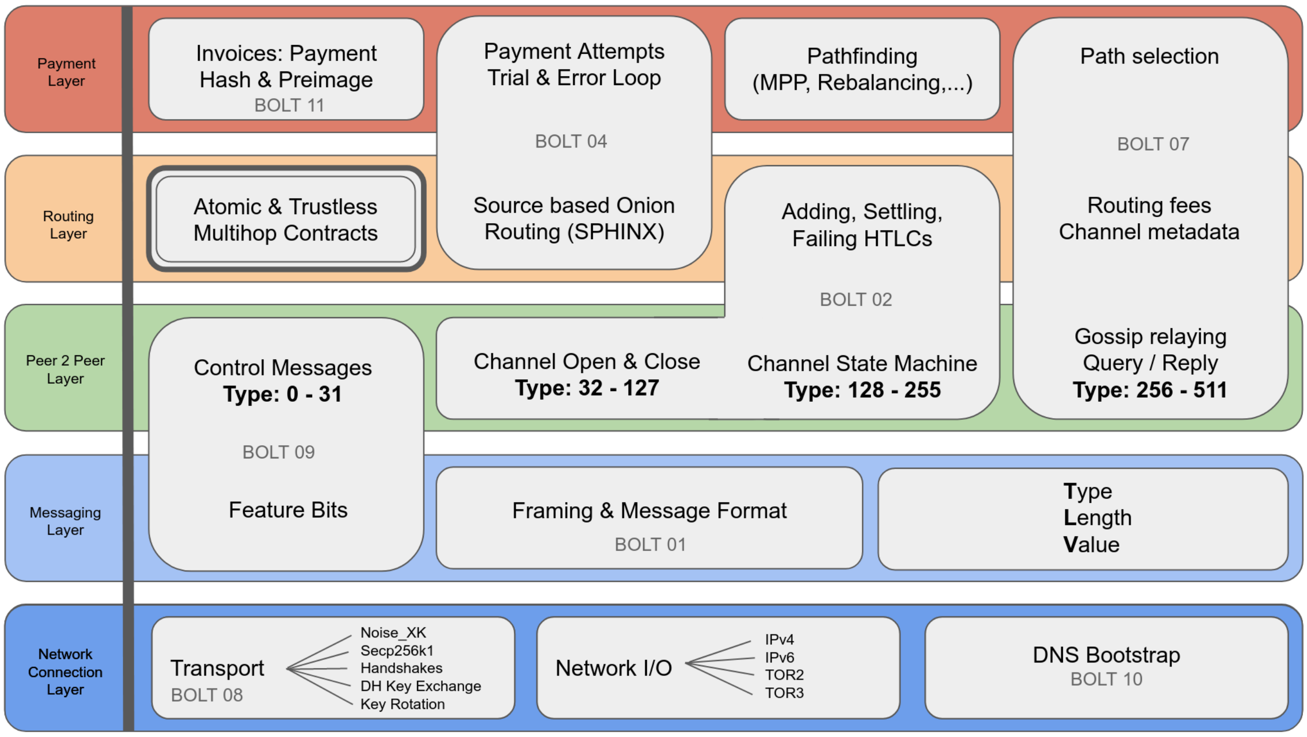
Routing on a Network of Payment channels

In this section we will finally unpack how payment channels can be connected to a network of other payment channels via a process called *routing*. Specifically, we will look at the first part of the routing layer, the Atomic & Trustless Multihop Payment protocol. This is highlighted by a double outline in the protocol suite diagram:



Payment Layer
Invoices: Payment Hash & Preimage
BOLT 11
Payment Attempts Trial & Error Loop
BOLT 04
Pathfinding (MPP, Rebalancing,...)
Path selection
BOLT 07
Routing Layer
Atomic & Trustless Multihop Contracts
Source based Onion Routing (SPHINX)
Adding, Settling, Failing HTLCs
BOLT 02
Routing fees Channel metadata
Peer 2 Peer Layer
Control Messages Type: 0 - 31
BOLT 09
Channel Open & Close Type: 32 - 127
Channel State Machine Type: 128 - 255
Gossip relaying Query / Reply Type: 256 - 511
Messaging Layer
Feature Bits
Framing & Message Format
BOLT 01
Type Length Value
Network Connection Layer
Transport
BOLT 08
Noise_XK
Secp256k1
Handshakes
DH Key Exchange
Key Rotation
Network I/O
IPv4
IPv6
TOR2
TOR3
DNS Bootstrap
BOLT 10

Figure 1. The Lightning Network Protocol Suite

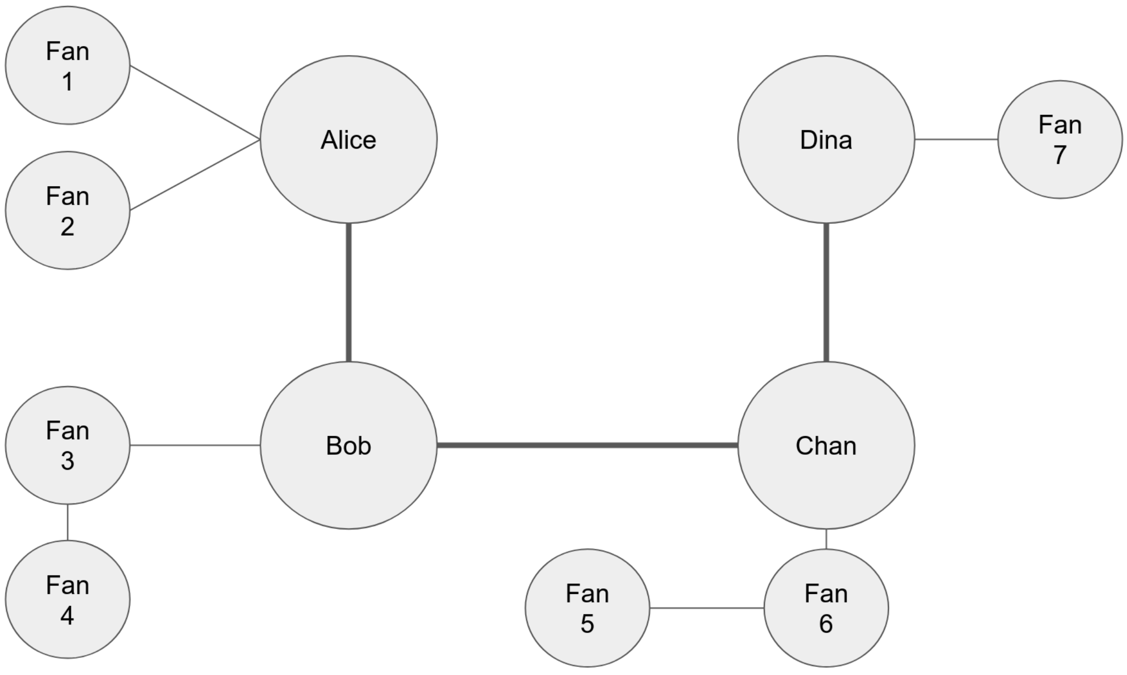
Routing a payment

In this section we will examine routing from the perspective of Dina, a gamer who receives donations from her fans while she streams her game sessions.

The innovation of routed payment channels allows Dina to receive tips without maintaining a separate channel with every one of her fans who want to tip her. As long as there exists a path of well-funded channels from that viewer to Dina, she will be able to receive payment from that fan. The nodes along the path from the fan to Dina are intermediaries and called "routing nodes" in the context of routing a payment.

There is no functional difference between the "routing nodes" and the nodes operated by Dina's fans. Any Lightning node is capable of routing payments across its payment channels.

Any one of Dina's fans in the diagram can pay her by routing via the nodes in between them and Dina



Importantly, the routing nodes are unable to steal the funds while routing a payment from a fan to Dina. Furthermore, routing nodes cannot lose money while participating in the routing process. Routing nodes can charge a routing fee for acting as an intermediary, although they don't have to and may choose to route payments for free.

Another important detail is that due to the use of onion routing, intermediary nodes are only explicitly aware of the one node preceding them and the one node following them in the route. They will not necessarily know who is the sender and recipient of the payment. This enables fans to use intermediary nodes to pay Dina, without leaking private information and without risking theft.

