 32: channel_id
  - Funding_txid ^ funding_output_index
- 64: signature
  - 2nd signature for the first commitment tx
  - If invalid
    - fail channel
    - MUST NOT broadcast the funding_tx
  - If valid SHOULD broadcast the funding_tx

# On Transaction Malleability and the need for segWit

- After signatures for commitment tx are exchanged via funding_signed
  - Funding tx is published by A
  - B can catch this funding tx
- If B malleates the tx it gets a new txid
- Ways to malleate the tx via
  - Signature
  - sigScript
  - https://en.bitcoin.it/wiki/Transaction_malleability
- B can try to publish the malleated version
- Commitment tx refers to txid
- If txid of funding tx is malleated the commitment tx becomes worthless
- B can now blackmail A
  - A cannot access the funds A provided without the help of B
  - Since B did not invest some funds this way for blackmail is "cost free" for B
- Segwit mitigates the possibility to malleate a transaction



# The funding_locked message (type 36) ----> <----

- 32: channel_id
- 33: next_per_commitment_point
  - As defined in BOLT 03 Key derivation
  - Used for next commitment transaction
    - Yes every commitment transaction uses a different basepoint
    - Future base points cannot be derived from old ones
      - This shall improve the privacy
      - Especially when using watching services

- Funding_locked message is necessary for the channel to become operational
  - Sender MUST wait until minimum_depth of funding tx before sending this message
  - Recipient SHOULD forget the channel if message does not come
    - Mitigates DoS attack risks.

# Closing a channel

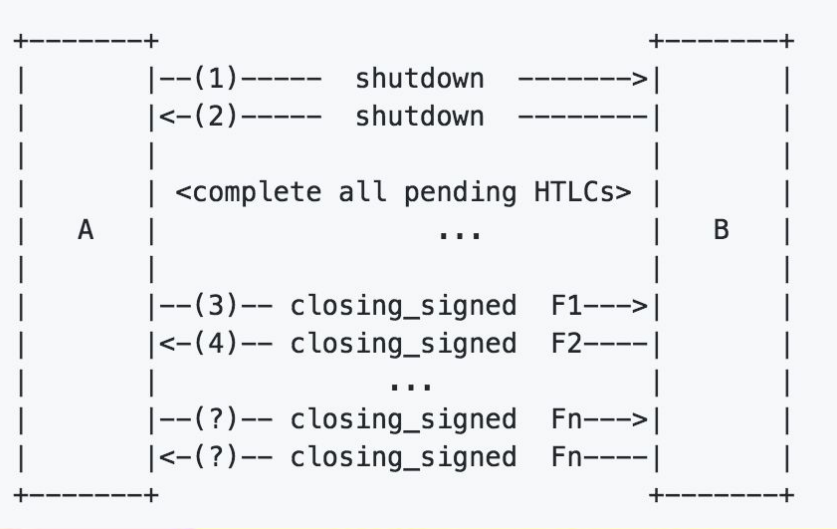
(Chapter 6b / BOLT 02 / BOLT 05)

# 3 ways for closing a channel (BOLT 05)

- The good (Mutual close)
  - 1 closing transaction
  - Similar to commitment tx but no pending payments
  - Part of BOLT 02 - peer protocol
- The bad (unilateral close)
  - The protocol was breached
    - Look out for "... MUST fail the channel" in the BOLTs
    - Could be accidentally e.g. hardware failure
  - One side publishes latest commitment transaction
  - Part of BOLT 05 - onchain
- The ugly (revoked transaction close)
  - One party (deliberately?) tries to cheat
  - Software bugs
  - Part of BOLT 05 - onchain

# The good way - Mutual Close (always preferable)

- Excludes the time lock of the funds
  - Partners can spend the funds directly
- Fee can be negotiated when closing takes place
  - Potentially saves fee
- 2 Messages are included
  - shutdown
    - To signal intend
  - closing_signed
    - Mainly to negotiate fees



# The shutdown message

- Type: 38
- Data
  - 32: channel_id
  - 2: len
  - len: scriptpubkey (one of the following script templates)
    - P2PKH: `OP_DUP OP_HASH160 20` 20-bytes `OP_EQUALVERIFY OP_CHECKSIG`
    - P2SH: `OP_HASH160 20` 20-bytes `OP_EQUAL`
    - P2WPKH: `OP_0 20` 20-bytes
    - P2WSH: `OP_0 32` 32-bytes

- MUST NOT be before funding_created / funding_signed messages
- MAY be done before funding_locked messages
- No future add_update_htlc messages will be accepted
- No future update messages will be accepted

# The closing_signed message

- Type: 39
- Data
  - 32: channel_id
  - 8: fee_satoshis
  - 64: signature

- All htlcs must be settled / cleared in the current Commitment Tx
- If fees of one roundtrip of closing_signed message are equal
  - SHOULD sign and broadcast the closing tx
  - MAY close the connection
- Fee_satoshis should converge
  - MUST propose a value strictly between received fee_satoshis and previously-sent one.
- No real risk for DOS as channel partner could fail the channel

# The bad way - Unilateral / Force Close

- The most recent commitment transaction is pushed to the chain
- Channel stops immediately to be operational
- Almost exact channel state is seen on chain
  - Balance
  - All fully committed htlcs in flight whose outputs are higher than dust
    - Including preimages
- Spend htlcs with either
  - Htlc - success transactions (assuming knowledge of the preimage)
  - Htlc - timeout transactions

```
OP_IF
    # Penalty transaction
    <revocationpubkey>
OP_ELSE
    `to_self_delay`
    OP_CSV
    OP_DROP
    <local_delayedpubkey>
OP_ENDIF
OP_CHECKSIG
```

# The ugly way - Revoked Transaction Close

- An old commitment transaction was published
- Revocation secrets have been shared via revoke_and_ack message
  - We see how in the following 'normal channel operation' part
- **Other side claims all funds within a time lock**
  - This is the penalty mechanism that incentivises both parties to be honest and not go for the ugly way
  - For the mechanism to work there should always be a channel reserve on channels so that there is actually some penalty to collect.
- A watching service can help to do so
- If no claiming with the time lock the breaching party can spend their outputs
- This case should never ocurr

# Normal operation of a channel

(Chapter 6c / BOLT 02)

# 3 messages are necessary to operate a channel

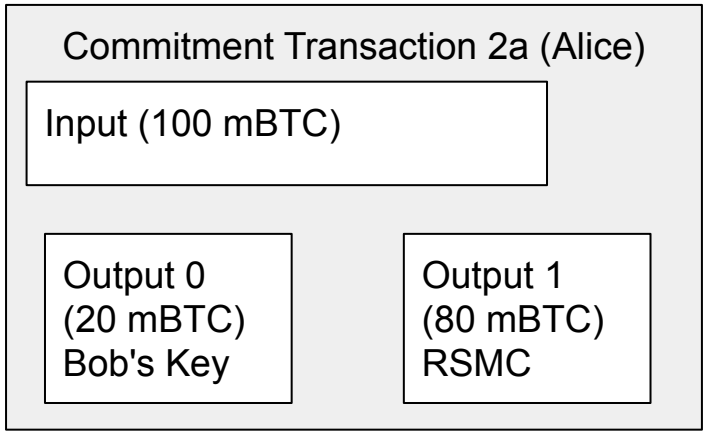
# The 5 stages for a htlc to become valid

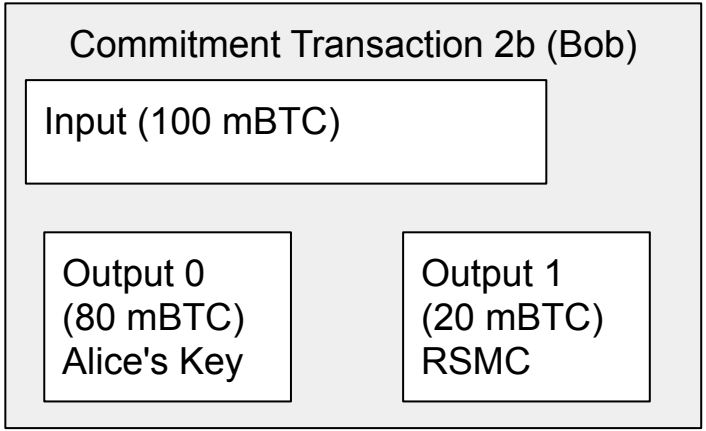
Alice wants to offer a payment to Bob of 15 mBTC

1. Pending on the receiver
2. In the receivers latest commitment tx
3. Receivers old commitment tx is revoked, update is pending at the sender
4. In the senders latest commitment tx
5. Senders old commitment tx is revoked

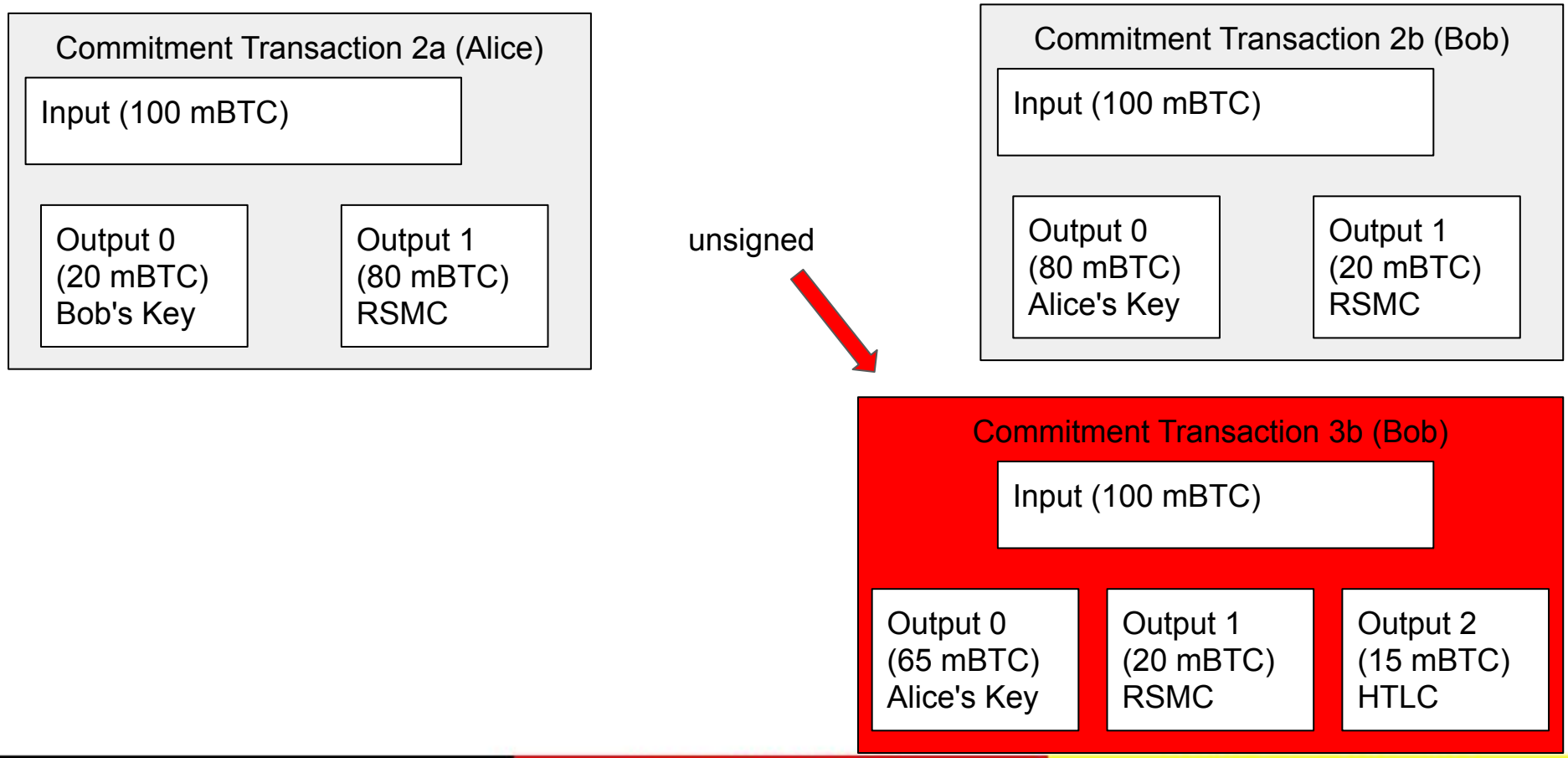
# The 5 stages for a htlc to become valid

