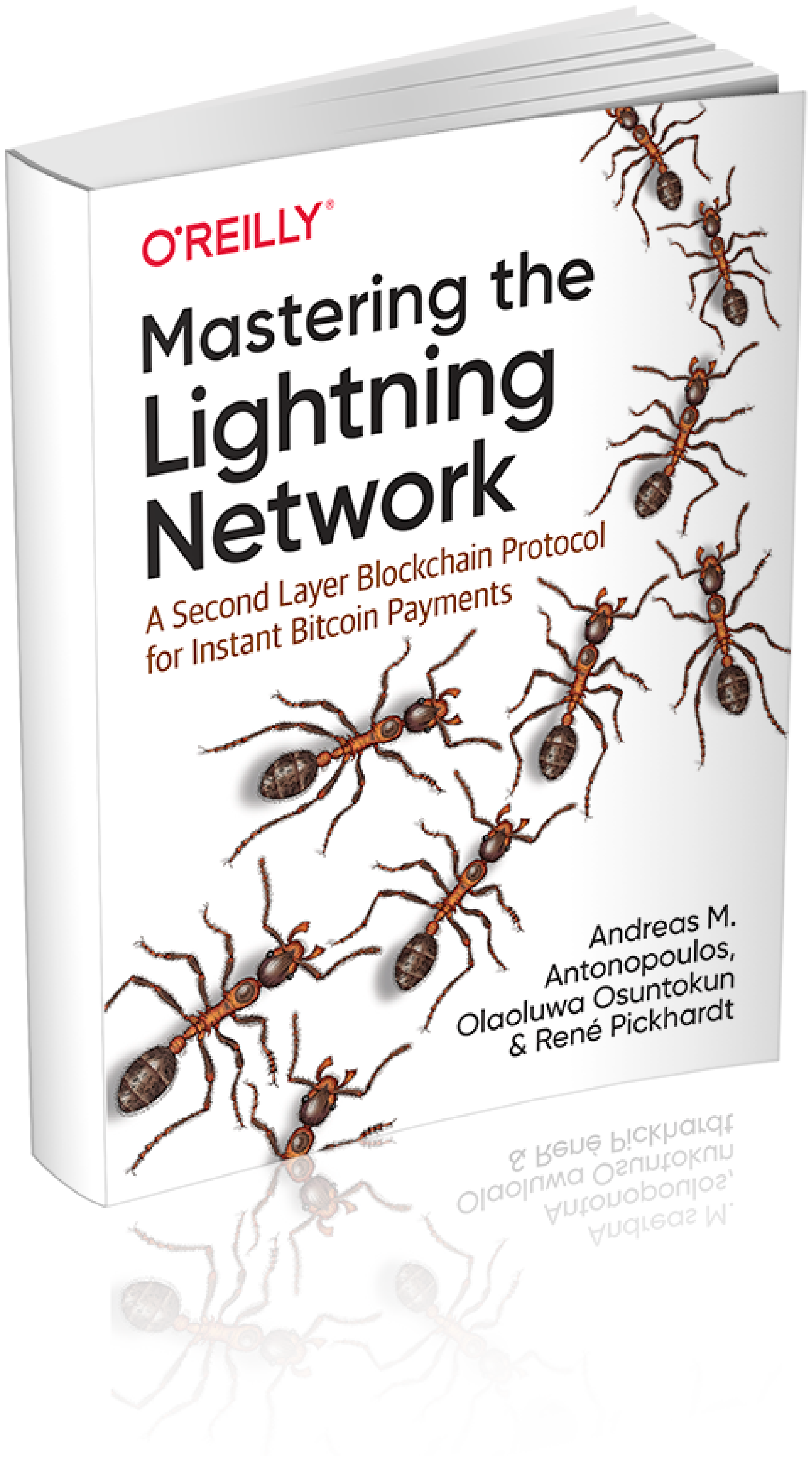
Mastering Lightning


O'REILLY
Mastering the Lightning Network
A Second Layer Blockchain Protocol for Instant Bitcoin Payments
Andreas M. Antonopoulos, Olaoluwa Osuntokun & René Pickhardt

Mastering the Lightning Network

A Second Layer Blockchain Protocol for Instant Bitcoin Payments

by Andreas M. Antonopoulos, Olaoluwa Osuntokun, and René Pickhardt

Printed in the United States of America.

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

- Acquisitions Editor: Michelle Smith
- Development Editor: Michele Cronin
- Production Editor: Kristen Brown
- Copyeditor: Tom Sullivan
- Proofreader: Sonia Saruba
- Indexer: WordCo Indexing Services, Inc.
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea
- December 2021: First Edition

Revision History for the First Edition

- 2021-11-22: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492054863 for release details.

ISBN-13: 978-1492054863

ISBN-10: 1492054860

About the Authors

Andreas M. Antonopoulos is a best-selling author, speaker, educator, and highly sought after expert in Bitcoin and open blockchain technologies. He is known for making complex subjects easy to understand and highlighting both the positive and negative impacts these technologies can have on our global societies.

Andreas has written two more best-selling technical books for programmers with O'Reilly Media, *Mastering Bitcoin* and *Mastering Ethereum*. He has also published *The Internet of Money* series of books, which focus on the social, political, and economic importance and implications of these technologies. Andreas produces free educational content on his YouTube channel weekly and teaches virtual workshops on his website. Learn more at aantonop.com.

Olaoluwa Osuntokun is the cofounder and CTO of Lightning Labs, and also the lead developer of lnd, one of the main implementations of Lightning. He received his BS and MS in Computer Science from UCSB and was a member of the Forbes 30 Under 30 class of 2019. During his graduate studies he focused on the field of applied cryptography, specifically encrypted search. He has been an active Bitcoin developer for over five years, and is an author of several Bitcoin Improvement Proposals (BIP-157 and 158). These days, his primary focus lies in building, designing, and evolving private, scalable off-chain blockchain protocols, such as Lightning.

René Pickhardt is a trained mathematician and data science consultant who uses his statistical knowledge to do research with NTNU about pathfinding, privacy, reliability of payments, and service-level agreements of the Lightning Network. René maintains a technical and developer-oriented YouTube channel about the Lightning Network and has answered roughly half of the questions about the Lightning Network on the Bitcoin Stack Exchange, making him the go-to point for almost all new developers who want to join the space. René has held numerous workshops about the Lightning Network in public and private, including teaching students of the 2019 Chaincode Labs residency together with other core Lightning developers.

Preface

The Lightning Network (LN) is a second layer peer-to-peer network that allows us to make Bitcoin payments "off-chain," meaning without committing them as transactions to the Bitcoin blockchain.

The Lightning Network gives us Bitcoin payments that are secure, cheap, fast, and much more private, even for very small payments.

Building on the idea of payment channels, first proposed by Bitcoin's inventor Satoshi Nakamoto, the Lightning Network is a routed network of payment channels where payments "hop" across a path of payment channels from the sender to the recipient.

The initial idea of the Lightning Network was proposed in 2015 in the groundbreaking paper "The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments," by Joseph Poon and Thaddeus Dryja. By 2017, there was a "test" Lightning Network running on the internet, as different groups built compatible implementations and coordinated to set some interoperability standards. In 2018, the Lightning Network went "live" and payments started flowing.

In 2019, Andreas M. Antonopoulos, Olaoluwa Osuntokun, and René Pickhardt agreed to collaborate to write this book. It appears we have been successful!

How to Use This Book

Intended Audience

This book is mostly intended for technical readers with an understanding of the fundamentals of Bitcoin and other open blockchains.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`

Shows commands or other text that should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or by values determined by context.

TIP This element signifies a tip or suggestion.

NOTE This element signifies a general note.

WARNING This element indicates a warning or caution.

Code Examples

The examples are illustrated in Go, C++, Python, and using the command line of a Unix-like operating system. All code snippets are available in the GitHub repository under the *code* subdirectory. Fork the book code, try the code examples, or submit corrections via GitHub.

All the code snippets can be replicated on most operating systems with a minimal installation of compilers, interpreters, and libraries for the corresponding languages. Where necessary, we provide basic installation instructions and step-by-step examples of the output of those instructions.

Some of the code snippets and code output have been reformatted for print. In all such cases, the lines have been split by a backslash (\) character, followed by a newline character. When transcribing the examples, remove those two characters and join the lines again and you should see identical results to those shown in the example.

All the code snippets use real values and calculations where possible, so that you can build from example to example and see the same results in any code you write to calculate the same values. For example, the private keys and corresponding public keys and addresses are all real.

Using Code Examples

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, ISBN, and copyright. For example: "*Mastering the Lightning Network* by Andreas M. Antonopoulos, Olaoluwa Osuntokun, and René Pickhardt (O'Reilly). Copyright 2022 aantonop Books LLC, René Pickhardt, and uuddlrlrbas LLC, ISBN 978-1-492-05486-3."

Mastering the Lightning Network is offered under the Creative Commons Attribution-Noncommercial-No Derivative Works 4.0 International License (CC BY-NC-ND 4.0).

If you feel your use of code examples falls outside fair use or the permission given previously, feel free to contact us at permissions@oreilly.com.

References to Companies and Products

All references to companies and products are intended for educational, demonstration, and reference purposes. The authors do not endorse any of the companies or products mentioned. We have not tested the operation or security of any of the products, projects, or code segments shown in this book. Use them at your own risk!

Addresses and Transactions in This Book

The Bitcoin addresses, transactions, keys, QR codes, and blockchain data used in this book are, for the most part, real. That means you can browse the blockchain, look at the transactions offered as examples, retrieve them with your own scripts or programs, etc.

However, note that the private keys used to construct the addresses printed in this book have been "burned." This means that if you send money to any of these addresses, the money will either be lost forever or (more likely) appropriated, since anyone who reads the book can take it using the private keys printed herein.

WARNING DO NOT SEND MONEY TO ANY OF THE ADDRESSES IN THIS BOOK. Your money will be taken by another reader, or lost forever.

O’Reilly Online Learning

NOTE For more than 40 years, *O’Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O’Reilly’s online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O’Reilly and 200+ other publishers. For more information, visit *http://oreilly.com*.

How to Contact Us

Information about *Mastering the Lightning Network* as well as the Open Edition and translations are available at https://lnbook.info/.

Please address comments and questions concerning this book to the publisher:

```
O Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)
```

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and information about our books and courses, visit http://oreilly.com.

Find us on Facebook: http://facebook.com/oreilly

Follow us on Twitter: http://twitter.com/oreillymedia

Watch us on YouTube: http://www.youtube.com/oreillymedia

Contacting Andreas

You can contact Andreas M. Antonopoulos on his personal site: https://aantonop.com/

Subscribe to Andreas's channel on YouTube: https://www.youtube.com/aantonop

Like Andreas's page on Facebook: https://www.facebook.com/AndreasMAntonopoulos

Follow Andreas on Twitter: https://twitter.com/aantonop

Connect with Andreas on LinkedIn: https://linkedin.com/company/aantonop

Andreas would also like to thank all of the patrons who support his work through monthly donations. You can support Andreas on Patreon at https://patreon.com/aantonop.

Contacting René

You can contact René Pickhardt on his personal site: https://ln.rene-pickhardt.de/

Subscribe to René's channel on YouTube: https://www.youtube.com/user/RenePickhardt

Follow René on Twitter: https://twitter.com/renepickhardt

Connect with René on LinkedIn: https://www.linkedin.com/in/rene-pickhardt-80313744/

René would also like to thank all of the patrons who support his work through monthly donations. You can support René on Patreon at https://patreon.com/renepickhardt.

Or you can support his work directly with Bitcoin (also via the Lightning Network) at https://donate.ln.rene-pickhardt.de for which René is equally thankful as for his patreons.

Contacting Olaoluwa Osuntokun

You can contact Olaoluwa Osuntokun at his professional email address: *laolu@lightning.engineering*

Follow Olaoluwa on Twitter: https://twitter.com/roasbeef

Acknowledgments By Andreas

I owe my love of words and books to my mother, Theresa, who raised me in a house with books lining every wall. My mother also bought me my first computer in 1982, despite being a self-described technophobe. My father, Menelaos, a civil engineer who published his first book at 80 years old, was the one who taught me logical and analytical thinking and a love of science and engineering.

Thank you all for supporting me throughout this journey.

Acknowledgments By René

I want to thank the German education system through which I acquired the knowledge upon which my work builds. It is one of the greatest gifts I was given. Similarly I want to thank the German public healthcare system and every person devoting their time into working within that industry. Your effort and endurance make you my personal heroes and I will never forget the help, patience, and support I received when I was in need. Thanks goes to all the students I was allowed to teach and who engaged in interesting discussions and questions. From you I learned the most. I am also grateful to the Bitcoin and Lightning Network community that warmly welcomed me and to the enthusiasts and private persons who financially supported and continue to support my work. In particular I am grateful to all the open source developers (not only Bitcoin and Lightning Network) and to the people who fund them to make that technology possible. A special thanks goes to my coauthors for riding with me through the storm. Last but not least, I am thankful to my loved ones.

Acknowledgments by Olaoluwa Osuntokun

I'd like to thank the amazing team at Lightning Labs, as without you all, there would be no LND. I'd also like to thank the original set of authors of the BOLT specification: Rusty Russell, Fabrice Drouin, Conner Fromnkchet, Pierre-Marie Padiou, Lisa Neigut, and Christian Decker. Last but not least, I'd like to thank Joseph Poon and Tadge Dryja, the authors of the original Lightning Network paper, as without them, there would be no Lightning Network to write a book about.

Contributions

Many contributors offered comments, corrections, and additions to the book as it was collaboratively written on GitHub.

Following is an alphabetically sorted list of all the GitHub contributors, including their GitHub IDs in parentheses:

- 8go (@8go)

- Aaqil Aziz (@batmanscode)
- Alexander Gnip (@quantumcthulhu)
- Alpha Q. Smith (@alpha_github_id)
- Ben Skee (@benskee)
- Brian L. McMichael (@brianmcmichael)
- CandleHater (@CandleHater)
- Daniel Gockel (@dancodery)
- Dapeng Li (@luislee818)
- Darius E. Parvin (@DariusParvin)
- Doru Muntean (@chriton)
- Eduardo Lima III (@elima-iii)
- Emilio Norrmann (@enorrmann)
- Francisco Calderón (@grunch)
- Francisco Requena (@FrankyFFV)
- François Degros (@fdegros)
- Giovanni Zotta (@GiovanniZotta)
- Gustavo Silva (@GustavoRSSilva)
- Guy Thayakorn (@saguywalker)
- Haoyu Lin (@HAOYUatHZ)
- Hatim Boufnichel (@boufni95)
- Imran Lorgat (@ImranLorgat)
- Jeffrey McLarty (@jnmclarty)
- John Davies (@tigeryant)
- Julien Wendling (@trigger67)
- Jussi Tiira (@juhi24)
- Kory Newton (@korynewton)
- Lawrence Webber (@lwebbz)
- Luigi (@gin)
- Maximilian Karasz (@mknoszlig)
- Omega X. Last (@omega_github_id)
- Owen Gunden (@ogunden)
- Patrick Lemke (@PatrickLemke)
- Paul Wackerow (@wackerow)
- Randy McMillan (@RandyMcMillan)
- René Köhnke (@rene78)

- Ricardo Marques (@RicardoM17)
- Sebastian Falbesoner (@theStack)
- Sergei Tikhomirov (@s-tikhomirov)
- Severin Alexander Bühler (@SeverinAlexB)
- Simone Bovi (@SimoneBovi)
- Srijan Bhushan (@srijanb)
- Taylor Masterson (@tjmasterson)
- Umar Bolatov (@bolatovumar)
- Warren Wan (@wlwanpan)
- Yibin Zhang (@z4y1b2)
- Zachary Haddenham (@senf42)

Without the help offered by everyone listed here, this book would not have been possible. Your contributions demonstrate the power of open source and open culture, and we are eternally grateful for your help.

Thank you.

Sources

Some of the material in this book has been sourced from a variety of public domain sources, open license sources, or with permission. See Sources and License Notices for source, license, and attribution details.

Glossary

This quick glossary contains many of the terms used in relation to Bitcoin and the Lightning Network. These terms are used throughout the book, so bookmark this for a quick reference.

address

Bitcoin addresses compactly encode the information necessary to pay a receiver. A modern address consists of a string of letters and numbers that starts with bc1 and looks like bc1qw508d6qejxtdg4y5r3zarvary0c5xw7kv8f3t4. An address is shorthand for a receiver's locking script, which can be used by a sender to sign over funds to the receiver. Most addresses either represent the receiver's public key or some form of script that defines more complex spending conditions. The preceding example is a bech32 address encoding a witness program locking funds to the hash of a public key (See *Pay-to-Witness-Public-Key-Hash*). There are also older address formats that start with 1 or 3 that use the Base58Check address encoding to represent public key hashes or script hashes.

asymmetric cryptographic system

Asymmetric cryptography, or public-key cryptography, is a cryptographic system that uses pairs of keys: public keys which may be disseminated widely, and private keys which are known only to the owner. The generation of such keys depends on cryptographic algorithms based on mathematical problems to produce functions that are easy to solve one way, but very difficult to solve in reverse. Effective security only requires keeping the private key private; the public key can be openly distributed without compromising security.

autopilot

An autopilot is a recommendation engine for Lightning nodes that uses statistics of the Lightning Network topology to suggest which nodes they should open channels with. Depending on the implementation of the autopilot, the channel capacity may also be recommended. An autopilot is not part of the LN Protocol.

balance

The balance of a payment channel is the amount of bitcoin that belongs to each channel partner. For example, Alice could open a channel with Bob for the value of 1 BTC. The channel balance is then 1 BTC to Alice and 0 BTC to Bob. As the users transact, the channel balance will update. For example, if Alice sends 0.2 BTC to Bob, then the balance is now 0.8 BTC to Alice and 0.2 to Bob. When the channel is closed, the bitcoin in the channel will be divided between the two channel partners according to the latest balance encoded in the commitment transaction. In the Lightning Network, the ability to send and receive payments is limited by channel balances. See *capacity*.

bech32

bech32 refers to a generic check-summed base32-encoded format featuring strong error-detection guarantees. While bech32 was originally developed to be used as the address format for native SegWit outputs (BIP-173), it is also used to encode lightning invoices (BOLT #11). While native SegWit version 0 outputs (P2WPKH and P2WSH) use bech32, higher native SegWit output versions (e.g., Pay-to-Taproot or P2TR) use the improved variant bech32m (BIP-350). bech32m addresses are sometimes referred to as "bc1" addresses, reflecting the prefix of such addresses.

Native SegWit outputs are more blockspace-efficient than older addresses and therefore may reduce transaction fees for the owner of such an address.

Bitcoin Improvement Proposal (BIP)

A proposal that members of the Bitcoin community have submitted to improve Bitcoin. For example, BIP-21 is a proposal to improve the Bitcoin uniform resource identifier (URI) scheme. BIPs can be found at GitHub.

bitcoin, Bitcoin

Depending on the context, could refer to the name of the currency unit (the coin), the network, or the underlying enabling protocol. Written as bitcoin with a lowercase "b" usually refers to the currency unit. Bitcoin with an uppercase "B" usually refers to the protocol or system.

Bitcoin mining

Bitcoin mining is the process of constructing a block from recent Bitcoin transactions and then solving a computational problem required as proof of work. It is the process by which the shared bitcoin ledger (i.e., the Bitcoin blockchain) is updated and by which new transactions are included in the ledger. It is also the process by which new bitcoin is issued. Every time a new block is created, the mining node will receive new bitcoin created within the coinbase transaction of that block.

block

A block is a data structure in the Bitcoin blockchain that consists of a header and a body of Bitcoin transactions. The block is marked with a timestamp and commits to a specific predecessor (parent) block. When hashed, the block header provides the proof of work that makes the blockchain probabilistically immutable. Blocks must adhere to the rules enforced by network consensus to extend the blockchain. When a block is appended to the blockchain, the included transactions are considered to have their first confirmation.

blockchain

The blockchain is a distributed log, or database, of all Bitcoin transactions. Transactions are grouped in discrete updates called blocks, limited up to 4 million weight units. Blocks are produced approximately every 10 minutes via a stochastic process called mining. Each block includes a computationally intensive "proof of work." The proof of work requirement is used to regulate the block intervals and protect the blockchain against attacks to rewrite history: an attacker would need to outdo existing proof of work to replace already published blocks, making each block probabilistically immutable as it is buried under subsequent blocks.

BOLT

BOLT, or Basis of Lightning Technology, is the formal specification of the Lightning Network. Unlike Bitcoin, which has a reference implementation that also serves as the protocol’s specification, the various LN implementations follow BOLT so they can work with one another to form the same network. It is available at GitHub.

capacity

The capacity of a payment channel is equivalent to the amount of bitcoin provided by the funding transaction. Because the funding transaction is publicly visible on the blockchain, and the channel is announced via the gossip protocol, the capacity is public information. It does not reveal any information about how much bitcoin each of the channel partners owns in the channel, i.e., the balance. A high capacity does not guarantee that the channel can be used for routing in both directions.

c-lightning

Implementation of the LN Protocol by the Victoria-based company Blockstream. It is written in C. Source code is at GitHub.

closing transaction

If both channel partners agree to close a channel, they will create a settlement transaction that reflects the most recent commitment transaction. After exchanging signatures for a closing transaction, no further channel updates should be made. Mutually closing a channel with the help of a closing transaction has the advantage that fewer blockchain transactions are required to claim all funds, in comparison to unilaterally forcing a channel close by publishing a commitment transaction. Additionally, funds for both parties are immediately spendable from a closing transaction.

CLTV

CLTV is an acronym/abbreviation for the Bitcoin Script operator OP_CHECKLOCKTIMEVERIFY. This defines an absolute blockheight before an output can be spent. The atomicity of the routing process heavily depends on CLTV values in HTLCs. Routing nodes announce, via the gossip protocol, their expected CLTV expiry deltas that they wish for any incoming and outgoing HTLCs.

coinbase

The coinbase is a special field only permitted in the sole input of coinbase transactions. The coinbase allows up 100 bytes of arbitrary data, but since BIP-34, it must first feature the current block height to ensure that coinbase transactions are unique. Not to be confused with coinbase transaction.

coinbase transaction

The first transaction in a block which is always created by a miner and which includes a single coinbase. The coinbase transaction may claim the block reward and assign it to one or more outputs. The block reward consists of the block subsidy (newly created bitcoin) and the sum of all transaction fees from transactions included in the block. Coinbase outputs can only be spent after maturing for 100 blocks. If the block includes any SegWit transactions, the coinbase transaction must include a commitment to the witness transaction identifiers in an additional output.

cold storage

Refers to keeping an amount of bitcoin offline. Cold storage is achieved when Bitcoin private keys are created and stored in a secure offline environment. Cold storage is important to protect bitcoin holdings. Online computers are vulnerable to hackers and should not be used to store a significant amount of bitcoin.

commitment transaction

A commitment transaction is a Bitcoin transaction, signed by both channel partners, that encodes the latest balance of a channel. Every time a new payment is made or forwarded using the channel, the channel balance will update, and a new commitment transaction will be signed by both parties. Importantly, in a channel between Alice and Bob, both Alice and Bob keep their own version of the commitment transaction, which is also signed by the other party. At any point, the channel can be closed by either Alice or Bob if they submit their commitment transaction to the Bitcoin blockchain. Submitting an older (outdated) commitment transaction is considered *cheating* (i.e., a protocol breach) in the Lightning Network and can be penalized by the other party, claiming all the funds in the channel for themselves, via a penalty transaction.

confirmations

Once a transaction is included in a block, it has one confirmation. As soon as *another* block is mined on the blockchain, the transaction has two confirmations, and so on. Six or more confirmations are considered sufficient proof that a transaction cannot be reversed.

contract

A contract is a set of Bitcoin transactions that together result in a certain desired behavior. Examples are RSMCs to create a trustless, bidirectional payment channel, or HTLCs to create a mechanism that allows trustless forwarding of payments through third parties.

Diffie–Hellman Key Exchange (DHKE)

On the Lightning Network, the Elliptic Curve Diffie–Hellman (ECDH) method is used. It is an anonymous key agreement protocol that allows two parties, each having an elliptic curve public-private key pair, to establish a shared secret over an insecure communication channel. This shared secret may be directly used as a key, or to derive another key. The key, or the derived key, can then be used to encrypt subsequent communications using a symmetric-key cipher. An example of the derived key would be the shared secret between the ephemeral session key of a sender of an onion with the node's public key of a hop of the onion, as described and used by the SPHINX Mix Format.

digital signature

A digital signature is a mathematical scheme for verifying the authenticity and integrity of digital messages or documents. It can be seen as a cryptographic commitment in which the message is not hidden.

double-spending

Double-spending is the result of successfully spending some money more than once. Bitcoin protects against double-spending by verifying that each transaction added to the blockchain adheres to the rules of consensus; this means checking that the inputs for the transaction have not been previously spent.

Elliptic Curve Digital Signature Algorithm (ECDSA)

Elliptic Curve Digital Signature Algorithm or ECDSA is a cryptographic algorithm used by Bitcoin to ensure that funds can only be spent by the holder of the correct private key.

Eclair

Implementation of the LN Protocol by the Paris-based company ACINQ. It is written in Scala. Source code is at GitHub.

encoding

Encoding is the process of converting a message into a different form. For example, converting a number from decimal to a hexadecimal.

Electrum server

An Electrum server is a Bitcoin node with an additional interface (API). It is often required by bitcoin wallets that do not run a full node. For example, these wallets check the status of specific transactions or broadcast transactions to the mempool using Electrum server APIs. Some Lightning wallets also use Electrum servers.

ephemeral key

Ephemeral keys are keys that are only used for a short time and not retained after use. They are often derived for use in one session from another key that is held long-term. Ephemeral keys are mainly used within the SPHINX Mix Format and onion routing on the Lightning Network. This increases the security of transported messages or payments. Even if an ephemeral key leaks, only information about a single session becomes public.

feature bits

A binary string that Lightning nodes use to communicate to each other which features they support. Feature bits are included in many Lightning messages as well as BOLT #11. They can be decoded using BOLT #9, and will tell nodes which features the node has enabled, and whether these are backward compatible. Also known as feature flags.

fees

In the context of the Lightning Network, nodes will charge routing fees for forwarding other users' payments. Individual nodes can set their own fee policies which will be calculated as the sum of a fixed base_fee and a fee_rate that depends on the payment amount. In the context of Bitcoin, the sender of a transaction pays a transaction fee to miners for including the transaction in a block. Bitcoin transaction fees do not include a base fee and depend linearly on the weight of the transaction, but not on the amount.

funding transaction

The funding transaction is used to open a payment channel. The value (in bitcoin) of the funding transaction is exactly the capacity of the payment channel. The output of the funding transaction is a 2-of-2 multisignature script (multisig) where each channel partner controls one key. Due to its multisig nature, it can only be spent by mutual agreement between the channel partners. It will eventually be spent by one of the commitment transactions or by the closing transaction.

global features (globalfeatures field)

Global features of a Lightning node are the features of interest for all other nodes. Most commonly they are related to supported routing formats. They are announced in the `init` message of the peer protocol as well as the `channel_announcement` and `node_announcement` messages of the gossip protocol.

gossip protocol

LN nodes send and receive information about the topology of the Lightning Network through gossip messages which are exchanged with their peers. The gossip protocol is mainly defined in BOLT #7 and defines the format of the `node_announcement`, `channel_announcement`, and `channel_update` messages. To prevent spam, node announcement messages will only be forwarded if the node already has a channel, and channel announcement messages will only be forwarded if the funding transaction of the channel has been confirmed by the Bitcoin network. Usually, Lightning nodes connect with their channel partners, but it is fine to connect with any other Lightning node to process gossip messages.

hardware wallet

A hardware wallet is a special type of Bitcoin wallet which stores the user's private keys in a secure hardware device. As of writing the book, hardware wallets are not available for LN nodes because the keys used by Lightning need to be online to participate in the protocol.

hash

A fixed-size digital fingerprint of some arbitrary-length binary input. Also known as a *digest*.

hash-based message authentication code (HMAC)

HMAC is an algorithm for verifying the integrity and authenticity of a message based on a hash function and a cryptographic key. It is used in onion routing to ensure the integrity of a packet at each hop, as well as within the Noise Protocol variant used for message encryption.

hash function

A cryptographic hash function is a mathematical algorithm that maps data of arbitrary size to a bit string of a fixed size (a hash) and is designed to be a one-way function, that is, a function that is infeasible to invert. The only way to recreate the input data from an ideal cryptographic hash function's output is to attempt a brute-force search of possible inputs to see if they produce a match.

hashlock

A hashlock is a Bitcoin Script spending condition that restricts the spending of an output until a specified piece of data is revealed. Hashlocks have the useful property that once any hashlock is revealed through spending, any other hashlocks secured using the same key can also be spent. This makes it possible to create multiple outputs that are all encumbered by the same hashlock and which all become spendable at the same time.

hash time-locked contract (HTLC)

A hash time-locked contract (HTLC) is a Bitcoin Script that consists of hashlocks and timelocks to require that the recipient of a payment either spends the payment prior to a deadline by presenting the hash preimage or the sender can claim a refund after the timelock expires. On the Lightning Network, HTLCs are outputs in the commitment transaction of a payment channel and are used to enable the trustless routing of payments.

invoice

The payment process on the Lightning Network is initiated by the recipient (payee) who issues an invoice, also known as a *payment request*. Invoices include the payment hash, the amount, a description, and the expiry time. Lightning invoices are defined in BOLT #11. Invoices can also include a fallback Bitcoin address to which the payment can be made in case no route can be found, as well as hints for routing a payment through a private channel.

just-in-time (JIT) routing

Just-in-time (JIT) routing is an alternative to source-based routing that was first proposed by coauthor René Pickhardt. With JIT routing, intermediary nodes along a path can pause an in-flight payment to rebalance their channels before proceeding with the payment. This might allow them to successfully forward payments that might otherwise have failed due to a lack of outgoing capacity.

Lightning message

A Lightning message is an encrypted data string that can be sent between two peers on the Lightning Network. Similar to other communication protocols, Lightning messages consist of a header and a body. The header and the body have their own HMAC. Lightning messages are the main building block of the messaging layer.

Lightning Network, Lightning Network Protocol, Lightning Protocol

The Lightning Network is a protocol on top of Bitcoin (or other cryptocurrencies). It creates a network of payment channels which enables the trustless forwarding of payments through the network with the help of HTLCs and onion routing. Other components of the Lightning Network are the gossip protocol, the transport layer, and payment requests.

Lightning Network protocol suite

The Lightning Network protocol suite consists of five layers that are responsible for various parts of the protocol. From bottom (the first layer) to the top (the fifth layer), these layers are called the network communication layer, the messaging layer, the peer-to-peer layer, the routing layer, and the payment layer. Various BOLTs define parts of one or several layers.

Lightning Network node, Lightning node

A computer participating in the Lightning Network, via the Lightning peer-to-peer protocol. Lightning nodes have the ability to open channels with other nodes, send and receive payments, and route payments from other users. Typically, a Lightning node user will also run a Bitcoin node.

lnd

Implementation of the LN Protocol by the San Francisco-based company Lightning Labs. It is written in Go. Source code is at GitHub.

local features (field: localfeatures)

Local features of an LN node are the configurable features of direct interest to its peers. They are announced in the `init` message of the peer protocol as well as in the `channel_announcement` and `node_announcement` messages of the gossip protocol.

locktime

Locktime, or more technically nLockTime, is the part of a Bitcoin transaction that indicates the earliest time or earliest block when that transaction may be added to the blockchain.

messaging layer

The messaging layer builds on top of the network connection layer of the Lightning Network protocol suite. It is responsible for ensuring an encrypted and secure communication and exchange of information via the chosen network connection layer protocol. The messaging layer defines the framing and format of Lightning Messages as defined in BOLT #1. The feature bits defined in BOLT #9 are also part of this layer.

millisatoshi

The smallest unit of account on the Lightning Network. A millisatoshi is one hundred billionth of a single bitcoin. A millisatoshi is one thousandth of one satoshi. Millisatoshis do not exist on, nor can they be settled on, the Bitcoin network.

multipart payments (MPP)

Multipart payments (MPP), often also referred to as multipath payments, are a method for splitting the payment amount into multiple smaller parts and delivering them along one or more paths. Since MPP can send many or all parts over a single path, the term multipart payment is more accurate than multipath payment. In computer science, multipart payments are modeled as network flows.

multisignature

Multisignature (multisig) refers to a script that requires more than one signature to authorize spending. Payment channels are always encoded as multisig addresses requiring one signature from each partner of the payment channel. In the standard case of a two-party payment channel, a 2-of-2 multisig address is used.

node

See *Lightning Network node*.

network capacity

LN capacity is the total amount of bitcoin locked and circulated inside the Lightning Network. It is the sum of capacities of each public channel. It reflects the usage of the Lightning Network to some extent because we expect that people put bitcoin into Lightning channels to spend it or forward other users' payments. Hence the higher the amount of bitcoin in Lightning channels, the higher the expected usage of the Lightning Network. Note that since only public channel capacity can be observed, the true network capacity is unknown. Also, since a channel’s capacity can enable an unlimited number of payments back and forth, network capacity does not imply a limit of value transferred on the Lightning Network.

network connection layer

The lowest layer of the Lightning Network protocol suite. Its responsibility is to support internet protocols like IPv4, IPv6, TOR2, and TOR3, and use them to establish a secure cryptographic communication channel as defined in BOLT #8, or to speak DNS for the bootstrapping of the network as defined in BOLT #10.

Noise_XK

The template of the Noise Protocol Framework to establish an authenticated and encrypted communication channel between two peers of the Lightning Network. X means that no public key needs to be known from the initiator of the connection. K means that the public key of the receiver needs to be known.

onion routing

Onion routing is a technique for anonymous communication over a computer network. In an onion network, messages are encapsulated in layers of encryption, analogous to layers of an onion. The encrypted data is transmitted through a series of network nodes called onion routers, each of which peels away a single layer, uncovering the data’s next destination. When the final layer is decrypted, the message arrives at its destination. The sender remains anonymous because each intermediary knows only the location of the immediately preceding and following nodes.

output

The output of a Bitcoin transaction, also called an unspent transaction output (UTXO). An output is an indivisible amount of bitcoin that can be spent, as well as a script that defines what conditions need to be fulfilled for that bitcoin to be spent. Every bitcoin transaction consumes some outputs of previously recorded transactions and creates new outputs that can be spent later by subsequent transactions. A typical bitcoin output will require a signature to be spent, but outputs can require the fulfillment of more complex scripts. For example, a multisignature script requires two or more key holders sign before the output can be spent, which is a fundamental building block of the Lightning Network.

Pay-to-Public-Key-Hash (P2PKH)

P2PKH is a type of output that locks bitcoin to the hash of a public key. An output locked by a P2PKH script can be unlocked (spent) by presenting the public key matching the hash and a digital signature created by the corresponding private key.

Pay-to-Script-Hash (P2SH)

P2SH is a versatile type of output that allows the use of complex Bitcoin Scripts. With P2SH, the complex script that details the conditions for spending the output (redeem script) is not presented in the locking script. Instead, value is locked to the hash of a script, which must be presented and fulfilled to spend the output.

P2SH address

P2SH addresses are Base58Check encodings of the 20-byte hash of a script. P2SH addresses start with a "3." P2SH addresses hide all of the complexity, so that the sender of a payment does not see the script.

Pay-to-Witness-Public-Key-Hash (P2WPKH)

P2WPKH is the SegWit equivalent of P2PKH, using a segregated witness. The signature to spend a P2WPKH output is put in the witness tree instead of the ScriptSig field. See *SegWit*.

P2WPKH address

The "native SegWit v0" address format, P2WPKH addresses are bech32-encoded and start with "bc1q".

Pay-to-Witness-Script-Hash (P2WSH)

P2WSH is the SegWit equivalent of P2SH, using a segregated witness. The signature and script to spend a P2WSH output is put in the witness tree instead of the ScriptSig field. See *SegWit*.

P2WSH address

The "native Segwi v0" script address format, P2WSH addresses are bech32-encoded and start with "bc1q".

Pay-to-Taproot (P2TR)

Activating in November 2021, Taproot is a new output type that locks bitcoin to a tree of spending conditions, or a single root condition.

P2TR address

The Taproot address format, representing SegWit v1, is a bech32m-encoded address and starts with "bc1p".

payment

A Lightning payment occurs when bitcoin is transferred within the Lightning Network. Payments are generally not seen on the Bitcoin blockchain.

payment channel

A payment channel is a financial relationship between two nodes on the Lightning Network, created using a bitcoin transaction paying a multisignature address. The channel partners can use the channel to send bitcoin back and forth between each other without committing all of the transactions to the Bitcoin blockchain. In a typical payment channel only two transactions, the funding transaction and the commitment transaction, are added to the blockchain. Payments sent across the channel are not recorded in the blockchain and are said to occur "off-chain."

payment layer

The top and fifth layer of the Lightning Network protocol suite operates on top of the routing layer. Its responsibility is to enable the payment process via BOLT #11 invoices. While it heavily uses the channel graph from the gossip protocol as defined in BOLT #7, the actual strategies to deliver a payment are not part of the specification of the protocol and are left to the implementations. As this topic is very important to ensure reliability of the payment delivery process, we included it in this book.

peer

The participants in a peer-to-peer network. In the Lightning Network, peers connect to each other via encrypted, authenticated communication through a TCP socket, over IP or Tor.

peer-to-peer layer

The peer-to-peer layer is the third layer of the Lightning Network protocol suite and works on top of the messaging layer. It is responsible for defining the syntax and semantics of information exchanged between peers via Lightning messages. This consists of control messages as defined in BOLT #9; channel establishment, operation, and closing messages as defined in BOLT #2; as well as gossip and routing messages as defined in BOLT #7.

private channel

A channel not announced to the rest of the network. Technically, "private" is a misnomer because these channels can still be identified through routing hints and commitment transactions. They are better described as "unannounced" channels. With an unannounced channel, the channel partners can send and receive payments between each other as normal. However, the rest of the network will not be aware of the channel and so cannot typically use it to route payments. Because the number and capacity of unannounced channels is unknown, the total public channel count and capacity only accounts for a portion of the total Lightning Network.

preimage

In the context of cryptography and specifically in the Lightning Network, the preimage refers to the input of a hash function that produces a specific hash. It is not feasible to compute the preimage from the hash (hash functions only go one way). By selecting a secret random value as a preimage and calculating its hash, we can commit to that preimage and later reveal it. Anyone can confirm that the revealed preimage correctly produces the hash.

Proof of Work (PoW)

Data that requires significant computation to find, and can be easily verified by anyone to prove the amount of work that was required to produce it. In Bitcoin, miners must find a numeric solution to the SHA-256 algorithm that meets a network-wide target, called the difficulty target. See *Bitcoin mining* for more information.

Point Time-Locked Contract (PTLC)

A Point Time-Locked Contract (PTLC) is a Bitcoin script that allows a conditional spend either on the presentation of a secret or after a certain blockheight has passed, similar to an HTLC. Unlike HTLCs, PTLCs do not depend on a preimage of a hash function but rather on the private key from an elliptic curve point. The security assumption is thus based on the discrete logarithm. PTLCs are not yet implemented on the Lightning Network.

relative timelock

A relative timelock is a type of timelock that allows an input to specify the earliest time the input can be added to a block. The time is relative and is based on when the output referenced by that input was recorded in a block. Relative timelocks are set by the nSequence transaction field and CHECKSEQUENCEVERIFY (CSV) Bitcoin Script opcode, which was introduced by BIP-68/112/113.

Revocable Sequence Maturity Contract (RSMC)

This contract is used to construct a payment channel between two Bitcoin or LN users who do not need to trust each other. The name comes from a sequence of states that are encoded as commitment transactions and can be revoked if wrongfully published and mined by the Bitcoin network.

revocation key

Every RSMC contains two revocation keys. Each channel partner knows one revocation key. Knowing both revocation keys, the output of the RSMC can be spent within the predefined timelock. While negotiating a new channel state, the old revocation keys are shared, thereby "revoking" the old state. Revocation keys are used to discourage channel partners from broadcasting an old channel state.

RIPEMD-160

RIPEMD-160 is a cryptographic hash function that produces a 160-bit (20-byte) hash.

routing layer

The fourth layer of the Lightning Network protocol suite operates on top of the peer-to-peer layer. Its responsibility is to define the cryptographic primitives and necessary communication protocol to allow the secure and atomic transport of bitcoin from a sending node to a recipient node. While BOLT #4 defines the onion format that is used to communicate transport information to remote peers with whom no direct connections exist, the actual transport of the onions and cryptographic primitives are defined in BOLT #2.

topology

The topology of the Lightning Network describes the shape of the Lightning Network as a mathematical graph. Nodes of the graph are the Lightning nodes (network participants/peers). The edges of the graph are the payment channels. The topology of the Lightning Network is publicly broadcast with the help of the gossip protocol, with the exception of unannounced channels. This means that the Lightning Network may be significantly larger than the announced number of channels and nodes. Knowing the topology is of particular interest in the source-based routing process of payments in which the sender discovers a route.

satoshi

A satoshi is the smallest unit (denomination) of bitcoin that can be recorded on the blockchain. One satoshi is 1/100 millionth (0.00000001) of a bitcoin and is named after the creator of Bitcoin, Satoshi Nakamoto.

Satoshi Nakamoto

Satoshi Nakamoto is the name used by the person or group of people who designed Bitcoin and created its original reference implementation. As part of the implementation, they also devised the first blockchain database. In the process, they were the first to solve the double-spending problem for digital currency. Their real identity remains unknown.

Schnorr signature

A new digital signature scheme that will be activated in Bitcoin in November 2021. It enables innovations on the Lightning Network, such as efficient PTLCs (an improvement on HTLCs).

script, Bitcoin Script

Bitcoin uses a scripting system for transactions called Bitcoin Script. Resembling the Forth programming language, it is simple, stack-based, and processed from left to right. It is purposefully Turing-incomplete, without loops or recursion.

ScriptPubKey (aka pubkey script)

ScriptPubKey or pubkey script, is a script included in outputs which sets the conditions that must be fulfilled for those outputs to be spent. Data for fulfilling the conditions can be provided in a signature script. See also *ScriptSig*.

ScriptSig (aka signature script)

ScriptSig or signature script is the data generated by a spender, which are almost always used as variables to satisfy a pubkey script.

secret key (aka private key)

The secret number that unlocks bitcoin sent to the corresponding address. A secret key looks like this: 5J76sF8L5j&#x200b;TtzE96r66Sf8cka9y44wdpJjMwCxR3tzLh3i&#x200b;bVPxh.

Segregated Witness (SegWit)

Segregated Witness (SegWit) is an upgrade to the Bitcoin protocol introduced in 2017 that adds a new witness for signatures and other transaction authorization proofs. This new witness field is exempt from the calculation of the transaction ID, which solves most classes of third-party transaction malleability. Segregated Witness was deployed as a soft fork and is a change that technically makes Bitcoin's protocol rules more restrictive.

Secure Hash Algorithm (SHA)

The Secure Hash Algorithm or SHA is a family of cryptographic hash functions published by the National Institute of Standards and Technology (NIST). The Bitcoin protocol currently uses SHA-256, which produces a 256-bit hash.

short channel ID (scid)

Once a channel is established, the index of the funding transaction on the blockchain is used as the short channel ID to uniquely identify the channel. The short channel ID consists of eight bytes referring to three numbers. In its serialized form, it depicts these three numbers as decimal values separated by the letter "x" (e.g., 600123x01x00) The first number (4 bytes) is the block height. The second number (2 bytes) is the index of the funding transaction with the blocks. The last number (2 bytes) is the transaction output.

simplified payment verification (SPV)

SPV or simplified payment verification is a method for verifying that particular transactions were included in a block without downloading the entire block. The method is used by some lightweight Bitcoin and Lightning wallets.

source-based routing

On the Lightning Network, the sender of a payment decides the route of the payment. While this decreases the success rate of the routing process, it increases the privacy of payments. Due to the SPHINX Mix Format used by onion routing, all routing nodes do not know the originator of a payment or the final recipient. Source-based routing is fundamentally different from how routing works on the Internet Protocol.

soft fork

Soft fork, or soft-forking change, is a protocol upgrade that's forward and backward compatible, so it allows both old nodes and new nodes to continue using the same chain.

SPHINX Mix Format

A particular technique for onion routing used in the Lightning Network and invented by George Danezis and Ian Goldberg in 2009. With the SPHINX Mix Format, each message of the onion package is padded with some random data so that no single hop can estimate how far along the route it has traveled. While the privacy of the sender and receiver of the payment is protected, each node is still able to return an error message along the path to the originator of the message.

submarine swap

A submarine swap is a trustless atomic swap between on-chain Bitcoin addresses and off-chain Lightning Network payments. Just as LN payments use HTLCs that make the final claim on funds conditional on the recipient revealing a secret (hash preimage), submarine swaps use the same mechanism to transfer funds across the on-chain/off-chain barrier with minimal trust. Reverse submarine swaps allow swaps in the opposite direction, from an off-chain LN payment to an on-chain Bitcoin address.

timelock

A timelock is a type of encumbrance that restricts the spending of some bitcoin until a specified future time or block height. Timelocks feature prominently in many Bitcoin contracts, including payment channels and HTLCs.

transaction

Transactions are data structures used by Bitcoin to transfer bitcoin from one address to another. Several thousand transactions are aggregated in a block, which is then recorded (mined) on the blockchain. The first transaction in each block, called the coinbase transaction, generates new bitcoin.

transaction malleability

Transaction malleability is a property that the hash of a transaction can change without changing the semantics of the transaction. For example, altering the signature can change the hash of a transaction. A commitment transaction needs the hash of a funding transaction, and if the hash of the funding transaction changes, transactions depending on it will become invalid. This will make users unable to claim the refunds if there are any. The Segregated Witness soft fork addresses this issue and was therefore an important upgrade to support the Lightning Network.

transport layer

In computer networking, the transport layer is a conceptual division of the methods used by computers (and ultimately applications) to talk to each other. The transport layer provides communication services between computers, such as flow control, verification, and multiplexing (to allow multiple applications to work on a computer at the same time).

unspent transaction output (UTXO)

See *output*.

wallet

A wallet is a piece of software that holds Bitcoin private keys. It is used to create and sign Bitcoin transactions. In the context of the Lightning Network, it also holds revocation secrets of old channel state and the latest presigned commitment transactions.

watchtower

Watchtowers are a security service on the Lightning Network that monitor payment channels for potential protocol breaches. If one of the channel partners goes offline or loses their backup, a watchtower keeps backups and can restore their channel information.

Watchtowers also monitor the Bitcoin blockchain and can submit a penalty transaction if one of the partners tries to "cheat" by broadcasting an outdated state. Watchtowers can be run by the channel partners themselves, or as a paid service offered by a third party. Watchtowers have no control over the funds in the channels themselves.

Some contributed definitions have been sourced under a CC-BY license from the Bitcoin Wiki, Wikipedia, *Mastering Bitcoin*, or from other open source publications.

Understanding the Lightning Network

An overview of the Lightning Network suitable for anyone interested in understanding the basic concepts and use of the Lightning Network.

Introduction

Welcome to *Mastering the Lightning Network*!

The Lightning Network (often abbreviated as LN), is changing the way people exchange value online, and it's one of the most exciting advancements to happen in Bitcoin's history. Today, in 2021, the Lightning Network is still in its infancy. The Lightning Network is a protocol for using Bitcoin in a smart and nonobvious way. It is a second layer technology on top of Bitcoin.

The concept of the Lightning Network was proposed in 2015, and the first implementation was launched in 2018. As of 2021, we're only beginning to see the opportunities the Lightning Network provides to Bitcoin, including improved privacy, speed, and scale. With core knowledge of the Lightning Network, you can help shape the future of the network while also building opportunities for yourself.

We assume you already have some basic knowledge about Bitcoin, but if not, don't worry—we will explain the most important Bitcoin concepts, those you must know to understand the Lightning Network, in Bitcoin Fundamentals Review. If you want to learn more about Bitcoin, you can read *Mastering Bitcoin*, 2nd edition, by Andreas M. Antonopoulos (O'Reilly), available for free online.

While the bulk of this book is written for programmers, the first few chapters are written to be approachable by anyone regardless of technical experience. In this chapter, we'll start with some terminology, then move to look at trust and its application in these systems, and finally we'll discuss the history and future of the Lightning Network. Let's get started.

Lightning Network Basic Concepts

As we explore how the Lightning Network actually works, we will encounter some technical terminology that might, at first, be a bit confusing. While all of these concepts and terms will be explained in detail as we progress through the book and are defined in the glossary, some basic definitions now will make it easier to understand the concepts in the next two chapters. If you don't understand all of the words in these definitions yet, that's OK. You'll understand more as you move through the text.

Blockchain

A distributed transaction ledger, produced by a network of computers. Bitcoin, for example, is a system that produces a blockchain. The Lightning Network is not itself a blockchain, nor does it produce a blockchain. It is a network that relies on an existing external blockchain for its security.

Digital signature

A digital signature is a mathematical scheme for verifying the authenticity of digital messages or documents. A valid digital signature gives a recipient reason to believe that the message was created by a known sender, that the sender cannot deny having sent the message, and that the message was not altered in transit.

Hash function

A cryptographic hash function is a mathematical algorithm that maps data of arbitrary size to a bit string of a fixed size (a hash) and is designed to be a one-way function, that is, a function which is infeasible to invert.

Node

A computer that participates in a network. A Lightning node is a computer that participates in the Lightning Network. A Bitcoin node is a computer that participates in the Bitcoin network. Typically an LN user will run a Lightning node *and* a Bitcoin node.

On-chain versus off-chain

A payment is *on-chain* if it is recorded as a transaction on the Bitcoin (or other underlying) blockchain. Payments sent via payment channels between Lightning nodes, and which are not visible in the underlying blockchain, are called *off-chain* payments. Usually in the Lightning Network, the only on-chain transactions are those used to open and close a Lightning payment channel. A third type of channel modifying transaction exists, called splicing, which can be used to add/remove the amount of funds committed in a channel.

Payment

When value is exchanged on the Lightning Network, we call this a "payment" as compared to a "transaction" on the Bitcoin blockchain.

Payment channel

A *financial relationship* between two nodes on the Lightning Network, typically implemented by multisignature Bitcoin transactions that share control over bitcoin between the two Lightning nodes.

Routing versus sending

Unlike Bitcoin where transactions are "sent" by broadcasting them to everyone, Lightning is a routed network where payments are "routed" across one or more payment channels following a *path* from sender to recipient.

Transaction

A data structure that records the transfer of control over some funds (e.g., some bitcoin). The Lightning Network relies on Bitcoin transactions (or those of another blockchain) to track control of funds.

More detailed definitions of these and many other terms can be found in the Glossary. Throughout this book, we will explain what these concepts mean and how these technologies actually work.

TIP Throughout this book, you will see "Bitcoin" with the first letter capitalized, which refers to the *Bitcoin system* and is a proper noun. You will also see "bitcoin," with a lowercase *b*, which refers to the currency unit. Each bitcoin is further subdivided into 100 million units each called a "satoshi" (singular) or "satoshis" (plural).

Now that you're familiar with these basic terms, let's move to a concept you are already comfortable with: trust.

Trust in Decentralized Networks

You will often hear people calling Bitcoin and the Lightning Network "trustless." At first glance this is confusing. After all, isn’t trust a good thing? Banks even use it in their names! Isn’t a "trustless" system, a system devoid of trust, a bad thing?

The use of the word "trustless" is intended to convey the ability to operate without *needing* trust in the other participants in the system. In a decentralized system like Bitcoin, you can always choose to transact with someone you trust. However, the system ensures you can’t be cheated even if you can’t trust the other party in a transaction. Trust is a nice-to-have instead of a must-have property of the system.

Contrast that to traditional systems like banking where you must place your trust in a third party, since it controls your money. If the bank violates your trust, you may be able to find some recourse from a regulator or court, but at an enormous cost of time, money, and effort.

Trustless does not mean devoid of trust. It means that trust is not a necessary prerequisite to all transactions and that you can transact even with people you don’t trust because the system prevents cheating.

Before we get into how the Lightning Network works, it’s important to understand one basic concept that underlies Bitcoin, the Lightning Network, and many other such systems: something we call a *fairness protocol*. A fairness protocol is a way to achieve fair outcomes between participants, who do not need to trust each other, without the need for a central authority, and it is the backbone of decentralized systems like Bitcoin.

Fairness Without Central Authority

When people have competing interests, how can they establish enough trust to engage in some cooperative or transactional behavior? The answer to this question lies at the core of several scientific and humanistic disciplines, such as economics, sociology, behavioral psychology, and mathematics. Some of those disciplines give us "soft" answers that depend on concepts such as reputation, fairness, morality, and even religion. Other disciplines give us concrete answers that depend only on the assumption that the participants in these interactions will act rationally, with their self-interest as the main objective.

In broad terms, there are a handful of ways to ensure fair outcomes in interactions between individuals who may have competing interests:

Require trust

You only interact with people whom you already trust, due to prior interactions, reputation, or familial relationships. This works well enough at small scale, especially within families and small groups, that it is the most common basis for cooperative behavior. Unfortunately, it doesn’t scale and it suffers from tribalist (in-group) bias.

Rule of law

Establish rules for interactions that are enforced by an institution. This scales better, but it cannot scale globally due to differences in customs and traditions, as well as the inability to scale the institutions of enforcement. One nasty side effect of this solution is that the institutions become more and more powerful as they get bigger and that may lead to corruption.

Trusted third parties

Put an intermediary in every interaction to enforce fairness. Combined with the "rule of law" to provide oversight of intermediaries, this scales better, but suffers from the same imbalance of power: the intermediaries get very powerful and may attract corruption. Concentration of power leads to systemic risk and systemic failure ("too big to fail").

Game theoretical fairness protocols

This last category emerges from the combination of the internet and cryptography and is the subject of this section. Let’s see how it works and what its advantages and disadvantages are.

Trusted Protocols Without Intermediaries

Cryptographic systems like Bitcoin and the Lightning Network are systems that allow you to transact with people (and computers) that you don’t trust. This is often referred to as "trustless" operation, even though it is not actually trustless. You have to trust in the software that you run, and you have to trust that the protocol implemented by that software will result in fair outcomes.

The big distinction between a cryptographic system like this and a traditional financial system is that in traditional finance you have a *trusted third party*, for example a bank, to ensure that outcomes are fair. A significant problem with such systems is that they give too much power to the third party, and they are also vulnerable to a *single point of failure*. If the trusted third party itself violates trust or attempts to cheat, the basis of trust breaks.

As you study cryptographic systems, you will notice a certain pattern: instead of relying on a trusted third party, these systems attempt to prevent unfair outcomes by using a system of incentives and disincentives. In cryptographic systems you place trust in the *protocol*, which is effectively a system with a set of rules that, if properly designed, will correctly apply the desired incentives and disincentives. The advantage of this approach is twofold: not only do you avoid trusting a third party, you also reduce the need to enforce fair outcomes. So long as the participants follow the agreed protocol and stay within the system, the incentive mechanism in that protocol achieves fair outcomes without enforcement.

The use of incentives and disincentives to achieve fair outcomes is one aspect of a branch of mathematics called *game theory*, which studies "models of strategic interaction among rational decision makers."[1] Cryptographic systems that control financial interactions between participants, such as Bitcoin and the Lightning Network, rely heavily on game theory to prevent participants from cheating and allow participants who don’t trust each other to achieve fair outcomes.

While game theory and its use in cryptographic systems may appear confounding and unfamiliar at first, chances are you’re already familiar with these systems in your everyday life; you just don’t recognize them yet. In the following section we’ll use a simple example from childhood to help us identify the basic pattern. Once you understand the basic pattern, you will see it everywhere in the blockchain space and you will come to recognize it quickly and intuitively.

In this book, we call this pattern a *fairness protocol*, defined as a process that uses a system of incentives and/or disincentives to ensure fair outcomes for participants who don’t trust each other.

Enforcement of a fairness protocol is only necessary to ensure that the participants can’t escape the incentives or disincentives.

A Fairness Protocol in Action

Let’s look at an example of a fairness protocol that you may already be familiar with.

Imagine a family lunch, with a parent and two children. The children are fussy eaters and the only thing they will agree to eat is fried potatoes. The parent has prepared a bowl of fried potatoes ("french fries" or "chips" depending on which English dialect you use). The two siblings must share the plate of chips. The parent must ensure a fair distribution of chips to each child; otherwise, the parent will have to hear constant complaining (maybe all day), and there’s always a possibility of an unfair situation escalating to violence. What is a parent to do?

There are a few different ways that fairness can be achieved in this strategic interaction between two siblings that do not trust each other and have competing interests. The naive but commonly used method is for the parent to use their authority as a trusted third party: they split the bowl of chips into two servings. This is similar to traditional finance, where a bank, accountant, or lawyer acts as a trusted third party to prevent any cheating between two parties who want to transact.

The problem with this scenario is that it vests a lot of power and responsibility in the hands of the trusted third party. In this example, the parent is fully responsible for the equal allocation of chips, and the parties merely wait, watch, and complain. The children accuse the parent of playing favorites and not allocating the chips fairly. The siblings fight over the chips, yelling "that chip is bigger!" and dragging the parent into their fight. It sounds awful, doesn’t it? Should the parent yell louder? Take all of the chips away? Threaten to never make chips again and let those ungrateful children go hungry?

A much better solution exists: the siblings are taught to play a game called "split and choose." At each lunch one sibling splits the bowl of chips into two servings and the *other* sibling gets to choose which serving they want. Almost immediately, the siblings figure out the dynamic of this game. If the one splitting makes a mistake or tries to cheat, the other sibling can "punish" them by choosing the bigger bowl. It is in the best interest of both siblings, but especially the one splitting the bowl, to play fair. Only the cheater loses in this scenario. The parent doesn’t even have to use their authority or enforce fairness. All the parent has to do is *enforce the protocol*; as long as the siblings cannot escape their assigned roles of "splitter" and "chooser," the protocol itself ensures a fair outcome without the need for any intervention. The parent can’t play favorites or distort the outcome.

WARNING

While the infamous chip battles of the 1980s neatly illustrate the point, any similarity between the preceding scenario and any of the authors' actual childhood experiences with their cousins is entirely coincidental…or is it?

Security Primitives as Building Blocks

In order for a fairness protocol like this to work, there need to be certain guarantees, or *security primitives*, that can be combined to ensure enforcement. The first security primitive is *strict time ordering/sequencing*: the "splitting" action must happen before the "choosing" action. It’s not immediately obvious, but unless you can guarantee that action A happens before action B, then the protocol falls apart. The second security primitive is *commitment with nonrepudiation*. Each sibling must commit to their choice of role: either splitter or chooser. Also, once the splitting has been completed, the splitter is committed to the split they created—they cannot repudiate that choice and go try again.

Cryptographic systems offer a number of security primitives that can be combined in different ways to construct a fairness protocol. In addition to sequencing and commitment, we can also use many other tools:

- Hash functions to fingerprint data, as a form of commitment, or as the basis for a digital signature
- Digital signatures for authentication, nonrepudiation, and proof of ownership of a secret
- Encryption/decryption to restrict access to information to authorized participants only

This is only a small list of a whole "menagerie" of security and cryptographic primitives that are in use. More basic primitives and combinations are invented all the time.

In our real-life example, we saw one form of fairness protocol called "split and choose." This is just one of a myriad different fairness protocols that can be built by combining the building blocks of security primitives in different ways. But the basic pattern is always the same: two or more participants interact without trusting each other by engaging in a series of steps that are part of an agreed protocol. The protocol’s steps arrange incentives and disincentives to ensure that if the participants are rational, cheating is counterproductive and fairness is the automatic outcome. Enforcement is not necessary to get fair outcomes—it is only necessary to keep the participants from breaking out of the agreed protocol.

Now that you understand this basic pattern, you will start seeing it everywhere in Bitcoin, the Lightning Network, and many other systems. Let’s look at some specific examples next.

Example of the Fairness Protocol

The most prominent example of a fairness protocol is Bitcoin’s consensus algorithm, Proof of Work (PoW). In Bitcoin, miners compete to verify transactions and aggregate them in blocks. To ensure that the miners do not cheat, without entrusting them with authority, Bitcoin uses a system of incentives and disincentives. Miners have to use electricity and dedicate hardware doing "work" that is embedded as a "proof" inside every block. This is achieved because of a property of hash functions where the output value is randomly distributed across the entire range of possible outputs. If miners succeed in producing a valid block fast enough, they are rewarded by earning the block reward for that block. Forcing miners to use a lot of electricity before the network considers their block means that they have an incentive to correctly validate the transactions in the block. If they cheat or make any kind of mistake, their block is rejected and the electricity they used to "prove" it is wasted. No one needs to force miners to produce valid blocks; the reward and punishment incentivize them to do so. All the protocol needs to do is ensure that only valid blocks with Proof of Work are accepted.

The fairness protocol pattern can also be found in many different aspects of the Lightning Network:

- Those who fund channels make sure that they have a refund transaction signed before they publish the funding transaction.

- Whenever a channel is moved to a new state, the old state is "revoked" by ensuring that if anyone tries to broadcast it, they lose the entire balance and get punished.
- Those who forward payments know that if they commit funds forward, they can either get a refund or get paid by the node preceding them.

Again and again, we see this pattern. Fair outcomes are not enforced by any authority. They emerge as the natural consequence of a protocol that rewards fairness and punishes cheating, a fairness protocol that harnesses self-interest by directing it toward fair outcomes.

Bitcoin and the Lightning Network are both implementations of fairness protocols. So why do we need the Lightning Network? Isn’t Bitcoin enough?

Motivation for the Lightning Network

Bitcoin is a system that records transactions on a globally replicated public ledger. Every transaction is seen, validated, and stored by every participating computer. As you can imagine, this generates a lot of data and is difficult to scale.

As Bitcoin and the demand for transactions grew, the number of transactions in each block increased until it eventually reached the block size limit. Once blocks are "full," excess transactions are left to wait in a queue. Many users will increase the fees they’re willing to pay to buy space for their transactions in the next block.

If demand continues to outpace the capacity of the network, an increasing number of users' transactions are left waiting unconfirmed. Competition for fees also increases the cost of each transaction, making many smaller-value transactions (e.g., microtransactions) completely uneconomical during periods of particularly high demand.

To solve this problem, we could increase the block size limit to create space for more transactions. An increase in the "supply" of block space will lead to a lower price equilibrium for transaction fees.

However, increasing block size shifts the cost to node operators and requires them to expend more resources to validate and store the blockchain. Because blockchains are gossip protocols, each node is required to know and validate every single transaction that occurs on the network. Furthermore, once validated, each transaction and block must be propagated to the node’s "neighbors," multiplying the bandwidth requirements. As such, the greater the block size, the greater the bandwidth, processing, and storage requirements for each individual node. Increasing transaction capacity in this way has the undesirable effect of centralizing the system by reducing the number of nodes and node operators. Since node operators are not compensated for running nodes, if nodes are very expensive to run, only a few well-funded node operators will continue to run nodes.

Scaling Blockchains

The side effects of increasing the block size or decreasing the block time with respect to centralization of the network are severe, as a few calculations with the numbers show.

Let us assume the usage of Bitcoin grows so that the network has to process 40,000 transactions per second, which is the approximate transaction processing level of the Visa network during peak usage.

Assuming 250 bytes on average per transaction, this would result in a data stream of 10 megabytes per second (MBps) or 80 megabits per second (Mbps) just to be able to receive all the transactions. This does not include the traffic overhead of forwarding the transaction information to other peers. While 10 MBps does not seem extreme in the context of high-speed fiber optic and 5G mobile speeds, it would effectively exclude anyone who cannot meet this requirement from running a node, especially in countries where high-performance internet is not affordable or widely available.

Users also have many other demands on their bandwidth and cannot be expected to expend this much only to receive transactions.

Furthermore, storing this information locally would result in 864 gigabytes per day. This is roughly one terabyte of data, or the size of a hard drive.

Verifying 40,000 Elliptic Curve Digital Signature Algorithm (ECDSA) signatures per second is also barely feasible (see this article on StackExchange), making the *initial block download (IBD)* of the Bitcoin blockchain (synchronizing and verifying everything starting from the genesis block) almost impossible without very expensive hardware.

While 40,000 transactions per second seems like a lot, it only achieves parity with traditional financial payment networks at peak times. Innovations in machine-to-machine payments, microtransactions, and other applications are likely to push demand to many orders higher than that.

Simply put: you can’t scale a blockchain to validate the entire world’s transactions in a decentralized way.

But what if each node wasn’t required to know and validate every single transaction? What if there was a way to have scalable off-chain transactions, without losing the security of the Bitcoin network?

In February 2015, Joseph Poon and Thaddeus Dryja proposed a possible solution to the Bitcoin scalability problem, with the publication of "The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments."[2]

In the (now outdated) whitepaper, Poon and Dryja estimate that in order for Bitcoin to reach the 47,000 transactions per second processed at peak by Visa, it would require 8 GB blocks. This would make running a node completely untenable for anyone but large-scale enterprises and industrial-grade operations. The result would be a network in which only a few users could actually validate the state of the ledger. Bitcoin relies on users validating the ledger for themselves, without explicitly trusting third parties, in order to stay decentralized. Pricing users out of running nodes would force the average user to trust third parties to discover the state of the ledger, ultimately breaking the trust model of Bitcoin.

The Lightning Network proposes a new network, a second layer, where users can make payments to each other peer-to-peer, without the necessity of publishing a transaction to the Bitcoin blockchain for each payment. Users may pay each other on the Lightning Network as many times as they want, without creating additional Bitcoin transactions or incurring on-chain fees. They only make use of the Bitcoin blockchain to load bitcoin onto the Lightning Network initially and to *settle*, that is, to remove bitcoin from the Lightning Network. The result is that many more Bitcoin payments can take place off-chain, with only the initial loading and final settlement transactions needing to be validated and stored by Bitcoin nodes. Aside from reducing the burden on nodes, payments on the Lightning Network are cheaper for users because they do not need to pay blockchain fees, and more private for users because they are not published to all participants of the network and furthermore are not stored permanently.

While the Lightning Network was initially conceived for Bitcoin, it can be implemented on any blockchain that meets some basic technical requirements. Other blockchains, such as Litecoin, already support the Lightning Network. Additionally, several other blockchains are developing similar second layer or "layer 2" solutions to help them scale.

The Lightning Network's Defining Features

The Lightning Network is a network that operates as a second layer protocol on top of Bitcoin and other blockchains. The Lightning Network enables fast, secure, private, trustless, and permissionless payments. Here are some of the features of the Lightning Network:

- Users of the Lightning Network can route payments to each other for low cost and in real time.
- Users who exchange value over the Lightning Network do not need to wait for block confirmations for payments.
- Once a payment on the Lightning Network has completed, usually within a few seconds, it is final and cannot be reversed. Like a Bitcoin transaction, a payment on the Lightning Network can only be refunded by the recipient.
- Whereas on-chain Bitcoin transactions are broadcast and verified by all nodes in the network, payments routed on the Lightning Network are transmitted between pairs of nodes and are not visible to everyone, resulting in much greater privacy.
- Unlike transactions on the Bitcoin network, payments routed on the Lightning Network do not need to be stored permanently. Lightning thus uses fewer resources and hence is cheaper. This property also has benefits for privacy.
- The Lightning Network uses onion routing, similar to the protocol used by The Onion Router (Tor) privacy network, so that even the nodes involved in routing a payment are only directly aware of their predecessor and successor in the payment route.
- When used on top of Bitcoin, the Lightning Network uses real bitcoin, which is always in the possession (custody) and full control of the user. Lightning is not a separate token or coin, it *is* Bitcoin.

Lightning Network Use Cases, Users, and Their Stories

To better understand how the Lightning Network actually works, and why people use it, we'll be following a number of users and their stories.

In our examples, some of the people have already used Bitcoin and others are completely new to the Bitcoin network. Each person and their story, as listed here, illustrate one or more specific use cases. We'll be revisiting them throughout this book:

Consumer

Alice is a Bitcoin user who wants to make fast, secure, cheap, and private payments for small retail purchases. She buys coffee with bitcoin, using the Lightning Network.

Merchant

Bob owns a coffee shop, "Bob’s Cafe." On-chain Bitcoin payments don’t scale for small amounts like a cup of coffee, so he uses the Lightning Network to accept Bitcoin payments almost instantaneously and for low fees.

Software service business

Chan is a Chinese entrepreneur who sells information services related to the Lightning Network, as well as Bitcoin and other cryptocurrencies. Chan is selling these information services over the internet by implementing micropayments over the Lightning Network. Additionally, Chan has implemented a liquidity provider service that rents inbound channel capacity on the Lightning Network, charging a small bitcoin fee for each rental period.

Gamer

Dina is a teenage gamer from Russia. She plays many different computer games, but her favorite ones are those that have an "in-game economy" based on real money. As she plays games, she also earns money by acquiring and selling virtual in-game items. The Lightning Network allows her to transact in small amounts for in-game items as well as earn small amounts for completing quests.

Chapter Summary

In this chapter, we talked about the fundamental concept that underlies both Bitcoin and the Lightning Network: the fairness protocol.

We looked at the history of the Lightning Network and the motivations behind second layer scaling solutions for Bitcoin and other blockchain-based networks.

We learned basic terminology including node, payment channel, on-chain transactions, and off-chain payments.

Finally, we met Alice, Bob, Chan, and Dina, whom we’ll be following throughout the rest of the book. In the next chapter, we’ll meet Alice and walk through her thought process as she selects a Lightning wallet and prepares to make her first Lightning payment to buy a cup of coffee from Bob’s Cafe.

Getting Started

In this chapter, we will begin where most people start when encountering the Lightning Network for the first time—choosing software to participate in the LN economy. We will examine the choices of two users who represent a common use case for the Lightning Network and learn by example. Alice, a coffee shop customer, will be using a Lightning wallet on her mobile device to buy coffee from Bob's Cafe. Bob, a merchant, will be using a Lightning node and wallet to run a point-of-sale system at his cafe, so he can accept payments over the Lightning Network.

Alice's First Lightning Wallet

Alice is a longtime Bitcoin user. We first met Alice in Chapter 1 of *Mastering Bitcoin*,[3] when she bought a cup of coffee from Bob's cafe using a Bitcoin transaction. If you are not yet familiar with how Bitcoin transactions work or need a refresher, please read *Mastering Bitcoin* or the summary in Bitcoin Fundamentals Review.

Alice recently learned that Bob's Cafe just started accepting LN payments! Alice is eager to learn about and experiment with the Lightning Network; she wants to be one of Bob's first LN customers. To do this, first, Alice has to select a Lightning wallet that meets her needs.

Alice does not want to entrust custody of her bitcoin to third parties. She has learned enough about cryptocurrency to know how to use a wallet. She also wants a mobile wallet so that she can use it for small payments on the go, so she chooses the *Eclair* wallet, a popular noncustodial mobile Lightning wallet. Let's learn more about how and why she's made these choices.

Lightning Nodes

The Lightning Network is accessed via software applications that can speak the LN protocol. A *Lightning Network node* (LN node or simply node) is a software application that has three important characteristics. First, Lightning nodes are wallets, so they send and receive payments over the Lightning Network as well as on the Bitcoin network. Second, nodes must communicate on a peer-to-peer basis with other Lightning nodes creating the network. Finally, Lightning nodes also need access to the Bitcoin blockchain (or other blockchains for other cryptocurrencies) to secure the funds used for payments.

Users have the highest degree of control by running their own Bitcoin node and Lightning node. However, Lightning nodes can also use a lightweight Bitcoin client, commonly referred to as simplified payment verification (SPV), to interact with the Bitcoin blockchain.

Lightning Explorers

LN explorers are useful tools to show the statistics of nodes, channels, and network capacity.

Following is an inexhaustive list:

- Lightning explorer
- ACINQ's Lightning explorer, with fancy visualization

- Amboss Space Lightning explorer, with community metrics and intuitive visualizations
- Fiatjaf’s Lightning explorer with many diagrams
- hashXP Lightning explorer

TIP

Note that when using Lightning explorers, just like with other block explorers, privacy can be a concern. If users are careless, the website may track their IP addresses and collect their behavior records (for example, the nodes users are interested in).

Also, it should be noted that because there is no global consensus of the current Lightning graph or the current state of any existing channel policy, users should never rely on Lightning explorers to retrieve the most current information. Furthermore, as users open, close, and update channels, the graph will change and individual Lightning explorers may not be up to date. Use Lightning explorers to visualize the network or gather information, but not as an authoritative source of what is happening on the Lightning Network. To have an authoritative view of the Lightning Network, run your own Lightning node that will build a channel graph and collect various statistics, which you can view with a web-based interface.

Lightning Wallets

The term *Lightning wallet* is somewhat ambiguous because it can describe a broad variety of components combined with some user interface. The most common components of Lightning wallet software include:

- A keystore that holds secrets, such as private keys
- An LN node (Lightning node) that communicates on the peer-to-peer network, as described previously
- A Bitcoin node that stores blockchain data and communicates with other Bitcoin nodes
- A database "map" of nodes and channels that are announced on the Lightning Network
- A channel manager that can open and close LN channels
- A close-up system that can find a path of connected channels from payment source to payment destination

A Lightning wallet may contain all of these functions, acting as a "full" wallet, with no reliance on any third-party services. Or one or more of these components may rely (partially or entirely) on third-party services that mediate those functions.

A *key* distinction (pun intended) is whether the keystore function is internal or outsourced. In blockchains, control of keys determines custody of funds, as memorialized by the phrase "your keys, your coins; not your keys, not your coins." Any wallet that outsources management of keys is called a *custodial* wallet because a third party acting as custodian has control of the user’s funds, not the user. A *noncustodial* or *self-custody* wallet, by comparison, is one where the keystore is part of the wallet, and keys are controlled directly by the user. The term noncustodial wallet just implies that the keystore is local and under the user’s control. However, one or more of the other wallet components may or may not be outsourced and rely on trusted third parties.

Blockchains, especially open blockchains like Bitcoin, attempt to minimize or eliminate trust in third parties and empower users. This is often called a "trustless" model, though "trust minimized" is a better term. In such systems, the user trusts the software rules, not third parties. Therefore, the issue of control over keys is a principal consideration when choosing a Lightning wallet.

Every other component of a Lightning wallet brings similar considerations of trust. If all the components are under the control of the user, then the amount of trust in third parties is minimized, bringing maximum power to the user. Of course, this brings a direct trade-off because with that power comes the corresponding responsibility to manage complex software.

Every user must consider their own technical skills before deciding what type of Lightning wallet to use. Those with strong technical skills should use a Lightning wallet that puts all of the components under the direct control of the user. Those with fewer technical skills, but with a desire to control their funds, should choose a noncustodial Lightning wallet. Often the trust in these cases relates to privacy. If users decide to outsource some functionality to a third party, they usually give up some privacy as the third party will learn some information about them.

Finally, those seeking simplicity and convenience, even at the expense of control and security, may choose a custodial Lightning wallet. This is the least technically challenging option, but it *undermines the trust model of cryptocurrency* and should therefore be considered only as a stepping stone toward more control and self-reliance.

There are many ways wallets can be characterized or categorized. The most important questions to ask about a specific wallet are:

1. Does this Lightning wallet have a full Lightning node or does it use a third-party Lightning node?
2. Does this Lightning wallet have a full Bitcoin node or does it use a third-party Bitcoin node?
3. Does this Lightning wallet store its own keys under user control (self-custody) or are the keys held by a third-party custodian?

TIP If a Lightning wallet uses a third-party Lightning node, it is this third-party Lightning node that decides how to communicate with Bitcoin. Hence, using a third-party Lightning node implies that you are also using a third-party Bitcoin node. Only when the Lightning wallet uses its own Lightning node does the choice between full Bitcoin node and third-party Bitcoin node exist.

At the highest level of abstraction, Questions 1 and 3 are the most elementary ones. From these two questions, we can derive four possible categories. We can place these four categories into a quadrant, as seen in Lightning wallets quadrant. But remember that this is just one way of categorizing Lightning wallets.

Table 1. Lightning wallets quadrant

	Full Lightning node	**Third-party Lightning node**
Self-custody	Q1: High technical skill, least trust in third parties, most permissionless	Q2: Below medium technical skills, below medium trust in third parties, requires some permissions
Custodial	Q3: Above medium technical skills, above medium trust in third parties, requires some permissions	Q4: Low technical skills, high trust in third parties, least permissionless

Quadrant 3 (Q3), where a full Lightning node is used, but the keys are held by a custodian, is currently not common. Future wallets from that quadrant may let a user worry about the operational aspects of their node, but then delegate access to the keys to a third party which primarily uses cold storage.