This process of connecting a series of payment channels with end-to-end security, and the incentive structure for nodes to *forward* payments, is one of the key innovations of the Lightning Network.

In this chapter, we'll dive into the mechanism of routing in the Lightning Network, detailing the precise manner in-which payments flow through the network.

First, we will clarify the concept of "routing" and compare it to that of "path finding", as these are often confused and used interchangeably.

Next, we will construct the fairness protocol: A Conditional Chained End-to-End Secure Payment (CCESP) used to route payments. To demonstrate how this fairness protocol works, we will be using a physical equivalent of transferring gold coins between 4 people.

Finally, we will look at the CCESP implementation currently used in the Lightning Network, which is called a Hash Time-Locked Contract (HTLC).

Routing vs. Path Finding

It's important to note that we separate the concept of *routing* from the concept of *path finding*. These two concepts are often confused and the term "routing" is often used to describe both concepts. Let’s remove the ambiguity, before we proceed any further.

Path Finding, which is covered in [path_finding] is the process of finding and choosing a contiguous path made of payment channels which connects the sender A to the recipient B. The sender of a payment does the path finding, by examining the *channel graph* which they have assembled from channel announcements gossiped by other nodes.

Routing refers to the series of interactions across the network that allow a payment to *flow* from A to B, across the path previously selected by path finding. Routing is the *active* process of sending a payment on a path, which involves the cooperation of all the intermediary nodes along that path.

An important rule of thumb is that it’s possible for a *path* to exist between Alice and Bob, yet there may not be an active *route* on which to send the payment.

One example is the scenario where all the nodes connecting Alice and Bob are currently off-line.

In theory, one can examine the *channel graph* and connect a series of payment channels from Alice to Bob, hence a *path* exists. However, as the intermediary nodes are offline, the payment cannot be sent and so no *route* exists.

Creating a network of payment channels

Before we dive into the concept of a conditional chained end-to-end secure payment, let’s work through an example. Let us to return to Alice who, in previous chapters, purchased a coffee from Bob with whom she has an open channel. Alice now watches a live stream from Dina the gamer, and wants to send her a tip via the Lightning Network. However, Alice has no direct channel with Dina. Alice could open a direct channel, however that would require liquidity and on-chain fees which could be more than the value of the tip itself.

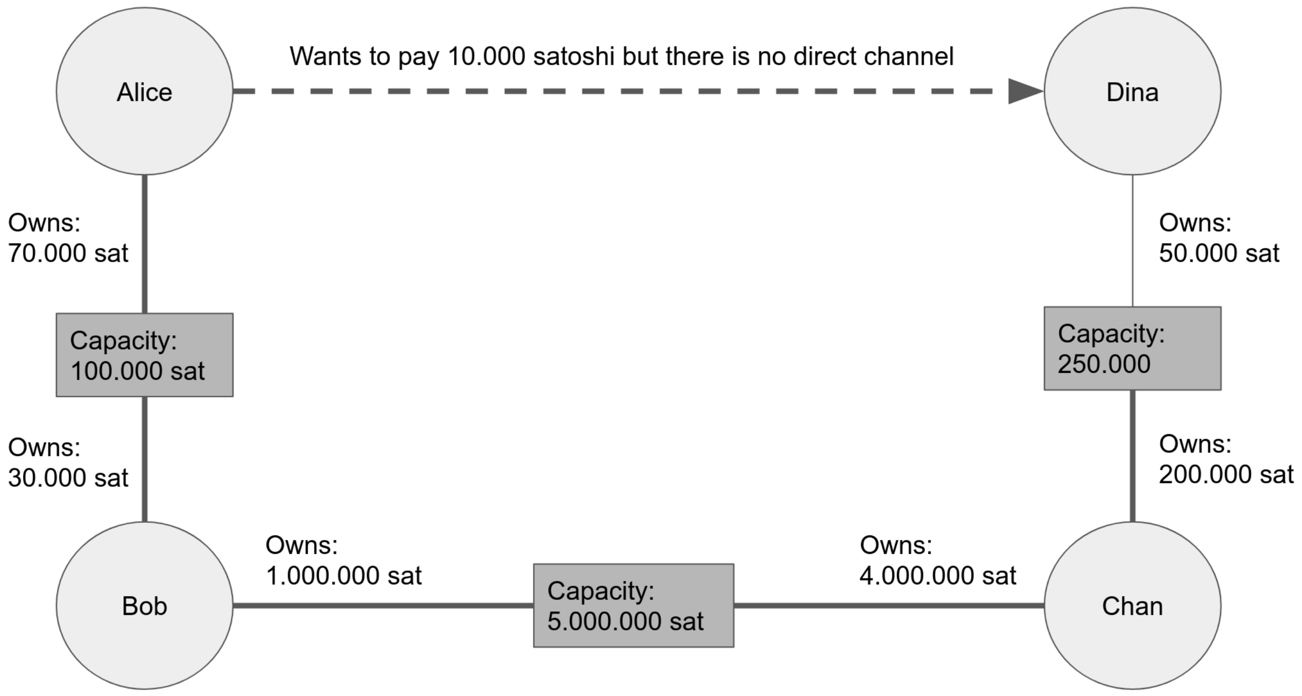
Instead, Alice can use her existing open channels to send a tip to Dina *without* the need to open a channel directly with Dina.

This is possible, as long as there exists some path of channels from Alice to Dina with sufficient capacity to route the tip.

From previous chapters, we know Alice has an open channel with Bob, the coffee shop owner. Bob, in turn, has an open channel with the software developer Chan who helps him with the point of sale system he uses in his coffee shop. Chan is also the owner of a large software company which develops the game that Dina plays, and they already have an open channel which Dina uses to pay for the game’s license and in-game items.

If we draw out this series of payment channels, it’s possible to manually trace a *path* from Alice to Dina that uses Bob and Chan as intermediary routing nodes. Alice can then craft a *route* from this outlined path, and use it to send a tip of a few thousand satoshis to Dina, with the payment being *forwarded* by Bob and Chan. Essentially, Alice will pay Bob, who will pay Chan, who will pay Dina. And no direct channel from Alice to Dina is required.

A network of payment channels



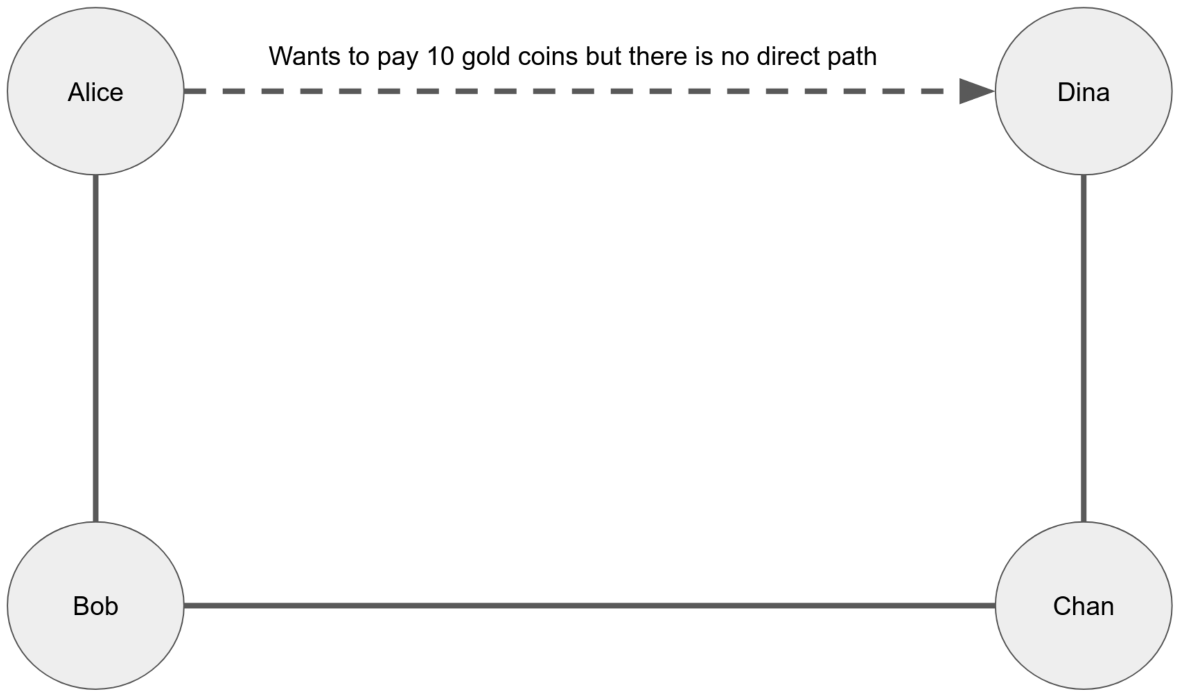
The main challenge is to do this in a way that prevents Bob and Chan from stealing the money that Alice wants delivered to Dina.