- Alice wants to offer a payment to Bob of 15 mBTC

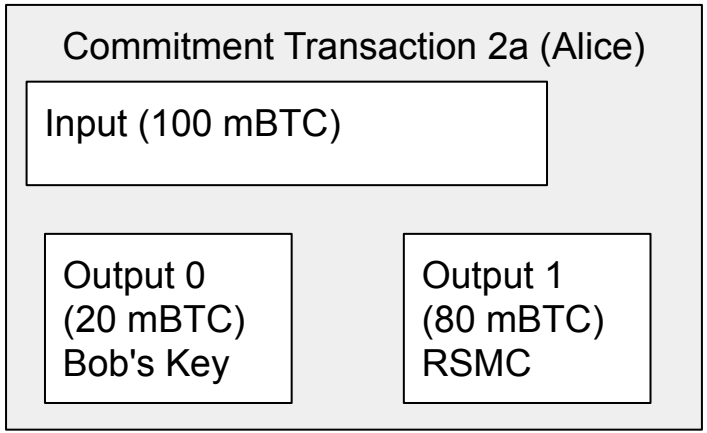


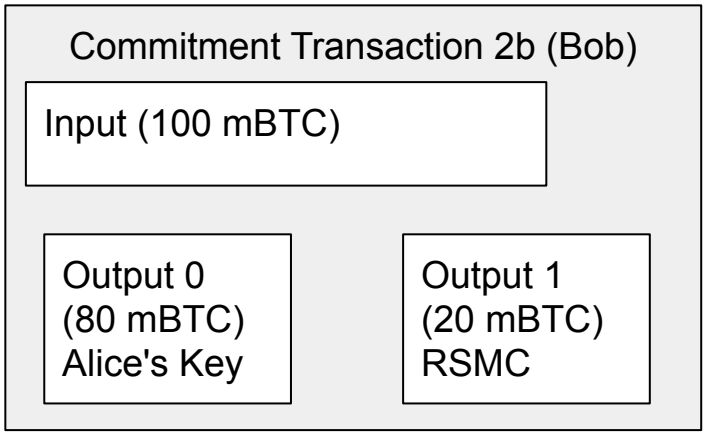



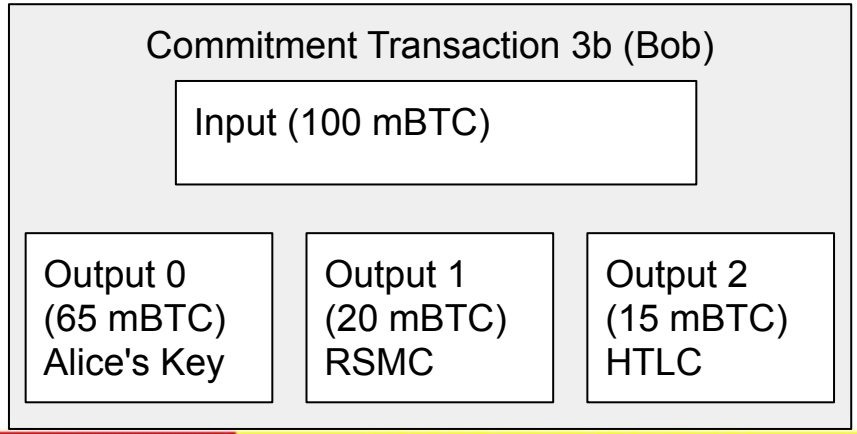
# 1. Alice sends an update_add_htlc message to Bob



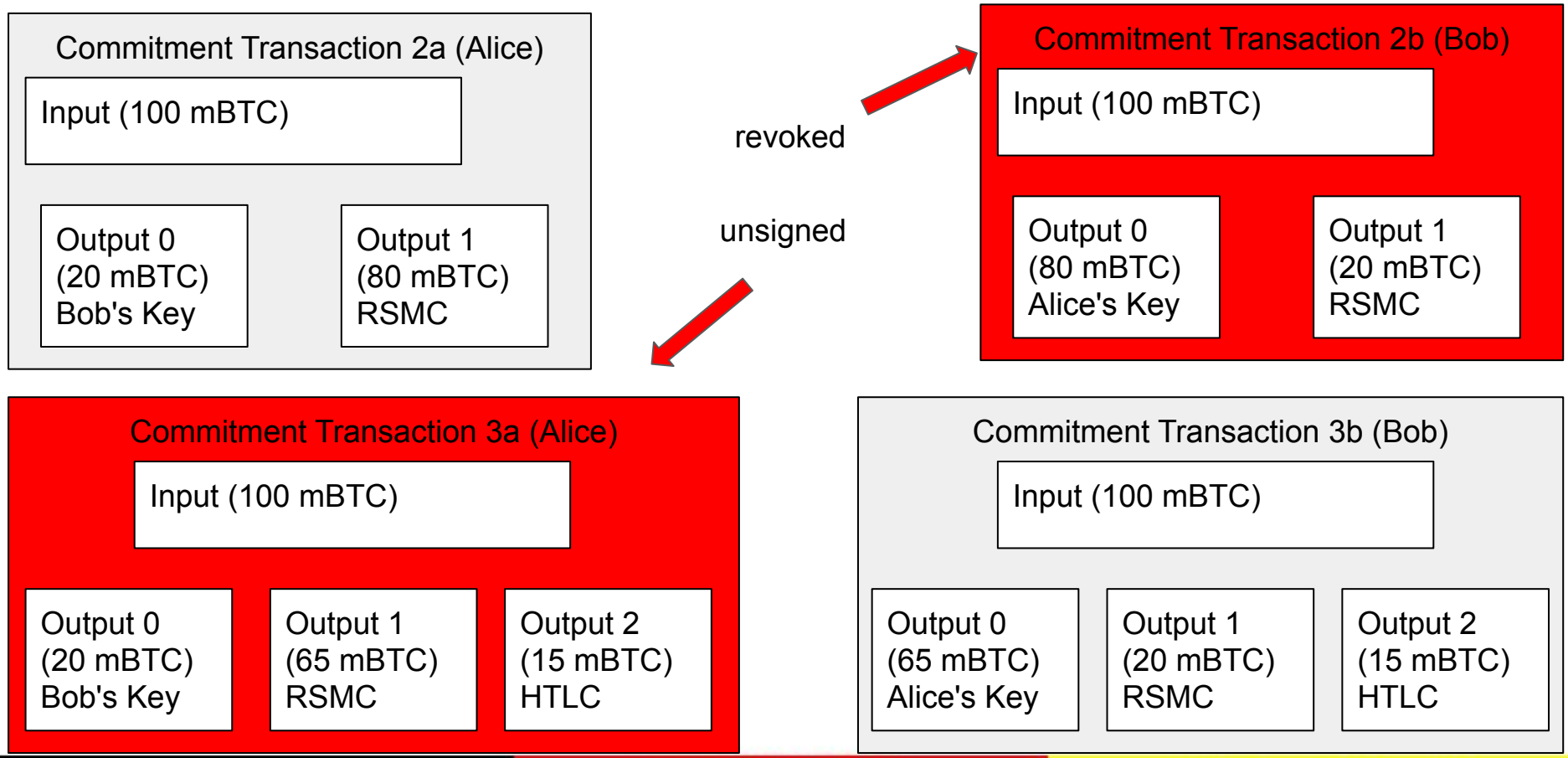
# 2. Alice sends a commitment_signed message to Bob



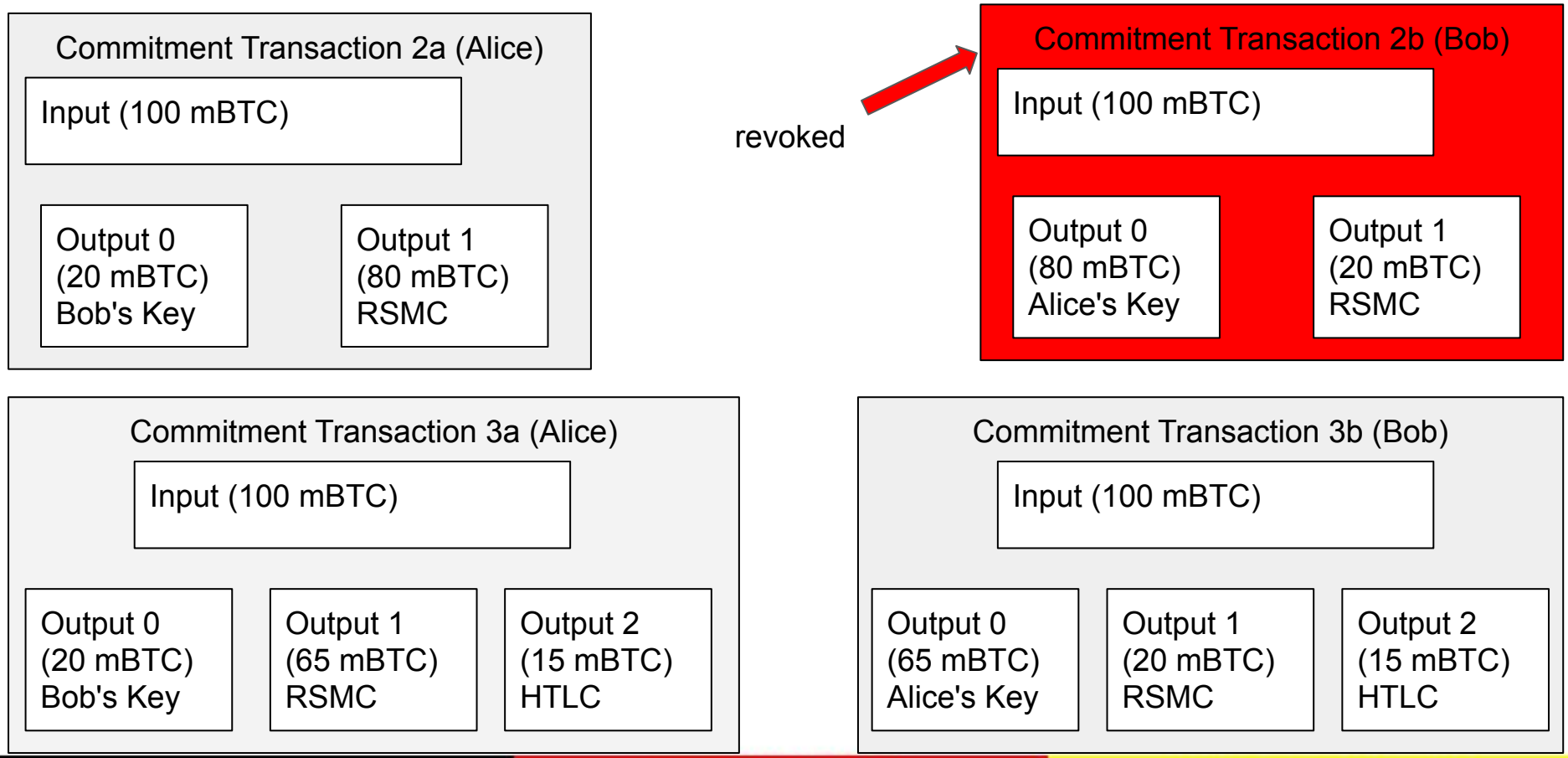



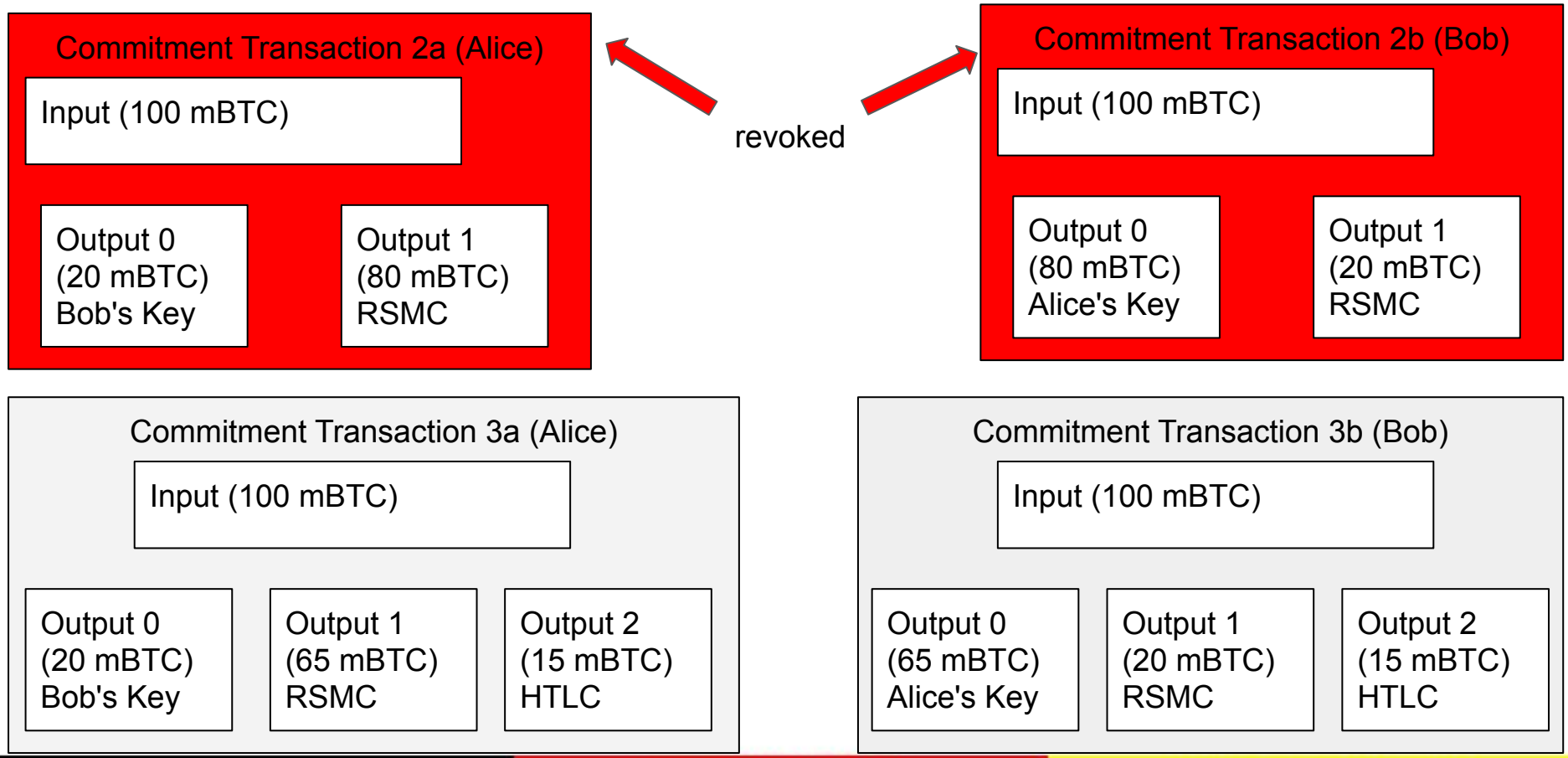
# 3. Bob sends a revoke_and_ack message to Alice