Lightning wallets can be installed on a variety of devices, including laptops, servers, and mobile devices. To run a full Lightning node, you will need to use a server or desktop computer, because mobile devices and laptops are usually not powerful enough in terms of capacity, processing, battery life, and connectivity.

The category third-party Lightning nodes can again be subdivided:

Lightweight

This means that the wallet does not operate a Lightning node and thus needs to obtain information about the Lightning Network over the internet from someone else’s Lightning node.

None

This means that not only is the Lightning node operated by a third party, but most of the wallet is operated by a third party in the cloud. This is a custodial wallet where someone else controls custody of the funds.

These subcategories are used in Examples of popular Lightning wallets.

Other terms that need explanation in Examples of popular Lightning wallets in the column "Bitcoin node" are:

Neutrino

This wallet does not operate a Bitcoin node. Instead, a Bitcoin node operated by someone else (a third party) is accessed via the Neutrino Protocol.

Electrum

This wallet does not operate a Bitcoin node. Instead, a Bitcoin node operated by someone else (a third party) is accessed via the Electrum Protocol.

Bitcoin Core

This is an implementation of a Bitcoin node.

btcd

This is another implementation of a Bitcoin node.

In Examples of popular Lightning wallets, we see some examples of currently popular Lightning node and wallet applications for different types of devices. The list is sorted first by device type and then alphabetically.

Table 2. Examples of popular Lightning wallets

Application	Device	Lightning node	Bitcoin node	Keystore
Blue Wallet	Mobile	None	None	Custodial
Breez Wallet	Mobile	Full node	Neutrino	Self-custody
Eclair Mobile	Mobile	Lightweight	Electrum	Self-custody
lntxbot	Mobile	None	None	Custodial
Muun	Mobile	Lightweight	Neutrino	Self-custody
Phoenix Wallet	Mobile	Lightweight	Electrum	Self-custody
Zeus	Mobile	Full node	Bitcoin Core/btcd	Self-custody
Electrum	Desktop	Full node	Bitcoin Core/Electrum	Self-custody
Zap Desktop	Desktop	Full node	Neutrino	Self-custody
c-lightning	Server	Full node	Bitcoin Core	Self-custody
Eclair Server	Server	Full node	Bitcoin Core/Electrum	Self-custody
lnd	Server	Full node	Bitcoin Core/btcd	Self-custody

Testnet Bitcoin

The Bitcoin system offers an alternative chain for testing purposes called *testnet*, in contrast with the "normal" Bitcoin chain which is referred to as *mainnet*. On testnet, the currency is *testnet bitcoin* (*tBTC*), which is a worthless copy of bitcoin used exclusively for testing. Every function of Bitcoin is replicated exactly, but the money is worth nothing, so you literally have nothing to lose!

Some Lightning wallets can also operate on testnet, allowing you to make Lightning payments with testnet bitcoin, without risking real funds. This is a great way to experiment with Lightning safely. Eclair Mobile, which Alice uses in this chapter, is one example of a Lightning wallet that supports testnet operation.

You can get some tBTC to play with from a *testnet bitcoin faucet*, which gives out free tBTC on demand. Here are a few testnet faucets:

```
<ul class="simplelist">
<li><a href="https://coinfaucet.eu/en/btc-testnet/"><em>https://coinfaucet.eu/en/btc-
testnet</em></a></li>
<li><a href="https://testnet-faucet.mempool.co/"><em>https://testnet-
faucet.mempool.co/</em></a></li>
<li><a
href="https://bitcoinfaucet.uo1.net/"><em>https://bitcoinfaucet.uo1.net/</em></a></li>
<li><a
href="https://testnet.help/en/btcfaucet/testnet"><em>https://testnet.help/en/btcfaucet
/testnet</em></a></li>
</ul>
```

All of the examples in this book can be replicated exactly on testnet with tBTC, so you can follow along if you want without risking real money.

Balancing Complexity and Control

Lightning wallets have to strike a careful balance between complexity and user control. Those that give the user the most control over their funds, the highest degree of privacy, and the greatest independence from third-party services are necessarily more complex and difficult to operate. As the technology advances, some of these trade-offs will become less stark, and users may be able to get more control without more complexity. However, for now, different companies and projects are exploring different positions along this control-complexity spectrum, hoping to find the "sweet spot" for the users they are targeting.

When selecting a wallet, keep in mind that even if you don’t see these trade-offs, they still exist. For example, many wallets will attempt to remove the burden of channel management from their users. To do so, they introduce central *hub nodes* that all their wallets connect to automatically. While this trade-off simplifies the user interface and user experience, it introduces a single point of failure (SPoF) as these hub nodes become indispensable for the wallet’s operation. Furthermore, relying on a "hub" like this can reduce user privacy since the hub knows the sender and potentially (if constructing the payment route on behalf of the user) also the recipient of each payment made by the user’s wallet.

In the next section, we will return to our first user and walk through her first Lightning wallet setup. She has chosen a wallet that is more sophisticated than the easier custodial wallets. This allows us to show some of the underlying complexity and introduce some of the inner workings of an advanced wallet. You may find that your first ideal wallet is oriented toward ease of use, accepting some of the control and privacy trade-offs. Or perhaps you are more of a power user and want to run your own Lightning and Bitcoin nodes as part of your wallet solution.

Downloading and Installing a Lightning Wallet

When looking for a new cryptocurrency wallet, you must be very careful to select a secure source for the software.

Unfortunately, many fake wallet applications will steal your money, and some of these even find their way onto reputable and supposedly vetted software sites like the Apple and Google application stores. Whether you are installing your first or your tenth wallet, always exercise extreme caution. A rogue app may not just steal any money you entrust it with, but it might also be able to steal keys and passwords from other applications by compromising your mobile device operating system.

Alice uses an Android device and will use the Google Play Store to download and install the Eclair wallet. Searching on Google Play, she finds an entry for "Eclair Mobile," as shown in Eclair Mobile in the Google Play Store.

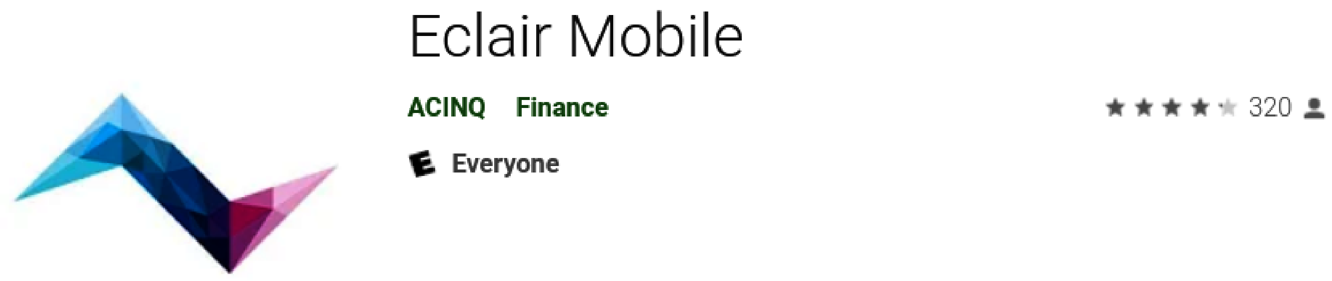


Figure 1. Eclair Mobile in the Google Play Store

TIP It is possible to experiment and test all Bitcoin-type software with zero risk (except for your own time) by using testnet bitcoins. You can also download the Eclair testnet wallet to try Lightning (on testnet) by going to the Google Play Store.

Alice notices a few different elements on this page that help her ascertain that this is, most likely, the correct "Eclair Mobile" wallet she is looking for. Firstly, the organization ACINQ[4] is listed as the developer of this mobile wallet, which Alice knows from her research is the correct developer. Secondly, the wallet has been installed "10,000+" times and has more than 320 positive reviews. It is unlikely that this is a rogue app that has snuck into the Google Play Store. As a third step, she goes to the ACINQ website. She verifies that the web page is secure by checking that the address begins with https, or prefixed by a padlock in some browsers. On the website, she goes to the Download section or looks for the link to the Google App Store. She finds the link and clicks it. She compares that this link brings her to the very same app in the Google App Store. Satisfied by these findings, Alice installs the Eclair app on her mobile device.

WARNING Always exercise great care when installing software on any device. There are many fake cryptocurrency wallets that will not only steal your money but might also compromise all other applications on your device.

Creating a New Wallet

When Alice opens the Eclair Mobile app for the first time, she is presented with a choice to "Create a New Wallet" or to "Import an Existing Wallet." Alice will create a new wallet, but let’s first discuss why these options are presented here and what it means to import an existing wallet.

Responsibility with Key Custody

As we mentioned at the beginning of this section, Eclair is a *noncustodial* wallet, meaning that Alice has sole custody of the keys used to control her bitcoin. This also means that Alice is responsible for protecting and backing up those keys. If Alice loses the keys, no one can help her recover the bitcoin, and they will be lost forever.

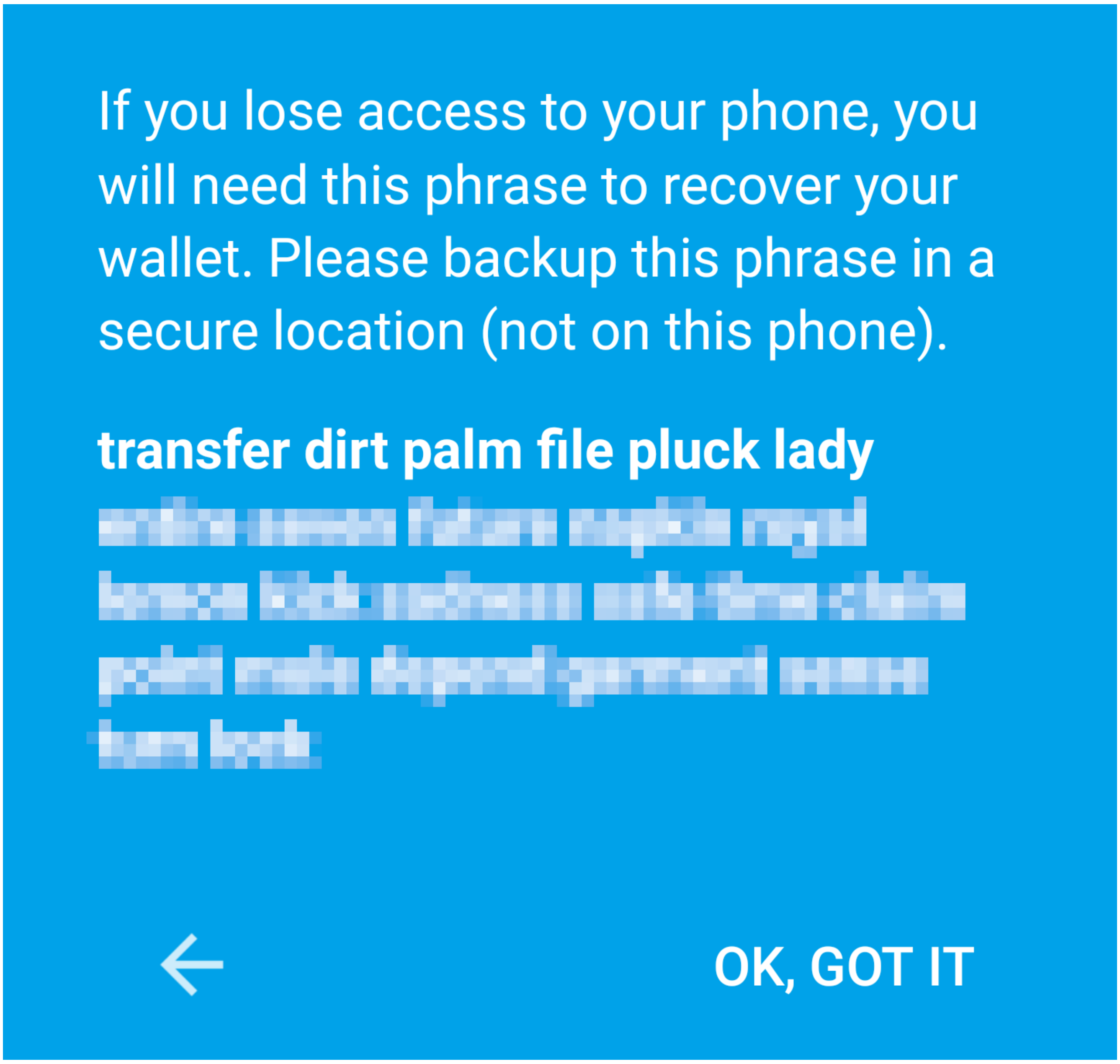
WARNING With the Eclair Mobile wallet, Alice has custody and control of the keys and, therefore, full responsibility to keep the keys safe and backed up. If she loses the keys, she loses the bitcoin, and no one can help her recover from that loss!

Mnemonic Words

Similar to most Bitcoin wallets, Eclair Mobile provides a *mnemonic phrase* (also sometimes called a "seed" or "seed phrase") for Alice to back up. The mnemonic phrase consists of 24 English words, selected randomly by the software and used as the basis for the keys that are generated by the wallet. Alice can use the mnemonic phrase to restore all the transactions and funds in the Eclair Mobile wallet in the case of a lost mobile device, a software bug, or memory corruption.

TIP The correct term for these backup words is "mnemonic phrase." We avoid the use of the term "seed" to refer to a mnemonic phrase because even though its use is common, it is incorrect.

When Alice chooses to create a new wallet, she will see a screen with her mnemonic phrase, which looks like the screenshot in New wallet mnemonic phrase.


If you lose access to your phone, you will need this phrase to recover your wallet. Please backup this phrase in a secure location (not on this phone).

transfer dirt palm file pluck lady
[redacted]

Figure 2. New wallet mnemonic phrase

In New wallet mnemonic phrase, we have purposely obscured part of the mnemonic phrase to prevent readers of this book from reusing the mnemonic.

Storing the Mnemonic Safely

Alice needs to be careful to store the mnemonic phrase in a way that prevents theft but also avoids accidental loss. The recommended way to properly balance these risks is to write two copies of the mnemonic phrase on paper, with each of the words numbered—the order matters.

Once Alice has recorded the mnemonic phrase, after touching "OK GOT IT" on her screen, she will be presented with a quiz to make sure that she correctly recorded the mnemonic. The quiz will ask for three or four of the words at random. Alice isn’t expecting a quiz, but since she recorded the mnemonic correctly, she passes without any difficulty.

Once Alice has recorded the mnemonic phrase and passed the quiz, she should store each copy in a separate secure location, such as a locked desk drawer or a fireproof safe.

WARNING Never attempt a "DIY" security scheme that deviates in any way from the best practice recommendation in Storing the Mnemonic Safely. Do not cut your mnemonic in half, make screenshots, store it on USB drives or cloud drives, encrypt it, or try any other nonstandard method. You will tip the balance in such a way as to risk permanent loss. Many people have lost funds, not from theft, but because they tried a nonstandard solution without having the expertise to balance the risks involved. The best practice recommendation is carefully considered by experts and suitable for the vast majority of users.

After Alice initializes her Eclair Mobile wallet, she will see a brief tutorial that highlights the various elements of the user interface. We won't replicate the tutorial here, but we will explore all of those elements as we follow Alice's attempt to buy a cup of coffee!

Loading Bitcoin onto the Wallet

Alice now has a Lightning wallet. But it's empty! She now faces one of the more challenging aspects of this experiment: she has to find a way to acquire some bitcoin and load it onto her Eclair wallet.

TIP If Alice already has bitcoin in another wallet, she could choose to send that bitcoin to her Eclair wallet instead of acquiring new bitcoin to load onto her new wallet.

Acquiring Bitcoin

There are several ways Alice can acquire bitcoin:

- She can exchange some of her national currency (e.g., USD) on a cryptocurrency exchange.
- She can buy some from a friend, or an acquaintance from a Bitcoin meetup, in exchange for cash.
- She can find a *Bitcoin ATM* in her area, which acts as a vending machine, selling bitcoin for cash.
- She can offer her skills or a product she sells and accept payment in bitcoin.
- She can ask her employer or clients to pay her in bitcoin.

All of these methods have varying degrees of difficulty, and many will involve paying a fee. Some will also require Alice to provide identification documents to comply with local banking regulations. However, with all these methods, Alice will be able to receive bitcoin.

Receiving Bitcoin

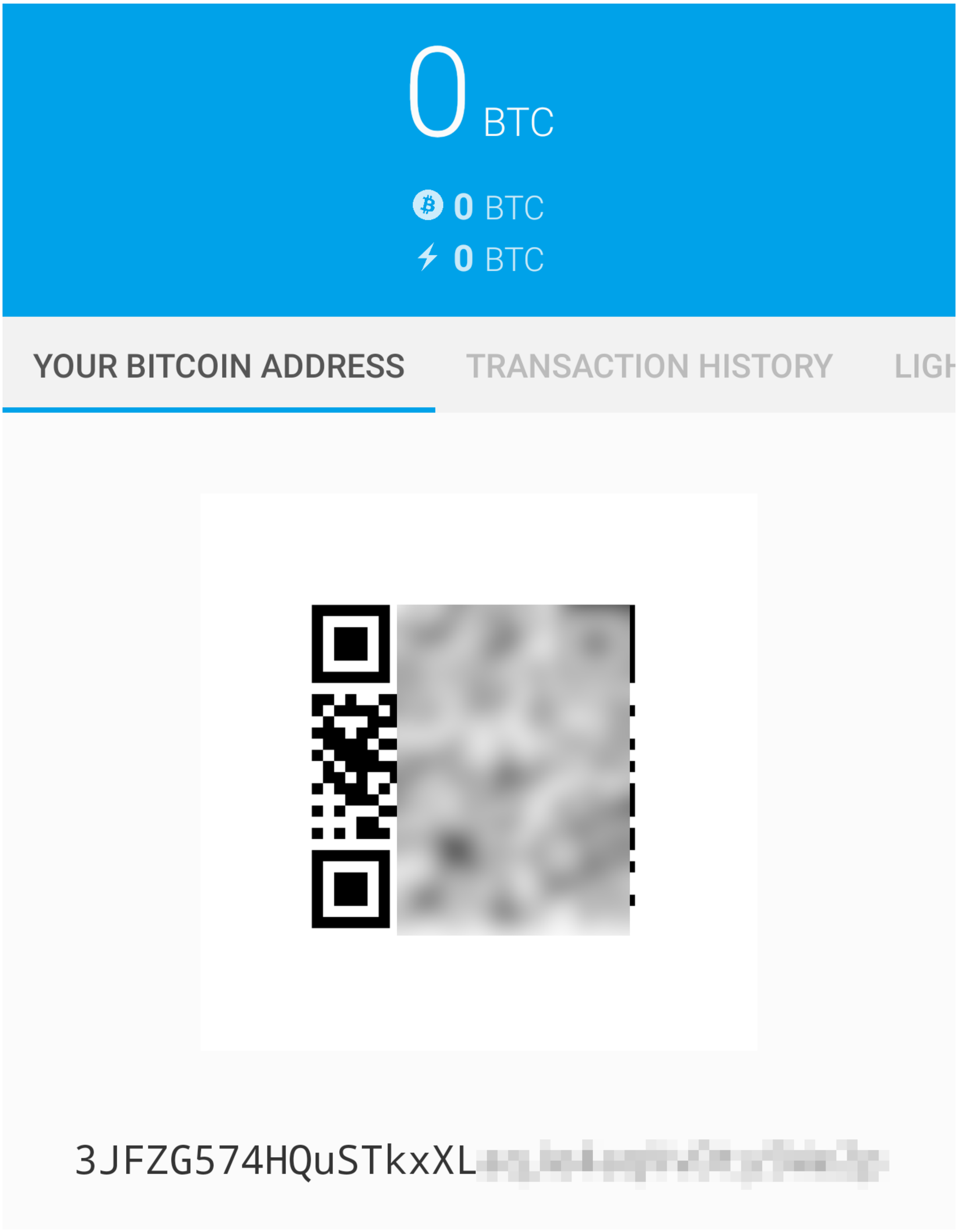
Let's assume Alice has found a local Bitcoin ATM and has decided to buy some bitcoin in exchange for cash. An example of a Bitcoin ATM, one built by the Lamassu Company, is shown in A Lamassu Bitcoin ATM. Such Bitcoin ATMs accept national currency (cash) through a cash slot and send bitcoin to a Bitcoin address scanned from a user's wallet using a built-in camera.

Figure 3. A Lamassu Bitcoin ATM

To receive the bitcoin in her Eclair Lightning wallet, Alice will need to present a Bitcoin address from the Eclair Lightning wallet to the ATM. The ATM can then send Alice's newly acquired bitcoin to this Bitcoin address.

To see a Bitcoin address on the Eclair wallet, Alice must swipe to the left column titled YOUR BITCOIN ADDRESS (see Alice's bitcoin address, shown in Eclair), where she will see a square barcode (called a *QR code*) and a string of letters and numbers below that.

The QR code contains the same string of letters and numbers shown below it, in an easy to scan format. This way, Alice doesn't have to type the Bitcoin address. In the screenshot (Alice's bitcoin address, shown in Eclair), we have purposely blurred both, to prevent readers from inadvertently sending bitcoin to this address.



3JFZG574HQuSTkxXL

Figure 4. Alice's bitcoin address, shown in Eclair

NOTE	Both Bitcoin addresses and QR codes contain error detection information that prevents any typing or scanning errors from producing a "wrong" Bitcoin address. If there is a mistake in the address, any Bitcoin wallet will notice the error and refuse to accept the Bitcoin address as valid.

Alice can take her mobile device to the ATM and show it to the built-in camera, as shown in Bitcoin ATM scans the QR code.. After inserting some cash into the slot, she will receive bitcoin in Eclair!

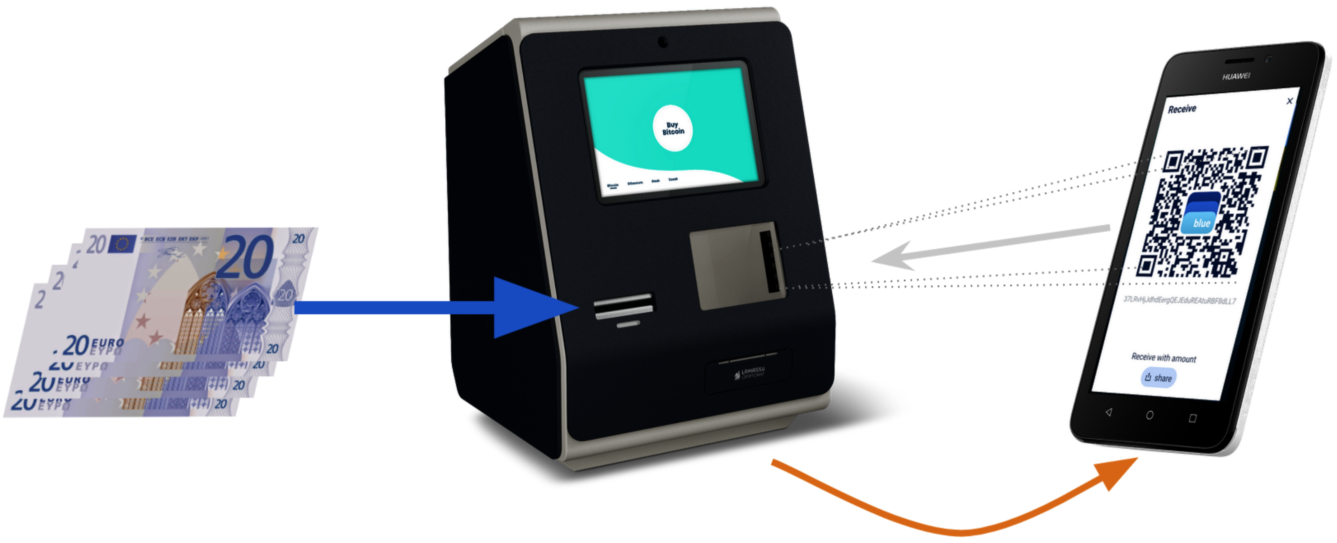


Figure 5. Bitcoin ATM scans the QR code.

Alice will see the transaction from the ATM in the TRANSACTION HISTORY tab of the Eclair wallet. Although Eclair will detect the bitcoin transaction in just a few seconds, it will take approximately one hour for the bitcoin transaction to be "confirmed" on the Bitcoin blockchain. As you can see in Alice receives bitcoin, Alice's Eclair wallet shows "6+ conf" below the transaction, indicating that the transaction has received the required minimum of six confirmations, and her funds are now ready to use.

TIP The number of confirmations on a transaction is the number of blocks mined since (and inclusive of) the block that contained that transaction. Six confirmations is best practice, but different Lightning wallets can consider a channel open after any number of confirmations. Some wallets even scale up the number of expected confirmations by the monetary value of the channel.

Although in this example Alice used an ATM to acquire her first bitcoin, the same basic concepts would apply even if she used one of the other methods in Acquiring Bitcoin. For example, if Alice wanted to sell a product or provide a professional service in exchange for bitcoin, her customers could scan the Bitcoin address with their wallets and pay her in bitcoin.

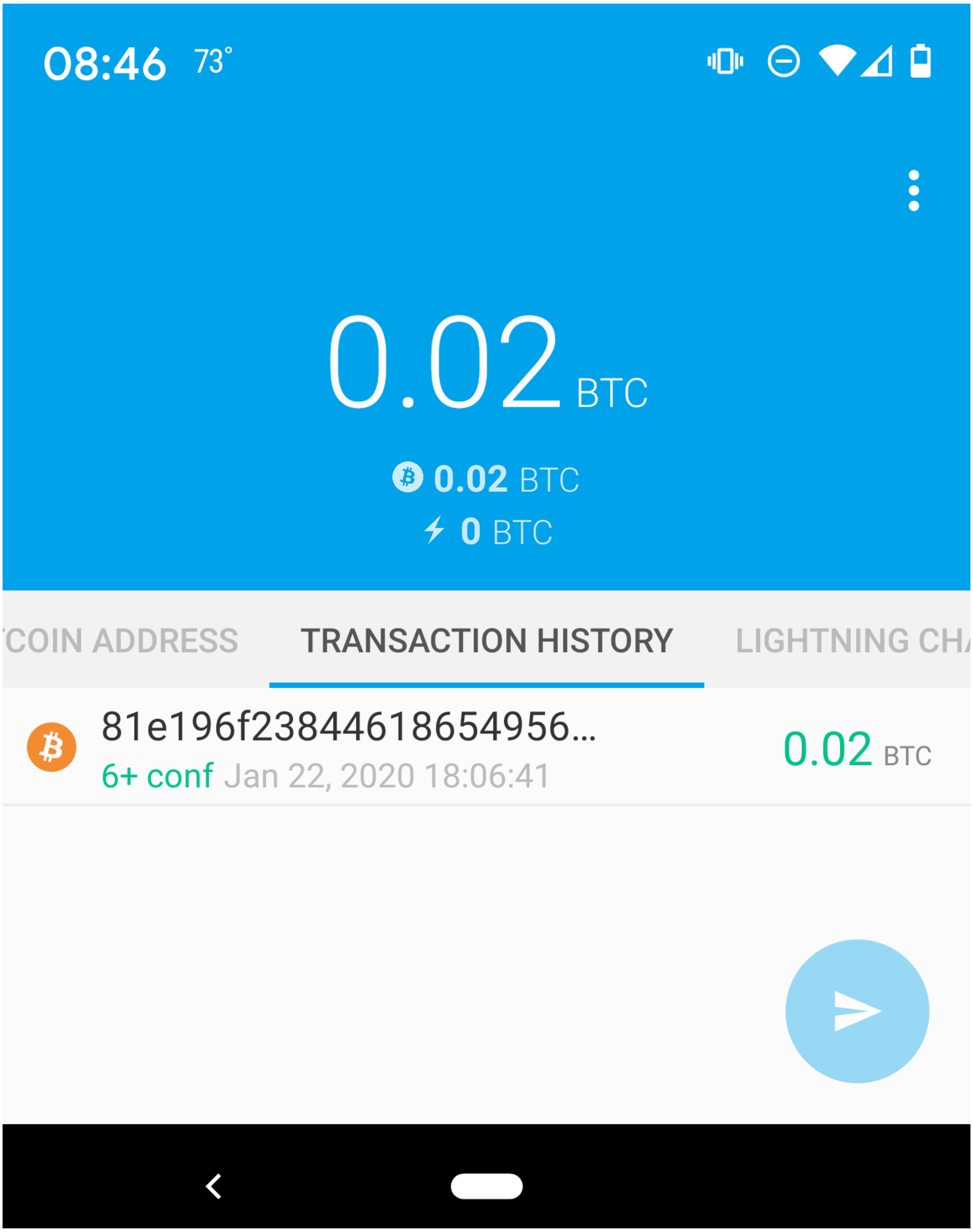


Figure 6. Alice receives bitcoin

Similarly, if she billed a client for a service offered over the internet, Alice could send an email or instant message with the Bitcoin address or the QR code to her client, and they could paste or scan the information into a Bitcoin wallet to pay her.

Alice could even print the QR code and affix it to a sign and display it publicly to receive tips. For example, she could have a QR code affixed to her guitar and receive tips while performing on the street![5]

Finally, if Alice bought bitcoin from a cryptocurrency exchange, she could (and should) "withdraw" the bitcoin by pasting her Bitcoin address into the exchange website. The exchange will then send the bitcoin to her address directly.

From Bitcoin to Lightning Network

Alice’s bitcoin is now controlled by her Eclair wallet and has been recorded on the Bitcoin blockchain. At this point, Alice’s bitcoin is *on-chain*, meaning that the transaction has been broadcast to the entire Bitcoin network, verified by all Bitcoin nodes, and *mined* (recorded) onto the Bitcoin blockchain.

So far, the Eclair Mobile wallet has behaved only as a Bitcoin wallet, and Alice hasn’t used the Lightning Network features of Eclair. As is the case with many Lightning wallets, Eclair bridges Bitcoin and the Lightning Network by acting as both a Bitcoin wallet and a Lightning wallet.

Now, Alice is ready to start using the Lightning Network by taking her bitcoin off-chain to take advantage of the fast, cheap, and private payments that the Lightning Network offers.

Lightning Network Channels

Swiping right, Alice accesses the LIGHTNING CHANNELS section of Eclair. Here she can manage the channels that will connect her wallet to the Lightning Network.

Let’s review the definition of an LN channel at this point, to make things a bit clearer. Firstly, the word "channel" is a metaphor for a *financial relationship* between Alice’s Lightning wallet and another Lightning wallet. We call it a channel because it is a means for Alice’s wallet and this other wallet to exchange many payments with each other on the Lightning Network (off-chain) without committing transactions to the Bitcoin blockchain (on-chain).

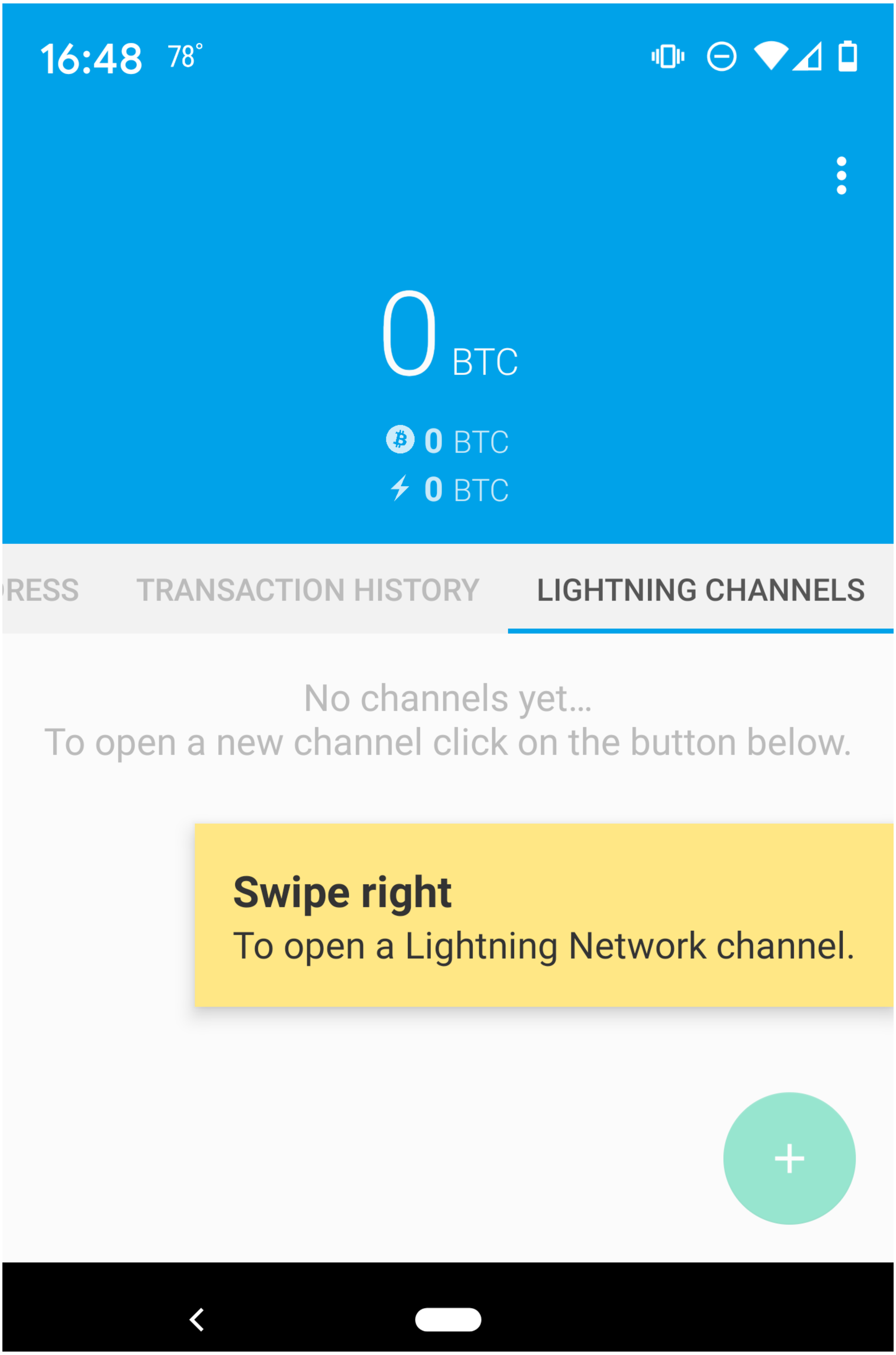
The wallet or *node* that Alice opens a channel to is called her *channel peer*. Once "opened," a channel can be used to send many payments back and forth between Alice’s wallet and her channel peer.

Furthermore, Alice’s channel peer can *forward* payments via other channels further into the Lightning Network. This way, Alice can *route* a payment to any wallet (e.g., Bob’s Lightning wallet) as long as Alice’s wallet can find a viable *path* made by hopping from channel to channel, all the way to Bob’s wallet.

TIP Not all channel peers are *good* peers for routing payments. Well-connected peers will be able to route payments over shorter paths to the destination, increasing the chance of success. Channel peers with ample funds will be able to route larger payments.

In other words, Alice needs one or more channels that connect her to one or more other nodes on the Lightning Network. She doesn’t need a channel to connect her wallet directly to Bob’s Cafe in order to send Bob a payment, though she can choose to open a direct channel, too. Any node in the Lightning Network can be used for Alice’s first channel. The more well-connected a node is, the more people Alice can reach. In this example, since we want to also demonstrate payment routing, we won’t have Alice open a channel directly to Bob’s wallet. Instead, we will have Alice open a channel to a well-connected node and then later use that node to forward her payment, routing it through any other nodes as necessary to reach Bob.

At first, there are no open channels, so as we see in LIGHTNING CHANNELS tab, the LIGHTNING CHANNELS tab displays an empty list. If you notice, in the bottom-right corner there is a plus symbol (+), which is a button to open a new channel.



No channels yet...
To open a new channel click on the button below.

Swipe right
To open a Lightning Network channel.

Figure 7. LIGHTNING CHANNELS tab

Alice presses the plus symbol and is presented with four possible ways to open a channel:

- Paste a node URI
- Scan a node URI
- Random node
- ACINQ node

A "node URI" is a Universal Resource Identifier (URI) that identifies a specific Lightning node. Alice can either paste such a URI from her clipboard or scan a QR code containing that same information. An example of a node URI is shown as a QR code in Node URI as a QR code and then as a text string.

Figure 8. Node URI as a QR code

node URI

```
0237fefbe8626bf888de0cad8c73630e32746a22a2c4faa91c1d9877a3826e1174@1.ln.aantonop.com:9735
```

While Alice could select a specific Lightning node, or use the "Random node" option to have the Eclair wallet select a node at random, she will select the ACINQ Node option to connect to one of ACINQ's well-connected Lightning nodes.

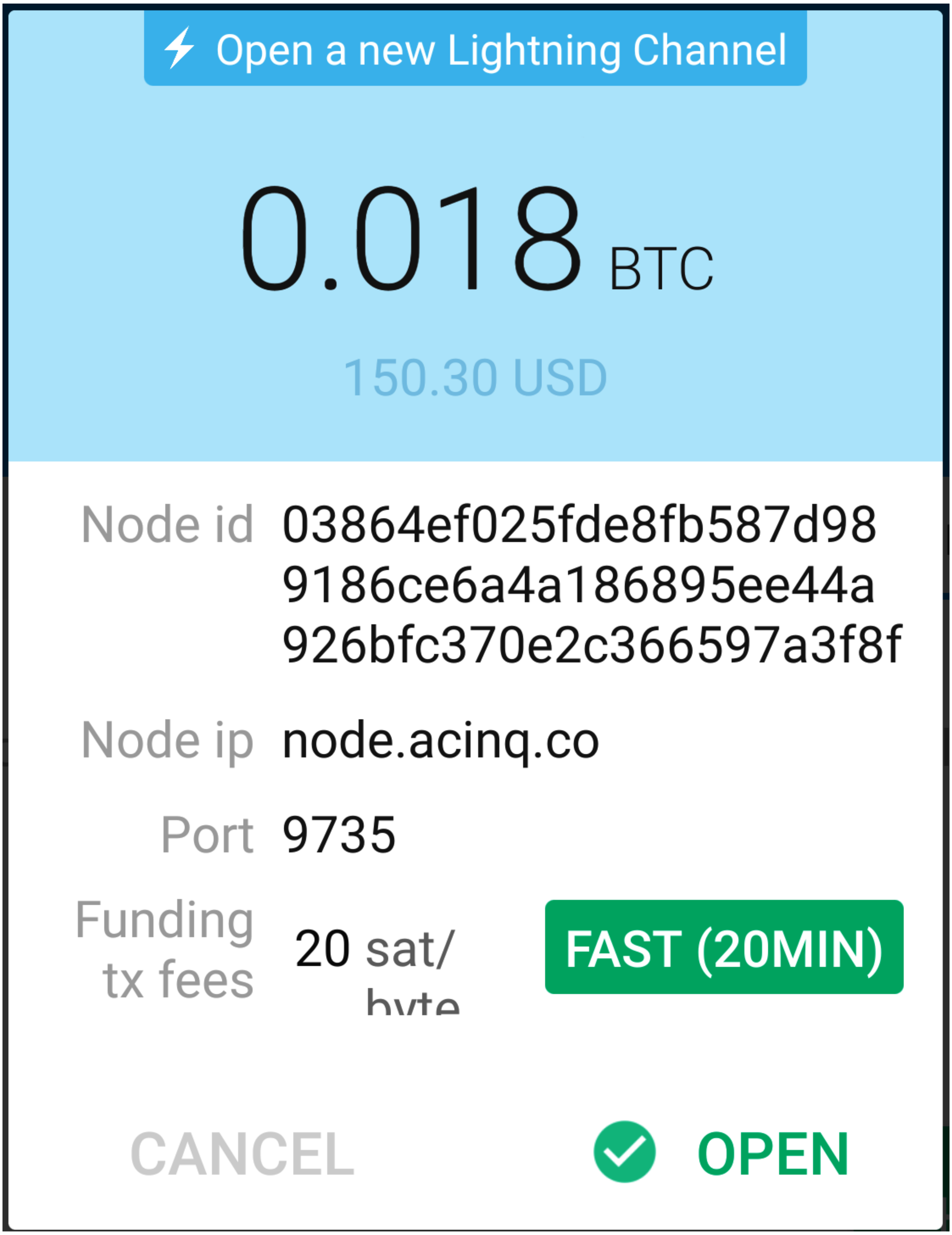
Choosing the ACINQ node will slightly reduce Alice's privacy, because it will give ACINQ the ability to see all of Alice's transactions. It will also create a single point of failure, since Alice will only have one channel, and if the ACINQ node is not available, Alice will not be able to make payments. To keep things simple at first, we will accept these trade-offs. In subsequent chapters, we will gradually learn how to gain more independence and make fewer trade-offs!

Alice selects ACINQ Node and is ready to open her first channel on the Lightning Network.

Opening a Lightning Channel

When Alice selects a node to open a new channel, she is asked to select how much bitcoin she wants to allocate to this channel. In subsequent chapters, we will discuss the implications of these choices, but for now, Alice will allocate almost all her funds to the channel. Since she will have to pay transaction fees to open the channel, she will select an amount slightly less than her total balance.[6]

Alice allocates 0.018 BTC of her 0.020 BTC total to her channel and accepts the default fee rate, as shown in Opening a Lightning channel.


Open a new Lightning Channel

0.018 BTC

150.30 USD

Node id 03864ef025fde8fb587d989186ce6a4a186895ee44a926bfc370e2c366597a3f8f

Node ip node.acinq.co

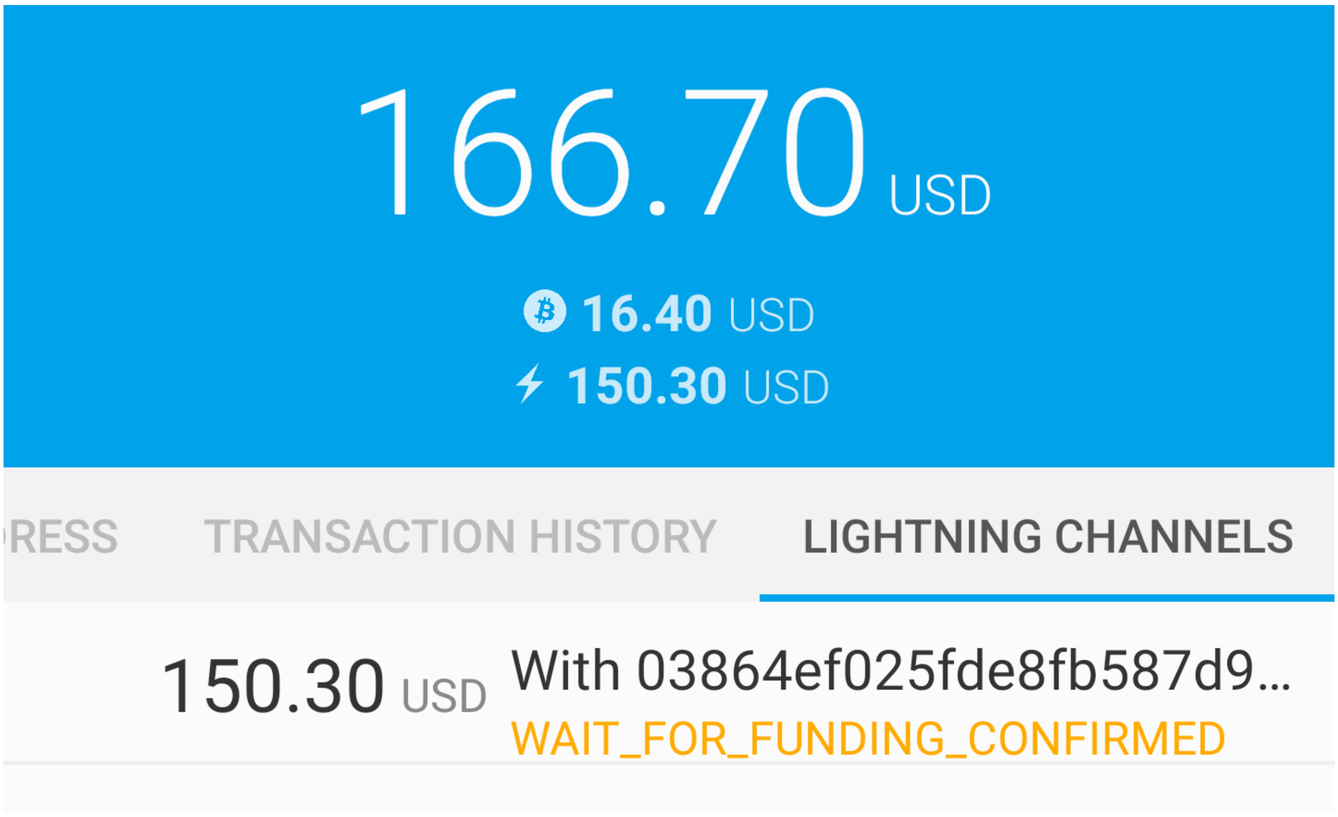
Port 9735

Funding tx fees 20 sat/byte

Figure 9. Opening a Lightning channel

Once she clicks OPEN, her wallet constructs the special Bitcoin transaction that opens a Lightning channel, known as the *funding transaction*. The on-chain funding transaction is sent to the Bitcoin network for confirmation.

Alice now has to wait again (see Waiting for the funding transaction to open the channel) for the transaction to be recorded on the Bitcoin blockchain. As with the initial Bitcoin transaction that she used to acquire her bitcoin, she has to wait for six or more confirmations (approximately one hour).


166.70 USD

16.40 USD

150.30 USD

150.30 USD With 03864ef025fde8fb587d9…

WAIT_FOR_FUNDING_CONFIRMED

Figure 10. Waiting for the funding transaction to open the channel

Once the funding transaction is confirmed, Alice’s channel to the ACINQ node is open, funded, and ready, as shown in Channel is open.

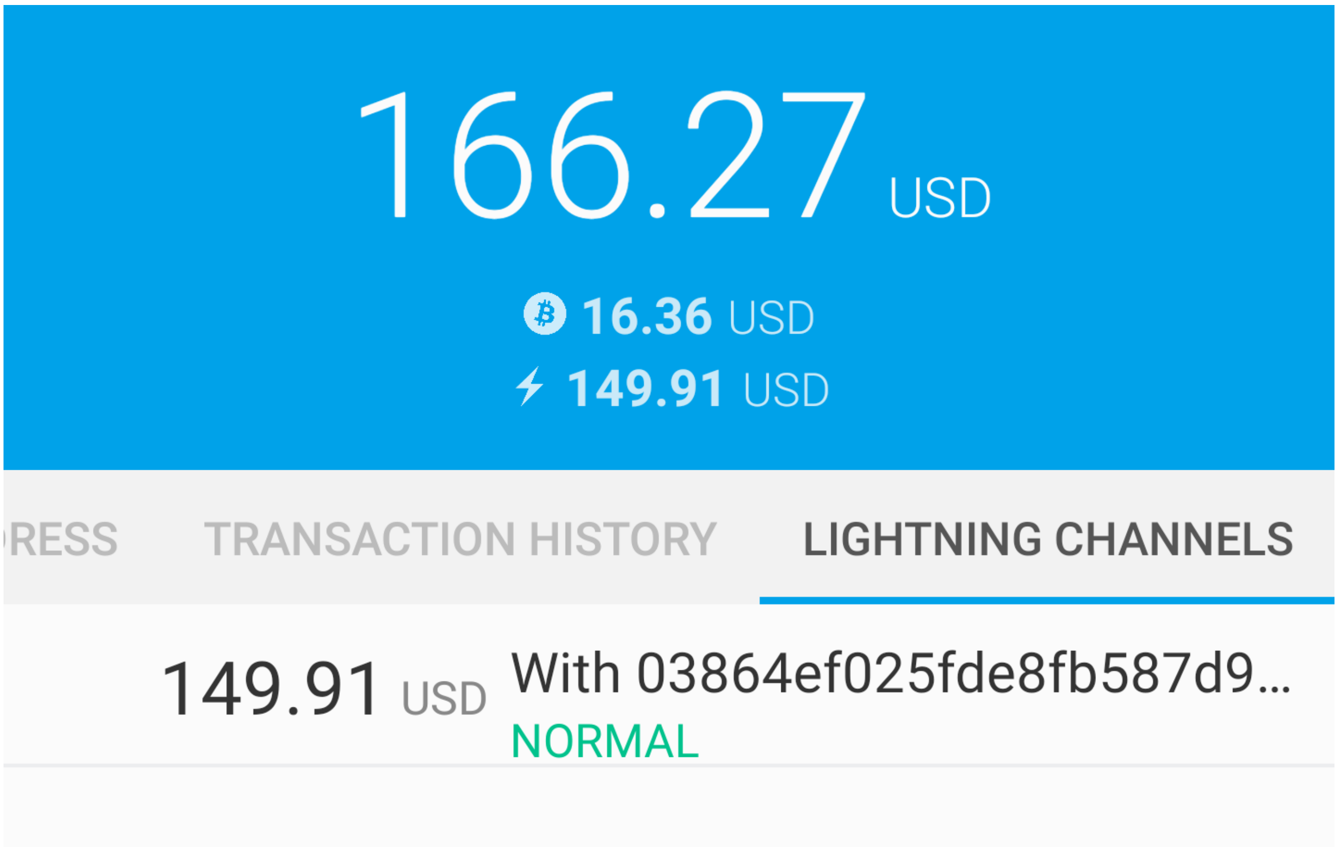


Figure 11. Channel is open

TIP Did you notice that the channel amount seems to have changed? It hasn't: the channel contains 0.018 BTC, but in the time between screenshots, the BTC exchange rate changed, so the USD value is different. You can choose to show balances in BTC or USD, but keep in mind that USD values are calculated in real time and will change!

Buying a Cup of Coffee Using the Lightning Network

Alice now has everything ready to start using the Lightning Network. As you can see, it took a bit of work and a bit of time waiting for confirmations. However, now subsequent actions are fast and easy. The Lightning Network enables payments without having to wait for confirmations, as funds get settled in seconds.

Alice grabs her mobile device and runs to Bob's Cafe in her neighborhood. She is excited to try her new Lightning wallet and use it to buy something!

Bob's Cafe

Bob has a simple point-of-sale (PoS) application for the use of any customer who wants to pay with bitcoin over the Lightning Network. As we will see in the next chapter, Bob uses the popular open source platform *BTCPay Server* which contains all the necessary components for an ecommerce or retail solution, such as:

- A Bitcoin node using the Bitcoin Core software
- A Lightning node using the c-lightning software
- A simple PoS application for a tablet

BTCPay Server makes it simple to install all the necessary software, upload pictures and product prices, and launch a store quickly.

On the counter at Bob’s Cafe, there is a tablet device showing what you see in Bob’s point-of-sale application.

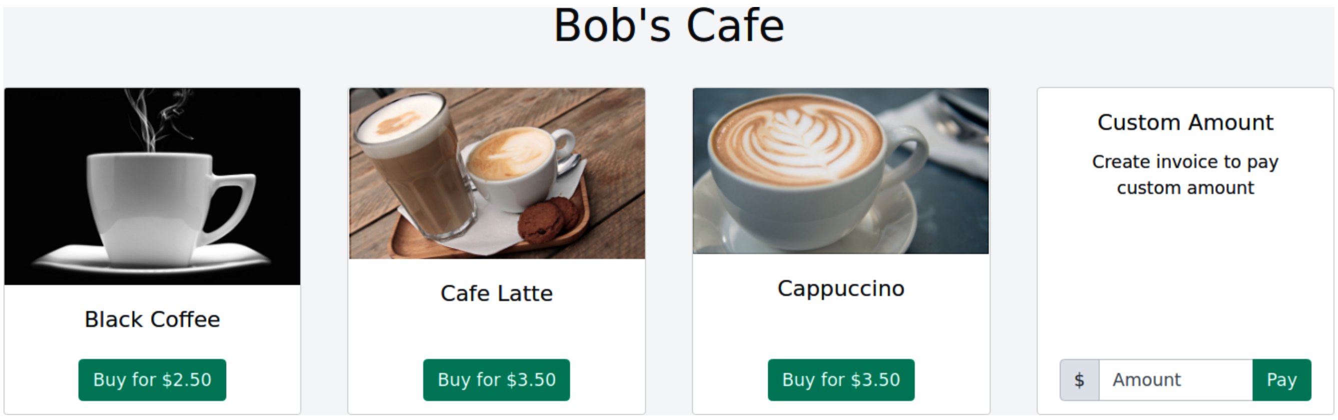
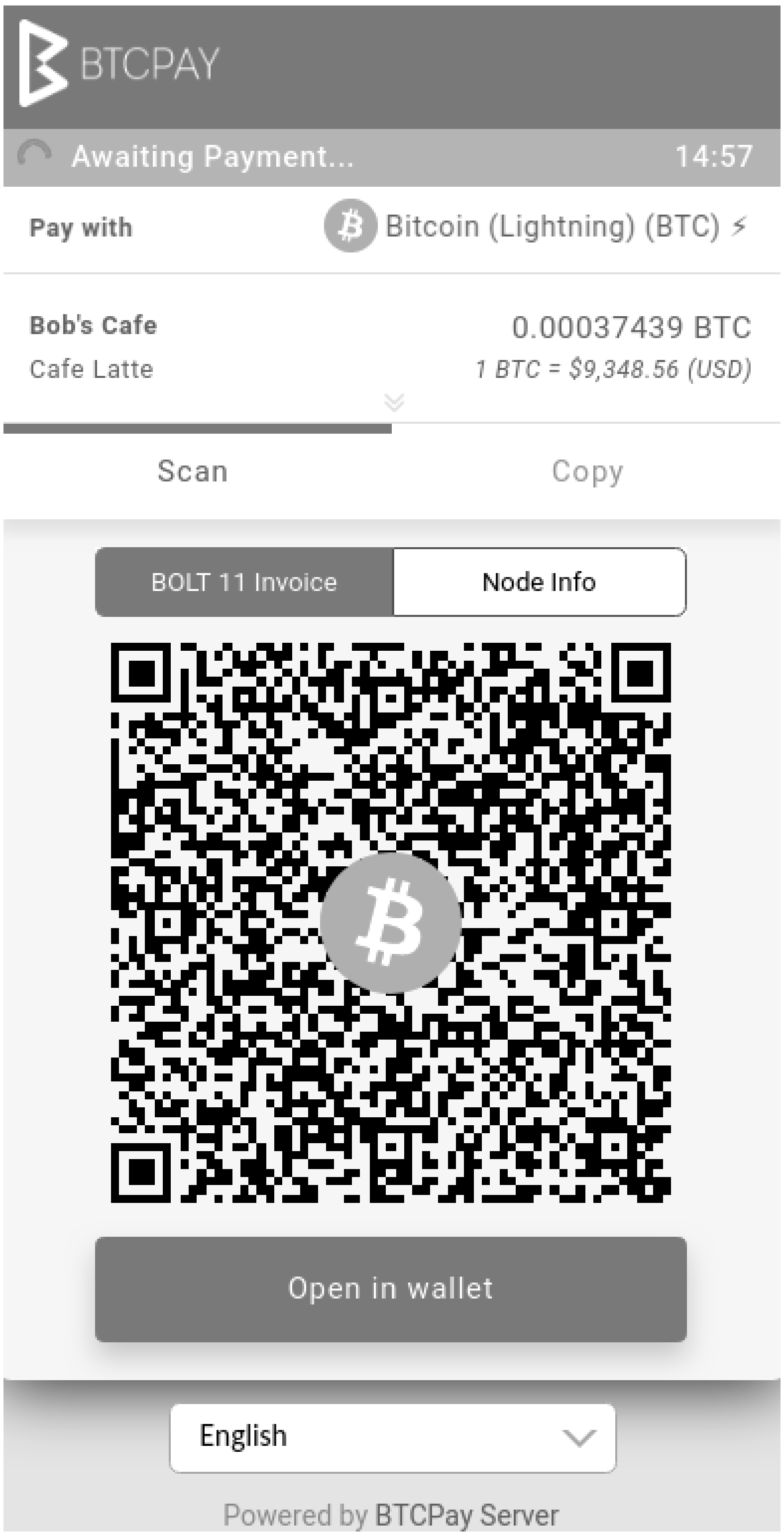


Figure 12. Bob’s point-of-sale application

A Lightning Invoice

Alice selects the Cafe Latte option from the screen and is presented with a *Lightning invoice* (also known as a "payment request"), as shown in Lightning invoice for Alice’s latte.

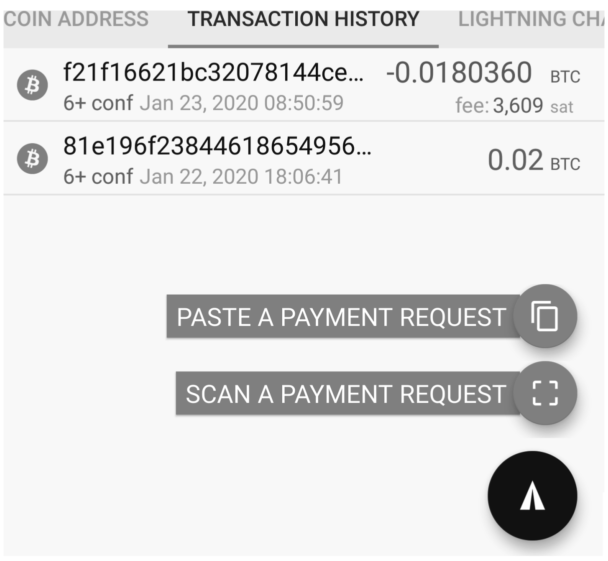


Pay with: Bitcoin (Lightning) (BTC)

Bob's Cafe — 0.00037439 BTC
Cafe Latte — 1 BTC = $9,348.56 (USD)

Figure 13. Lightning invoice for Alice's latte

To pay the invoice, Alice opens her Eclair wallet and selects the Send button (which looks like an up-facing arrow) under the TRANSACTION HISTORY tab, as shown in Alice selecting Send.



f21f16621bc32078144ce… -0.0180360 BTC
6+ conf Jan 23, 2020 08:50:59 fee: 3,609 sat

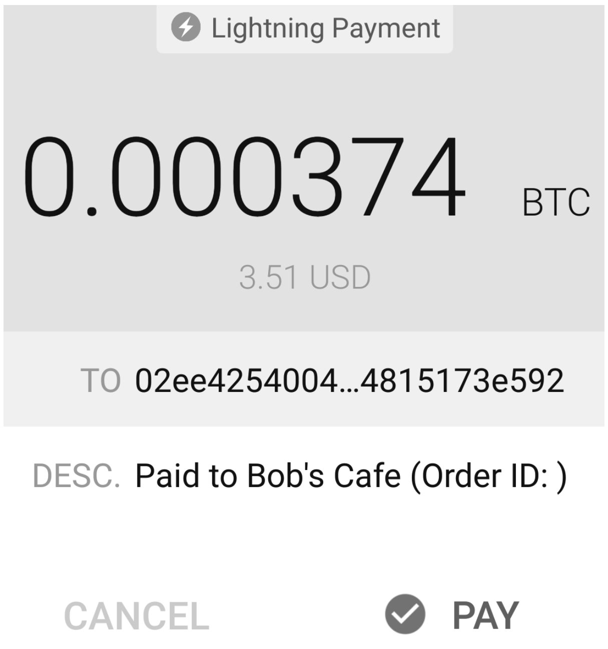
81e196f23844618654956… 0.02 BTC
6+ conf Jan 22, 2020 18:06:41

Figure 14. Alice selecting Send

TIP The term "payment request" can refer to a Bitcoin payment request or a Lightning invoice, and the terms "invoice" and "payment request" are often used interchangeably. The correct technical term is "Lightning invoice," regardless of how it is named in the wallet.

Alice selects the option to "scan a payment request," scans the QR code displayed on the screen of the tablet (see Lightning invoice for Alice's latte), and is prompted to confirm her payment, as shown in Alice's send confirmation.

Alice presses PAY, and a second later, Bob's tablet shows a successful payment. Alice has completed her first LN payment! It was fast, inexpensive, and easy. Now she can enjoy her latte which was purchased using bitcoin through a payment system that is fast, cheap, and decentralized. From now on, Alice can simply select an item on Bob's tablet screen, scan the QR code with her cell phone, click PAY, and be served a coffee, all within seconds and all without an on-chain transaction.



0.000374 BTC

3.51 USD

TO 02ee4254004…4815173e592

DESC. Paid to Bob's Cafe (Order ID:)

Figure 15. Alice's send confirmation

Lightning payments are better for Bob, too. He’s confident that he will be paid for Alice’s latte without waiting for an on-chain confirmation. In the future, whenever Alice feels like drinking a coffee at Bob’s Cafe, she can choose to pay with bitcoin on the Bitcoin network or the Lightning Network. Which one do you think she will choose?

Conclusion

In this chapter, we followed Alice as she downloaded and installed her first Lightning wallet, acquired and transferred some bitcoin, opened her first Lightning channel, and bought a cup of coffee by making her first payment on the Lightning Network. In the following chapters, we will look "under the covers" at how each component in the Lightning Network works and how Alice’s payment reached Bob’s Cafe.

How the Lightning Network Works

Now that we've followed Alice as she set up a Lightning wallet and purchased a coffee from Bob, we'll look under the hood and unpack the different components of the Lightning Network involved in that process. This chapter will give a high-level overview and will not delve into all the technical details. The goal is rather to help you to become aware of the most important concepts and building blocks of the Lightning Network.

If you have experience in computer science, cryptography, Bitcoin, and protocol development, then this chapter should be enough for you to be able to fill out the connecting details by yourself. If you are less experienced, this chapter will give you a good enough overview so you have an easier time understanding the formal protocol specifications, known as BOLTs (Basis of Lightning Technology). If you are a beginner, this chapter will help you better understand the technical chapters of the book.

If you need a refresher on the fundamentals of Bitcoin, you can find a summary review of the following topics in Bitcoin Fundamentals Review:

- Keys and addresses
- Hash functions
- Digital signatures
- Transaction structure
- Transaction inputs and outputs
- Transaction chaining
- Bitcoin Script
- Multisignature addresses and scripts
- Timelocks
- Complex scripts

We'll start with a one-sentence definition of what the Lightning Network is and break it down in the remainder of this chapter.

The Lightning Network is a peer-to-peer network of *payment channels* implemented as smart contracts on the *Bitcoin blockchain* as well as a communication protocol that defines how participants set up and execute these smart contracts.

What Is a Payment Channel?

There are several ways to describe a payment channel, depending on the context. Let's start at a high level and then add some more detail.

A payment channel is a *financial relationship* between two nodes on the Lightning Network, called the *channel partners*. The financial relationship allocates a *balance of funds* (denominated in millisatoshis), between the two channel partners.

The payment channel is managed by a *cryptographic protocol*, meaning a predefined process based on cryptography is used by the channel partners to redistribute the balance of the channel in favor of one or the other channel partner. The cryptographic protocol ensures that one channel partner cannot cheat the other, so that the partners do not need to trust each other.

The cryptographic protocol is established by the funding of a 2-of-2 *multisignature address* that requires the two channel partners to cooperate and prevents either channel partner from spending the funds unilaterally.

To summarize: a payment channel is a financial relationship between nodes, allocating funds from a multisignature address through a strictly defined cryptographic protocol.

Payment Channel Basics

Underlying the payment channel is simply a 2-of-2 multisignature address on the Bitcoin blockchain, for which you hold one key and your channel partner holds the other key.

You and your channel partner negotiate a sequence of transactions that spend from this multisignature address. Instead of transmitting and recording these transactions on the Bitcoin blockchain, you both hold on to them, unspent.

The latest transaction in that sequence encodes the balance of the channel and defines how that balance is divided between you and your channel partner.

Thus, adding a new transaction to this sequence is equivalent to moving some part of the channel balance from one channel partner to the other, without the Bitcoin network being aware of it. As you negotiate each new transaction, changing the allocation of funds in the channel, you also revoke the previous transaction, so that neither party can regress to a previous state.

Each transaction in the sequence makes use of Bitcoin’s scripting language, and thus the negotiation of funds between you and your channel partner is managed by a Bitcoin smart contract. The smart contract is set up to penalize a channel member if they try to submit a previously revoked state of the channel.

NOTE

If you have an unpublished transaction from a 2-of-2 multisignature address that pays you part of the balance, then a signature from the other party ensures that you can independently publish this transaction anytime by adding your own signature.

The ability to hold a partially signed transaction, offline and unpublished, with the option to publish and own that balance at any time, is the basis of the Lightning Network.

Routing Payments Across Channels

Once several participants have channels from one party to another, payments can also be "forwarded" from payment channel to payment channel by setting up a *path* across the network connecting several payment channels together.

For example, Alice can send money to Charlie if Alice has a channel with Bob and Bob has a channel with Charlie.

By the design of the Lightning Network, it is possible to extend the smart contracts that operate the channel so that Bob has no way of stealing the funds that are being forwarded through his channel.

In the same way that the smart contract protects the channel partners so they don't need to trust each other, the entire network protects the participants so that they can forward payments without trusting any of the other participants.

Because the channels are constructed from multisignature addresses and the balance update transactions are presigned Bitcoin transactions, all the trust that is needed to operate the Lightning Network comes from the trust in the decentralized Bitcoin network!

The aforementioned innovations are certainly the major breakthroughs that allowed the creation of the Lightning Network. However, the Lightning Network is so much more than the cryptographic protocols on top of the Bitcoin Script language. It is a comprehensive communication protocol that allows peers to exchange Lightning messages to achieve the transfer of bitcoin. The communication protocol defines how Lightning messages are encrypted and exchanged.

The Lightning Network also uses a gossip protocol to distribute public information about the channels (network topology) to all the participants.

Alice, for example, needs the network topology information to be aware of the channel between Bob and Charlie, so that she can construct a route to Charlie.

Last but not least, it is important to understand that the Lightning Network is nothing more than an application on top of Bitcoin, using Bitcoin transactions and Bitcoin Script. There is no "Lightning coin" or "Lightning blockchain." Beyond all the technical primitives, the LN protocol is a creative way to get more benefits out of Bitcoin by allowing an arbitrary amount of instant payments with instant settlements without the necessity of having to trust anyone else but the Bitcoin network.

Payment Channels

As we saw in the previous chapter, Alice used her wallet software to create a payment channel between herself and another LN participant.

A channel is only limited by three things:

- First, the time it takes for the internet to transfer the few hundred bytes of data that the protocol requires to move funds from one end of the channel to the other
- Second, the capacity of the channel, meaning the amount of bitcoin that is committed to the channel when it is opened
- Third, the maximum size limit of a Bitcoin transaction also limits the number of incomplete (in progress) routed payments that can be carried simultaneously over a channel

Payment channels have a few very interesting and useful properties:

- Because the time to update a channel is primarily bound by the communication speed of the internet, making a payment on a payment channel can be almost instant.

- If the channel is open, making a payment does not require the confirmation of Bitcoin blocks. In fact—as long as you and your channel partner follow the protocol—it does not require any interaction with the Bitcoin network or anyone else other than your channel partner.
- The cryptographic protocol is constructed such that there is little to no trust needed between you and your channel partner. If your partner becomes unresponsive or tries to cheat you, you can ask the Bitcoin system to act as a "court," resolving the smart contract you and your partner have previously agreed upon.
- Payments made in a payment channel are only known to you and your partner. In that sense, you gain privacy compared to Bitcoin, where every transaction is public. Only the final balance, which is the aggregate of all payments in that channel, will become visible on the Bitcoin blockchain.

Bitcoin was about five years old when talented developers first figured out how bidirectional, indefinite lifetime, routable payment channels could be constructed, and by now there are at least three different known methods.

This chapter will focus on the channel construction method first described in the Lightning Network whitepaper by Joseph Poon and Thaddeus Dryja in 2015. These are known as *Poon-Dryja* channels, and are the channel construction method currently used in the Lightning Network. The other two proposed methods are *Duplex Micropayment* channels, introduced by Christian Decker around the same time as the Poon-Dryja channels and *eltoo* channels, introduced in "eltoo: A Simple Layer2 Protocol for Bitcoin" by Christian Decker, Rusty Russel, and (coauthor of this book) Olaoluwa Osuntokun in 2018.

eltoo channels have some interesting properties and simplify the implementation of payment channels. However, eltoo channels require a change in the Bitcoin Script language and therefore cannot be implemented on the Bitcoin mainnet as of 2020.

Multisignature Address

Payment channels are built on top of 2-of-2 multisignature addresses.

In summary, a multisignature address is where bitcoin is locked so that it requires multiple signatures to unlock and spend. In a 2-of-2 multisignature address, as used in the Lightning Network, there are two participating signers and *both* need to sign to spend the funds.

Multisignature scripts and addresses are explained in more detail in Multisignature Scripts.

Funding Transaction

The fundamental building block of a payment channel is a 2-of-2 multisignature address. One of the two channel partners will fund the payment channel by sending bitcoin to the multisignature address. This transaction is called the *funding transaction*, and is recorded on the Bitcoin blockchain.[7]

Even though the funding transaction is public, it is not obvious that it is a Lightning payment channel until it is closed unless the channel is publicly advertised. Channels are typically publicly announced by routing nodes that wish to forward payments. However, nonadvertised channels also exist, and are usually created by mobile nodes that don’t actively participate in routing. Furthermore, channel payments are still not visible to anyone other than the channel partners, nor is the distribution of the channel balance between them.

The amount deposited in the multisignature address is called the *channel capacity* and sets the maximum amount that can be sent across the payment channel. However, since funds can be sent back and forth, the channel capacity is not the upper limit on how much value can flow across the channel. That’s because if the channel capacity is exhausted with payments in one direction, it can be used to send payments in the opposite direction again.

NOTE

The funds sent to the multisignature address in the funding transaction are sometimes referred to as "locked in a Lightning channel." However, in practice, funds in a Lightning channel are not "locked" but rather "unleashed." Lightning channel funds are more liquid than funds on the Bitcoin blockchain, as they can be spent faster, cheaper, and more privately. There are some disadvantages to moving funds into the Lightning Network (such as the need to keep them in a "hot" wallet), but the idea of "locking funds" in Lightning is misleading.

Example of a poor channel opening procedure

If you think carefully about 2-of-2 multisignature addresses, you will realize that putting your funds into such an address seems to carry some risk. What if your channel partner refuses to sign a transaction to release the funds? Are they stuck forever? Let’s now look at that scenario and how the LN protocol avoids it.

Alice and Bob want to create a payment channel. They each create a private/public key pair and then exchange public keys. Now, they can construct a multisignature 2-of-2 with the two public keys, forming the foundation for their payment channel.

Next, Alice constructs a Bitcoin transaction sending a few mBTC to the multisignature address created from Alice’s and Bob’s public keys. If Alice doesn’t take any additional steps and simply broadcasts this transaction, she has to trust that Bob will provide his signature to spend from the multisignature address. Bob, on the other hand, has the chance to blackmail Alice by withholding his signature and denying Alice access to her funds.

To prevent this, Alice will need to create an additional transaction that spends from the multisignature address, refunding her mBTC. Alice then has Bob sign the refund transaction *before* broadcasting her funding transaction to the Bitcoin network. This way, Alice can get a refund even if Bob disappears or fails to cooperate.

The "refund" transaction that protects Alice is the first of a class of transactions called *commitment transactions*, which we will examine in more detail next.

Commitment Transaction

A *commitment transaction* is a transaction that pays each channel partner their channel balance and ensures that the channel partners do not have to trust each other. By signing a commitment transaction, each channel partner "commits" to the current balance and gives the other channel partner the ability to get their funds back whenever they want.

By holding a signed commitment transaction, each channel partner can get their funds even without the cooperation of the other channel partner. This protects them against the other channel partner's disappearance, refusal to cooperate, or attempt to cheat by violating the payment channel protocol.

The commitment transaction that Alice prepared in the previous example was a refund of her initial payment to the multisignature address. More generally, however, a commitment transaction splits the funds of the payment channel, paying the two channel partners according to the distribution (balance) they each hold. At first, Alice holds all the balance, so it is a simple refund. But as funds flow from Alice to Bob, they will exchange signatures for new commitment transactions that represent the new balance distribution, with some part of the funds paid to Alice and some paid to Bob.

Let's assume that Alice opens a channel with a capacity of 100,000 satoshi with Bob. Initially, Alice owns 100,000 satoshi, the entirety of the funds in the channel. Here's how the payment channel protocol works:

1. Alice creates a new private/public key pair and informs Bob that she wishes to open a channel via the `open_channel` message (a message in the LN protocol).
2. Bob also creates a new private/public key pair and agrees to accept a channel from Alice, sending his public key to Alice via the `accept_channel` message.
3. Alice now creates a funding transaction from her wallet that sends 100k satoshi to the multisignature address with a locking script: 2 <PubKey Alice> <PubKey Bob> 2 CHECKMULTISIG.
4. Alice does not yet broadcast this funding transaction but sends Bob the transaction ID in a `funding_created` message along with her signature for Bob's commitment transaction.
5. Both Alice and Bob create their version of a commitment transaction. This transaction will spend from the funding transaction and send all the bitcoin back to an address controlled by Alice.
6. Alice and Bob don't need to exchange these commitment transactions, since they each know how they are constructed and can build both independently (because they've agreed on a canonical ordering of the inputs and outputs). They only need to exchange signatures.
7. Bob provides a signature for Alice's commitment transaction and sends this back to Alice via the `funding_signed` message.
8. Now that signatures have been exchanged, Alice will broadcast the funding transaction to the Bitcoin network.

By following this protocol, Alice does not give up ownership of her 100k satoshi even though the funds are sent to a 2-of-2 multisignature address for which Alice controls only one key. If Bob stops responding to Alice, she will be able to broadcast her commitment transaction and receive her funds back. Her only costs are the fees for the on-chain transactions. As long as she follows the protocol, this is her only risk when opening a channel.

After this initial exchange, commitment transactions are created each time the channel balance changes. In other words, each time a payment is sent between Alice and Bob, new commitment transactions are created and signatures are exchanged. Each new commitment transaction encodes the latest balance between Alice and Bob.

If Alice wants to send 30k satoshi to Bob, both would create a new version of their commitment transactions, which would now pay 70k satoshi to Alice and 30k satoshi to Bob. By encoding a new balance for Alice and Bob, the new commitment transactions are the means by which a payment is "sent" across the channel.

Now that we understand commitment transactions, let’s look at some of the more subtle details. You may notice that this protocol leaves a way for either Alice or Bob to cheat.

Cheating with Prior State

How many commitment transactions does Alice hold after she pays 30k satoshi to Bob? She holds two: the original one paying her 100k satoshi and the more recent one, paying her 70k satoshi and Bob 30k satoshi.

In the channel protocol we have seen so far, nothing stops Alice from publishing a previous commitment transaction. A cheating Alice could publish the commitment transaction that grants her 100k satoshi. Since that commitment transaction was signed by Bob, he can’t prevent Alice from transmitting it.

Some mechanism is needed to prevent Alice from publishing an old commitment transaction. Let’s now find out how this can be achieved and how it enables the Lightning Network to operate without requiring any trust between Alice and Bob.

Because Bitcoin is censorship resistant, no one can prevent someone from publishing an old commitment transaction. To prevent this form of cheating, commitment transactions are constructed so that if an old one is transmitted, the cheater can be punished. By making the penalty large enough, we create a strong incentive against cheating, and this makes the system secure.

The way the penalty works is by giving the cheated party an opportunity to claim the balance of the cheater. So if someone attempts to cheat by broadcasting an old commitment transaction, in which they are paid a higher balance than they are due, the other party can punish them by taking *both* their own balance and the balance of the cheater. The cheater loses everything.

TIP You might notice that if Alice drains her channel balance almost completely, she could then try cheating with little risk. Bob’s penalty wouldn’t be so painful if her channel balance is low. To prevent this, the Lightning protocol requires each channel partner to keep a minimum balance in the channel (called the *reserve*) so they always have "skin in the game."

Let us go through the channel construction scenario again, adding a penalty mechanism to protect against cheating:

1. Alice creates a channel with Bob and puts 100k satoshi into it.
2. Alice sends 30k satoshi to Bob.
3. Alice tries to cheat Bob out of his earned 30k satoshi by publishing an old commitment transaction claiming the full 100k satoshi for herself.

4. Bob detects the fraud and punishes Alice by taking the full 100k satoshi for himself.
5. Bob ends up with 100k satoshi, gaining 70k satoshi for catching Alice cheating.
6. Alice ends up with 0 satoshi.
7. Trying to cheat Bob out of 30k satoshi, she loses the 70k satoshi she owned.