A physical example of "routing"

To understand how the Lightning Network protects the payment while being routed, we can compare to an example of routing physical payments with gold coins in the real world.

Assume Alice wants to give 10 gold coins to Dina, but does not have direct access to Dina. However, Alice knows Bob, who knows Chan, who knows Dina and so she decides to ask Bob and Chan for help.

Alice wants to pay Dina 10 gold coins



She can pay Bob to pay Chan to pay Dina, but how does she make sure that Bob or Chan don't run off with the coins after receiving them? In the physical world contracts could be used for safely carrying out a series of payments.

Alice could negotiate a contract with Bob which reads:

> *I (Alice) will give you (Bob) 10 gold coins if you pass them on to Chan*

While this contract is nice in the abstract, in the real world, Alice runs the risk that Bob might breach the contract and hope to not get caught by law enforcement. Even if Bob gets caught by law enforcement, Alice faces the risk that he might be bankrupt and be unable to return her 10 gold coins. Assuming these issues are magically solved, it's still unclear how to leverage such a contract to achieve our desired outcome: the coins ultimately being delivered to Dina.

We thus improve our contract:

> *I (Alice) will reimburse you (Bob) with 10 gold coins if you can prove to me (for example via a receipt) that you already have delivered 10 gold coins to Chan*

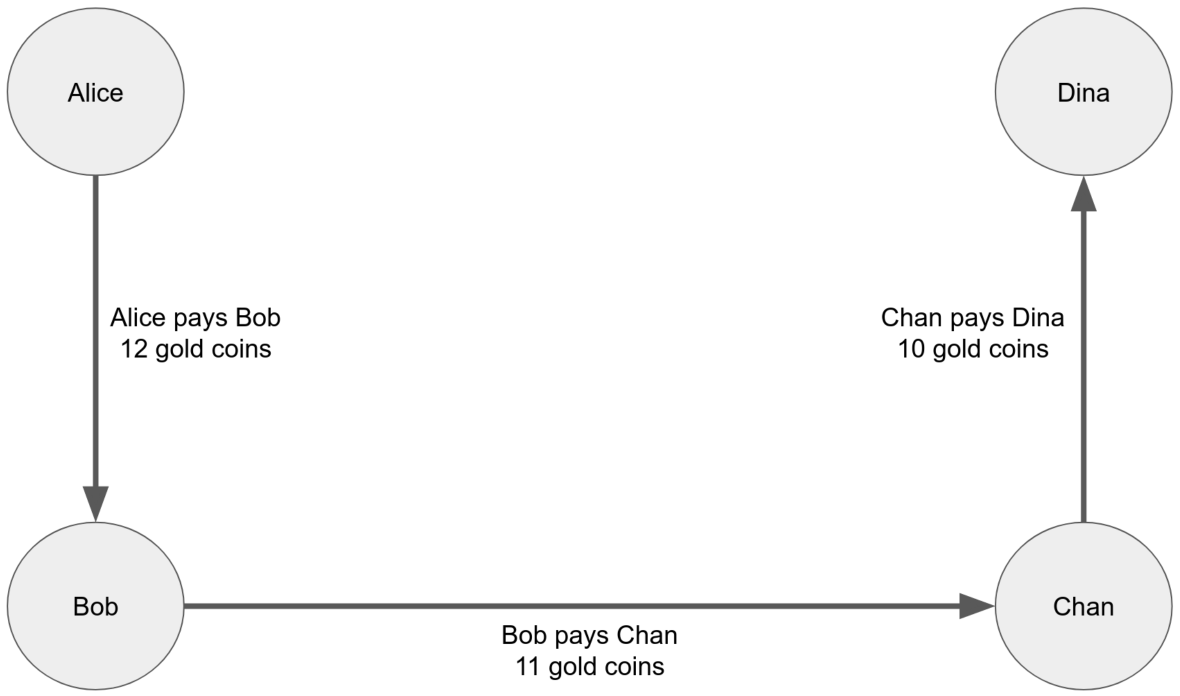
You might ask yourself why should Bob sign such a contract. He has to pay Chan but ultimately gets nothing out of the exchange, and he runs the risk that Alice might not reimburse him. Bob could offer Chan a similar contract to pay Dina, but similarly Chan has no reason to accept it either. Even putting aside the risk, Bob and Chan must *already* have 10 gold coins to send, otherwise they wouldn't be able to participate in the contract. Thus Bob and Chan face both risk and opportunity cost for agreeing to this contract, and they would need to be compensated in order for them to accept it.

Alice can this make this attractive to both Bob and Chan, by offering them fees of 1 gold coin each, if they transmit her payment to Dina. The final contract would instead read:

> *I (Alice) will reimburse you (Bob) with 12 gold coins if you can prove to me (for example via a receipt) that you already have delivered 11 golden coins to Chan*

Alice now promises Bob 12 gold coins. There are 10 to be delivered to Dina and 2 for the fees. She promises 12 to Bob if he can prove that he has forwarded 11 to Chan. The difference of 1 gold coin is the fee that Bob will earn for helping out with this particular payment.

Alice pays Bob, Bob pays Chan, Chan pays Dina


Alice
Dina
Alice pays Bob 12 gold coins
Chan pays Dina 10 gold coins
Bob
Chan
Bob pays Chan 11 gold coins

As there is still the issue of trust and the risk that either Alice or Bob don't honor the contract, all parties decide to use an escrow service. At the start of the exchange, Alice could "lock up" these 12 golden coins in escrow that will only be paid to Bob once he proves that he's paid 11 golden coins to Chan.

This escrow service is an "ideal functionality", which will later be replaced by a more trust-minimized mechanism. Let's assume for now that everyone trusts this escrow.

In the Lightning Network, the receipt (proof of payment) could take the form of a secret that only Dina knows. In practice, this secret would be a large random number that is large enough to prevent others from guessing it (typically *very, very* large number, encoded using 256 bits!).

Dina generates this secret value R from a random number generator.

The secret could then be committed to the contract by including the SHA256 hash of the secret in the contract itself, as follows:

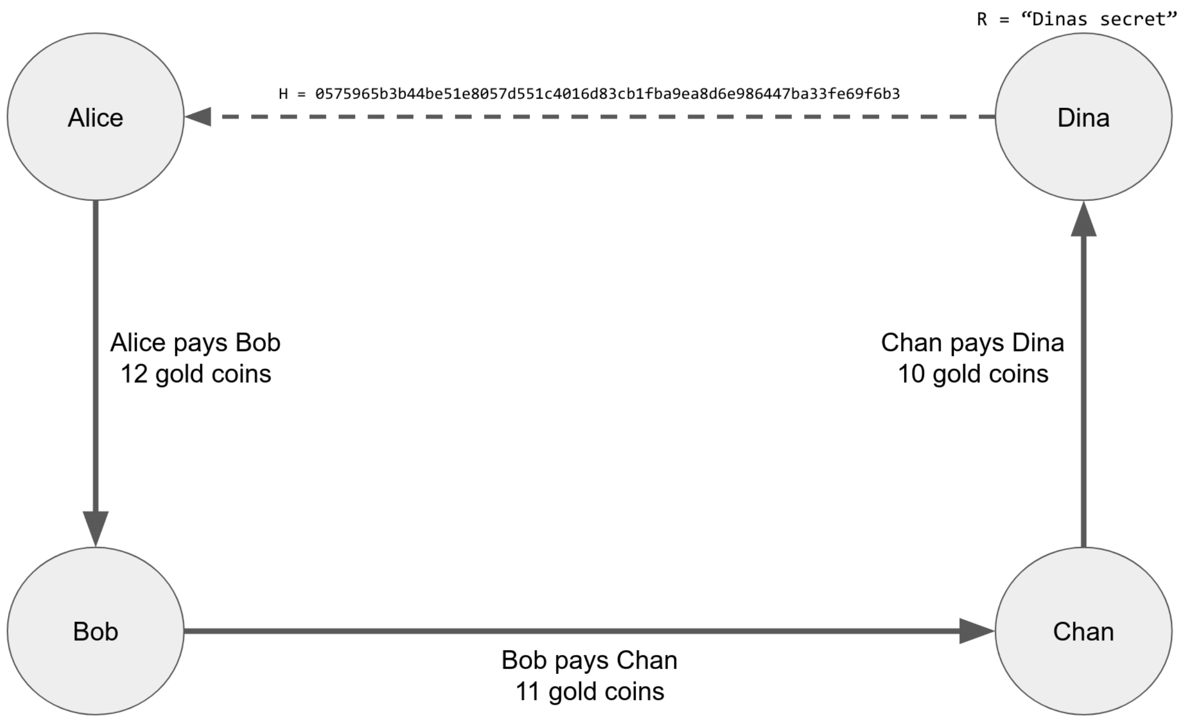
`\(H = SHA256(R)\)`

We call this hash of the payment's secret the payment hash. The secret which "unlocks" the payment is called the payment secret.

For now, we keep things simple and assume that Dina's secret is simply the text line: `Dinas secret`. In order to "commit" to this secret, she computes the SHA256 hash which when encoded in hex, can be displayed as: `0575965b3b44be51e8057d551c4016d83cb1fba9ea8d6e986447ba33fe69f6b3`. [1]

To facilitate Alice's payment, Dina will create the secret and the payment hash and send the payment hash to Alice.

Dina sends the hashed secret to Alice


R = "Dinas secret"
H = 0575965b3b44be51e8057d551c4016d83cb1fba9ea8d6e986447ba33fe69f6b3
Alice
Dina
Alice pays Bob 12 gold coins
Chan pays Dina 10 gold coins
Bob
Chan
Bob pays Chan 11 gold coins

Alice doesn't know the secret but she can rewrite her contract to use the hash of the secret as a proof of payment:

> *I (Alice) will reimburse you (Bob) with 12 gold coins if you can show me a valid message that hashes to: `057596...`. You can acquire this message by setting up a similar Contract with Chan who has to set up a similar contract with Dina. In order to assure you that you will get reimbursed I will provide the 12 gold coins to an trusted escrow before you set up your next contract.*

This new contract now protects Alice from Bob not forwarding to Chan, protects Bob from not being reimbursed by Alice, and ensures that there will be proof that Dina was ultimately paid via the hash of Dina's secret. This secret message that hashes to the 057596... is called a *pre-image*.

After Bob and Alice agree to the contract, and Bob receives the message from the escrow that Alice has deposited the 12 gold coins, Bob can now negotiate a similar contract with Chan.

Note that since Bob is taking a service fee of 1 coin, he will only forward 11 gold coins to Chan once Chan shows proof that he has paid Dina. Similarly, Chan will also demand a fee and will expect to receive 11 gold coins once he has proved that he has paid Dina the promised 10 gold coins.

Bob’s contract with Chan will read:

> *I (Bob) will reimburse you (Chan) with 11 gold coins if you can show me a valid message that hashes to:`057596...`. You can acquire this message by setting up a similar contract with Dina. In order to assure you that you will get reimbursed I will provide the 11 gold coins to an trusted escrow before you set up your next contract.*

Once Chan gets the message from the escrow that Bob has deposited the 11 gold coins, Chan sets up a similar contract with Dina:

> *I (Chan) will reimburse you (Dina) with 10 golden coins if you can show me a valid message that hashes to:`057596...`. In order to assure you that you will get reimbursed after revealing the secret I will provide the 10 gold coins to an trusted escrow.*

Everything is now in place. Alice has a contract with Bob and has placed 12 gold coins in escrow. Bob has a contract with Chan and has placed 11 gold coins in escrow Chan has a contract with Dina and has placed 10 gold coins in escrow. It is now up to Dina to reveal the secret, which is the pre-image to the hash she has established as proof of payment.

Dina now sends "Dinas secret" to Chan.

He checks that "Dinas secret" hashes to +057596.... Chan now has proof of payment and so instructs the escrow service to release the 10 golden coins to Dina.

Chan now provides the secret to Bob. Bob checks it and instructs the escrow service to release the 11 gold coins to Chan.

Bob now provides the secret to Alice. Alice checks it and instructs the escrow to release 12 gold coins to Bob.

All the contracts are now settled. Alice has paid a total of 12 gold coins, 1 of which was received by Bob, 1 of which was received by Chan, and 10 of which were received by Dina. With a chain of contracts like this in place, Bob and Chan could not run away with the money because they deposited it in escrow first.

However, one issue still remains. If Dina refused to release her secret pre-image, then Chan, Bob, and Alice would all have their coins stuck in escrow but wouldn’t be reimbursed. And similarly if anyone else along the chain failed to pass on the secret, the same thing would happen. So while no one can steal money from Alice everyone can still lose money.

Luckily, this can be resolved by adding a deadline to the contract.

We could amend the contract so that if it is not fulfilled by a certain deadline, then the contract expires and the escrow service returns the money to the person who made the original deposit. We call this deadline a "time lock".

The deposit is locked with the escrow service for a certain amount of time, and is eventually released even if no proof of payment was provided.

In order to factor this in, the contract between Alice and Bob is once again amended with a new clause:

> *Bob has 24 hours to show the secret after the contract was signed. If Bob does not provide the secret by this time, Alice's deposit will be refunded by the escrow service and the contract becomes invalid.*

Bob, of course, now has to make sure he receives the proof of payment within 24 hours. Even if he successfully pays Chan, if he receives the proof of payment later than 24 hours he will not be reimbursed. To remove that risk, Bob must give Chan and even shorter deadline.

In turn, Bob will alter his contract with Chan in the following way:

> *Chan has 22 hours to show the secret after the contract was signed. If he does not provide the secret by this time, Bob's deposit will be refunded by the escrow service and the contract becomes invalid.*

As you might have guessed, Chan is now incentivized to also alter his contract with Dina:

> *Dina has 20 hours to show the secret after the contract was signed. If he does not provide the secret by this time, Bob's deposit will be refunded by the escrow service and the contract becomes invalid.*

With such a chain of contracts we can ensure that, after 24 hours, the payment will successfully deliver from Alice to Bob to Chan to Dina, or it will fail and everyone will be refunded. Either the contract fails or succeeds, there's no middle ground.

In the context of the Lightning Network, we call this "all or nothing" property *atomicity*.

As long as the escrow is trustworthy and faithfully performs its duty, then no party will have their coins stolen in the process.

The pre-condition to this *route* working at all, is that all parties in the path have enough money to satisfy the required series of deposits.

While this seems like a minor detail we will see in later this chapter that this requirement is actually one of the more difficult issues for Lightning Network nodes. It becomes progressively more difficult as the size of the payment increases. Furthermore, the parties cannot use their money while it is locked in escrow.

Thus users forwarding payments face an opportunity cost for locking the money, which is ultimately reimbursed through routing fees, as we saw in the example above.

Now that we've seen a physical payment routing example, we will see how this can be implemented on the Bitcoin blockchain, without any need for third-party escrow. To do this we will be setting up the contracts between the participants using Bitcoin Script. We replace the third-party escrow with *smart contracts* that implement a fairness protocol. Let’s break that concept down and implement it!

Fairness Protocol

As we saw in the first chapter of this book, the innovation of Bitcoin is the ability to use cryptographic primitives to implement a fairness protocol that substitutes trust in third parties (intermediaries), with a trusted protocol.