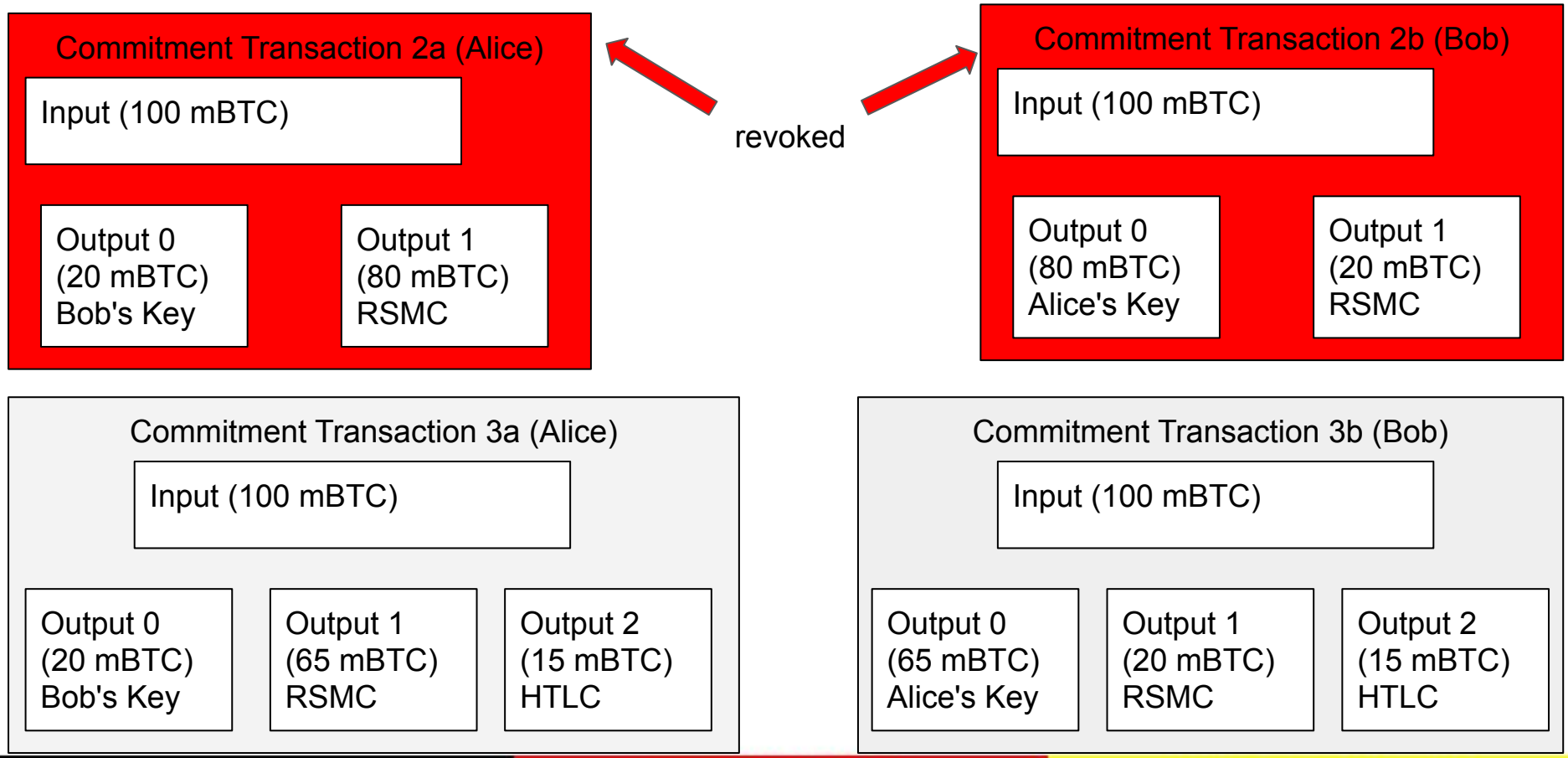
# 4. Bob sends a commitment_signed message to Alice



# 5. Alice sends a revoke_and_ack message to Bob



# Only now Bob SHOULD forward the htlc!



# The update_add_htlc offers a htlc to another node

- Type: 128
- Data
  - 32: channel_id
  - 8: id
    - Htlcs are building a sequence of channel states MUST increment by one
  - 8: amount_msat
    - Amount offered
  - 32: payment_hash
    - As given in the invoice and used in the htlc scripts
  - 4: cltv_expiry
    - Timelock should fit channel parameters and relate to the path for routing
  - 1366: onion_routing_packet
    - Always has this size
    - Is even needed in the case of the last node
    - Contains a session key for DH and HMAC
- Includes no signature (Remember Transport has strong authentication)

# The commitment_signed message

- Type: 132
- Data:
  - 32: channel_id
  - 64: signature
    - For the other side to be used and verified to sign the commitment tx
  - 2: num_htlcs
  - num_htlcs*64: htlc_signatures
    - Each htlcs should have a different payment hash
    - Basepoint changed for each commitment tx
      - Thus signatures change

# The revoke_and_ack message

- Type: 133
- Data
  - 32: channel_id
  - 32: per_commitment_secret
    - MUST generate the current (previous) per_commitment_point
    - SHOULD be generated following the key derivation protocol (BOLT 03)
      - Otherwise node MAY fail channel
      - Compact / efficient key storage
    - Acts as revocation key
  - 32: next_per_commitment_point

# 4 reasons for removing an htlc

- Payment preimage is being supplied
  - update_fulfill _htlc
- Htlc is timeouted
  - update_fail_htlc
- Htlc failed to route (at a later node of the path)
  - update_fail_htlc
- Malformed
  - update_fail_malformed_htlc
- Only a node that received an htlc can remove it
  - For simplicity

# The update_fullfil_htlc message

- Type: 130
- Data:
  - 32: channel_id
  - 8: id
  - 32: payment_preimage

- Obviously the payment_preimage MUST hash to the payment hash
- The next commitment tx can leave this htlc out
  - Including it would only waste space
  - Commitment tx are not explicitly updated at this point

# The update_fail_htlc message

- Type: 131
- Data:
  - 32: channel_id
  - 8: id
  - 2: len
  - Len: reason

- The SPHINX mix format allows for nodes to send back an error to the unknown initiator of the message
- The reason is only be readable by the initiator of the payment
- The initiator can't expect a failure message
  - In intermediate connection could break done before the error message is sent

# The update_fail_malformed_htlc

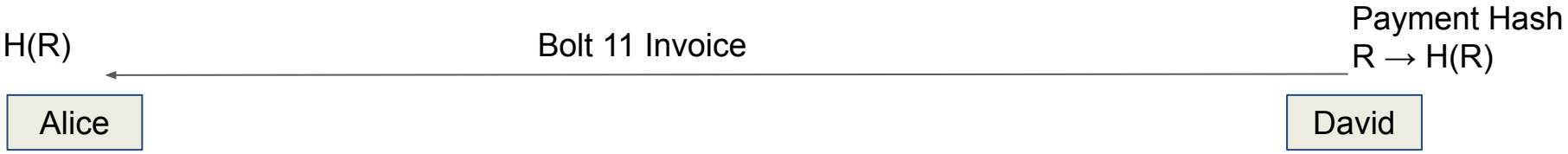
- Sent if the onion is not parsable
- Type: 135
- Data:
  - 32: channel_id
  - 8: id
  - 32: sha256_of_onion
  - 2: failure_code
    - As defined in BOLT 04

# Rational for Retransmission messages

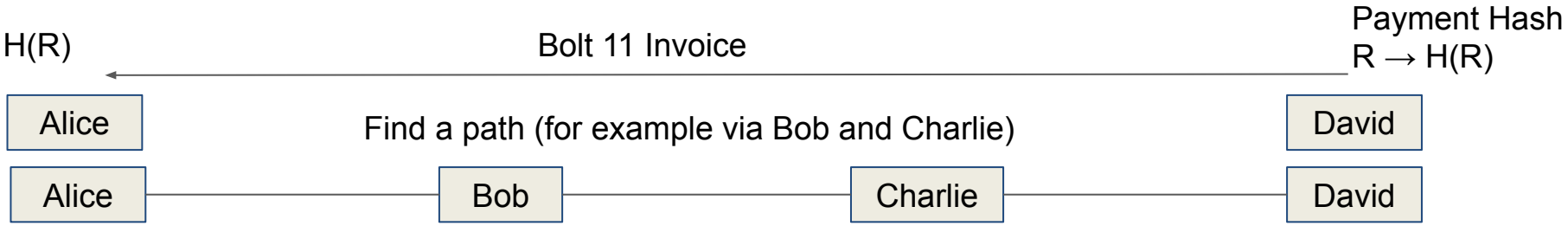
- Communication transports are unreliable
  - Cannot be assumed updates_add_htlc were received unless commitment_signed was received
  - Only store updates upon receipt of commitment_signed
  - This ensures retransmission after channel_reestablishment

- Type: 136 (channel_reestablish)
- Data:
  - 32: channel_id
  - 8: next_local_commitment_number
  - 8: next_remote_revocation_number
  - 32: your_last_per_commitment_secret
  - 32: my_current_per_commitment_point

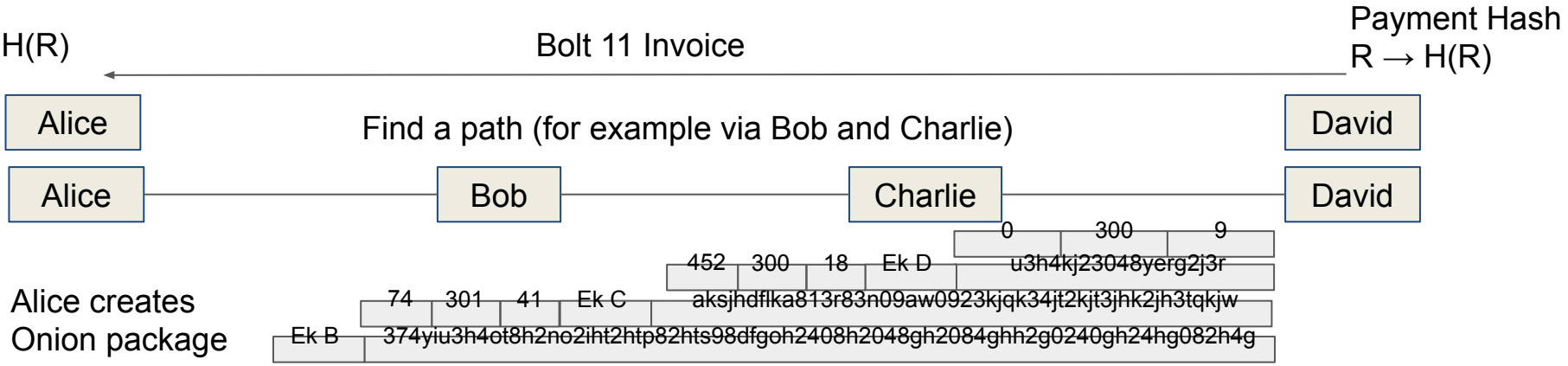
# Workflow of a Payment starts with a BOLT 11 invoice



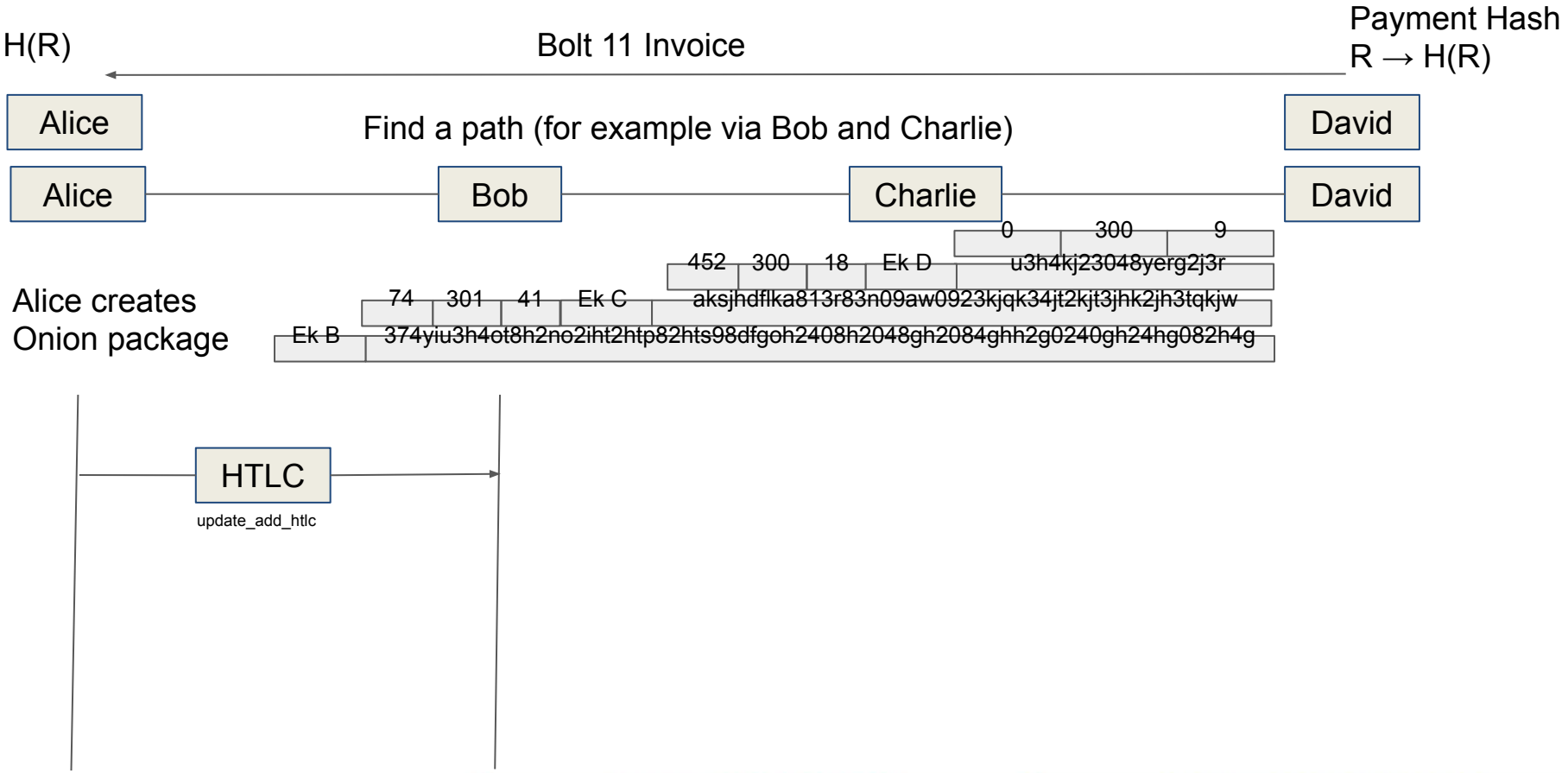
# After receiving the invoice Alice selects a path to David