With a strong penalty mechanism, Alice is not tempted to cheat by publishing an old commitment transaction because she risks losing her entire balance.

NOTE In Chapter 12 of *Mastering Bitcoin*, Andreas Antonopoulos (the coauthor of this book) states it as follows: "A key characteristic of Bitcoin is that once a transaction is valid, it remains valid and does not expire. The only way to cancel a transaction is by double-spending its inputs with another transaction before it is mined."

Now that we understand *why* a penalty mechanism is needed and how it will prevent cheating, let's see *how* it works in detail.

Usually, the commitment transaction has at least two outputs, paying each channel partner. We change this to add a *timelock delay* and a *revocation secret* to one of the payments. The timelock prevents the owner of the output from spending it immediately once the commitment transaction is included in a block. The revocation secret allows either party to immediately spend that payment, bypassing the timelock.

So, in our example, Bob holds a commitment transaction that pays Alice *immediately*, but his own payment is delayed and revocable. Alice also holds a commitment transaction, but hers is the opposite: it pays Bob immediately but her own payment is delayed and revocable.

The two channel partners hold half of the revocation secret, so that neither one knows the whole secret. If they share their half, then the other channel partner has the full secret and can use it to exercise the revocation condition. When signing a new commitment transaction, each channel partner revokes the previous commitment by giving the other party their half of the revocation secret.

We will examine the revocation mechanism in more detail in Revoking and Recommitting, where we will learn the details of how revocation secrets are constructed and used.

In simple terms, Alice signs Bob's new commitment transaction only if Bob offers his half of the revocation secret for the previous commitment. Bob only signs Alice's new commitment transaction if she gives him her half of the revocation secret from the previous commitment.

With each new commitment, they exchange the necessary "punishment" secret that allows them to effectively *revoke* the prior commitment transaction by making it unprofitable to transmit. Essentially, they destroy the ability to use old commitments as they sign the new ones. What we mean is that while it is still technically possible to use old commitments, the penalty mechanism makes it economically irrational to do so.

The timelock is set to a number of blocks up to 2,016 (approximately two weeks). If either channel partner publishes a commitment transaction without cooperating with the other partner, they will have to wait for that number of blocks (e.g., two weeks) to claim their balance. The other channel partner can claim their own balance at any time. Furthermore, if the commitment they published was previously revoked, the channel partner can *also* immediately claim the cheating party's balance, bypassing the timelock and punishing the cheater.

The timelock is adjustable and can be negotiated between channel partners. Usually, it is longer for larger capacity channels, and shorter for smaller channels, to align the incentives with the value of the funds.

For every new update of the channel balance, new commitment transactions and new revocation secrets have to be created and saved. As long as a channel remains open, all revocation secrets *ever created* for the channel need to be kept because they might be needed in the future. Fortunately, the secrets are rather small and it is only the channel partners who need to keep them, not the entire network. Furthermore, due to a smart derivation mechanism used to derive revocation secrets, we only need to store the most recent secret, because previous secrets can be derived from it (see Cheating and Penalty in Practice).

Nevertheless, managing and storing the revocation secrets is one of the more elaborate parts of Lightning nodes that require node operators to maintain backups.

NOTE Technologies such as watchtower services or changing the channel construction protocol to the eltoo protocol might be future strategies to mitigate these issues and reduce the need for revocation secrets, penalty transactions, and channel backups.

Alice can close the channel at any time if Bob does not respond, claiming her fair share of the balance. After publishing the *last* commitment transaction on-chain, Alice has to wait for the timelock to expire before she can spend her funds from the commitment transaction. As we will see later, there is an easier way to close a channel without waiting, as long as Alice and Bob are both online and cooperate to close the channel with the correct balance allocation. But the commitment transactions stored by each channel partner act as a fail-safe, ensuring they do not lose funds if there is a problem with their channel partner.

Announcing the Channel

Channel partners can agree to announce their channel to the whole Lightning Network, making it a *public channel*. To announce the channel, they use the Lightning Network's gossip protocol to tell other nodes about the existence, capacity, and fees of the channel.

Announcing channels publicly allows other nodes to use them for payment routing, thereby also generating routing fees for the channel partners.

By contrast, the channel partners may decide not to announce the channel, making it an *unannounced* channel.

NOTE You may hear the term "private channel" used to describe an unannounced channel. We avoid using that term because it is misleading and creates a false sense of privacy. Although an unannounced channel will not be known to others while it is in use, its existence and capacity will be revealed when the channel closes because those details will be visible on-chain in the final settlement transaction. Its existence can also leak in a variety of other ways, so we avoid calling it "private."

Unannounced channels are still used to route payments but only by the nodes that are aware of their existence, or given "routing hints" about a path that includes an unannounced channel.

When a channel and its capacity are publicly announced using the gossip protocol, the announcement can also include information about the channel (metadata), such as its routing fees and timelock duration.

When new nodes join the Lightning Network, they collect the channel announcements propagated via the gossip protocol from their peers, building an internal map of the Lightning Network. This map can then be used to find paths for payments, connecting channels together end-to-end.

Closing the Channel

The best way to close a channel is…to not close it! Opening and closing channels requires an on-chain transaction, which will incur transaction fees. So it's best to keep channels open as long as possible. You can keep using your channel to make and forward payments, as long as you have sufficient capacity on your end of the channel. But even if you send all the balance to the other end of the channel, you can then use the channel to receive payments from your channel partner. This concept of using a channel in one direction and then using it in the opposite direction is called "rebalancing," and we will examine it in more detail in another chapter. By rebalancing a channel, it can be kept open almost indefinitely and used for an essentially unlimited number of payments.

However, sometimes closing a channel is desirable or necessary. For example:

- You want to reduce the balance held on your Lightning channels for security reasons and want to send funds to "cold storage."
- Your channel partner becomes unresponsive for a long time and you cannot use the channel anymore.
- The channel is not being used often because your channel partner is not a well-connected node, so you want to use the funds for another channel with a better-connected node.
- Your channel partner has breached the protocol either due to a software bug or on purpose, forcing you to close the channel to protect your funds.

There are three ways to close a payment channel:

- Mutual close (the good way)
- Force close (the bad way)
- Protocol breach (the ugly way)

Each of these methods is useful for different circumstances, which we will explore in the next sections of this chapter. For example, if your channel partner is offline, you will not be able to follow "the good way" because a mutual close cannot be done without a cooperating partner. Usually, your LN software will automatically select the best closing mechanism available under the circumstances.

Mutual close (the good way)

Mutual close is when both channel partners agree to close a channel, and is the preferred method of channel closure.

When you decide that you want to close a channel, your LN node will inform your channel partner about your intention. Now both your node and the channel partner’s node work together to close the channel. No new routing attempts will be accepted from either channel partner, and any ongoing routing attempts will be settled or removed after they time out. Finalizing the routing attempts takes time, so a mutual close can also take some time to complete.

Once there are no pending routing attempts, the nodes cooperate to prepare a *closing transaction*. This transaction is similar to the commitment transaction: it encodes the last balance of the channel, but the outputs are NOT encumbered with a timelock.

The on-chain transaction fees for the closing transaction are paid by the channel partner who opened the channel and not by the one who initiated the closing procedure. Using the on-chain fee estimator, the channel partners agree on the appropriate fee and both sign the closing transaction.

Once the closing transaction is broadcast and confirmed by the Bitcoin network, the channel is effectively closed and each channel partner has received their share of the channel balance. Despite the waiting time, a mutual close is typically faster than a force close.

Force close (the bad way)

A force close is when one channel partner attempts to close a channel without the other channel partner’s consent.

This usually happens when one of the channel partners is unreachable, so a mutual close is not possible. In this case, you would initiate a force close to unilaterally close the channel and "free" the funds.

To initiate a force close, you can simply publish the last commitment transaction your node has. After all, that’s what commitment transactions are for—they offer a guarantee that you don’t need to trust your channel partner to retrieve the balance of your channel.

Once you broadcast the last commitment transaction to the Bitcoin network and it is confirmed, it will create two spendable outputs, one for you and one for your partner. As we discussed previously, the Bitcoin network has no way of knowing if this was the most recent commitment transaction or an old one which was published to steal from your partner. Hence this commitment transaction will give a slight advantage to your partner. The partner who initiated the force close will have their output encumbered by a timelock, and the other partner’s output will be spendable immediately. In the case that you broadcasted an earlier commitment transaction, the timelock delay gives your partner the opportunity to dispute the transaction using the revocation secret and punish you for cheating.

When publishing a commitment transaction during a force close, the on-chain fees will be higher than a mutual close for several reasons:

1. When the commitment transaction was negotiated, the channel partners didn’t know how much the on-chain fees would be at the future time the transaction would be broadcast. Since the fees cannot be changed without changing the outputs of the commitment transaction (which needs both signatures), and since the force close happens when a channel partner is not

available to sign, the protocol developers decided to be very generous with the fee rate included in the commitment transactions. It can be up to five times higher than the fee estimators suggest at the time the commitment transaction is negotiated.

2. The commitment transaction includes additional outputs for any pending routing attempts hash time-locked contracts (HTLCs), which makes the commitment transaction larger (in terms of bytes) than a mutual close transaction. Larger transactions incur more fees.
3. Any pending routing attempts will have to be resolved on-chain, causing additional on-chain transactions.

NOTE Hash time-locked contracts (HTLCs) will be covered in detail in Hash Time-Locked Contracts. For now, assume that these are payments that are routed across the Lightning Network, rather than payments made directly between the two channel partners. These HTLCs are carried as additional outputs in the commitment transactions, thereby increasing the transaction size and on-chain fees.

In general, a force close is not recommended unless absolutely necessary. Your funds will be locked for a longer time and the person who opened the channel will have to pay higher fees. Furthermore, you might have to pay on-chain fees to abort or settle routing attempts even if you didn’t open the channel.

If the channel partner is known to you, you might consider contacting that individual or company to inquire why their Lightning node is down and request that they restart it so that you can achieve a mutual close of the channel.

You should consider a force close only as the last resort.

Protocol breach (the ugly way)

A protocol breach is when your channel partner tries to cheat you, whether deliberately or not, by publishing an outdated commitment transaction to the Bitcoin blockchain, essentially initiating a (dishonest) force close from their side.

Your node must be online and watching new blocks and transactions on the Bitcoin blockchain to detect this.

Because your channel partner’s payment will be encumbered by a timelock, your node has some time to act to detect a protocol breach and publish a *punishment transaction* before the timelock expires.

If you successfully detect the protocol breach and enforce the penalty, you will receive all of the funds in the channel, including your channel partner’s funds.

In this scenario, the channel closure will be rather fast. You will have to pay on-chain fees to publish the punishment transaction, but your node can set these fees according to the fee estimation and not overpay. You will generally want to pay higher fees to guarantee confirmation as soon as possible. However, because you will eventually receive all of the cheater’s funds, it is essentially the cheater who will be paying for this transaction.

If you fail to detect the protocol breach and the timelock expires, you will receive only the funds allocated to you by the commitment transaction your partner published. Any funds you received after this will have been stolen by your partner. If there is any balance allocated to you, you will have to pay on-chain fees to collect that balance.

As with a force close, all pending routing attempts will also have to be resolved in the commitment transaction.

A protocol breach can be executed faster than a mutual close because you do not wait to negotiate a close with your partner, and faster than a force close because you do not need to wait for your timelock to expire.

Game theory predicts that cheating is not an appealing strategy because it is easy to detect a cheater, and the cheater risks losing *all* of their funds while only standing to gain what they had in an earlier state. Furthermore, as the Lightning Network matures, and watchtowers become widely available, cheaters will be detectable by a third party even if the cheated channel partner is offline.

Hence, we do not recommend cheating. We do, however, recommend that anyone catching a cheater punish them by taking their funds.

So, how do you catch a cheat or a protocol breach in your day-to-day activities? You do so by running software that monitors the public Bitcoin blockchain for on-chain transactions that correspond to any commitment transactions for any of your channels. This software is one of three types:

- A properly maintained Lightning node, running 24/7
- A single-purpose watchtower node that you run to watch your channels
- A third-party watchtower node that you pay to watch your channels

Remember that the commitment transaction has a timeout period specified in a given number of blocks, up to a maximum of 2,016 blocks. As long as you run your Lightning node once before the timeout period is reached, it will catch all cheating attempts. It is not advisable to take this kind of risk; it is important to keep a well-maintained node running continuously (see Why Is Reliability Important for Running a Lightning Node?).

Invoices

Most payments on the Lightning Network start with an invoice, generated by the recipient of the payment. In our previous example, Bob creates an invoice to request a payment from Alice.

NOTE There is a way to send an unsolicited payment without an invoice, using a workaround in the protocol called keysend. We will examine this in Keysend Spontaneous Payments.

An invoice is a simple payment instruction containing information such as a unique payment identifier (called a payment hash), a recipient, an amount, and an optional text description.

The most important part of the invoice is the payment hash, which allows the payment to travel across multiple channels in an *atomic* way. Atomic, in computer science, means any action or state change that is either completed successfully or not at all—there is no possibility of an intermediate state or partial action. In the Lightning Network, that means that the payment either travels the whole path or fails completely. It cannot be partially completed such that an intermediate node on the path can receive the payment and keep it. There is no such thing as a "partial payment" or "partly successful payment."

Invoices are not communicated over the Lightning Network. Instead, they are communicated "out of band," using any other communication mechanism. This is similar to how Bitcoin addresses are communicated to senders outside the Bitcoin network: as a QR code, over email, or a text message. For example, Bob can present a Lightning invoice to Alice as a QR code, via email, or through any other message channel.

Invoices are usually encoded either as a long *bech32*-encoded string or as a QR code, to be scanned by a smartphone Lightning wallet. The invoice contains the amount of bitcoin that is requested and a signature of the recipient. The sender uses the signature to extract the public key (also known as the node ID) of the recipient so that the sender knows where to send the payment.

Did you notice how this contrasts with Bitcoin and how different terms are used? In Bitcoin, the recipient passes an address to the sender. In Lightning, the recipient creates an invoice and sends an invoice to the sender. In Bitcoin, the sender sends funds to an address. In Lightning, the sender pays an invoice and the payment gets routed to the recipient. Bitcoin is based on the concept of an "address," and Lightning is a payment network based on the concept of an "invoice." In Bitcoin, we create a "transaction," whereas in Lightning we send a "payment."

Payment Hash and Preimage

The most important part of the invoice is the *payment hash*. When constructing the invoice, Bob will make a payment hash as follows:

1. Bob chooses a random number *r*. This random number is called the *preimage* or *payment secret*.
2. Bob uses SHA-256 to calculate the hash *H* of *r* called the *payment hash*:
   H = SHA-256(*r*).

NOTE The term *preimage* comes from mathematics. In any function $y = f(x)$, the set of inputs that produce a certain value *y* are called the preimage of *y*. In this case, the function is the SHA-256 hash algorithm, and any value *r* that produces the hash *H* is called a preimage.

There is no known way to find the inverse of SHA-256 (i.e., compute a preimage from a hash). Only Bob knows the value *r*, so it is Bob’s secret. But once Bob reveals *r*, anyone who has the hash *H* can check that *r* is the correct secret, by calculating SHA-256(*r*) and seeing that it matches *H*.

The payment process of the Lightning Network is only secure if *r* is chosen completely randomly and is not predictable. This security relies on the fact that hash functions cannot be inverted or feasibly brute-forced and, therefore, no one can find *r* from *H*.

Additional Metadata

Invoices can optionally include other useful metadata such as a short text description. If a user has several invoices to pay, the user can read the description and be reminded of what the invoice is about.

The invoice can also include some *routing hints*, which allow the sender to use unannounced channels to construct a route to the recipient. Routing hints can also be used to suggest public channels, for example, channels known by the recipient to have enough inbound capacity to route the payment.

In case the sender's Lightning node is unable to send the payment over the Lightning Network, invoices can optionally include an on-chain Bitcoin address as a fallback.

NOTE While it is always possible to "fall back" to an on-chain Bitcoin transaction, it is actually better to open a new channel to the recipient instead. If you have to incur on-chain fees to make a payment, you might as well incur those fees to open a channel and make the payment over Lightning. After the payment is made, you are left with an open channel that has liquidity on the recipient's end and can be used to route payments back to your Lightning node in the future. One on-chain transaction gives you a payment and a channel for future use.

Lightning invoices contain an expiry date. Since the recipient must keep the preimage *r* for every invoice issued, it is useful to have invoices expire so that these preimages do not need to be kept forever. Once an invoice expires or is paid, the recipient can discard the preimage.

Delivering the Payment

We have seen how the recipient creates an invoice that contains a payment hash. This payment hash will be used to move the payment across a series of payment channels, from sender to recipient, even if they do not have a direct payment channel between them.

In the next few sections, we will dive into the ideas and methods that are being used to deliver a payment over the Lightning Network and use all the concepts we have presented so far.

First, let's look at the Lightning Network's communication protocol.

The Peer-to-Peer Gossip Protocol

As we mentioned previously, when a payment channel is constructed, the channel partners have the option of making it public, announcing its existence and details to the whole Lightning Network.

Channel announcements are communicated over a peer-to-peer *gossip protocol*. A peer-to-peer protocol is a communications protocol in which each node connects to a random selection of other nodes in the network, usually over TCP/IP. Each of the nodes that are directly connected (over TCP/IP) to your node are called your *peers*. Your node, in turn, is one of their peers. Keep in mind that when we say that your node is connected to other peers, we don't mean that you have payment channels, but only that you are connected via the gossip protocol.

After opening a channel, a node may choose to send out an announcement of the channel via the `channel_announcement` message to its peers. Every peer validates the information from the `channel_announcement` message and verifies that the funding transaction is confirmed on the Bitcoin blockchain. After verification, the node will forward the gossip message to its own peers, and they will forward it to their peers, and so on, spreading the announcement across the entire network. To avoid excessive communication, the channel announcement is only forwarded by each node if it has not already forwarded that announcement previously.

The gossip protocol is also used to announce information about known nodes with the `node_announcement` message. For this message to be forwarded, a node has to have at least one public channel announced on the gossip protocol, again to avoid excessive communication traffic.

Payment channels have various metadata that are useful for other participants of the network. This metadata is mainly used for making routing decisions. Because nodes might occasionally change the metadata of their channels, this information is shared in a `channel_update` message. These messages will only be forwarded approximately four times a day (per channel) to prevent excessive communication. The gossip protocol also has a variety of queries and messages to initially synchronize a node with the view of the network or to update the node's view after being offline for a while.

A major challenge for the participants of the Lightning Network is that the topology information being shared by the gossip protocol is only partial. For example, the capacity of the payment channels is shared on the gossip protocol via the `channel_announcement` message. However, this information is not as useful as the actual distribution of the capacity in terms of the local balance between the two channel partners. A node can only forward as much bitcoin as it actually owns (local balance) within that channel.

Although the Lightning Network could have been designed to share balance information of channels and a precise topology, this has not been done for several reasons:

- To protect the privacy of the users, it does not shout out every financial transaction and payment. Channel balance updates would reveal that a payment has moved across the channel. This information could be correlated to reveal all payment sources and destinations.
- To scale the amount of payments that can be conducted with the Lightning Network. Remember that the Lightning Network was created in the first place because notifying every participant about every payment does not scale well. Thus, the Lightning Network cannot be designed in a way that shares channel balance updates among participants.
- The Lightning Network is a dynamic system. It changes constantly and frequently. Nodes are being added, other nodes are being turned off, balances change, etc. Even if everything is always communicated, the information will be valid only for a short amount of time. As a matter of fact, information is often outdated by the time it is received.

We will examine the details of the gossip protocol in a later chapter.

For now, it is only important to know that the gossip protocol exists and that it is used to share topology information of the Lightning Network. This topology information is crucial for delivering payments through the network of payment channels.

Pathfinding and Routing

Payments on the Lightning Network are forwarded along a *path* made of channels linking one participant to another, from the payment source to the payment destination. The process of finding a path from source to destination is called *pathfinding*. The process of using that path to make the payment is called *routing*.

NOTE A frequent criticism of the Lightning Network is that routing is not solved, or even that it is an "unsolvable" problem. In fact, routing is trivial. Pathfinding, on the other hand, is a difficult problem. The two terms are often confused and need to be clearly defined to identify which problem we are attempting to solve.

As we will see next, the Lightning Network currently uses a *source-based* protocol for pathfinding and an *onion-routed* protocol for routing payments. Source-based means that the sender of the payment has to find a path through the network to the intended destination. Onion-routed means that the elements of the path are layered (like an onion), with each layer encrypted so that it can only be seen by one node at a time. We will discuss onion routing in the next section.

Source-Based Pathfinding

If we knew the exact channel balances of every channel, we could easily compute a payment path using any of the standard pathfinding algorithms taught in any computer science class. This could even be solved in a way that optimizes the fees paid to nodes for forwarding the payment.

However, the balance information of all channels is not and cannot be known to all participants of the network. We need more innovative pathfinding strategies.

With only partial information about the network topology, pathfinding is a real challenge, and active research is still being conducted into this part of the Lightning Network. The fact that the pathfinding problem is not "fully solved" in the Lightning Network is a major point of criticism toward the technology.

NOTE One common criticism of pathfinding in the Lightning Network is that it is unsolvable because it is equivalent to the NP-complete *traveling salesperson problem* (TSP), a fundamental problem in computational complexity theory. In fact, pathfinding in Lightning is not equivalent to TSP and falls into a different class of problems. We successfully solve these types of problems (pathfinding in graphs with incomplete information) every time we ask Google to give us driving directions with traffic avoidance. We also successfully solve this problem every time we route a payment on the Lightning Network.

Pathfinding and routing can be implemented in a number of different ways, and multiple pathfinding and routing algorithms can coexist on the Lightning Network, just as multiple pathfinding and routing algorithms exist on the internet. Source-based pathfinding is one of many possible solutions and is successful at the current scale of the Lightning Network.

The pathfinding strategy currently implemented by Lightning nodes is to iteratively try paths until one is found that has enough liquidity to forward the payment. This is an iterative process of trial and error, until success is achieved or no path is found. The algorithm currently does not necessarily result in the path with the lowest fees. While this is not optimal and certainly can be improved, even this simplistic strategy works quite well.

This "probing" is done by the Lightning node or wallet and is not directly seen by the user. The user might only realize that probing is taking place if the payment does not complete instantly.

NOTE On the internet, we use the Internet Protocol and an IP forwarding algorithm to forward internet packages from the sender to the destination. While these protocols have the nice property of allowing internet hosts to collaboratively find a path for information flow through the internet, we cannot reuse and adopt this protocol for forwarding payments on the Lightning Network. Unlike the internet, Lightning payments have to be *atomic*, and channel balances have to remain *private*. Furthermore, the channel capacity in Lightning changes frequently, unlike the internet where connection capacity is relatively static. These constraints require novel strategies.

Of course, pathfinding is trivial if we want to pay our direct channel partner and we have enough balance on our side of the channel to do so. In all other cases, our node uses information from the gossip protocol to do pathfinding. This includes currently known public payment channels, known nodes, known topology (how known nodes are connected), known channel capacities, and known fee policies set by the node owners.

Onion Routing

The Lightning Network uses an *onion routing protocol* similar to the famous Tor (The Onion Router) network. The onion routing protocol used in Lightning is called the *SPHINX Mix Format*,[8] which will be explained in detail in a later chapter.

NOTE Lightning's onion routing SPHINX Mix Format is only similar to the Tor network routing in concept, but both the protocol and the implementation are entirely different from those used in the Tor network.

A payment package used for routing is called an "onion."[9]

Let's use the onion analogy to follow a routed payment. On its route from payment sender (payer) to payment destination (payee) the onion is passed from node to node along the path. The sender constructs the entire onion, from the center out. First, the sender creates the payment information for the (final) recipient of the payment and encrypts it with a layer of encryption that only the recipient can decrypt. Then, the sender wraps that layer with instructions for the node in the path *immediately preceding the final recipient* and encrypts with a layer that only that node can decrypt.

The layers are built up with instructions, working backward until the entire path is encoded in layers. The sender then gives the complete onion to the first node in the path, which can only read the outermost layer. Each node peels a layer, finds instructions inside revealing the next node in the path, and passes the onion on. As each node peels one layer, it can't read the rest of the onion. All it knows is where the onion has just come from and where it is going next, without any indication as to who is the original sender or the ultimate recipient.

This continues until the onion reaches the payment destination (payee). Then, the destination node opens the onion and finds there are no further layers to decrypt and can read the payment information inside.

NOTE Unlike a real onion, when peeling each layer, the nodes add some encrypted padding to keep the size of the onion the same for the next node. As we will see, this makes it impossible for any of the intermediate nodes to know anything about the size (length) of the path, how many nodes are involved in routing, how many nodes preceded them, or how many follow. This increases privacy by preventing trivial traffic analysis attacks.

The onion routing protocol used in Lightning has the following properties:

- An intermediary node can only see on which channel it received an onion and on which channel to forward the onion. This means that no routing node can know who initiated the payment and to whom the payment is destined. This is the most important property, which results in a high degree of privacy.
- The onions are small enough to fit into a single TCP/IP packet and even a link layer (e.g., Ethernet) frame. This makes traffic analysis of the payments significantly more difficult, increasing privacy further.
- The onions are constructed such that they will always have the same length independent of the position of the processing node along the path. As each layer is "peeled," the onion is padded with encrypted "junk" data to keep the size of the onion the same. This prevents intermediary nodes from knowing their position in the path.
- Onions have an HMAC (hash-based message authentication code) at each layer so that manipulations of onions are prevented and practically impossible.
- Onions can have up to around 26 hops, or onion layers if you prefer. This allows for sufficiently long paths. The precise path length available depends on the amount of bytes allocated to the routing payload at each hop.
- The encryption of the onion for every hop uses different ephemeral encryption keys. Should a key (in particular, the private key of a node) leak at some point in time, an attacker cannot decrypt them. In simpler terms, keys are never reused in order to achieve more security.
- Errors can be sent back from the erring node to the original sender, using the same onion-routed protocol. Error onions are indistinguishable from routing onions to external observers and intermediary nodes. Error routing enables the trial-and-error "probing" method used to find a path that has sufficient capacity to successfully route a payment.

Onion routing will be examined in detail in Onion Routing.

Payment Forwarding Algorithm

Once the sender of a payment finds a possible path across the network and constructs an onion, the payment is forwarded by each node in the path. Each node processes one layer of the onion and forwards it to the next node in the path.

Each intermediary node receives a Lightning message called `update_add_htlc` with a payment hash and an onion. The intermediary node executes a series of steps, called the *payment forwarding algorithm*:

1. The node decrypts the outer layer of the onion and checks the message's integrity.
2. It confirms that it can fulfill the routing hints, based on the channel fees and available capacity on the outgoing channel.
3. It works with its channel partner on the incoming channel to update the channel state.
4. It adds some padding to the end of the onion to keep it at a constant length since it removed some data from the beginning.
5. It follows the routing hints to forward the modified onion package on its outgoing payment channel by also sending an `update_add_htlc` message which includes the same payment hash and the onion.
6. It works with its channel partner on the outgoing channel to update the channel state.

Of course, these steps are interrupted and aborted if an error is detected, and an error message is sent back to the originator of the `update_add_htlc` message. The error message is also formatted as an onion and sent backward on the incoming channel.

As the error propagates backward on each channel along the path, the channel partners remove the pending payment, rolling back the payment in the opposite way from which it started.

While the likelihood for a payment failure is high if it does not settle quickly, a node should never initiate another payment attempt along a different path before the onion returns with an error. The sender would pay twice if both payment attempts eventually succeeded.

Peer-to-Peer Communication Encryption

The LN protocol is mainly a peer-to-peer protocol between its participants. As we saw in previous sections, there are two overlapping functions in the network, forming two logical networks that together are *the Lightning Network*:

1. A broad peer-to-peer network that uses a gossip protocol to propagate topology information, where peers randomly connect to each other. Peers don't necessarily have payment channels between them, so they are not always channel partners.
2. A network of payment channels between channel partners. Channel partners also gossip about topology, meaning they are peer nodes in the gossip protocol.

All communication between peers is sent via messages called *Lightning messages*. These messages are all encrypted, using a cryptographic communications framework called the *Noise Protocol Framework*. The Noise Protocol Framework allows the construction of cryptographic communication protocols that offer authentication, encryption, forward secrecy, and identity privacy. The Noise Protocol Framework is also used in a number of popular end-to-end encrypted communications systems such as WhatsApp, WireGuard, and I2P. More information can be found at the Noise Protocol Framework website.

The use of the Noise Protocol Framework in the Lightning Network ensures that every message on the network is both authenticated and encrypted, increasing the privacy of the network and its resistance to traffic analysis, deep packet inspection, and eavesdropping. However, as a side effect, this makes protocol development and testing a bit tricky because one can't simply observe the network with a packet capture or network analysis tool such as Wireshark. Instead, developers have to use specialized plug-ins that decrypt the protocol from the perspective of one node, such as the *lightning dissector*, a Wireshark plug-in.

Thoughts About Trust

As long as a person follows the protocol and has their node secured, there is no major risk of losing funds when participating in the Lightning Network. However, there is the cost of paying on-chain fees when opening a channel. Any cost should come with a corresponding benefit. In our case, the reward for Alice for bearing the cost of opening a channel is that Alice can send and, after moving some of the coins to the other end of the channel, receive payments of bitcoin on the Lightning Network at any time, and that she can earn fees in bitcoin by forwarding payments for other people. Alice knows that in theory Bob can close the channel immediately after opening, resulting in on-chain closing fees for Alice. Alice will need to have a small amount of trust in Bob. Alice has been to Bob's Cafe and clearly Bob is interested in selling her coffee, so Alice can trust Bob in this sense. There are mutual benefits to both Alice and Bob. Alice decides that the reward is enough for her to take on the cost of the on-chain fee for creating a channel to Bob. In contrast, Alice will not open a channel to someone unknown who just uninvited sent her an email asking her to open a new channel.

Comparison with Bitcoin

While the Lightning Network is built on top of Bitcoin and inherits many of its features and properties, there are important differences that users of both networks need to be aware of.

Some of these differences are differences in terminology. There are also architectural differences and differences in the user experience. In the next few sections, we will examine the differences and similarities, explain the terminology, and adjust our expectations.

Addresses versus Invoices, Transactions versus Payments

In a typical payment using Bitcoin, a user receives a Bitcoin address (e.g., scanning a QR code on a web page, or receiving it in an instant message or email from a friend). They then use their Bitcoin wallet to create a transaction to send funds to this address.

On the Lightning Network, the recipient of a payment creates an invoice. A Lightning invoice can be seen as analogous to a Bitcoin address. The intended recipient gives the Lightning invoice to the sender as a QR code or character string, just like a Bitcoin address.

The sender uses their Lightning wallet to pay the invoice, copying the invoice text or scanning the invoice QR code. A Lightning payment is analogous to a Bitcoin "transaction."

There are some differences in the user experience, however. A Bitcoin address is *reusable*. Bitcoin addresses never expire, and if the owner of the address still holds the keys, the funds held within are always accessible. A sender can send any amount of bitcoin to a previously used address, and a recipient can post a single static address to receive many payments. While this goes against the best practices for privacy reasons, it is technically possible and in fact quite common.

In Lightning, however, each invoice can only be used once for a specific payment amount. You cannot pay more or less, you cannot use an invoice again, and the invoice has an expiry time built in. In Lightning, a recipient has to generate a new invoice for each payment, specifying the payment amount in advance. There is an exception to this, a mechanism called *keysend*, which we will examine in Keysend Spontaneous Payments.

Selecting Outputs versus Finding a Path

To make a payment on the Bitcoin network, a sender needs to consume one or more unspent transaction outputs (UTXOs). If a user has multiple UTXOs, they (or rather their wallet) will need to select which UTXO(s) to send. For instance, a user making a payment of 1 BTC can use a single output with value 1 BTC, two outputs with value 0.25 BTC and 0.75 BTC, or four outputs with value 0.25 BTC each.

On Lightning, payments do not require inputs to be consumed. Instead, each payment results in an update of the channel balance, redistributing it between the two channel partners. The sender experiences this as "moving" the channel balance from their end of a channel to the other end, to their channel partner. Lightning payments use a series of channels to route from sender to recipient. Each of these channels must have sufficient capacity to route the payment.

Because many possible channels and paths can be used to make a payment, the Lightning user’s choice of channels and paths is somewhat analogous to the Bitcoin user’s choice of UTXO.

With technologies such as atomic multipath payments (AMP) and multipart payments (MPP), which we will review in subsequent chapters, several Lightning paths can be aggregated into a single atomic payment, just like several Bitcoin UTXOs can be aggregated into a single atomic Bitcoin transaction.

Change Outputs on Bitcoin versus No Change on Lightning

To make a payment on the Bitcoin network, the sender needs to consume one or more unspent transaction outputs (UTXOs). UTXOs can only be spent in full; they cannot be divided and partially spent. So if a user wishes to spend 0.8 BTC, but only has a 1 BTC UTXO, they need to spend the entire 1 BTC UTXO by sending 0.8 BTC to the recipient and 0.2 BTC back to themselves as change. The 0.2 BTC change payment creates a new UTXO called a "change output."

On Lightning, the funding transaction spends some Bitcoin UTXO, creating a multisignature UTXO to open the channel. Once the bitcoin is locked within that channel, portions of it can be sent back and forth within the channel, without the need to create any change. This is because the channel partners simply update the channel balance and only create a new UTXO when the channel is eventually closed using the channel closing transaction.

Mining Fees versus Routing Fees

On the Bitcoin network, users pay fees to miners to have their transactions included in a block. These fees are paid to the miner who mines that particular block. The amount of the fee is based on the *size* of the transaction in *bytes* that the transaction is using in a block, as well as how quickly the user wants that transaction mined. Because miners will typically mine the most profitable transactions first, a user who wants their transaction mined immediately will pay a *higher* fee per byte, while a user who is not in a hurry will pay a *lower* fee per byte.

On the Lightning Network, users pay fees to other (intermediary node) users to route payments through their channels. To route a payment, an intermediary node will have to move funds in two or more channels they own, as well as transmit the data for the sender's payment. Typically, the routing user will charge the sender based on the *value* of the payment, having established a minimum *base fee* (a flat fee for each payment) and a *fee rate* (a prorated fee proportional to the value of the payment). Higher value payments will thus cost more to route, and a market for liquidity is formed, where different users charge different fees for routing through their channels.

Varying Fees Depending on Traffic versus Announced Fees

On the Bitcoin network, miners are profit seeking and will typically include as many transactions in a block as possible, while staying within the block capacity called the *block weight*.

If there are more transactions in the queue (called the *mempool*) than can fit in a block, they will begin by mining the transactions that pay the highest fees per unit (bytes) of *transaction weight*. Thus, when there are many transactions in the queue, users have to pay a higher fee to be included in the next block, or they have to wait until there are fewer transactions in the queue. This naturally leads to the emergence of a fee market where users pay based on how urgently they need their transaction included in the next block.

The scarce resource on the Bitcoin network is the space in the blocks. Bitcoin users compete for block space, and the Bitcoin fee market is based on available block space. The scarce resources in the Lightning Network are the *channel liquidity* (capacity of funds available for routing in channels) and *channel connectivity* (how many well-connected nodes channels can reach). Lightning users compete for capacity and connectivity; therefore, the Lightning fee market is driven by capacity and connectivity.

On the Lightning Network, users are paying fees to the users routing their payments. Routing a payment, in economic terms, is nothing more than providing and assigning capacity to the sender. Naturally, routers who charge lower fees for the same capacity will be more attractive to route through. Thus a fee market exists where routers are in competition with each other over the fees they charge to route payments through their channels.

Public Bitcoin Transactions versus Private Lightning Payments

On the Bitcoin network, every transaction is publicly visible on the Bitcoin blockchain. While the addresses involved are pseudonymous and are not typically tied to an identity, they are still seen and validated by every other user on the network. In addition, blockchain surveillance companies collect and analyze this data en masse and sell it to interested parties such as private firms, governments, and intelligence agencies.

LN payments, on the other hand, are almost completely private. Typically, only the sender and the recipient are fully aware of the source, destination, and amount transacted in a particular payment. Furthermore, the receiver may not even know the source of the payment. Because payments are onion routed, the users who route the payment are only aware of the amount of the payment, and they can determine neither the source nor the destination.

In summary, Bitcoin transactions are broadcast publicly and stored forever. Lightning payments are executed between a few selected peers, and information about them is privately stored only until the channel is closed. Creating mass surveillance and analysis tools equivalent to those used on Bitcoin will be much harder on Lightning.

Waiting for Confirmations versus Instant Settlement

On the Bitcoin network, transactions are only settled once they have been included in a block, in which case they are said to be "confirmed" in that block. As more blocks are mined, the transaction acquires more "confirmations" and is considered more secure.

On the Lightning Network, confirmations only matter for opening and closing channels on-chain. Once a funding transaction has reached a suitable number of confirmations (e.g., 3), the channel partners consider the channel open. Because the bitcoin in the channel is secured by a smart contract that manages that channel, payments settle *instantly* once received by the final recipient. In practical terms, instant settlement means that payments take only a few seconds to execute and settle. As with Bitcoin, Lightning payments are not reversible.

Finally, when the channel is closed, a transaction is made on the Bitcoin network; once that transaction is confirmed, the channel is considered closed.

Sending Arbitrary Amounts versus Capacity Restrictions

On the Bitcoin network, a user can send any amount of bitcoin that they own to another user, without capacity restrictions. A single transaction can theoretically send up to 21 million bitcoin as a payment.

On the Lightning Network, a user can only send as much bitcoin as currently exists on their side of a particular channel to a channel partner. For instance, if a user owns one channel with 0.4 BTC on their side, and another channel with 0.2 BTC on their side, then the maximum they can send with one payment is 0.4 BTC. This is true regardless of how much bitcoin the user currently has in their Bitcoin wallet.

Multipart payments (MPP) is a feature which, in the preceding example, allows the user to combine both their 0.4 BTC and 0.2 BTC channels to send a maximum of 0.6 BTC with one payment. MPPs are currently being tested across the Lightning Network and are expected to be widely available and used by the time this book is completed. For more detail on MPP, see Multipart Payments (MPP).

If the payment is routed, every routing node along the routing path must have channels with capacity at least the same as the payment amount being routed. This must hold true for every single channel that the payment is routed through. The capacity of the lowest-capacity channel in a path sets the upper limit for the capacity of the entire path.

Hence, capacity and connectivity are critical and scarce resources in the Lightning Network.

Incentives for Large Value Payment versus Small Value Payments

The fee structure in Bitcoin is independent of the transaction value. A $1 million transaction has the same fee as a $1 transaction on Bitcoin, assuming a similar transaction size, in bytes (more specifically "virtual" bytes after SegWit [Segregated Witness protocol]). In Lightning the fee is a fixed-base fee plus a percentage of the transaction value. Therefore, in Lightning the payment fee increases with payment value. These opposing fee structures create different incentives and lead to different usage in regards to transaction value. A transaction of greater value will be cheaper on Bitcoin; hence, users will prefer Bitcoin for large value transactions. Similarly, on the other end of the scale, users will prefer Lightning for small value transactions.

Using the Blockchain as a Ledger versus as a Court System

On the Bitcoin network, every transaction is eventually recorded in a block on the blockchain. The blockchain thus forms a complete history of every transaction since Bitcoin's creation, and a way to fully audit every bitcoin in existence. Once a transaction is included in the blockchain, it is final. Thus, no disputes can arise and it is unambiguous how much bitcoin is controlled by a particular address at a particular point in the blockchain.

On the Lightning Network, the balance in a channel at a particular time is known only to the two channel partners, and is only made visible to the rest of the network when the channel is closed. When the channel is closed, the final balance of the channel is submitted to the Bitcoin blockchain, and each partner receives their share of the bitcoin in that channel. For instance, if the opening balance was 1 BTC paid by Alice, and Alice made a payment of 0.3 BTC to Bob, then the final balance of the channel is 0.7 BTC for Alice and 0.3 BTC for Bob. If Alice tries to cheat by submitting the opening state of the channel to the Bitcoin blockchain, with 1 BTC for Alice and 0 BTC for Bob, then Bob can retaliate by submitting the true final state of the channel, as well as creating a penalty transaction that gives him all the bitcoin in the channel. For the Lightning Network, the Bitcoin blockchain acts as a court system. Like a robotic judge, Bitcoin records the initial and final balances of each channel and approves penalties if one of the parties tries to cheat.

Offline versus Online, Asynchronous versus Synchronous

When a Bitcoin user sends funds to a destination address, they do not need to know anything about the recipient. The recipient might be offline or online, and no interaction between sender and recipient is needed. The interaction is between sender and the Bitcoin blockchain. Receiving bitcoin on the Bitcoin blockchain is a *passive* and *asynchronous* activity that does not require any interaction by the recipient or for the recipient to be online at any time. Bitcoin addresses can even be generated offline and are never "registered" with the Bitcoin network. Only spending bitcoin requires interaction.

In Lightning, the recipient must be online to complete the payment before it expires. The recipient must run a node or have someone that runs a node on their behalf (a third-party custodian). To be precise, both nodes, the sender's and the recipient's, must be online at the time of payment and must coordinate. Receiving a Lightning payment is an *active* and *synchronous* activity between sender and recipient, without the participation of most of the Lightning Network or the Bitcoin network (except for the intermediary routing nodes, if any).

The synchronous and always-online nature of the Lightning Network is probably the biggest difference in the user experience, and this often confounds users who are accustomed to Bitcoin.

Satoshis versus Millisatoshis

On the Bitcoin network, the smallest amount is a *satoshi*, which cannot be divided any further. Lightning is a bit more flexible, and Lightning nodes work with *millisatoshis* (thousandths of a satoshi). This allows tiny payments to be sent via Lightning. A single millisatoshi payment can be sent across a payment channel, an amount so small it should properly be characterized as a *nanopayment*.

The millisatoshi unit cannot, of course, be settled on the Bitcoin blockchain at that granularity. Upon channel closure, balances are rounded to the nearest satoshi. But over the lifetime of a channel, millions of nanopayments are possible at millisatoshi levels. The Lightning Network breaks through the micropayment barrier.

Commonality of Bitcoin and Lightning

While the Lightning Network differs from Bitcoin in a number of ways, including in architecture and user experience, it is built from Bitcoin and retains many of Bitcoin's core features.

Monetary Unit

Both the Bitcoin network and the Lightning Network use the same monetary units: bitcoin. Lightning payments use the very same bitcoin as Bitcoin transactions. As an implication, because the monetary unit is the same, the monetary limit is the same: less than 21 million bitcoin. Of Bitcoin's 21 million total bitcoin, some are already allocated to 2-of-2 multisignature addresses as part of payment channels on the Lightning Network.

Irreversibility and Finality of Payments

Both Bitcoin transactions and Lightning payments are irreversible and immutable. There is no "undo" operation or "chargeback" for either system. As a sender of either one, you have to act responsibly, but also, as a recipient you are guaranteed finality of your transactions.

Trust and Counterparty Risk

As with Bitcoin, Lightning requires the user only to trust mathematics, encryption, and that the software does not have any critical bugs. Neither Bitcoin nor Lightning requires the user to trust a person, a company, an institution, or a government. Because Lightning sits on top of Bitcoin and relies on Bitcoin as its underlying base layer, it is clear that the security model of Lightning reduces to the security of Bitcoin. This means that Lightning offers broadly the same security as Bitcoin under most circumstances, with only a slight reduction in security under some narrow circumstances.

Permissionless Operation

Both Bitcoin and Lightning can be used by anybody with access to the internet and to the appropriate software, e.g., node and wallet. Neither network requires users to get permission, vetting, or authorization from third parties, companies, institutions, or a government. Governments can outlaw Bitcoin or Lightning within their jurisdiction, but cannot prevent their global use.

Open Source and Open System

Both Bitcoin and Lightning are open source software systems built by a decentralized global community of volunteers, available under open licenses. Both are based on open and interoperable protocols that operate as open systems and open networks. Global, open, and free.

Conclusion

In this chapter we looked at how the Lightning Network actually works and all of the constituent components. We examined each step in constructing, operating, and closing a channel. We looked at how payments are routed, and finally, we compared Lightning with Bitcoin and analyzed their differences and commonalities.

In the next several chapters we will revisit all these topics, but in much more detail.

Lightning Node Software

As we have seen in previous chapters, a Lightning node is a computer system that participates in the Lightning Network. The Lightning Network is not a product or company; it is a set of open standards that define a baseline for interoperability. As such, Lightning node software has been built by a variety of companies and community groups. The vast majority of Lightning software is *open source*, meaning that the source code is open and licensed in such a way as to enable collaboration, sharing, and community participation in the development process. Similarly, the Lightning node implementations we will present in this chapter are all open source and are collaboratively developed.

Unlike Bitcoin, where the standard is defined by a *reference implementation* in software (Bitcoin Core), in Lightning the standard is defined by a series of standards documents called *Basis of Lightning Technology* (*BOLT*), found at the *lightning-rfc* repository.

There is no reference implementation of the Lightning Network, but there are several competing, BOLT-compliant, and interoperable implementations developed by different teams and organizations. The teams that develop software for the Lightning Network also contribute in the development and evolution of the BOLT standards.

Another major difference between Lightning node software and Bitcoin node software is that Lightning nodes do not need to operate in lockstep with consensus rules and can have extended functionality beyond the baseline of the BOLTs. Therefore, different teams may pursue various experimental features that, if successful and broadly deployed, may become part of the BOLTs later.

In this chapter, you will learn how to set up each of the software packages for the most popular Lightning node implementations. We've presented them in alphabetical order to emphasize that we generally do not prefer or endorse one over the other. Each has its strengths and weaknesses, and choosing one will depend on a variety of factors. Since they are developed in different programming languages (e.g., Go, C, etc.), your choice may also depend on your level of familiarity and expertise with a specific language and development toolset.

Lightning Development Environment

If you're a developer, you will want to set up a development environment with all the tools, libraries, and support software for writing and running Lightning software. In this highly technical chapter, we'll walk through that process step-by-step. If the material becomes too dense or you're not actually setting up a development environment, then feel free to skip to the next chapter, which is less technical.

Using the Command Line

The examples in this chapter, and more broadly in most of this book, use a command-line terminal. That means that you type commands into a terminal and receive text responses. Furthermore, the examples are demonstrated on an operating system based on the Linux kernel and GNU software system, specifically the latest long-term stable release of Ubuntu (Ubuntu 20.04 LTS). The majority of the examples can be replicated on other operating systems such as Windows or macOS, with small modifications to the commands. The biggest difference between operating systems is the *package manager* that installs the various software libraries and their prerequisites. In the given examples, we will use apt, which is the package manager for Ubuntu. On macOS, a common package manager used for open source development is Homebrew, which is accessed by the command brew.

In most of the examples here, we will be building the software directly from the source code. While this can be quite challenging, it gives us the most power and control. You may choose to use Docker containers, precompiled packages, or other installation mechanisms instead if you get stuck!

TIP In many of the examples in this chapter we will be using the operating system's command-line interface (also known as a *shell*), accessed via a *terminal* application. The shell will first display a prompt as an indicator that it is ready for your command. Then you type a command and press the Enter key, to which the shell responds with some text and a new prompt for your next command. The prompt may look different on your system, but in the following examples it is denoted by a $ symbol. In the examples, when you see text after a $ symbol, don't type the $ symbol but type the command immediately following it. Then press the Enter key to execute the command. In the examples, the lines following each command are the operating system's responses to that command. When you see the next $ prefix, you'll know it is a new command and you should repeat the process.

To keep things consistent, we use the bash shell in all command-line examples. While other shells will behave in a similar way, and you will be able to run all the examples without it, some of the shell scripts are written specifically for the bash shell and may require some changes or customizations to run in another shell. For consistency, you can install the bash shell on Windows and macOS, and it comes installed by default on most Linux systems.

Downloading the Book Repository

All the code examples are available in the book's online repository. Because the repository will be kept up-to-date as much as possible, you should always look for the latest version in the online repository instead of copying it from the printed book or the ebook.

You can download the repository as a ZIP bundle by visiting GitHub and selecting the green Code button on the right.

Alternatively, you can use the git command to create a version-controlled clone of the repository on your local computer. Git is a distributed version control system that is used by most developers to collaborate on software development and track changes to software repositories. Download and install git by following the instructions from the Git Project.

To make a local copy of the repository on your computer, run the git command as follows:

```
$ git clone https://github.com/lnbook/lnbook.git
```

You now have a complete copy of the book repository in a folder called lnbook. You will want to change to the newly downloaded directory by running:

```
$ cd lnbook
```

All subsequent examples will assume that you are running commands from inside this folder.

Docker Containers

Many developers use a *container*, which is a type of virtual machine, to install a pre-configured operating system and applications with all the necessary dependencies. Much of the Lightning software can also be installed using a container system such as *Docker* found at the Docker home page. Container installations are a lot easier, especially for those who are not used to a command-line environment.

The book's repository contains a collection of Docker containers that can be used to set up a consistent development environment to practice and replicate the examples on any system. Because the container is a complete operating system that runs with a consistent configuration, you can be sure that the examples will work on your computer without the need to worry about dependencies, library versions, or differences in configuration.

Docker containers are often optimized to be small, i.e., occupy the minimum disk space. However, in this book we are using containers to *standardize* the environment and make it consistent for all readers. Furthermore, these containers are not meant to be used to run services in the background. Instead, they are meant to be used to test the examples and learn by interacting with the software. For these reasons, the containers are quite large and come with a lot of development tools and utilities. Commonly, the Alpine distribution is used for Linux containers due to their reduced size. Nonetheless, we provide containers built on Ubuntu because more developers are familiar with Ubuntu, and this familiarity is more important to us than size.

The installation and use of Docker and its commands are detailed in Docker Basic Installation and Use. If you are unfamiliar with Docker, now is a good time to quickly review that section.

You can find the latest container definitions and build configurations in the book's repository under the *code/docker* folder. Each container is in a separate folder, as can be seen in the following:

```
$ tree -F --charset=asciii code/docker
```

```
code/docker
|-- bitcoind/
|   |-- bashrc
|   |-- bitcoind/
|   |   |-- bitcoin.conf
|   |   `-- keys/
|   |       |-- demo_address.txt
|   |       |-- demo_mnemonic.txt
|   |       `-- demo_privkey.txt
|   |-- bitcoind-entrypoint.sh
|   |-- cli
|   |-- Dockerfile
|   `-- mine.sh*
|-- c-lightning/
|   |-- bashrc
|   |-- cli
|   |-- c-lightning-entrypoint.sh
|   |-- devkeys.pem
|   |-- Dockerfile
|   |-- fund-c-lightning.sh
|   |-- lightningd/
|   |   `-- config
|   |-- logtail.sh
|   `-- wait-for-bitcoind.sh
|-- eclair/
|   |-- bashrc
|   |-- cli
|   |-- Dockerfile
|   |-- eclair/
|   |   `-- eclair.conf
|   |-- eclair-entrypoint.sh
|   |-- logtail.sh
|   `-- wait-for-bitcoind.sh
|-- lnd/
|   |-- bashrc
|   |-- cli
|   |-- Dockerfile
|   |-- fund-lnd.sh
|   |-- lnd/
|   |   `-- lnd.conf
|   |-- lnd-entrypoint.sh
|   |-- logtail.sh
|   `-- wait-for-bitcoind.sh
|-- check-versions.sh
|-- docker-compose.yml
|-- Makefile
`-- run-payment-demo.sh*
```

As we will see in the next few sections, you can build these containers locally, or you can pull them from the book's repository on *Docker Hub*. The following sections will assume that you have installed Docker and are familiar with the basic use of the docker command.

Bitcoin Core and Regtest

Most of the Lightning node implementations need access to a full Bitcoin node to work.

Installing a full Bitcoin node and syncing the Bitcoin blockchain is outside the scope of this book and is a relatively complex endeavor in itself. If you want to try it, refer to *Mastering Bitcoin*, "Chapter 3: Bitcoin Core: The Reference Implementation," which discusses the installation and operation of a Bitcoin node.

A Bitcoin node can be operated in `regtest` mode, where the node creates a local simulated Bitcoin blockchain for testing purposes. In the following examples, we will be using the regtest mode to allow us to demonstrate Lightning without having to synchronize a Bitcoin node or risk any funds.

The container for Bitcoin Core is bitcoind. It is configured to run Bitcoin Core in regtest mode and to mine 6 new blocks every 10 seconds. Its remote procedure call (RPC) port is exposed on port 18443 and is accessible for RPC calls with the username regtest and the password regtest. You can also access it with an interactive shell and run bitcoin-cli commands locally.

Building the Bitcoin Core Container

Let’s prepare the bitcoind container. The easiest way is to pull the latest container from *Docker Hub*:

```
$ docker pull lnbook/bitcoind
Using default tag: latest
latest: Pulling from lnbook/bitcoind
35807b77a593: Pull complete
e1b85b9c5571: Pull complete
[...]
288f1cc78a00: Pull complete
Digest: sha256:861e7e32c9ad650aa367af40fc5acff894e89e47aff4bd400691ae18f1b550e2
Status: Downloaded newer image for lnbook/bitcoind:latest
docker.io/lnbook/bitcoind:latest
```

Alternatively, you can build the container yourself from the local container definition that is in *code/docker/bitcoind/Dockerfile*.

NOTE You don’t need to build the container if you used the pull command previously to pull it from Docker Hub.

Building the container locally will use a bit less of your network bandwidth, but will take more of your CPU time to build. We use the docker build command to build it:

```
$ cd code/docker
$ docker run -it --name bitcoind lnbook/bitcoind
Starting bitcoind...
Bitcoin Core starting
Waiting for bitcoind to start
bitcoind started
================================================
Imported demo private key
Bitcoin address:  2NBKgwSWY5qEmfN2Br4WtMDGuamjpuUc5q1
Private key:  cSaejkcWwU25jMweWEewRSsrVQq2FGTij1xjXv4x1XvxVRF1ZCr3
================================================
================================================
Balance: 0.00000000
================================================
Mining 101 blocks to unlock some bitcoin
[
  "34c744207fd4dd32b70bac467902bd8d030fba765c9f240a2e98f15f05338964",
  "64d82721c641c378d79b4ff2e17572c109750bea1d4eddbae0b54f51e4cdf23e",

 [...]

  "7a8c53dc9a3408c9ecf9605b253e5f8086d67bbc03ea05819b2c9584196c9294",
  "39e61e50e34a9bd1d6eab51940c39dc1ab56c30b21fc28e1a10c14a39b67a1c3",
  "4ca7fe9a55b0b767d2b7f5cf4d51a2346f035fe8c486719c60a46dcbe33de51a"
]
Mining 6 blocks every 10 seconds
Balance: 50.00000000
[
  "5ce76cc475e40515b67e3c0237d1eef597047a914ba3f59bbd62fc3691849055",
  "1ecb27a05ecfa9dfa82a7b26631e0819b2768fe5e6e56c7a2e1078b078e21e9f",
  "717ceb8b6c329d57947c950dc5668fae65bddb7fa03203984da9d2069e20525b",
  "185fc7cf3557a6ebfc4a8cdd1f94a8fa08ed0c057040cdd68bfb7aee2d5be624",
  "59001ae237a3834ebe4f6e6047dcec8fd67df0352ddc70b6b02190f982a60384",
  "754c860fe1b9e0e7292e1de96a65eaa78047feb4c72dbbde2a1d224faa1499dd"
]
```

As you can see, bitcoind starts up and mines 101 simulated blocks to get the chain started. This is because under the Bitcoin consensus rules, newly mined bitcoin is not spendable until 100 blocks have elapsed. By mining 101 blocks, we make the first block’s coinbase spendable. After that initial mining activity, 6 new blocks are mined every 10 seconds to keep the chain moving forward.

For now, there are no transactions. But we have some test bitcoin that has been mined in the wallet and is available to spend. When we connect some Lightning nodes to this chain, we will send some bitcoin to their wallets so that we can open some Lightning channels between the Lightning nodes.

Interacting with the bitcoin core container

In the meantime, we can also interact with the bitcoind container by sending it shell commands. The container is sending a logfile to the terminal, displaying the mining process of the bitcoind process. To interact with the shell we can issue commands in another terminal, using the docker exec command. Since we previously named the running container with the name argument, we can refer to it by that name when we run the docker exec command. First, let’s run an interactive bash shell:

```
$ docker exec -it bitcoind /bin/bash
root@e027fd56e31a:/bitcoind# ps x
  PID TTY      STAT   TIME COMMAND
    1 pts/0    Ss+    0:00 /bin/bash /usr/local/bin/mine.sh
    7 ?        Ssl    0:03 bitcoind -datadir=/bitcoind -daemon
   97 pts/1    Ss     0:00 /bin/bash
  124 pts/0    S+     0:00 sleep 10
  125 pts/1    R+     0:00 ps x
root@e027fd56e31a:/bitcoind#
```

Running the interactive shell puts us "inside" the container. It logs in as user root, as we can see from the prefix root@ in the new shell prompt root@e027fd56e31a:/bitcoind#. If we issue the ps x command to see what processes are running, we see both bitcoind and the script mine.sh are running in the background. To exit this shell, press Ctrl-D or type **exit**, and you will be returned to your operating system prompt.

Instead of running an interactive shell, we can also issue a single command that is executed inside the container. For convenience, the bitcoin-cli command has an alias "cli" that passes the correct configuration. So let’s run it to ask Bitcoin Code about the blockchain. We run cli getblockchaininfo:

```
$ docker exec bitcoind cli getblockchaininfo
{
  "chain": "regtest",
  "blocks": 131,
  "headers": 131,
  "bestblockhash": "2cf57aac35365f52fa5c2e626491df634113b2f1e5197c478d57378e5a146110",

[...]

  "warnings": ""
}
```

The cli command in the bitcoind container allows us to issue RPC commands to the Bitcoin Core node and get JavaScript Object Notation (JSON) encoded results.

Additionally, all our Docker containers have a command-line JSON encoder/decoder named jq preinstalled. jq helps us to process JSON-formatted data via the command line or from inside scripts. You can send the JSON output of any command to jq using the | character. This character as well as this operation is called a "pipe." Let’s apply a pipe and jq to the previous command as follows:

```
$ docker exec bitcoind bash -c "cli getblockchaininfo | jq .blocks"
197
```

jq .blocks instructs the jq JSON decoder to extract the field blocks from the getblockchaininfo result. In our case, it extracts and prints the value of 197 which we could use in a subsequent command.

As you will see in the following sections, we can run several containers at the same time and then interact with them individually. We can issue commands to extract information such as the Lightning node public key or to take actions such as opening a Lightning channel to another node. The docker run and docker exec commands, together with jq for JSON decoding, are all we need to build a working Lightning Network that mixes many different node implementations. This enables us to try out diverse experiments on our own computer.

The c-lightning Lightning Node Project

`c-lightning` is a lightweight, highly customizable, and standard-compliant implementation of the LN protocol, developed by Blockstream as part of the Elements Project. The project is open source and developed collaboratively on GitHub.

In the following sections, we will build a Docker container that runs a `c-lightning` node connecting to the bitcoind container we built previously. We will also show you how to configure and built the `c-lightning` software directly from the source code.

Building c-lightning as a Docker Container

The `c-lightning` software distribution has a Docker container, but it is designed for running `c-lightning` in production systems and alongside a bitcoind node. We will be using a somewhat simpler container configured to run `c-lightning` for demonstration purposes.

Let’s pull the `c-lightning` container from the book’s Docker Hub repository:

```
$ docker pull lnbook/c-lightning
Using default tag: latest
latest: Pulling from lnbook/c-lightning

[...]

Digest: sha256:bdefcefe8a9712e7b3a236dcc5ab12d999c46fd280e209712e7cb649b8bf0688
Status: Downloaded image for lnbook/c-lightning:latest
docker.io/lnbook/c-lightning:latest
```

Alternatively, we can build the `c-lightning` Docker container from the book’s files which you previously downloaded into a directory named lnbook. As before, we will use the docker build command in the code/docker subdirectory. We will tag the container image with the tag lnbook/c-lightning, like this:

```
$ cd code/docker
$ docker build -t lnbook/c-lightning c-lightning
Sending build context to Docker daemon  91.14kB
Step 1/34 : ARG OS=ubuntu
Step 2/34 : ARG OS_VER=focal
Step 3/34 : FROM ${OS}:${OS_VER} as os-base
 ---> fb52e22af1b0

 [...]

Step 34/34 : CMD ["/usr/local/bin/logtail.sh"]
 ---> Running in 8d3d6c8799c5
Removing intermediate container 8d3d6c8799c5
 ---> 30b6fd5d7503
Successfully built 30b6fd5d7503
Successfully tagged lnbook/c-lightning:latest
```

Our container is now built and ready to run. However, before we run the `c-lightning` container, we need to start the bitcoind container in another terminal because `c-lightning` depends on bitcoind. We will also need to set up a Docker network that allows the containers to connect to each other as if residing on the same local area network.

TIP Docker containers can "talk" to each other over a virtual local area network managed by the Docker system. Each container can have a custom name, and other containers can use that name to resolve its IP address and easily connect to it.

Setting Up a Docker Network

Once a Docker network is set up, Docker will activate the network on our local computer every time Docker starts, e.g., after rebooting. So we only need to set up a network once by using the docker network create command. The network name itself is not important, but it has to be unique on our computer. By default, Docker has three networks named host, bridge, and none. We will name our new network lnbook and create it like this:

```
$ docker network create lnbook
ad75c0e4f87e5917823187febedfc0d7978235ae3e88eca63abe7e0b5ee81bfb
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
7f1fb63877ea        bridge              bridge              local
4e575cba0036        host                host                local
ad75c0e4f87e        lnbook              bridge              local
ee8824567c95        none                null                local
```

As you can see, running docker network ls gives us a listing of the Docker networks. Our lnbook network has been created. We can ignore the network ID, because it is automatically managed.

Running the bitcoind and c-lightning Containers

The next step is to start the bitcoind and `c-lightning` containers and connect them to the lnbook network. To run a container in a specific network, we must pass the network argument to docker run. To make it easy for containers to find each other, we will also give each one a name with the name argument. We start bitcoind like this:

```
$ docker run -it --network lnbook --name bitcoind lnbook/bitcoind
```

You should see bitcoind start up and start mining blocks every 10 seconds. Leave it running and open a new terminal window to start `c-lightning`. We use a similar docker run command with the network and name arguments to start `c-lightning` as follows:

```
$ docker run -it --network lnbook --name c-lightning lnbook/c-lightning
Waiting for bitcoind to start...
Waiting for bitcoind to mine blocks...
Starting c-lightning...
2021-09-12T13:14:50.434Z UNUSUAL lightningd: Creating configuration directory
/lightningd/regtest
Startup complete
Funding c-lightning wallet
8a37a183274c52d5a962852ba9f970229ea6246a096ff1e4602b57f7d4202b31
lightningd: Opened log file /lightningd/lightningd.log
lightningd: Creating configuration directory /lightningd/regtest
lightningd: Opened log file /lightningd/lightningd.log
```

The `c-lightning` container starts up and connects to the bitcoind container over the Docker network. First, our `c-lightning` node will wait for bitcoind to start, and then it will wait until bitcoind has mined some bitcoin into its wallet. Finally, as part of the container startup, a script will send an RPC command to the bitcoind node, which creates a transaction that funds the `c-lightning` wallet with 10 test BTC. Now our `c-lightning` node is not only running, but it even has some test bitcoin to play with!

As we demonstrated with the bitcoind container, we can issue commands to our `c-lightning` container in another terminal to extract information, open channels, etc. The command that allows us to issue command-line instructions to the `c-lightning` node is called lightning-cli. This lightning-cli command is also aliased as cli inside this container. To get the `c-lightning` node's information, use the following docker exec command in another terminal window:

```
$ docker exec c-lightning cli getinfo
{
   "id": "026ec53cc8940df5fed5fa18f8897719428a15d860ff4cd171fca9530879c7499e",
   "alias": "IRATEARTIST",
   "color": "026ec5",
   "num_peers": 0,
   "num_pending_channels": 0,

[...]

   "version": "0.10.1",
   "blockheight": 221,
   "network": "regtest",
   "msatoshi_fees_collected": 0,
   "fees_collected_msat": "0msat",
   "lightning-dir": "/lightningd/regtest"
}
```

We now have our first Lightning node running on a virtual network and communicating with a test Bitcoin blockchain. Later in this chapter we will start more nodes and connect them to each other to make some Lightning payments.

In the next section we will also look at how to download, configure, and compile `c-lightning` directly from the source code. This is an optional and advanced step that will teach you how to use the build tools and allow you to make modifications to `c-lightning` source code. With this knowledge you can write some code, fix some bugs, or create a plug-in for `c-lightning`.

NOTE If you are not planning on diving into the source code or programming of a Lightning node, you can skip the next section entirely. The Docker container we just built is sufficient for most of the examples in the book.

Installing c-lightning from Source Code

The `c-lightning` developers have provided detailed instructions for building `c-lightning` from source code. We will be following the instructions from GitHub.

Installing Prerequisite Libraries and Packages

These installation instructions assume you are building `c-lightning` on a Linux or similar system with GNU build tools. If that is not the case, look for the instructions for your operating system in the Elements Project repository.

The common first step is the installation of prerequisite libraries. We use the apt package manager to install these:

```
$ sudo apt-get update

Get:1 http://security.ubuntu.com/ubuntu bionic-security InRelease [88.7 kB]
Hit:2 http://eu-north-1b.clouds.archive.ubuntu.com/ubuntu bionic InRelease
Get:3 http://eu-north-1b.clouds.archive.ubuntu.com/ubuntu bionic-updates InRelease
[88.7 kB]

[...]

Fetched 18.3 MB in 8s (2,180 kB/s)
Reading package lists... Done

$ sudo apt-get install -y \
  autoconf automake build-essential git libtool libgmp-dev \
  libsqlite3-dev python python3 python3-mako net-tools zlib1g-dev \
  libsodium-dev gettext

Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  autotools-dev binutils binutils-common binutils-x86-64-linux-gnu cpp cpp-7 dpkg-dev
fakeroot g++ g++-7 gcc gcc-7 gcc-7-base libalgorithm-diff-perl

 [...]

Setting up libsigsegv2:amd64 (2.12-2) ...
Setting up libltdl-dev:amd64 (2.4.6-14) ...
Setting up python2 (2.7.17-2ubuntu4) ...
Setting up libsodium-dev:amd64 (1.0.18-1) ...

[...]
$
```

After a few minutes and a lot of on-screen activity, you will have installed all the necessary packages and libraries. Many of these libraries are also used by other Lightning packages and are needed for software development in general.

Copying the c-lightning Source Code

Next, we will copy the latest version of `c-lightning` from the source code repository. To do this, we will use the git clone command, which clones a version-controlled copy onto your local machine, thereby allowing you to keep it synchronized with subsequent changes without having to download the whole repository again:

```
$ git clone --recurse https://github.com/ElementsProject/lightning.git
Cloning into 'lightning'...
remote: Enumerating objects: 24, done.
remote: Counting objects: 100% (24/24), done.
remote: Compressing objects: 100% (22/22), done.
remote: Total 53192 (delta 5), reused 5 (delta 2), pack-reused 53168
Receiving objects: 100% (53192/53192), 29.59 MiB | 19.30 MiB/s, done.
Resolving deltas: 100% (39834/39834), done.

$ cd lightning
```

We now have a copy of `c-lightning` cloned into the *lightning* subfolder, and we have used the cd (change directory) command to enter that subfolder.

Compiling the c-lightning Source Code

Next, we use a set of *build scripts* that are commonly available in many open source projects. These build scripts use the configure and make commands which allow us to:

- Select the build options and check necessary dependencies (configure)
- Build and install the executables and libraries (make)

Running configure with the help option will show us all the available options:

```
$ ./configure --help
Usage: ./configure [--reconfigure] [setting=value] [options]

Options include:
  --prefix= (default /usr/local)
    Prefix for make install
  --enable/disable-developer (default disable)
    Developer mode, good for testing
  --enable/disable-experimental-features (default disable)
    Enable experimental features
  --enable/disable-compat (default enable)
    Compatibility mode, good to disable to see if your software breaks
  --enable/disable-valgrind (default (autodetect))
    Run tests with Valgrind
  --enable/disable-static (default disable)
    Static link sqlite3, gmp and zlib libraries
  --enable/disable-address-sanitizer (default disable)
    Compile with address-sanitizer
```

We don’t need to change any of the defaults for this example. Hence we run configure again without any options to use the defaults:

```
$ ./configure

Compiling ccan/tools/configurator/configurator...done
checking for python3-mako... found
Making autoconf users comfortable... yes
checking for off_t is 32 bits... no
checking for __alignof__ support... yes

[...]

Setting COMPAT... 1
PYTEST not found
Setting STATIC... 0
Setting ASAN... 0
Setting TEST_NETWORK... regtest
$
```

Next, we use the make command to build the libraries, components, and executables of the `c-lightning` project. This part will take several minutes to complete and will use your computer's CPU and disk heavily. Expect some noise from the fans! Run make:

```
$ make

cc -DBINTOPKGLIBEXECDIR="\"../libexec/c-lightning\"" -Wall -Wundef -Wmis...

[...]

cc   -Og  ccan-asort.o ccan-autodata.o ccan-bitmap.o ccan-bitops.o ccan-...
```

If all goes well, you will not see any ERROR message stopping the execution of the preceding command. The `c-lightning` software package has been compiled from source, and we are now ready to install the executable components we created in the previous step:

```
$ sudo make install

mkdir -p /usr/local/bin
mkdir -p /usr/local/libexec/c-lightning
mkdir -p /usr/local/libexec/c-lightning/plugins
mkdir -p /usr/local/share/man/man1
mkdir -p /usr/local/share/man/man5
mkdir -p /usr/local/share/man/man7
mkdir -p /usr/local/share/man/man8
mkdir -p /usr/local/share/doc/c-lightning
install cli/lightning-cli lightningd/lightningd /usr/local/bin
[...]
```

To verify that the lightningd and lightning-cli commands have been installed correctly, we will ask each executable for its version information:

```
$ lightningd --version
v0.10.1-34-gfe86c11
$ lightning-cli --version
v0.10.1-34-gfe86c11
```

The version consists of the latest release version (v0.10.1), followed by the number of changes since the release (34), and finally a hash identifying exactly which revision (fe86c11). You may see a different version from that shown previously as the software continues to evolve long after this book is published. However, no matter what version you see, the fact that the commands execute and respond with version information means that you have succeeded in building the `c-lightning` software.

The Lightning Network Daemon (LND) Node Project

The Lightning Network Daemon (LND) is a complete implementation of an LN node by Lightning Labs. The LND project provides a number of executable applications, including lnd (the daemon itself) and lncli (the command-line utility). LND has several pluggable backend chain services, including btcd (a full node), bitcoind (Bitcoin Core), and Neutrino (a new, experimental light client). LND is written in the Go programming language. The project is open source and developed collaboratively on GitHub.

In the next few sections we will build a Docker container to run LND, build LND from source code, and learn how to configure and run LND.

The LND Docker Container

We can pull the LND example Docker container from the book's Docker Hub repository:

```
$ docker pull lnbook/lnd
Using default tag: latest
latest: Pulling from lnbook/lnd
35807b77a593: Already exists
e1b85b9c5571: Already exists
52f9c252546e: Pull complete

[...]

Digest: sha256:e490a0de5d41b781c0a7f9f548c99e67f9d728f72e50cd4632722b3ed3d85952
Status: Downloaded newer image for lnbook/lnd:latest
docker.io/lnbook/lnd:latest
```

Alternatively, we can build the LND container locally. The container is located in *code/docker/lnd*. We change the working directory to *code/docker* and perform the docker build command:

```
$ cd code/docker
$ docker build -t lnbook/lnd lnd
Sending build context to Docker daemon  9.728kB
Step 1/29 : FROM golang:1.13 as lnd-base
 ---> e9bdcb0f0af9
Step 2/29 : ENV GOPATH /go

[...]

Step 29/29 : CMD ["/usr/local/bin/logtail.sh"]
 ---> Using cache
 ---> 397ce833ce14
Successfully built 397ce833ce14
Successfully tagged lnbook/lnd:latest
```

Our container is now ready to run. As with the `c-lightning` container we built previously, the LND container also depends on a running instance of Bitcoin Core. As before, we need to start the bitcoind container in another terminal and connect LND to it via a Docker network. We have already set up a Docker network called lnbook and will be using that again here.

TIP Normally, each node operator runs their own Lightning node and their own Bitcoin node on their own server. For us, a single bitcoind container can serve many Lightning nodes. On our simulated network we can run several Lightning nodes, all connecting to a single Bitcoin node in regtest mode.

Running the bitcoind and LND Containers

As before, we start the bitcoind container in one terminal and LND in another. If you already have the bitcoind container running, you do not need to restart it. Just leave it running and skip the next step. To start bitcoind in the lnbook network, we use docker run like this:

```
$ docker run -it --network lnbook --name bitcoind lnbook/bitcoind
```

Next, we start the LND container we just built. As done before, we need to attach it to the lnbook network and give it a name:

```
$ docker run -it --network lnbook --name lnd lnbook/lnd
Waiting for bitcoind to start...
Waiting for bitcoind to mine blocks...
Starting lnd...
Startup complete
Funding lnd wallet
{"result":"dbd1c8e2b224e0a511c11efb985dabd84d72d935957ac30935ec4211d28beacb","error":n
ull,"id":"lnd-run-container"}
[INF] LTND: Version: 0.13.1-beta commit=v0.13.1-beta, build=production,
logging=default, debuglevel=info
[INF] LTND: Active chain: Bitcoin (network=regtest)
[INF] RPCS: Generating TLS certificates...
```

The LND container starts up and connects to the bitcoind container over the Docker network. First, our LND node will wait for bitcoind to start, and then it will wait until bitcoind has mined some bitcoin into its wallet. Finally, as part of the container startup, a script will send an RPC command to the bitcoind node, thereby creating a transaction that funds the LND wallet with 10 test BTC.

As we demonstrated previously, we can issue commands to our container in another terminal to extract information, open channels, etc. The command that allows us to issue command-line instructions to the lnd daemon is called lncli. Once again, in this container we have provided the alias cli that runs lncli with all the appropriate parameters. Let’s get the node information using the docker exec command in another terminal window:

```
$ docker exec lnd cli getinfo
{
    "version": "0.13.1-beta commit=v0.13.1-beta",
    "commit_hash": "596fd90ef310cd7abbf2251edaae9ba4d5f8a689",
    "identity_pubkey":
"02d4545dccbeda29a10f44e891858940f4f3374b75c0f85dcb7775bb922fdeaa14",

[...]

}
```

We now have another Lightning node running on the lnbook network and communicating with bitcoind. If you are still running the `c-lightning` container, then there are now two nodes running. They’re not yet connected to each other, but we will be connecting them to each other soon.

If desired, you can run any combination of LND and `c-lightning` nodes on the same Lightning Network. For example, to run a second LND node you would issue the docker run command with a different container name, like so:

```
$ docker run -it --network lnbook --name lnd2 lnbook/lnd
```

In the preceding command, we start another LND container, naming it lnd2. The names are entirely up to you, as long as they are unique. If you don’t provide a name, Docker will construct a unique name by randomly combining two English words such as "naughty_einstein." This was the actual name Docker chose for us when we wrote this paragraph. How funny!

In the next section we will look at how to download and compile LND directly from the source code. This is an optional and advanced step that will teach you how to use the Go language build tools and allow you to make modifications to LND source code. With this knowledge you can write some code or fix some bugs.

NOTE If you are not planning on diving into the source code or programming of a Lightning node, you can skip the next section entirely. The Docker container we just built is sufficient for most of the examples in the book.

Installing LND from Source Code

In this section we will build LND from scratch. LND is written in the Go programming language. If you want to find out more about Go, search for golang instead of go to avoid irrelevant results. Because it is written in Go and not C or C++, it uses a different "build" framework than the GNU autotools/make framework we saw used in `c-lightning` previously. Don’t fret though, it is quite easy to install and use the golang tools, and we will show each step here. Go is a fantastic language for collaborative software development because it produces very consistent, precise, and easy-to-read code regardless of the number of authors. Go is focused and "minimalist" in a way that encourages consistency across versions of the language. As a compiled language, it is also quite efficient. Let’s dive in.

We will follow the installation instructions found in the LND project documentation.