In our gold coin example, we needed an "escrow" service in order to prevent any one of the parties from reneging on their obligations. The innovation of cryptographic fairness protocols allows us to replace the escrow service with a protocol.

The properties of the fairness protocol we want to create are:

Trsutless Operation

The participants in a routed payment do not need to trust each other, or any intermediary or third party. Instead, they trust the protocol to protect them from cheating.

Atomicity

The payment is fully executed, or it fails and everyone is refunded. There is no possibility of an intermediary collecting a routed payment and not forwarding it to the next hop. Thus, the intermediaries can’t cheat or steal.

Multihop

The security of the system extends end-to-end for payments routed through multiple payment channels, just as it is for a payment between the two ends of a single payment channel.

An optional, additional property, is the ability to split payments into multiple parts while maintaining atomicity for the entire payment. These are called *Multi-Part Payments (MPP)* and are explored further in [multipart_payments].

Implementing Atomic Trustless Multihop Payments

Bitcoin Script is flexible enough that there are dozens of ways to implement a fairness protocol that has the properties of atomicity, trustless operation and multihop security. Choosing a specific implementation is dependent on certain tradeoffs between privacy, efficiency and complexity.

The fairness protocol for routing used in the Lightning Network today is called a Hash Time-Locked Contract (HTLC). HTLCs use a hash pre-image as the secret that unlocks a payment, as we saw in the gold coin example in this chapter. The recipient of a payment generates a random secret number and calculates its hash. The hash becomes the condition of payment and once the secret is revealed, all the participants can redeem their incoming payments. HTLCs offer atomicity, trustless operation and multihop security. While HTLCs are efficient and very simple, they involve a small compromise of privacy (see [htlc_privacy_compromise]).

Another proposed mechanism for implementing routing is a *Point Time-Locked Contract (PTLC)*. PTLCs also achieve atomicity, trustless operation and multihop security, but do so with increased efficincy and better privacy. Efficient implementation of PTLCs depends on a new digital signature algorithm called *Schnorr signatures*, which is expected to active in Bitcoin in 2021.

Revisiting our example

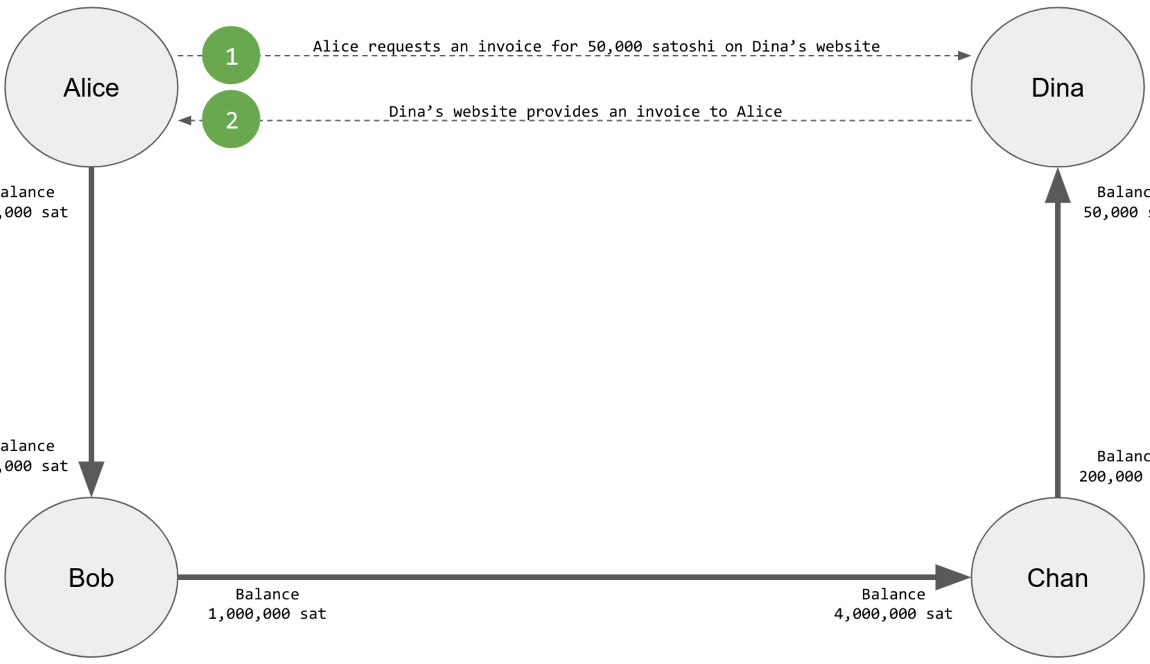
Let’s revisit our example from the first part of this chapter. Alice wants to "tip" Dina, with a Lightning payment. Let’s say Alice wants to send Dina 50,000 satoshis as a tip.

For Alice to pay Dina, Alice will need Dina’s node to generate a Lightning invoice. We will discuss this in more detail in [bolt11_invoices]. For now, let’s assume that Dina has a website that can produce a Lightning invoice for tips.

TIP Lightning payments can be sent without an invoice, using a feature called *keysend*, which we will discuss in more detail in [keysend]. For now, we will explain the simpler payment flow using an invoice.

Alice visits Dina’s site, enter the amount of 50,000 satoshis in a form and in response, Dina’s Lightning node generate a payment request for 50,000 satoshis in the form of a Lightning invoice.

Alice requests an invoice from Dina’s website



As we saw in previous examples, we assume that Alice does not have a direct payment channel to Dina. Instead, Alice has a channel to Bob, Bob has a channel to Chan and Chan has a channel to Dina. To pay Dina, Alice must find a path that connects her to Dina. We will discuss that step in more detail in [path_finding]. For now, let’s assume that Alice is able to gather information about available channels and sees that there is a path from her to Dina, via Bob and Chan.

Now, Alice’s node can construct a Lightning payment. In the next few sections we will see how Alice’s node constructs a Hash Time-Locked Contract (HTLCs) to pay Dina and how that HTLC is forwarded along the path from Alice to Dina.

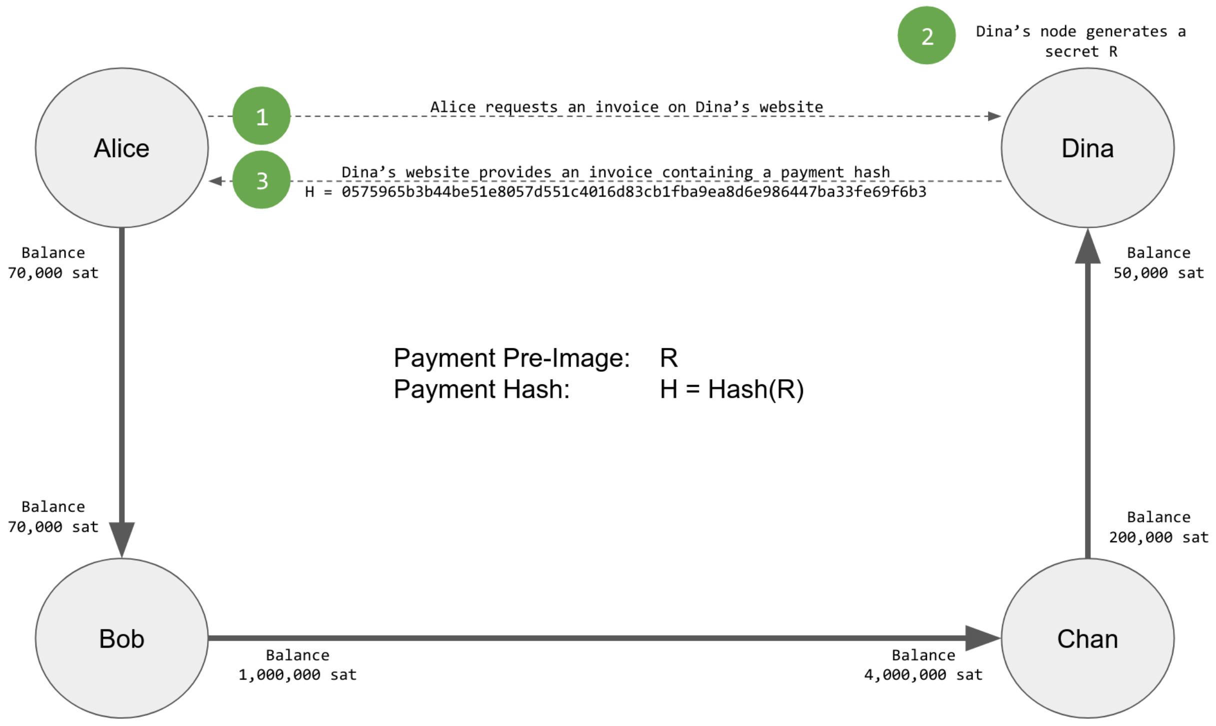
Hash Time Locked Contracts (HTLCs)

In this section we explain how Hash Time Locked Contracts (HTLCs) work.