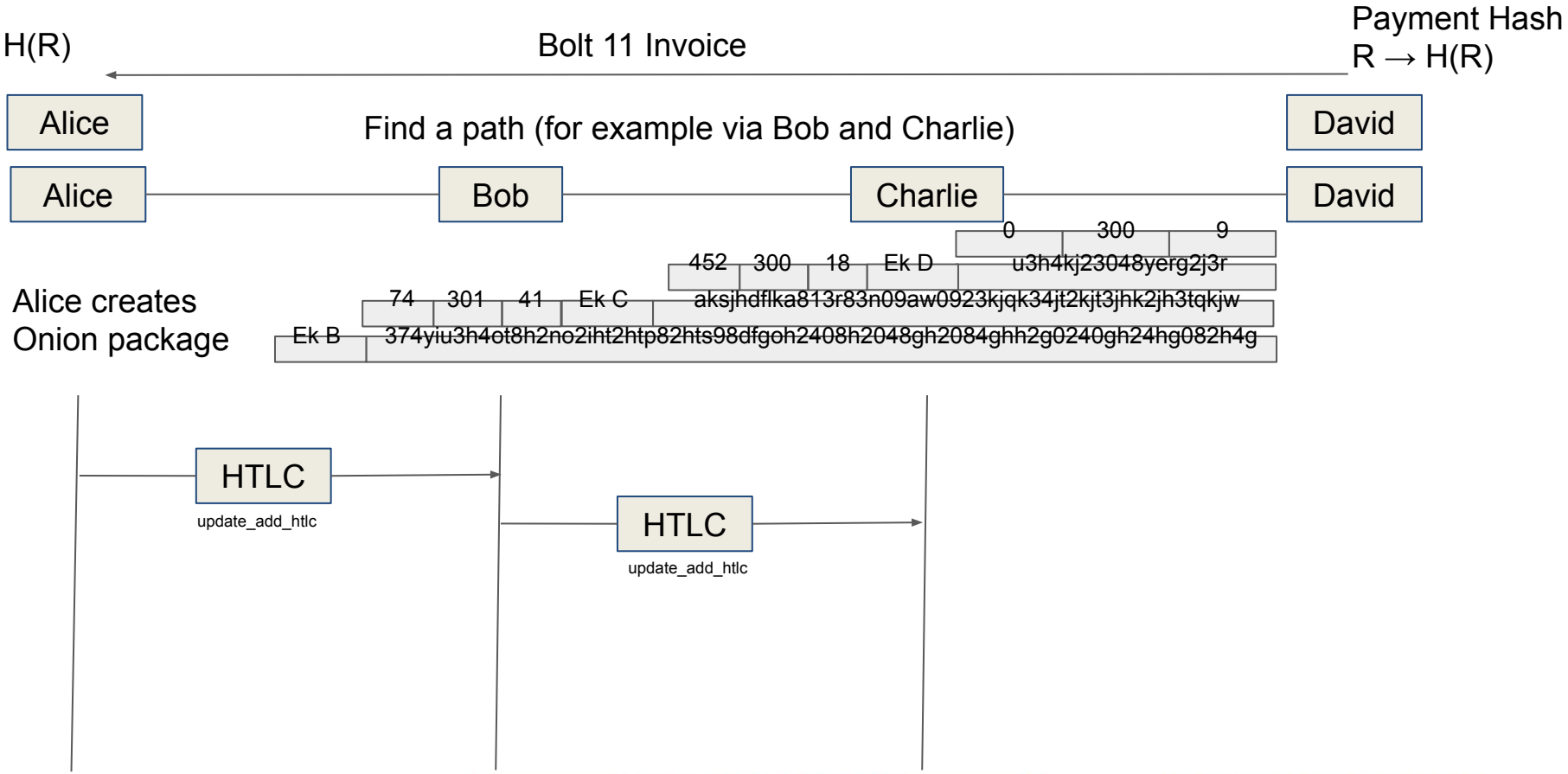
# Alice creates the Onion package (starting from David)



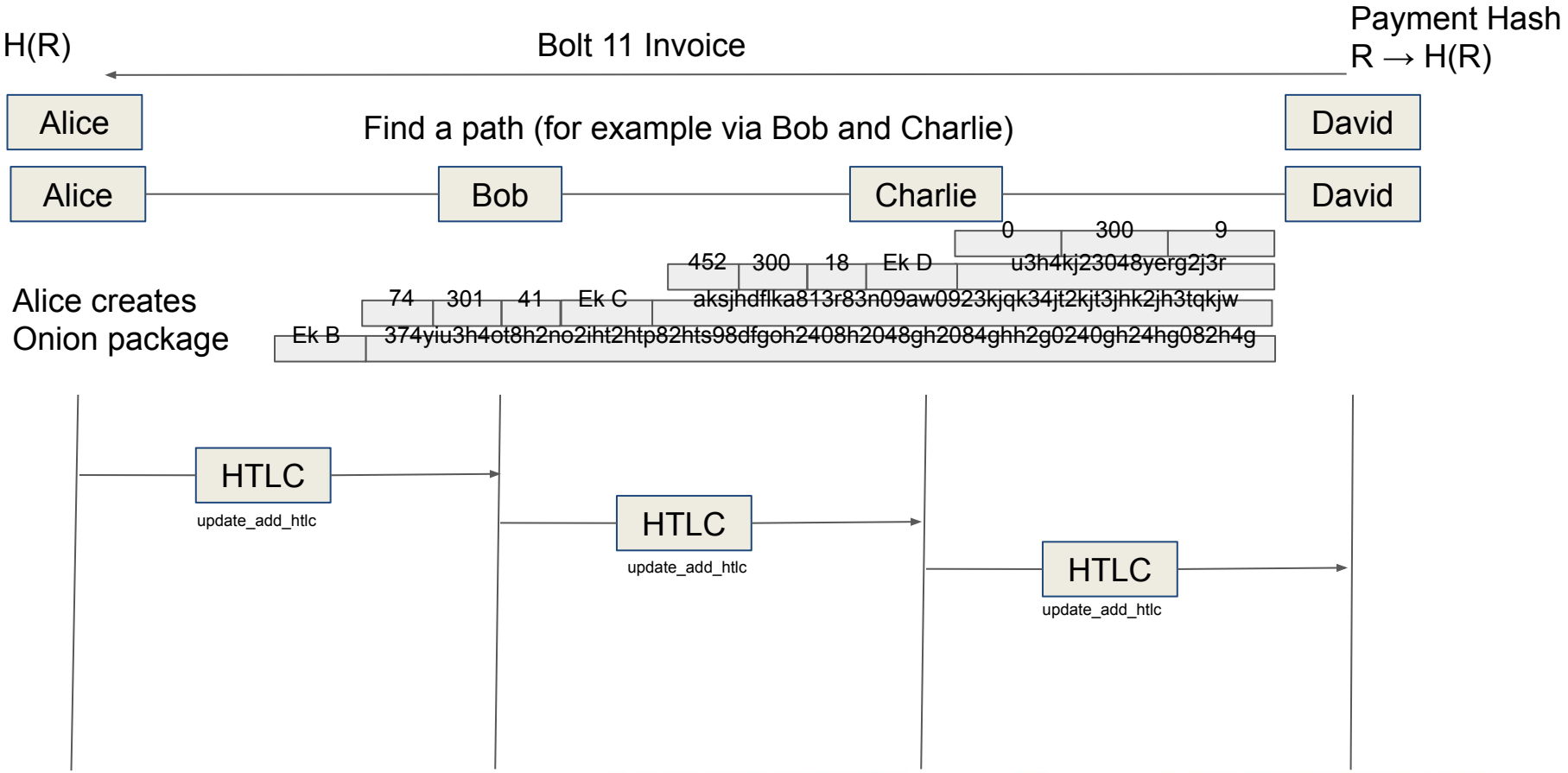
# Alice offers the first htlc to Bob



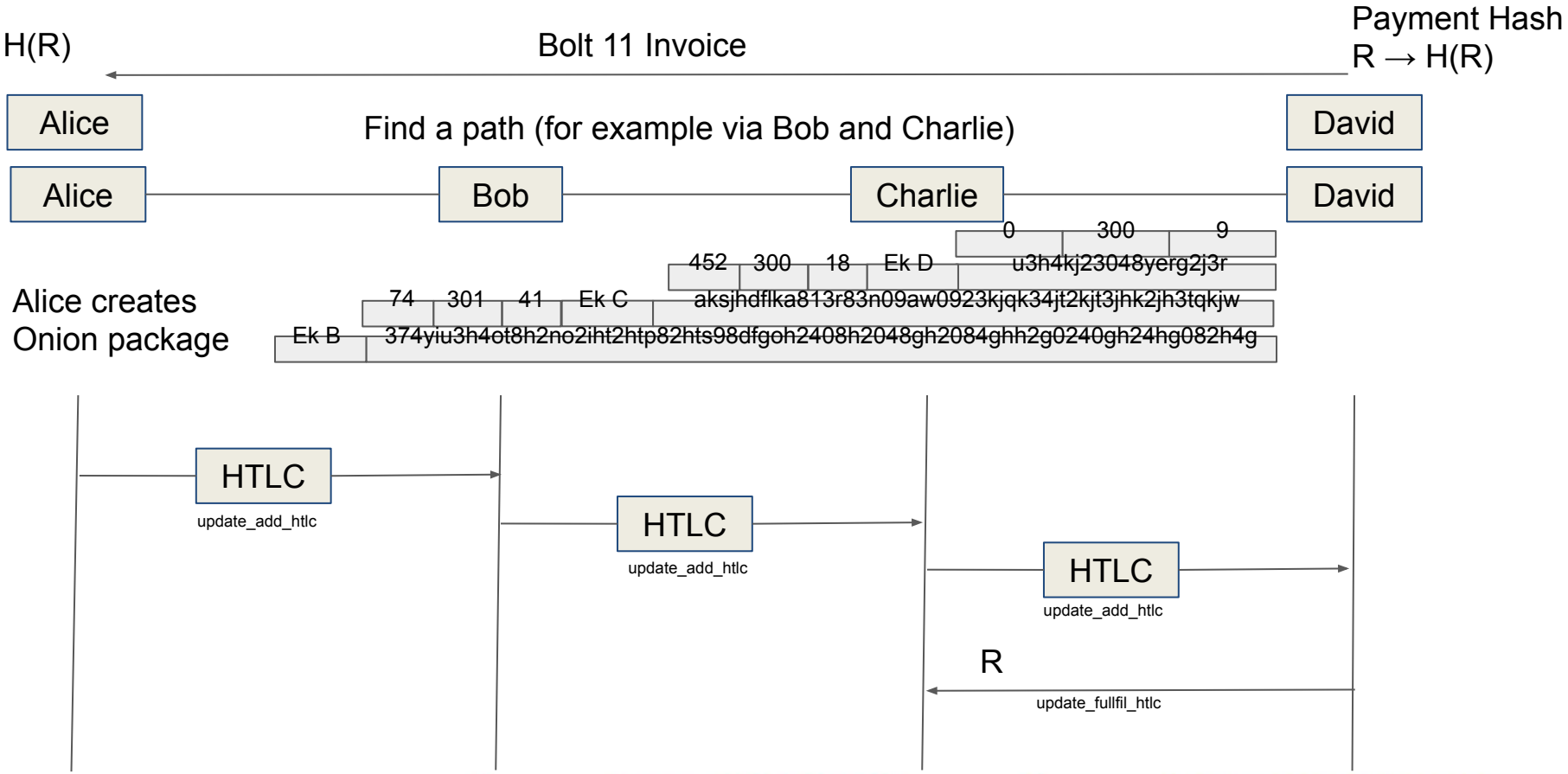
# Bob processes the onion and offers an htlc to Charlie