First, we will install the golang package and associated libraries. We strictly require Go version 1.13 or later. The official Go language packages are distributed as binaries from the Go Project. For convenience they are also packaged as Debian packages available through the apt command. You can follow the instructions from the Go Project or use the following apt commands on a Debian/Ubuntu Linux system as described on GitHub’s wiki page on the Go language:

```
$ sudo apt install golang-go
```

Check that you have the correct version installed and ready to use by running:

```
$ go version
go version go1.13.4 linux/amd64
```

We have 1.13.4, so we’re ready to...Go! Next we need to tell any programs where to find the Go code. This is accomplished by setting the environment variable GOPATH. Usually the Go code is located in a directory named *gocode* directly in the user’s home directory. With the following two commands we consistently set the GOPATH and make sure your shell adds it to your executable PATH. Note that the user’s home directory is referred to as ~ in the shell.

```
$ export GOPATH=~/gocode
$ export PATH=$PATH:$GOPATH/bin
```

To avoid having to set these environment variables every time you open a shell, you can add those two lines to the end of your bash shell configuration file *.bashrc* in your home directory, using the editor of your choice.

Copying the LND Source Code

As with many open source projects nowadays, the source code for LND is on GitHub (*www.github.com*). The go get command can fetch it directly using the Git protocol:

```
$ go get -d github.com/lightningnetwork/lnd
```

Once go get finishes, you will have a subdirectory under GOPATH that contains the LND source code.

Compiling the LND Source Code

LND uses the make build system. To build the project, we change directory to LND's source code and then use make like this:

```
$ cd $GOPATH/src/github.com/lightningnetwork/lnd
$ make && make install
```

After several minutes you will have two new commands, lnd and lncli, installed. Try them out and check their version to ensure they are installed:

```
$ lnd --version
lnd version 0.10.99-beta commit=clock/v1.0.0-106-
gc1ef5bb908606343d2636c8cd345169e064bdc91
$ lncli --version
lncli version 0.10.99-beta commit=clock/v1.0.0-106-
gc1ef5bb908606343d2636c8cd345169e064bdc91
```

You will likely see a different version from that shown previously, as the software continues to evolve long after this book is published. However, no matter what version you see, the fact that the commands execute and show you version information means that you have succeeded in building the LND software.

The Eclair Lightning Node Project

Eclair (French for lightning) is a Scala implementation of the Lightning Network made by ACINQ. Eclair is also one of the most popular and pioneering mobile Lightning wallets, which we used to demonstrate a Lightning payment in Getting Started. In this section we examine the Eclair server project, which runs a Lightning node. Eclair is an open source project and can be found on GitHub.

In the next few sections we will build a Docker container to run Eclair, as we did previously with `c-lightning` and LND. We will also build Eclair directly from the source code.

The Eclair Docker Container

Let's pull the book's Eclair container from the Docker Hub repository:

```
$ docker pull lnbook/eclair
Using default tag: latest
latest: Pulling from lnbook/eclair
35807b77a593: Already exists
e1b85b9c5571: Already exists

[...]

c7d5d5c616c2: Pull complete
Digest: sha256:17a3d52bce11a62381727e919771a2d5a51da9f91ce2689c7ecfb03a6f028315
Status: Downloaded newer image for lnbook/eclair:latest
docker.io/lnbook/eclair:latest
```

Alternatively, we can build the container locally, instead. By now, you are almost an expert in the basic operations of Docker! In this section we will repeat many of the previously seen commands to build the Eclair container. The container is located in *code/docker/eclair*. We start in a terminal by switching the working directory to *code/docker* and issuing the docker build command:

```
$ cd code/docker
$ docker build -t lnbook/eclair eclair
Sending build context to Docker daemon  11.26kB
Step 1/27 : ARG OS=ubuntu
Step 2/27 : ARG OS_VER=focal
Step 3/27 : FROM ${OS}:${OS_VER} as os-base
 ---> fb52e22af1b0

[...]

Step 27/27 : CMD ["/usr/local/bin/logtail.sh"]
 ---> Running in fe639120b726
Removing intermediate container fe639120b726
 ---> e6c8fe92a87c
Successfully built e6c8fe92a87c
Successfully tagged lnbook/eclair:latest
```

Our image is now ready to run. The Eclair container also depends on a running instance of Bitcoin Core. As before, we need to start the bitcoind container in another terminal and connect Eclair to it via a Docker network. We have already set up a Docker network called lnbook, and will be reusing it here.

One notable difference between Eclair and LND or `c-lightning` is that Eclair doesn't contain a separate bitcoin wallet but instead relies directly on the bitcoin wallet in Bitcoin Core. Recall that using LND we funded its bitcoin wallet by executing a transaction to transfer bitcoin from Bitcoin Core's wallet to LND's bitcoin wallet. This step is not necessary using Eclair. When running Eclair, the Bitcoin Core wallet is used directly as the source of funds to open channels. As a result, unlike the LND or `c-lightning` containers, the Eclair container does not contain a script to transfer bitcoin into its wallet on startup.

Running the bitcoind and Eclair Containers

As before, we start the bitcoind container in one terminal and the Eclair container in another. If you already have the bitcoind container running, you do not need to restart it. Just leave it running and skip the next step. To start bitcoind in the lnbook network, we use docker run like this:

```
$ docker run -it --network lnbook --name bitcoind lnbook/bitcoind
```

Next, we start the Eclair container we just built. We will need to attach it to the lnbook network and give it a name, just as we did with the other containers:

```
$ docker run -it --network lnbook --name eclair lnbook/eclair
Waiting for bitcoind to start...
Waiting for bitcoind to mine blocks...
Starting eclair...
Eclair node started
INFO  o.b.Secp256k1Context - secp256k1 library successfully loaded
INFO  fr.acinq.eclair.Plugin - loading 0 plugins
INFO  a.e.slf4j.Slf4jLogger - Slf4jLogger started
INFO  fr.acinq.eclair.Setup - hello!
INFO  fr.acinq.eclair.Setup - version=0.4.2 commit=52444b0

[...]
```

The Eclair container starts up and connects to the bitcoind container over the Docker network. First, our Eclair node will wait for bitcoind to start, and then it will wait until bitcoind has mined some bitcoin into its wallet.

As we demonstrated previously, we can issue commands to our container in another terminal to extract information, open channels, etc. The command that allows us to issue command-line instructions to the eclair daemon is called eclair-cli. As before, in this container we have provided a useful alias to eclair-cli, called simply cli, which offers the necessary arguments and parameters. Using the docker exec command in another terminal window, we get the node info from Eclair:

```
$ docker exec eclair cli getinfo
{
  "version": "0.4.2-52444b0",
  "nodeId": "02fa6d5042eb8098e4d9c9d99feb7ebc9e257401ca7de829b4ce757311e0301de7",
  "alias": "eclair",
  "color": "#49daaa",
  "features": {

[...]

  },
  "chainHash": "06226e46111a0b59caaf126043eb5bbf28c34f3a5e332a1fc7b2b73cf188910f",
  "network": "regtest",
  "blockHeight": 779,
  "publicAddresses": [],
  "instanceId": "01eb7a68-5db0-461b-bdd0-29010df40d73"
}
```

We now have another Lightning node running on the lnbook network and communicating with bitcoind. You can run any number and any combination of Lightning nodes on the same Lightning network. Any number of Eclair, LND, and `c-lightning` nodes can coexist. For example, to run a second Eclair node you would issue the docker run command with a different container name, as follows:

```
$ docker run -it --network lnbook --name eclair2 lnbook/eclair
```

In the preceding command we start another Eclair container named eclair2.

In the next section we will also look at how to download and compile Eclair directly from the source code. This is an optional and advanced step that will teach you how to use the Scala and Java language build tools and allow you to make modifications to Eclair's source code. With this knowledge, you can write some code or fix some bugs.

NOTE If you are not planning on diving into the source code or programming of a Lightning node, you can skip the next section entirely. The Docker container we just built is sufficient for most of the examples in the book.

Installing Eclair from Source Code

In this section we will build Eclair from scratch. Eclair is written in the Scala programming language, which is compiled using the Java compiler. To run Eclair, we first need to install Java and its build tools. We will be following the instructions found in the *BUILD.md* document of the Eclair project.

The required Java compiler is part of OpenJDK 11. We will also need a build framework called Maven, version 3.6.0 or above.

On a Debian/Ubuntu Linux system, we can use the apt command to install both OpenJDK 11 and Maven, as shown in the following:

```
$ sudo apt install openjdk-11-jdk maven
```

Verify that you have the correct version installed by running:

```
$ javac -version
javac 11.0.7
$ mvn -v
Apache Maven 3.6.1
Maven home: /usr/share/maven
Java version: 11.0.7, vendor: Ubuntu, runtime: /usr/lib/jvm/java-11-openjdk-amd64
```

We have OpenJDK 11.0.7 and Maven 3.6.1, so we’re ready.

Copying the Eclair Source Code

The source code for Eclair is on GitHub. The git clone command can create a local copy for us. Let’s change to our home directory and run it there:

```
$ cd ~
$ git clone https://github.com/ACINQ/eclair.git
```

Once git clone finishes, you will have a subdirectory eclair containing the source code for the Eclair server.

Compiling the Eclair Source Code

Eclair uses the Maven build system. To build the project, we change the working directory to Eclair’s source code and then use mvn package like this:

```
$ cd eclair
$ mvn package
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Build Order:
[INFO]
[INFO] --------------------< fr.acinq.eclair:eclair_2.13 >---------------------
[INFO] Building eclair_2.13 0.4.3-SNAPSHOT                                [1/4]
[INFO] --------------------------------[ pom ]---------------------------------

[...]


[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  01:06 min
[INFO] Finished at: 2020-12-12T09:43:21-04:00
[INFO] ------------------------------------------------------------------------
```

After several minutes, the build of the Eclair package should complete. However, the "package" action will also run tests, and some of these connect to the internet and could fail. If you want to skip tests, add -DskipTests to the command.

Now, unzip and run the build package by following the instructions for installing Eclair from GitHub.

Congratulations! You have built Eclair from source and you are ready to code, test, fix bugs, and contribute to this project!

Building a Complete Network of Diverse Lightning Nodes

Our final example, presented in this section, will bring together all the various containers we've built to form a Lightning Network made of diverse (LND, `c-lightning`, Eclair) node implementations. We'll compose the network by connecting the nodes together and opening channels from one node to another. As the final step, we'll route a payment across these channels!

In this example, we will build a demonstration Lightning Network made of four Lightning nodes named Alice, Bob, Chan, and Dina. We will connect Alice to Bob, Bob to Chan, and Chan to Dina. This is shown in A small demonstration network of four nodes.

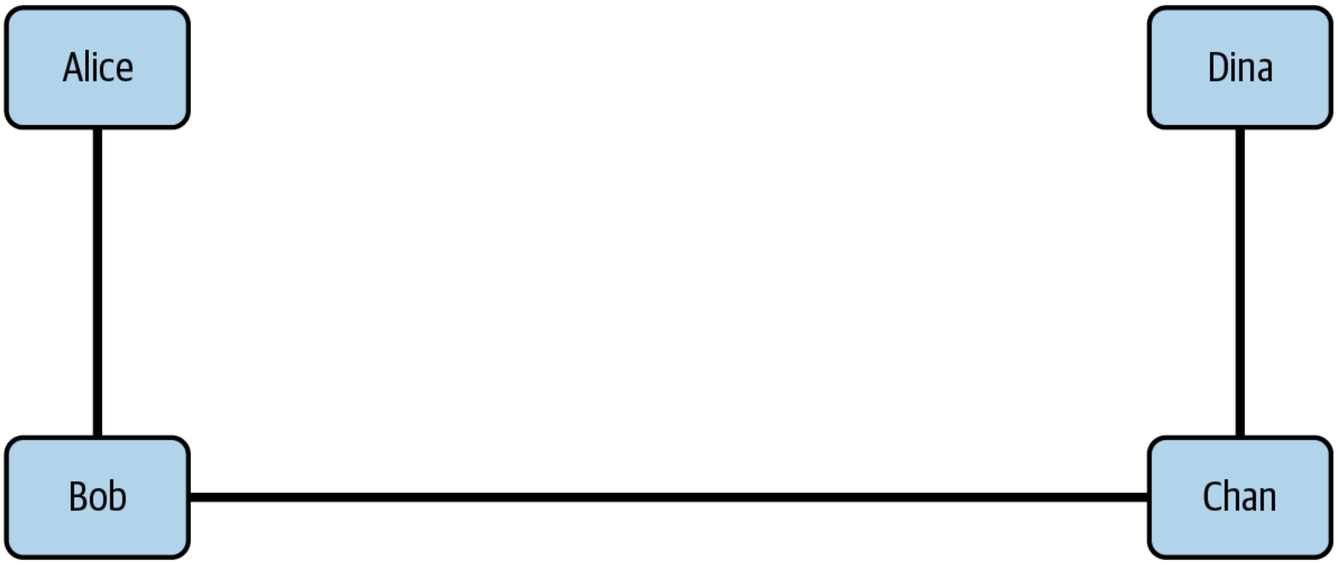


Figure 16. A small demonstration network of four nodes

Finally, we will have Dina create an invoice and have Alice pay that invoice. Since Alice and Dina are not directly connected, the payment will be routed as an HTLC across all the payment channels.

Using docker-compose to Orchestrate Docker Containers

To make this example work, we will be using a *container orchestration* tool that is available as a command called docker-compose. This command allows us to specify an application composed of several containers and run the application by launching all the cooperating containers together.

First, let's install docker-compose. The instructions depend on your operating system.

Once you have completed installation, you can verify your installation by running `docker-compose` like this:

```
$ docker-compose version
docker-compose version 1.21.0, build unknown
[...]
```

The most common docker-compose commands we will use are up and down, e.g., docker-compose up.

docker-compose Configuration

The configuration file for docker-compose is found in the *code/docker* directory and is named *docker-compose.yml*. It contains a specification for a network and each of the four containers. The top looks like this:

```
version: "3.3"
networks:
  lnnet:

services:
  bitcoind:
    container_name: bitcoind
    build:
        context: bitcoind
    image: lnbook/bitcoind:latest
    networks:
      - lnnet
    expose:
      - "18443"
      - "12005"
      - "12006"

  Alice:
    container_name: Alice
```

The preceding fragment defines a network called lnnet and a container called bitcoind which will attach to the lnnet network. The container is the same one we built at the beginning of this chapter. We expose three of the container's ports, allowing us to send commands to it and monitor blocks and transactions. Next, the configuration specifies an LND container called "Alice." Further down you will also see specifications for containers called "Bob" (`c-lightning`), "Chan" (Eclair), and "Dina" (LND again).

Since all these diverse implementations follow the Basis of Lightning Technology (BOLT) specification and have been extensively tested for interoperability, they have no difficulty working together to build a Lightning network.

Starting the Example Lightning Network

Before we get started, we should make sure we're not already running any of the containers. If a new container shares the same name as one that is already running, then it will fail to launch. Use docker ps, docker stop, and docker rm as necessary to stop and remove any currently running containers!

TIP Because we use the same names for these orchestrated Docker containers, we might need to "clean up" to avoid any name conflicts.

To start the example, we switch to the directory that contains the *docker-compose.yml* configuration file and we issue the command docker-compose up:

```
$ cd code/docker
$ docker-compose up
Creating Chan     ... done
Creating Dina     ... done
Creating bitcoind ... done
Creating Bob      ... done
Creating Alice    ... done
Attaching to Chan, Dina, Alice, bitcoind, Bob
Alice       | Waiting for bitcoind to start...
Bob         | Waiting for bitcoind to start...
Dina        | Waiting for bitcoind to start...
Chan        | Waiting for bitcoind to start...
bitcoind    | Starting bitcoind...
bitcoind    | Waiting for bitcoind to start
bitcoind    | bitcoind started
bitcoind    | ================================================

[...]

Chan        | Starting eclair...
Dina        | Starting lnd...
Chan        | Eclair node started
Alice       | ...Waiting for bitcoind to mine blocks...
Bob         | ...Waiting for bitcoind to mine blocks...
Alice       | Starting lnd...
Bob         | Starting c-lightning...

[...]
```

Following the startup, you will see a whole stream of logfiles as each node starts up and reports its progress. It may look quite jumbled on your screen, but each output line is prefixed by the container name, as seen previously. If you wanted to watch the logs from only one container, you can do so in another terminal window by using the docker-compose logs command with the f (*follow*) flag and the specific container name:

```
$ docker-compose logs -f Alice
```

Opening Channels and Routing a Payment

Our Lightning network should now be running. As we saw in the previous sections of this chapter, we can issue commands to a running Docker container with the docker exec command. Regardless of whether we started the container with docker run or started a bunch of them with docker-compose up, we can still access containers individually using the Docker commands.

The payment demo is contained in a Bash shell script called run-payment-demo.sh. To run this demo you must have the Bash shell installed on your computer. Most Linux and Unix-like systems (e.g., macOS) have bash preinstalled. Windows users can install the Windows Subsystem for Linux and use a Linux distribution like Ubuntu to get a native bash command on their computer.

Let’s run the script to see its effect, and then we will look at how it works internally. We use bash to run it as a command:

```
$ cd code/docker
$ bash run-payment-demo.sh
Starting Payment Demo
======================================================

Waiting for nodes to startup
- Waiting for bitcoind startup...
- Waiting for bitcoind mining...
- Waiting for Alice startup...
- Waiting for Bob startup...
- Waiting for Chan startup...
- Waiting for Dina startup...
All nodes have started
======================================================

Getting node IDs
- Alice:  0335e200756e156f1e13c3b901e5ed5a28b01a3131cd0656a27ac5cc20d4e71129
- Bob:    033e9cb673b641d2541aaaa821c3f9214e8a11ada57451ed5a0eab2a4afbce7daa
- Chan:   02f2f12182f56c9f86b9aa7d08df89b79782210f0928cb361de5138364695c7426
- Dina: 02d9354cec0458e0d6dee5cfa56b83040baddb4ff88ab64960e0244cc618b99bc3
======================================================

[...]

Setting up connections and channels
- Alice to Bob
- Open connection from Alice node to Bob's node

- Create payment channel Alice->Bob


[...]

Get 10k sats invoice from Dina
- Dina invoice:
lnbcrt100u1psnuzzrpp5rz5dg4wy27973yr7ehwns5ldeusceqdaq0hguu8c29n4nsqkznjsdqqcqzpgxqyz5
vqsp5vdpehw33fljnmmexa6ljk55544f3syd8nfttqlm3ljewu4r0q20q9qyyssqxh5nhkpjgfm47yxn4p9ecv
ndz7zddlsgpufnpyjl0kmnq227tdujlm0acdv39hcuqp2vhs40aav70c9yp0tee6tgzk8ut79mr877q0cpkjcf
vr
======================================================

Attempting payment from Alice to Dina
Successful payment!
```

As you can see from the output, the script first gets the node IDs (public keys) for each of the four nodes. Then, it connects the nodes and sets up a 1,000,000 satoshi channel from each node to the next in the network. Finally, it issues an invoice for 10,000 satoshis from Dina's node and pays the invoice from Alice's node.

TIP If the script fails, you can try running it again from the beginning. Or you can manually issue the commands found in the script one by one and look at the results.

There is a lot to review in that script, but as you gain understanding of the underlying technology, more and more of that information will become clear. You are invited to revisit this example later.

Of course, you can do a lot more with this test network than a three-channel, four-node payment. Here are some ideas for your experiments:

- Create a more complex network by launching many more nodes of different types. Edit the *docker-compose.yml* file and copy sections, renaming containers as needed.
- Connect the nodes in more complex topologies: circular routes, hub-and-spoke, or full mesh.
- Run lots of payments to exhaust channel capacity. Then run payments in the opposite direction to rebalance the channels. See how the routing algorithm adapts.
- Change the channel fees to see how the routing algorithm negotiates multiple routes and what optimizations it applies. Is a cheap, long route better than an expensive, short route?
- Run a circular payment from a node back to itself in order to rebalance its own channels. See how that affects all the other channels and nodes.
- Generate hundreds or thousands of small invoices in a loop and then pay them as fast as possible in another loop. Measure how many transactions per second you can squeeze out of this test network.

TIP Lightning Polar allows you to visualize the network you have been experimenting with using Docker.

Conclusion

In this chapter we looked at various projects that implement the BOLT specifications. We built containers to run a sample Lightning network and learned how to build each project from source code. You are now ready to explore further and dig deeper.

Operating a Lightning Network Node

After having read this far, you have probably set up a Lightning wallet. In this chapter, we will take things one step further and set up a full Lightning node. In addition to setting one up, we will learn how to operate it and maintain it over time.

There are many reasons why you might want to set up your own Lightning node. They include:

- To be a full, active participant in the Lightning Network, not just an end user
- To run an ecommerce store or receive income via Lightning payments
- To earn income from Lightning routing fees or by renting channel liquidity
- To develop new services, applications, or plug-ins for the Lightning Network
- To increase your financial privacy while using Lightning
- To use some apps built on top of Lightning, like Lightning-powered instant messaging apps
- For financial freedom, independence, and sovereignty

There are costs associated with running an LN node. You need a computer, a permanent internet connection, lots of disk space, and lots of time! Operational costs will include electricity expenses.

But the skills you will learn from this experience are valuable and can be applied to a variety of other tasks too.

Let’s get started!

NOTE

It is important that you set your own expectations correctly on accurate facts. If you plan to operate a Lightning node *solely* to gain income by earning routing fees, please do your homework diligently first. Running a profitable business by operating a Lightning node is definitely *not* easy. Calculate all your initial and ongoing costs in a spreadsheet. Study LN statistics carefully. What is the current payment volume? What is the volume per node? What are the current average routing fees? Consult forums and ask for advice or feedback from other community members who have already gained real-world experience. Form your own educated opinion only *after* you have done this due diligence exercise. Most people will find their motivation for running a node not in financial gain, but somewhere else.

Choosing Your Platform

There are many ways you can run a Lightning node, ranging from a small mini PC hosted in your home or a dedicated server, to a hosted server in the cloud. The method you choose will depend on the resources you have and how much money you want to spend.

Why Is Reliability Important for Running a Lightning Node?

In Bitcoin, hardware is not particularly important unless one is specifically running a mining node. The Bitcoin Core node software can be run on any machine that meets its minimum requirements and does not need to be online to receive payments—only to send them. If a Bitcoin node goes down for an extended period of time, the user can simply reboot the node, and once it connects to the rest of the network, it will resync the blockchain.

In Lightning, however, the user needs to be online both to send *and* to receive payments. If the Lightning node is offline, it cannot receive any payments from anyone, and thus its open invoices cannot be fulfilled. Furthermore, the open channels of an offline node cannot be used to route payments. Your channel partners will notice that you are offline and cannot contact you to route a payment. If you are offline too often, they may consider the bitcoin locked up in their channels with you to be underutilized capacity, and may close those channels. We already discussed the case of a protocol attack in which your channel partner tries to cheat you by submitting an earlier commitment transaction. If you are offline and your channels aren't being monitored, then the attempted theft could succeed, and you will have no recourse once the timelock expires. Hence, node reliability is extremely important for a Lightning node.

There are also the issues of hardware failure and loss of data. In Bitcoin, a hardware failure can be a trivial problem if the user has a backup of their mnemonic phrase or private keys. The Bitcoin wallet and the bitcoin inside the wallet can be easily restored from the private keys on a new computer. Most information can be redownloaded from the blockchain.

In contrast, in Lightning the information about the user's channels, including the commitment transactions and revocation secrets, are not publicly known and are only stored on the individual user's hardware. Thus, software and hardware failures in the Lightning Network can easily result in loss of funds.

Types of Hardware Lightning Nodes

There are three main types of hardware Lightning nodes:

General-purpose computers

An LN node can be run on a home computer or laptop running Windows, macOS, or Linux. Typically this is run alongside a Bitcoin node.

Dedicated hardware

A Lightning node can also be run on dedicated hardware like a Raspberry Pi, Rock64, or mini PC. This setup would usually run a software stack, including a Bitcoin node and other applications. This setup is popular because the hardware is dedicated to running and maintaining the Lightning node only and is usually set up with an installation "helper."

Preconfigured hardware

An LN node can also be run on purpose-built hardware specifically selected and configured for it. This would include "out-of-the-box" Lightning node solutions that can be purchased as a kit or a turnkey system.

Running in the "Cloud"

Virtual private server (VPS) and cloud computing services such as Microsoft Azure, Google Cloud, Amazon Web Services (AWS), or DigitalOcean are quite affordable and can be set up very quickly. A Lightning node can be hosted for between $20 and $40 per month on such a service.

However, as the saying goes, "Cloud is just other people's computers." Using these services means running your node on other people's computers. This brings along the corresponding advantages and disadvantages. The key advantages are convenience, efficiency, uptime, and possibly even cost. The cloud operator manages and runs the node to a high degree, automatically providing you with convenience and efficiency. They provide excellent uptime and availability, often much better than what an individual can achieve at home. If you consider that just the electricity cost of running a server in many Western countries is around $10 per month, then add to that the cost of network bandwidth and the hardware itself, the VPS offering becomes financially competitive. Lastly, with a VPS you need no space for a PC at home and don't have any issues with PC noise or heat. On the other hand, there are several notable disadvantages. A Lightning node running in the "cloud" will always be less secure and less private than one running on your own computer. Additionally, these cloud computing services are very centralized. The vast majority of Bitcoin and Lightning nodes running on such services are located in a handful of data centers in Virginia, Sunnyvale, Seattle, London, and Frankfurt. When the networks or data centers of these providers have service problems, it affects thousands of nodes on so-called "decentralized" networks.

If you have the possibility and capacity of running a node on your own computer at home or in your office, then this might be preferable to running it in the cloud. Nonetheless, if running your own server is not an option, by all means consider running one on a VPS.

Running a Node at Home

If you have a reasonable-capacity internet connection at home or in your office, you can certainly run a Lightning node there. Any "broadband" connection is sufficient for the purpose of running a lightweight node, and a fast connection will allow you to run a Bitcoin full node too.

While you can run a Lightning node (and even a Bitcoin node) on your laptop, it will become annoying quite fast. These programs consume your computer's resources and need to run 24/7. Your user applications like your browser or your spreadsheet will be competing against the Lightning background services for your computer's resources. In other words, your browser and other desktop workloads will be slowed down. And when your word-processing app freezes up your laptop, your Lightning node will go down as well, leaving you unable to receive transactions and potentially vulnerable to attacks. Furthermore, you should never turn off your laptop. All this combined together results in a setup that is not ideal. The same will apply to your daily-use personal desktop PC.

Instead, most users will choose to run a node on a dedicated computer. Fortunately, you don't need a "server" class computer to do this. You can run a Lightning node on a single-board computer, such as a Raspberry Pi or on a mini PC (usually marketed as home theater PCs). These are simple computers which are commonly used as a home automation hub or a media server. They are relatively inexpensive when compared to a PC or a laptop. The advantage of a dedicated device as a platform for Lightning and Bitcoin nodes is that it can run continuously, silently, and unobtrusively on your home network, tucked behind your router or TV. No one will even know that this little box is actually part of a global banking system!

WARNING Operating a node on a 32-bit operating system and/or 32-bit CPU is not recommended, because the node software may run into resource issues, causing a crash and possibly a loss of funds.

What Hardware Is Required to Run a Lightning Node?

At a minimum, the following are required to run a Lightning node:

CPU

Sufficient processing power is required to run a Bitcoin node, which will continuously download and validate new blocks. The user also needs to consider the initial block download (IBD) when setting up a new Bitcoin node, which can take anywhere from several hours to several days. A 2-core or 4-core CPU is recommended.

RAM

A system with 2 GB of RAM will *barely* run both Bitcoin and Lightning nodes. It will perform much better with at least 4 GB of RAM. The IBD will be especially challenging with less than 4 GB of RAM. More than 8 GB of RAM is unnecessary because the CPU is the greater bottleneck for these types of services, due to cryptographic operations such as signature validation.

Storage drive

This can be a hard disk drive (HDD) or a solid state drive (SSD). An SSD will be significantly quicker (but more expensive) for running a node. Most of the storage is used for the Bitcoin blockchain, which is hundreds of gigabytes in size. A fair trade-off (cost for complexity) is to buy a small SSD to boot the OS from, and a larger HDD to store large data objects (mostly databases).

NOTE

Raspberry Pis are a common choice for running node software, due to the cost and parts availability. The OS that runs on the device usually boots from a secure digital (SD) card. For most use cases, this is a nonissue, but Bitcoin Core is notorious for being I/O heavy. You should make sure to place the Bitcoin blockchain and Lightning data directory on a different drive because long-term intensive I/O can cause an SD card to fail.

Internet connection

A reliable internet connection is required to download new Bitcoin blocks, as well as to communicate with other Lightning peers. During operation the estimated data use ranges from 10 to 100 GB per month, depending on configuration. At startup, a Bitcoin full node downloads the full blockchain.

Power supply

A reliable power supply is required because Lightning nodes need to be online at all times. A power failure will cause in-progress payments to fail. For heavy duty routing nodes, a backup or uninterruptible power supply (UPS) is useful in the event of power outages. Ideally, you should connect your internet router to this UPS as well.

Backup

Backup is crucial because a failure can result in loss of data and hence in loss of funds. You will want to consider some kind of data backup solution. This could be a cloud-based automated backup to a server or web service you control. Alternatively, it could be an automated local hardware backup, such as a second hard drive. For best results, both local and remote backup can be combined.

Switching Server Configuration in the Cloud

When renting a cloud server, it is often cost effective to change the configuration between two phases of operation. A faster CPU and faster storage will be needed during the IBD (e.g., the first day). After the blockchain has synced, the CPU and storage speed requirements are much less, so the performance can be downgraded to a more cost-effective level.

For example, on Amazon's cloud, we would use an 8-16 GB RAM, 8-core CPU (e.g., t3-large or m3.large) and faster 400 GB SSD (1000+ provisioned input/output operations per second [IOPS]) for the IBD, reducing its time to just 6-8 hours. Once that is complete, we would switch the server instance to a 2 GB RAM, 2-core CPU (e.g., t3.small) and storage to a general purpose 1 TB HDD. This will cost about the same as if you ran it on the slower server the entire time, but it will get you up and running in less than a day instead of having to wait almost a week for the IBD.

Permanent data storage (drive)

If you use a mini PC or rent a server, the storage can be the most expensive part, costing as much as the computer and connectivity (data) added together.

Let's have a look at the different options available. First, there are two main types of drives, HDDs and SSDs. HDDs are cheaper and SSDs are faster, but both do the job.

The fastest SSDs available today use the Non-Volatile Memory Express (NVMe) interface. The NVMe SSDs are faster in high-end machines, but also more costly. Traditional SATA-based SSDs are cheaper, but not as fast. SATA SSDs perform sufficiently well for your node setup. Smaller computers might not be able to take advantage of NVMe SSDs. For example, the Raspberry Pi 4 cannot benefit from them because of the limited bandwidth of its USB port.

To choose the size, let's look at the Bitcoin blockchain. As of August 2021, its size is 360 GB, including the transaction index, and grows by roughly 60 GB per year. If you want to have some margin available for future growth or to install other data on your node, purchase at least a 512 GB drive, or better yet, a 1 TB drive.

Using an Installer or Helper

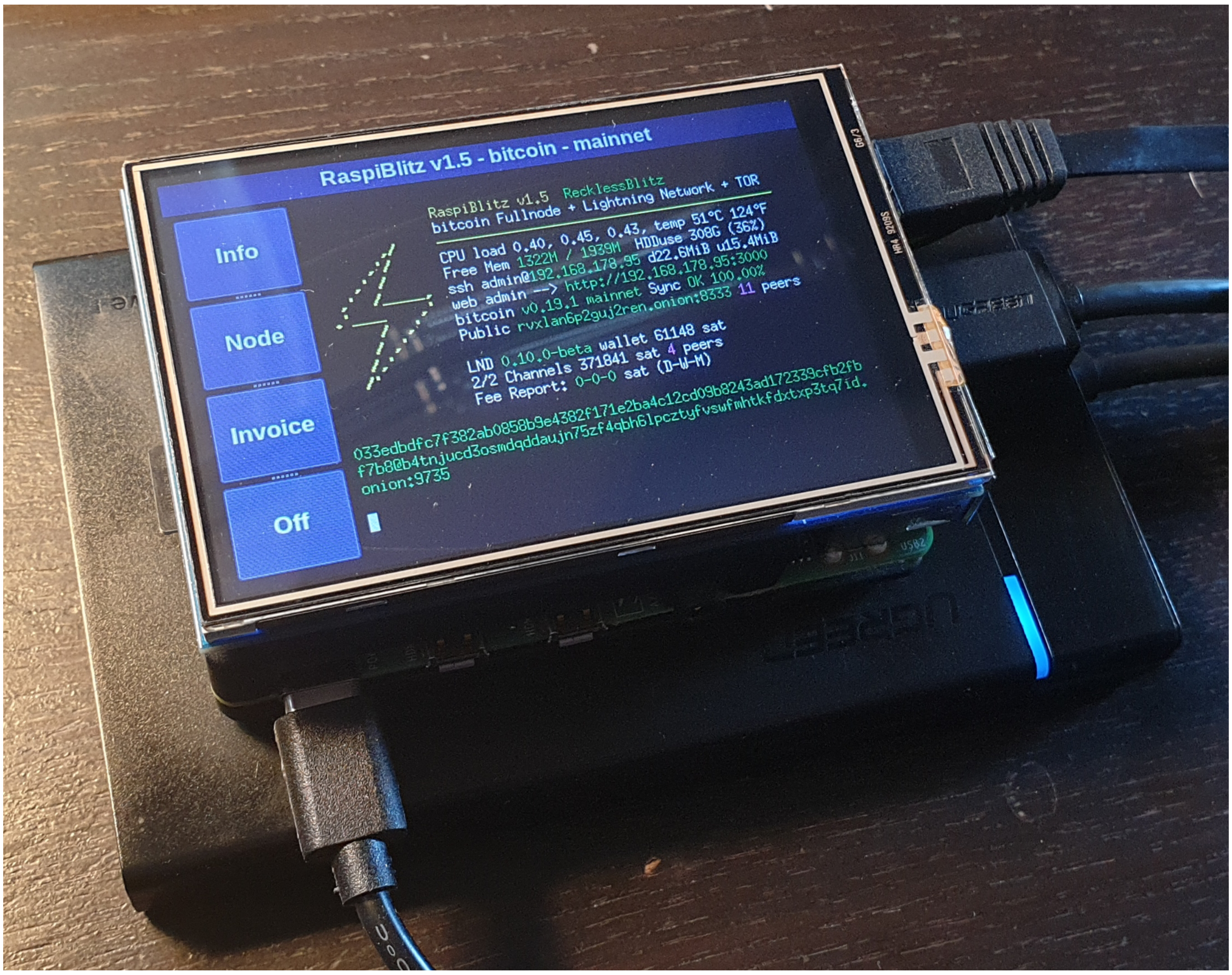
Installing a Lightning node or a Bitcoin node may be daunting if you are not familiar with a command-line environment. Luckily, there are a number of projects that make "helpers," i.e., software that installs and configures the various components for you. You will still need to learn some command-line incantations to interact with your node, but most of the initial work is done for you.

RaspiBlitz

One of the most popular and complete "helpers" is *RaspiBlitz* (A RaspiBlitz node), a project built by Christian Rotzoll. It is intended to be installed on a Raspberry Pi 4. RaspiBlitz comes with a recommended hardware kit that you can build in a matter of hours or at most a weekend. If you attend a Lightning "hackathon" in your city, you are likely to see many people working on their RaspiBlitz setup, swapping tips, and helping each other. You can find the RaspiBlitz project on GitHub.

In addition to a Bitcoin and Lightning node, RaspiBlitz can install a number of additional services, such as:

- Tor (run as hidden service)
- ElectRS (Electrum server in Rust)
- BTCPay Server (cryptocurrency payment processor)
- BTC RPC Explorer (Bitcoin blockchain explorer)
- Ride The Lightning (Lightning node management GUI)
- LNbits (Lightning wallet/accounts system)
- Specter Desktop (multisig Trezor, Ledger, Coldcard wallet, and Specter-DIY)
- lndmanage (command-line interface for advanced channel management)
- Loop (submarine swaps service)
- JoinMarket (CoinJoin service)



```
RaspiBlitz v1.5  RecklessBlitz
bitcoin Fullnode + Lightning Network + TOR
CPU load 0.40, 0.45, 0.43, temp 51°C 124°F
Free Mem 1322M / 1939M  HDDuse 308G (36%)
ssh admin@192.168.178.95 d22.6MiB u15.4MiB
web admin --> http://192.168.178.95:3000
bitcoin v0.19.1 mainnet Sync OK 100.00%
Public rvxlan6p2guj2ren.onion:8333 11 peers

LND 0.10.0-beta wallet 61148 sat
2/2 Channels 371841 sat 4 peers
Fee Report: 0-0-0 sat (D-W-M)

033edbdfc7f382ab0858b9e4382f171e2ba4c12cd09b8243ad172339cfb2fb
f7b8@b4tnjucd3osmdqddaujn75zf4qbh6lpcztyfvswfmhtkfdxtxp3tq7id.
onion:9735
```

Figure 17. A RaspiBlitz node

Mynode

myNode is another popular open source "helper" project including a lot of Bitcoin related software. It is easy to install: you "flash" the installer onto an SD card and boot your mini PC from the SD card.

You do not need any monitor to use myNode because the administrative tools are accessible remotely from a browser. If your mini PC has no monitor, mouse, or keyboard, you can manage it from another computer or even from your smartphone. Once installed, go to http://mynode.local and create a Lightning wallet and node in two clicks.

In addition to a Bitcoin and Lightning node, myNode can optionally install a variety of additional services, such as:

- Ride The Lightning (Lightning node management GUI)
- OpenVPN (virtual private network [VPN] support for remote management or wallet)
- lndmanage (command-line interface for advanced channel management)
- BTC RPC Explorer (a Bitcoin blockchain explorer)

Umbrel

Famous for their UX/UI (shown in The Umbrel web interface), *Umbrel* provides a very easy and accessible way to get your Bitcoin and Lightning node up and running in no time, especially for beginners. A very distinctive feature is that Umbrel utilizes Neutrino/SPV during the initial block download (IBD) so you can instantly start using your node. Once Bitcoin Core is fully synced in the background, it automatically switches over and disables SPV mode. Umbrel OS supports the Raspberry Pi 4 and can also be installed on any Linux-based OS or on a virtual machine on macOS or Windows. You can also connect any wallet that supports Bitcoin Core P2P, Bitcoin Core RPC, the Electrum protocol, or lndconnect.

There's no need to wait for a rainy day—you can go right to Umbrel to learn more.

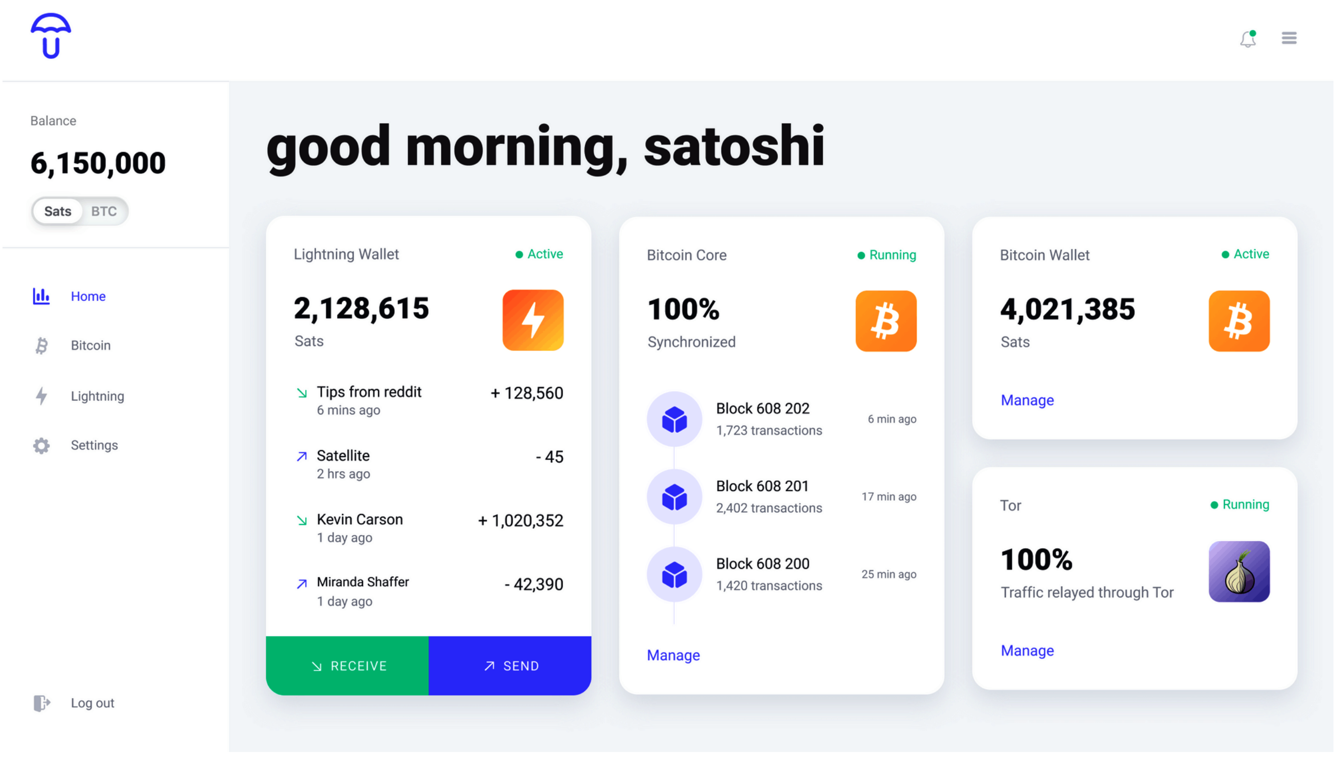


Figure 18. The Umbrel web interface

In addition to a Bitcoin and Lightning node, Umbrel introduced the Umbrel App Store, where you can easily install additional services, such as:

- Lightning Terminal (interface for managing channel liquidity, Loop In, and Loop Out)
- Ride The Lightning (Lightning node management GUI)
- Specter Desktop (watch-only coordinator for multisignature and single-key Bitcoin wallets)
- BTCPay Server (cryptocurrency payment processor)
- BTC RPC Explorer (Bitcoin blockchain explorer)
- ThunderHub (monitor and manage your node)
- Sphinx Relay (handling connectivity and storage for Sphinx chat)
- mempool.space (mempool visualizer and block explorer)
- LNbits (Lightning wallet/accounts system)

Umbrel is currently still in beta and is not considered secure.

BTCPay Server

While not initially designed as an installation "helper," the ecommerce and payment platform *BTCPay Server* has an incredibly easy installation system that uses Docker containers and docker-compose to install a Bitcoin node, Lightning node, and payment gateway, among many other services. It can be installed on a variety of hardware platforms, from a simple Raspberry Pi 4 (4 GB recommended) to a mini PC or old laptop, desktop, or server.

BTCPay Server is a fully featured self-hosted, self-custody ecommerce platform that can be integrated with many ecommerce platforms, such as WordPress WooCommerce and others. The installation of the full node is only a step of the ecommerce platform installation. While originally developed as a feature-for-feature replacement of the *BitPay* commercial payment service and API, it has evolved past that to be a complete platform for BTC and Lightning services related to ecommerce. For many sellers or shops it is a one-shop turnkey solution to ecommerce.

In addition to a Bitcoin and Lightning node, BTCPay Server can also install a variety of services, including:

- `c-lightning` or LND Lightning node
- Litecoin support
- Monero support
- Spark server (`c-lightning` web wallet)
- Charge server (`c-lightning` ecommerce API)
- Ride The Lightning (Lightning node management web GUI)
- Many BTC forks
- BTCTransmuter (event-action automation service supporting currency exchange)

The number of additional services and features is growing rapidly, so the preceding list is only a small subset of what is available on the BTCPay Server platform.

Bitcoin Node or Lightweight Lightning

One critical choice for your setup will be the choice of the Bitcoin node and its configuration. *Bitcoin Core*, the reference implementation, is the most common choice but not the only choice available. One alternative choice is *btcd*, which is a Go-language implementation of a Bitcoin node. btcd supports some features that are useful for running an LND Lightning node and are not available in Bitcoin Core.

A second consideration is whether you will run an *archival* Bitcoin node with a full copy of the blockchain (some 350 GB in mid-2021) or a *pruned* blockchain that only keeps the most recent blocks. A pruned blockchain can save you some disk space, but you will still need to download the full blockchain at least once (during the IBD). Hence it won't save you any network traffic. Using a pruned node to run a Lightning node is still an experimental capability and might not support all the functionality. However, many people are running a node like that successfully.

Finally, you also have the option of not running a Bitcoin node at all. Instead you can operate the LND Lightning node in "lightweight" mode, using the Neutrino Protocol to retrieve blockchain information from public Bitcoin nodes operated by others. Running like this means that you are taking resources from the Bitcoin network without offering any in return. Instead, you are offering your resources and contributing to the LN community. For smaller Lightning nodes this will generally reduce network traffic in comparison to running a full Bitcoin node.

Keep in mind that operating a Bitcoin node allows you to support other services, besides and on top of a Lightning node. These other services may require an archival (not pruned) Bitcoin node and often can't run without a Bitcoin node. Consider up front what other services you may want to run now or in the future to make an informed decision on the type of Bitcoin node you select.

The bottom line for this decision is: if you can afford a disk larger than 500 GB, run a full archival Bitcoin node. You will be contributing resources to the Bitcoin system and helping others who cannot afford to do so. If you can't afford such a big disk, run a pruned node. If you can't afford the disk or the bandwidth for even a pruned node, run a lightweight LND node over Neutrino.

Operating System Choice

The next step is to select an operating system for your node. The vast majority of internet servers run on some variant of Linux. Linux is the platform of choice for the internet because it is a powerful open source operating system. Linux, however, has a steep learning curve and requires familiarity with a command-line environment. It is often intimidating for new users.

Ultimately, most of the services can be run on any modern POSIX operating system, which includes macOS, Windows, and of course Linux. Your choice should be driven more by your familiarity and comfort with an operating system and your learning objectives. If you want to expand your knowledge and learn how to operate a Linux system, this is a great opportunity to do so with a specific project and a clear goal. If you just want to get a node up and running, go with what you know.

Nowadays, many services are also delivered in the form of containers, usually based on the Docker system. These containers can be deployed on a variety of operating systems, abstracting the underlying OS. You may need to learn some Linux CLI commands nonetheless, as most of the containers run some variant of Linux inside.

Choose Your Lightning Node Implementation

As with the choice of operating system, your choice of Lightning node implementation should depend primarily on your familiarity with the programming language and development tools used by the projects. While there are some small differences in features between the various node implementations, those are relatively minor, and most implementations converge on the common standards defined by the BOLTs.

Familiarity with the programming language and build system, on the other hand, is a good basis for choosing a node. That’s because installation, configuration, ongoing maintenance, and troubleshooting will all involve interacting with the various tools used by the build system. This includes:

- Make, Autotools, and GNU utilities for `c-lightning`
- Go utilities for LND
- Java/Maven for Eclair

The programming language influences not only the choice of build system but also many other aspects of the program. Each programming language comes with a whole design philosophy and affects many other aspects, such as:

- Format and syntax of configuration files
- File locations (in the filesystem)
- Command-line arguments and their syntax
- Error message formatting
- Prerequisite libraries
- Remote procedure call interfaces

When you choose your Lightning node, you are also choosing all the aforementioned characteristics. So your familiarity with these tools and design philosophies will make it easier to run a node. Or harder, if you land in an unfamiliar domain.

On the other hand, if this is your first foray into the command-line and server/service environment, you will find yourself unfamiliar with any implementation and have the opportunity to learn something completely new. In that case you might want to decide based on a number of other factors, such as:

- Quality of support forums and chat rooms
- Quality of documentation
- Degree of integration with other tools you want to run

As a final consideration, you may want to examine the performance and reliability of different node implementations. This is especially important if you will be using this node in a production environment and expect heavy traffic and high reliability requirements. This might be the case if you plan to run the payment system of a shop on it.

Installing a Bitcoin or Lightning Node

You decided not to use an installation "helper" and instead to dive into the command line of a Linux operating system? That is a brave decision, and we'll try to help you make it work. If you'd rather not try to do this manually, consider using an application that helps you install the node software or a container-based solution, as described in Using an Installer or Helper.

WARNING This section will delve into the advanced topic of system administration from the command line. Linux administration is its own skill set that is outside the scope of this book. It is a complicated topic and there are many pitfalls. Proceed with caution!

In the next few sections we will briefly describe how to install and configure a Bitcoin and Lightning node on a Linux operating system. You will need to review the installation instructions for the specific Bitcoin and Lightning node applications you decided to use. You can usually find these in a file called *INSTALL* or in the *docs* subdirectory of each project. We will only describe some of the common steps that apply to all such services, and the instructions we offer will be necessarily incomplete.

Background Services

For those accustomed to running applications on their desktop or smartphone, an application always has a graphical user interface even if it may sometimes run in the background. The Bitcoin and Lightning node applications, however, are very different. These applications do not have a graphical user interface built in. Instead, they run as *headless* background services, meaning they are always operating in the background and do not interact with the user directly.

This can create some confusion for users who are not used to running background services. How do you know if such a service is currently running? How do you start and stop it? How do you interact with it? The answers to these questions depend on the operating system you are using. For now we will assume you are using some Linux variant and answer them in that context.

Process Isolation

Background services usually run under a specific user account to isolate them from the operating system and each other. For example, Bitcoin Core is configured to run as user bitcoin. You will need to use the command line to create a user for each of the services you run.

In addition, if you have connected an external drive, you will need to tell the operating system to relocate the user's home directory to that drive. That's because a service like Bitcoin Core will create files under the user's home directory. If you are setting it up to download the full Bitcoin blockchain, these files will take up several hundred gigabytes. Here, we assume you have connected the external drive and it is located on the */external_drive/* path of the operating system.

On most Linux systems you can create a new user with the useradd command, like this:

```
$ sudo useradd -m -d /external_drive/bitcoin -s /dev/null bitcoin
```

The m and d flags create the user's home directory as specified by */external_drive/bitcoin* in this case. The s flag assigns the user's interactive shell. In this case, we set it to */dev/null* to disable interactive shell use. The last argument is the new user's username bitcoin.

Node Startup

For both Bitcoin and Lightning node services, "installation" also involves creating a so-called *startup script* to make sure that the node starts when the computer boots. Startup and shutdown of background services is handled by an operating system process, which in Linux is called init or systemd. You can usually find a system startup script in the contrib subdirectory of each project. For example, if you are on a modern Linux OS that uses systemd, you would find a script called *bitcoind.service* that can start and stop the Bitcoin Core node service.

Here's an example of what a Bitcoin node's startup script looks like, taken from the Bitcoin Core code repository:

From bitcoin/contrib/init/bitcoind.service

```
[Unit]
Description=Bitcoin daemon
After=network.target

[Service]
ExecStart=/usr/bin/bitcoind -daemon \
                            -pid=/run/bitcoind/bitcoind.pid \
                            -conf=/etc/bitcoin/bitcoin.conf \
                            -datadir=/var/lib/bitcoind

# Make sure the config directory is readable by the service user
PermissionsStartOnly=true
ExecStartPre=/bin/chgrp bitcoin /etc/bitcoin

# Process management
####################

Type=forking
PIDFile=/run/bitcoind/bitcoind.pid
Restart=on-failure
TimeoutStopSec=600

# Directory creation and permissions
####################################

# Run as bitcoin:bitcoin
User=bitcoin
Group=bitcoin

# /run/bitcoind
RuntimeDirectory=bitcoind
RuntimeDirectoryMode=0710

# /etc/bitcoin
ConfigurationDirectory=bitcoin
ConfigurationDirectoryMode=0710

# /var/lib/bitcoind
StateDirectory=bitcoind
StateDirectoryMode=0710

[...]

[Install]
WantedBy=multi-user.target
```

As the root user, install the script by copying it into the systemd service folder */lib/systemd/system/* and then reload systemd:

```
$ sudo systemctl daemon-reload
```

Next, enable the service:

```
$ sudo systemctl enable bitcoind
```

You can now start and stop the service. Don’t start it yet, as we haven’t configured the Bitcoin node.

```
$ sudo systemctl start bitcoind
$ sudo systemctl stop bitcoind
```

Node Configuration

To configure your node, you need to create and reference a configuration file. By convention, this file is usually created in */etc*, under a directory with the name of the program. For example, Bitcoin Core and LND configurations would usually be stored in */etc/bitcoin/bitcoin.conf* and */etc/lnd/lnd.conf*, respectively.

These configuration files are text files with each line expressing one configuration option and its value. Default values are assumed for anything not defined in the configuration file. You can see what options can be set in the configuration in two ways. First, running the node application with a help argument will show the options that can be defined on the command line. These same options can be defined in the configuration file. Second, you can usually find an example configuration file, with all the default options, in the code repository of the software.

You can find one example of a configuration file in each of the Docker images we used in Lightning Node Software. For example, the file *code/docker/bitcoind/bitcoind/bitcoin.conf*:

Configuration file for docker bitcoind (code/docker/bitcoind/bitcoind/bitcoin.conf)

```
regtest=1
server=1
debuglogfile=debug.log
debug=1
txindex=1
printtoconsole=0

[regtest]
fallbackfee=0.000001
port=18444
noconnect=1
dnsseed=0
dns=0
upnp=0
onlynet=ipv4
rpcport=18443
rpcbind=0.0.0.0
rpcallowip=0.0.0.0/0
rpcuser=regtest
rpcpassword=regtest
zmqpubrawblock=tcp://0.0.0.0:12005
zmqpubrawtx=tcp://0.0.0.0:12006
```

That particular configuration file configures Bitcoin Core for operation as a regtest node and provides a weak username and password for remote access, so you shouldn't use it for your node configuration. However, it serves to illustrate the syntax of a configuration file and you can make adjustments to it in the Docker container to experiment with different options. See if you can use the bitcoind -help command to understand what each of the options does in the context of the Docker network we built in Lightning Node Software.

Often, the defaults suffice, and with a few modifications your node software can be configured quickly. To get a Bitcoin Core node running with minimal customization, you only need four lines of configuration:

```
server=1
daemon=1
txindex=1
rpcuser=USERNAME
rpcpassword=PASSWORD
```

Even the txindex option is not strictly necessary, though it will ensure your Bitcoin node creates an index of all transactions, which is required for some applications. The txindex option is not required to run a Lightning node.

A `c-lightning` Lightning node running on the same server also requires only a few lines in the configuration:

```
network=mainnet
bitcoin-rpcuser=USERNAME
bitcoin-rpcpassword=PASSWORD
```

In general, it is a good idea to minimize the amount of customization of these systems. The default configuration is carefully designed to support the most common deployments. If you modify a default value, it may cause problems later on or reduce the performance of your node. In short, modify only when necessary!

Network Configuration

Network configuration is normally not an issue when configuring a new application. However, peer-to-peer networks like Bitcoin and the Lightning Network present some unique challenges for network configuration.

In a centralized service, your computer connects to the "big servers" of some corporation, and not vice versa. Your home internet connection is actually configured on the assumption that you are simply a consumer of services provided by others. But in a peer-to-peer system, every peer both consumes from and provides services to other nodes. If you’re running a Bitcoin or Lightning node at your home, you’re providing a service to other computers on the internet. Your internet service by default is not configured to allow you to run servers and may need some additional configuration to enable others to reach your node.

If you want to run a Bitcoin or Lightning node, you need to make it possible for other nodes on the internet to connect to you. That means enabling incoming TCP connections to the Bitcoin port (port 8333 by default) or Lightning port (port 9735 by default). While you can run a Bitcoin node without incoming connectivity, you can’t do that with a Lightning node. A Lightning node must be accessible to others from outside your network.

By default, your home internet router does not expect incoming connections from the outside, and in fact incoming connections are blocked. Your internet router IP address is the only externally accessible IP address, and all the computers you run inside your home network share that single IP address. This is achieved by a mechanism called *Network Address Translation* (*NAT*), which allows your internet router to act as an intermediary for all outbound connections. If you want to allow an inbound connection, you have to set up *port forwarding*, which tells your internet router that incoming connections on specific ports should be forwarded to specific computers inside the network. You can do this manually by changing your internet router configuration or, if your router supports it, through an automatic port forwarding mechanism called *Universal Plug and Play* (*UPnP*).

An alternative mechanism to port forwarding is to enable The Onion Router (Tor), which provides a kind of virtual private network overlay that allows incoming connections to an *onion address*. If you run Tor, you don’t need to do port forwarding or enable incoming connections to Bitcoin or Lightning ports. If you run your nodes using Tor, all traffic goes through Tor and no other ports are used.

Let’s look at different ways you can make it possible for others to connect to your node. We’ll look at these alternatives in order, from easiest to most difficult.

It just works!

There is a possibility that your internet service provider or router is configured to support UPnP by default and everything just works automatically. Let's try this approach first, just in case we are lucky.

Assuming you already have a Bitcoin or Lightning node running, we will try and see if they are accessible from the outside.

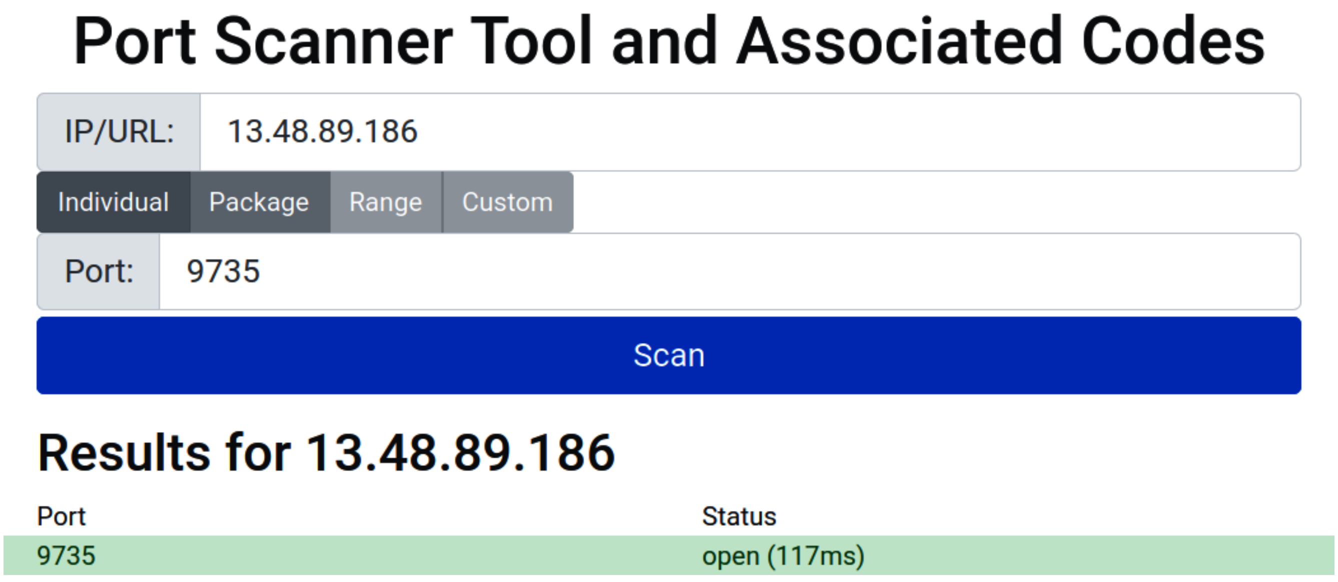
NOTE	For this test to work, you have to have either a Bitcoin or Lightning node (or both) up and running on your home network. If your router supports UPnP, the incoming traffic will automatically be forwarded to the corresponding ports on the computer running the node.

You can use some very popular and useful websites to find out what is your external IP address and whether it allows and forwards incoming connections to a known port. Here are two that are reliable:

- https://canyouseeme.org/
- https://www.whatismyip.com/port-scanner/

By default, these services only allow you to check incoming connections to the IP address from which you are connecting. This is done to prevent you from using the service to scan other people's networks and computers. You will see your router's external IP address and a field for entering a port number. If you haven't changed the default ports in your node configuration, try port 8333 (Bitcoin) and/or 9735 (Lightning).

In Checking for incoming port 9735 you can see the result of checking port 9735 on a server running Lightning, using the *whatismyip.com* port scanner tool. It shows that the server is accepting incoming connections to the Lightning port. If you see a result like this, you are all set!


Port Scanner Tool and Associated Codes

IP/URL: 13.48.89.186

Port: 9735

Results for 13.48.89.186

Port	Status
9735	open (117ms)

Figure 19. Checking for incoming port 9735

Automatic port forwarding using UPnP

Sometimes, even if your internet router supports UPnP, it may be turned off by default. In that case you need to change your internet router configuration from its web administration interface:

1. Connect to your internet router's configuration website. Usually this can be done by connecting to the *gateway address* of your home network using a web browser. You can find the gateway address by looking at the IP configuration of any computer on your home network. It is often the first address in one of the nonroutable networks, like 192.168.0.1 or 10.0.0.1. Check all stickers on your router as well for the *gateway address*. Once found, open a browser and enter the IP address into the browser URL/Search box, e.g., "192.168.0.1" or "http://192.168.0.1."
2. Find the administrator username and password for the web configuration panel of the router. This is often written on a sticker on the router itself and may be as simple as "admin" and "password." A quick web search for your ISP and router model can also help you find this information.
3. Find a setting for UPnP and turn it on.

Restart your Bitcoin and/or Lightning node and repeat the open port test with one of the websites we used in the previous section.

Using Tor for incoming connections

The Onion Router (*Tor*) is a VPN with the special property that it encrypts communications between hops, such that any intermediary node cannot determine the origin or destination of a packet. Both Bitcoin and Lightning nodes support operation over Tor, which enables you to operate a node without revealing your IP address or location. Hence, it provides a high level of privacy to your network traffic. An added benefit of running Tor is that, because it operates as a VPN, it resolves the problem of port forwarding from your internet router. Incoming connections are received over the Tor tunnel, and your node can be found through an ad hoc generated *onion address* instead of an IP address.

Enabling Tor requires two steps. First, you must install the Tor router and proxy on your computer. Second, you must enable the use of the Tor proxy in your Bitcoin or Lightning configuration.

To install Tor on an Ubuntu Linux system that uses the apt package manager, run:

```
sudo apt install tor
```

Next, we configure our Lightning node to use Tor for its external connectivity. Here is an example configuration for LND:

```
[Tor]
tor.active=true
tor.v3=true
tor.streamisolation=true
listen=localhost
```

This will enable Tor (tor.active), establish a v3 onion service (tor.v3=true), use a different onion stream for each connection (tor.streamisolation), and restrict listening for connections to the local host only, to avoid leaking your IP address (`listen=localhost`).

You can check if Tor is correctly installed and working by running a simple one-line command. This command should work on most flavors of Linux:

```
curl --socks5 localhost:9050 --socks5-hostname localhost:9050 -s
https://check.torproject.org/ | cat | grep -m 1 Congratulations | xargs
```

If everything is working properly, the response of this command should be "Congratulations. This browser is configured to use Tor."

Due to the nature of Tor, you can’t easily use an external service to check if your node is reachable via an onion address. Nonetheless, you should see your Tor onion address in the logs of your Lightning node. It is a long string of letters and numbers followed by the suffix .onion. Your node should now be reachable from the internet, with the added bonus of privacy!

Manual port forwarding

This is the most complex process and requires quite a bit of technical skill. The details depend on the type of internet router you have, your service provider settings and policies, and a lot of other context. Try UPnP or Tor first, before you try this much more difficult mechanism.

The basic steps are as follows:

1. Find the IP address of the computer your node is on. This is usually dynamically allocated by the Dynamic Host Configuration Protocol (DHCP) and is often somewhere in the 192.168.x.x or 10.x.x.x range.
2. Find the media access control (MAC) address of your node’s network interface. This can be found in the internet settings of that computer.
3. Assign a static IP address for your node so that it is always the same one. You can use the IP address it currently has. On your internet router, look for "Static Leases" under the DHCP configuration. Map the MAC address to the IP address you selected. Now your node will always have that IP address allocated to it. Alternatively, you can look at your router’s DHCP configuration and find out what its DHCP address range is. Select an unused address *outside* of the DHCP address range. Then, on the server, configure the network to stop using DHCP and hardcode the selected non-DHCP IP address into the operating system network configuration.
4. Finally, set up "Port Forwarding" on your internet router to route incoming traffic on specific ports to the selected IP address of your server.

Once done reconfiguring, repeat the port check using one of the websites from the previous sections.

Security of Your Node

A Lightning node is, by definition, a *hot wallet*. That means that the funds (both on-chain and off-chain) controlled by a Lightning node are directly controlled by keys that are loaded in the node's memory or stored on the node's hard disk. If a Lightning node is compromised, it is trivial to create on-chain or off-chain transactions to drain its funds. It is therefore critically important that you protect it from unauthorized access.

Security is a holistic effort, meaning that you have to secure every layer of a system. As the saying goes: the chain is only as strong as the weakest link. This is an important concept in information security, and we will apply it to our node.

Despite all the security measures you will take, remember that the Lightning Network is an early-stage experimental technology and there are likely to be exploitable bugs in the code of any project you use. *Do not put more money than you are willing to risk losing on the Lightning Network.*

Operating System Security

Securing an operating system is a vast topic that is beyond the scope of this book. However, we can establish some basic principles.

To secure your operating system, here are some of the top items to consider:

Provenance

Start by ensuring that you are downloading the correct operating system image, and verify any signatures or checksums before installing it. Extend this to any software that you install. Double-check any source or URL from where you download. Verify the integrity and correctness of the downloaded software via signature and checksum verification.

Maintenance

Make sure that you keep your operating system up to date. Enable automated daily or weekly installation of security updates. Least privilege: set up users for specific processes and give them the least access needed to run a service. Do not run processes with admin privileges (e.g., root).

Process isolation

Use the operating system features to isolate processes from each other.

Filesystem permissions

Configure the filesystem carefully, on the least-privilege principle. Do not make files readable or writable by everyone.

Strong authentication

Use strong randomly generated passwords or, whenever possible, public-key authentication. For example, it is safer to use Secure Shell (SSH) with a cryptographic key pair instead of a password.

Two-factor authentication (2FA)

Use two-factor authentication wherever possible, including Universal 2nd Factor (U2F) with hardware security keys. This applies to all external services you might be using, such as your cloud service provider. You can apply this also to your own setup, such as your own SSH configuration. Use 2FA also for indirect services. For example, say you are using a cloud service. You gave your cloud service provider an email address, so you should also protect your email address with 2FA.

Backup

Make backups of your system, and make sure you protect the backups with encryption too. Perform these backups periodically. At least once, test if you can restore your backup and that your backup is complete and accessible. If possible, keep one copy of your backups on a different disk to avoid a single hard disk failure destroying *both* your active node as well as your backup copies.

Vulnerability and exposure management

Use remote scanning to ensure you have minimized the attack surface of your system. Close any unnecessary services or ports. Install only software and packages that you really need and use. Uninstall packages that you no longer use. It is recommended that you do *not* use your node computer for non-node activities that you can perform on another of your computers. Especially, if you can, do *not* use your node computer for browsing, surfing the internet, or reading your email.

This is a list of the most basic security measures. It is by no means exhaustive.

Node Access

Your Lightning node will expose a remote procedure call (RPC) API. This means that your node can be controlled remotely by commands sent to a specific TCP port. Access control to that RPC API is achieved by some form of user authentication. Depending on the type of Lightning node you set up, this will either be done by username/password authentication or by a mechanism called an authentication *macaroon*. As the name implies, a macaroon is a more sophisticated type of cookie. Unlike a cookie, it is cryptographically signed and can express a set of access capabilities.

For example, LND uses macaroons to grant access to the RPC API. By default, the LND software creates three macaroons with different levels of access, called admin, invoice, and readonly. Depending on which macaroon you copy and use in your RPC client, you either have *read-only* access, *invoice* access (which includes the read-only capabilities), or *admin* access, which gives you full control. There is also a macaroon bakery function in LND that can construct macaroons with any combination of capabilities with very fine-grained control.

If you use a username/password authentication model, make sure you select a long and random password. You will not have to type this password often, because it will be stored in the configuration files. You should therefore pick one that cannot be guessed. Many of the examples you will see include poorly chosen passwords, and often people copy these into their own systems, providing easy access to anyone. Don’t do that! Use a password manager to generate a long random alphanumeric password. Since certain special characters such as $?/!*\&%`"' can interfere with the command line, it is best to avoid these for passwords that will be used in a shell environment. To avoid problems, stick with long random alphanumeric passwords.

A plain alphanumeric sequence that is longer than 12 characters and randomly generated is usually sufficient. If you plan to store large amounts of money on your Lightning node and are concerned about remote brute-force attacks, select a password length of more than 20 characters to make such attacks practically infeasible.

Node and Channel Backups

A very important consideration when running a Lightning node is the issue of backups. Unlike a Bitcoin wallet, where a BIP-39 mnemonic phrase can recover all the state of the wallet, in Lightning this is *not* the case.

Lightning wallets do use a BIP-39 mnemonic phrase backup, but only for the on-chain wallet. However, due to the way channels are constructed, the mnemonic phrase is *not* sufficient to restore a Lightning node. An additional layer of backups is needed, which is called the *static channel backup* (*SCB*). Without an SCB, a Lightning node operator may lose *all* the funds that are in channels if they lose the Lightning node data store.

WARNING Do *not* fund channels until you have put a system in place to continuously back up your channel state. Your backups should be moved "offsite" to a different system and location from your node, so that they can survive a variety of system failures (power loss, data corruption, etc.) or natural disasters (flood, fire, etc.).

SCBs are not a panacea. First, the state of each channel needs to be backed up every time there is a new commitment transaction. Second, restoring from a channel backup is dangerous. If you do not have the *last* commitment transaction and you accidentally broadcast an old (revoked) commitment, your channel peer will assume you are trying to cheat and claim the entire channel balance with a penalty transaction. To make sure you are closing the channel, you need to do a *cooperative close*. But a malicious peer could mislead your node into broadcasting an old, revoked commitment during that cooperative close, thereby cheating you by making your node inadvertently try to cheat.

Furthermore, the backups of your channels need to be encrypted to maintain your privacy and your channel security. Otherwise, anyone who finds the backups can not only see all your channels but also could use the backups to close all your channels in a way that hands over the balance to your channel peers. In other words, a malicious person that gets access to your backups can cause you to lose all your channel funds.

You can see that SCBs are not a foolproof safeguard. They are a weak compromise because they swap one type of risk (data corruption or loss) for another type of risk (malicious peer). To restore from an SCB, you have to interact with your channel peers and hope they don’t try to cheat you by either giving you an old commitment or by fooling your node into broadcasting a revoked commitment so they can penalize you. Despite the weaknesses of SCB, SCBs do make sense and you should perform them. If you do not perform SCBs and you lose your node data, you will lose your channel funds forever. Guaranteed! However, if you *do* perform SCBs and you lose your node data, then you have a reasonable chance that some of your peers are honest and that you can recover some of your channel funds. If you are lucky, you might recover all your funds. In conclusion, it is best for you to perform continuous SCBs to a disk other than the primary node hard disk.

Channel backup mechanisms are still a work in progress and a weakness in most Lightning implementations.

At the time of writing this book, only LND offers a built-in mechanism for SCBs. Eclair has a similar mechanism deployed for server-side deployments, although Eclair Mobile does offer optional backup to a Google Drive. `c-lightning` recently merged the necessary interfaces for a plug-in to implement channel backups. Unfortunately, there is no consistent, agreed upon backup mechanism across different node implementations.

File-based backups of the Lightning node databases are at best a partial solution because you run the risk of backing up an inconsistent database state. In addition, you may not reliably catch the latest state commitments. It is much better to have a backup mechanism that is triggered every time there is a state change in a channel, thereby ensuring data consistency.

To set up SCBs in LND, set the backupfilepath parameter either on the command line or in the configuration file. LND will then save an SCB file in that directory path. Of course, that's only the first step of the solution. Now, you have to set up a mechanism that monitors this file for changes. Each time the file changes, the backup mechanism must copy this file to another, preferably off-site disk. Such backup mechanisms are beyond the scope of this book. Nonetheless, any sophisticated backup solution should be able to handle this scenario. Recall, the backup files should be encrypted too.

Hot Wallet Risk

As we've discussed previously, the Lightning Network consists of a network of *hot wallets*. The funds you store in a Lightning wallet are online all the time. This makes them vulnerable. Hence, you should not store large amounts in a Lightning wallet. Large amounts should be kept in a *cold* wallet that is *not* online and which can transact only on-chain.

Even if you start small, as time passes you may still find you have a significant amount of money in a Lightning wallet. This is a typical scenario for store owners. If you use a Lightning node for an ecommerce operation, your wallet will likely receive funds often, but send funds rarely. You will therefore end up having two problems simultaneously. First, your channels will be imbalanced, with large local balances outweighing small remote balances. Secondly, you will have too much money in the wallet. Fortunately, you can also solve both of these problems simultaneously.

Let's look at some of the solutions you can use to reduce the funds exposed in a hot wallet.

Sweeping Funds

If your Lightning wallet balance becomes too large for your risk tolerance, you will need to "sweep" funds out of the wallet. You can do so in three ways: on-chain, off-chain, and Loop Out. Let's look at each of these options in the next few sections.

On-chain sweep

Sweeping funds on-chain is accomplished by moving the funds from the Lightning wallet to a Bitcoin wallet. You do that by closing channels. When you close a channel, all funds from your local balance are "swept" to a Bitcoin address. The Bitcoin address for on-chain funds is usually generated by your Lightning wallet, so it is still a hot wallet. You may need to do an additional on-chain transaction to move the funds to a more secure address, such as one generated on your hardware wallet.

Closing channels will incur an on-chain fee and will reduce your Lightning node’s capacity and connectivity. However, if you run a popular ecommerce node, you will not lack incoming capacity and can strategically close channels with large local balances, essentially "bundling" your funds for movement on-chain. You may need to use some channel rebalancing techniques (see Rebalancing Channels) before you close channels to maximize the benefits of this strategy.

Off-chain sweep

Another technique you can use involves running a second Lightning node that is not advertised on the network. You can establish large capacity channels from your public node (e.g., the one running your shop) to your unadvertised (hidden) node. On a regular basis, "sweep" funds by making a Lightning payment to your hidden node.

The advantage of this technique lies in the fact that the Lightning node that receives payments for your shop will be publicly known. This makes it a target for hackers, as any Lightning node associated with a shop would be assumed to have a large balance. A second node that is not associated with your shop will not easily be identified as a valuable target.

As an additional measure of security, you can make your second node a hidden Tor service so that its IP address is not known. That further reduces the opportunity for attacks and increases your privacy.

You will need to set up a script that runs at regular intervals. The purpose of this script is to create an invoice on your hidden node and to pay that invoice from your shop’s node, thereby shifting funds over to your hidden node.

Keep in mind that this technique does not move funds into cold storage. Both Lightning nodes are hot wallets. The objective of this sweep is to move funds from a very well-known hot wallet to an obscure hot wallet.

Submarine swap sweep

Another way to reduce your Lightning hot-wallet balance is to use a technique called a *submarine swap*. Submarine swaps, conceptualized by coauthor Olaoluwa Osuntokun and Alex Bosworth, allow the exchange of on-chain bitcoin for Lightning payments and vice versa. Essentially, submarine swaps are atomic swaps between Lightning off-chain funds and Bitcoin on-chain funds.

A node operator can initiate a submarine swap and send all available channel balances to the other party, who will send them on-chain bitcoin in exchange.

In the future, this could be a paid service offered by nodes on the Lightning Network who advertise exchange rates or charge a flat fee for the conversion.

The advantage of a submarine swap for sweeping funds is that no channel needs to be closed. That means that we preserve our channels, only rebalancing our channels through this operation. As we send a Lightning payment out, we shift some balance from local to remote on one or more of our channels. Not only does that reduce the balance exposed in our node’s hot wallet, it also increases the balance available for future incoming payments.

You could do this by trusting an intermediary to act as a gateway, but this risks your coins being stolen. However, in the case of a submarine swap, the operation does not require trust. Submarine swaps are noncustodial *atomic* operations. That means that the counterparty in your submarine swap cannot steal your funds because the on-chain payment depends on the completion of the off-chain payment and vice versa.

Submarine swaps with Loop

One example of a submarine swap service is *Loop* by Lightning Labs, the same company that builds LND. Loop comes in two variations: Loop In and Loop Out. *Loop In* accepts a Bitcoin on-chain payment and converts it into a Lightning off-chain payment. *Loop Out* converts a Lightning payment into a Bitcoin payment.

NOTE To use the Loop service you must be running an LND Lightning node.

For the purpose of reducing the balance of your Lightning hot wallet, you would use the Loop Out service. To use the Loop service, you need to install some additional software on your node. The Loop software runs alongside your LND node and provides some command-line tools to execute submarine swaps. You can find the Loop software and installation instructions on GitHub.