The first part of a Hash Time-Locked Contract is the "Hash". This refers to the use of a cryptographic hash algorithm to commit to a randomly generated secret. Knowledge of the secret allows redemption of the payment. The cryptographic hash function, guarantees that while it’s infeasible for anyone to guess the secret pre-image, it’s easy for anyone to verify the hash, and there’s only one possible pre-image that resolves the payment condition.

Alice has a Lightning invoice from Dina. Inside that invoice Dina has encoded a *payment hash*, which is the cryptographic hash of a secret that Dina’s node produced. Dina’s secret is called the *payment pre-image*. The payment hash acts as an identifier that can be used to route the payment to Dina. The payment pre-image acts as a receipt and proof of payment once the payment is complete.

Alice gets a payment hash from Dina


2
Dina’s node generates a secret R
Alice
1
Alice requests an invoice on Dina’s website
Dina
3
Dina’s website provides an invoice containing a payment hash
H = 0575965b3b44be51e8057d551c4016d83cb1fba9ea8d6e986447ba33fe69f6b3
Balance 70,000 sat
Balance 50,000 sat
Payment Pre-Image: R
Payment Hash: H = Hash(R)
Balance 70,000 sat
Balance 200,000 sat
Bob
Balance 1,000,000 sat
Balance 4,000,000 sat
Chan

In the Lightning Network, Dina’s payment pre-image won’t be a phrase like "Dina’s secret", but a random number generated by Dina’s node. Let’s call that random number R.

Dina’s node will calculate a cryptographic hash of R, such that:

Calculating the payment hash

```
H = SHA256(R)
```

In Calculating the payment hash H is the hash, or *payment hash* and R is the secret or *payment pre-image*.

The use of a cryptographic hash function is one element that guarantees *trustless operation*. The payment intermediaries do not need to trust each other because they know that no one can guess the secret or fake it.

HTLCs in Bitcoin Script

In our gold coin example, Alice had a contract enforced by escrow like this:

> *Alice will reimburse Bob with 12 gold coins if you can show a valid message that hashes to:* 0575...f6b3. *Bob has 24 hours to show the secret after the contract was signed. If Bob does not provide the secret by this time, Alice's deposit will be refunded by the escrow service and the contract becomes invalid.*

Let's see how we would implement this as an HTLC in Bitcoin Script. In HTLC implemented in Bitcoin Script (BOLT3) we see an HTLC Bitcoin Script as currently used in the Lightning Network. You can find this definition in BOLT3 - Transactions:

HTLC implemented in Bitcoin Script (BOLT3)

```
# 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
```

Wow that looks complicated! Don't worry though, we will take it one step at a time and simplfy it.

The Bitcoin Script currently used in the Lightning Network is quite complex because it is optimized for on-chain space efficiency, which makes very compact but difficult to read.

In the following sections, we will focus on the main elements of the script and present simplified scripts that are slightly different from what is actually used in Lightning.

Payment pre-image and hash verification

The core of an HTLC is the "hash", where payment can be made if the recipient knows the payment pre-image. Alice locks the payment to a specific payment hash and Bob has to present a payment pre-image to claim the funds. The Bitcoin system can verify that Bob's payment pre-image is correct by hashing it and comparing the result to the payment hash that Alice used to lock the funds.

This part of an HTLC can be implemented in Bitcoin Script as follows:

```
OP_SHA256 <H> OP_EQUAL
```

Alice can create a transaction output that pays, 50,200 satoshi with a locking script above, replacing <H> with the hash value 0575...f6b3 provided by Dina. Then, Alice can sign this transaction and offer it to Bob:

Alice's offers a 50,200 satoshi HTLC to Bob

```
OP_SHA256 0575...f6b3 OP_EQUAL
```

Bob can't spend this HTLC until he knows Dina's secret, so spending the HTLC is conditional on Bob's fullfilment of the payment all the way to Dina.

Once Bob has Dina's secret, Bob can spend this output with an unlocking script containing the secrect pre-image value R

The unlocking script and locking script would combined to produce:

```
<R> OP_SHA256 <H> OP_EQUAL
```

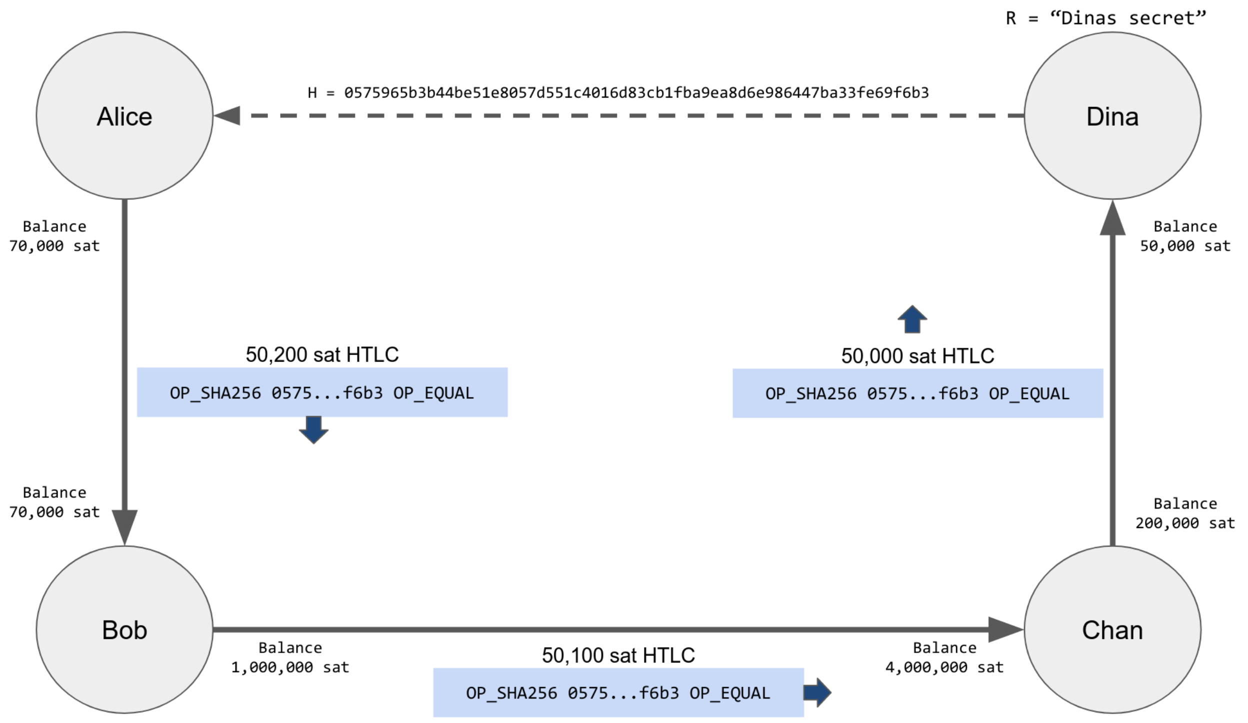
The Bitcoin Script engine would evaluate this script as follows:

1. R is pushed to the stack
2. The OP_SHA256 operator takes the value R off the stack and hashes it, pushing the result H_R to the stack
3. H is pushed to the stack
4. The OP_EQUAL operator compares H and H_R. If they are equal, the result is TRUE, the script is complete and the payment is verified.