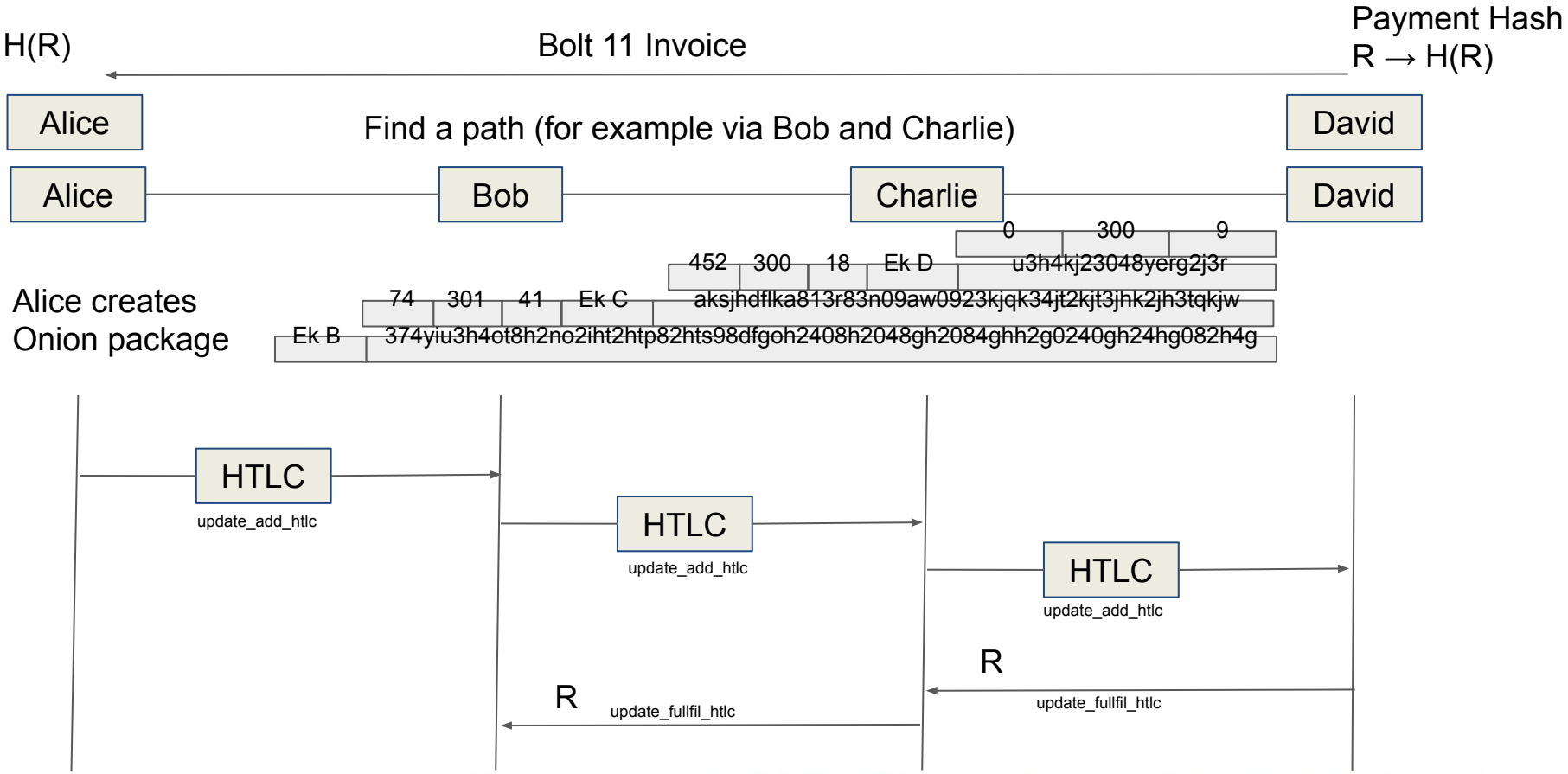
# Charlie processes the onion and offers htlc to David



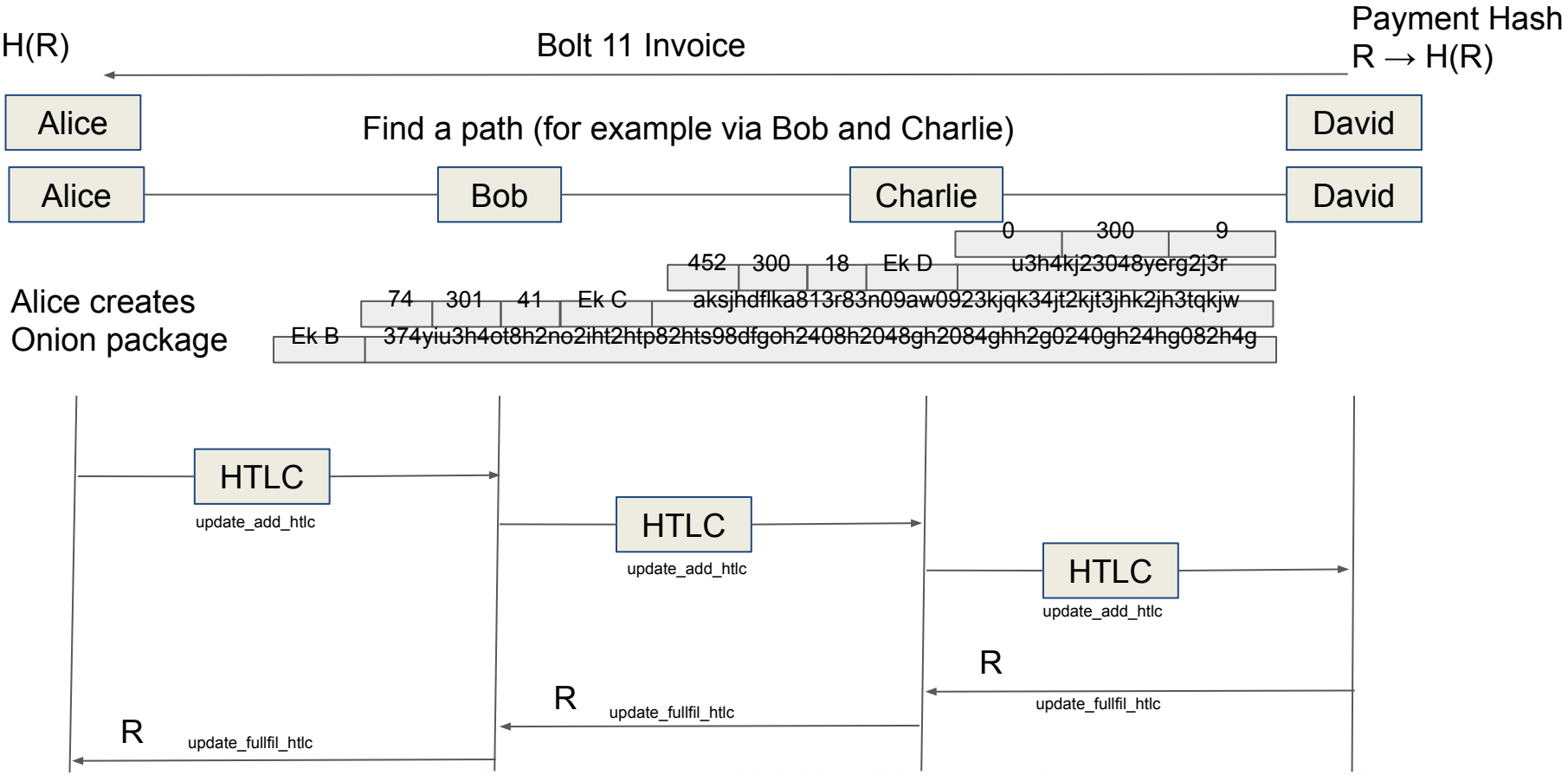
# David checks the amount an releases the preimage



# Upon receipt of preimage charly claims funds from Bob



# Workflow of a Payment on the Lightning Network



# What can be seen by channel closes on chain?

- Type of close
- Channel state
- Pending htlc outputs
  - If not below dust limit
  - There might be a fight about htlc sizes round the dust limit
    - Nodes have an interest to be larger than the dust limit
    - Privacy aware people might want to be below the dust limit

# The Gossip Protocol

(Chapter 7 / BOLT 07)

# Purpose of the Gossip Protocol

- Routing is source based
- Nodes need to be aware of the topology of the lightning network
  - Which nodes exist?
    - How to connect to them?
    - How to open channels with them?
    - What are channel policies?
  - Which channels exist?
    - What are the policies of the channels for routing?
      - Fees
      - Htlc_minimum_msat
      - Cltv_expiry_delta
      - Channel flags
    - Capacity
- Share information between peers

# 4 announcement messages

- Announcment_signatures
  - Needed to negotiate if the channel shall become public
- Channel_announcement
  - Makes a channel publicly known to the network
  - Proofs ownership of the channel
- Node_announcement
  - Shares meta data of a node (like its IP address)
  - Only possible if at least one channel was announced
- Channel_update
  - Updates channel policies
  - Not yet relevant to the capacity or owners

# The announcement_signatures message

- Needed to negotiate if the channel shall become public
- Type: 259
- Data:
  - 32: channel_id
  - 8: short_channel_id
  - 64: node_signature
  - 64: bitcoin_signature
- The signatures are used / exchanged to construct a channel_announcement message
  - This ensures that a channel is only announced if both parties agree
  - If shared they link (!!!) the onchain funds and addresses to the node_id

# The channel_announcement message

- Makes a channel publically known to the network
- Proves ownership of the channel
  - Proving funding tx pays to bitcoin_key_1 and bitcoin_key_2
    - Verify that P2WSH funding tx output is of this form
      - 2 <pubkey1><pubkey2> 2 OP_CHECKMULTISIG
    - pubkey1 is nummerically lesser of the DER-encoded funing_pubkeys
  - Proving that node1 owns bitcoin_key_1
  - Proving that node2 owns bitcoin_key_2
- Bitcoin_key ownership is explicitly done with signatures
- Node_signatures verify that the nodes agree on the proofs and message

# The channel_announcment message format

- Type: 256
- Data
  - 64: node_sig_1 (commiting to SHA-256(SHA-256(message[256:])))
  - 64: node_sig_2 (commiting to SHA-256(SHA-256(message[256:])))
  - 64: bitcoin_sig_1
  - 64: bitcoin_sig_2
  - 2: len
  - len: features
    - According to BOLT 09 a mean for backwards compatibility
  - 32: chain_hash
    - 6fe28c0ab6f1b372c1a6a246ae63f74f931e8365e15a089c68d6190000000000 (BTC)
  - 8: short_channel_id
    - Must confirm to the funding tx
  - 33: node_id_1
  - 33: node_id_2
  - 33: bitcoin_key_1
  - 33: bitcoin_key_2

# The node_announcement message

- Shares meta data of a node (like its IP address)
  - Can be spoofed
  - Ipv4 & IPv6 are both supported
  - Can be a tor onion
- Only possible if at least one channel was announced
- Type: 257
- Data
  - 64: signature
  - 2: flen
  - flen: features
  - 4: node_id
  - 33: node_id
  - 3: rgb_color
  - 32: alias
  - 2: addrlen
  - addrlen: addresses

# The channel_update message

- Defines channel policies
  - Mainly for routing (meaning forwarding of htlcs)
- Type: 258
- Data:
  - 64: signature
  - 32: chain_hash
  - 8: short_channel_id
  - 4: timestamp
    - Used for others to decide if the channel can be updated
  - 1: message_flags (option_channel_htlc_max)
  - 1:  channel_flags (includes direction, availability of the channel)
  - 2: cltv_expiry_delta
  - 8: htcl_minimum_msat
  - 4: fee_base_msat
  - 4: fee_proportional_millionths
  - 8: htlc_maximum_msat (static option not designed to leak balance)

# Deleting channels from the network view

- Attempts to close a channel are not broadcasted via gossip
- Also successful closing is not broadcasted
  - Can be seen on the bitcoin network by seeing a spending of the funding tx
- nodes might still try to route htlcs if the mutual shutdown was initiated
- Channel lifetime can be seen on the blockchain
- Initial balance (person bringen funds) is public
- Final balance is public
  - Intermediate payments and utilization of the channel are not seen directly
- Closing type is public
  - Good: Mutual
  - Bad: Unilateral
  - Ugly: Breach

# Querying the gossip protocol for information

- Nodes can retrieve old gossip messages from peers
- Query_schort_channel_ids / reply_short_channel_ids_end
  - Ask for channel_announcment and channel_update messages for a specific channel
  - Usually because seen channel_update before a channel_announcement
  - Or because unknow short_channel_ids from reply_channel_range
- Query_channel_range / reply_channel_range
  - Asks for short channel ids from a starting block up to a certain hight
- Gossip_timestamp_filter
  - Only channel_updates have a timestamp
  - Sends channel_announcements that correspond to channel_updates
- Initial Sync
  - Requested in the init message via feature flag
  - Gossip_queries was not part of the initial protocol so initial_routing_sync must be supported
- Rebroadcasting happens on a basis of newer timestamps
- Pruning the network by monitoring spends of the funding transactions

# Advanced Topics

(Chapter 8)

# Future enhancements to the BOLTs

(Chapter 8a)

# Some advanced topics to talk about

- Routing upgrades
  - Rendezvous
  - AMP
  - JIT
- Channel protocols
  - Eltoo
  - Channel factories
    - Probably done between friends / gives social graph information
- Autopilots
- Watch tower solutions
  - Eased up with eltoo by a lot since only newest update tx is needed
- Scriptless scripts / 2 party ECDSA
- Impact of Schnorr

# Rendezvous Routing

- Payee defines a rendezvous point
- Encrypted onion from rendez vous point to payee is included to invoice
- Payer just extends the onion up to the rendezvous point
- Severe change in protocol
  - New onion format will be needed
  - Private channels will stay hidden
  - Virtual payment channels (not backed by a funding transaction) will become possible
- https://lists.linuxfoundation.org/pipermail/lightning-dev/2018-November/001498.html

# AMP - Routing (Atomic Multipath Routing)

- Payee can split the payment across several paths
  - All below the dust limit if that was needed for privacy
- A special flag in the node announcements signals that a payee supports this
- Routing nodes don't need to change anything
- Payment hash for all paths stays the same
- More nodes will be involved
- Payments could in future have a uniform size
- Atomicity somewhat dropped
  - Lightning is not really a connection oriented protocol
  - HORNET protocol is proposed but on hold
    - https://www.scion-architecture.net/pdf/2015-HORNET.pdf
- https://lists.linuxfoundation.org/pipermail/lightning-dev/2018-November/001577.html

# JIT - Routing (Just in Time Routing)

- Logical equivalent to AMP-Routing
- Way to mitigate disadvantages of source based routing in path finding
- Solve the issue at a node that lacks outgoing or incoming capacity
- Rebalance a channel before processing original htlc
  - Proposal exists to reuse payment hash
- Would mitigate probing attacks as there would probably always be a path of htlcs considering conditional rebalancing
- https://lists.linuxfoundation.org/pipermail/lightning-dev/2019-March/001891.html

# Dual funded channels

- Currently channels are only funded from one node
- Channel establishment protocol needs to be defined
- Game theoretic difficulties with fees / locking btc
- Current proposal allows free of cost probing attack for nodes balance:
  - https://lists.linuxfoundation.org/pipermail/lightning-dev/2019-March/001912.html

# Splicing in & Splicing out funds

- Similar to dual funded channels
- Change channel capacity while maintaining the channel operational
- Currently one splicing operation at a time
  - Until enough confirmations no second splicing operation can be achieved
- Splicing needs at least one on chain transaction and allows for chain analysis
- https://lists.linuxfoundation.org/pipermail/lightning-dev/2018-October/001434.html