Once you have the software installed and running, a Loop Out operation is as simple as running a single command:

```
loop out --amt 501000 --conf_target 400
Max swap fees for 501000 sat Loop Out: 25716 sat
Regular swap speed requested, it might take up to 30m0s for the swap to be executed.
CONTINUE SWAP? (y/n), expand fee detail (x): x

Estimated on-chain sweep fee:        149 sat
Max on-chain sweep fee:              14900 sat
Max off-chain swap routing fee:      10030 sat
Max no show penalty (prepay):        1337 sat
Max off-chain prepay routing fee:    36 sat
Max swap fee:                        750 sat
CONTINUE SWAP? (y/n): y
Swap initiated

Run `loop monitor` to monitor progress.
```

Note that your maximum fee, which represents a worst-case scenario, will depend on the confirmation target that you select.

Lightning Node Uptime and Availability

Unlike Bitcoin, Lightning nodes need to be online almost continuously. Your node needs to be online to receive payments, open channels, close channels (cooperatively), and monitor protocol violations. Node availability is such an important requirement in the Lightning Network that it is a metric used by various automatic channel management tools (e.g., autopilot) to decide which nodes to open channels with. You can also see "availability" as a node metric on popular node explorers (see Lightning Explorers) such as 1ML.

Node availability is especially important to mitigate and resolve potential protocol violations (i.e., revoked commitments). While you can afford short interruptions from an hour up to one or two days, you cannot have your node offline for longer periods of time without risking loss of funds.

Keeping a node online continuously is not easy, as various bugs and resource limitations can and will occasionally cause downtime. Especially if you run a busy and popular node, you will run into limitations of memory, swap space, number of open files, disk space, and so forth. A whole host of different problems will cause your node or your server to crash.

Tolerate Faults and Automate

If you have the time and skills, you should test some basic fault scenarios on the Lightning testnet. On the testnet you will learn valuable lessons without risking any funds. Any step you perform to automate your system will improve your availability:

Automatic computer server restart

What happens when your server or the operating system crashes? What happens when there is a power outage? Simulate this fault by pressing the "reset" button on your PC or by unplugging the power cable. After a crash, reset, or power failure, the computer should automatically restart itself. Some computers have a setting in their BIOS to specify how the computer should react on power failures. Test it to make sure the computer really restarts automatically on power failure without human intervention.

Automatic node restart

What happens when your node or one of your nodes crashes? Simulate this fault by killing the corresponding node processes. If a node crashes, it should automatically restart itself. Test it to make sure the node or nodes really restart automatically on failure without human intervention. If this is not the case, most likely your node is not set up correctly as an operating system service.

Automatic network reconnection

What happens if your network goes down? What happens when your ISP goes down temporarily? What happens when your ISP assigns a new IP address to your router or your computer? When the network comes back, do the nodes you are running automatically reconnect to the network? Simulate this fault by unplugging and later replugging the Ethernet cable from the device hosting your nodes. The nodes should automatically reconnect and continue operation without human intervention.

Configure your logfiles

All of the preceding failures should leave textual entries behind in the corresponding logfiles. Turn up the verbosity of logging if needed. Find these error entries in the logfiles and use them for monitoring.

Monitoring Node Availability

Monitoring your node is an important part of keeping it running. You need to monitor not only the availability of the computer itself, but also the availability and correct operation of the Lightning node software.

There are a number of ways to do this, but most require some customization. You can use generic infrastructure monitoring or application monitoring tools, but you have to customize them specifically to query the Lightning node API to ensure the node is running, synchronized to the blockchain, and connected to channel peers.

Lightning.watch provides a specialized service that offers Lightning node monitoring. It uses a Telegram bot to notify you of any interruptions in service. This is a free service, though you can pay (over Lightning, of course) to get faster alerts.

Over time, we expect more third-party services to provide specialized Lightning node monitoring payable via micropayments. Perhaps such services and their APIs will become standardized and will one day be directly supported by Lightning node software.

Watchtowers

Watchtowers are a mechanism for outsourcing the monitoring and penalty resolution of Lightning protocol violations.

As we mentioned in previous chapters, the Lightning protocol maintains security through a penalty mechanism. If one of your channel partners broadcasts an old commitment transaction, your node will need to exercise the revocation clause and broadcast a penalty transaction to avoid losing money. But if your node is down during the protocol violation, you might lose money.

To solve this problem, we can use one or more watchtowers to outsource the job of monitoring protocol violations and issuing penalty transactions. There are two parts to a watchtower setup: a watchtower server (or simply watchtower) that monitors the blockchain and a watchtower client that asks the watchtower server for this monitoring service.

Watchtower technology is still in the early stages of development and is not widely supported. However, in the following passage we list some experimental implementations that you can try.

LND software includes both a watchtower server and a watchtower client. You can activate the watchtower server by adding the following configuration options:

```
[watchtower]
watchtower.active=1
watchtower.towerdir=/path_to_watchtower_data_directory
```

You can use LND's watchtower client by activating it in the configuration and then using the command line to connect it to a watchtower server. The configuration is:

```
[wtclient]
wtclient.active=1
```

LND's command-line client lncli shows the following options for managing the watchtower client:

```
$ lncli wtclient

NAME:
   lncli wtclient - Interact with the watchtower client.

USAGE:
   lncli wtclient command [command options] [arguments...]

COMMANDS:
     add     Register a watchtower to use for future sessions/backups.
     remove  Remove a watchtower to prevent its use for future sessions/backups.
     towers  Display information about all registered watchtowers.
     tower   Display information about a specific registered watchtower.
     stats   Display the session stats of the watchtower client.
     policy  Display the active watchtower client policy configuration.

OPTIONS:
   --help, -h  show help
```

`c-lightning` has the API hooks necessary for a watchtower client plug-in, though no such plug-in has been implemented yet.

Finally, a popular standalone watchtower server is *The Eye of Satoshi* (TEOS). It can be found on GitHub.

Channel Management

As a Lightning node operator, one of the recurring tasks you will need to perform is management of your channels. This means opening outbound channels from your node to other nodes, as well as getting other nodes to open inbound channels to your node. In the future, cooperative channel construction may be possible, so you can open symmetric channels that have funds committed on both ends on creation. For now, however, new channels only have funds on one end, on the originator's side. Hence, to make your node *balanced* with both inbound and outbound capacity, you need to open channels to others and entice others to open channels to your node.

Opening Outbound Channels

As soon as you get your Lightning node up and running, you can fund its Bitcoin wallet and then start opening channels with those funds.

You must choose channel partners carefully because your node's ability to send payments depends on who your channel partners are and how well connected they are to the rest of the Lightning Network. You also want to have more than one channel to avoid being susceptible to a single point of failure. Since Lightning now supports multipart payments, you can split your initial funds into several channels and route bigger payments by combining their capacity. At the same time, avoid making your channels too small. Since you need to pay Bitcoin transaction fees to open and close a channel, the channel balance should not be so small that the on-chain fees consume a significant portion. It's all about balance!

To summarize:

- Connect to a few well-connected nodes
- Open more than one channel
- Don’t open too many channels
- Don’t make the channels too small

One way to find well-connected nodes is to open a channel to a popular merchant selling products on the Lightning Network. These nodes tend to be well funded and well connected. So, when you are ready to buy something online via Lightning, you can open a channel directly to the merchant’s node. The merchant’s node ID will be in the invoice you will receive when you try to buy something. That makes it easy.

Another way to find well-connected nodes is to use a Lightning Explorer (see Lightning Explorers) such as 1ML and browse the list of nodes sorted by channel capacity and number of channels. Don’t go for the biggest nodes, because that encourages centralization. Go for a node in the middle of the list so that you can help them grow. Another factor to consider might be the time span a node has been in operation. Nodes established for more than a year are likely to be more reputable and less risky than nodes that started operation a week ago.

Autopilot

The task of opening channels can be partially automated with the use of an *autopilot*, which is software that opens channels automatically based on some heuristic rules. Autopilot software is still relatively new, and it doesn’t always select the best channel partners for you. Especially in the beginning, it might be better to open channels manually. Autopilots currently exist in three forms:

- lnd incorporates an autopilot that is fully integrated with lnd and runs constantly in the background while turned on.
- lib_autopilot.py can offer autopilot computations for any node implementation based on the gossip and channel data.
- A c-lightning plug-in based on lib_autopilot.py exists that provides an easy-to-use interface for c-lightning users.

Be aware that the lnd autopilot will start running in the background as soon as it is turned on via the config file. As a result it will start opening channels immediately if you have on-chain outputs in your lnd wallet. If you want to have full control over the bitcoin transactions that you make and the channels that you open, make sure to turn the autopilot off *before* you load your lnd wallet with bitcoin funds. If the autopilot was previously turned on, you might have to restart your lnd before you top up your wallet with an on-chain transaction or before you close channels, which effectively gives you on-chain funds again. It is crucial that you set key configuration values if you want to run the autopilot. Have a look at this example configuration:

```
[lnd-autopilot]
autopilot.active=1
autopilot.maxchannels=40
autopilot.allocation=0.70
autopilot.minchansize=500000
autopilot.maxchansize=5000000
autopilot.heuristic=top_centrality:1.0
```

This configuration file would activate the autopilot. It would open channels as long as the following two conditions are met:

1. Your node currently has less than 40 channels open.
2. Less than 70% of your total funds are off-chain in payment channels.

The numbers 40 and 0.7 are chosen completely arbitrarily here because we cannot make any recommendations that are valid for everyone about how many channels you should have open and what percentage of your funds should be off-chain. The autopilot in lnd will not take into account on-chain fees. In other words, it will not delay opening channels to a time period when fees are low. To reduce fees, you can manually open channels during a time period when fees are low, e.g., during the weekend. The autopilot will make channel recommendations whenever the conditions are met and will immediately try to open a channel by using the appropriate current fees. According to the preceding configuration file, the channels will be between 5 mBTC (`minchansize` = 500,000 satoshi) and 50 mBTC (`maxchansize` = 5,000,000 satoshi) in size. As is common, the amounts in the configuration file are enumerated in satoshi. Currently, channels below 1 mBTC are not very useful, and we do not recommend you open channels that are too small and below this amount. With the wider adoption of multipart payments, smaller channels are less of a burden. But for the time being, this is our recommendation.

The c-lightning plug-in, which was originally written by René Pickhardt (a coauthor of this book), works very differently in comparison with the lnd autopilot. First, it differs in the algorithms used to make the recommendations. We will not cover this here. Secondly, it differs in its user interface. You will need to download the *autopilot plug-in* from the c-lightning plug-in repository and activate it.

NOTE

To activate a plug-in in c-lightning, place it into the *~/.lightning/plugins* directory, ensure that it's executable (e.g., `chmod +x ~/.lightning/plugins/autopilot.py`), then restart lightningd.

Alternatively, if you don't want a plug-in to automatically activate when you start lightningd, you can place it in a different directory and manually activate it with the plugin argument to lightningd:

```
lightningd --plugin=~/lightning-plugins/autopilot.py
```

The autopilot in c-lightning is controlled via three configuration values that can be set in the config file or as command-line arguments when you start lightningd:

```
[c-lightning-autopilot]
autopilot-percent=75
autopilot-num-channels=10
autopilot-min-channel-size-msat=100000000msat
```

These values are the actual default config, and you do not need to set them at all.

The autopilot will not automatically run in the background like in lnd. Instead, you have to start a run specifically with `lightning-cli autopilot-run-once` if you want the autopilot to open the recommended channels. But if you want it to just provide you with recommendations, from which you can handpick the nodes, you can append the optional `dryrun` argument.

A key difference between the lnd and the c-lightning autopilots is that the c-lightning autopilot will also make a recommendation for the channel size. For example, if the autopilot recommends opening a channel with a small node that only has small channels, it will not recommend opening a large channel. However, if it opens a channel with a well-connected node that also has many large channels, it will probably recommend a larger channel size.

As you can see, the c-lightning autopilot is not as automatic as lnd's, but it gives you a little bit more control. These differences reflect personal preferences and could actually be the deciding factor for you to choose one implementation over the other.

Keep in mind that current autopilots will primarily use public information from the gossip protocol about the current topology of the Lightning Network. It is obvious that your personal requirements for channels can only be reflected to a certain degree. More advanced autopilots would use historical and usage information that your node has gathered when running in the past, including information about routing successes, who you have paid in the past, and who paid you. In the future, such improved autopilots might also use this collected data to make recommendations on closing channels and reallocating funds.

Overall, at the time of writing of this book, be cautious not to depend or rely too heavily on autopilots.

Getting Inbound Liquidity

In the current design of the Lightning Network, it is more typical for users to obtain outbound liquidity *before* obtaining inbound liquidity. They will do so by opening a channel with another node, and more often they'll be able to spend bitcoin before they can receive it. There are three typical ways of getting inbound liquidity:

- Open a channel with outbound liquidity and then spend some of those funds. Now the balance is on the other end of the channel, which means that you can receive payments.
- Ask someone to open a channel to your node. Offer to reciprocate, so that both of your nodes become better connected and balanced.
- Use a submarine swap (e.g., Loop In) to exchange on-chain BTC for an inbound channel to your node.
- Pay a third-party service to open a channel with you. Several such services exist. Some charge a

fee to provide liquidity, some are free.

Here is a list of currently available liquidity providers that will open a channel to your node for a fee:

- Bitrefill's Thor service
- Lightning To Me
- LNBig
- Lightning Conductor

Creating inbound liquidity is challenging from both practical and user experience perspectives. Inbound liquidity does not happen automatically, so you have to find ways to build it for your node. This asymmetry of payment channels is also not intuitive. In most other payment systems, you get paid first (inbound) before you pay others (outbound).

The challenge of creating inbound liquidity is most noticeable if you are a merchant or sell your services for Lightning payments. In that case, you need to be vigilant to ensure that you have enough inbound liquidity to be able to continue to receive payments. What if there is a surge of buyers on your store, but they can't actually pay you because there is no more inbound capacity?

In the future, these challenges can be partially mitigated by the implementation of dual-funded channels, which are funded from both sides and offer balanced inbound and outbound capacity. The burden could also be mitigated by more sophisticated autopilot software, which could request and pay for inbound capacity as needed.

Ultimately, Lightning users need to be strategic and proactive about channel management to ensure that sufficient inbound capacity is available to meet their needs.

Closing Channels

As discussed earlier in the book, a *mutual close* is the preferred way of closing a channel. However, there are instances where a *force close* is necessary.

Some examples:

- Your channel partner is offline and cannot be contacted to initiate a mutual close.
- Your channel partner is online, but is not responding to requests to initiate a mutual close.
- Your channel partner is online and your nodes are negotiating a mutual close, but they become stuck and cannot reach a resolution.

Rebalancing Channels

In the course of transacting and routing payments on Lightning, the combination of inbound and outbound capacities can become unbalanced.

For example, if one of your channel partners is frequently routing payments through your node, you will exhaust the inbound capacity on that channel, while also exhausting the outbound capacity on the outgoing channels. Once that happens, you can no longer route payments through that route.

There are many ways to rebalance channels, each with different advantages and disadvantages. One way is to use a submarine swap (e.g., Loop Out), as described previously in this chapter. Another way to rebalance is to simply wait for routed payments that flow in the opposite direction. If your node is well connected, when a specific route becomes exhausted in one direction, the same route becomes available in the opposite direction. Other nodes may "discover" that route in the opposite direction and start using it as part of their payment path, thereby rebalancing the funds again.

A third way to rebalance channels is to purposefully create a *circular route* that sends a payment from your node back to your node, via the Lightning Network. By sending a payment out on a channel with large local capacity and arranging the path so that it returns to your node on a channel with large remote capacity, both of those channels will become more balanced. An example of a circular route rebalancing strategy can be seen in Circular route rebalancing.

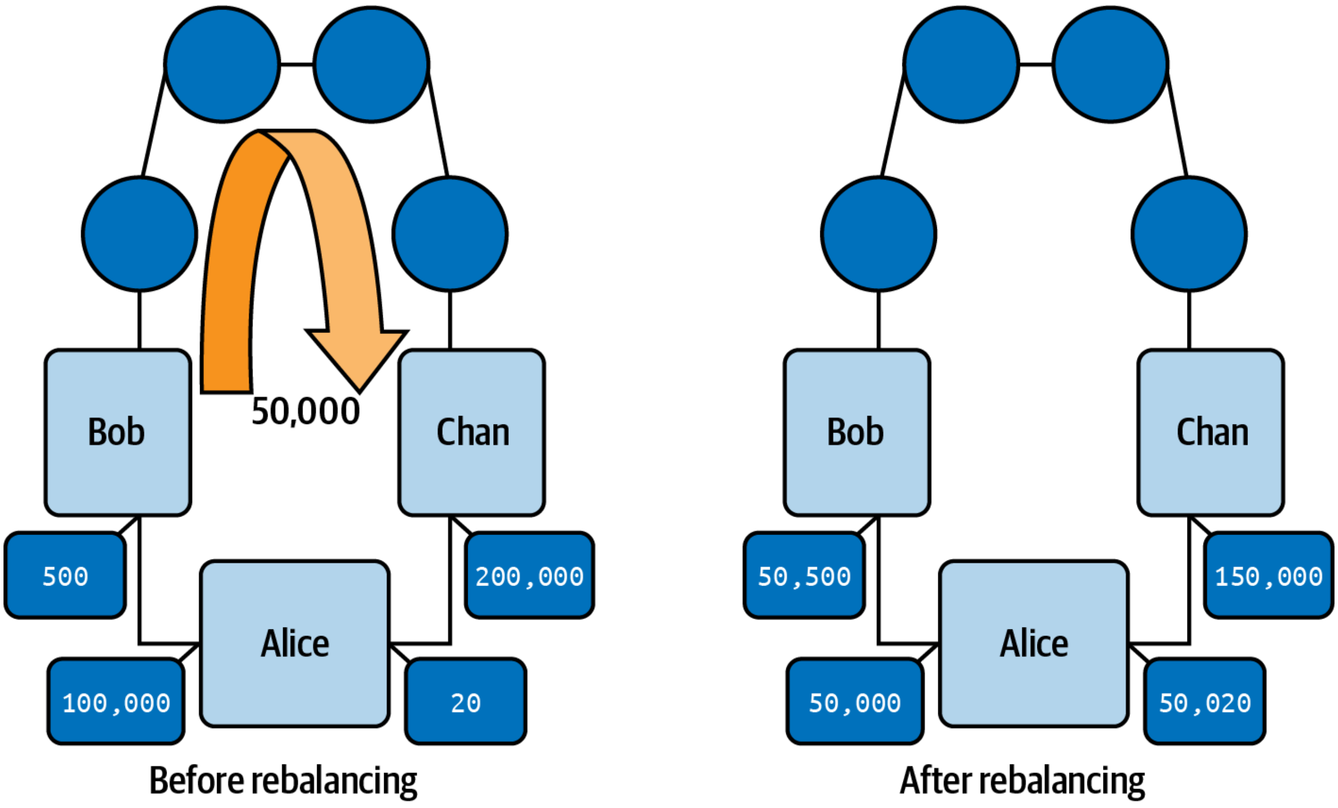


Figure 20. Circular route rebalancing

Circular rebalancing is supported by most Lightning node implementations and can be done on the command line or via one of the web management interfaces such as Ride The Lightning (see Ride The Lightning).

Channel rebalancing is a complex issue that is the subject of active research and covered in more detail in Rebalancing Channels.

Routing Fees

Running a Lightning node allows you to earn fees by routing payments across your channels. Routing fees are generally not a significant source of income and dwarfed by the cost of operating a node. For example, on a relatively busy node that routes a dozen payments a day, the fees amount to no more than 2,000 satoshis.

Nodes compete for routing fees by setting their desired fee rate on each channel. Routing fees are set by two parameters on each channel: a fixed *base fee* that is charged for any payment and an additional variable *fee rate* that is proportional to the payment amount.

When sending a Lightning payment, a node will select a path so as to minimize fees, minimize hops, or both. As a result, a routing fee market emerges from these interactions. There are currently many nodes that charge very low or no fees for routing, creating downward pressure on the routing fee market.

If you make no choices, your Lightning node will set a default base fee and fee rate for each new channel. The default values depend on the node implementation you use. The base fee is set in the unit of *millisatoshi* (thousandths of a satoshi). The proportional fee rate is set in the unit of *millionths* and is applied to the payment amount. The unit of millionths is often abbreviated with *ppm* (parts per million). For example, a base fee of 1,000 (millisatoshi) and a fee rate of 1,000 ppm (millionths) would result in the following charges for a 100,000 satoshi payment:

```
\begin{equation}
\begin{aligned}
P &= 100,000 \text{ satoshi} \\
F_{base} &= 1,000 \text{ millisatoshi} = 1 \text{ satoshi} \\
F_{rate} &= 1,000 \text{ ppm} = 1,000/1,000,000 = 1/1,000 = \text{0.001} = 0.1\% \\
F_{total} &= F_{base} + ( P * F_{rate} ) \\
 \Rightarrow  F_{total} &= 1 \text{ satoshi} + ( 100,000/1,000 ) \text{ satoshi} \\
 \Rightarrow  F_{total} &= 1 \text{ satoshi} + 100 \text{ satoshi} = 101 \text{
satoshi} \\
\end{aligned}
\end{equation}
```

Broadly speaking, you can take one of two approaches to routing fees. You can route lots of payments with low fees, making up for low fees by high volume. Alternatively, you can choose to charge higher fees. If you choose to set higher fees, your node will be selected only when other cheaper routes don’t exist. Therefore, you will route less frequently but earn more per successful routing.

For most nodes, it is usually best to use the default routing fee values. This way, your node is competing on a mostly level playing field with other nodes that use the default values.

You can also use the routing fee settings to rebalance channels. If most of your channels have the default fees but you want to rebalance a particular channel, just decrease the fees on that specific channel to zero or to very low rates. Then sit back and wait for someone to route a payment over your "cheap" route and rebalance your channels for you as a side effect.

Node Management

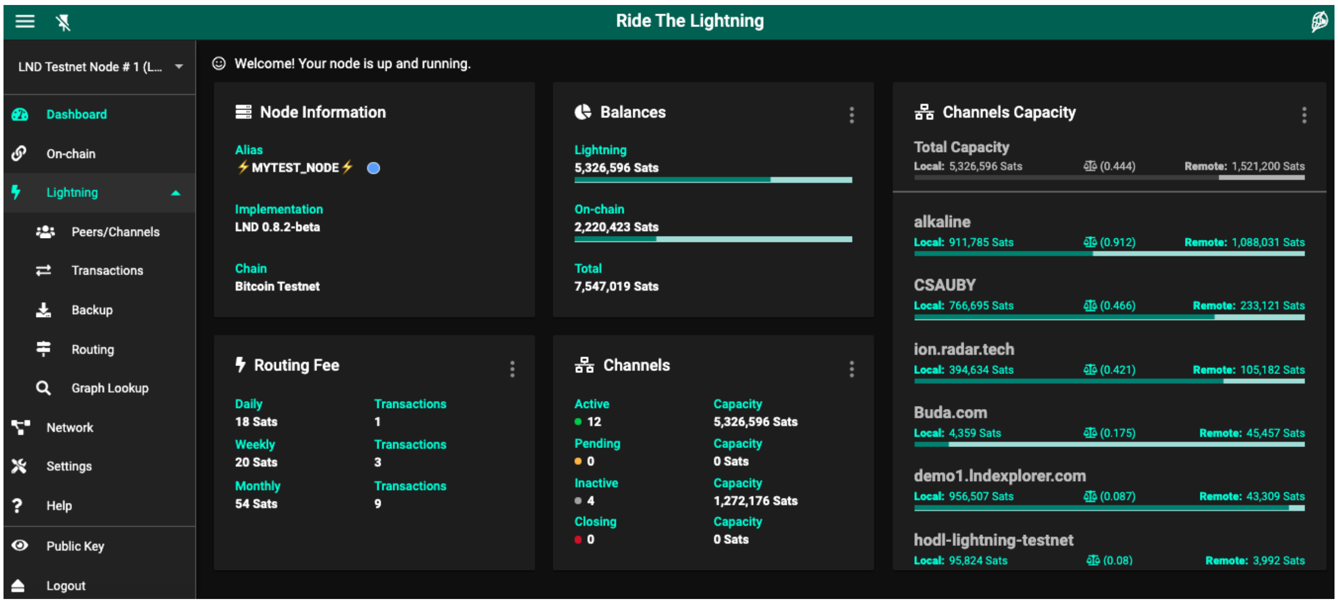
Managing your Lightning node on the command line is obviously not easy. It gives you the full flexibility of the node's API and the ability to write your own custom scripts to satisfy your personal requirements. But if you don't want to deal with the complexity of the command line and only need some basic node management capabilities, you should consider installing a web-based user interface that makes node management easier.

There are a number of competing projects that offer web-based Lightning node management. Some of the most popular ones are described in the following section.

Ride The Lightning

Ride The Lightning (RTL) is a graphical web user interface to help users manage Lightning node operations for the three main Lightning node implementations (LND, `c-lightning`, and Eclair). RTL is an open source project developed by Shahana Farooqui and many other contributors. You can find the RTL software on GitHub.

Example Ride The Lightning (RTL) web interface shows an example screenshot of RTL's web interface, as provided on the project repository.



Welcome! Your node is up and running.

Node Information
Alias: ⚡MYTEST_NODE⚡
Implementation: LND 0.8.2-beta
Chain: Bitcoin Testnet

Balances
Lightning: 5,326,596 Sats
On-chain: 2,220,423 Sats
Total: 7,547,019 Sats

Channels Capacity
Total Capacity — Local: 5,326,596 Sats (0.444) Remote: 1,521,200 Sats
alkaline — Local: 911,785 Sats (0.912) Remote: 1,088,031 Sats
CSAUBY — Local: 766,695 Sats (0.466) Remote: 233,121 Sats
ion.radar.tech — Local: 394,634 Sats (0.421) Remote: 105,182 Sats
Buda.com — Local: 4,359 Sats (0.175) Remote: 45,457 Sats
demo1.lndexplorer.com — Local: 956,507 Sats (0.087) Remote: 43,309 Sats
hodl-lightning-testnet — Local: 95,824 Sats (0.08) Remote: 3,992 Sats

Routing Fee
Daily: 18 Sats — Transactions: 1
Weekly: 20 Sats — Transactions: 3
Monthly: 54 Sats — Transactions: 9

Channels
Active: ● 12 — Capacity: 5,326,596 Sats
Pending: ● 0 — Capacity: 0 Sats
Inactive: ● 4 — Capacity: 1,272,176 Sats
Closing: ● 0 — Capacity: 0 Sats

Figure 21. Example Ride The Lightning (RTL) web interface

lndmon

Lightning Labs, the makers of LND, provide a web-based graphical user interface called lndmon to monitor the various metrics of an LND Lightning node. lndmon only works with LND nodes. It is a read-only interface for monitoring and as such does not allow you to actively manage the node. It cannot open channels or make payments. Find lndmon on GitHub.

ThunderHub

ThunderHub is a very aesthetically pleasing web-based graphical user interface similar to RTL but exclusive to LND. It can be used to make payments, rebalance channels, and manage the node through a variety of features.

Conclusion

As you maintain your node and gain experiences, you will learn a lot about the Lightning Network. Being a node operator is a challenging but rewarding task. Mastering these skills will allow you to contribute to the growth and development of this technology and the Lightning Network itself. You will furthermore gain the ability to send and receive Lightning payments with the greatest degree of control and ease. You will play a central role in the network's infrastructure and not just be a participant on the edges.

The Lightning Network in Detail

A detailed explanation of all the components of the Lightning Network and how they work. This part is highly technical and expects the reader to have some programming and computer science experience.

Lightning Network Architecture

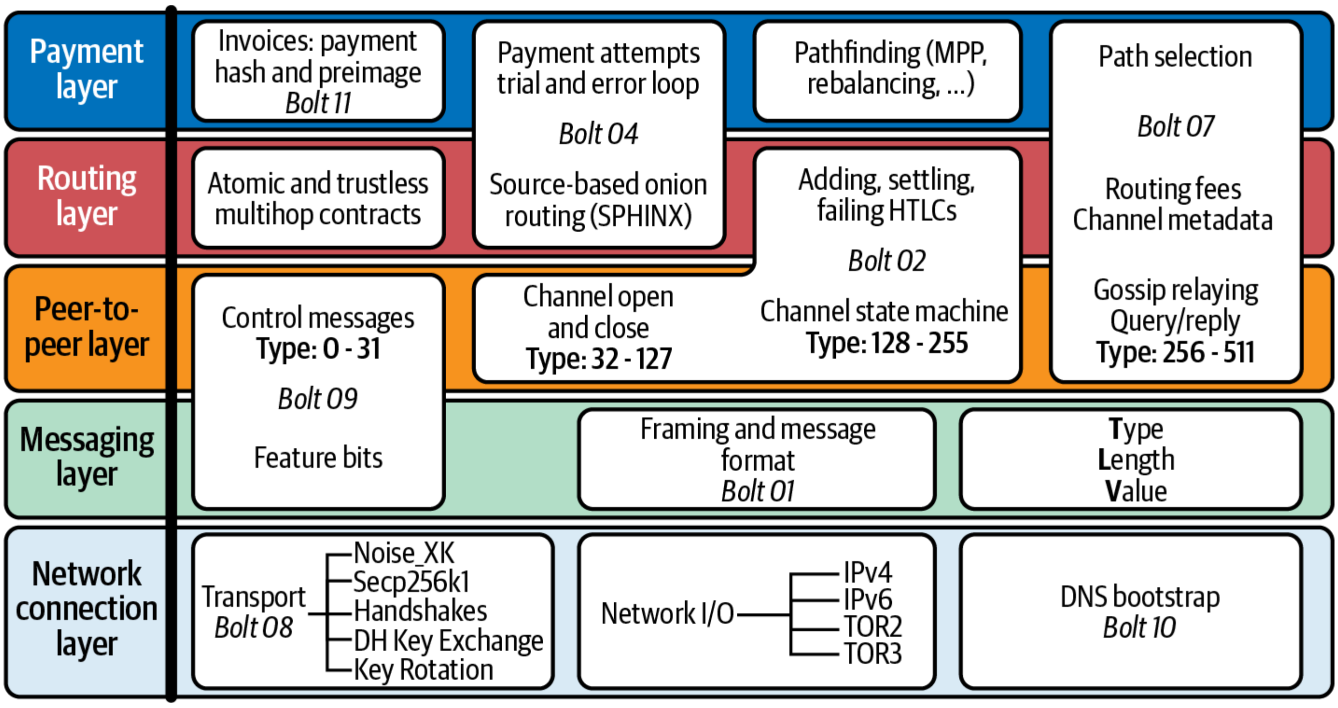
In the first part of this book we introduced the main concepts of the Lightning Network and worked through a comprehensive example of routing a payment and setting up the tools we can use to explore further. In the second part of the book we will explore the Lightning Network in a lot more technical detail, dissecting each of the building blocks.

In this section we will outline the components of the Lightning Network in more detail and provide a "big picture" perspective to guide you through the following chapters.

The Lightning Network Protocol Suite

The Lightning Network is composed of a complex collection of protocols that run on top of the internet. We can broadly classify these protocols into five distinct layers that make up a *protocol stack*, where each layer builds upon and uses the protocols in the layer below. Also, each protocol layer abstracts the underlying layers and "hides" some of the complexity.

The architecture diagram shown in The Lightning Network protocol suite provides an overview of these layers and their component protocols.



Payment layer
Routing layer
Peer-to-peer layer
Messaging layer
Network connection layer
Invoices: payment hash and preimage Bolt 11
Payment attempts trial and error loop
Bolt 04
Source-based onion routing (SPHINX)
Pathfinding (MPP, rebalancing, ...)
Path selection
Bolt 07
Routing fees Channel metadata
Gossip relaying Query/reply Type: 256 - 511
Atomic and trustless multihop contracts
Adding, settling, failing HTLCs
Bolt 02
Channel state machine Type: 128 - 255
Channel open and close Type: 32 - 127
Control messages Type: 0 - 31
Bolt 09
Feature bits
Framing and message format Bolt 01
Type Length Value
Transport Bolt 08
Noise_XK
Secp256k1
Handshakes
DH Key Exchange
Key Rotation
Network I/O
IPv4
IPv6
TOR2
TOR3
DNS bootstrap Bolt 10

Figure 22. The Lightning Network protocol suite

The five layers of the Lightning Network, from the bottom up, are:

Network connection layer

This contains the protocols that interact directly with the internet core protocols (TCP/IP), overlay protocols (Tor v2/v3), and internet services (DNS). This layer also contains the cryptographic transport protocols that protect Lightning messages.

Messaging layer

This layer contains the protocols that nodes use to negotiate features, format messages, and encode message fields.

Peer-to-peer (P2P) layer

This layer is the primary protocol layer for communication between Lightning nodes and contains all the different messages exchanged between nodes.

Routing layer

This layer contains the protocols used to route payments between nodes, end-to-end and atomically. This layer contains the core functionality of the Lightning Network: routed payments.

Payment layer

The highest layer of the network, which presents a reliable payment interface to applications.

Lightning in Detail

Over the next 10 chapters, we will dissect the protocol suite and examine each component of the Lightning Network in detail.

We spent quite some time trying to decide the best order of presenting this detail. It’s not an easy choice because there is so much interdependence between different components: as you start explaining one, you find that it pulls in quite a few of the other componenents. Instead of a top-down or bottom-up approach, we ended up choosing a more meandering path that starts with the most fundamental building blocks that are unique to the Lightning Network-Payment Channels and moves outward from there. But since that path is not obvious, we will use the Lightning Protocol Suite shown in The Lightning Network protocol suite as a map. In each chapter will focus on one or more related components, and you will see them highlighted in the protocol suite. Kind of like a map marker saying "You are here!"

Here’s what we will cover:

Payment Channels

In this chapter we will look at how payment channels work, in significantly more depth than we saw in the earlier parts of the book. We will look at the structure and Bitcoin Script of the funding and commitment transactions, and the process used by nodes to negotiate each step in the protocol.

Routing on a Network of Payment Channels

Next, we will put together several payment channels in a network and route a payment from one end to the other. In that process we will dive into the hash time-locked contract (HTLC) smart contract and the Bitcoin Script that we use to construct it.

Channel Operation and Payment Forwarding

Putting together the concepts of a simple payment channel and a routed payment using HTLCs, we will now look at how HTLCs are part of each channel’s commitment transaction. We will also look at the protocol for adding, settling, failing, and removing HTLCs from the commitments.

Onion Routing

Next, we will look at how the HTLC information is propagated across the network inside the onion routing protocol. We will look at the mechanism for layered encryption and decryption that gives the Lightning Network some of its privacy characteristics.

Gossip and the Channel Graph

In this chapter we will look at how Lightning nodes find each other and learn about published channels to construct a channel graph that they can use to find paths across the network.

Pathfinding and Payment Delivery

Next, we will see how the information from the gossip protocol is used by each node to build a "map" of the entire network, which it can use to find paths from one point to another to route payments. We'll also look at the exiting innovations in pathfinding, such as multipart payments.

Lightning Payment Requests

A key part of the Lightning Network is payment requests, also known as Lightning invoices. In this chapter we dissect the structure and encoding of an invoice.

Wire Protocol: Framing and Extensibility

Underpinning the Lightning Network is the peer-to-peer protocol that nodes use to exchange messages about the network and about their channels. In this chapter we look at how those messages are constructed and the extension capabilities built into messages with feature bits and Type-Length-Value (TLV) encoding.

Lightning's Encrypted Message Transport

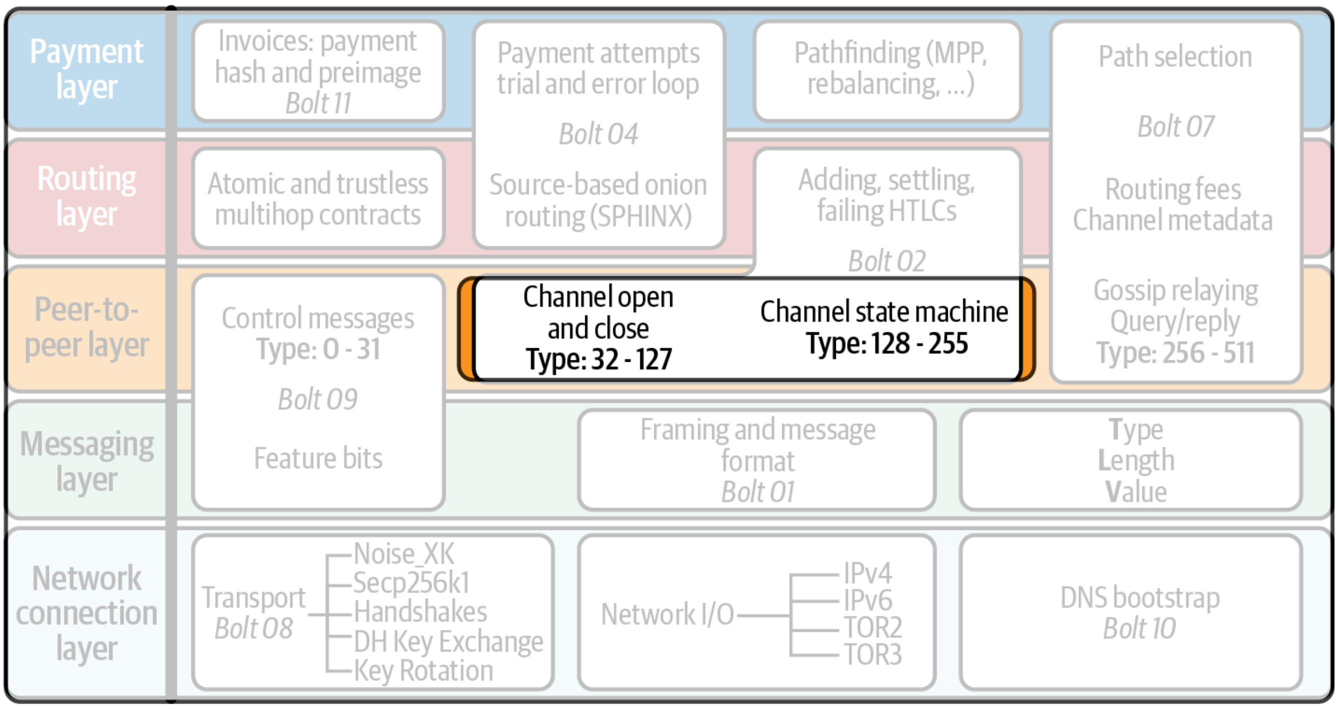
Moving down to the lower-level part of the network, we will look at the underlying encrypted transport system that ensures the secrecy and integrity of all communications between nodes.

Let's dive in!

Payment Channels

In this chapter we will dive into payment channels and see how they are constructed. We will start with Alice's node opening a channel to Bob's node, building on the examples presented in the beginning of this book.

The messages exchanged by Alice and Bob's nodes are defined in "BOLT #2: Peer Protocol for Channel Management". The transactions created by Alice and Bob's nodes are defined in "BOLT #3: Bitcoin Transaction and Script Formats". In this chapter we are focusing on the "Channel open and close" and "Channel state machine" parts of the Lightning protocol architecture, highlighted by an outline in the center (peer-to-peer layer) of Payment channels in the Lightning protocol suite.



Payment layer
Routing layer
Peer-to-peer layer
Messaging layer
Network connection layer
Invoices: payment hash and preimage Bolt 11
Payment attempts trial and error loop
Bolt 04
Source-based onion routing (SPHINX)
Pathfinding (MPP, rebalancing, ...)
Path selection
Bolt 07
Routing fees Channel metadata
Gossip relaying Query/reply Type: 256 - 511
Atomic and trustless multihop contracts
Adding, settling, failing HTLCs
Bolt 02
Channel open and close Type: 32 - 127
Channel state machine Type: 128 - 255
Control messages Type: 0 - 31
Bolt 09
Feature bits
Framing and message format Bolt 01
Type Length Value
Transport Bolt 08
Noise_XK
Secp256k1
Handshakes
DH Key Exchange
Key Rotation
Network I/O
IPv4
IPv6
TOR2
TOR3
DNS bootstrap Bolt 10

Figure 23. Payment channels in the Lightning protocol suite

A Different Way of Using the Bitcoin System

The Lightning Network is often described as a "Layer 2 Bitcoin Protocol," which makes it sound distinct from Bitcoin. Another way to describe Lightning is as a "smarter way to use Bitcoin" or just as an "application on top of Bitcoin." Let's explore that.

Historically, Bitcoin transactions are broadcast to everyone and recorded on the Bitcoin blockchain to be considered valid. As we will see, however, if someone holds a presigned Bitcoin transaction that spends a 2-of-2 multisig output that gives them the exclusive ability to spend that Bitcoin, they effectively own that Bitcoin even if they don't broadcast the transaction.

You can think of the presigned Bitcoin transaction like a postdated check (or cheque), one that can be cashed at any time. Unlike the traditional banking system, however, this transaction is not a "promise" of payment (also known as an IOU), but a verifiable bearer instrument that is equivalent to cash. So long as the bitcoin referenced in the transaction has not already been spent at the time of redemption (or at the time you try to "cash" the check), the Bitcoin system guarantees that this presigned transaction can be broadcast and recorded at any time. This is only true, of course, if this is the only presigned transaction. Within the Lightning Network two or more such presigned transactions exist at the same time; therefore, we need a more sophisticated mechanism to still have the functionality of such a verifiable bearer instrument, as you will also learn in this chapter.

The Lightning Network is simply a different and creative way of using Bitcoin. In the Lightning Network a combination of recorded (on-chain) and presigned but withheld (off-chain) transactions form a "layer" of payments that is a faster, cheaper, and more private way to use Bitcoin. You can see this relationship between on-chain and off-chain Bitcoin transactions in Lightning payment channel made of on-chain and off-chain transactions.

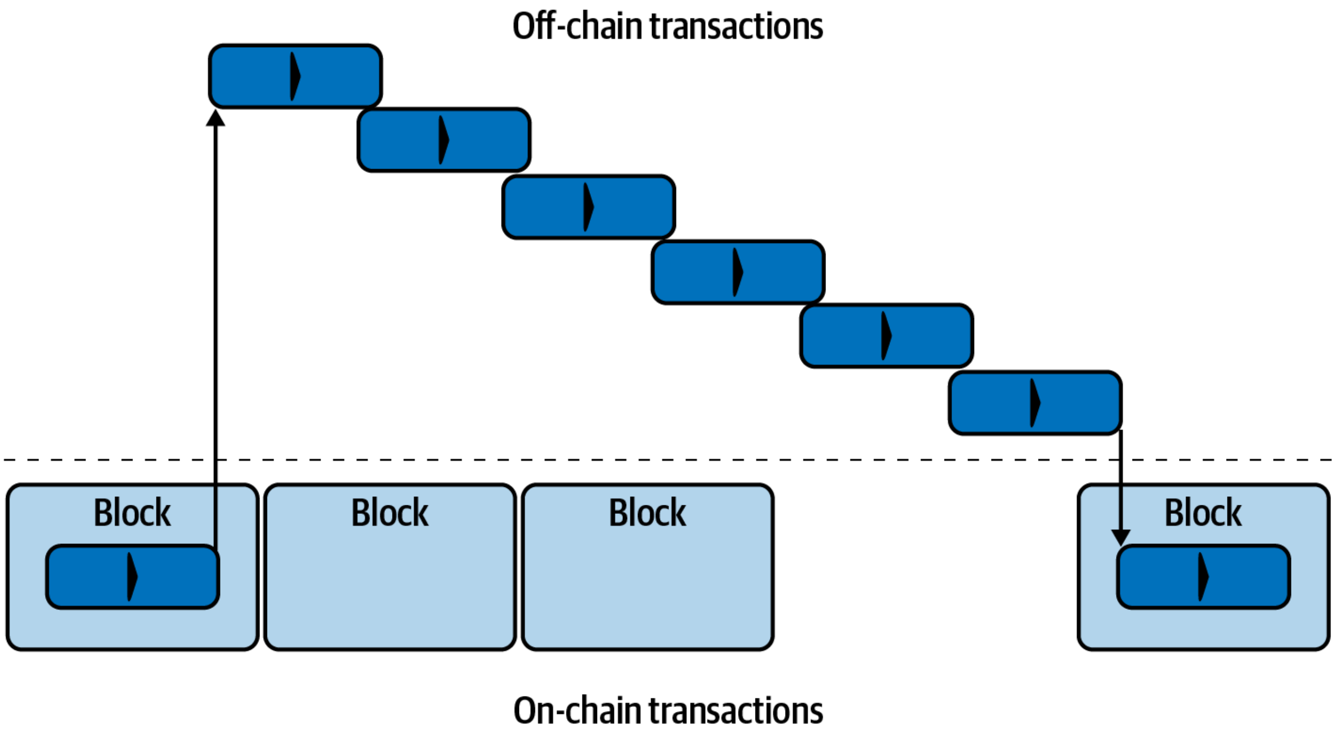


Figure 24. Lightning payment channel made of on-chain and off-chain transactions

Lightning is Bitcoin. It’s just a different way of using the Bitcoin system.

Bitcoin Ownership and Control

Before we understand payment channels, we have to take a small step back and understand how ownership and control work in Bitcoin.

When someone says they "own" bitcoin, they typically mean that they know the private key of a Bitcoin address that has some unspent transaction outputs (see Bitcoin Fundamentals Review). The private key allows them to sign a transaction to spend that bitcoin by transferring it to a different address. In Bitcoin "ownership" of bitcoin can be defined as the *ability to spend* that bitcoin.

Keep in mind that the term "ownership" as used in Bitcoin is distinct from the term "ownership" used in a legal sense. A thief who has the private keys and can spend Bitcoin is a *de facto owner* of that Bitcoin even though they are not a lawful owner.

TIP Bitcoin ownership is only about control of keys and the ability to spend the Bitcoin with those keys. As the popular Bitcoin saying goes: "Your keys, your coins—not your keys, not your coins."

Diversity of (Independent) Ownership and Multisig

Ownership and control of private keys is not always in the hands of one person. That’s where things get interesting and complicated. We know that more than one person can come to know the same private key, either through theft or because the original holder of the key makes a copy and gives it to someone else. Are all these people owners? In a practical sense, they are, because any one of the people who know the private key can spend the bitcoin without the approval of any other.

Bitcoin also has multisignature addresses where multiple private keys are needed to sign before spending (see Multisignature Scripts). From a practical perspective, ownership in a multisignature address depends on the quorum (*K*) and total (*N*) defined in the *K*-of-*N* scheme. A 1-of-10 multisignature scheme would allow any 1 (*K*) of 10 (*N*) signers to spend a bitcoin amount locked in that address. This is similar to the scenario where 10 people have a copy of the same private key and any of them can independently spend it.

Joint Ownership Without Independent Control

There is also the scenario where *no one* has quorum. In a 2-of-2 scheme like that used in the Lightning Network, neither signer can spend the bitcoin without obtaining a signature from the other party. Who owns the bitcoin in that case? No one really has ownership because no one has control. They each own the equivalent of a voting share in the decision, but both votes are needed. A key problem (pun intended) with a 2-of-2 scheme, in both Bitcoin and the law, is what happens if one of the parties is unavailable, or if there is a vote deadlock and any one party refuses to cooperate.

Preventing "Locked" and Un-Spendable Bitcoin

If one of the two signers of a 2-of-2 multisig cannot or will not sign, the funds become un-spendable. Not only can this scenario occur accidentally (loss of keys), but it can be used as a form of blackmail by either party: "I won’t sign unless you pay me a part of the funds."

Payment channels in Lightning are based on a 2-of-2 multisig address, with the two channel partners as signers in the multisig. At this time, channels are funded only by one of the two channel partners: when you choose to "open" a channel, you deposit funds into the 2-of-2 multisig address with a transaction. Once that transaction is mined and the funds are in the multisig, you can’t get them back without cooperation from your channel partner, because you need their signature (also) to spend the bitcoin.

In the next section, as we look at how to open (create) a Lightning channel, we will see how we can prevent loss of funds or any blackmail scenario between the two partners by implementing a fairness protocol for the channel construction with the help of presigned transactions that spend the multisig output in a way that gives the peers in the channel exclusive ability to spend one of the outputs which encodes the amount of bitcoin they own in the channel.

Constructing a Payment Channel

In What Is a Payment Channel?, we described payment channels as a *financial relationship* between two Lightning nodes, which is established by funding a 2-of-2 multisignature address from the two channel partners.

Let’s assume that Alice wants to construct a payment channel allowing her to connect to Bob’s store directly. First, the two nodes (Alice’s and Bob’s) have to establish an internet connection to each other, so that they can negotiate a payment channel.

Node Private and Public Keys

Every node on the Lightning Network is identified by a *node public key*. The public key uniquely identifies the specific node and is usually presented as a hexadecimal encoding. For example, René Pickhardt currently runs a Lightning Node (ln.rene-pickhardt.de) that is identified by the following node public key:

```
02a1cebfacb2674143b5ad0df3c22c609e935f7bc0ebe801f37b8e9023d45ea7b8
```

Each node generates a root private key when first initialized. The private key is kept private at all times (never shared) and securely stored in the node’s wallet. From that private key, the node derives a public key that is the node identifier and shared with the network. Since the key space is enormous, as long as each node generates the private key randomly, it will have a unique public key, which can therefore uniquely identify it on the network.

Node Network Address

Additionally, every node also advertises a network address where it can be reached, in one of several possible formats:

TCP/IP
: An IPv4 or IPv6 address and TCP port number

TCP/Tor
: A Tor "onion" address and TCP port number

The network address identifier is written as Address:Port, which is consistent with international standards for network identifiers, as used, for example, on the web.

For example, René’s node with node public key 02a1ceb...45ea7b8 currently advertises its network address as the TCP/IP address:

```
172.16.235.20:9735
```

TIP The default TCP port for the Lightning Network is 9735, but a node can choose to listen on any TCP port.

Node Identifiers

Together, the node public key and network address are written in the following format, separated by an @ sign, as *NodeID@Address:Port*.

So the full identifier for René’s node would be:

```
02a1cebfacb2674143b5ad0df3c22c609e935f7bc0ebe801f37b8e9023d45ea7b8@172.16.235.20:9735
```

TIP The alias of René’s node is ln.rene-pickhardt.de; however, this name exists just for better readability. Every node operator can announce whatever alias they want, and there is no mechanism that prevents node operators from selecting an alias that is already being used. Thus to refer to a node, one must use the *NodeID@Address:Port* schema.

The preceding identifier is often encoded in a QR code, making it easier for users to scan if they want to connect their own node to the specific node identified by that address.

Much like Bitcoin nodes, Lightning nodes advertise their presence on the Lightning Network by "gossiping" their node public key and network address. That way, other nodes can find them and keep an inventory (database) of all the known nodes that they can connect to and exchange the messages that are defined in the Lightning P2P message protocol.

Connecting Nodes as Direct Peers

In order for Alice’s node to connect to Bob’s node, she will need Bob’s node public key, or the full address containing the public key, IP or Tor address, and port. Because Bob runs a store, Bob’s node address can be retrieved from an invoice or a store payment page on the web. Alice can scan a QR code that contains the address and instruct her node to connect to Bob’s node.

Once Alice has connected to Bob’s node, their nodes are now directly connected peers.

TIP To open a payment channel, two nodes must first be connected as direct peers by opening a connection over the internet (or Tor).

Constructing the Channel

Now that Alice’s and Bob’s Lightning nodes are connected, they can begin the process of constructing a payment channel. In this section we will review the communications between their nodes, known as the *Lightning Peer Protocol for Channel Management*, and the cryptographic protocol that they use to build Bitcoin transactions.

TIP We describe two different protocols in this scenario. First, there is a *message protocol*, which establishes how the Lightning nodes communicate over the internet and what messages they exchange with each other. Second, there is the *cryptographic protocol*, which establishes how the two nodes construct and sign Bitcoin transactions.

Peer Protocol for Channel Management

The Lightning Peer Protocol for Channel Management is defined in BOLT #2: Peer Protocol for Channel Management. In this chapter we will be reviewing the "Channel Establishment" and

"Channel Closing" sections of BOLT #2 in more detail.

Channel Establishment Message Flow

Channel establishment is achieved by the exchange of six messages between Alice and Bob's nodes (three from each peer): open_channel, accept_channel, funding_created, funding_signed, funding_locked, and funding_locked. The six messages are shown as a time-sequence diagram in The channel establishment message flow.

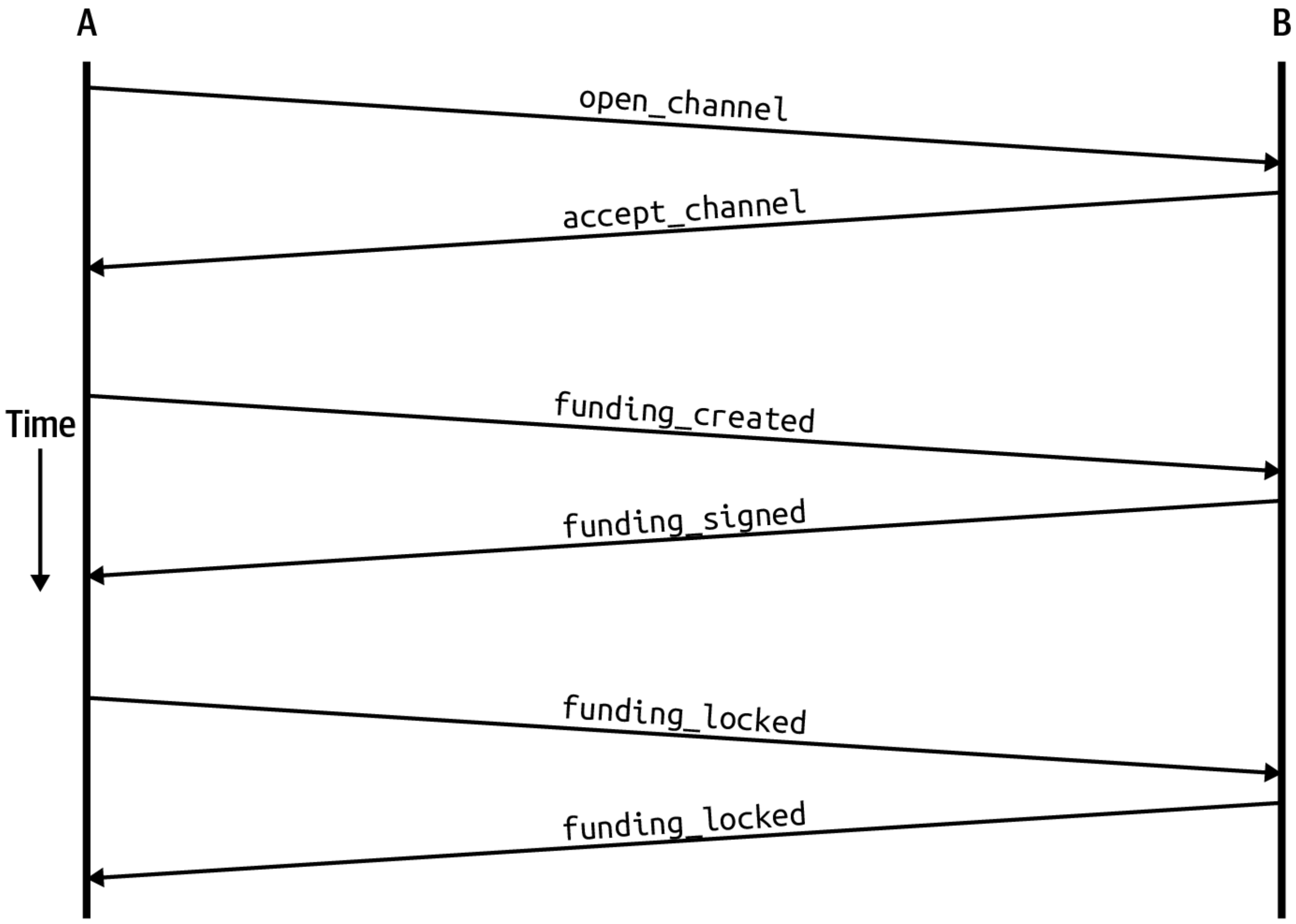


Figure 25. The channel establishment message flow

In The channel establishment message flow, Alice and Bob's nodes are represented by the vertical lines "A" and "B" on either side of the diagram. A time-sequence diagram like this shows time flowing downward, and messages flowing from one side to the other between the two communication peers. The lines are sloped down to represent the elapsed time needed to transmit each message, and the direction of the message is shown by an arrow at the end of each line.

The channel establishment involves three parts. First, the two peers communicate their capabilities and expectations, with Alice initiating a request through open_channel and Bob accepting the channel request through accept_channel.

Second, Alice constructs the funding and refund transactions (as we will see later in this section) and sends funding_created to Bob. Another name for the "refund" transaction is a "commitment" transaction, as it commits to the current distribution of balances in the channel. Bob responds by sending back the necessary signatures with funding_signed. This interaction is the basis for the *cryptographic protocol* to secure the channel and prevent theft. Alice will now broadcast the funding transaction (on-chain) to establish and anchor the payment channel. The transaction will need to be confirmed on the Bitcoin blockchain.

TIP The name of the funding_signed message can be a bit confusing. This message does not contain a signature for the funding transaction, but rather it contains Bob's signature for the refund transaction that allows Alice to claim her bitcoin back from the multisig.

Once the transaction has sufficient confirmations (as defined by the `minimum_depth` field in the `accept_channel` message), Alice and Bob exchange funding_locked messages, and the channel enters normal operating mode.

The open_channel message

Alice's node requests a payment channel with Bob's node by sending an open_channel message. The message contains information about Alice's *expectations* for the channel setup, which Bob may accept or decline.

The structure of the open_channel message (taken from BOLT #2) is shown in The `open_channel` message.

Example 1. The `open_channel` message

```
[chain_hash:chain_hash]
[32*byte:temporary_channel_id]
[u64:funding_satoshis]
[u64:push_msat]
[u64:dust_limit_satoshis]
[u64:max_htlc_value_in_flight_msat]
[u64:channel_reserve_satoshis]
[u64:htlc_minimum_msat]
[u32:feerate_per_kw]
[u16:to_self_delay]
[u16:max_accepted_htlcs]
[point:funding_pubkey]
[point:revocation_basepoint]
[point:payment_basepoint]
[point:delayed_payment_basepoint]
[point:htlc_basepoint]
[point:first_per_commitment_point]
[byte:channel_flags]
[open_channel_tlvs:tlvs]
```

The fields contained in this message specify the channel parameters that Alice wants, as well as various configuration settings from Alice's nodes that reflect the security expectations for the operation of the channel.

Some of the channel construction parameters are listed here:

chain_hash

This identifies which blockchain (e.g., Bitcoin mainnet) will be used for this channel. It is usually the hash of the genesis block of that blockchain.

funding_satoshis

The amount Alice will use to fund the channel, which is the total channel capacity.

channel_reserve_satoshis

The minimum balance, in satoshis, that is reserved on each side of a channel. We will come back to this when we talk about penalties.

push_msat

An optional amount that Alice will immediately "push" to Bob as a payment upon channel funding. *Setting this value to anything but 0 means effectively gifting money to your channel partner and should be used with caution.*

to_self_delay

A very important security parameter for the protocol. The value in the `open_channel` message is used in the responder's commitment transaction, and the `accept_channel` in the initiator's. This asymmetry exists to allow each side to express how long the other side needs to wait to unilaterally claim the funds in a commitment transaction. If Bob at any time unilaterally closes the channel against the will of Alice, he commits to not accessing his own funds for the delay defined here. The higher this value, the more security Alice has, but the longer Bob might have his funds locked.

funding_pubkey

The public key that Alice will contribute to the 2-of-2 multisig that anchors this channel.

X_basepoint

Master keys, used to derive child keys for various parts of the commitment, revocation, routed payment (HTLCs), and closing transactions. These will be used and explained in subsequent chapters.

TIP

If you want to understand the other fields and Lightning peer protocol messages that we do not discuss in this book, we suggest you look them up in the BOLT specifications. These messages and fields are important, but cannot be covered in enough detail in the scope of this book. We want you to understand the fundamental principles well enough that you can fill in the details by reading the actual protocol specification (BOLTs).

The accept_channel message

In response to Alice's open_channel message, Bob sends back the accept_channel message shown in The `accept_channel` message.

Example 2. The `accept_channel` *message*

```
[32*byte:temporary_channel_id]
[u64:dust_limit_satoshis]
[u64:max_htlc_value_in_flight_msat]
[u64:channel_reserve_satoshis]
[u64:htlc_minimum_msat]
[u32:minimum_depth]
[u16:to_self_delay]
[u16:max_accepted_htlcs]
[point:funding_pubkey]
[point:revocation_basepoint]
[point:payment_basepoint]
[point:delayed_payment_basepoint]
[point:htlc_basepoint]
[point:first_per_commitment_point]
[accept_channel_tlvs:tlvs]
```

As you can see, this is similar to the open_channel message and contains Bob's node expectations and configuration values.

The two most important fields in accept_channel that Alice will use to construct the payment channel are:

funding_pubkey

The public key Bob's node contributes for the 2-of-2 multisig address that anchors the channel.

minimum_depth

The number of confirmations that Bob's node expects for the funding transaction before it considers the channel "open" and ready to use.

The Funding Transaction

Once Alice's node receives Bob's accept_channel message, it has the information necessary to construct the *funding transaction* that anchors the channel to the Bitcoin blockchain. As we discussed in earlier chapters, a Lightning payment channel is anchored by a 2-of-2 multisignature address. First, we need to generate that multisignature address to allow us to construct the funding transaction (and the refund transaction as described subsequently).

Generating a Multisignature Address

The funding transaction sends some amount of bitcoin (funding_satoshis from the open_channel message) to a 2-of-2 multisignature output that is constructed from Alice and Bob's funding_pubkey public keys.

Alice's node constructs a multisignature script as shown here:

A 2-of-2 multisig script with Alice and Bob's funding_pubkey values

```
2 <_`Alice_funding_pubkey`_> <_`Bob_funding_pubkey`_> 2 CHECKMULTISIG
```

Note that, in practice, the funding keys are deterministically *sorted* (using lexicographical order of the serialized compressed form of the public keys) before being placed in the witness script. By agreeing to this sorted order ahead of time, we ensure that both parties will construct an identical funding transaction output, which is signed by the exchanged commitment transaction signature.

This script is encoded as a Pay-to-Witness-Script-Hash (P2WSH) Bitcoin address, which looks something like this:

```
bc1q89ju02heg32yrqdrnqghe6132wek25p6sv6e564znvrvez7tq5zqt4dn02
```

Constructing the Funding Transaction

Alice's node can now construct a funding transaction, sending the amount agreed on with Bob (`funding_satoshis`) to the 2-of-2 multisig address. Let's assume that funding_satoshis was 140,000 and Alice is spending a 200,000 satoshi output and creating 60,000 satoshi change. The transaction will look something like Figure 7-4.

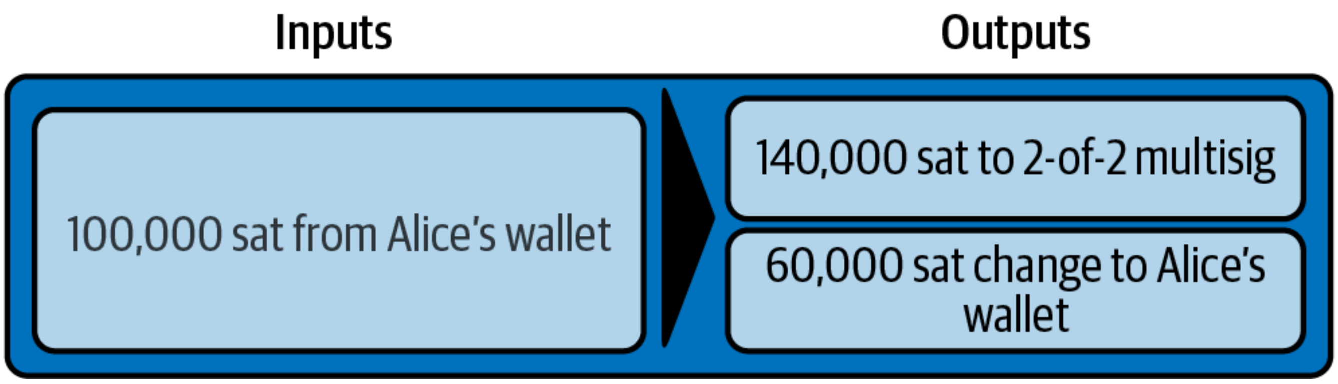

Inputs
Outputs
100,000 sat from Alice's wallet
140,000 sat to 2-of-2 multisig
60,000 sat change to Alice's wallet

Figure 26. Alice constructs the funding transaction

Alice *does not broadcast* this transaction because doing so would put her 140,000 satoshi at risk. Once spent to the 2-of-2 multisig, there is no way for Alice to recover her money without Bob's signature.

Dual-Funded Payment Channels

In the current implementation of Lightning, channels are funded only by the node initiating the channel (Alice in our example). Dual-funded channels have been proposed, but not yet implemented. In a dual-funded channel, both Alice and Bob would contribute inputs to the funding transaction. Dual-funded channels require a slightly more complicated message flow and cryptographic protocol, so they have not been implemented yet but are planned for a future update to the Lightning BOLTs. The `c-lightning` implementation includes an experimental version of a variant on dual-funded channels.

Holding Signed Transactions Without Broadcasting

An important Bitcoin feature that makes Lightning possible is the ability to construct and sign transactions, but not broadcast them. The transaction is *valid* in every way, but until it is broadcast and confirmed on the Bitcoin blockchain it is not recognized and its outputs are not spendable because they have not been created on the blockchain. We will use this capability many times in the Lightning Network, and Alice's node uses the capability when constructing the funding transaction: holding it and not broadcasting it yet.

Refund Before Funding

To prevent loss of funds, Alice cannot put her bitcoin into a 2-of-2 until she has a way to get a refund if things go wrong. Essentially, she must plan the "exit" from the channel before she enters into this arrangement.

Consider the legal construct of a prenuptial agreement, also known as a "prenup." When two people enter into a marriage their money is bound together by law (depending on jurisdiction). Prior to entering into the marriage, they can sign an agreement that specifies how to separate their assets if they dissolve their marriage through divorce.

We can create a similar agreement in Bitcoin. For example, we can create a refund transaction, which functions like a prenup, allowing the parties decide how the funds in their channel will be divided before their funds are actually locked into the multisignature funding address.

Constructing the Presigned Refund Transaction

Alice will construct the refund transaction immediately after constructing (but not broadcasting) the funding transaction. The refund transaction spends the 2-of-2 multisig back to Alice's wallet. We call this refund transaction a *commitment transaction* because it commits both channel partners to distributing the channel balance fairly. Since Alice funded the channel on her own, she gets the entire balance, and both Alice and Bob commit to refunding Alice with this transaction.

In practice, it is a bit more complicated as we will see in subsequent chapters, but for now let's keep things simple and assume it looks like Figure 7-5.

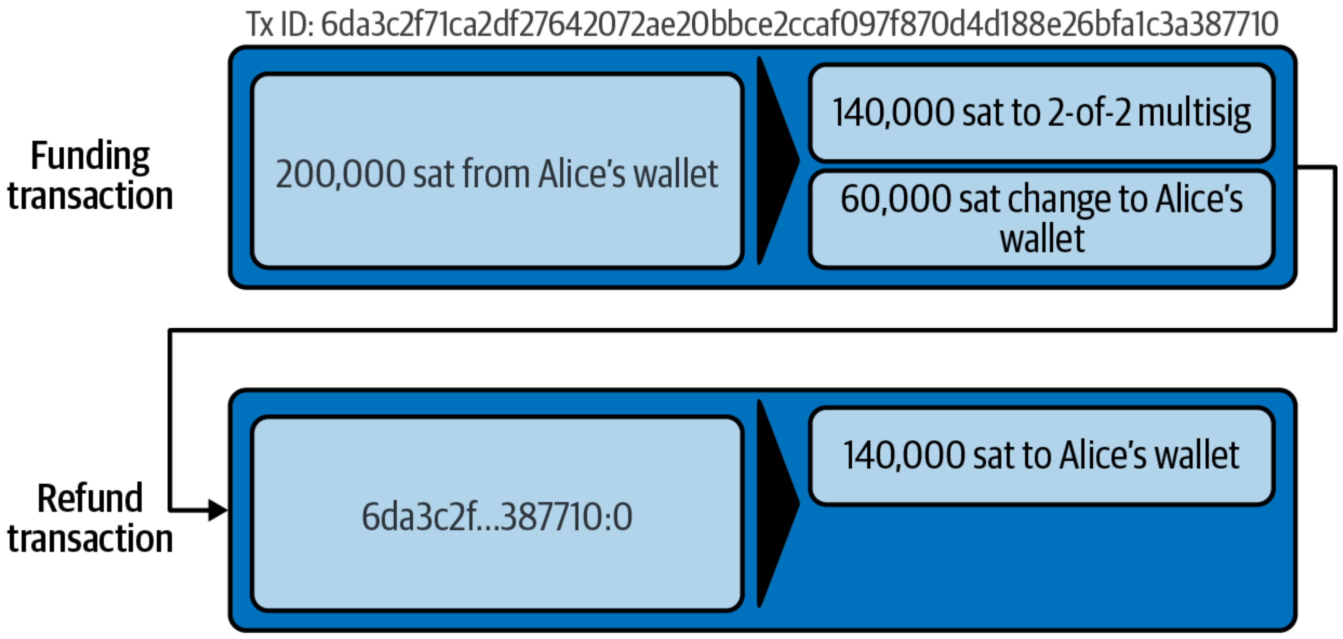

Tx ID: 6da3c2f71ca2df27642072ae20bbce2ccaf097f870d4d188e26bfa1c3a387710
Funding transaction
200,000 sat from Alice's wallet
140,000 sat to 2-of-2 multisig
60,000 sat change to Alice's wallet
Refund transaction
6da3c2f...387710:0
140,000 sat to Alice's wallet

Figure 27. Alice also constructs the refund transaction

Later in this chapter we will see how more commitment transactions can be made to distribute the balance of the channel in different amounts.

Chaining Transactions Without Broadcasting

So now, Alice has constructed the two transactions shown in Alice also constructs the refund transaction. But you might be wondering how this is possible. Alice hasn't broadcast the funding transaction to the Bitcoin blockchain. As far as everyone on the network is concerned, that transaction doesn't exist. The refund transaction is constructed so as to *spend* one of the outputs of the funding transaction, even though that output doesn't exist yet either. How can you spend an output that hasn't been confirmed on the Bitcoin blockchain?

The refund transaction is not yet a valid transaction. For it to become a valid transaction two things must happen:

- The funding transaction must be broadcast to the Bitcoin network. (To ensure the security of the Lightning Network, we will also require it to be confirmed by the Bitcoin blockchain, though this is not strictly necessary to chain transactions.)
- The refund transaction's input needs Alice's and Bob's signatures.

But even though these two things haven't happened, and even though Alice's node hasn't broadcast the funding transaction, she can still construct the refund transaction. She can do so because she can calculate the funding transaction's hash and reference it as an input in the refund transaction.

Notice how Alice has calculated 6da3c2...387710 as the funding transaction hash? If and when the funding transaction is broadcast, that hash will be recorded as the transaction ID of the funding transaction. Therefore, the `0` output of the funding transaction (the 2-of-2 address output) will then be referenced as output ID 6da3c2...387710:0. The refund transaction can be constructed to spend that funding transaction output even though it doesn't exist yet, because Alice knows what its identifier will be once confirmed.

This means that Alice can create a chained transaction by referencing an output that doesn't yet exist, knowing that the reference will be valid if the funding transaction is confirmed, making the refund transaction valid too. As we will see in the next section, this "trick" of chaining transactions before they are broadcast requires a very important feature of Bitcoin that was introduced in August of 2017: *Segregated Witness*.

Solving Malleability (Segregated Witness)

Alice has to depend on the transaction ID of the funding transaction being known before confirmation. But before the introduction of Segregated Witness (SegWit) in August 2017, this was not sufficient to protect Alice. Because of the way transactions were constructed with the signatures (witnesses) included in the transaction ID, it was possible for a third party (e.g., Bob) to broadcast an alternative version of a transaction with a *malleated* (modified) transaction ID. This is known as *transaction malleability*, and prior to SegWit, this problem made it difficult to implement indefinite lifetime payment channels securely.

If Bob could modify Alice's funding transaction before it was confirmed, and produce a replica that had a different transaction ID, Bob could make Alice's refund transaction invalid and hijack her bitcoin. Alice would be at Bob's mercy to get a signature to release her funds and could easily be blackmailed. Bob couldn't steal the funds, but he could prevent Alice from getting them back.

The introduction of SegWit made unconfirmed transaction IDs immutable from the point of view of third parties, meaning that Alice could be sure that the transaction ID of the funding transaction would not change. As a result, Alice can be confident that if she gets Bob's signature on the refund transaction, she has a way to recover her money. She now has a way to implement the Bitcoin equivalent of a "prenup" before locking her funds into the multisig.

TIP You might have wondered how Bob would be able to alter (malleate) a transaction created and signed by Alice. Bob certainly does not have Alice's private keys. However ECDSA signatures for a message are not unique. Knowing a signature (which is included in a valid transaction) allows one to produce many different-looking signatures that are still valid. Before SegWit removed signatures from the transaction digest algorithm, Bob could replace the signature with an equivalent valid signature that produced a different transaction ID, breaking the chain between the funding transaction and the refund transaction.

The funding_created message

Now that Alice has constructed the necessary transactions, the channel construction message flow continues. Alice transmits the funding_created message to Bob. You can see the contents of this message here:

The funding_created message

```
[32*byte:temporary_channel_id]
[sha256:funding_txid]
[u16:funding_output_index]
[signature:signature]
```

With this message, Alice gives Bob the important information about the funding transaction that anchors the payment channel:

funding_txid

This is the transaction ID (TxID) of the funding transaction, and is used to create the channel ID once the channel is established.

funding_output_index

This is the output index, so Bob knows which output of the transaction (e.g., output `0`) is the 2-of-2 multisig output funded by Alice. This is also used to form the channel ID.

Finally, Alice also sends the signature corresponding to Alice's `funding_pubkey` and used to spend from the 2-of-2 multisig. This is needed by Bob because he will also need to create his own version of a commitment transaction. That commitment transaction needs a signature from Alice, which she provides to him. Note that the commitment transactions of Alice and Bob look slightly different, thus the signatures will be different. Knowing what the commitment transaction of the other party looks like is crucial and part of the protocol to provide the valid signature.

TIP

In the Lightning protocol we often see nodes sending signatures instead of entire signed transactions. That's because either side can reconstruct the same transaction and therefore only the signature is needed to make it valid. Sending only the signature and not the entire transaction saves a lot of network bandwidth.

The funding_signed message

After receiving the funding_created message from Alice, Bob now knows the funding transaction ID and output index. The channel ID is made by a bitwise "exclusive or" (XOR) of the funding transaction ID and output index:

```
channel_id = funding_txid XOR funding_output_index
```

More precisely, a `channel_id`, which is the 32-byte representation of a funding UTXO, is generated by XORing the lower 2 bytes of the funding TxID with the index of the funding output.

Bob will also need to send Alice his signature for the refund transaction, based on Bob's `funding_pubkey` that formed the 2-of-2 multisig. Although Bob already has his local refund transaction, this will allow Alice to complete the refund transaction with all necessary signatures and be sure her money is refundable in case something goes wrong.

Bob constructs a funding_signed message and sends it to Alice. Here we see the contents of this message:

The funding_signed message

```
[channel_id:channel_id]
[signature:signature]
```

Broadcasting the Funding Transaction

Upon receiving the funding_signed message from Bob, Alice now has both signatures needed to sign the refund transaction. Her "exit plan" is now secure, and therefore she can broadcast the funding transaction without fear of having her funds locked. If anything goes wrong, Alice can simply broadcast the refund transaction and get her money back, without any further help from Bob.

Alice now sends the funding transaction to the Bitcoin network so that it can be mined into the blockchain. Both Alice and Bob will be watching for this transaction and waiting for minimum_depth confirmations (e.g., six confirmations) on the Bitcoin blockchain.

TIP Of course Alice will use the Bitcoin Protocol to verify that the signature that Bob sent her is indeed valid. This step is very crucial. If for some reason Bob was sending wrongful data to Alice, her "exit plan" would be sabotaged.

The funding_locked message

As soon as the funding transaction has reached the required number of confirmations, both Alice and Bob send the funding_locked message to each other and the channel is ready for use.