Extending HTLCs from Alice to Dina

Alice will now extend the HTLC across the network so that it reaches Dina.

Propagating the HTLC across the network



Alice has given Bob an HTLC for 50,200 satoshi. Bob can now create an HTLC for 50,100 satoshi and give it to Chan. Bob knows that Chan can’t redeem Bob’s HTLC without broadcasting the secret, at which point Bob can also use the secret to redeem Alice’s HTLC.

Since Alice’s HTLC is 100 satoshi more that the HTLC Bob gave to Chan, Bob will earn 100 satoshi as a routing fee if this payment completes.

Bob isn’t taking a risk and isn’t trusting Alice or Chan. Instead, Bob is trusting that a signed transaction together with the secret will be redeemable on the Bitcoin blockchain.

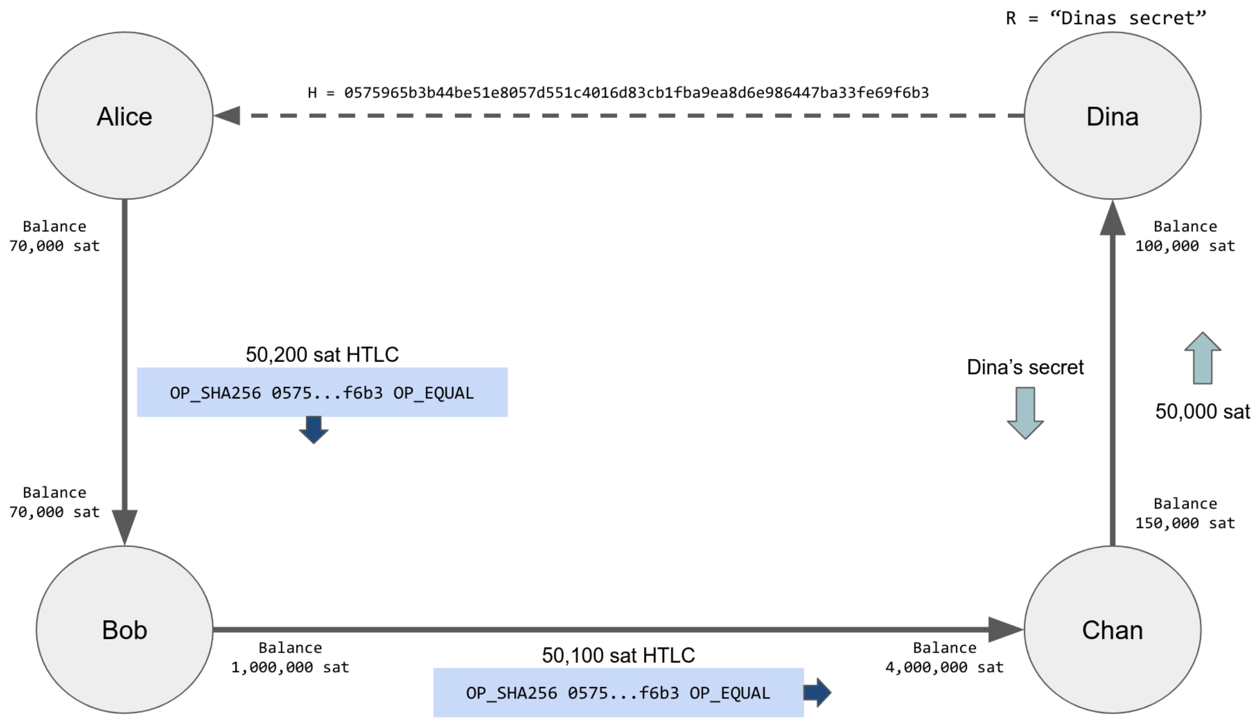
Similarly, Chan can extend a 50,000 HTLC to Dina. He isn’t risking anything or trusting Bob or Dina. To redeem the HTLC, Dina would have to broadcast the secret, which Chan could use to redeem Bob’s HTLC. Chan would also earn 100 satoshis as a routing fee.

Back-propagating the secret

Once Dina receives a 50,000 HTLC from Chan, she can now get paid. Dina could simply commit this HTLC on-chain and spend it by revealing the secret in the spending transaction. Or, instead, Dina can update the channel balance with Chan by giving him the secret. There’s no reason to incur a transaction fee and go on-chain. So, instead, Dina sends the secret to Chan and they agree to update their channel balances to reflect a 50,000 satoshi Lightning payment to Dina.

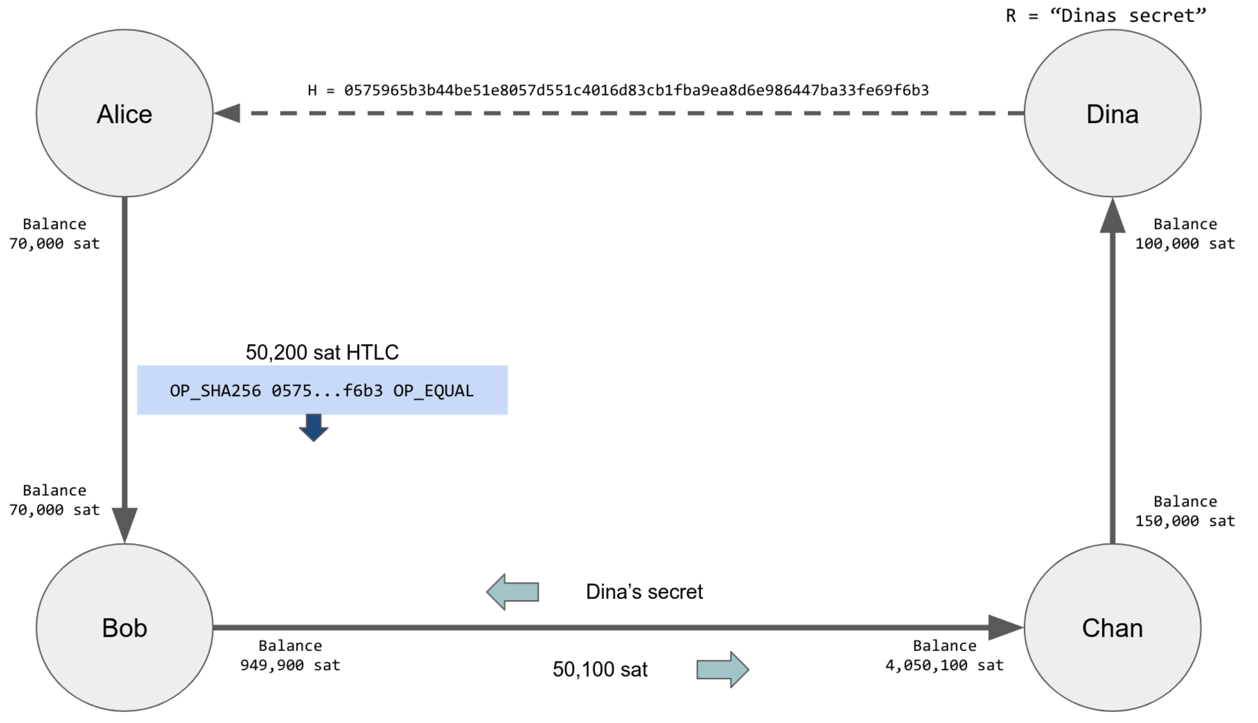
Notice Dina’s channel balance goes from 50,000 satoshi to 100,000 satoshi. Chan’s channel balance is reduced from 200,000 satoshi to 150,000 satoshi. The channel capacity hasn’t changed, but 50,000 has moved from Chan’s side of the channel to Dina’s side of the channel.