# Eltoo channels

- Symmetric channel protocol
  - Way less implementation overhead
  - Suitable for multiparty channel / channel factories
- No penalties involved for protocol breach
- BIP 118 SIGHASH_NOINPUT required
  - Softfork necessary
- Severe change to watching services
  - they only have to store the latest update transaction
  - Update transactions don't have any information about the channel balance
- Heavily criticised by some developers
  - channel breaches are not discouraged without penalties
- https://blockstream.com/eltoo.pdf

# Multiparty channels / channel factories

- Have the funding tx sent to a m-n wallet
- More efficient with Schnorr and Eltoo
- Channel protocol has yet to be developed / specified
- Multiparty channel can create subchannels
  - Private to rest of the multiparty
  - Splicing within the party does not need to be onchain
  - Sub channels can stay operational even if a node in the party becomes unresponsive
- Force close of the multiparty channel will serve as funding tx for subchannels
- https://www.tik.ee.ethz.ch/file/a20a865ce40d40c8f942cf206a7cba96/Scalable_Funding_Of_Blockchain_Micropayment_Networks%20(1).pdf

# Autopilots - Recommendation engines for channels

- Currently Barabasi Albert model
  - Probability distribution proportional to the degree of nodes
- Other features could be taken into consideration
  - Past routing statistics
  - Channel balances
  - The fee graph
- There might be a demand for collaborative autopilots to share information
  - Most likely channel balances
- Will not be part of the protocol in BOLT 1.1
  - Rather an implementation detail

# Watchtower solutions

- Have a third party pay attention for channel breach
- Economic incentive not clear
  - Theoretically a watching services could be forced to store an arbitrary amount of data
- Electrum's lightning implementation is said to come with a subscription based trusted watching service
- Watching service needs revocation secret
- Watching service does not necessarily need to be aware of all channel balances
  - Remember key derivation BOLT 03
- With eltoo channel way more privacy
- Will not be part of BOLT yet
  - https://lists.linuxfoundation.org/pipermail/lightning-dev/2018-November/001562.html

# Scriptless Scripts / 2 party ECDSA

- Create a 2-2 multisig wallet based solely on ECDSA P2PKH
- Makes use of homomorphic Paillier encryption
- Be able to have changing preimages along a route
  - Payment decorrelation
  - Switch to payment points instead of payment hashes
- Atomic swaps as a regular P2PKH on chain transaction
- Will probably not be implemented as Schnorr Signatures can do the same
- https://lists.linuxfoundation.org/pipermail/lightning-dev/2018-May/001244.html

# Schnorr Signatures / MuSig

- Linear signature Scheme
- Signatures can be added up → One verification
- Multisig wallets have just one signature
- Complex scripts / smart contracts can be hidden
- Can be computed from only knowing pubkeys and sigs.
  - Little communication overhead / protocol required
- By now implemented in libsecp256k1
- Desire to push to bitcoin
- Ability to hide complex scripts / contracts
- https://blockstream.com/2019/02/18/musig-a-new-multisignature-standard/

# Chross chain atomic swaps / American Call options

- Potentially atomic swaps are possible
  - Sending an onion through multiple chains
  - Recall hash of genesis block was part of channel announcements
- Atomic swaps on the Lightning Network are in practise probably unusable
  - Atomic swaps together with HTLCs lead to american style call options without fee
  - Pointed out by ZmnSCPxj and CJP
- Only channels can be traced and linked to nodes
  - Rember private nodes exist and are not announced via gossip
- Connects addresses of various chains
  - Via the gossip protocol
- https://lists.linuxfoundation.org/pipermail/lightning-dev/2018-December/001752.html

# Possible DoS Attack Vectors & spamming attacks

- Spamming HTLCs on Lightning
  - A channel can only support 483 htlcs in flight
  - Attacker can create onions without preimages
  - Few satoshis sufficient to do this attack
- Spamming HTLCs to Bitcoin
  - Create many circulare routes through inbound channels
    - Goal to have large commitment transactions
  - Force channel close before releasing the transaction
    - Costs almost nothing for attacker but channel partner (if they funded) the channel pays fees
- Spam the gossip protocol
  - Mainly update messages for fees / channel policies. Comes technically at no cost

# Privacy Considerations

(Chapter 8b)

# The Lightning Network as spying tool?

- The Gossip protocol takes away a great deal of privacy
  - Channel announcements
    - Blockchain transactions that create a payment channel are referenced
    - Node_id's of channel partners are referenced
  - Node announcements
    - IP addresses of nodes are announced
    - Closing channels links blockchain transactions and ownership of coins with the ownership of a server
- Private Nodes / channels can be compromised by BOLT 11 strings
- Probing channels for balance via corrupted onions
- Invoices link payments to IP addresses
- Payment_hash links onions across different hops

https://1ml.com/node/03efccf2c383d7bf340da9a3f02e2c23104a0e4fe8ac1a880c8e2dc92fbdacd9df

1ML

Search

Directory Locations Tools Newsletter Feedback Log in

# Node: ln.rene-pickhardt.de

Follow

## Public Node

**Capacity**

1.06993183 BTC (0.101%)
106,993,183 sat
$4,287.57

**Channel Count**

33 (0.084%)

**Connected Node Count**

33 (0.839%)

**Color**

#03efcc

**IP Addresses**

144.76.235.20:9735

Overview | Channels | Statistics | History | Neighborhood | Monitor | Claim Node

**Public Key: 03efccf2c383d7bf340da9a3f02e2c23104a0e4fe8ac1a880c8e2dc92fbdacd9df**

03efccf2c383d7bf340da9a3f02e2c23104a0e4fe8ac1a880c8e2dc92fbdacd9df@144.76.235.20:9735

https://1ml.com/node/03efccf2c383d7bf340da9a3f02e2c23104a0e4fe8ac1a880c8e2dc92fbdacd9df

1ML

## Node: ln.rene-pickhard

**Public Node**

**Capacity**

1.06993183 BTC (0.101%)
106,993,183 sat
$4,287.57

**Channel Count**

33 (0.084%)

**Connected Node Count**

33 (0.839%)

**Color**

#03efcc

**IP Addresses**

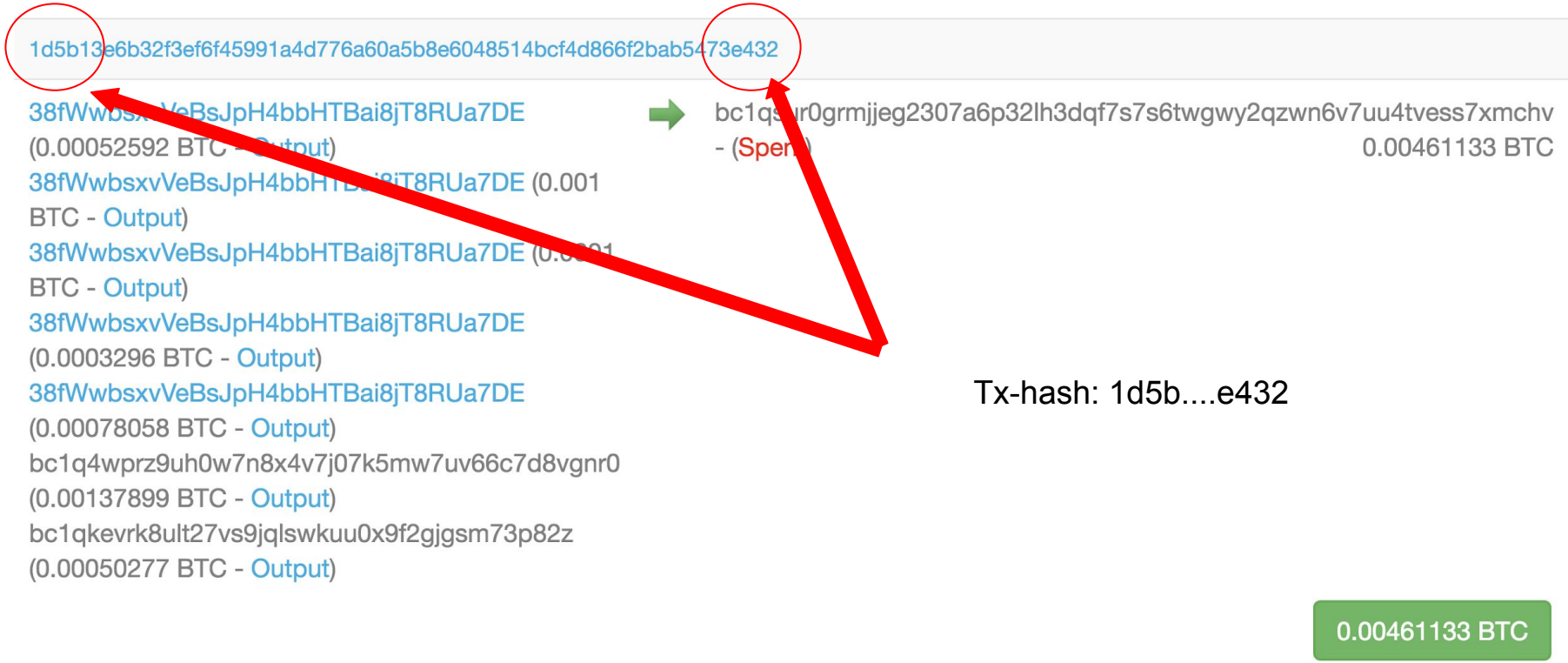
144.76.235.20:9735

https://ln.rene-pickhardt.de

# Rene Pickhardt's c-lightning node

| | |
|---|---|
| **id** | 03efccf2c383d7bf340da9a3f02e2c23104a0e4fe8ac1a880c8e2dc92fbdacd9df |
| **alias** | ln.rene-pickhardt.de |
| **color** | 03efcc |
| **num_peers** | 45 |
| **num_pending_channels** | 4 |
| **num_active_channels** | 41 |
| **num_inactive_channels** | 0 |
| **address** | **type** / **address** / **port**: ipv4 / 144.76.235.20 / 9735 |
| **binding** | **type** / **address** / **port**: ipv4 / 144.76.235.20 / 9735 |
| **version** | v0.7.0-3-g9aaf2fe |
| **blockheight** | 567393 |
| **network** | bitcoin |

# Transaction related to a Lightning network channel?



# Spent by

## Transaction View information about a bitcoin transaction

ceb951e6edb675762b86b2857ae6b7b02fac6a8b2635d019953610ed84237acf

bc1qsur0grmjjeg2307a6p32lh3dqf7s7s6twgwy2qzwn6v7uu4tvess7xmchv (0.00461133 BTC - Output)

bc1q7wmn62hgkq9a66khja3g0esmwlmwk5qj28h3jh - (Spent) 0.00015073 BTC

bc1qjxqz63xs7l3x4c86u3h2qpytxygkfs6gc22r3cr70lnrs4uvqeqsf7qdzg - (Spent) 0.00436192 BTC

0.00451265 BTC

| Summary | |
|---|---|
| Size | 345 (bytes) |
| Weight | 720 |
| Received Time | 2018-11-23 06:21:06 |
| Lock Time | 1987-05-26 20:02:27 |
| Included In Blocks | 551153 ( 2018-11-23 06:47:04 + 26 minutes ) |
| Confirmations | 16537 |
| Visualize | View Tree Chart |

| Inputs and Outputs | |
|---|---|
| Total Input | 0.00461133 BTC |
| Total Output | 0.00451265 BTC |
| Fees | 0.00009868 BTC |
| Fee per byte | 28.603 sat/B |
| Fee per weight unit | 13.706 sat/WU |
| Estimated BTC Transacted | 0.00015073 BTC |
| Scripts | Hide scripts & coinbase |

## Input Scripts

ScriptSig:
Witness:
040047304402200 29fd9433c61fcdff38c7736a3bf943e79f4817085c071fbf636914e1e59810f022051f8135759320fab876f8835e7b9ac862c47dca7688b4448c8ccd53c103241c501473

## Output Scripts

0[] PUSHDATA(20)[f3b73d2ae8b00bdd6ad7976287e61b77f6eb5012]

0[] PUSHDATA(32)[91802d44d0f7e26ae0fae46ea0048b311164c348c29438e07e7fe638578c0641]

**Channel Id: 605429585201725440** Tx-hash: 1d5b....e432 **Closed**

**Short Channel Id** 550635x630x0

**Channel Point** 1d5b13e6b32f3ef6f45991a4d776a60a5b8e6048514bcf4d866f2bab5473e432:0 [block explorer]

**Capacity** 0.00461133 BTC 461,133 sat (0.000431%) $18.42

**Last Update** 3 months ago **First Seen** Monday, November 19, 2018 **Last Seen** 3 months ago **Age** 4 days

**Node 1: 026c7d28784791a4b31a64eb34d9ab01552055b795919165e6ae886de637632efb**

| Alias | Capacity | Time Lock Delta | Min HTLC | Base Fee | Fee Rate |
|---|---|---|---|---|---|
| LivingRoomOfSatoshi.com_LND_1 (Sydney, NSW, Australia) | 7.03551143 BTC $28,104.41 703,551,143 sat | 144 | 0 | 1.000 sat $0.000039947 | 0.000001 sat $0.000000000040 |

**Node 2 [this node]: 03efccf2c383d7bf340da9a3f02e2c23104a0e4fe8ac1a880c8e2dc92fbdacd9df**

| Alias | Capacity | Time Lock Delta | Min HTLC | Base Fee | Fee Rate |
|---|---|---|---|---|---|
| ln.rene-pickhardt.de (Germany) | 0.96493183 BTC $3,854.56 96,493,183 sat | 14 | 1,000 | 0.001 sat $0.000000040 | 0.000001 sat $0.000000000040 |

**Channel Id: 605429585201725440** Tx-hash: 1d5b....e432 **Closed**

**Short Channel Id** 550635x630x0

**Channel Point** 1d5b13e6b32f3ef6f45991a4d776a60a5b8e6048514bcf4d866f2bab5473e4321:0 [block explorer]

**Capacity** 0.00461133 BTC 461,133 sat (0.000431%) $18.42

**Last Update** 3 months ago **First Seen** Monday, November 19, 2018 **Last Seen** 3 months ago **Age** 4 days

**Node 1: 026c7d28784791a4b31a64eb34d9ab01552055b795919165e6ae886de637632efb**

| Alias | Capacity | Time Lock Delta | Min HTLC | Base Fee | Fee Rate |
|---|---|---|---|---|---|
| LivingRoomOfSatoshi.com_LND_1 (Sydney, NSW, Australia) | 7.03551143 BTC $28,104.41 703,551,143 sat | 144 | 0 | 1.000 sat $0.000039947 | 0.000001 sat $0.000000000040 |