Sending Payments Across the Channel

The channel has been set up, but in its initial state, all the capacity (140,000 satoshis) is on Alice's side. This means that Alice can send payments to Bob across the channel, but Bob has no funds to send to Alice yet.

In the next few sections we will show how payments are made across the payment channel and how the *channel state* is updated.

Let's assume that Alice wants to send 70,000 satoshis to Bob to pay her bill at Bob's coffee shop.

Splitting the Balance

In principle, sending a payment from Alice to Bob is simply a matter of redistributing the balance of the channel. Before the payment is sent, Alice has 140,000 satoshis and Bob has none. After the 70,000 satoshi payment is sent, Alice has 70,000 satoshis and Bob has 70,000 satoshis.

Therefore, all Alice and Bob have to do is create and sign a transaction that spends the 2-of-2 multisig to two outputs paying Alice and Bob their corresponding balances. We call this updated transaction a *commitment transaction*.

Alice and Bob operate the payment channel by *advancing the channel state* through a series of commitments. Each commitment updates the balances to reflect payments that have flowed across the channel. Both Alice and Bob can initiate a new commitment to update the channel.

In Multiple commitment transactions we see several commitment transactions.

The first commitment transaction shown in Multiple commitment transactions is the refund transaction that Alice constructed before funding the channel. In the diagram, this is Commitment #0. After Alice pays Bob 70,000 satoshis, the new commitment transaction (Commitment #1) has two outputs paying Alice and Bob their respective balances. We have included two subsequent commitment transactions (Commitment #2 and Commitment #3) which represent Alice paying Bob an additional 10,000 satoshis and then 20,000 satoshis, respectively.

Each signed and valid commitment transaction can be used by either channel partner at any time to close the channel by broadcasting it to the Bitcoin network. Since they both have the most recent commitment transaction and can use it at any time, they can also just hold it and not broadcast it. It’s their guarantee of a fair exit from the channel.

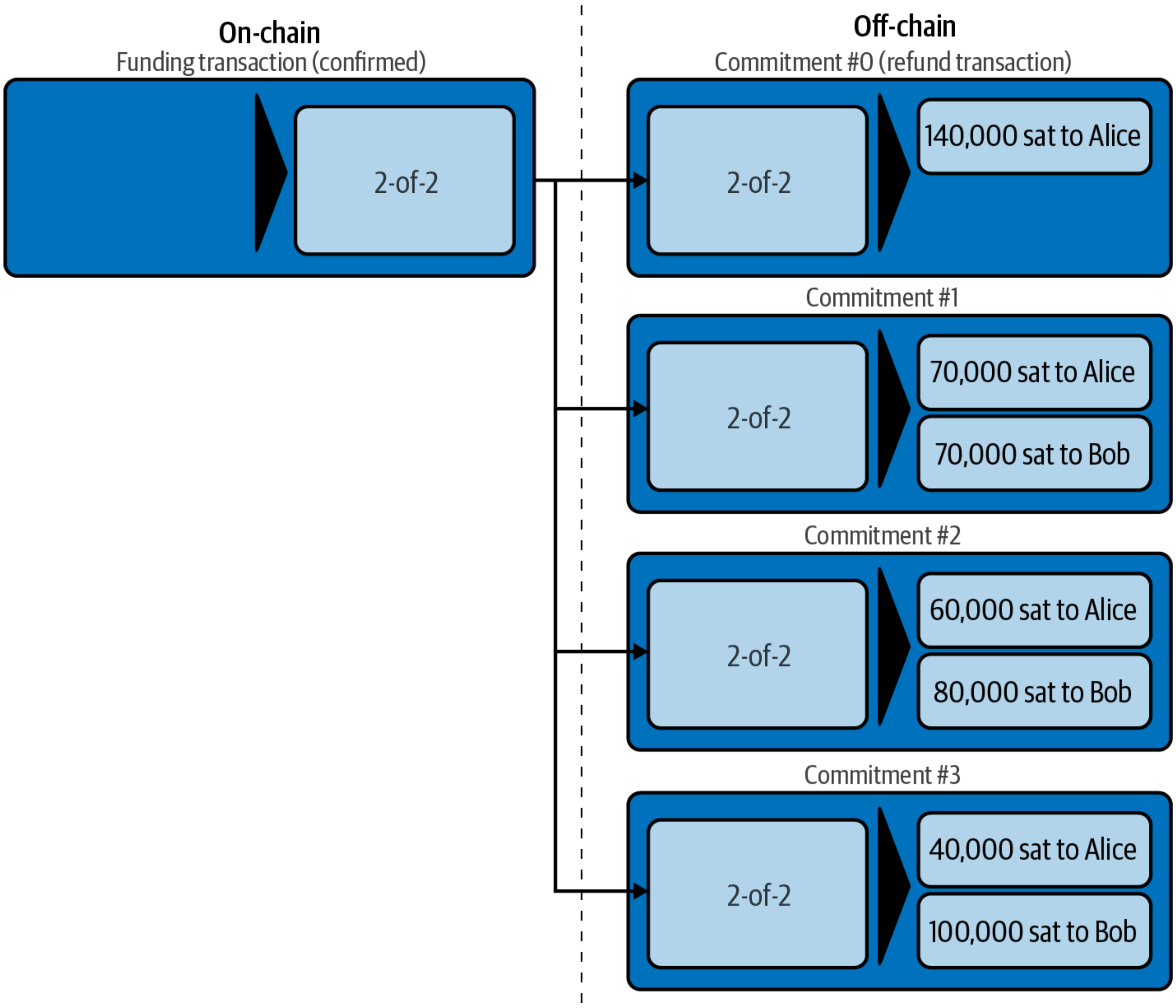


Figure 28. Multiple commitment transactions

Competing Commitments

You may be wondering how it is possible for Alice and Bob to have multiple commitment transactions, all of them attempting to spend the same 2-of-2 output from the funding transaction. Aren’t these commitment transactions conflicting? Isn’t this a "double-spend" that the Bitcoin system is meant to prevent?

It is indeed! In fact, we rely on Bitcoin’s ability to *prevent* a double-spend to make Lightning work.

No matter how many commitment transactions Alice and Bob construct and sign, only one of them can actually get confirmed.

As long as Alice and Bob hold these transactions and don't broadcast them, the funding output is unspent. But if a commitment transaction is broadcast and confirmed, it will spend the funding output. If Alice or Bob attempts to broadcast more than one commitment transaction, only one of them will be confirmed and the others will be rejected as attempted (and failed) double-spends.

If more than one commitment transaction is broadcast, there are many factors that will determine which one gets confirmed first: the amount of fees included, the speed of propagation of these competing transactions, network topology, etc. Essentially it becomes a race without a predictable outcome. That doesn't sound very secure. It sounds like someone could cheat.

Cheating with Old Commitment Transactions

Let's look more carefully at the commitment transactions in Multiple commitment transactions. All four commitment transactions are signed and valid. But only the last one accurately reflects the most recent channel balances. In this particular scenario, Alice has an opportunity to cheat by broadcasting an older commitment and getting it confirmed on the Bitcoin blockchain. Let's say Alice transmits Commitment #0 and gets it confirmed: she will effectively close the channel and take all 140,000 satoshis herself. In fact, in this particular example any commitment but Commitment #3 improves Alice's position and allows her to "cancel" at least part of the payments reflected in the channel.

In the next section we will see how the Lightning Network resolves this problem—preventing older commitment transactions from being used by the channel partners by a mechanism of revocation and penalties. There are other ways to prevent the transmission of older commitment transactions, such as eltoo channels, but they require an upgrade to Bitcoin called input rebinding (see Bitcoin Protocol and Bitcoin Script Innovation).

Revoking Old Commitment Transactions

Bitcoin transactions do not expire and cannot be "canceled." Neither can they be stopped or censored once they have been broadcast. So how do we "revoke" a transaction that another person holds that has already been signed?

The solution used in Lightning is another example of a fairness protocol. Instead of trying to control the ability to broadcast a transaction, there is a built-in *penalty mechanism* that ensures it is not in the best interest of a would-be cheater to transmit an old commitment transaction. They can always broadcast it, but they will most likely lose money if they do so.

TIP

The word "revoke" is a misnomer because it implies that older commitments are somehow made invalid and cannot be broadcast and confirmed. But this is not the case, since valid Bitcoin transactions cannot be revoked. Instead, the Lightning protocol uses a penalty mechanism to punish the channel partner who broadcasts an old commitment.

There are three elements that make up the Lightning protocol's revocation and penalty mechanism:

Asymmetric commitment transactions

Alice's commitment transactions are slightly different from those held by Bob.

Delayed spending

The payment to the party holding the commitment transaction is delayed (timelocked), whereas the payment to the other party can be claimed immediately.

Revocation keys

Used to unlock a penalty option for old commitments.

Let's look at these three elements in turn.

Asymmetric Commitment Transactions

Alice and Bob hold slightly different commitment transactions. Let's look specifically at Commitment #2 from Multiple commitment transactions, in more detail in Figure 7-7.

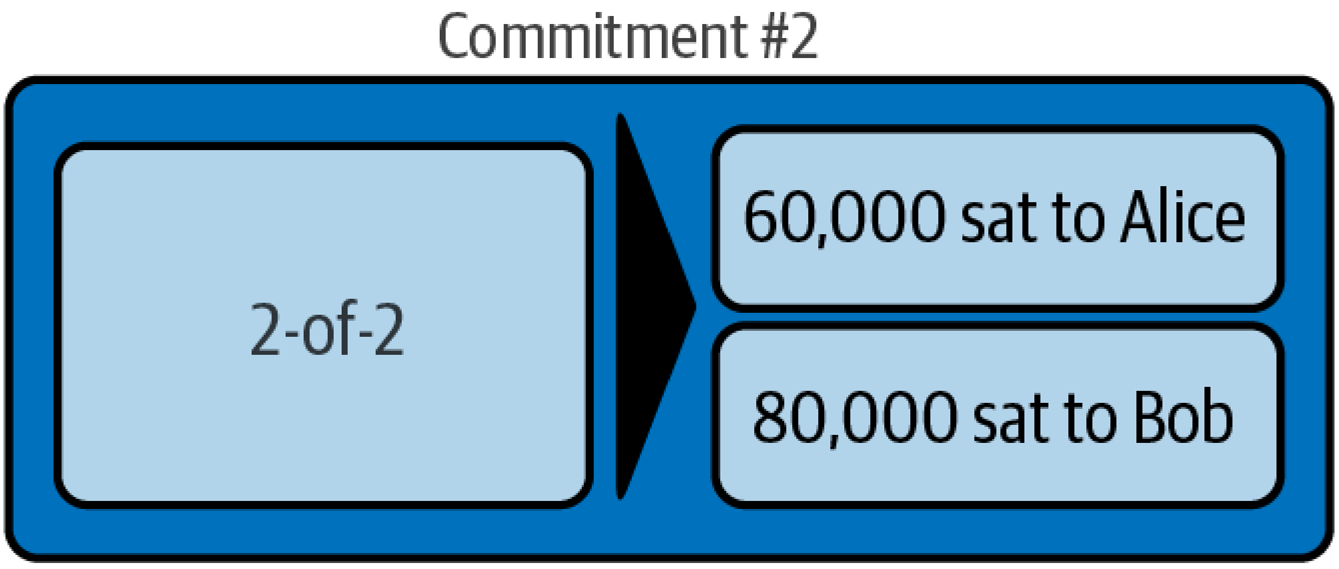


Figure 29. Commitment transaction #2

Alice and Bob hold two different variations of this transaction, as shown in Asymmetric commitment transactions.

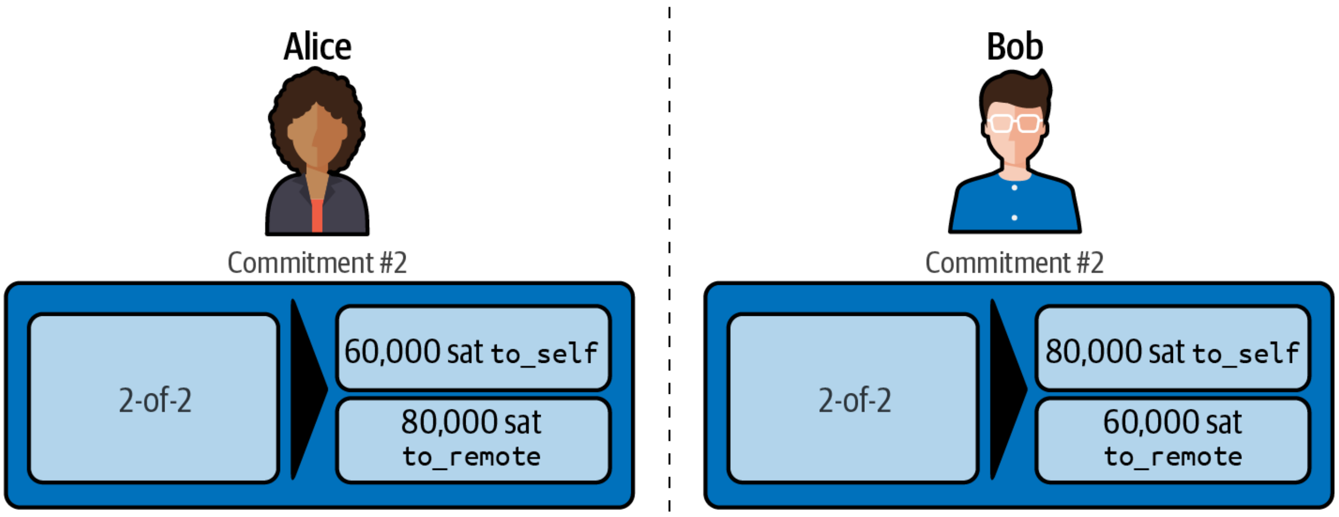


Figure 30. Asymmetric commitment transactions

By convention, within the Lightning protocol, we refer to the two channel partners as `self` (also known as `local`) and `remote`, depending on which side we're looking at. The outputs that pay each channel partner are called `to_local` and `to_remote`, respectively.

In Asymmetric commitment transactions we see that Alice holds a transaction that pays 60,000 satoshis `to_self` (can be spent by Alice's keys), and 80,000 satoshis `to_remote` (can be spent by Bob's keys).

Bob holds the mirror image of that transaction, where the first output is 80,000 satoshis `to_self` (can be spent by Bob's keys), and 60,000 satoshis `to_remote` (can be spent by Alice's keys).

Delayed (Timelocked) Spending to_self

Using asymmetric transactions allows the protocol to easily ascribe *blame* to the cheating party. An invariant that the *broadcasting* party must always wait ensures that the "honest" party has time to refute the claim and revoke their funds. This asymmetry is manifested in the form of differing outputs for each side: the `to_local` output is always timelocked and can't be spent immediately, whereas the `to_remote` output is not timelocked and can be spent immediately.

In the commitment transaction held by Alice, for example, the `to_local` output that pays her is timelocked for 432 blocks, whereas the `to_remote` output that pays Bob can be spent immediately (see Figure 7-9). Bob's commitment transaction for Commitment #2 is the mirror image: his own (`to_local`) output is timelocked and Alice's `to_remote` output can be spent immediately.

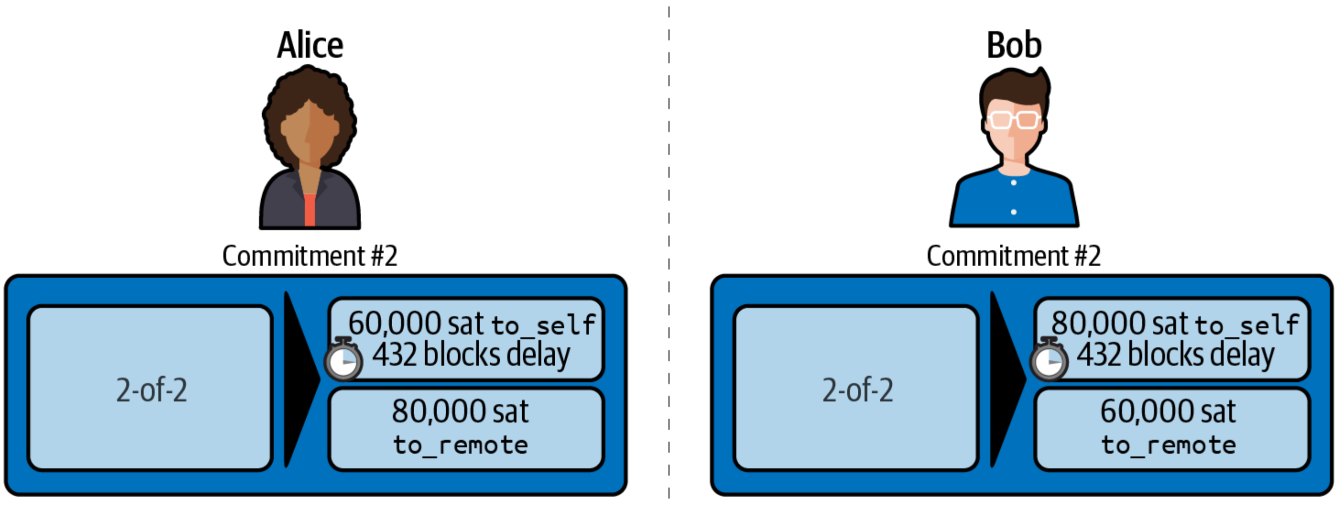


Figure 31. Asymmetric and delayed commitment transactions

That means that if Alice closes the channel by broadcasting and confirming the commitment transaction she holds, she cannot spend her balance for 432 blocks, but Bob can claim his balance immediately. If Bob closes the channel using the commitment transaction he holds, he cannot spend his output for 432 blocks while Alice can immediately spend hers.

The delay is there for one reason: to allow the *remote* party to exercise a penalty option if an old (revoked) commitment should be broadcast by the other channel partner. Let's look at the revocation keys and penalty option next.

The delay is negotiated by Alice and Bob, during the initial channel construction message flow, as a field called to_self_delay. To ensure the security of the channel, the delay is scaled to the capacity of the channel—meaning a channel with more funds has longer delays in the to_self outputs in commitments. Alice's node includes a desired to_self_delay in the open_channel message. If Bob finds this acceptable, his node includes the same value for to_self_delay in the accept_channel message. If they do not agree, then the channel is rejected (see The Shutdown Message).

Revocation Keys

As we discussed previously, the word "revocation" is a bit misleading because it implies that the "revoked" transaction cannot be used.

In fact, the revoked transaction can be used, but if it is used, and it has been revoked, then one of the channel partners can take all of the channel funds by creating a penalty transaction.

The way this works is that the `to_local` output is not only timelocked, but it also has two spending conditions in the script: it can be spent by *self* after the timelock delay *or* it can be spent by *remote* immediately with a revocation key for this commitment.

So, in our example, each side holds a commitment transaction that includes a revocation option in the `to_local` output, as shown in Asymmetric, delayed, and revocable commitments.

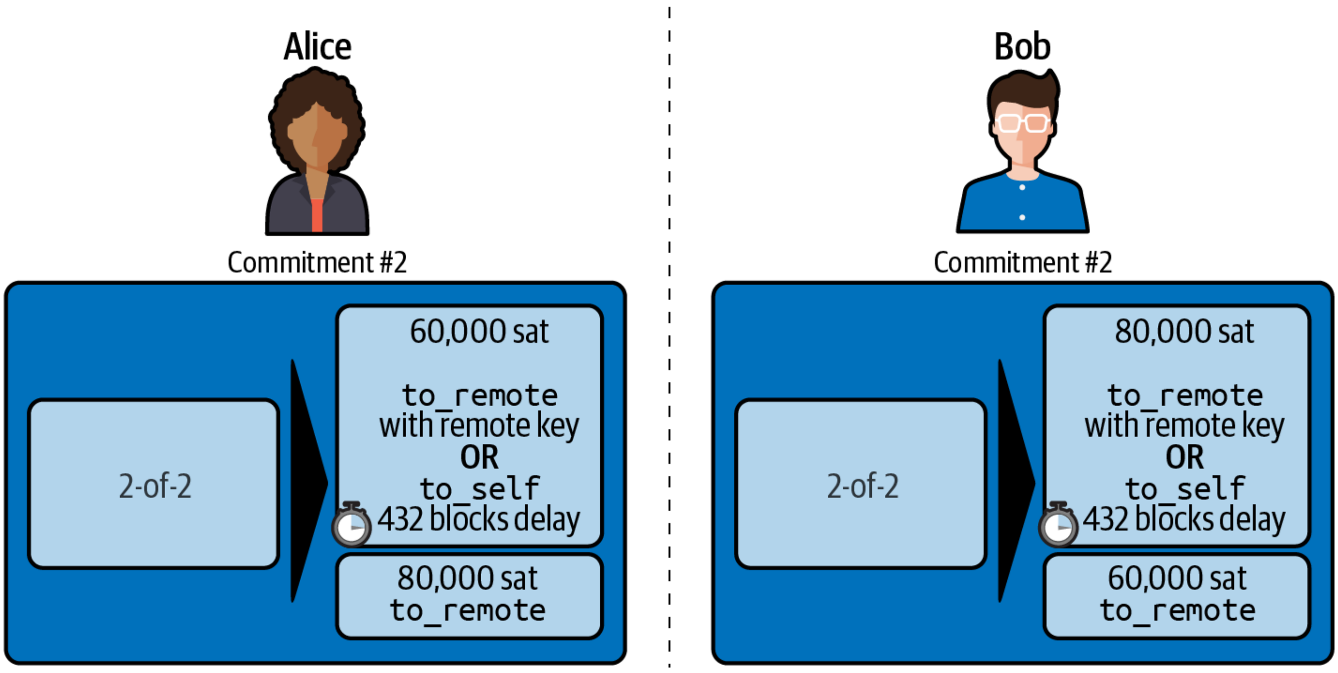


Figure 32. Asymmetric, delayed, and revocable commitments

The Commitment Transaction

Now that we understand the structure of commitment transactions and why we need asymmetric, delayed, revocable commitments, let's look at the Bitcoin Script that implements this.

The first (`to_local`) output of a commitment transaction is defined in BOLT #3: Commitment Transaction, `to_local` Output, as follows:

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

This is a conditional script (see Scripts with Multiple Conditions), which means the output can be spent if *either* of the two conditions is met. The first clause allows the output to be spent by anyone who can sign for <revocationpubkey>. The second clause is timelocked by <to_self_delay> blocks and can only be spent after that many blocks by anyone who can sign for <local_delayedpubkey>. In our example, we had set the <to_self_delay> timelock to 432 blocks, but this is a configurable delay that is negotiated by the two channel partners. The to_self_delay timelock duration is usually chosen in proportion to the channel capacity, meaning that larger capacity channels (more funds), have longer to_self_delay timelocks to protect the parties.

The first clause allows the output to be spent by anyone who can sign for <revocationpubkey>. A critical requirement to the security of this script is that the remote party *cannot* unilaterally sign with the `revocationpubkey`. To see why this is important, consider the scenario in which the remote party breaches a previously revoked commitment. If they can sign with this key, then they can simply take the revocation clause *themselves* and steal all the funds in the channel. Instead, we derive the `revocationpubkey` for *each* state based on information from *both* the self (local) and remote party. A clever use of symmetric and asymmetric cryptography is used to allow both sides to compute the `revocationpubkey` public key, but only allow the honest self party to compute the private key given their secret information, as detailed in Revocation and Commitment Secret Derivations.

Revocation and Commitment Secret Derivations

Each side sends a revocation_basepoint during the initial channel negotiation messages as well as a first_per_commitment_point. The revocation_basepoint is static for the lifetime of the channel, while each new channel state will be based off a new first_per_commitment_point.

Given this information, the revocationpubkey for each channel state is derived via the following series of elliptic curve and hashing operations:

```
revocationpubkey = revocation_basepoint * sha256(revocation_basepoint ||
per_commitment_point) + per_commitment_point * sha256(per_commitment_point ||
revocation_basepoint)
```

Due to the commutative property of the abelian groups that elliptic curves are defined over, once the per_commitment_secret (the private key for the per_commitment_point) is revealed by the remote party, self can derive the private key for the revocationpubkey with the following operation:

```
revocation_priv = (revocationbase_priv * sha256(revocation_basepoint ||
per_commitment_point)) + (per_commitment_secret * sha256(per_commitment_point ||
revocation_basepoint)) mod N
```

To see why this works in practice, notice that we can *reorder* (commute) and expand the public key computation of the original formula for revocationpubkey:

```
revocationpubkey = G*(revocationbase_priv * sha256(revocation_basepoint ||
per_commitment_point) + G*(per_commitment_secret * sha256(per_commitment_point ||
revocation_basepoint))
                 = revocation_basepoint * sha256(revocation_basepoint ||
per_commitment_point) + per_commitment_point * sha256(per_commitment_point ||
revocation_basepoint))
```

In other words, the revocationbase_priv can only be derived (and used to sign for the revocationpubkey) by the party that knows *both* the revocationbase_priv *and* the per_commitment_secret. This little trick is what makes the public-key-based revocation system used in the Lightning Network secure.

TIP The timelock used in the commitment transaction with CHECKSEQUENCEVERIFY is a *relative timelock*. It counts elapsed blocks from the confirmation of this output. That means it will not be spendable until the to_self_delay block *after* this commitment transaction is broadcast and confirmed.

The second output (to_remote) output of the commitment transaction is defined in BOLT #3: Commitment Transaction, to_remote Output, and in the simplest form is a Pay-to-Witness-Public-

Key-Hash (P2WPKH) for <remote_pubkey>, meaning that it simply pays the owner who can sign for <remote_pubkey>.

Now that we've defined the commitment transactions in detail, let's see how Alice and Bob advance the state of the channel, create and sign new commitment transactions, and revoke old commitment transactions.

Advancing the Channel State

To advance the state of the channel, Alice and Bob exchange two messages: commitment_signed and revoke_and_ack messages. The commitment_signed message can be sent by either channel partner when they have an update to the channel state. The other channel partner then may respond with revoke_and_ack to *revoke* the old commitment and *acknowledge* the new commitment.

In Commitment and revocation message flow we see Alice and Bob exchanging two pairs of commitment_signed and revoke_and_ack. The first flow shows a state update initiated by Alice (left to right commitment_signed), to which Bob responds (right to left revoke_and_ack). The second flow shows a state update initiated by Bob and responded to by Alice.

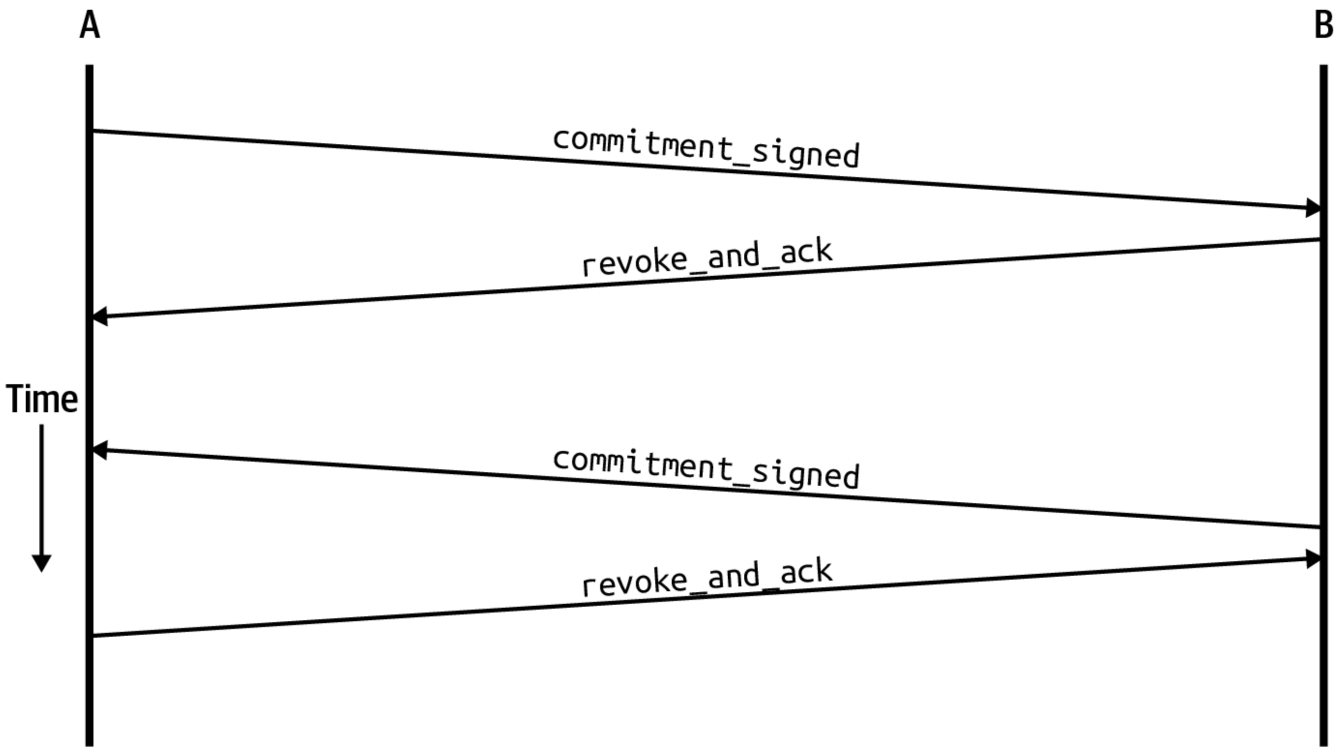


Figure 33. Commitment and revocation message flow

The commitment_signed Message

The structure of the commitment_signed message is defined in BOLT #2: Peer Protocol, `commitment_signed`, and shown here:

The commitment_signed message

```
[channel_id:channel_id]
[signature:signature]
[u16:num_htlcs]
[num_htlcs*signature:htlc_signature]
```

channel_id

The identifier of the channel

signature

The signature for the new remote commitment

num_htlcs

The number of updated HTLCs in this commitment

htlc_signature

The signatures for the updates

> **NOTE** The use of HTLCs to commit updates will be explained in detail in Hash Time-Locked Contracts and in Channel Operation and Payment Forwarding.

Alice's commitment_signed message gives Bob the signature needed (Alice's part of the 2-of-2) for a new commitment transaction.

The revoke_and_ack Message

Now that Bob has a new commitment transaction, he can revoke the previous commitment by giving Alice a revocation key, and construct the new commitment with Alice's signature.

The revoke_and_ack message is defined in BOLT #2: Peer Protocol, `revoke_and_ack`, and shown here:

The revoke_and_ack message

```
[channel_id:channel_id]
[32*byte:per_commitment_secret]
[point:next_per_commitment_point]
```

channel_id

This is the identifier of the channel.

per_commitment_secret

Used to generate a revocation key for the previous (old) commitment, effectively revoking it.

next_per_commitment_point

Used to build a `revocation_pubkey` for the new commitment, so that it can later be revoked.

Revoking and Recommitting

Let’s look at this interaction between Alice and Bob more closely.

Alice is giving Bob the means to create a new commitment. In return, Bob is revoking the old commitment to assure Alice that he won’t use it. Alice can only trust the new commitment if she has the revocation key to punish Bob for publishing the old commitment. From Bob’s perspective, he can safely revoke the old commitment by giving Alice the keys to penalize him, because he has a signature for a new commitment.

When Bob responds with revoke_and_ack, he gives Alice a per_commitment_secret. This secret can be used to construct the revocation signing key for the old commitment, which allows Alice to seize all channel funds by exercising a penalty.

As soon as Bob has given this secret to Alice, he *must not* ever broadcast that old commitment. If he does, he will give Alice the opportunity to penalize him by taking the funds. Essentially, Bob is giving Alice the ability to hold him accountable for broadcasting an old commitment, and in effect he has revoked his ability to use that old commitment.

Once Alice has received the revoke_and_ack from Bob, she can be sure that Bob cannot broadcast the old commitment without being penalized. She now has the keys necessary to create a penalty transaction if Bob broadcasts an old commitment.

Cheating and Penalty in Practice

In practice, both Alice and Bob have to monitor for cheating. They are monitoring the Bitcoin blockchain for any commitment transactions related to any of the channels they are operating. If they see a commitment transaction confirmed on-chain, they will check to see if it is the most recent commitment. If it is an "old" commitment, they must immediately construct and broadcast a penalty transaction. The penalty transaction spends *both* the to_local and to_remote outputs, closing the channel and sending both balances to the cheated channel partner.

To more easily allow both sides to keep track of the commitment numbers of the passed revoke commitments, each commitment actually *encodes* the number of the commitment within the lock time and sequence fields in a transition. Within the protocol, this special encoding is referred to as *state hints*. Assuming a party knows the current commitment number, they’re able to use the state hints to easily recognize if a broadcasted commitment was a revoked one, and if so, which commitment number was breached, as that number is used to easily look up which revocation secret should be used in the revocation secret tree (shachain).

Rather than encode the state hint in plain sight, an *obfuscated* state hint is used in its place. This obfuscation is achieved by first XORing the current commitment number with a set of random bytes generated deterministically using the funding public keys of both sides of the channel. A total of 6 bytes across the lock time and sequence (24 bits of the locktime and 24 bits of the sequence) are used to encode the state hint within the commitment transaction, so 6 random bytes are needed to use for XORing. To obtain these 6 bytes, both sides obtain the SHA-256 hash of the initiator’s funding key concatenated to the responder’s funding key. Before encoding the current commitment height, the integer is XORed with this state hint obfuscator, and then encoded in the lower 24 bits of the locktime, and the upper 64 bits of the sequence.

Let's review our channel between Alice and Bob and show a specific example of a penalty transaction. In Revoked and current commitments we see the four commitments on Alice and Bob's channel. Alice has made three payments to Bob:

- 70,000 satoshis paid and committed to Bob with Commitment #1
- 10,000 satoshis paid and committed to Bob with Commitment #2
- 20,000 satoshis paid and committed to Bob with Commitment #3

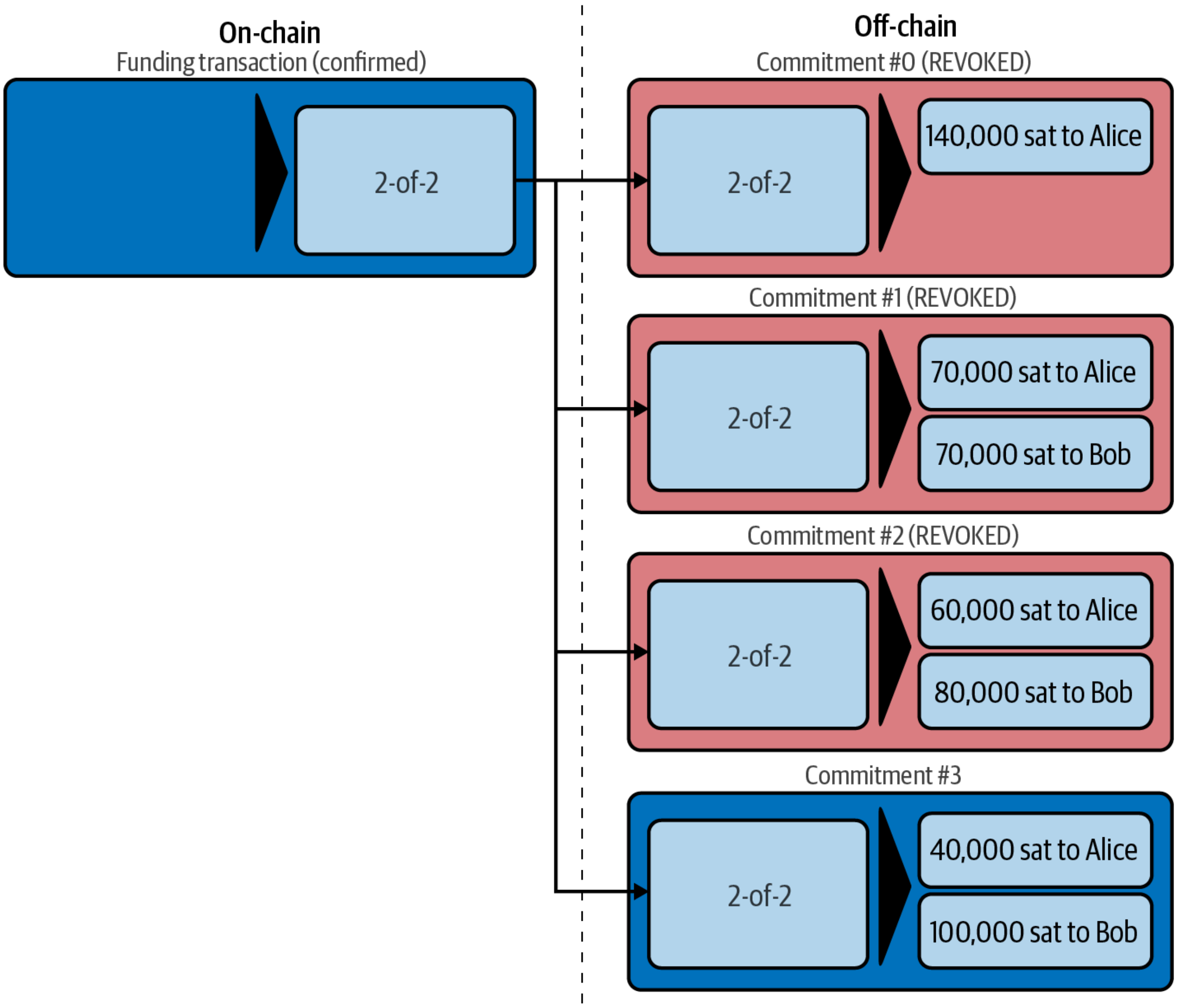


Figure 34. Revoked and current commitments

With each commitment, Alice has revoked the previous (older) commitment. The current state of the channel and the correct balance is represented by Commitment #3. All previous commitments have been revoked, and Bob has the keys necessary to issue penalty transactions against them, in case Alice tries to broadcast one of them.

Alice might have an incentive to cheat because all the previous commitment transactions would give her a higher proportion of the channel balance than she is entitled to. Let's say for example that Alice tried to broadcast Commitment #1. That commitment transaction would pay Alice 70,000 satoshis and Bob 70,000 satoshis. If Alice was able to broadcast and spend her to_local output, she would effectively be stealing 30,000 satoshis from Bob by rolling back her last two payments to Bob.

Alice decides to take a huge risk and broadcast the revoked Commitment #1, to steal 30,000 satoshis from Bob. In Alice cheating we see Alice's old commitment that she broadcasts to the Bitcoin blockchain.

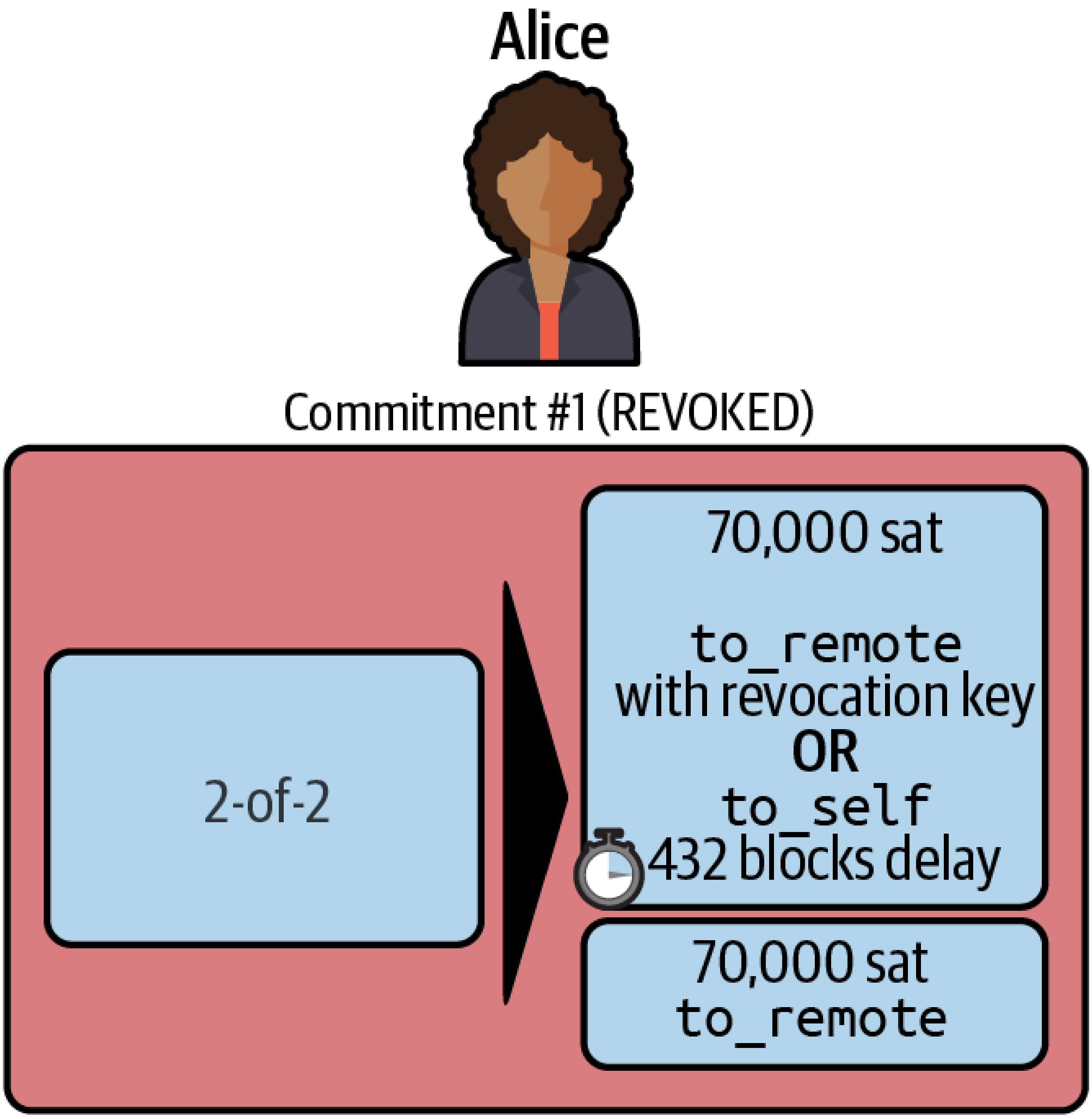


Figure 35. Alice cheating

As you can see, Alice's old commitment has two outputs, one paying herself 70,000 satoshis (to_local output) and one paying Bob 70,000 satoshis. Alice can't yet spend her 70,000 to_local output because it has a 432 block (3 day) timelock. She is now hoping that Bob doesn't notice for three days.

Unfortunately for Alice, Bob's node is diligently monitoring the Bitcoin blockchain and sees an old commitment transaction broadcast and (eventually) confirmed on-chain.

Bob's node will immediately broadcast a penalty transaction. Since this old commitment was revoked by Alice, Bob has the per_commitment_secret that Alice sent him. He uses that secret to construct a signature for the revocation_pubkey. While Alice has to wait for 432 blocks, Bob can spend *both* outputs immediately. He can spend the to_remote output with his private keys because it was meant to pay him anyway. He can also spend the output meant for Alice with a signature from the revocation key. His node broadcasts the penalty transaction shown in Cheating and penalty.

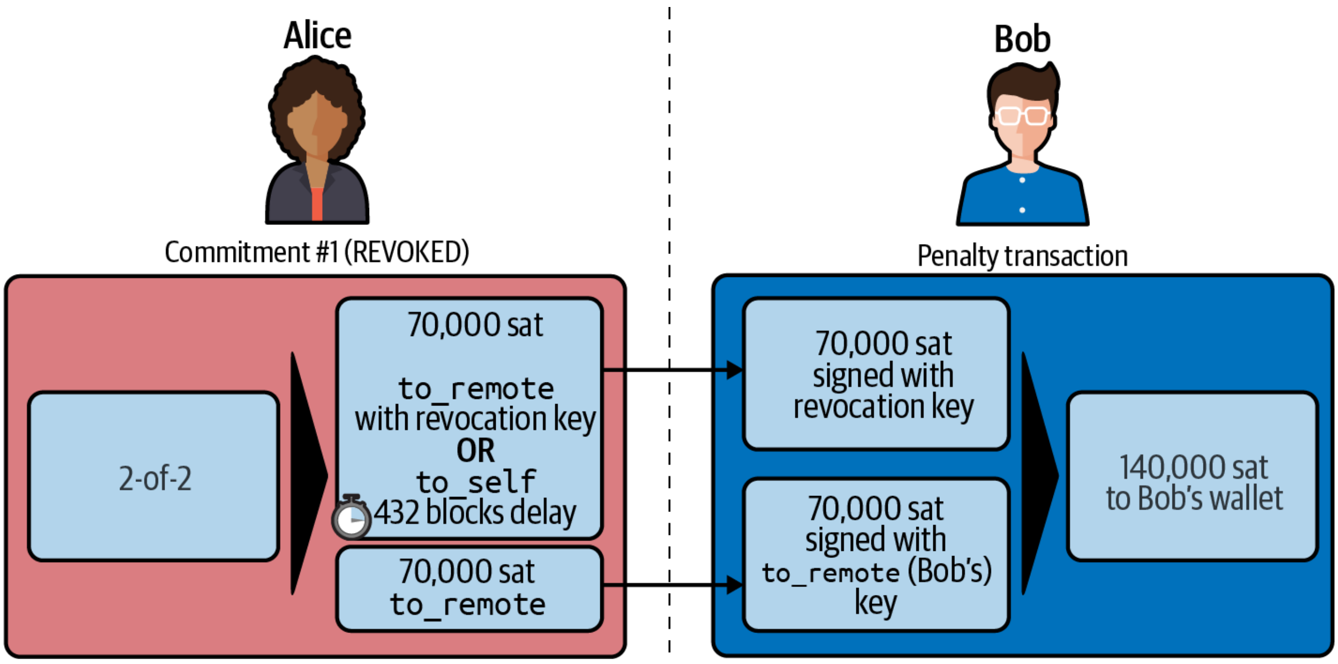


Figure 36. Cheating and penalty

Bob's penalty transaction pays 140,000 satoshis to his own wallet, taking the entire channel capacity. Alice has not only failed to cheat, she has lost everything in the attempt!

The Channel Reserve: Ensuring Skin in the Game

You may have noticed there is a special situation that needs to be dealt with. If Alice could keep spending her balance until it is zero, she would be in a position to close the channel by broadcasting an old commitment transaction without risking a penalty: either the revoked commitment transaction succeeds after the delay, or the cheater gets caught but there's no consequence because the penalty is zero. From a game theory perspective, it is free money to attempt to cheat in this situation. This is why the channel reserve is in play, so a prospective cheater always faces the risk of a penalty.

Closing the Channel (Cooperative Close)

So far we've looked at the commitment transactions as one possible way to close a channel, unilaterally. This type of channel closure is not ideal because it forces a timelock on the channel partner that uses it.

A better way to close a channel is a cooperative close. In a cooperative close, the two channel partners negotiate a final commitment transaction called the *closing transaction* that pays each party their balance immediately to the destination wallet of their choice. Then, the partner that initiated the channel closing flow will broadcast the closing transaction.

The closing message flow is defined in BOLT #2: Peer Protocol, Channel Close, and is shown in The channel close message flow.

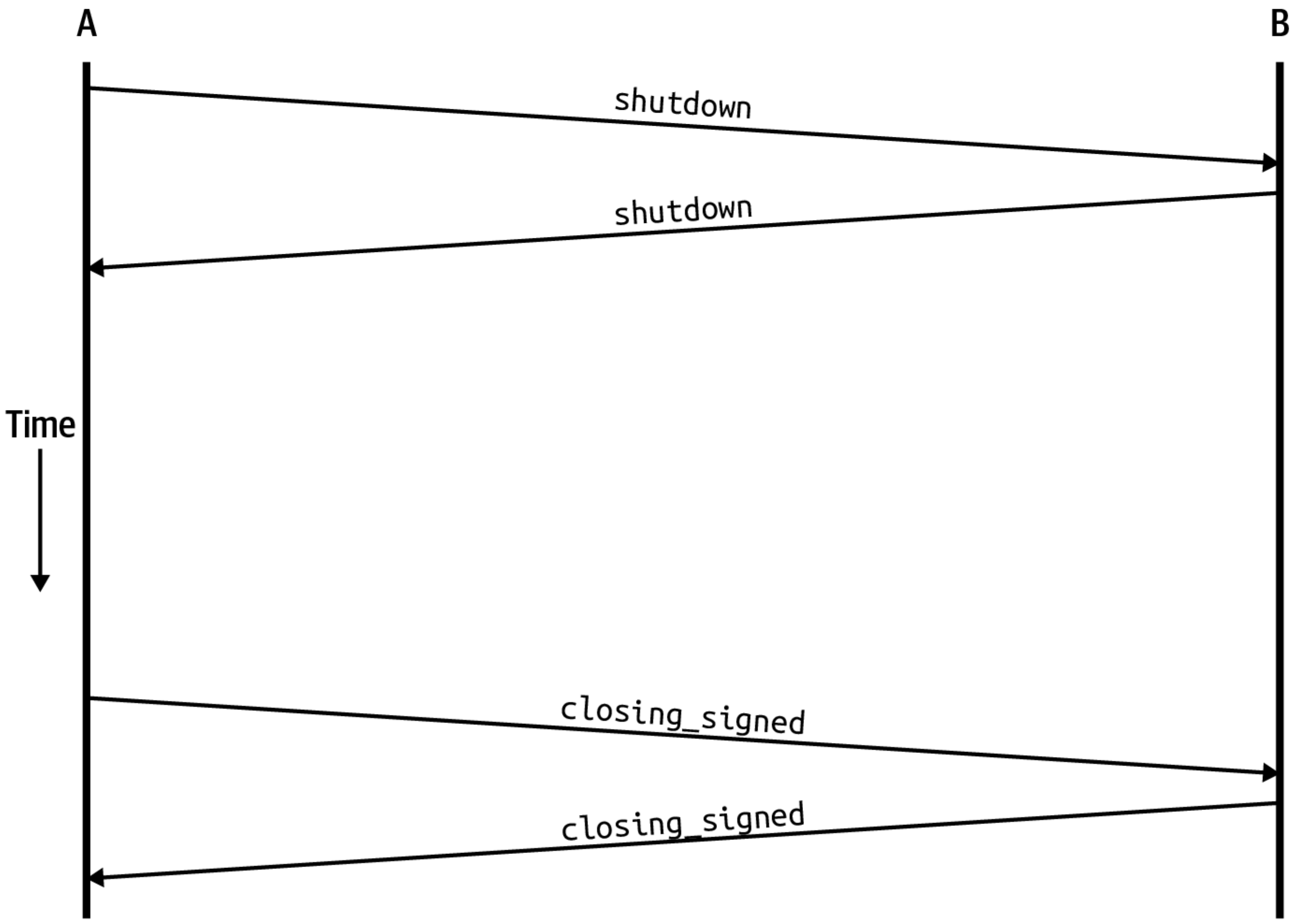


Figure 37. The channel close message flow

The Shutdown Message

Channel closing starts with one of the two channel partners sending the shutdown message. The contents of this message are shown here:

The shutdown message

```
[channel_id:channel_id]
[u16:len]
[len*byte:scriptpubkey]
```

channel_id

The channel identifier for the channel we want to close

len

The length of the script of the destination wallet that this channel partner wants to receive their balance

scriptpubkey

A Bitcoin script of the destination wallet, in one of the "standard" Bitcoin address formats (P2PKH, P2SH, P2WPKH, P2WSH, etc.; see the Glossary)

Let’s say Alice sends the shutdown message to Bob to close their channel. Alice will specify a Bitcoin script that corresponds to the Bitcoin address of her wallet. She’s telling Bob: let’s make a closing transaction that pays my balance to this wallet.

Bob will respond with his own shutdown message indicating that he agrees to cooperatively close the channel. His shutdown message includes the script for his wallet address.

Now both Alice and Bob have each other’s preferred wallet address, and they can construct identical closing transactions to settle the channel balance.

The closing_signed Message

Assuming the channel has no outstanding commitments or updates and the channel partners have exchanged the shutdown messages shown in the previous section, they can now finish this cooperative close.

The *funder* of the channel (Alice in our example) starts by sending a closing_signed message to Bob. This message proposes a transaction fee for the on-chain transaction, and Alice’s signature (the 2-of-2 multisig) for the closing transaction. The closing_signed message is shown here:

The closing_signed message

```
[channel_id:channel_id]
[u64:fee_satoshis]
[signature:signature]
```

channel_id
: The channel identifier

fee_satoshis
: The proposed on-chain transaction fee, in satoshis

signature
: The sender’s signature for the closing transaction

When Bob receives this, he can reply with a closing_signed message of his own. If he agrees with the fee, he simply returns the same proposed fee and his own signature. If he disagrees, he must propose a different fee_satoshis fee.

This negotiation may continue with back-and-forth closing_signed messages until the two channel partners agree on a fee.

Once Alice receives a closing_signed message with the same fee as the one she proposed in her last message, the negotiation is complete. Alice signs and broadcasts the closing transaction and the channel is closed.

The Cooperative Close Transaction

The cooperative close transaction looks similar to the last commitment transaction that Alice and Bob had agreed on. However, unlike the last commitment transaction, it does not have timelocks or penalty revocation keys in the outputs. Since both parties cooperate to produce this transaction and they won't be making any further commitments, there is no need for the asymmetric, delayed, and revocable elements in this transaction.

Typically the addresses used in this cooperative close transaction are generated freshly for each channel being closed. However, it's also possible for both sides to *lock in* a "delivery" address to be used to send their cooperatively settled funds to. Within the TLV namespace of both the `open_channel` and `accept_channel` messages, both sides are free to specify an "up-front shutdown script." Commonly, this address is derived from keys that reside in cold storage. This practice serves to increase the security of channels: if a channel partner is somehow hacked, then the hacker isn't able to cooperatively close the channel using an address they control. Instead, the uncompromised honest channel partner will refuse to cooperate on a channel closure if the specified up-front shutdown address isn't used. This feature effectively creates a "closed loop," restricting the flow of funds out of a given channel.

Alice broadcasts a transaction shown in The cooperative close transaction to close the channel.

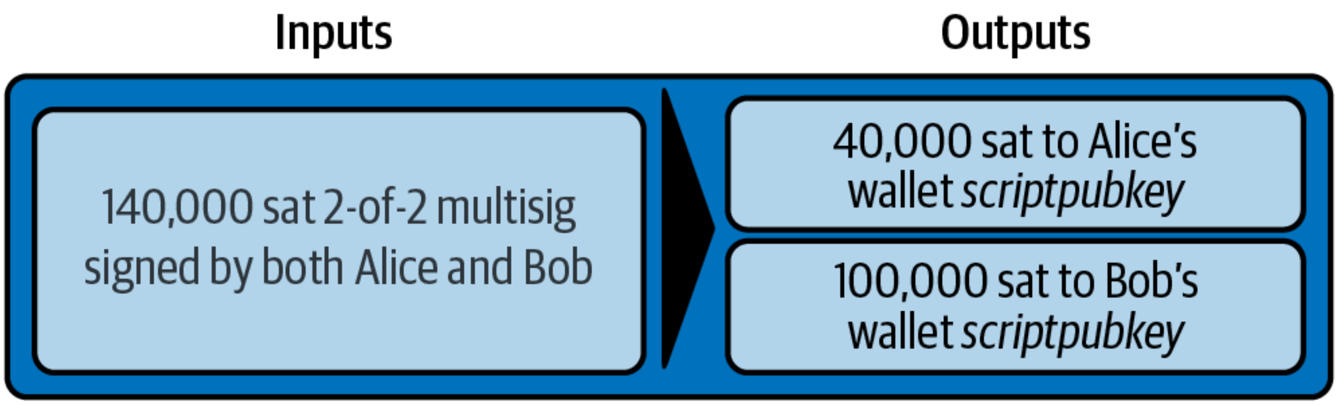

Inputs
Outputs
140,000 sat 2-of-2 multisig signed by both Alice and Bob
40,000 sat to Alice's wallet *scriptpubkey*
100,000 sat to Bob's wallet *scriptpubkey*

Figure 38. The cooperative close transaction

As soon as this closing transaction is confirmed on the Bitcoin blockchain, the channel is closed. Now, Alice and Bob can spend their outputs as they please.

Conclusion

In this section we looked at payment channels in much more detail. We examined three message flows used by Alice and Bob to negotiate funding, commitments, and closing of the channel. We also showed the structure of the funding, commitment, and closing transactions, and looked at the revocation and penalty mechanisms.

As we will see in the next few chapters, HTLCs are used even for local payments between channel partners. They are not necessary, but the protocol is much simpler if local (one channel) and routed (many channels) payments are done in the same way.

In a single payment channel, the number of payments per second is only bound by the network capacity between Alice and Bob. As long as the channel partners are able to send a few bytes of data back and forth to agree to a new channel balance, they have effectively made a payment. This is why we can achieve a much greater throughput of payments on the Lightning Network (off-chain) than the transaction throughput that can be handled by the Bitcoin blockchain (on-chain).

In the next few chapters we will discuss routing, HTLCs, and their use in channel operations.

Routing on a Network of Payment Channels

In this chapter we will finally unpack how payment channels can be connected to form a network of payment channels via a process called *routing*. Specifically, we will look at the first part of the routing layer, the "Atomic and trustless multihop contracts" protocol. This is highlighted by an outline in the protocol suite, shown in Atomic payment routing in the Lightning protocol suite.

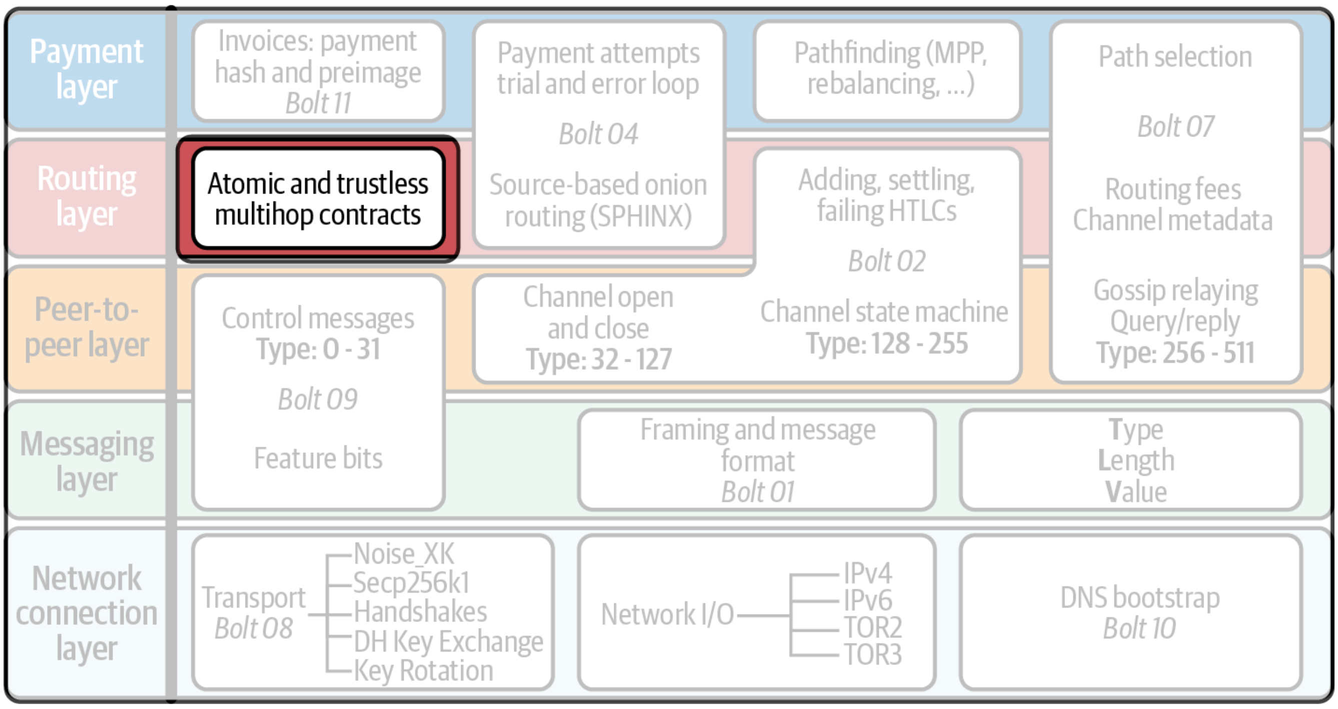


Figure 39. Atomic payment routing in the Lightning protocol suite

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.

In Fans connected (in)directly to Dina on the Lightning Network we see a possible network layout created by various payment channels between Lightning nodes. Everyone in this diagram can send Dina a payment by constructing a path. Imagine that Fan 4 wants to send Dina a payment. Do you see the path that could allow that to happen? Fan 4 could route a payment to Dina via Fan 3, Bob, and Chan. Similarly, Alice could route a payment to Dina via Bob and Chan.

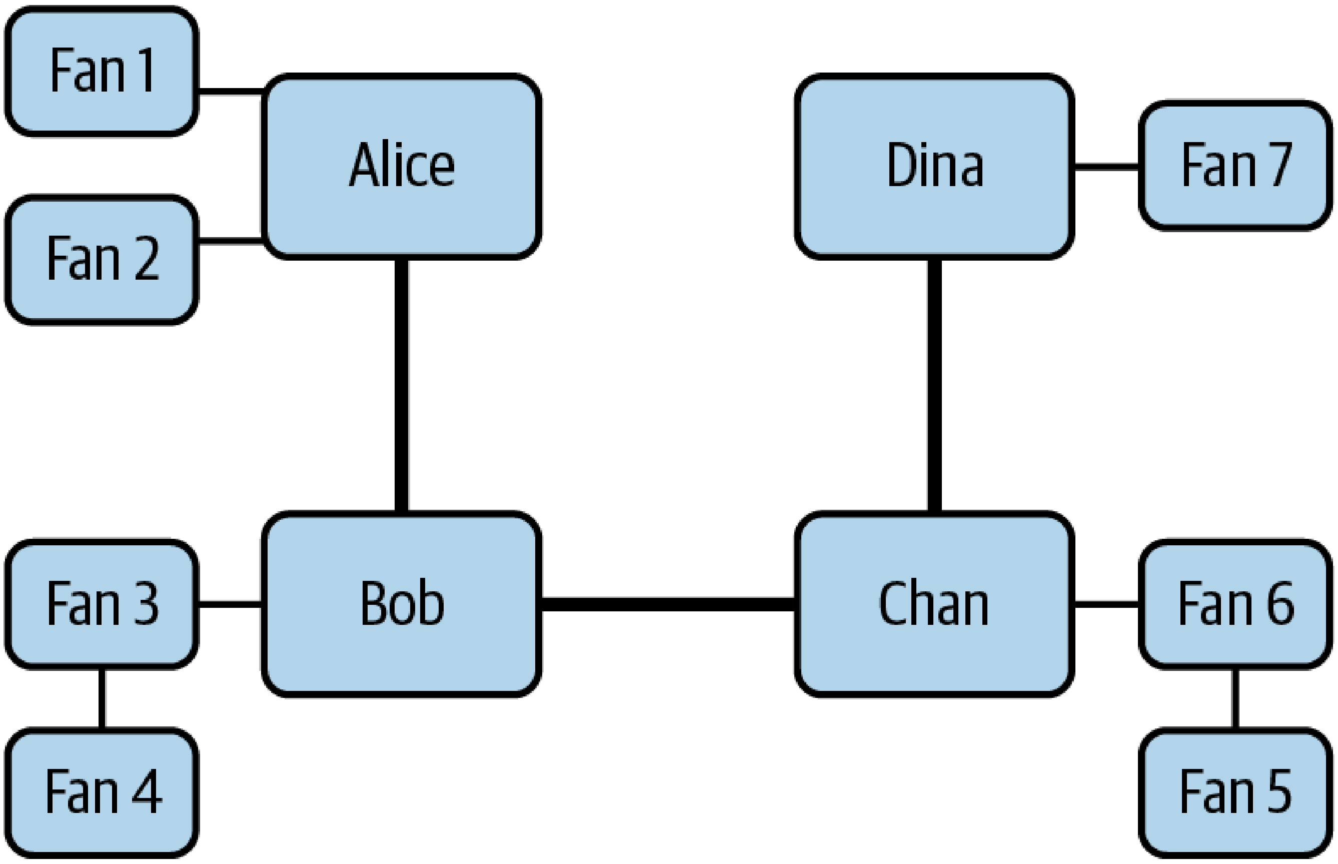


Figure 40. Fans connected (in)directly to Dina on the Lightning Network

The nodes along the path from the fan to Dina are intermediaries 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.

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 pathfinding, because these are often confused and used interchangeably. Next, we will construct the fairness protocol: an atomic, trustless, multihop protocol used to route payments. To demonstrate how this fairness protocol works, we will be using a physical equivalent of transferring gold coins between four people. Finally, we will look at the atomic, trustless, multihop protocol implementation currently used in the Lightning Network, which is called a hash time-locked contract (HTLC).

Routing versus Pathfinding

It’s important to note that we separate the concept of *routing* from the concept of *pathfinding*. 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.

Pathfinding, which is covered in Pathfinding and Payment Delivery, is the process of finding and choosing a contiguous path made of payment channels that connects sender A to recipient B. The sender of a payment does the pathfinding by examining the *channel graph* that they have assembled from channel announcements gossiped by other nodes.

Routing refers to the series of interactions across the network that attempt to forward a payment from some point A to another point B, across the path previously selected by pathfinding. 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 (perhaps even more than one), yet there may not be an active *route* on which to send the payment. One example is the scenario in which all the nodes connecting Alice and Bob are currently offline. In this example, one can examine the channel graph and connect a series of payment channels from Alice to Bob, hence a *path* exists. However, because 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 an atomic trustless multihop payment, let’s work through an example. Let’s return to Alice who, in previous chapters, purchased a coffee from Bob with whom she has an open channel. Now Alice is watching a live stream from Dina, the gamer, and wants to send Dina a tip of 50,000 satoshis via the Lightning Network. But Alice has no direct channel with Dina. What can Alice do?

Alice could open a direct channel with Dina; 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.

As you can see in A network of payment channels between Alice and Dina, 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.

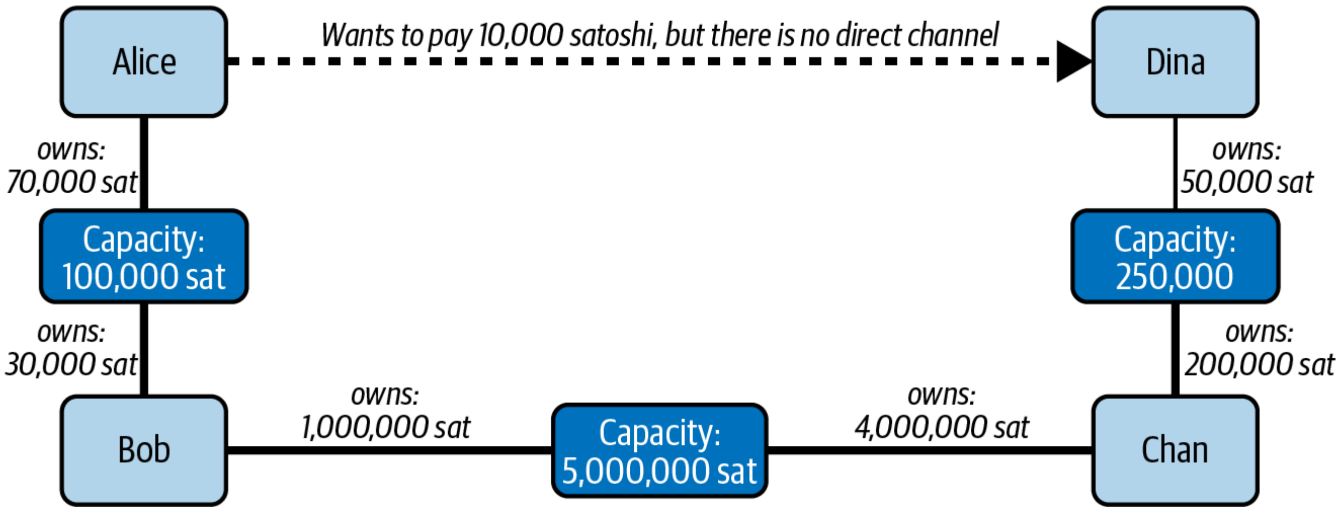


Figure 41. A network of payment channels between Alice and Dina

It’s possible to 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. No direct channel from Alice to Dina is required.

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 it 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, so she decides to ask Bob and Chan for help. This is shown in Alice wants to pay Dina 10 gold coins.

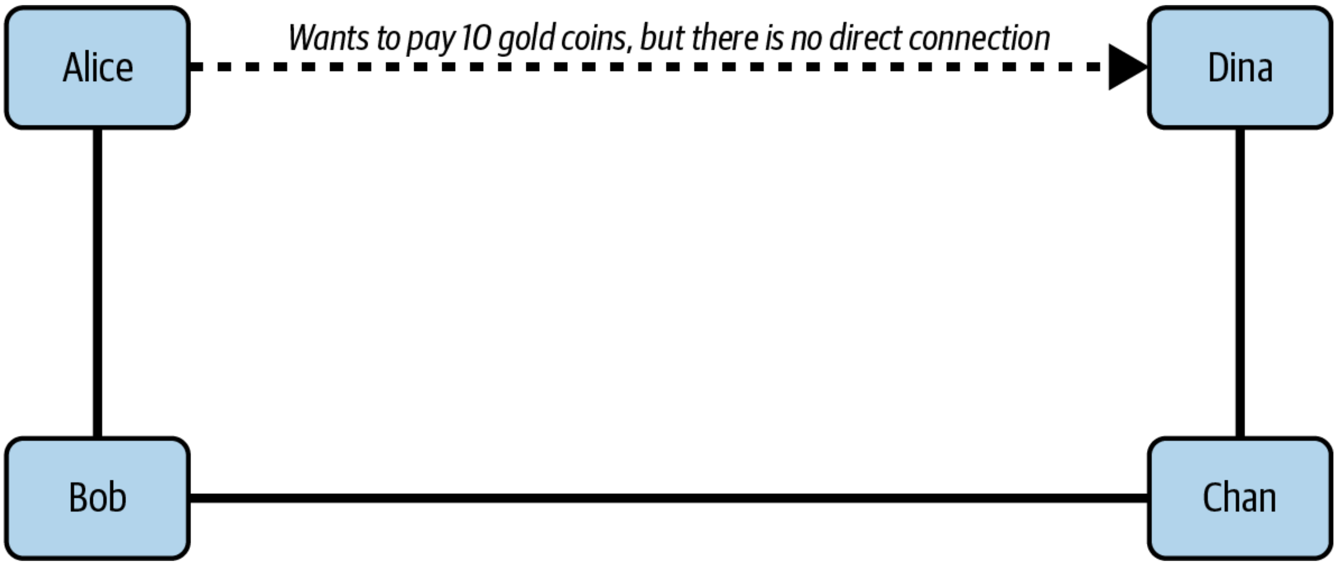

Alice
Wants to pay 10 gold coins, but there is no direct connection
Dina
Bob
Chan

Figure 42. Alice wants to pay Dina 10 gold coins

Alice 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 not to get caught. Even if Bob is caught and prosecuted, 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: getting the coins delivered to Dina.

Let's improve our contract to incorporate these considerations:

> *I, Alice, will reimburse you, Bob, with 10 gold coins if you can prove to me (for example, via a receipt) that you 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 to accept it.

Alice can then make this attractive to both Bob and Chan by offering them fees of one gold coin each, if they transmit her payment to Dina.

The contract would then read:

> *I, Alice, will reimburse you, Bob, with 12 gold coins if you can prove to me (for example, via a receipt) that you have delivered 11 gold 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 one gold coin is the fee that Bob will earn for helping out with this particular payment. In Alice pays Bob, Bob pays Chan, Chan pays Dina we see how this arrangement would get 10 gold coins to Dina via Bob and Chan.

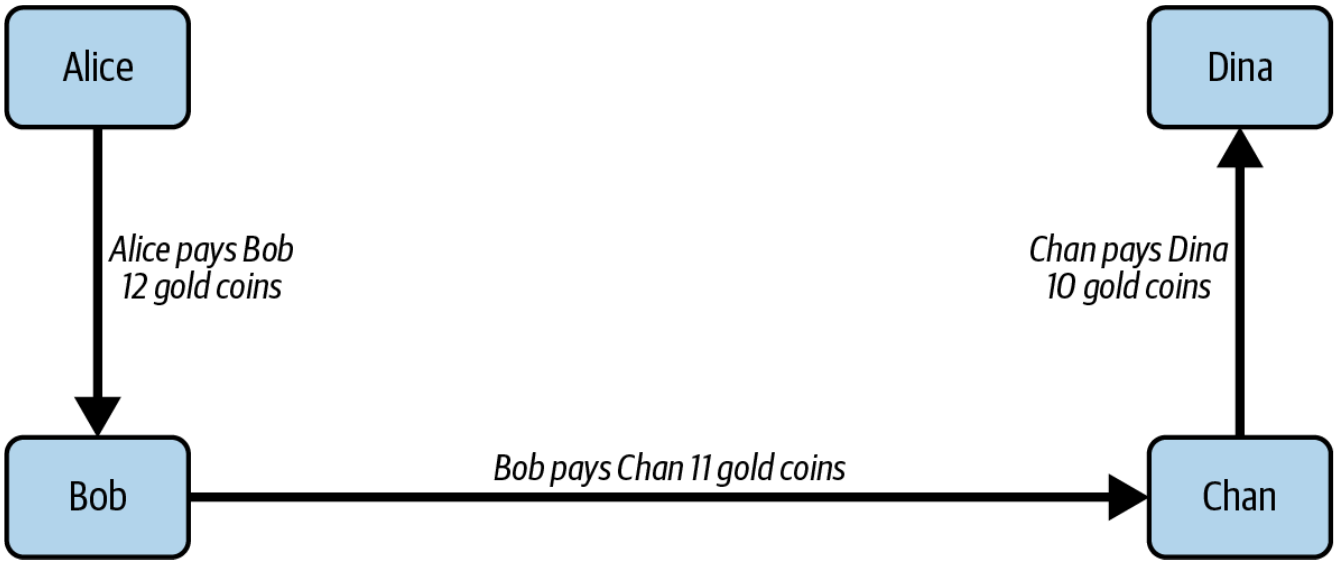

Alice
Dina
Alice pays Bob 12 gold coins
Chan pays Dina 10 gold coins
Bob pays Chan 11 gold coins
Bob
Chan

Figure 43. Alice pays Bob, Bob pays Chan, Chan pays Dina

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

This escrow service is an idealized one, which does not introduce other risks (e.g., counterparty risk). Later we will see how we can replace the escrow with a Bitcoin smart contract. Let's assume for now that everyone trusts this escrow service.

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 random number that is large enough to prevent others from guessing it (typically a *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 SHA-256 hash of the secret in the contract itself, as follows:

```
<ul class="simplelist">
<li><em>H</em> = SHA-256(<em>R</em>)</li>
</ul>
```

We call this hash of the payment's secret the *payment hash*. The secret that "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`. This secret message is called the *payment secret* or *payment preimage*.

To "commit" to this secret, Dina computes the SHA-256 hash, which when encoded in hexadecimal, can be displayed as follows: `0575965b3b44be51e8057d551c4016d83cb1fba9ea8d6e986447ba33fe69f6b3`.

To facilitate Alice's payment, Dina will create the payment secret and the payment hash, and send the payment hash to Alice. In Dina sends the hashed secret to Alice we see that Dina sends the payment hash to Alice via some external channel (dashed line), such as an email or text message.