Dina settles Chan’s HTLC off-chain



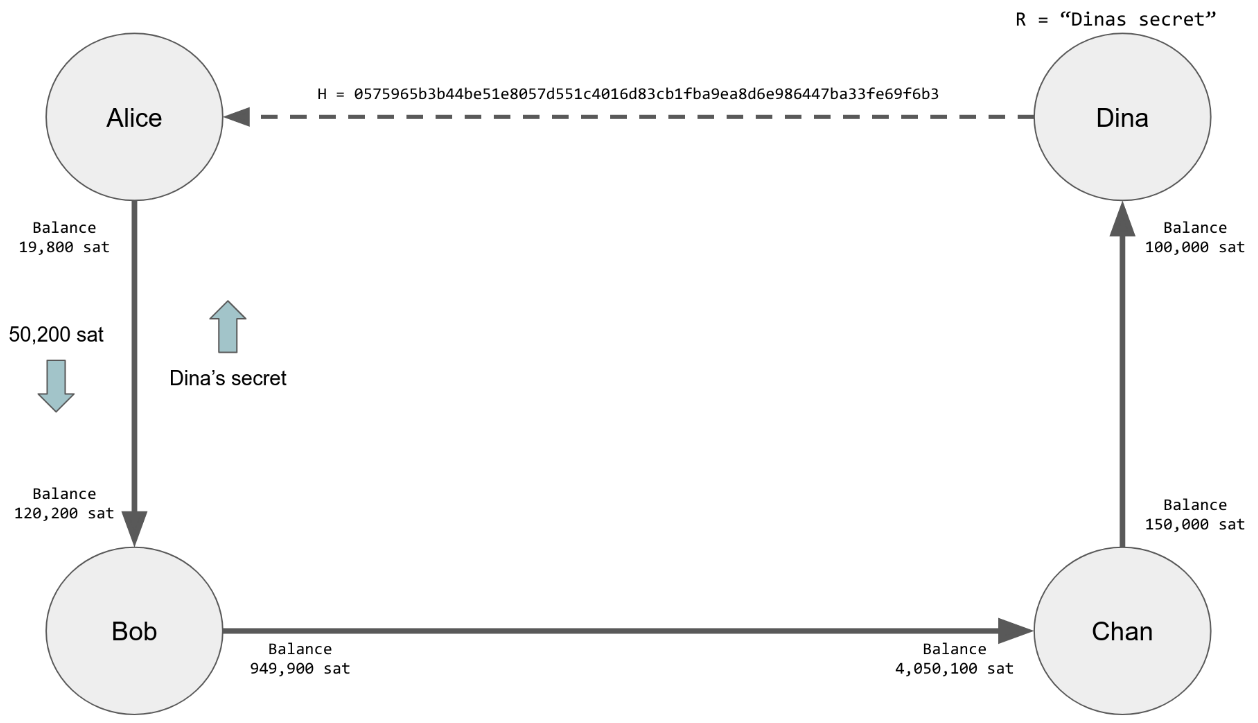
Chan now has the secret and has paid Dina 50,000 satoshi. He can do this without any risk, because the secret allows Chan to redeem the 50,100 HTLC from Bob. Chan has the option to commit that HTLC on chain and spend it by revealing the secret on the Bitcoin blockchain. But, like Dina, he'd rather avoid transaction fees. So instead, he sends the secret to Bob so they can update their channel balances to reflect a 50,100 satoshi Lightning payment from Bob to Chan. Chan has paid Dina 50,000 satoshi, and received 50,100 satoshi from Bob. So Chan has 100 satoshi more in his channel balances, which he earned as a routing fee.

Chan settles Bob's HTLC off-chain



Bob now has the secret too. He can use it to spend Alice's HTLC on-chain. Or, he can avoid transaction fees by settling the HTLC in the channel with Alice. Bob sends the secret to Alice and they update the channel balance to reflect a 50,200 satoshi Lightning payment from Alice to Bob. Bob has recieved 50,200 satoshi from Alice and paid 50,100 satoshi to Chan, so he has an extra 100 satoshi in his channel balances from routing fees.

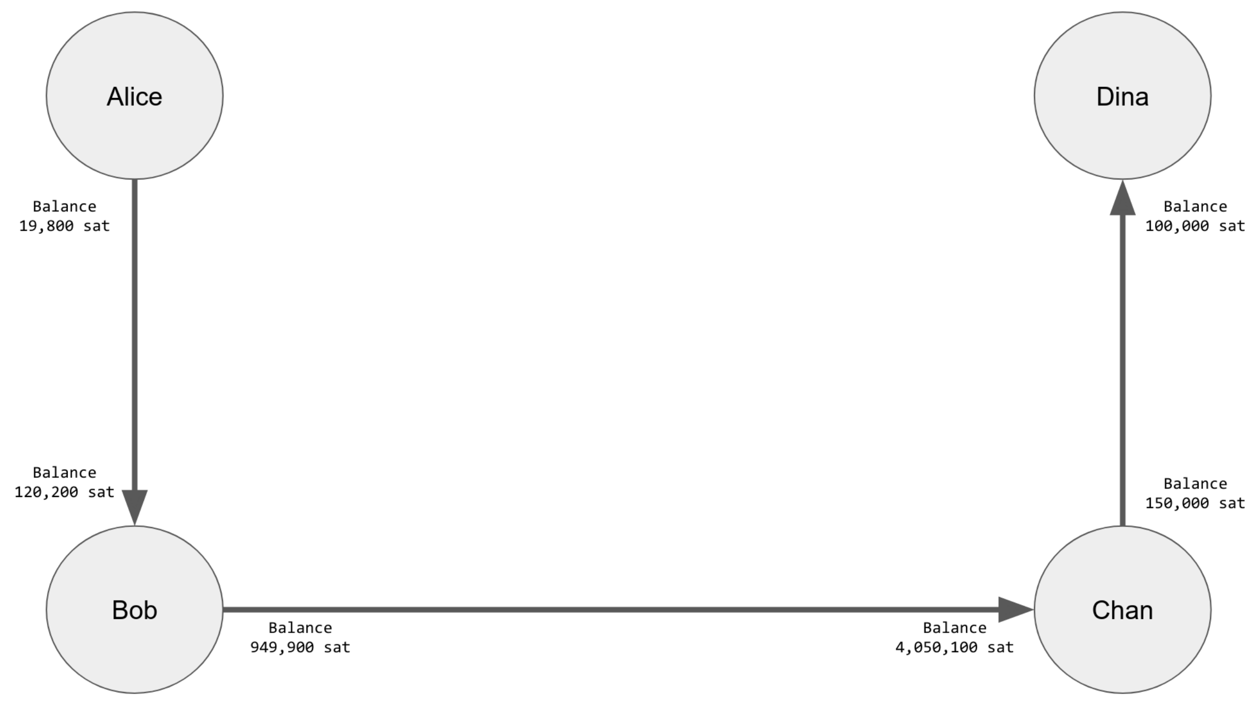
Bob settles Alice's HTLC off-chain



Alice receives the secret and has settled the 50,200 satoshi HTLC. The secret can be used as a *receipt* to prove that Dina got paid for that specific payment hash.

The final channel balances reflect Alice's payment to Dina and the routing fees paid at each hop

Channel balances after the payment



Preventing theft of HTLCs

HTLC timeout (failure)

```
OP_DROP <cltv_expiry> OP_CHECKLOCKTIMEVERIFY OP_DROP
OP_CHECKSIG
```

Incremental timelocks

In order to implement the "refund" functionality, we rely on the "absolute time lock" functionality of Bitcoin script.

Alice can present this script to Bob in order to kick off the conditional payment. For the chained aspect, Alice needs to be able to communicate the proper payment details to each hop in the route. Recall that each hop will specify a forwarding fee rate, as well as other parameters that express their forwarding policy. In addition to this forwarding rate, Alice also needs to be concerned about what time locks to use. Each node in the hop needs some time to be able to settle the outgoing, then incoming payment on-chain in the worst case. As a result, when constructing the final route, we need to give each node some buffer time, we call this before time, the "time lock delta". Factoring in this time-lock delta, the time-lock of the outgoing HTLC will decrease as the route progresses, as the outgoing HTLC will expire before the incoming HTLC. This set of decrementing time-locks is critical to the operation of the system, as it ensure out atomicity property for each hop, assuming they're able to get into the chain in time.

In the next section, we'll go into the exact mechanism of how Alice is able to deliver forwarding details to each hop in the route. In addition, we'll dive further into proper time-lock construction, as incorrect time-lock set up can violate our atomicity property and lead to a loss of funds.

[1] You can verify this by typing `echo -n "Dinas secret" | sha256sum` to your Linux command line shell.