**Node 2 [this node]: 03efccf2c383d7bf340da9a3f02e2c23104a0e4fe8ac1a880c8e2dc92fbdacd9df**

| Alias | Capacity | Time Lock Delta | Min HTLC | Base Fee | Fee Rate |
|---|---|---|---|---|---|
| ln.rene-pickhardt.de (Germany) | 0.96493183 BTC $3,854.56 96,493,183 sat | 14 | 1,000 | 0.001 sat $0.000000040 | 0.000001 sat $0.000000000040 |

# Lightning node on chain wallets

- Currently don't seem to provide privacy features
- Funding tx has
  - 2-2 multisig output
  - Optionally a regular change output
  - Such tx do not have to be funding tx
- Mutual close will be hard to connect to a channel
- Unilateral / breach close could more easily be identified
- Lightning nodes wallets could include privacy features in future
  - like coin join
  - Other measures from https://en.bitcoin.it/wiki/Privacy

# Channel probing attack for channel balance

- Source based routing was a requirement
- An attacker can request routing a cycle route to herself
  - And never release the preimage
- If route is successfully set up information about channel balance is revealed
- This attack is free as
  - setting up htlcs comes at literally no cost.
    - Only networking & small computational overhead
- This gives a lower / upper estimate about channel balances…
  - As long as JIT Routing is not implemented

- Users might share channel balance for services like route / path finder

# The Lightning Network a tool to increase privacy?

- Single payments can be hidden and are not stored to the blockchain
- Nodes can run on tor onions
- Running nodes without announcing them or channel partners is possible
  - Rendez vous routing will increase such an operation by a lot
- Payments are sent via onions and hard to trace, correlate
  - Scriptless scripts will allow for per hop preimages
- No routing tables or public information about channel balances
- Future AMP-Routing might create standard sized payments
  - making traffic analysis even more difficult as onions cannot be correlated
- Random overpayment allowed by protocol to obfuscate onions
  - C-lightning has a configurable max fee rate that nodes are willing to pay
  - Overpayment will be between max accepted fee and the actual fee

# Security breaches by poor implementations

(Chapter 8c)

# Implementations can (deliberately) compromise privacy

- Protocol does not guarantee perfect privacy
  - Remember IP addresses and Bitcoin addresses can be linked via gossip
- Poor implementations yield a risk
  - Even without bad intend
- Some points to watch out for will be discussed
  - The goal is to have a growing list of points to check for with the open source implementations

# Not randomizing CLTV deltas in route construction

- From Bolt 7 : routing gossip:
  - If a route is computed by simply routing to the intended recipient and summing the `cltv_expiry_deltas`, then it's possible for intermediate nodes to guess their position in the route. Knowing the CLTV of the HTLC, the surrounding network topology, and the `cltv_expiry_deltas` gives an attacker a way to guess the intended recipient. Therefore, it's highly desirable to add a random offset to the CLTV that the intended recipient will receive, which bumps all CLTVs along the route.
- Possible infringement:
  - Create a popular implementation / plugin that uses the minimum CLTV deltas

# Implementations avoiding random overpayments

- From Bolt 4 : Onion Routing:
  - If the amount paid is more than twice the amount expected:
    - SHOULD fail the HTLC.
    - SHOULD return an `incorrect_or_unknown_payment_details` error.
      - Note: this allows the origin node to reduce information leakage by altering the amount while not allowing for accidental gross overpayment.
  -
- Possible infringement:
  - Create a popular implementation / plugin that never overpays.
  - Nodes do not have to do overpayments
    - C-lightning does random overpayment to an amount up to the maximum accepted fee rate for the payments
    - Other implementations could use other measures

# Man in the middle attacks of the Noise protocol

- From Bolt 08 : Transport
  - Authenticating each message sent ensures that a man-in-the-middle (MITM) hasn't modified or replaced any of the data sent as part of a handshake, as the MAC check would fail on the other side if so.
  - A successful check of the MAC by the receiver indicates implicitly that all authentication has been successful up to that point. If a MAC check ever fails during the handshake process, then the connection is to be immediately terminated.

- This means that a poor implementation in which checking the MAC is omitted allows for man in the middle attacks
  - In my opinion unlikely to happen

# Bolt 02 bolt 03 basepoints unpredictable commitment tx

The various _basepoint fields are used to derive unique keys as described in BOLT #3 for each commitment transaction. Varying these keys ensures that the transaction ID of each commitment transaction is unpredictable to an external observer, even if one commitment transaction is seen; this property is very useful for preserving privacy when outsourcing penalty transactions to third parties.

Possible infringement:

- Don't follow the spec for key derivation (will not be detected)

# Ephemeral keys on the Transport layer

- Noise XK handshake uses ephemeral keys
- It is not specified in a deterministic way which keys should be chosen

Possible infringement:

- A node could reuse the ephemeral keys from onion routing
- A node could reuse the commitment_base_points / secrets

# Dual funded channels used to probe for offchain balance

- Current dual funded channel proposal has no means of stopping to probe peers for their onchain liquidity

# Security breaches by services

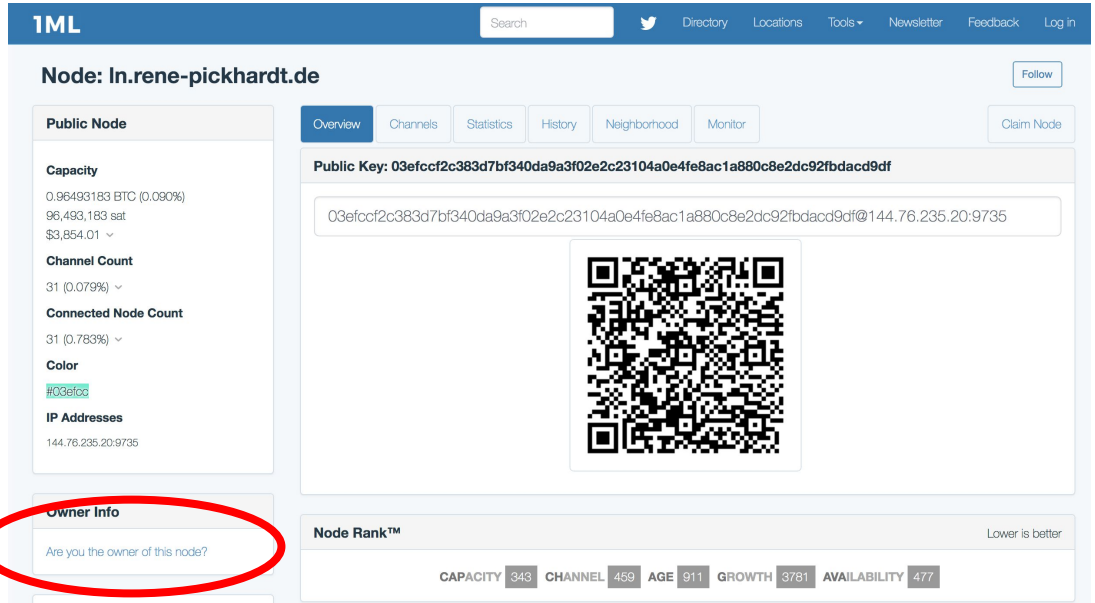
(Chapter 8d)

# 3rd party services for lightning can compromise privacy

- General gist services increasing the user experience often do this by trading for data
- We will explain how some services can explicitly be constructed to collect data
- The thoughts here should be taken into consideration for any third party service and are not particular for services on top of lightning

# Lightning explorers like 1ml.com

- Users can claim "their" node and provide authenticated meta data
- Web users showing interest in certain nodes can be tracked



# Custodial services with KYC are an obvious risk

- Centralized exchanges implementing lightning
- Custodial Wallets
  - Tip bots
- Web services
  - Web shops
  - Payment provider

# Decentralized Exchanges

- Zigzag.io / sideshift.ai
  - Can correlate an onchain tx with a lightning invoice
  - Can collect meta data like IP address of the user
- Submarine swaps
  - Same as zigzag.io

# Implementations (wallets) / plugins

- Implementations can obviously know anything
- C-lightning offers plugins
  - Plugins can basically access all data of the node
  - Potential ideas for critical plugins
    - Routing service
    - Autopilot
    - Watchtower service
    - Escrow services

# User Interfaces / UX improving tools

- Remote control apps:
  - Can basically do everything a plugin / implementation can do
  - Examples
    - Spark-wallet (c-lightning)
    - Zap-wallet (lnd)
- Hardware projects can theoretically lead to a security breach (examples which should checked before using them)
  - Casa node
  - Hodlhodl
  - Raspiblitz
  - Hardware wallets (work in progress)
  - Point of sale systems

# References and helpful links

- https://github.com/lightningnetwork/lightning-rfc
- http://noiseprotocol.org/noise.html
- https://www.youtube.com/watch?v=ceGTgqypwnQ (Noise explained)
- https://www.youtube.com/watch?v=3gipxdJ22iM (Noise overview)
- https://cypherpunks.ca/~iang/pubs/Sphinx_Oakland09.pdf (SPHINX Mix Format Paper)
- https://www.youtube.com/watch?v=34TKXELJa2c (SPHINX Mix Format presented)
- https://en.wikipedia.org/wiki/Poly1305
- https://en.wikipedia.org/wiki/Salsa20#ChaCha_variant
- https://tools.ietf.org/html/rfc7539 **ChaCha20 and Poly1305 for IETF Protocols**
- https://www.youtube.com/user/RenePickhardt
- https://bitcoin.stackexchange.com/questions/tagged/lightning-network
- https://en.bitcoin.it/wiki/Script
- https://en.bitcoin.it/wiki/Transaction
- https://lightning.network/lightning-network-paper.pdf
- https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm

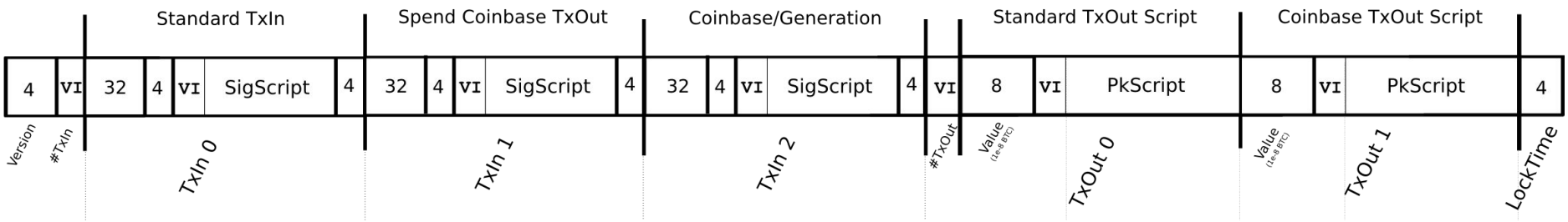
# Backup Slides

# A standard Bitcoin Transaction has 6 data fields

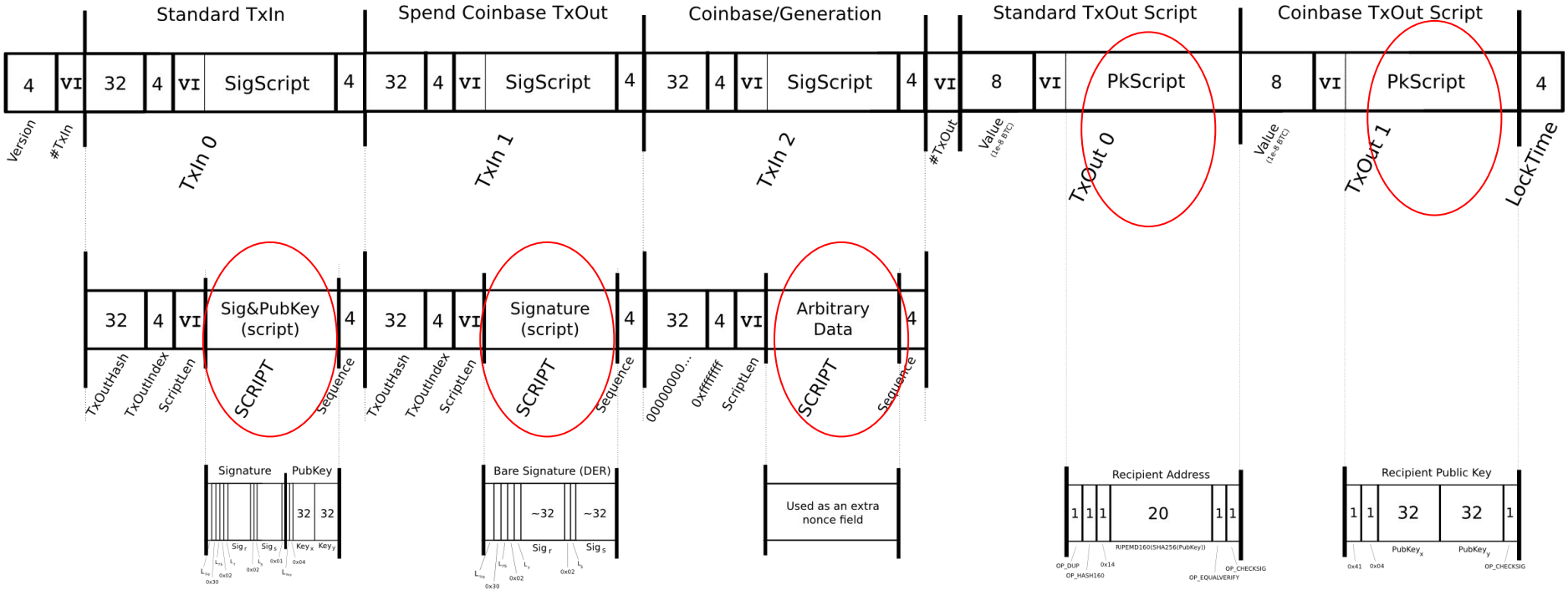
( The following is a brief summary of https://en.bitcoin.it/wiki/Transaction#Input )