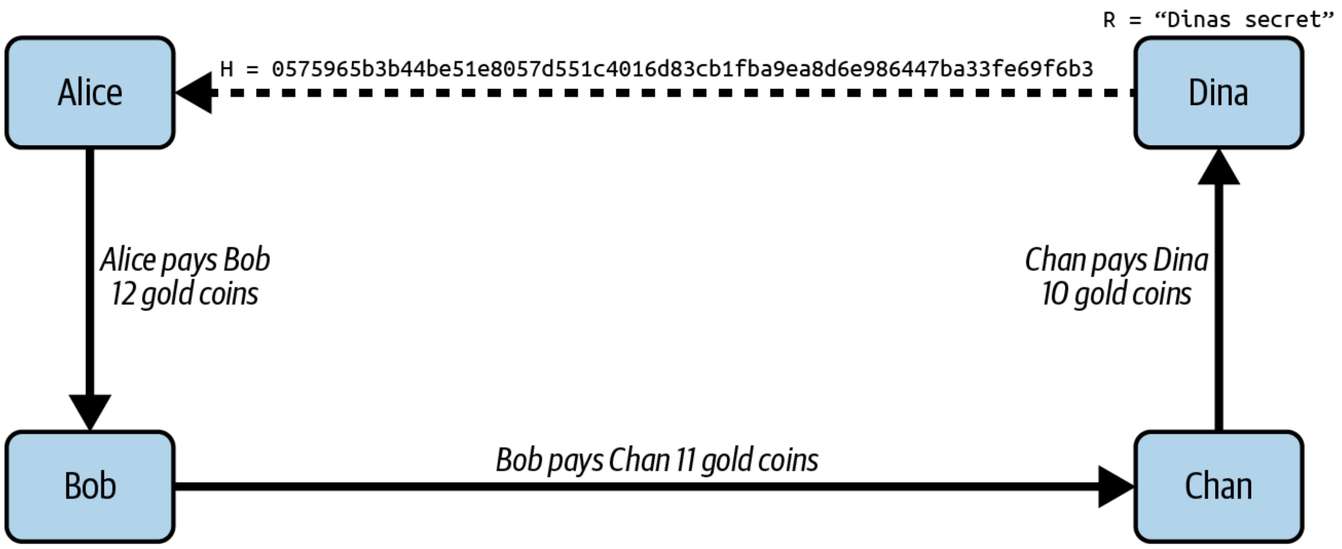

R = “Dinas secret”
H = 0575965b3b44be51e8057d551c4016d83cb1fba9ea8d6e986447ba33fe69f6b3
Alice
Dina
Alice pays Bob 12 gold coins
Chan pays Dina 10 gold coins
Bob
Bob pays Chan 11 gold coins
Chan

Figure 44. Dina sends the hashed secret to Alice

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. To assure you that you will be reimbursed, I will provide the 12 gold coins to a 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.

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. To assure you that you will be reimbursed, I will provide the 11 gold coins to a 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 gold coins if you can show me a valid message that hashes to:`057596…`. To assure you that you will be reimbursed after revealing the secret, I will provide the 10 gold coins to a 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 preimage to the hash she has established as proof of payment.

Dina now sends Dinas secret to Chan.

Chan checks that Dinas secret hashes to 057596.... Chan now has proof of payment and so instructs the escrow service to release the 10 gold 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 preimage, 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 would still have their money stuck in escrow permanently.

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 *timelock*.

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.

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 an even shorter deadline.

In turn, Bob will alter his contract with Chan as follows:

> *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 will also alter his contract with Dina:

> *Dina has 20 hours to show the secret after the contract was signed. If she does not provide the secret by this time, Chan'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 go 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, no party will have their coins stolen in the process.

The precondition 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 later in this chapter that this requirement is actually one of the more difficult issues for LN 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 preceding example.

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 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:

Trustless 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
: Either 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 *multipart payments* (*MPP*) and are explored further in Multipart Payments (MPP).

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 trade-offs among 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 preimage 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.

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 efficiency and better privacy. Efficient implementation of PTLCs depends on a new digital signature algorithm called *Schnorr signatures*, which is expected to be activated in Bitcoin in 2021.

Revisiting the Tipping 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 Lightning Payment Requests. 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 Spontaneous Payments. For now, we will explain the simpler payment flow using an invoice.

Alice visits Dina's site, enters the amount of 50,000 satoshis in a form, and in response, Dina's Lightning node generates a payment request for 50,000 satoshis in the form of a Lightning invoice. This interaction takes place over the web and outside the Lightning Network, as shown in Alice requests an invoice from Dina's website.

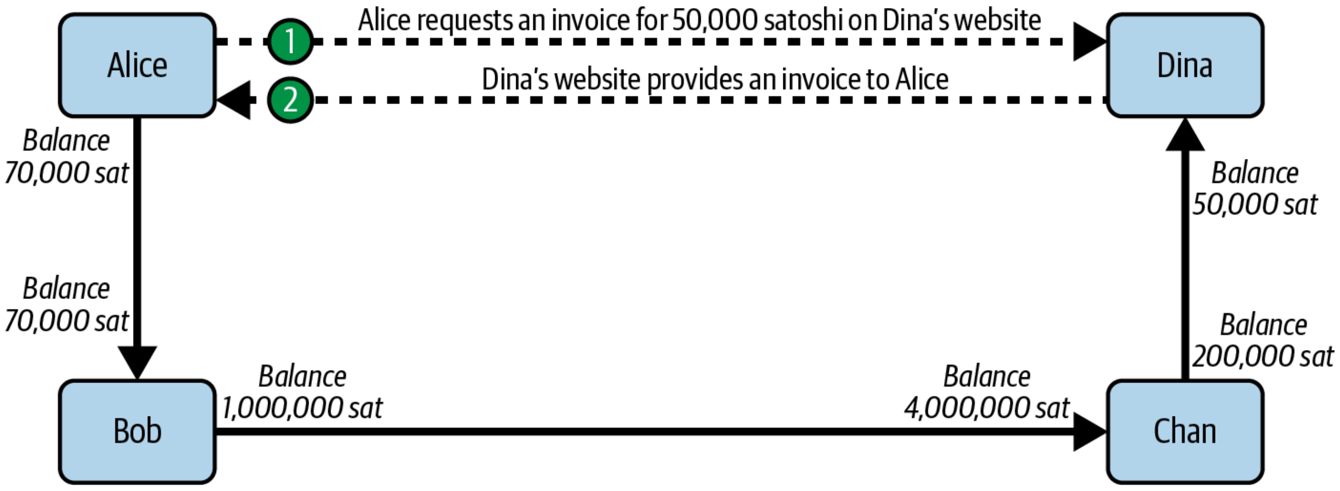


Figure 45. Alice requests an invoice from Dina's website

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 Pathfinding and Payment Delivery. 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.

NOTE Remember how Bob and Chan might expect a small compensation for routing the payment through their nodes? Alice wants to pay Dina 50,000 satoshis, but as you will see in the following sections she will send Bob 50,200 satoshis. The extra 200 satoshis will pay Bob and Chan 100 satoshis each, as a routing fee.

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 (HTLC) to pay Dina and how that HTLC is forwarded along the path from Alice to Dina.

On-Chain versus Off-Chain Settlement of HTLCs

The purpose of the Lightning Network is to enable *off-chain* transactions that are trusted just the same as on-chain transactions because no one can cheat. The reason no one can cheat is because at any time, any of the participants can take their off-chain transactions on-chain. Each off-chain transaction is ready to be submitted to the Bitcoin blockchain at any time. Thus, the Bitcoin blockchain acts as a dispute-resolution and final settlement mechanism if necessary.

The mere fact that any transaction can be taken on-chain at any time is precisely the reason that all those transactions can be kept off-chain. If you know you have recourse, you can continue to cooperate with the other participants and avoid the need for on-chain settlement and extra fees.

In all the examples that follow, we will assume that any of these transactions can be made on-chain at any time. The participants will choose to keep them off-chain, but there is no difference in the functionality of the system other than the higher fees and delay imposed by on-chain mining of the transactions. The example works the same if all the transactions are on-chain or off-chain.

Hash Time-Locked Contracts

In this section we explain how HTLCs work.

The first part of an HTLC 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 preimage, it's easy for anyone to verify the hash, and there's only one possible preimage that resolves the payment condition.

In Alice gets a payment hash from Dina we see Alice getting 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 preimage*. The payment hash acts as an identifier that can be used to route the payment to Dina. The payment preimage acts as a receipt and proof of payment once the payment is complete.

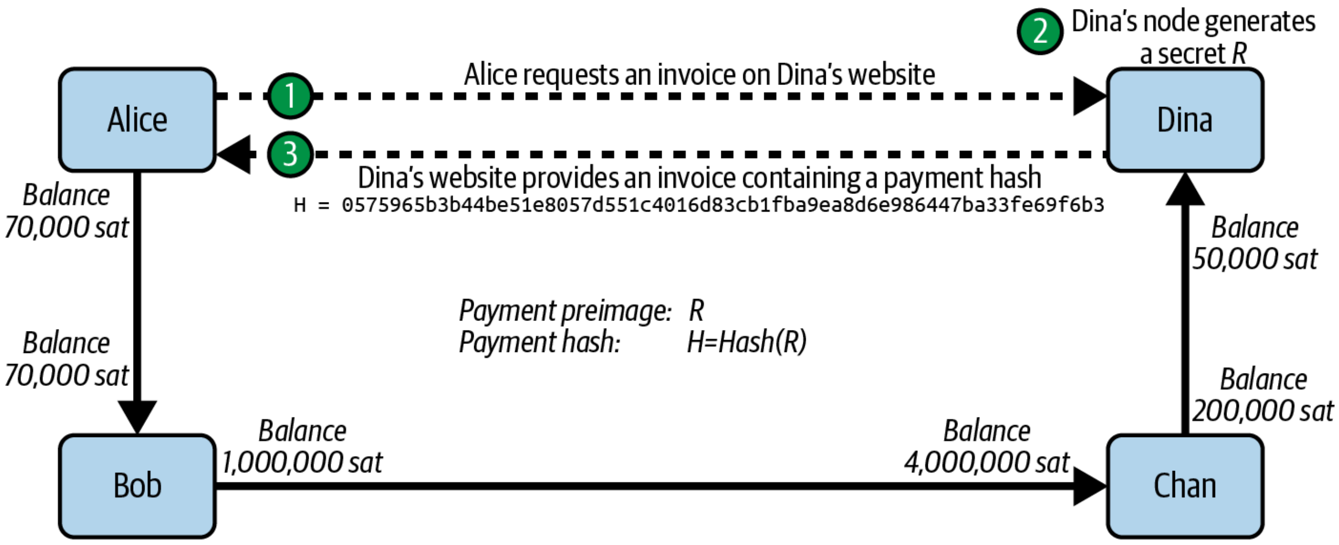

2 Dina's node generates a secret *R*
Alice
1 Alice requests an invoice on Dina's website
Dina
3 Dina's website provides an invoice containing a payment hash
H = 0575965b3b44be51e8057d551c4016d83cb1fba9ea8d6e986447ba33fe69f6b3
Balance 70,000 sat
Balance 50,000 sat
Payment preimage: R
Payment hash: H=Hash(R)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

Figure 46. Alice gets a payment hash from Dina

In the Lightning Network, Dina's payment preimage won't be a phrase like Dinas 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:

```
<ul class="simplelist">
<li><em>H</em> = SHA-256(<em>R</em>)</li>
</ul>
```

In this equation, *H* is the hash, or *payment hash* and *R* is the secret or *payment preimage*.

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 (BOLT #3) we see an HTLC Bitcoin Script as currently used in the Lightning Network. You can find this definition in BOLT #3, Transactions.

Example 3. HTLC implemented in Bitcoin Script (BOLT #3)

```
# 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
```

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

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

The main part of the HTLC is in line 10 of HTLC implemented in Bitcoin Script (BOLT #3). Let's build it up from scratch!

Payment Preimage and Hash Verification

The core of an HTLC is the hash, where payment can be made if the recipient knows the payment preimage. Alice locks the payment to a specific payment hash, and Bob has to present a payment preimage to claim the funds. The Bitcoin system can verify that Bob's payment preimage 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 fulfillment 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 secret preimage value *R*.

The unlocking script combined with the locking script would 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.

In Propagating the HTLC across the network, we see the HTLC propagated across the network from

Alice to Dina. 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. This is a really important point because it ensures end-to-end *atomicity* of the HTLC. To spend the HTLC, one needs to reveal the secret, which then makes it possible for others to spend their HTLC also. Either all the HTLCs are spendable, or none of the HTLCs are spendable: atomicity!

Because Alice's HTLC is 100 satoshi more than 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.

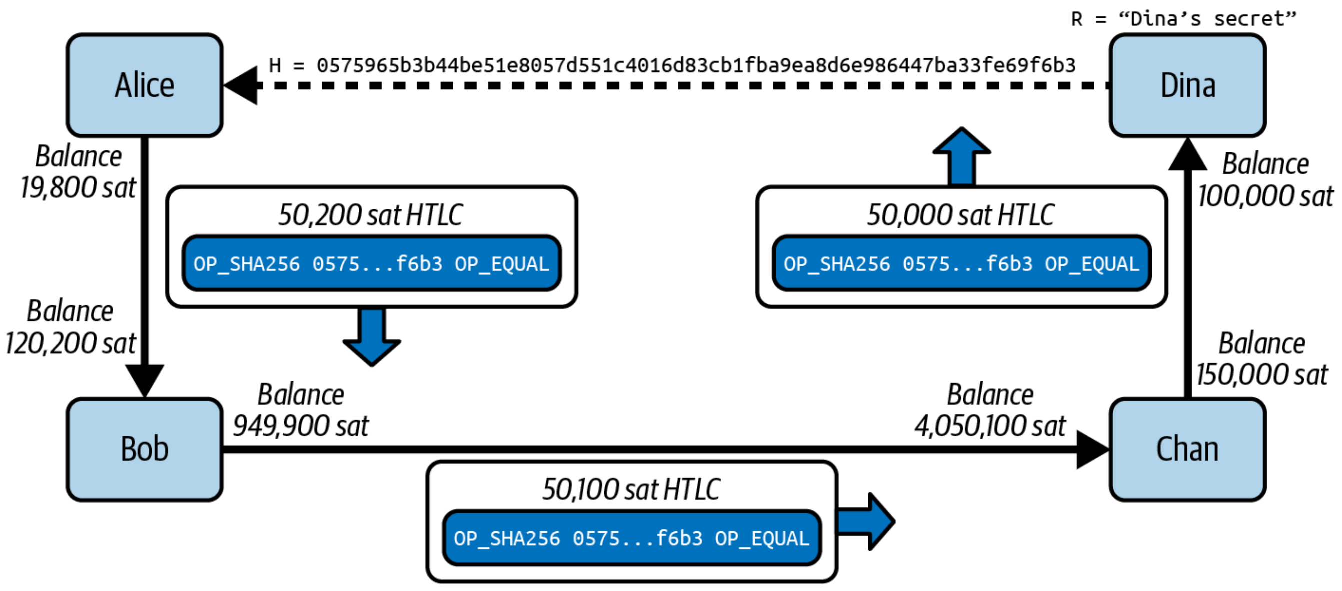


Figure 47. Propagating the HTLC across the network

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. In Dina settles Chan's HTLC off-chain we see Dina giving the secret to Chan, thereby fulfilling the HTLC.

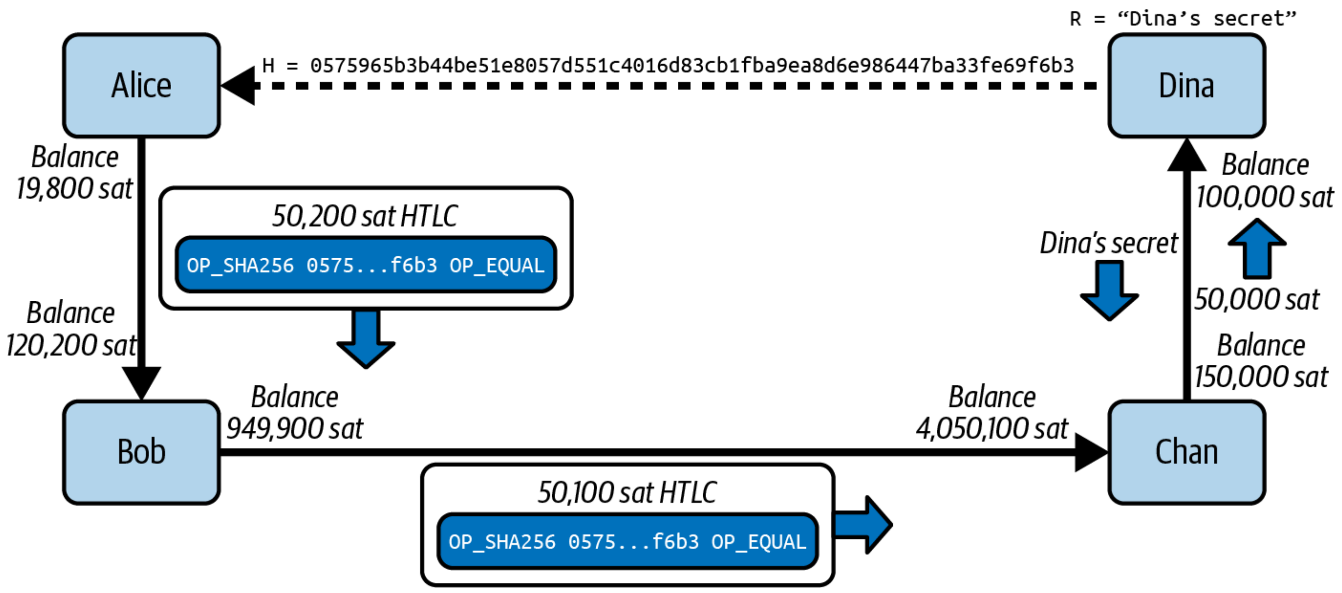


Figure 48. Dina settles Chan's HTLC off-chain

Notice that 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.

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. In Chan settles Bob's HTLC off-chain we see Chan sending the secret to Bob and receiving a payment in return.

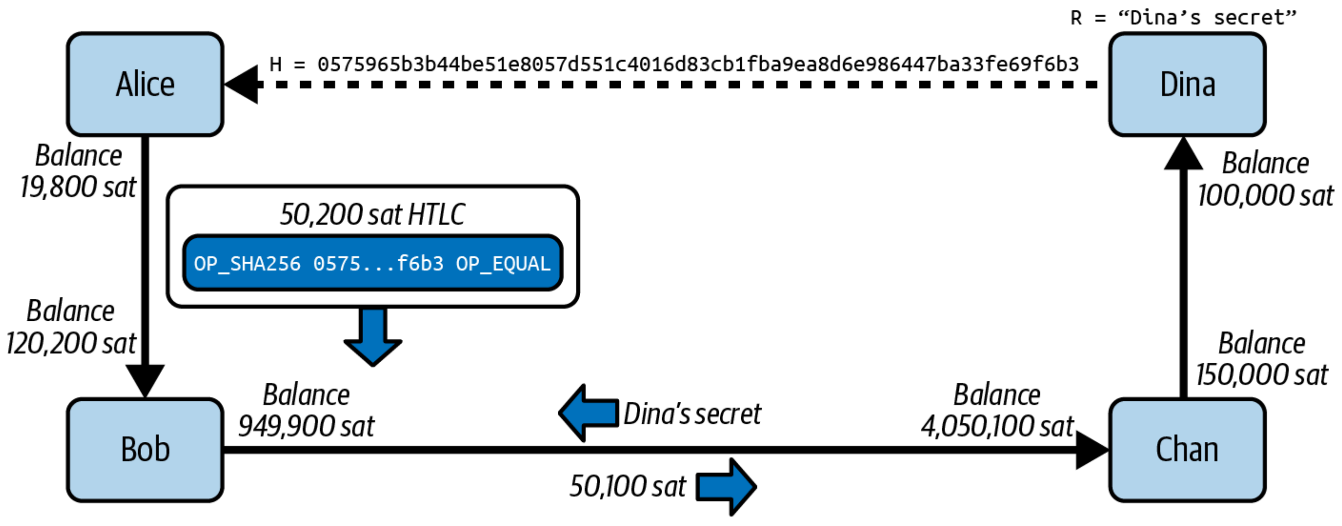


Figure 49. Chan settles Bob's HTLC off-chain

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.

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. In Bob settles Alice's HTLC off-chain we see that Bob sends the secret to Alice and they update the channel balance to reflect a 50,200 satoshi Lightning payment from Alice to Bob.

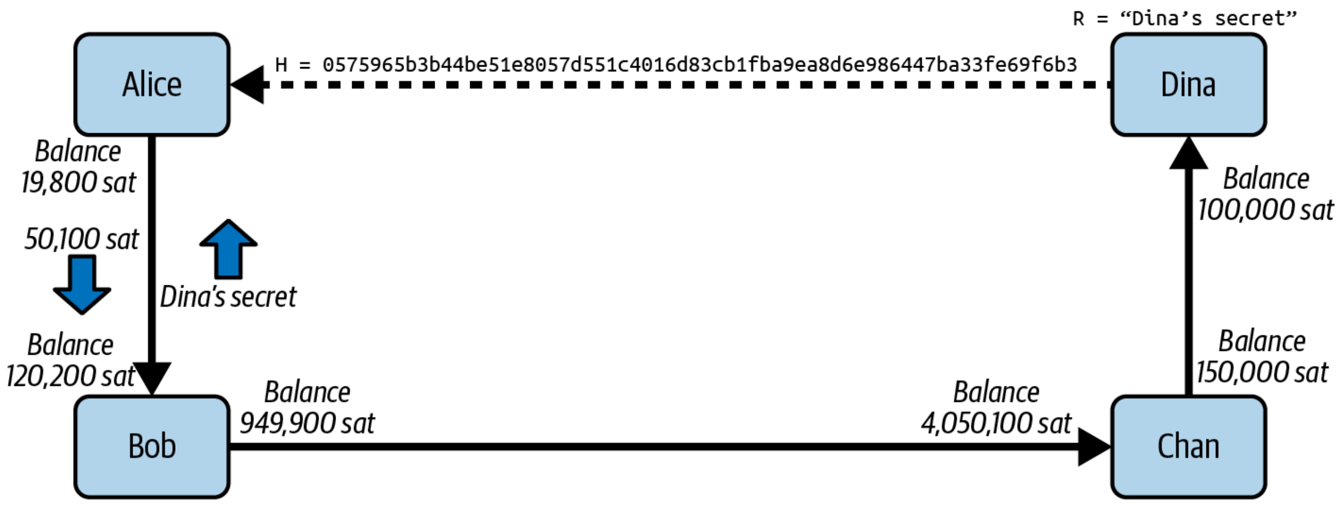


Figure 50. Bob settles Alice’s HTLC off-chain

Bob has received 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.

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, as shown in Channel balances after the payment.

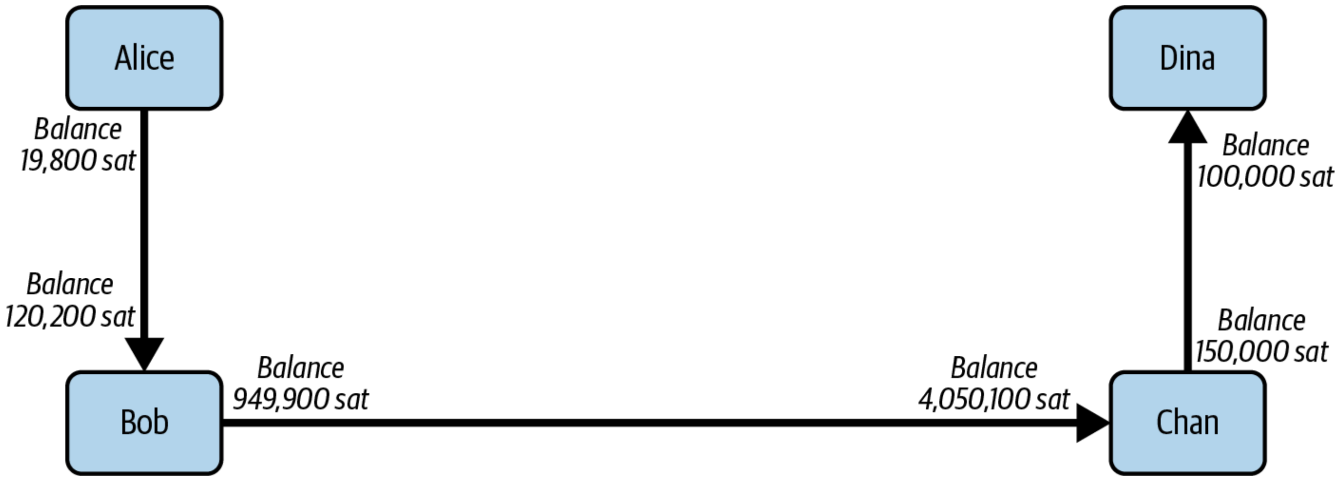


Figure 51. Channel balances after the payment

Signature Binding: Preventing Theft of HTLCs

There’s a catch. Did you notice it?

If Alice, Bob, and Chan create the HTLCs as shown in Channel balances after the payment, they face a small but not insignificant risk of loss. Any of those HTLCs can be redeemed (spent) by anyone who knows the secret. At first only Dina knows the secret. Dina is supposed to only spend the HTLC from Chan. But Dina could spend all three HTLCs at the same time, or even in a single spending transaction! After all, Dina knows the secret before anyone else. Similarly, once Chan knows the secret, he is only supposed to spend the HTLC offered by Bob. But what if Chan also spends Alice’s offered HTLC?

This is not *trustless*! It fails the most important security feature. We need to fix this.

The HTLC script must have an additional condition that binds each HTLC to a specific recipient. We do this by requiring a digital signature that matches the public key of each recipient, thereby preventing anyone else from spending that HTLC. Since only the designated recipient has the ability to produce a digital signature matching that public key, only the designated recipient can spend that HTLC.

Let’s look at the scripts again with this modification in mind. Alice’s HTLC for Bob is modified to include Bob’s public key and the OP_CHECKSIG operator.

Here’s the modified HTLC script:

```
OP_SHA256 <H> OP_EQUALVERIFY <Bob's Pub> OP_CHECKSIG
```

TIP Notice that we also changed OP_EQUAL to OP_EQUALVERIFY. When an operator has the suffix VERIFY, it does not return TRUE or FALSE on the stack. Instead, it *halts* execution and fails the script if the result is false and continues without any stack output if it is true.

To redeem this HTLC, Bob has to present an unlocking script that includes a signature from Bob’s private key as well as the secret payment preimage, like this:

```
<Bob's Signature> <R>
```

The unlocking and locking scripts are combined and evaluated by the scripting engine, as follows:

```
<Bob's Sig> <R> OP_SHA256 <H> OP_EQUALVERIFY <Bob's Pub> OP_CHECKSIG
```

1. <Bob's Sig> is pushed to the stack.
2. R is pushed to the stack.
3. OP_SHA256 pops and hashes R from the top of the stack and pushes H~R~ to the stack.
4. H is pushed to the stack.
5. OP_EQUALVERIFY pops H and H~R~ and compares them. If they are not the same, execution halts. Otherwise, we continue without output to the stack.
6. <Bob's Pub> key is pushed to the stack.
7. OP_CHECKSIG pops <Bob's Sig> and <Bob's Pub> and verifies the signature. The result (`TRUE/FALSE`) is pushed to the stack.

As you can see, this is slightly more complicated, but now we have fixed the HTLC and made sure only the intended recipient can spend it.

Hash Optimization

Let’s look at the first part of the HTLC script so far:

```
OP_SHA256 <H> OP_EQUALVERIFY
```

If we look at this in the preceding symbolic representation, it looks like the OP_ operators take up the most space. But that's not the case. Bitcoin Script is encoded in binary, with each operator representing one byte. Meanwhile, the <H> value we use as a placeholder for the payment hash is a 32-byte (256-bit) value. You can find a listing of all the Bitcoin Script operators and their binary and hex encoding in Bitcoin Wiki: Script, or in Appendix D, "Transaction Script Language Operators, Constants, and Symbols," in *Mastering Bitcoin*.

Represented in hexadecimal, our HTLC script would look like this:

```
a8 0575965b3b44be51e8057d551c4016d83cb1fba9ea8d6e986447ba33fe69f6b3 88
```

In hexadecimal encoding, OP_SHA256 is a8 and OP_EQUALVERIFY is 88. The total length of this script is 34 bytes, of which 32 bytes are the hash.

As we've mentioned previously, any participant in the Lightning Network should be able to take an off-chain transaction they hold and put it on-chain if they need to enforce their claim to funds. To take a transaction on-chain, they'd have to pay transaction fees to the miners, and these fees are proportional to the size, in bytes, of the transaction.

Therefore, we want to find ways to minimize the on-chain "weight" of transactions by optimizing the script as much as possible. One way to do that is to add another hash function on top of the SHA-256 algorithm, one that produces smaller hashes. The Bitcoin Script language provides the OP_HASH160 operator that "double hashes" a preimage: first the preimage is hashed with SHA-256, and then the resulting hash is hashed again with the RIPEMD160 hash algorithm. The hash resulting from RIPEMD160 is 160 bits or 20 bytes—much more compact. In Bitcoin Script this is a very common optimization that is used in many of the common address formats.

So, let's use that optimization instead. Our SHA-256 hash is 057596...69f6b3. Putting that through another round of hashing with RIPEMD160 gives us the result:

```
R = "Dinas secret"
H256 = SHA256(R)
H256 = 0575965b3b44be51e8057d551c4016d83cb1fba9ea8d6e986447ba33fe69f6b3
H160 = RIPEMD160(H256)
H160 = 9e017f6767971ed7cea17f98528d5f5c0ccb2c71
```

Alice can calculate the RIPEMD160 hash of the payment hash that Dina provides and use the shorter hash in her HTLC, as can Bob and Chan!

The "optimized" HTLC script would look like this:

```
OP_HASH160 <H160> OP_EQUALVERIFY
```

Encoded in hex, this is:

```
a9 9e017f6767971ed7cea17f98528d5f5c0ccb2c71 88
```

Where OP_HASH160 is a9 and OP_EQUALVERIFY is 88. This script is only 22 bytes long! We've saved 12 bytes from every transaction that redeems an HTLC on-chain.

With that optimization, you now see how we arrive at the HTLC script shown in line 10 of HTLC implemented in Bitcoin Script (BOLT #3):

```
...
    # To local node via HTLC-success transaction.
    OP_HASH160 <RIPEMD160(payment_hash)> OP_EQUALVERIFY...
```

HTLC Cooperative and Timeout Failure

So far we looked at the "hash" part of HTLC and how it would work if everyone cooperated and was online at the time of payment.

What happens if someone goes offline or fails to cooperate? What happens if the payment cannot succeed?

We need to ensure a way to "fail gracefully," because occasional routing failures are inevitable. There are two ways to fail: cooperatively and with a time-locked refund.

Cooperative failure is relatively simple: the HTLC is unwound by every participant in the route, removing the HTLC output from their commitment transactions without changing the balance. We'll look at how that works in detail in Channel Operation and Payment Forwarding.

Let's look at how we can reverse an HTLC without the cooperation of one or more participants. We need to make sure that if one of the participants does not cooperate, the funds are not simply locked in the HTLC *forever*. This would give someone the opportunity to ransom the funds of another participant: "I'll leave your funds tied up forever if you don't pay me ransom."

To prevent this, every HTLC script includes a refund clause that is connected to a timelock. Remember our original escrow contract? "Bob has 24 hours to show the secret after the contract is signed. If Bob does not provide the secret by this time, Alice's deposit will be refunded."

The time-locked refund is an important part of the script that ensures *atomicity*, so that the entire end-to-end payment either succeeds or fails gracefully. There is no "half paid" state to worry about. If there is a failure, every participant can either unwind the HTLC cooperatively with their channel partner or put the time-locked refund transaction on-chain unilaterally to get their money back.

To implement this refund in Bitcoin Script, we use a special operator `OP_CHECKLOCKTIMEVERIFY` also known OP_CLTV for short. Here's the script, as seen previously in line 13 of HTLC implemented in Bitcoin Script (BOLT #3):

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

The OP_CLTV operator takes an expiry time defined as the block height after which this transaction is valid. If the transaction timelock is not set the same as <cltv_expiry>, the evaluation of the script fails and the transaction is invalid. Otherwise, the script continues without any output to the stack. Remember, the VERIFY suffix means this operator does not output TRUE or FALSE but instead either halts/fails or continues without stack output.

Essentially, the OP_CLTV acts as a "gatekeeper" preventing the script from proceeding any further if the <cltv_expiry> block height has not been reached on the Bitcoin blockchain.

The OP_DROP operator simply drops the topmost item on the script stack. This is necessary in the beginning because there is a "leftover" item from the previous script lines. It is necessary *after* OP_CLTV to remove the <cltv_expiry> item from the top of the stack because it is no longer necessary.

Finally, once the stack has been cleaned up, there should be a public key and signature left behind that OP_CHECKSIG can verify. As we saw in Signature Binding: Preventing Theft of HTLCs, this is necessary to ensure that only the rightful owner of the funds can claim them, by binding this output to their public key and requiring a signature.

Decrementing Timelocks

As the HTLCs are extended from Alice to Dina, the time-locked refund clause in each HTLC has a *different* cltv_expiry value. We will see this in more detail in Onion Routing. But suffice it to say that to ensure an orderly unwinding of a payment that fails, each hop needs to wait a bit less for their refund. The difference between timelocks for each hop is called the cltv_expiry_delta, and is set by each node and advertised to the network, as we will see in Gossip and the Channel Graph.

For example, Alice sets the refund timelock on the first HTLC to a block height of current + 500 blocks ("current" being the current block height). Bob would then set the timelock cltv_expiry on the HTLC to Chan to current + 450 blocks. Chan would set the timelock to current + 400 blocks from the current block height. This way, Chan can get a refund on the HTLC he offered to Dina *before* Bob gets a refund on the HTLC he offered to Chan. Bob can get a refund of the HTLC he offered to Chan before Alice can get a refund for the HTLC she offered to Bob. The decrementing timelock prevents race conditions and ensures the HTLC chain is unwound backward, from the destination toward the origin.

Conclusion

In this chapter we saw how Alice can pay Dina even if she doesn't have a direct payment channel. Alice can find a path that connects her to Dina and route a payment across several payment channels so that it reaches Dina.

To ensure that the payment is atomic and trustless across multiple hops, Alice must implement a fairness protocol in cooperation with all the intermediary nodes in the path. The fairness protocol is currently implemented as an HTLC, which commits funds to a payment hash derived from a secret payment preimage.

Each of the participants in the payment route can extend an HTLC to the next participant, without worrying about theft or stuck funds. The HTLC can be redeemed by revealing the secret payment preimage. Once an HTLC reaches Dina, she reveals the preimage, which flows backward, resolving all the HTLCs offered.

Finally, we saw how a time-locked refund clause completes the HTLC, ensuring that every participant can get a refund if the payment fails but for whatever reason one of the participants doesn’t cooperate in unwinding the HTLCs. By always having the option to go on-chain for a refund, the HTLC achieves the fairness goal of atomicity and trustless operation.

Channel Operation and Payment Forwarding

In this chapter we will bring together payment channels and hash time-locked contracts (HTLCs). In Payment Channels we explained the way Alice and Bob construct a payment channel between their two nodes. We also looked at the commitment and penalty mechanisms that secure the payment channel. In Routing on a Network of Payment Channels we looked at hash time-locked contracts (HTLCs) and how these can be used to route a payment across a path made of multiple payment channels. In this chapter we bring the two concepts together by looking at how HTLCs are managed on each payment channel, how the HTLCs are committed to the channel state, and how they are settled to update the channel balances.

Specifically, we will be discussing "Adding, settling, failing HTLCs" and the "Channel state machine" that form the overlap between the peer-to-peer layer and the routing layer, as highlighted by an outline in Channel operation and payment forwarding in the Lightning protocol suite.

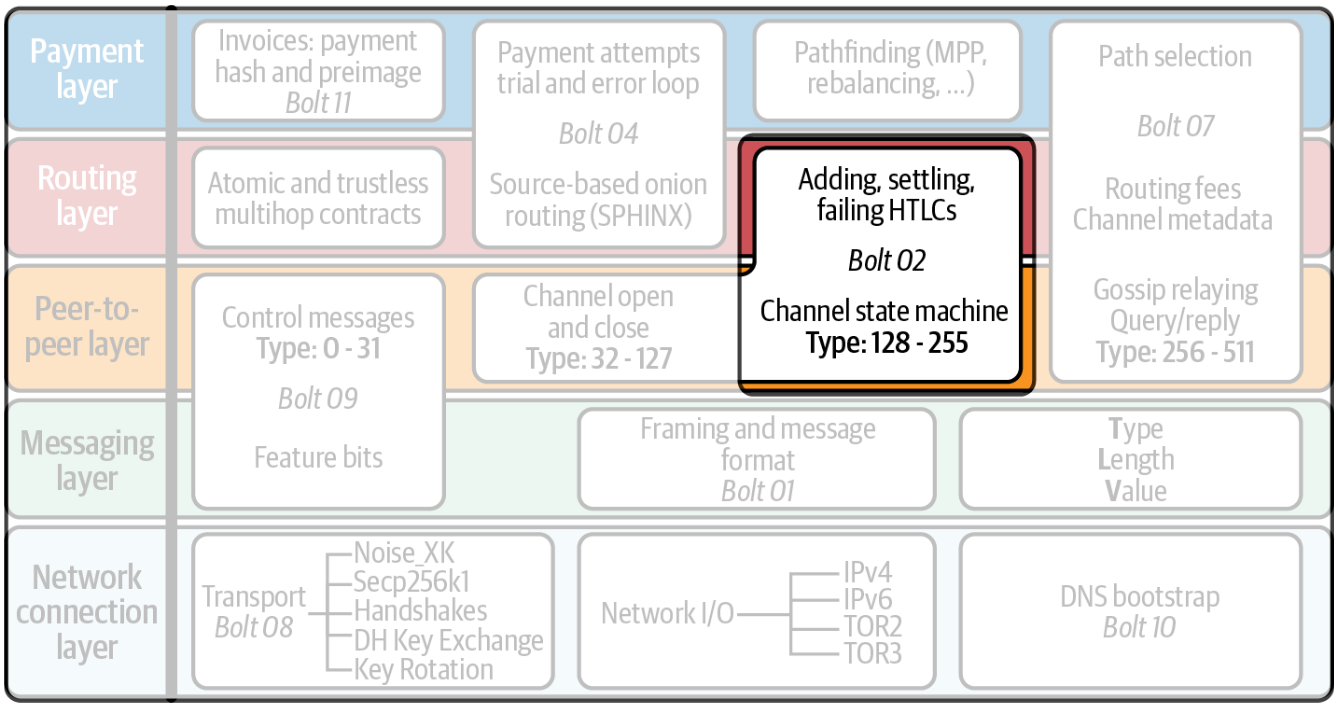


Figure 52. Channel operation and payment forwarding in the Lightning protocol suite

Local (Single Channel) versus Routed (Multiple Channels)

Even though it is possible to send payments across a payment channel simply by updating the channel balances and creating new commitment transactions, the Lightning protocol uses HTLCs even for "local" payments across a payment channel. The reason for this is to maintain the same protocol design regardless of whether a payment is only one hop (across a single payment channel) or several hops (routed across multiple payment channels).

By maintaining the same abstraction for both local and remote, we not only simplify the protocol design but also improve privacy. For the recipient of a payment there is no discernible difference between a payment made directly by their channel partner and a payment forwarded by their channel partner on behalf of someone else.

Forwarding Payments and Updating Commitments with HTLCs

We will revisit our example from Routing on a Network of Payment Channels to demonstrate how the HTLCs from Alice to Dina get committed to each payment channel. As you recall in our example, Alice is paying Dina 50,000 satoshis by routing an HTLC via Bob and Chan. The network is shown in Alice pays Dina with an HTLC routed via Bob and Chan.

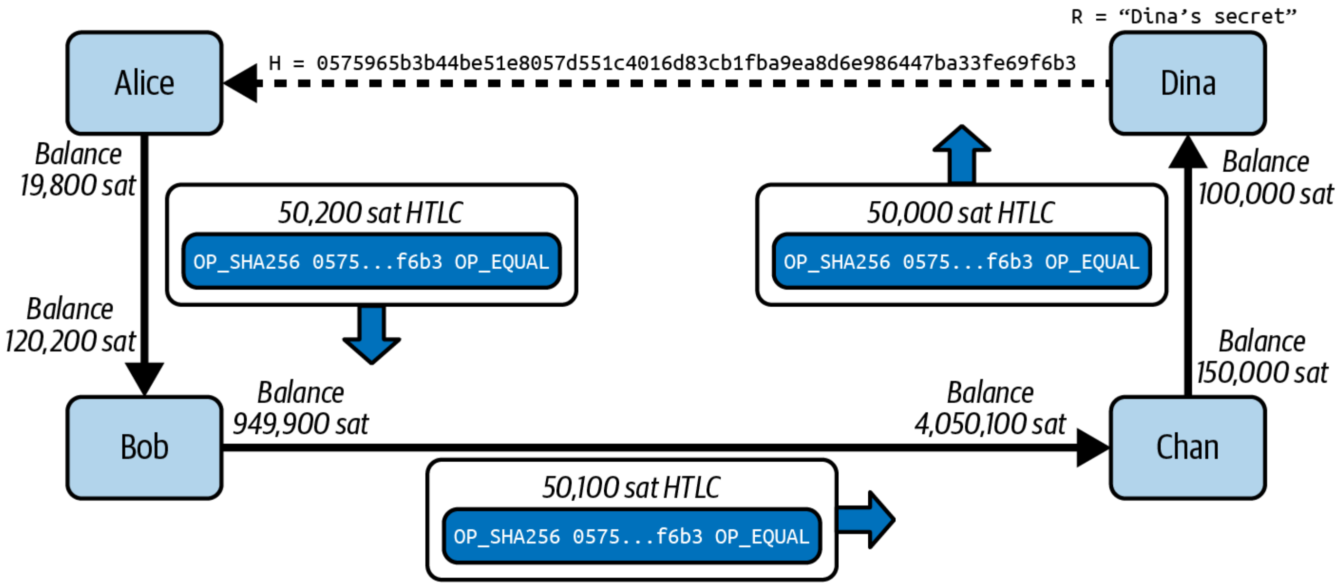


Figure 53. Alice pays Dina with an HTLC routed via Bob and Chan

We will focus on the payment channel between Alice and Bob and review the messages and transactions that they use to process this HTLC.

HTLC and Commitment Message Flow

The message flow between Alice and Bob (and also between any pair of channel partners) is shown in The message flow for HTLC commitment between channel partners.

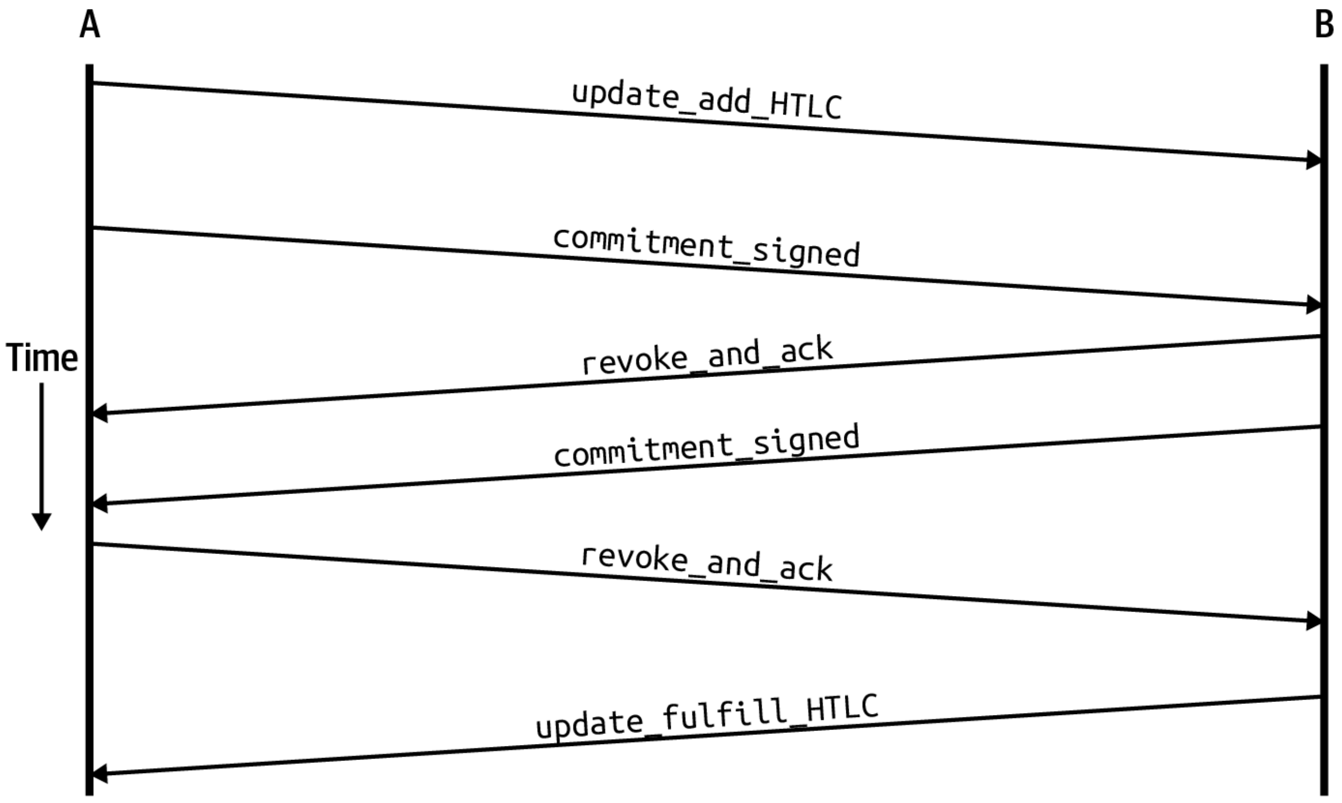


Figure 54. The message flow for HTLC commitment between channel partners

We've already seen the commitment_signed and revoke_and_ack in Payment Channels. Now we will see how HTLCs fit into the commitment scheme. The two new messages are update_add_htlc, which Alice uses to ask Bob to add an HTLC, and update_fulfill_htlc, which Bob uses to redeem the HTLC once he has received the payment secret (Dina's secret).

Forwarding Payments with HTLCs

Alice and Bob start with a payment channel that has a 70,000 satoshi balance on each side.

As we saw in Payment Channels, this means that Alice and Bob have negotiated and each hold commitment transactions. These commitment transactions are asymmetric, delayed, and revocable, and look like the example in Alice and Bob's initial commitment transactions.

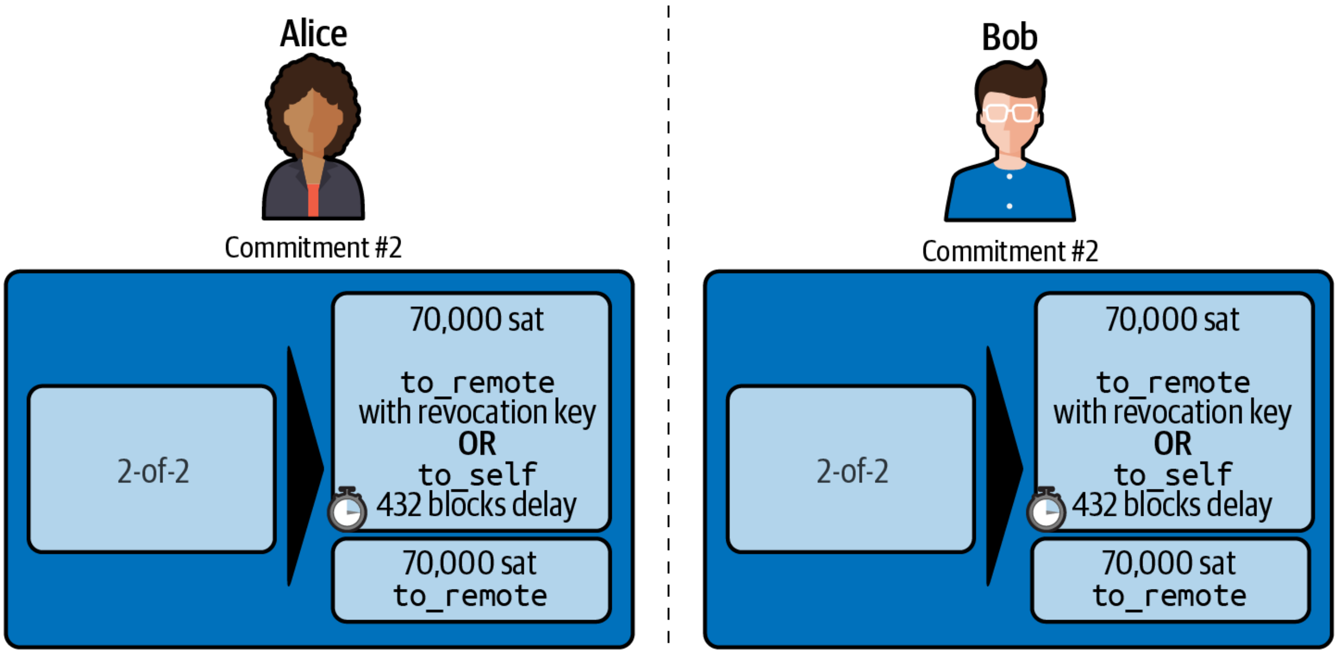


Figure 55. Alice and Bob's initial commitment transactions

Adding an HTLC

Alice wants Bob to accept an HTLC worth 50,200 satoshis to forward to Dina. To do so, Alice must send the details of this HTLC, including the payment hash and amount, to Bob. Bob will also need to know where to forward it, which is something we discuss in detail in Onion Routing.

To add the HTLC, Alice starts the flow we saw in The message flow for HTLC commitment between channel partners by sending the update_add_htlc message to Bob.

The update_add_HTLC Message

Alice sends the `update_add_HTLC` Lightning message to Bob. This message is defined in BOLT #2: Peer Protocol, `update_add_HTLC`, and is shown in Example 9-1.

Example 4. The `update_add_HTLC` message

```
[channel_id:channel_id]
[u64:id]
[u64:amount_msat]
[sha256:payment_hash]
[u32:cltv_expiry]
[1366*byte:onion_routing_packet]
```

channel_id

This is the channel that Alice has with Bob where she wants to add the HTLC. Remember that Alice and Bob may have multiple channels with each other.

id

This is an HTLC counter and starts at 0 for the first HTLC offered to Bob by Alice and is incremented for each subsequent offered HTLC.

amount_msat

This is the amount (value) of the HTLC in millisatoshis. In our example this is 50,200,000 millisatoshis (i.e., 50,200 satoshis).

payment_hash

This is the payment hash calculated from Dina's invoice. It is *H* = RIPEMD160(SHA-256(*R*)), where *R* is Dina's secret that is known only by Dina and will be revealed if Dina is paid.

cltv_expiry

This is the expiry time for this HTLC, which will be encoded as a time-locked refund in case the HTLC fails to reach Dina in this time.

onion_routing_packet

This is an onion-encrypted route that tells Bob where to forward this HTLC next (to Chan). Onion routing is covered in detail in Onion Routing.

TIP As a reminder, accounting within the Lightning Network is in units of millisatoshis (thousandths of a satoshi), whereas Bitcoin accounting is in satoshis. Any amounts in HTLCs are millisatoshis, which are then rounded to the nearest satoshi in the Bitcoin commitment transactions.

HTLC in Commitment Transactions

The received information is enough for Bob to create a new commitment transaction. The new commitment transaction has the same two outputs to_self and to_remote for Alice and Bob's balance, and a *new* output representing the HTLC offered by Alice.

We've already seen the basic structure of an HTLC in Routing on a Network of Payment Channels. The complete script of an offered HTLC is defined in BOLT #3: Transactions, Offered HTLC Output and is shown in Offered HTLC output script.

Example 5. Offered HTLC output script

```
# Revocation ①
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
        # Redemption ②
        OP_HASH160 <RIPEMD160(payment_hash)> OP_EQUALVERIFY
        2 OP_SWAP <local_HTLCpubkey> 2 OP_CHECKMULTISIG
    OP_ELSE
        # Refund ③
        OP_DROP <cltv_expiry> OP_CHECKLOCKTIMEVERIFY OP_DROP
        OP_CHECKSIG
    OP_ENDIF
OP_ENDIF
```

① The first clause of the `OP_IF` conditional is redeemable by Alice with a revocation key. If this commitment is later revoked, Alice will have a revocation key to claim this output in a penalty transaction, taking the whole channel balance.

② The second clause is redeemable by the preimage (payment secret, or in our example, Dina's secret) if it is revealed. This allows Bob to claim this output if he has the secret from Dina, meaning he has successfully delivered the payment to Dina.

③ The third and final clause is a refund of the HTLC to Alice if the HTLC expires without reaching Dina. It is time-locked with the expiration cltv_expiry. This ensures that Alice's balance is not "stuck" in an HTLC that can't be routed to Dina.

There are three ways to claim this output. Try to read the script and see if you can figure it out (remember, it is a stack-based language, so things appear "backward").

New Commitment with HTLC Output

Bob now has the necessary information to add this HTLC script as an additional output and create a new commitment transaction. Bob's new commitment will have 50,200 satoshis in the HTLC output. That amount will come from Alice's channel balance, so Alice's new balance will be 19,800 satoshis (70,000 – 50,200 = 19,800). Bob constructs this commitment as a tentative "Commitment #3," shown in Bob's new commitment with an HTLC output.

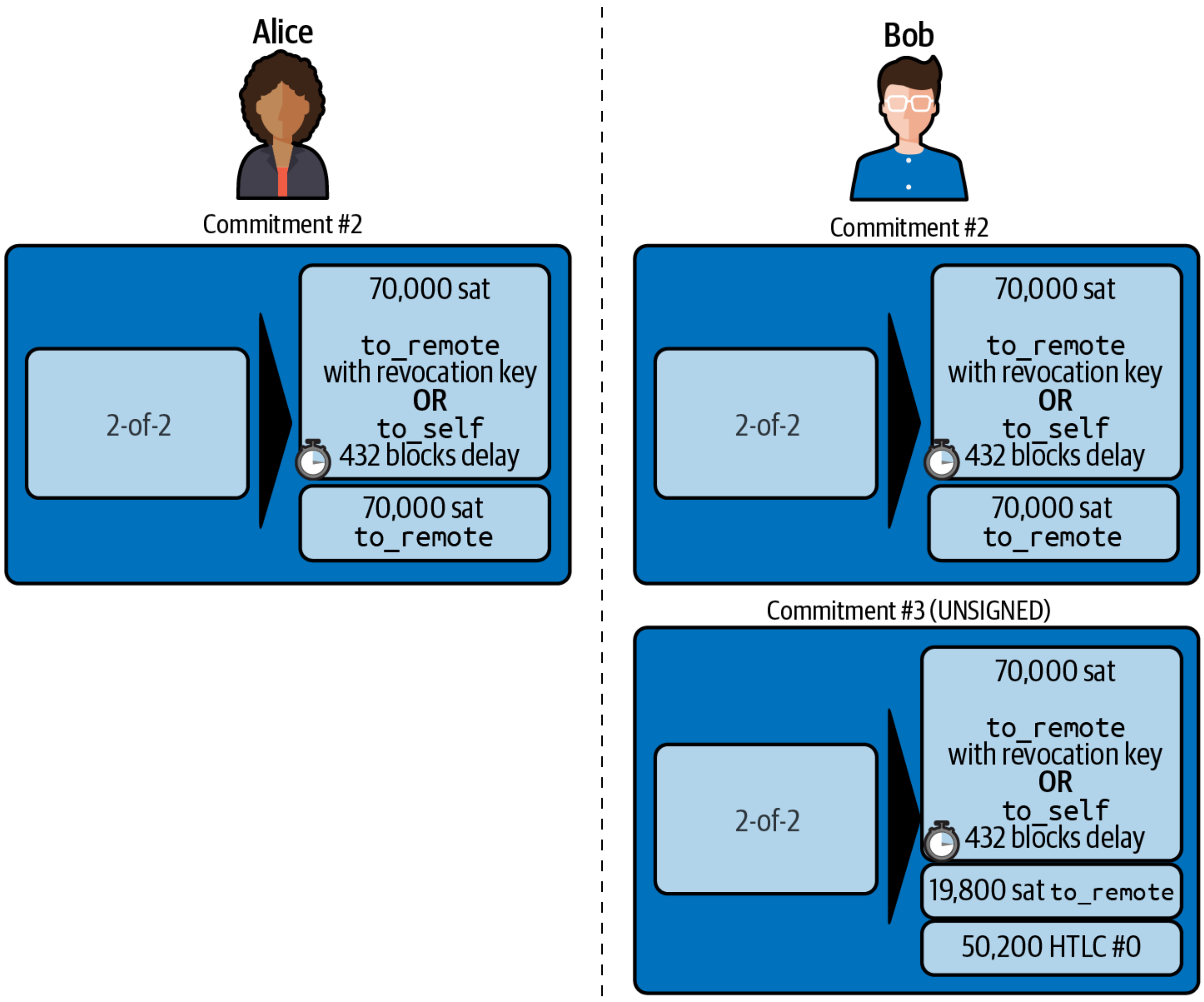


Figure 56. Bob's new commitment with an HTLC output

Alice Commits

Shortly after sending the update_add_htlc message, she will commit to the new state of the channel, so that the HTLC can be safely added by Bob. Bob has the HTLC information and has constructed a new commitment but does not yet have this new commitment signed by Alice.

Alice sends commitment_signed to Bob, with the signature for the new commitment and for the HTLC within. We saw the commitment_signed message in Payment Channels, but now we can understand the rest of the fields. As a reminder, it is shown in Example 9-3.

Example 6. The `commitment_signed` message

```
[channel_id:channel_id]
[signature:signature]
[u16:num_htlcs]
[num_htlcs*signature:htlc_signature]
```

The fields num_htlcs and htlc_signature now make more sense:

num_htlcs

This is the number of HTLCs that are outstanding in the commitment transaction. In our example, just one HTLC, the one Alice offered.

htlc_signature

This is an array of signatures (num_htlcs in length), containing signatures for the HTLC outputs.

Alice can send these signatures without hesitation: she can always get a refund if the HTLC expires without being routed to Dina.

Now, Bob has a new signed commitment transaction, as shown in Bob has a new signed commitment.

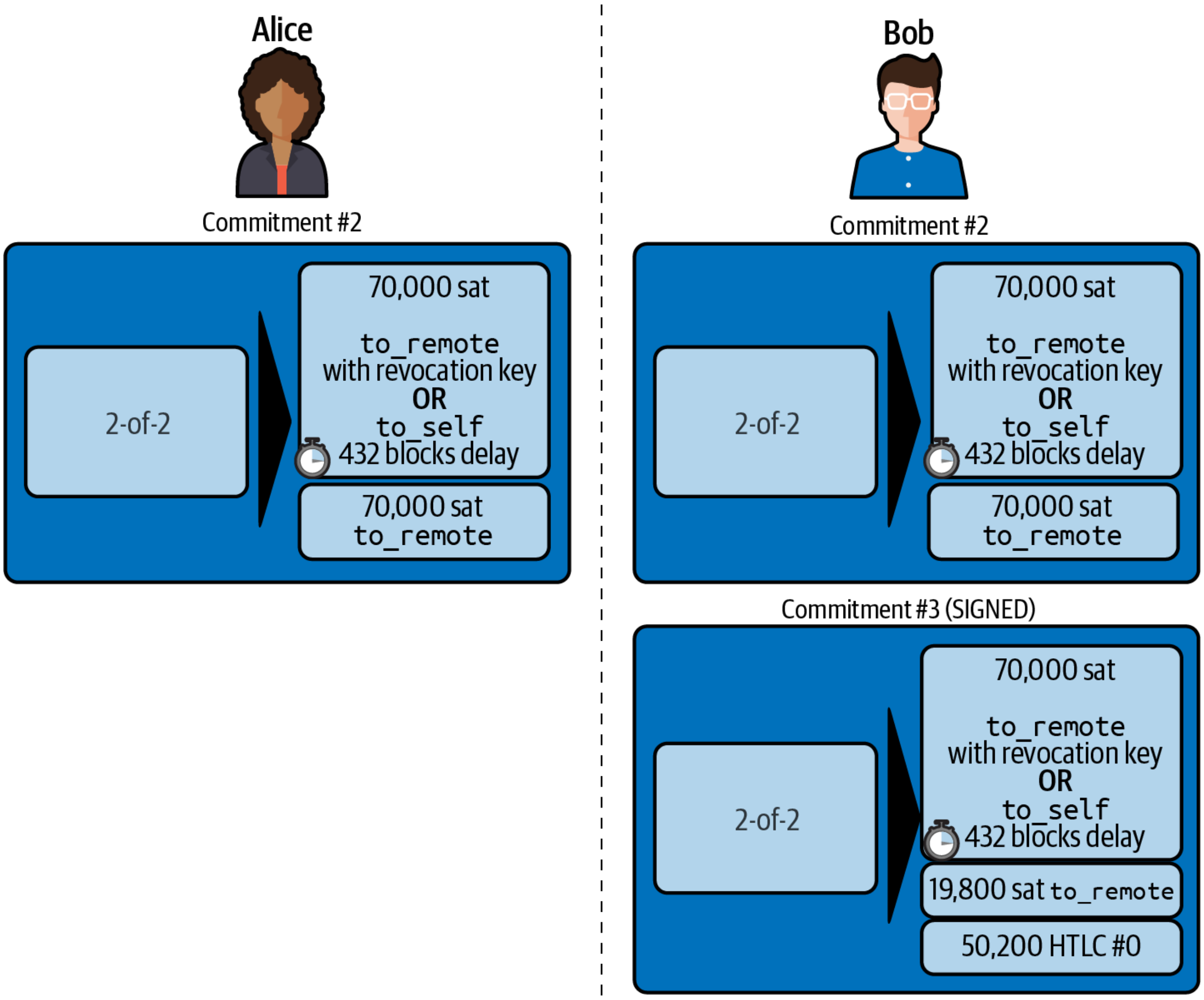


Figure 57. Bob has a new signed commitment

Bob Acknowledges New Commitment and Revokes Old One

Now that Bob has a new signed commitment, he needs to acknowledge it and revoke the old commitment. He does so by sending the revoke_and_ack message, as we saw in Payment Channels previously. As a reminder, that message is shown in The revoke_and_ack message.

Example 7. The revoke_and_ack message

```
[channel_id:channel_id]
[32*byte:per_commitment_secret]
[point:next_per_commitment_point]
```

Bob sends the per_commitment_secret that allows Alice to construct a revocation key to build a penalty transaction spending Bob's old commitment. Once Bob has sent this, he can never publish "Commitment #2" without risking a penalty transaction and losing all his money. So, the old commitment is effectively revoked.

Bob has effectively moved the channel state forward, as shown in Bob has revoked the old commitment.

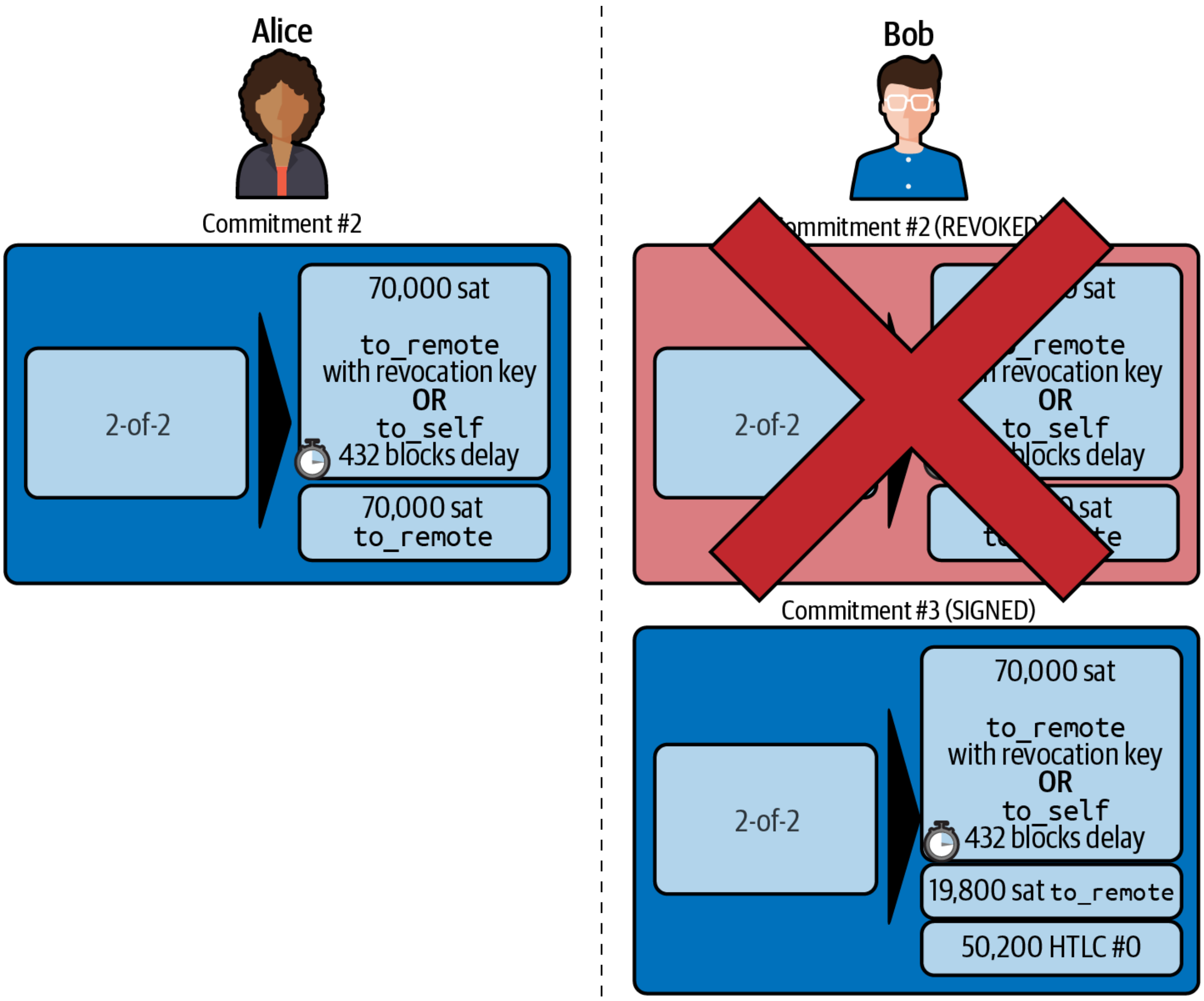


Figure 58. Bob has revoked the old commitment

Despite the fact that Bob has a new (signed) commitment transaction and an HTLC output inside, he cannot consider his HTLC as being set up successfully.

He first needs to have Alice revoke her old commitment, because otherwise, Alice can roll back her balance to 70,000 satoshis. Bob needs to make sure that Alice also has a commitment transaction containing the HTLC and has revoked the old commitment.

That is why, if Bob is not the final recipient of the HTLC funds, he should not forward the HTLC yet by offering an HTLC on the next channel with Chan.

Alice has constructed a mirror-image new commitment transaction containing the new HTLC, but it is yet to be signed by Bob. We can see it in Alice's new commitment with an HTLC output.

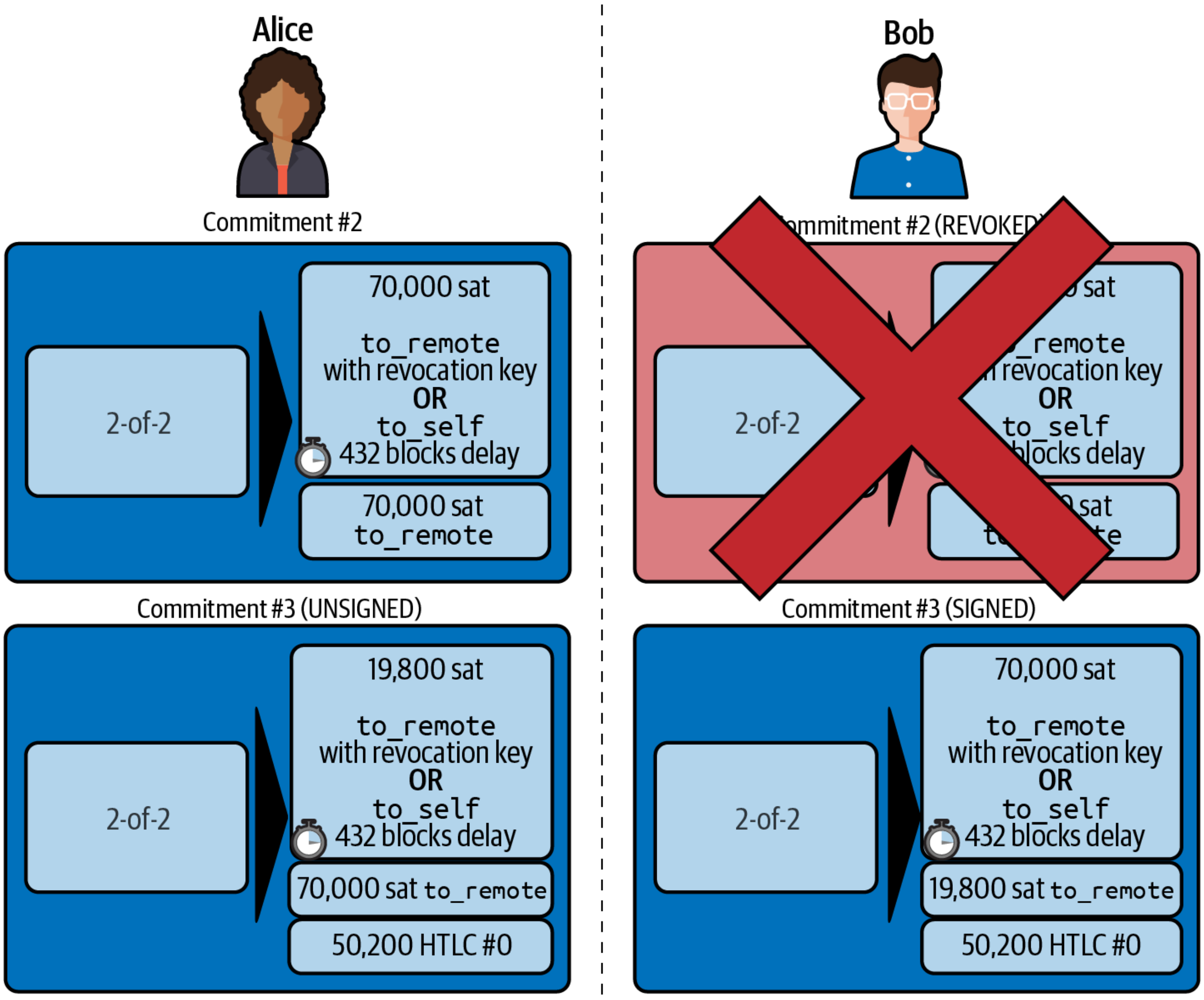


Figure 59. Alice's new commitment with an HTLC output

As we described in Payment Channels, Alice's commitment is the mirror image of Bob's, as it contains the asymmetric, delayed, revocable construct for revocation and penalty enforcement of old commitments. Alice's 19,800 satoshi balance (after deducting the HTLC value), is delayed and revocable. Bob's 70,000 satoshi balance is immediately redeemable.

Next, the message flow for commitment_signed and revoke_and_ack is now repeated, but in the opposite direction. Bob sends commitment_signed to sign Alice's new commitment, and Alice responds by revoking her old commitment.

For completeness sake, let's quickly review the commitment transactions as this round of commitment/revocation happens.

Bob Commits

Bob now sends a commitment_signed back to Alice, with his signatures for Alice's new commitment transaction, including the HTLC output she has added.

Now Alice has the signature for the new commitment transaction. The state of the channel is shown in Alice has a new **signed** commitment.

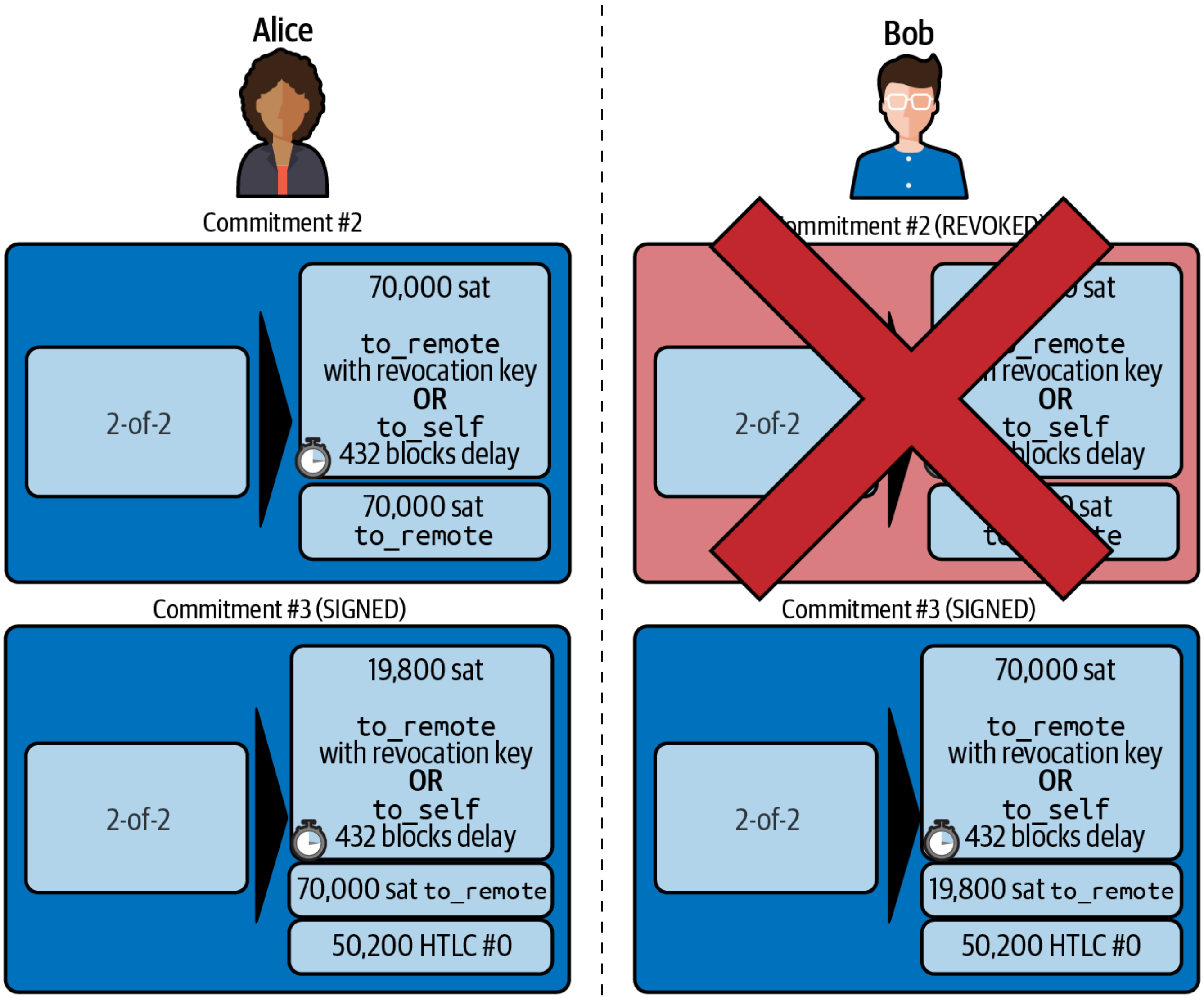


Figure 60. Alice has a new **signed** *commitment*

Alice can now acknowledge the new commitment by revoking the old one. Alice sends the revoke_and_ack message containing the necessary per_commitment_point that will allow Bob to construct a revocation key and penalty transaction. Thus, Alice revokes her old commitment.

The channel state is shown in Alice has revoked the old commitment.