- Version number (4 Bytes)
- In-Counter (1-9 Bytes)
- List of inputs (depending on the Value of <In-Counter>)
- Out-Counter (1-9 Bytes)
- List of Outputs (depending on the Value of <Out-Counter>)
- Lock_time (4 Bytes)

# Bitcoin Transaction with 3 inputs and 2 outputs

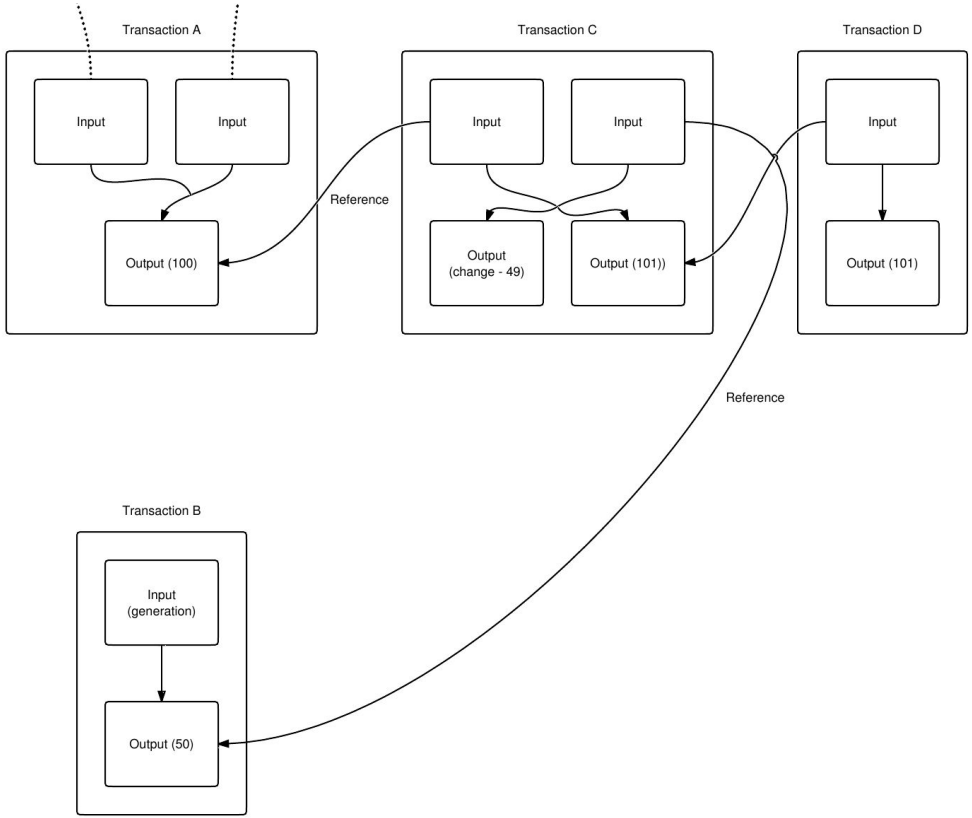


# Data consists mainly of executable scripts!




other serializations use little-endian

# A chain of Bitcoin Transactions with 2 UTXO




- Outputs are references by the inputs
- The Script in the output defines how the transaction can be spent
- Owning Bitcoins means being able to spend the output of an unspent transaction
  - Provide an input script
  - Concatenate it with with some outputscript of an unspent transaction
  - The combined Script needs to evaluate to True

# Spending a Pay-to-PubkeyHash (standard TX)

- ScriptPubKey (aka the Output Script)
  - OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
- ScriptSig: (aka the Input Script)
  - <sig> <pubKey>
- Explainations
  - OP_CODES are the instructions of the script language within Bitcoin
  - <data>  is depicted like html tags with lesser than and greater than signs
  - The complete script is concatenated as Input || Output and then being executed on a Stack machine

<sig>  <pubKey>  OP_DUP  OP_HASH160  <pubKeyHash>  OP_EQUALVERIFY  OP_CHECKSIG

A video dissecting a P2PKH transaction: https://www.youtube.com/watch?v=1n4g3eYX1UI

# Execution of Pay-to-PubkeyHash Script

<sig> <pubKey> OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG

Data is pushed on the stack

<sig>

# Execution of Pay-to-PubkeyHash Script

~~<sig>~~ <pubKey> OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG

Data is pushed on the stack again

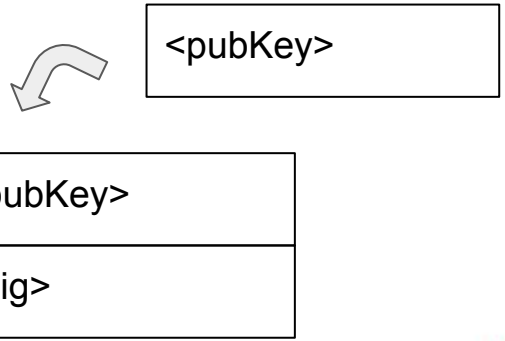
<pubKey>

<sig>

# Execution of Pay-to-PubkeyHash Script

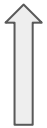
~~<sig> <pubKey>~~ OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG

The OP_CODE OP_DUP is being executed by pushing the top element to the stack again



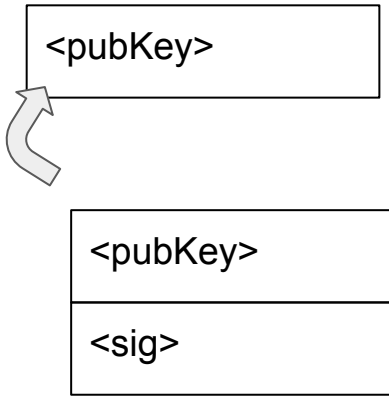
# Execution of Pay-to-PubkeyHash Script

~~<sig> <pubKey> OP_DUP~~ OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG



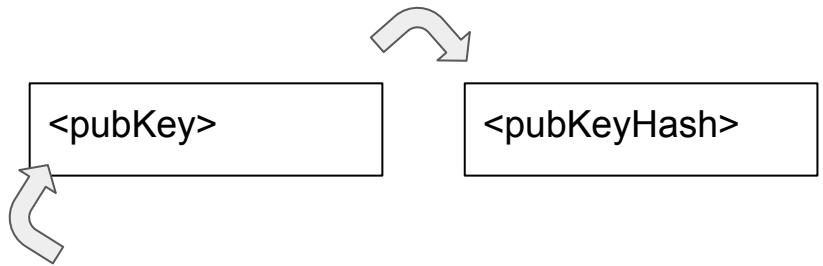
The OP_CODE OP_HASH160 is being executed:

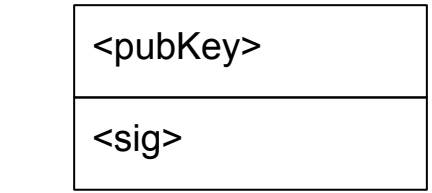
It takes the top element of the Stack



# Execution of Pay-to-PubkeyHash Script

~~<sig> <pubKey> OP_DUP~~ OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG



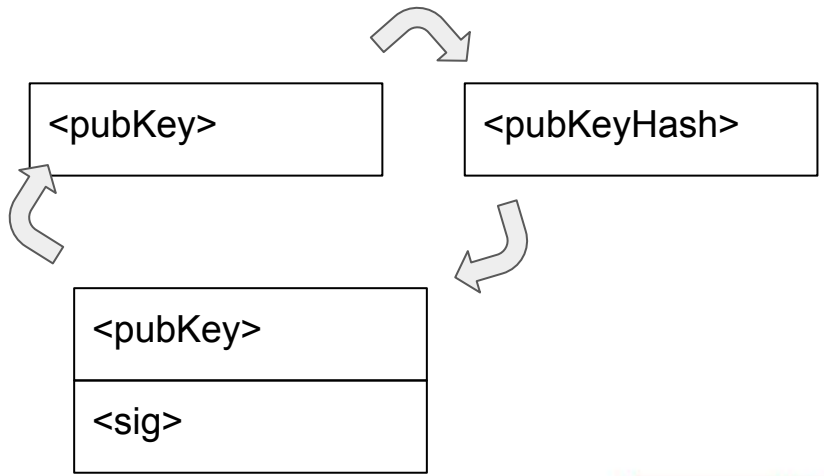

The OP_CODE OP_HASH160 is being executed:

It takes the top element of the Stack
The RIPEMD-160 Hash is calculated

By design of the input and output script this will be the Hash of the Public Key (aka the Bitcoin Address)

# Execution of Pay-to-PubkeyHash Script

~~<sig> <pubKey> OP_DUP~~ OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG



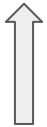

The OP_CODE OP_HASH160 is being executed:

It takes the top element of the Stack
The RIPEMD-160 Hash is calculated
And the result is pushed back on the stack

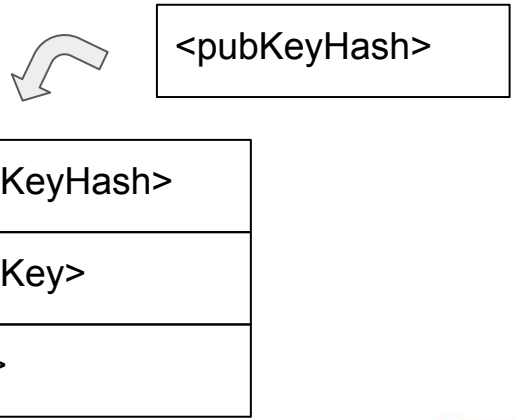
By design of the input and output script this will be the Hash of the Public Key (aka the Bitcoin Address)

# Execution of Pay-to-PubkeyHash Script

~~<sig> <pubKey> OP_DUP OP_HASH160~~ <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG

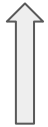


Data is pushed on the stack again



# Execution of Pay-to-PubkeyHash Script

~~<sig> <pubKey> OP_DUP OP_HASH160 <pubKeyHash>~~ OP_EQUALVERIFY OP_CHECKSIG



OP_EQUALVERIFY checks if the two top elements of the stack are equal.

While checking the elements will be removed

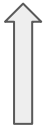
Like the instruction set in the ALU of the CPU OP_CODES know how much data they need to consume (and will do so or fail otherwise)

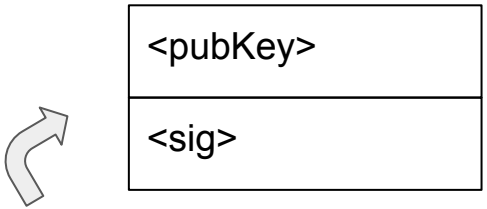
| <pubKeyHash> |
| --- |
| <pubKeyHash> |

| <pubKey> |
| --- |
| <sig> |

# Execution of Pay-to-PubkeyHash Script

~~<sig> <pubKey> OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY~~ OP_CHECKSIG






OP_CHECKSIG evaluates that the PubKey which was the top element fits to the signature which was the 2. Element on the stack

If this fits together ownership is proven and the Transaction will be added to the block with a new Output script.

After being mined the transaction is now considered to be spent. Double spending cannot take place since the assumption was that the output was not spent yet.

# The Bitcoin miners will build the next block

- Collect published and valid transactions
  - In particular the hashes of the transactions
  - Most likely those that offer highest mining fees
  - Mining fees are leftovers of inputs which have not assigned to outputs
  - As many as there is space within the block
  - Start with one special transaction which has no input generating the block reward as an output
- Take some additional meta data
  - The hash of the previous block
  - The nonce (and extra nonce as part of the coinbase)
- Compute the merkle tree of all the hashes
- Look out for a hash collision with respect to the mining difficulty
- If your Tx was in: Congratulations you now sent bitcoins


Let's talk lightning!


I love taking photographs of lightning bolts...

# Copyright notice

- This slide deck is openly licensed with a creative commons license CC-BY-SA-4.0.
  - The full license text can be found at: https://creativecommons.org/licenses/by-sa/4.0/legalcode
- You are
  - free to
    - Share
    - Remixe
  - As long as you
    - Link to the original work
    - State my Name and Website
    - Mark changes in your derivative work
    - Use the same license for your derivative work
- Screenshots in this slide deck are taken by me but the design of the websites might be protected by copyright
- This slide deck uses parts of the lightning-rfc which is licensed as CC-BY (the lightning developers)
  - The full license text can be found at: https://creativecommons.org/licenses/by/4.0/legalcode
- The graphics from the backup slides are taken from https://en.bitcoin.it/wiki/Transaction and are Public Domain

Thanks to Marietheres Viehler (aka journalspiration) for the design of the title slide.

# About this slide deck

The purpose is to help spreading education about the Lightning Network Protocol so that the technology will be adopted more quickly by more people. This shall be my contribution to the Bitcoin / Lightning Network Community.

This slide deck - to the best of my knowledge - is the most comprehensive work making an introduction to the BOLT standard. I will hold several talks about this slide deck in Juni 2019 at the Bitcoin / Lightning Residency organized by Chaincodelabs who will also record the talk. So watch out for them because actual presentations of the slide deck will probably be more valuable than just the slides you're holding in your hand.

The slides are part of my effort to create a book about the lightning network. You can follow that effort at: https://github.com/renepickhardt/The-Lightning-Network-Book or you can support the effort at my fundrasing pages at: https://tallyco.in/s/lnbook or at: https://www.patreon.com/renepickhardt or at 1GZx8tWgDd21Rd8b1QdMrzdZGHgyfVkzaD part of this effort also consists of creating video tutorials and teaching materials on my Youtube Channel over at: https://www.youtube.com/user/RenePickhardt

This work was funded (sorted by amount of contribution from top to bottom) by: Me personally, fulmo.org, everyone who contributed to the above mentioned fundraiser and George Danzer.

Thank you to the lightning Developers and people in various telegram groups for helpful discussions