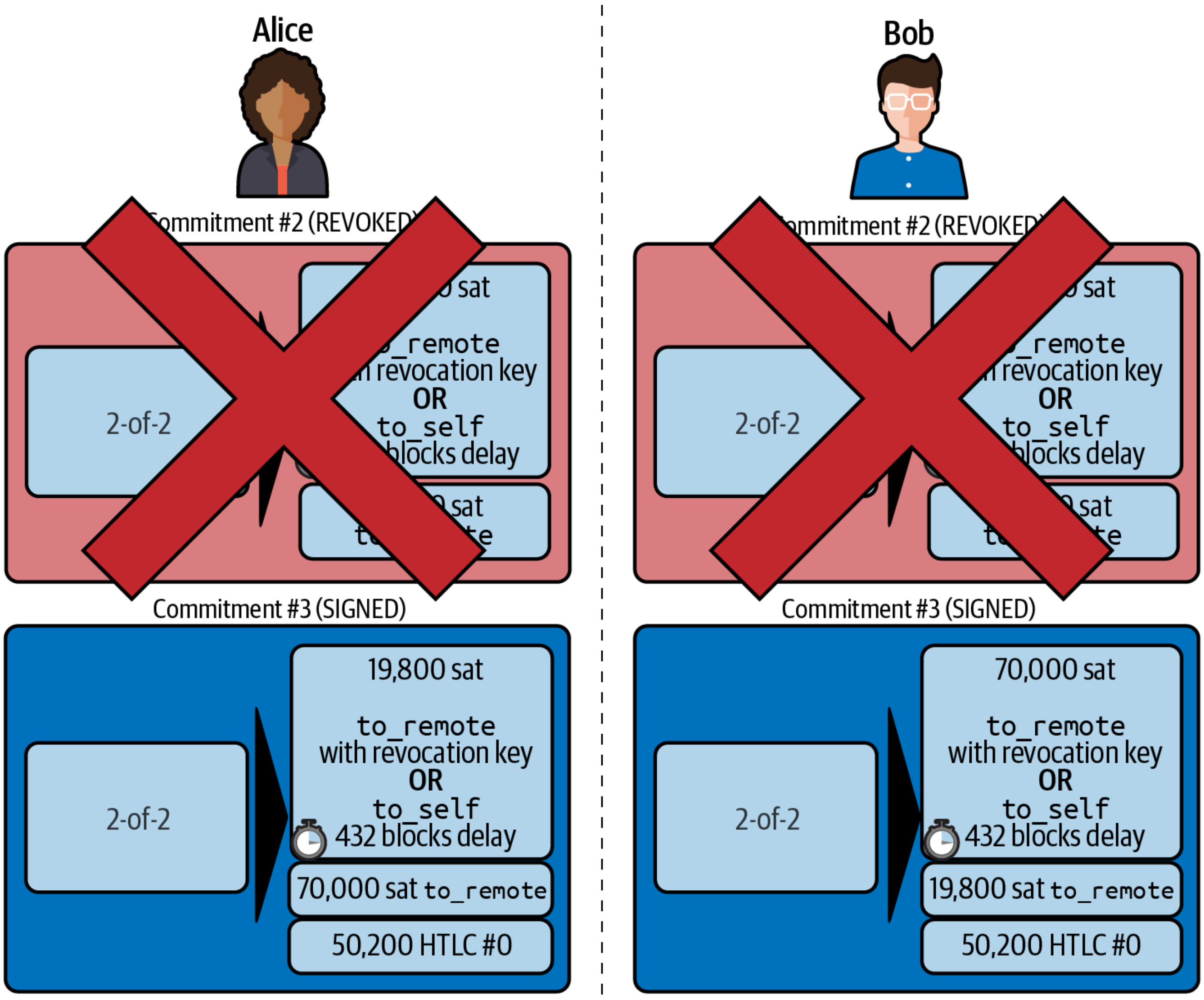


Figure 61. Alice has revoked the old commitment

Multiple HTLCs

At any point in time, Alice and Bob may have dozens or even hundreds of HTLCs across a single channel. Each HTLC is offered and added to the commitment transaction as an additional output. A commitment transaction therefore always has two outputs for the channel partner balances and any number of HTLC outputs, one per HTLC.

As we saw in the commitment_signed message, there is an array for HTLC signatures so that multiple HTLC commitments can be transmitted at the same time.

The current maximum number of HTLCs allowed on a channel is 483 HTLCs to account for the maximum Bitcoin transaction size and ensure that the commitment transactions continue to be valid Bitcoin transactions.

As we will see in the next section, the maximum is only for *pending* HTLCs because, once an HTLC is fulfilled (or fails due to timeout/error), it is removed from the commitment transaction.

HTLC Fulfillment

Now Bob and Alice both have a new commitment transaction with an additional HTLC output, and we have achieved a major step toward updating a payment channel.

The new balance of Alice and Bob does not reflect yet that Alice has successfully sent 50,200 satoshis to Bob.

However, the HTLCs are now set up in a way that secure settlement in exchange for the proof of payment will be possible.

HTLC Propagation

Let's assume that Bob continues the chain and sets up an HTLC with Chan for 50,100 satoshis. The process will be exactly the same as we just saw between Alice and Bob. Bob will send update_add_htlc to Chan, then they will exchange commitment_signed and revoke_and_ack messages in two rounds, progressing their channel to the next state.

Next, Chan will do the same with Dina: offer a 50,000 satoshi HTLC, commit, and revoke, etc. However, Dina is the final recipient of the HTLC. Dina is the only one that knows the payment secret (the preimage of the payment hash). Therefore, Dina can fulfill the HTLC with Chan immediately!

Dina Fulfills the HTLC with Chan

Dina can settle the HTLC by sending an update_fulfill_htlc message to Chan. The update_fulfill_htlc message is defined in BOLT #2: Peer Protocol, `update_fulfill_htlc` and is shown here:

The update_fulfill_htlc message

```
[channel_id:channel_id]
[u64:id]
[32*byte:payment_preimage]
```

It's a really simple message:

channel_id

The channel ID on which the HTLC is committed.

id

The ID of the HTLC (we started with 0 and incremented for each HTLC on the channel).

payment_preimage

The secret that proves payment was made and redeems the HTLC. This is the R value that was hashed by Dina to produce the payment hash in the invoice to Alice.

When Chan receives this message, he will immediately check if the `payment_preimage` (let's call it *R*) produces the payment hash (let's call it *H*) in the HTLC that he offered to Dina. He hashes it like this:

```
<ul class="simplelist">
<li><em>H</em> = RIPEMD160(SHA-256 (<em>R</em>))</li>
</ul>
```

If the result *H* matches the payment hash in the HTLC, Chan can do a little dance of celebration. This long-awaited secret can be used to redeem the HTLC, and will be passed back along the chain of payment channels all the way to Alice, resolving every HTLC that was part of this payment to Dina.

Let's go back to Alice and Bob's channel and watch them unwind the HTLC. To get there, let's assume Dina sent the update_fulfill_htlc to Chan, Chan sent update_fulfill_htlc to Bob, and Bob sent update_fulfill_htlc to Alice. The payment preimage has propagated all the way back to Alice.

Bob Settles the HTLC with Alice

When Bob sends the update_fulfill_htlc to Alice, it will contain the same payment_preimage that Dina selected for her invoice. That payment_preimage has traveled backward along the payment path. At each step, the channel_id will be different and id (HTLC ID) may be different. But the preimage is the same!

Alice will also validate the payment_preimage received from Bob. She will compare its hash to the HTLC payment hash in the HTLC she offered Bob. She will also find this preimage matches the hash in Dina's invoice. This is proof that Dina was paid.

The message flow between Alice and Bob is shown in The HTLC fulfillment message flow.

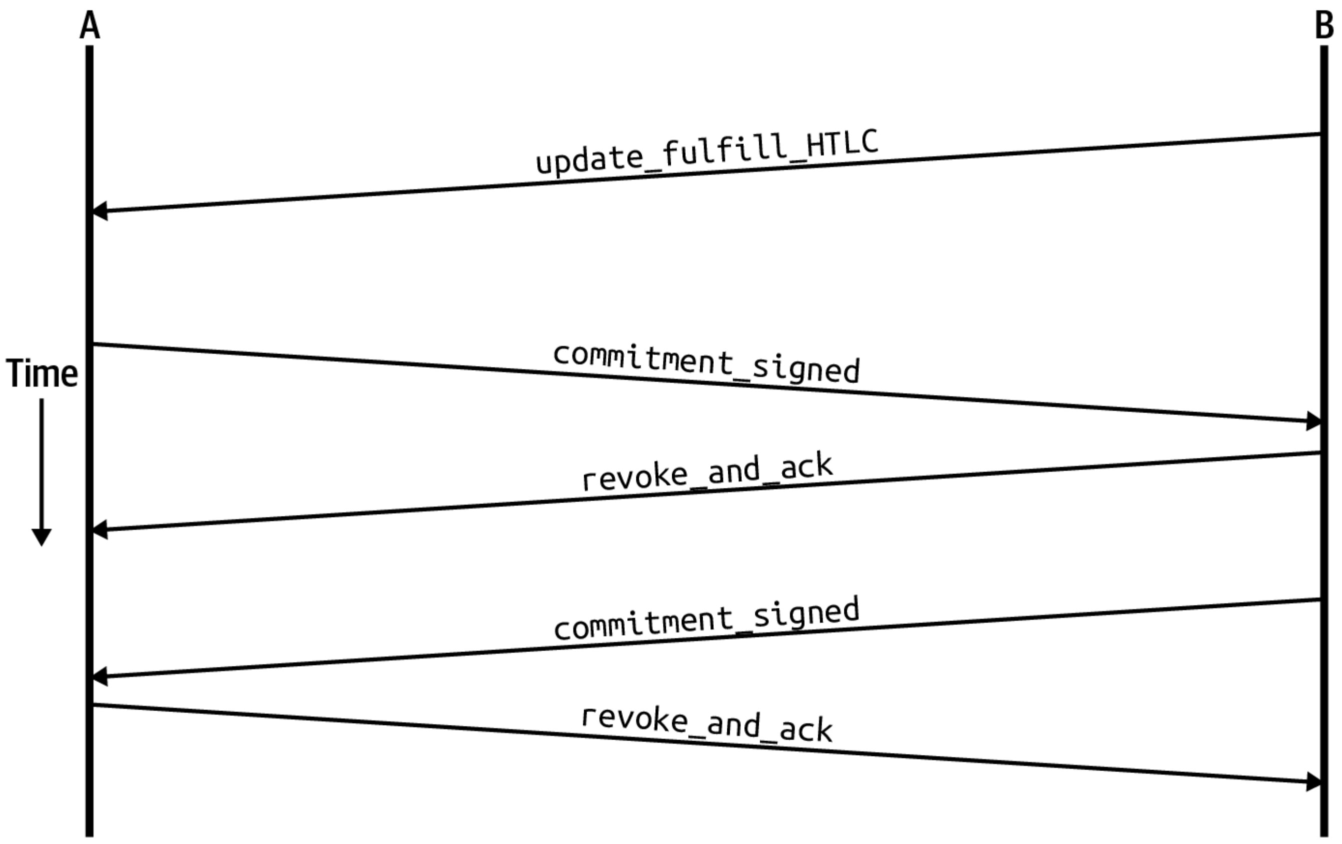


Figure 62. The HTLC fulfillment message flow

Both Alice and Bob can now remove the HTLC from the commitment transactions and update their channel balances.

They create new commitments (Commitment #4), as shown in The HTLC is removed and balances are updated in new commitments.

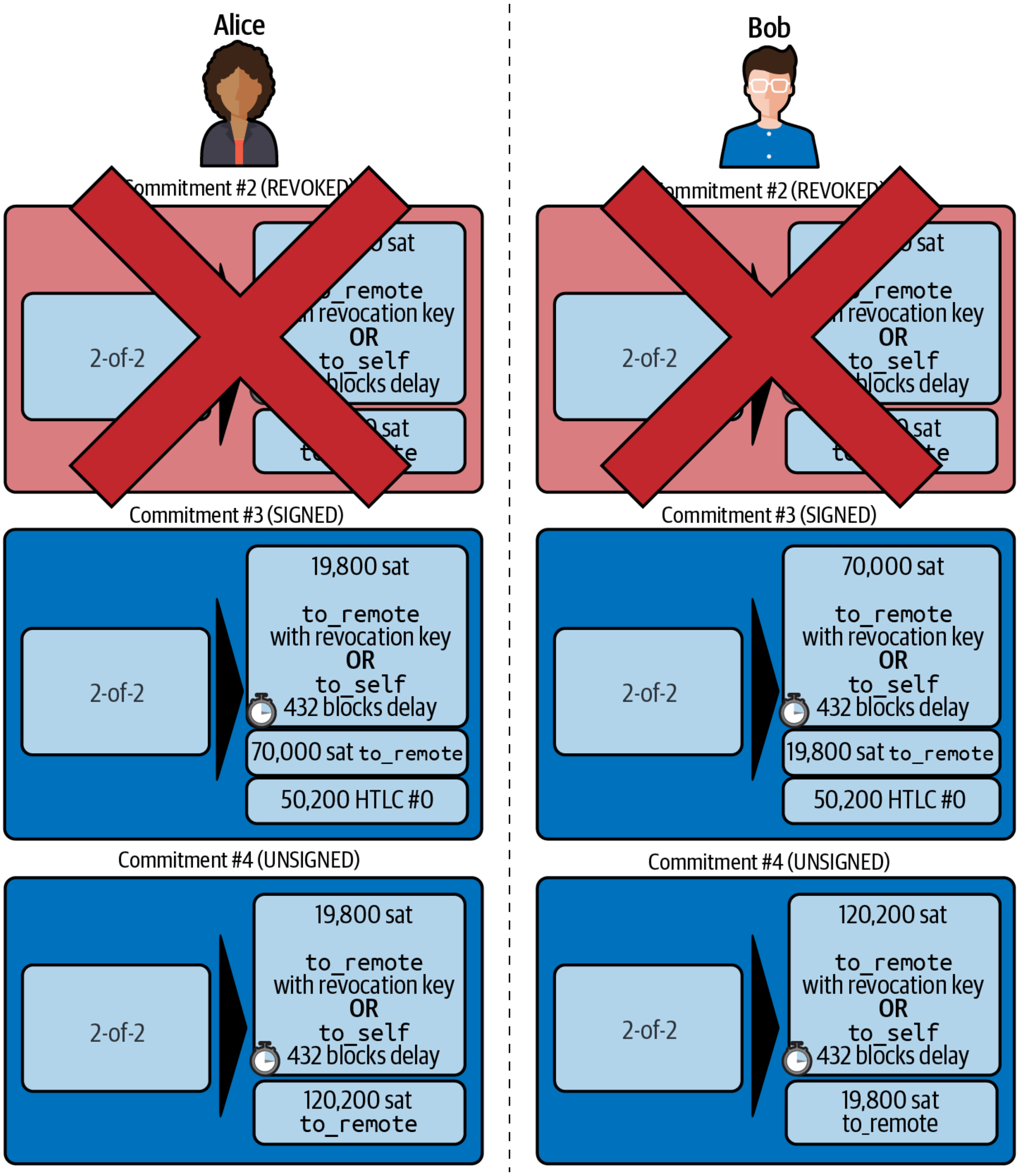


Figure 63. The HTLC is removed and balances are updated in new commitments

Next, they complete two rounds of commitment and revocation. First, Alice sends commitment_signed to sign Bob's new commitment transaction. Bob responds with revoke_and_ack to revoke his old commitment. Once Bob has moved the state of the channel forward, the commitments look like we see in Alice signs Bob's new commitment and Bob revoked the old one.

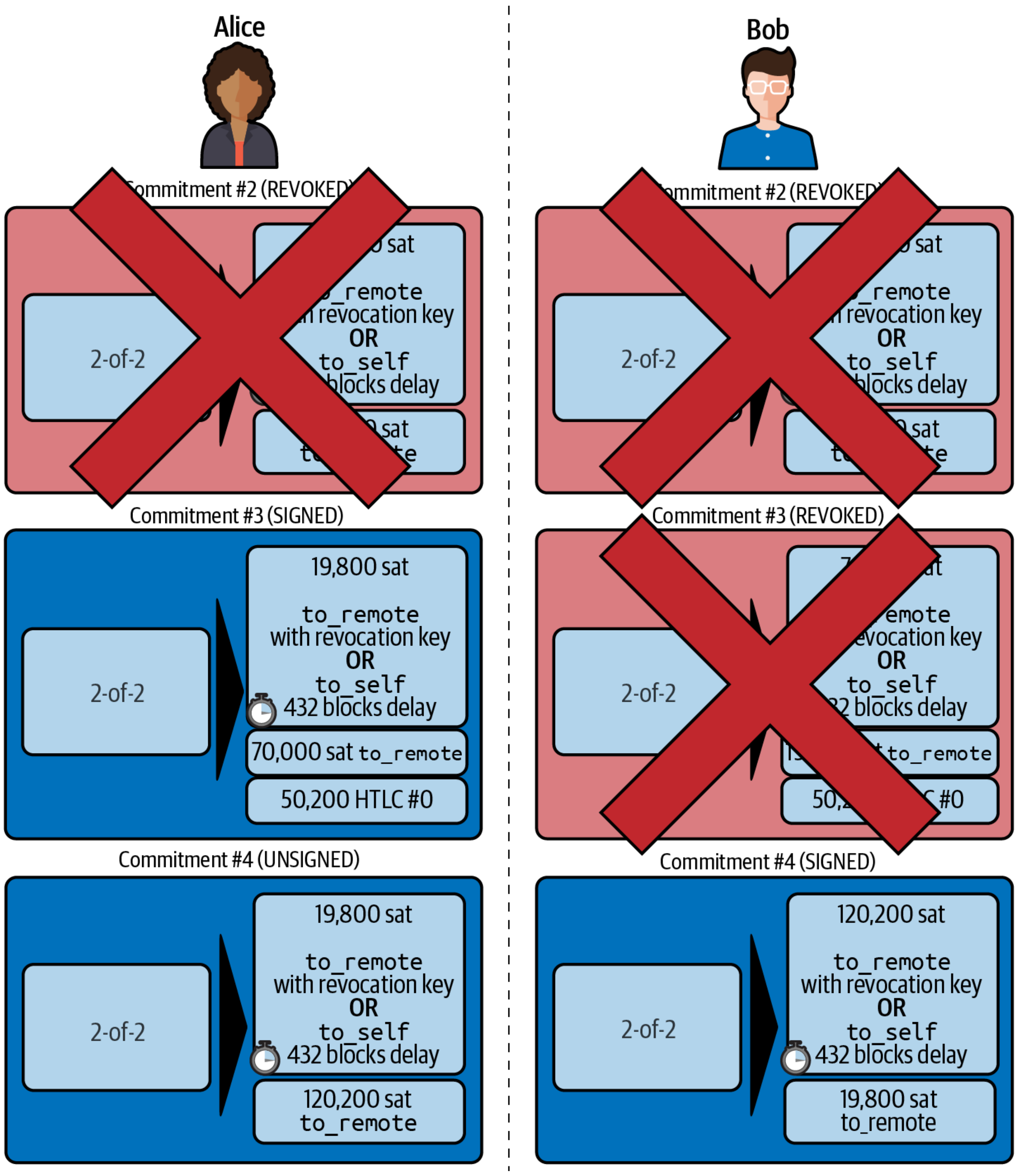


Figure 64. Alice signs Bob's new commitment and Bob revoked the old one

Finally, Bob signs Alice's commitment by sending Alice a commitment_signed message. Then Alice acknowledges and revokes her old commitment by sending revoke_and_ack to Bob. The end result is that both Alice and Bob have moved their channel state to Commitment #4, have removed the HTLC, and have updated their balances. Their current channel state is represented by the commitment transactions that are shown in Alice and Bob settle the HTLC and update balances.

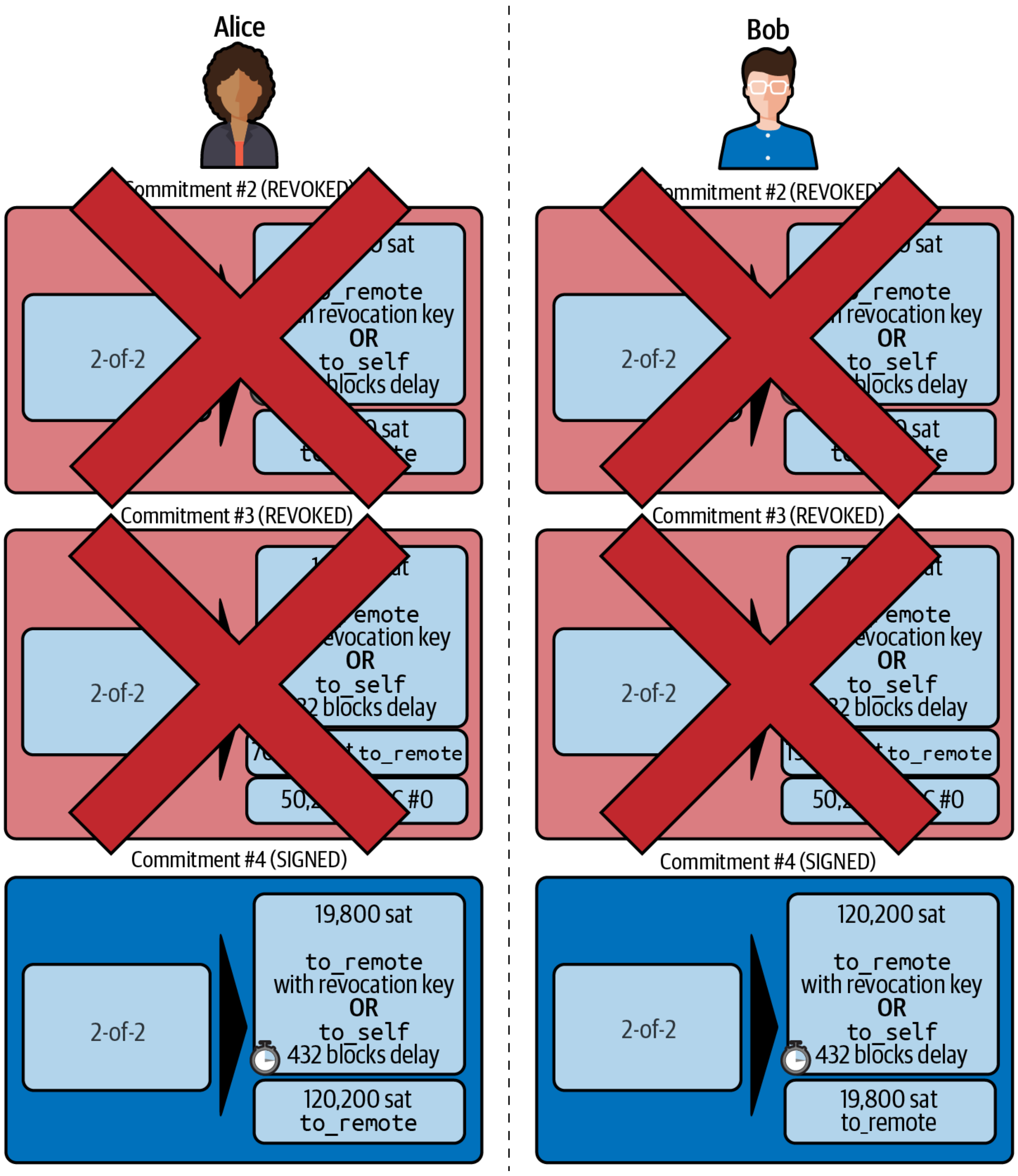


Figure 65. Alice and Bob settle the HTLC and update balances

Removing an HTLC Due to Error or Expiry

If an HTLC cannot be fulfilled, it can be removed from the channel commitment using the same process of commitment and revocation.

Instead of update_fulfill_htlc, Bob would send an update_fail_htlc or update_fail_malformed_htlc. These two messages are defined in BOLT #2: Peer Protocol, Removing an HTLC.

The update_fail_htlc message is shown in the following:

The update_fail_htlc message

```
[channel_id:channel_id]
[u64:id]
[u16:len]
[len*byte:reason]
```

It's pretty self-explanatory. The multibyte reason field is defined in BOLT #4: Onion Routing, which we will describe in more detail in Onion Routing.

If Alice received an update_fail_htlc from Bob, the process would unfold in much the same way: the two channel partners would remove the HTLC, create updated commitment transactions, and go through two rounds of commitment/revocation to move the channel state forward to the next commitment. The only difference: the end balances would revert back to what they were without the HTLC, essentially refunding Alice for the HTLC value.

Making a Local Payment

At this point, you will easily understand why HTLCs are used for both remote and local payments. When Alice pays Bob for a coffee, she doesn't just update the channel balance and commit to a new state. Instead, the payment is made with an HTLC, in the same way Alice paid Dina. The fact that there's only one channel hop makes no difference. It would work like this:

1. Alice orders a coffee from Bob's shop page.
2. Bob's shop sends an invoice with a payment hash.
3. Alice constructs an HTLC from that payment hash.
4. Alice offers the HTLC to Bob with update_add_htlc.
5. Alice and Bob exchange commitments and revocations adding the HTLC to their commitment transactions.
6. Bob sends update_fulfill_htlc to Alice with the payment preimage.
7. Alice and Bob exchange commitments and revocations removing the HTLC and updating the channel balances.

Whether an HTLC is forwarded across many channels or just fulfilled in a single channel "hop," the process is exactly the same

Conclusion

In this chapter we saw how commitment transactions (from Payment Channels) and HTLCs (from Routing on a Network of Payment Channels) work together. We saw how an HTLC is added to a commitment transaction, and how it is fulfilled. We saw how the asymmetric, delayed, revocable system for enforcing channel state is extended to HTLCs.

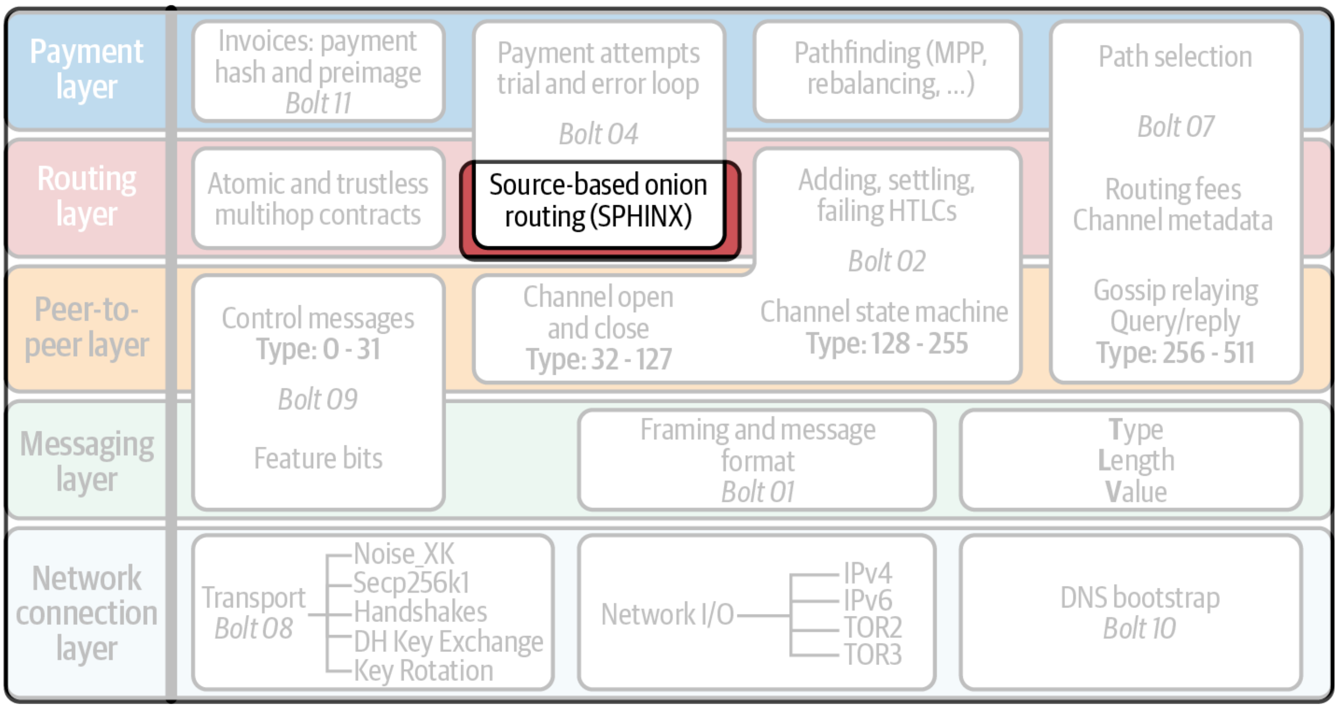
We also saw how a local payment and a multihop routed payment are handled identically: using HTLCs.

In the next chapter we will look at the encrypted message routing system called *onion routing*.

Onion Routing

In this chapter we will describe the Lightning Network's onion routing mechanism. The invention of *onion routing* precedes the Lightning Network by 25 years! Onion routing was invented by U.S. Navy researchers as a communications security protocol. Onion routing is most famously used by Tor, the onion-routed internet overlay that allows researchers, activists, intelligence agents, and everyone else to use the internet privately and anonymously.

In this chapter we are focusing on the "Source-based onion routing (SPHINX)" part of the Lightning protocol architecture, highlighted by an outline in the center (routing layer) of Onion routing in the Lightning protocol suite.



Payment layer
Routing layer
Peer-to-peer layer
Messaging layer
Network connection layer
Invoices: payment hash and preimage Bolt 11
Payment attempts trial and error loop Bolt 04
Pathfinding (MPP, rebalancing, ...)
Path selection Bolt 07
Atomic and trustless multihop contracts
Source-based onion routing (SPHINX)
Adding, settling, failing HTLCs Bolt 02
Routing fees Channel metadata
Control messages Type: 0 - 31 Bolt 09 Feature bits
Channel open and close Type: 32 - 127
Channel state machine Type: 128 - 255
Gossip relaying Query/reply Type: 256 - 511
Framing and message format Bolt 01
Type Length Value
Transport Bolt 08
Noise_XK
Secp256k1
Handshakes
DH Key Exchange
Key Rotation
Network I/O
IPv4
IPv6
TOR2
TOR3
DNS bootstrap Bolt 10

Figure 66. Onion routing in the Lightning protocol suite

Onion routing describes a method of encrypted communication where a message sender builds successive *nested layers of encryption* that are "peeled" off by each intermediary node, until the innermost layer is delivered to the intended recipient. The name "onion routing" describes this use of layered encryption that is peeled off one layer at a time, like the skin of an onion.

Each of the intermediary nodes can only "peel" one layer and see who is next in the communication path. Onion routing ensures that no one except the sender knows the destination or length of the communication path. Each intermediary only knows the previous and next hop.

The Lightning Network uses an implementation of onion routing protocol based on *Sphinx*,[10] developed in 2009 by George Danezis and Ian Goldberg.

The implementation of onion routing in the Lightning Network is defined in BOLT #4: Onion Routing Protocol.

A Physical Example Illustrating Onion Routing

There are many ways to describe onion routing, but one of the easiest is to use the physical equivalent of sealed envelopes. An envelope represents a layer of encryption, allowing only the named recipient to open it and read the contents.

Let's say Alice wants to send a secret letter to Dina, indirectly via some intermediaries.

Selecting a Path

The Lightning Network uses *source routing*, which means that the payment path is selected and specified by the sender, and only the sender. In this example, Alice's secret letter to Dina will be the equivalent of a payment. To make sure the letter reaches Dina, Alice will create a path from her to Dina, using Bob and Chan as intermediaries.

TIP There may be many paths that make it possible for Alice to reach Dina. We will explain the process of selecting the *optimum* path in Pathfinding and Payment Delivery. For now, we'll assume that the path selected by Alice uses Bob and Chan as intermediaries to get to Dina.

As a reminder, the path selected by Alice is shown in Path: Alice to Bob to Chan to Dina.

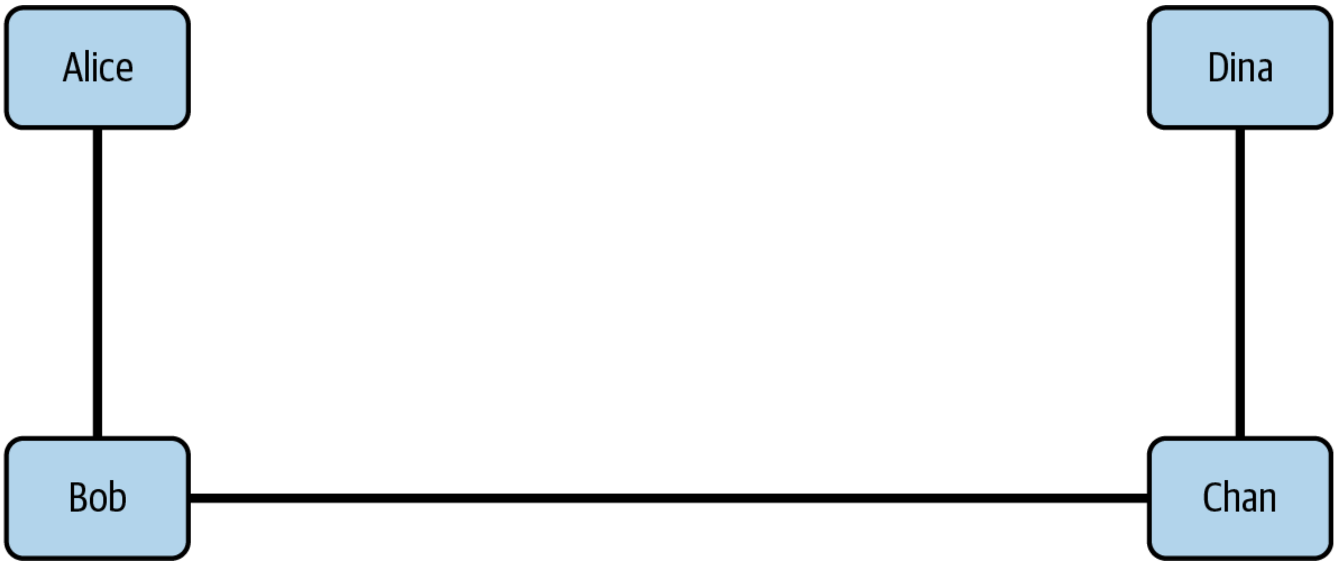


Figure 67. Path: Alice to Bob to Chan to Dina

Let's see how Alice can use this path without revealing information to intermediaries Bob and Chan.

Source-Based Routing

Source-based routing is not how packets are typically routed on the internet today, though source routing was possible in the early days. Internet routing is based on *packet switching* at each intermediary routing node. An IPv4 packet, for example, includes the sender and recipient's IP addresses, and every other IP routing node decides how to forward each packet toward the destination. However, the lack of privacy in such a routing mechanism, where every intermediary node sees the sender and recipient, makes this a poor choice for use in a payment network.

Building the Layers

Alice starts by writing a secret letter to Dina. She then seals the letter inside an envelope and writes "To Dina" on the outside (see Dina's secret letter, sealed in an envelope). The envelope represents encryption with Dina's public key so that only Dina can open the envelope and read the letter.

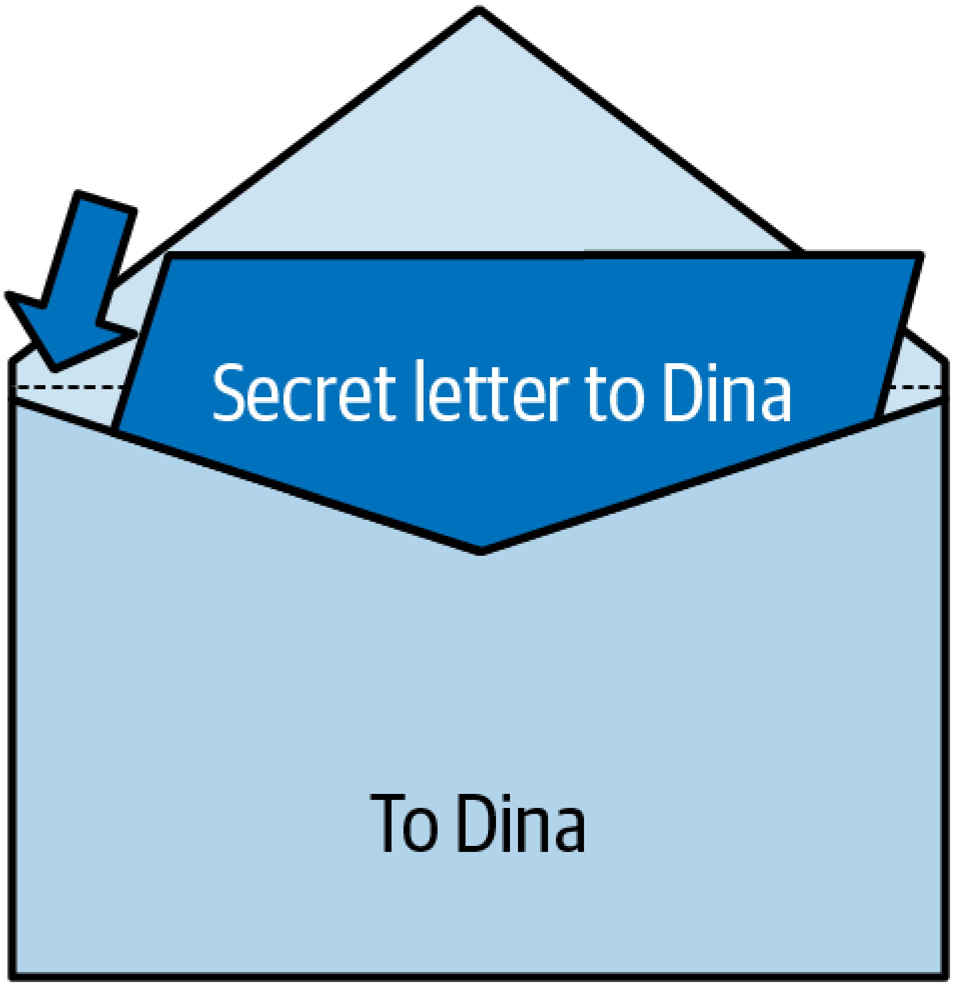


Figure 68. Dina's secret letter, sealed in an envelope

Dina's letter will be delivered to Dina by Chan, who is immediately before Dina in the "path." So, Alice puts Dina's envelope inside an envelope addressed to Chan (see Chan's envelope, containing Dina's sealed envelope). The only part that Chan can read is the destination (routing instructions): "To Dina." Sealing this inside an envelope addressed to Chan represents encrypting it with Chan's public key so that only Chan can read the envelope address. Chan still can't open Dina's envelope.

All he sees is the instructions on the outside (the address).

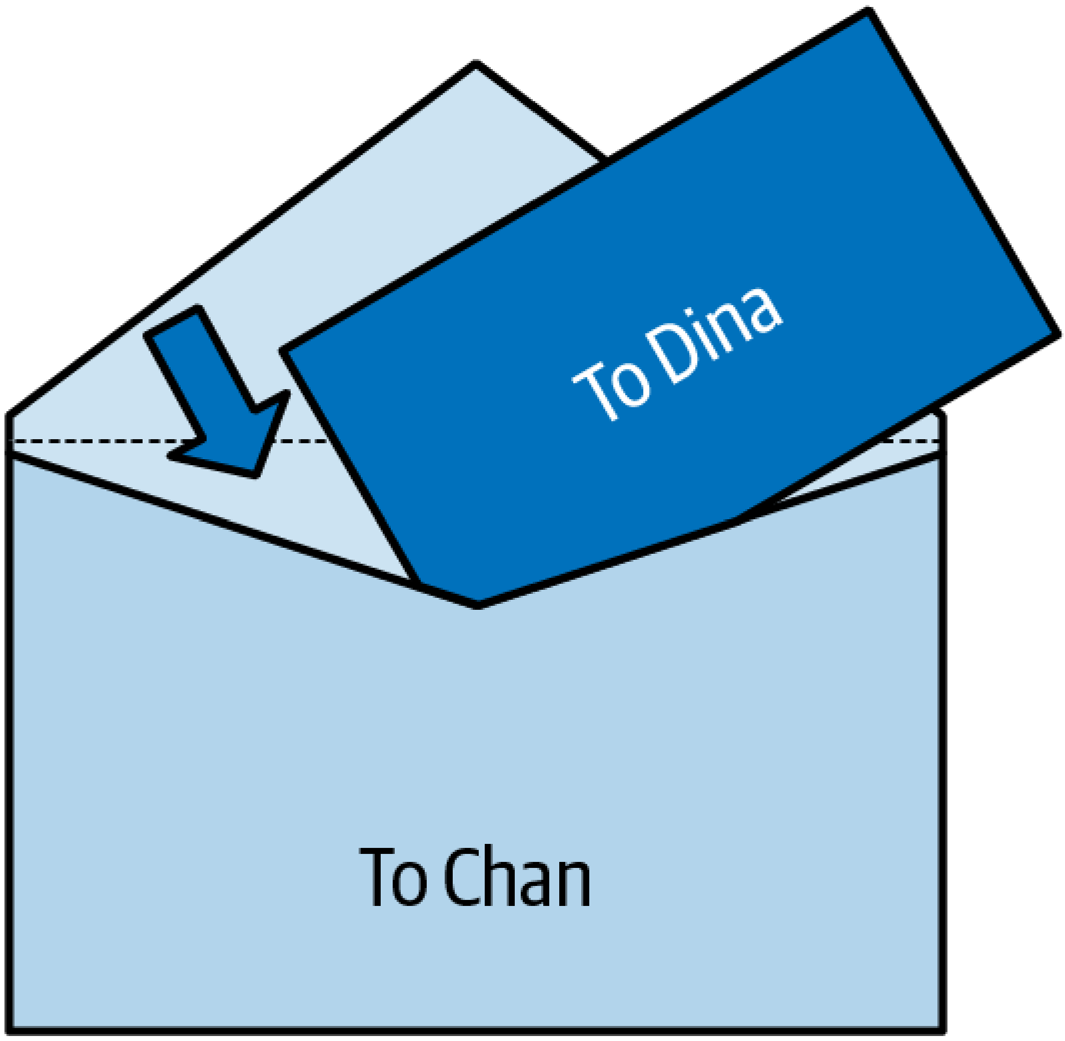


Figure 69. Chan's envelope, containing Dina's sealed envelope

Now, this letter will be delivered to Chan by Bob. So Alice puts it inside an envelope addressed to Bob (see Bob's envelope, containing Chan's sealed envelope). As before, the envelope represents a message encrypted to Bob that only Bob can read. Bob can only read the outside of Chan's envelope (the address), so he knows to send it to Chan.

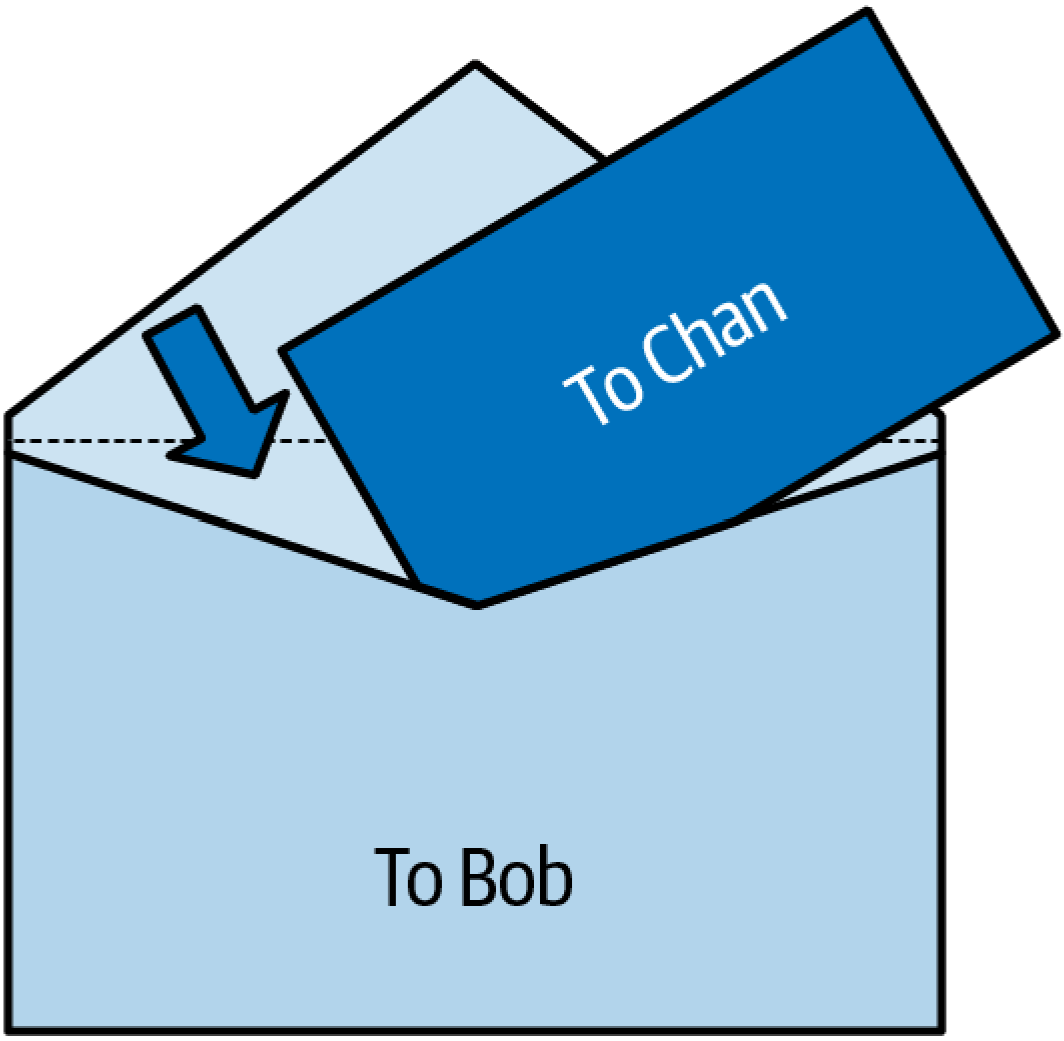


Figure 70. Bob’s envelope, containing Chan’s sealed envelope

Now, if we could look through the envelopes (with X-rays!) we would see the envelopes nested one inside the other, as shown in Nested envelopes.

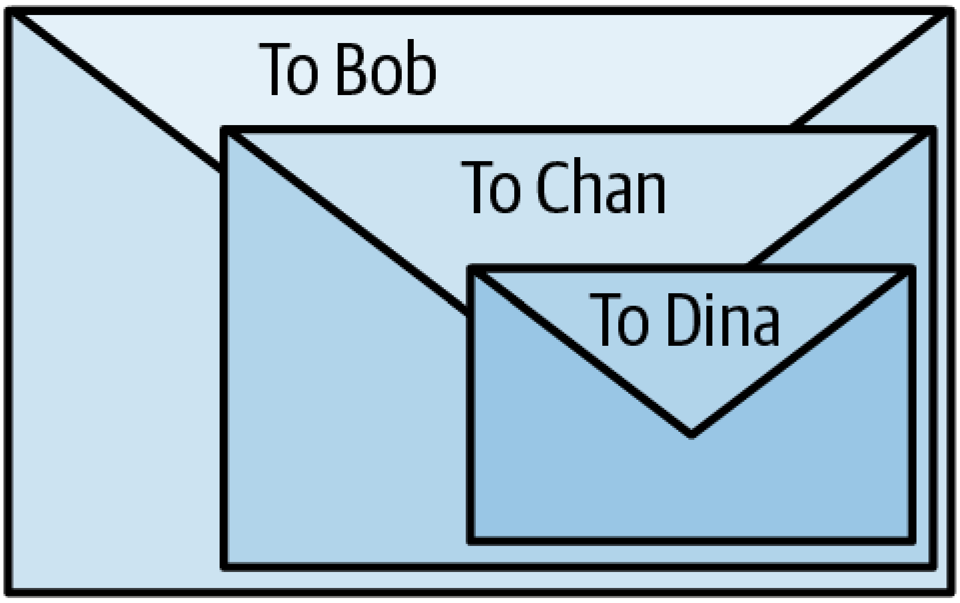


Figure 71. Nested envelopes

Peeling the Layers

Alice now has an envelope that says "To Bob" on the outside. It represents an encrypted message that only Bob can open (decrypt). Alice will now begin the process by sending this to Bob. The entire process is shown in Sending the envelopes.

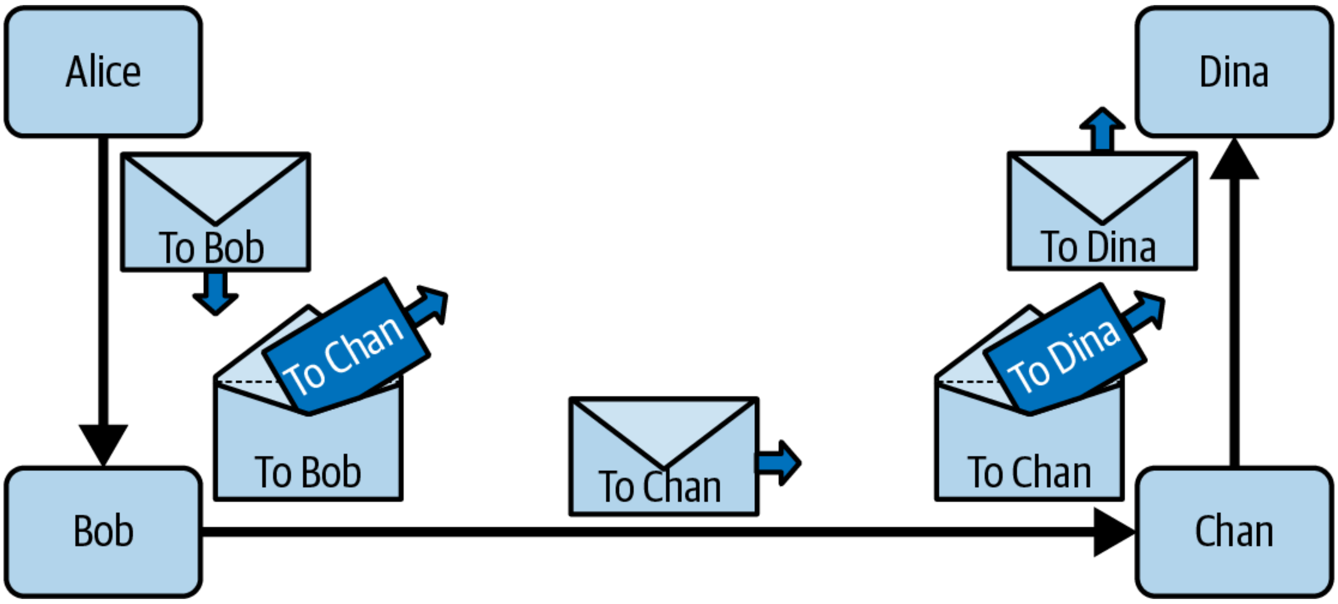


Figure 72. Sending the envelopes

As you can see, Bob receives the envelope from Alice. He knows it came from Alice, but doesn’t know if Alice is the original sender or just someone forwarding envelopes. He opens it to find an envelope inside that says "To Chan." Since this is addressed to Chan, Bob can’t open it. He doesn’t know what’s inside it and doesn’t know if Chan is getting a letter or another envelope to forward. Bob doesn’t know if Chan is the ultimate recipient or not. Bob forwards the envelope to Chan.

Chan receives the envelope from Bob. He doesn’t know that it came from Alice. He doesn’t know if Bob is an intermediary or the sender of a letter. Chan opens the envelope and finds another envelope inside addressed "To Dina," which he can’t open. Chan forwards it to Dina, not knowing if Dina is the final recipient.

Dina receives an envelope from Chan. Opening it she finds a letter inside, so now she knows she’s the intended recipient of this message. She reads the letter, knowing that none of the intermediaries know where it came from and no one else has read her secret letter!

This is the essence of onion routing. The sender wraps a message in layers, specifying exactly how it will be routed and preventing any of the intermediaries from gaining any information about the path or payload. Each intermediary peels one layer, sees only a forwarding address, and doesn’t know anything other than the previous and next hop in the path.

Now, let’s look at the details of the onion routing implementation in the Lightning Network.

Introduction to Onion Routing of HTLCs

Onion routing in the Lightning Network appears complex at first glance, but once you understand the basic concept, it is really quite simple.

From a practical perspective, Alice is telling every intermediary node what HTLC to set up with the next node in the path.

The first node, which is the payment sender or Alice in our example, is called the *origin node*. The last node, which is the payment recipient or Dina in our example, is called the *final node*.

Each intermediary node, or Bob and Chan in our example, is called a *hop*. Every hop must set up an *outgoing HTLC* to the next hop. The information communicated to each hop by Alice is called the *hop payload* or *hop data*. The message that is routed from Alice to Dina is called an *onion* and consists of encrypted *hop payload* or *hop data* messages encrypted to each hop.

Now that we know the terminology used in Lightning onion routing, let's restate Alice's task: Alice must construct an onion with hop data, telling each hop how to construct an outgoing HTLC to send a payment to the final node (Dina).

Alice Selects the Path

From Routing on a Network of Payment Channels we know that Alice will send a 50,000 satoshi payment to Dina via Bob and Chan. This payment is transmitted via a series of HTLCs, as shown in Payment path with HTLCs from Alice to Dina.

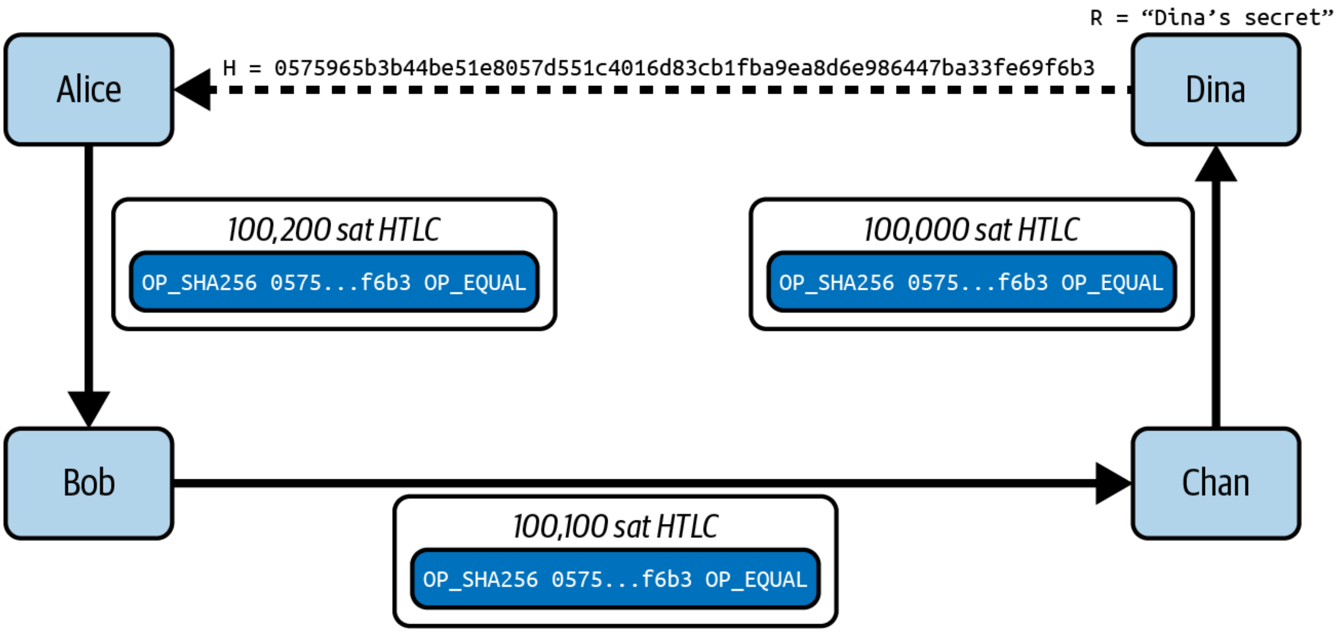

R = "Dina's secret"
H = 0575965b3b44be51e8057d551c4016d83cb1fba9ea8d6e986447ba33fe69f6b3
Alice
Dina
100,200 sat HTLC
OP_SHA256 0575...f6b3 OP_EQUAL
100,000 sat HTLC
OP_SHA256 0575...f6b3 OP_EQUAL
Bob
Chan
100,100 sat HTLC
OP_SHA256 0575...f6b3 OP_EQUAL

Figure 73. Payment path with HTLCs from Alice to Dina

As we will see in Gossip and the Channel Graph, Alice is able to construct this path to Dina because Lightning nodes announce their channels to the entire Lightning Network using the Lightning Gossip Protocol. After the initial channel announcement, Bob and Chan each sent out an additional `channel_update` message with their routing fee and timelock expectations for payment routing.

From the announcements and updates, Alice knows the following information about the channels between Bob, Chan, and Dina:

- A short_channel_id (short channel ID) for each channel, which Alice can use to reference the channel when constructing the path
- A cltv_expiry_delta (timelock delta), which Alice can add to the expiry time for each HTLC

- A fee_base_msat and fee_proportional_millionths, which Alice can use to calculate the total routing fee expected by that node for relay on that channel.

In practice, other information is also exchanged, such as the largest (`htlc_maximum_msat`) and smallest (`htlc_minimum_msat`) HTLCs a channel will carry, but these aren't used as directly during onion route construction as the preceding fields are.

This information is used by Alice to identify the nodes, channels, fees, and timelocks for the following detailed path, shown in A detailed path constructed from gossiped channel and node information.

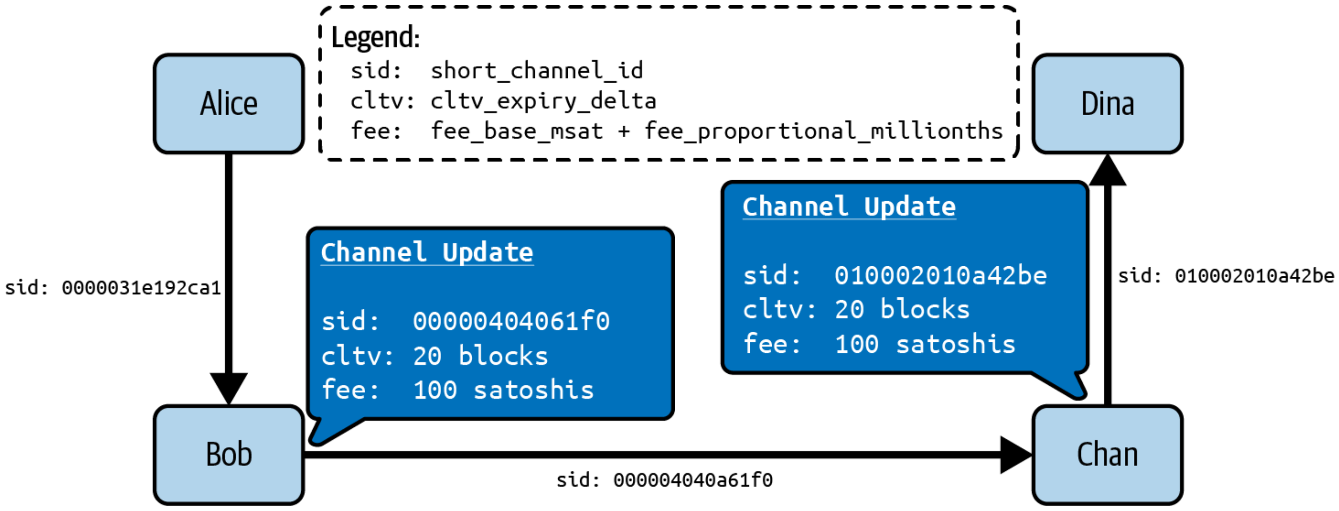


Figure 74. A detailed path constructed from gossiped channel and node information

Alice already knows her own channel to Bob and therefore doesn't need this info to construct the path. Note also that Alice didn't need a channel update from Dina because she has the update from Chan for that last channel in the path.

Alice Constructs the Payloads

There are two possible formats that Alice can use for the information communicated to each hop: a legacy fixed-length format called the *hop data* and a more flexible Type-Length-Value (TLV) based format called the *hop payload*. The TLV message format is explained in more detail in Type-Length-Value (TLV) Format. It offers flexibility by allowing fields to be added to the protocol at will.

NOTE Both formats are specified in BOLT #4: Onion Routing Protocol, Packet Structure.

Alice will start building the hop data from the end of the path backwards: Dina, Chan, then Bob.

Final node payload for Dina

Alice first builds the payload that will be delivered to Dina. Dina will not be constructing an "outgoing HTLC," because Dina is the final node and payment recipient. For this reason, the payload for Dina is different than all the others (uses all zeros for the `short_channel_id`), but only Dina will know this because it will be encrypted in the innermost layer of the onion. Essentially, this is the "secret letter to Dina" we saw in our physical envelope example.

The hop payload for Dina must match the information in the invoice generated by Dina for Alice and will contain (at least) the following fields in TLV format:

amt_to_forward

The amount of this payment in millisatoshis. If this is only one part of a multipart payment, the amount is less than the total. Otherwise, this is a single, full payment and it is equal to the invoice amount and total_msat value.

outgoing_cltv_value

The payment expiry timelock set to the value min_final_cltv_expiry in the invoice.

payment_secret

A special 256-bit secret value from the invoice, allowing Dina to recognize this incoming payment. This also prevents a class of probing that previously made zero-value invoices insecure. Probing by intermediate nodes is mitigated as this value is encrypted to *only* the recipient, meaning they can't reconstruct a final packet that "looks" legitimate.

total_msat

The total amount matching the invoice. This may be omitted if there is only one part, in which case it is assumed to match amt_to_forward and must equal the invoice amount.

The invoice Alice received from Dina specified the amount as 50,000 satoshis, which is 50,000,000 millisatoshis. Dina specified the minimum expiry for the payment min_final_cltv_expiry as 18 blocks (3 hours, given 10-minute on average Bitcoin blocks). At the time Alice is attempting to make the payment, let's say the Bitcoin blockchain has recorded 700,000 blocks. So Alice must set the outgoing_cltv_value to a *minimum* block height of 700,018.

Alice constructs the hop payload for Dina as follows:

```
amt_to_forward : 50,000,000
outgoing_cltv_value: 700,018
payment_secret: fb53d94b7b65580f75b98f10...03521bdab6d519143cd521d1b3826
total_msat: 50,000,000
```

Alice serializes it in TLV format, as shown (simplified) in Dina's payload is constructed by Alice.

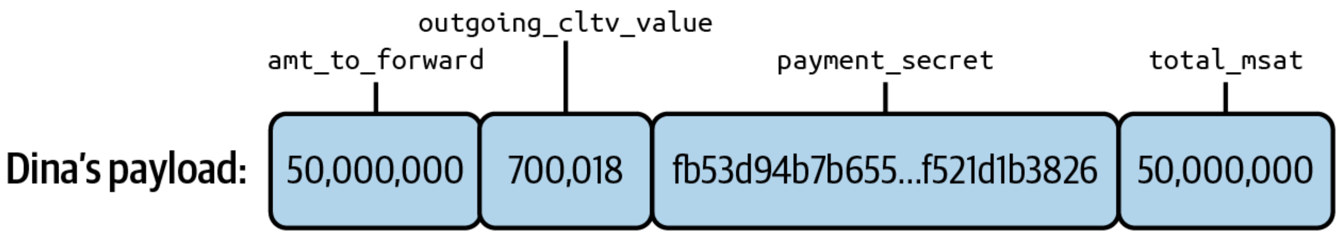


Figure 75. Dina's payload is constructed by Alice

Hop payload for Chan

Next, Alice constructs the hop payload for Chan. This will tell Chan how to set up an outgoing HTLC to Dina.

The hop payload for Chan includes three fields: short_channel_id, amt_to_forward, and outgoing_cltv_value:

```
short_channel_id: 010002010a42be
amt_to_forward: 50,000,000
outgoing_cltv_value: 700,018
```

Alice serializes this payload in TLV format, as shown (simplified) in Chan's payload is constructed by Alice.

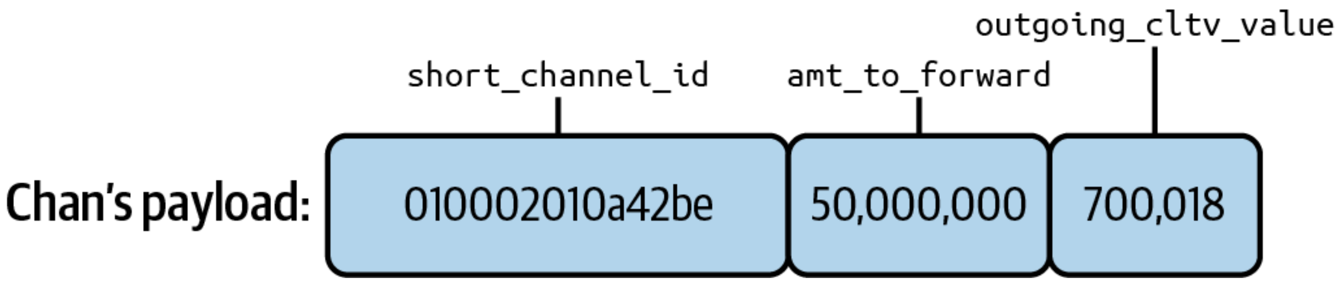


Figure 76. Chan's payload is constructed by Alice

Hop payload for Bob

Finally, Alice constructs the hop payload for Bob, which also contains the same three fields as the hop payload for Chan, but with different values:

```
short_channel_id: 000004040a61f0
amt_to_forward: 50,100,000
outgoing_cltv_value: 700,038
```

As you can see, the amt_to_forward field is 50,100,000 millisatoshis, or 50,100 satoshis. That's because Chan expects a fee of 100 satoshis to route a payment to Dina. In order for Chan to "earn" that routing fee, Chan's incoming HTLC must be 100 satoshis more than Chan's outgoing HTLC. Since Chan's incoming HTLC is Bob's outgoing HTLC, the instructions to Bob reflect the fee Chan earns. In simple terms, Bob needs to be told to send 50,100 satoshi to Chan, so that Chan can send 50,000 satoshi and keep 100 satoshi.

Similarly, Chan expects a timelock delta of 20 blocks. So Chan's incoming HTLC must expire 20 blocks *later* than Chan's outgoing HTLC. To achieve this, Alice tells Bob to make his outgoing HTLC to Chan expire at block height 700,038-20 blocks later than Chan's HTLC to Dina.

TIP Fees and timelock delta expectations for a channel are set by the difference between incoming and outgoing HTLCs. Since the incoming HTLC is created by the *preceding node*, the fee and timelock delta is set in the onion payload to that preceding node. Bob is told how to make an HTLC that meets Chan's fee and timelock expectations.

Alice serializes this payload in TLV format, as shown (simplified) in Bob's payload is constructed by Alice.

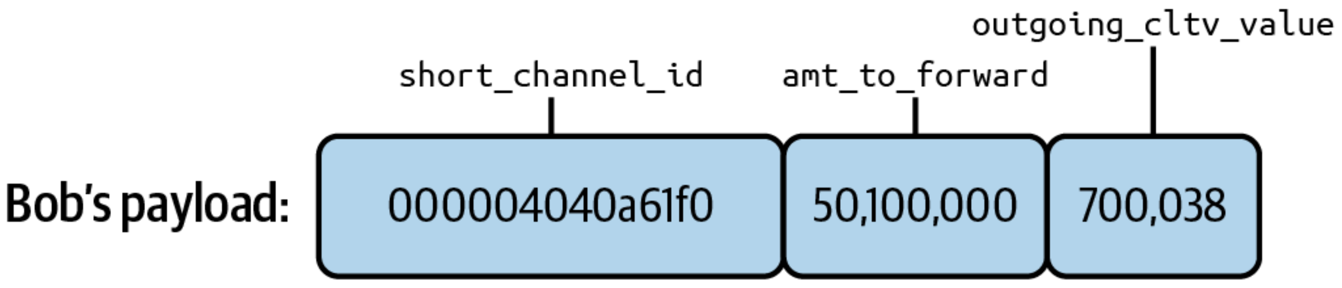


Figure 77. Bob's payload is constructed by Alice

Finished hop payloads

Alice has now built the three hop payloads that will be wrapped in an onion. A simplified view of the payloads is shown in Hop payloads for all the hops.

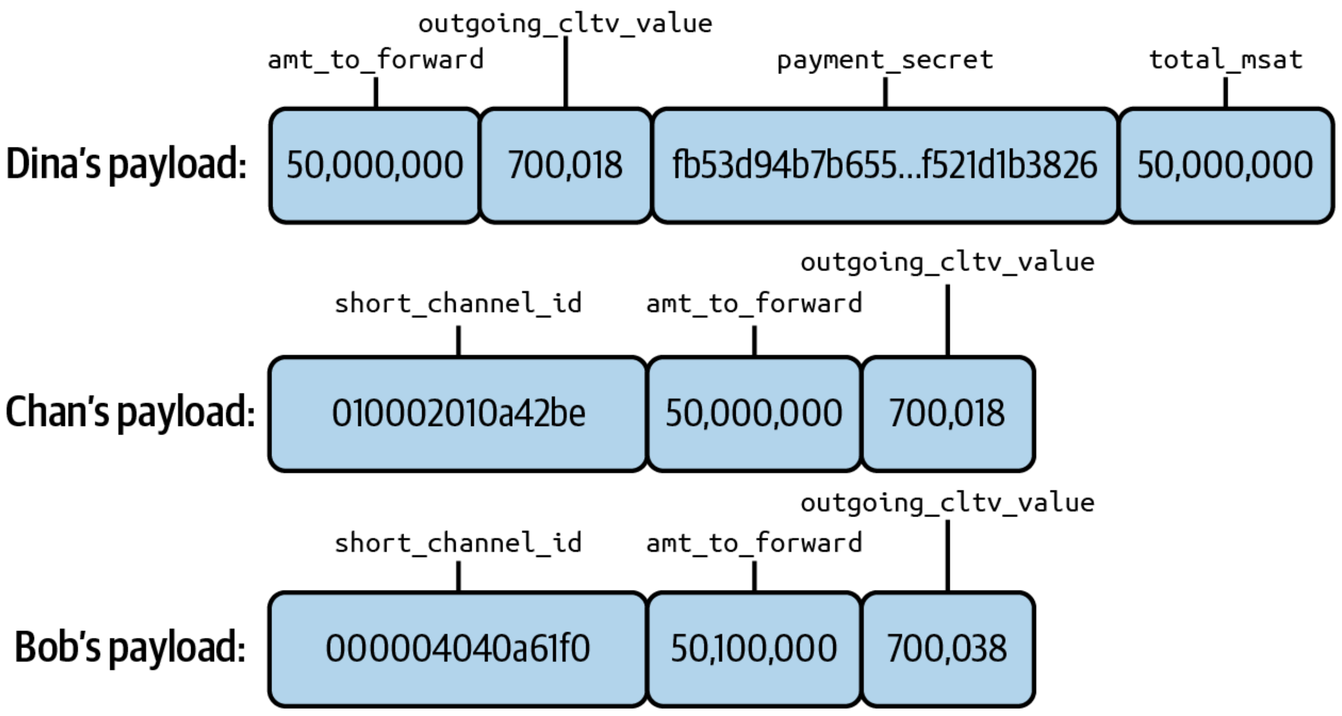


Figure 78. Hop payloads for all the hops

Key Generation

Alice must now generate several keys that will be used to encrypt the various layers in the onion.

With these keys, Alice can achieve a high degree of privacy and integrity:

- Alice can encrypt each layer of the onion so that only the intended recipient can read it.
- Every intermediary can check that the message is not modified.
- No one in the path will know who sent this onion or where it is going. Alice doesn't reveal her identity as the sender or Dina's identity as the recipient of the payment.
- Each hop only learns about the previous and next hop.
- No one can know how long the path is, or where in the path they are.

WARNING

Like a chopped onion, the following technical details may bring tears to your eyes. Feel free to skip to the next section if you get confused. Come back to this and read BOLT #4: Onion Routing, Packet Construction, if you want to learn more.

The basis for all the keys used in the onion is a *shared secret* that Alice and Bob can both generate independently using the Elliptic Curve Diffie–Hellman (ECDH) algorithm. From the shared secret (ss), they can independently generate four additional keys named rho, mu, um, and pad:

rho

Used to generate a stream of random bytes from a stream cipher (used as a CSPRNG). These bytes are used to encrypt/decrypt the message body as well as filler zero bytes during Sphinx packet processing.

mu

Used in the hash-based message authentication code (HMAC) for integrity/authenticity verification.

um

Used in error reporting.

pad

Used to generate filler bytes for padding the onion to a fixed length.

The relationship between the various keys and how they are generated is diagrammed in Onion key generation.

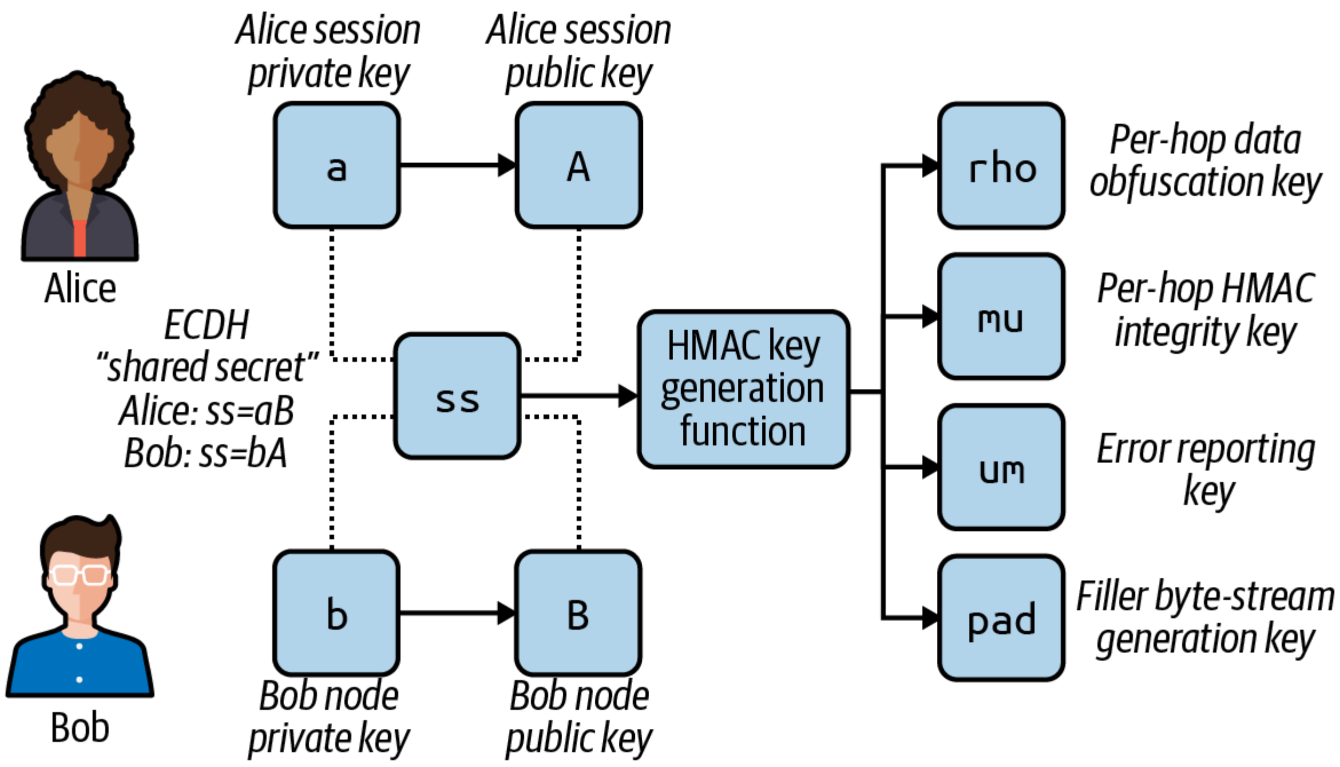

Alice session private key
Alice session public key
a
A
Alice
ECDH "shared secret" Alice: ss=aB Bob: ss=bA
ss
HMAC key generation function
rho
Per-hop data obfuscation key
mu
Per-hop HMAC integrity key
um
Error reporting key
pad
Filler byte-stream generation key
Bob
b
B
Bob node private key
Bob node public key

Figure 79. Onion key generation

Alice’s session key

To avoid revealing her identity, Alice does not use her own node’s public key in building the onion. Instead, Alice creates a temporary 32-byte (256-bit) key called the *session private key* and corresponding *session public key*. This serves as a temporary "identity" and key *for this onion only*. From this session key, Alice will build all the other keys that will be used in this onion.

Key generation details

The key generation, random byte generation, ephemeral keys, and how they are used in packet construction are specified in three sections of BOLT #4:

- Key Generation
- Random Byte Stream
- Packet Construction

For simplicity and to avoid getting too technical, we have not included these details in the book. See the preceding links if you want to see the inner workings.

Shared secret generation

One important detail that seems almost magical is the ability for Alice to create a *shared secret* with another node simply by knowing their public keys. This is based on the invention of Diffie–Hellman key exchange (DH) in the 1970s that revolutionized cryptography. Lightning onion routing uses Elliptic Curve Diffie–Hellman (ECDH) on Bitcoin’s secp256k1 curve. It’s such a cool trick that we try to explain it in simple terms in Elliptic Curve Diffie–Hellman (ECDH) Explained.

Elliptic Curve Diffie–Hellman (ECDH) Explained

Assume Alice's private key is *a* and Bob's private key is *b*. Using the elliptic curve, Alice and Bob each multiply their private key by the generator point *G* to produce their public keys *A* and *B*, respectively:

```
<ul class="simplelist">
<li><em>A</em> = <em>aG</em></li>
<li><em>B</em> = <em>bG</em></li>
</ul>
```

Now Alice and Bob can use *Elliptic Curve Diffie–Hellman Key Exchange* to create a shared secret *ss*, a value that they can both calculate independently without exchanging any information

The shared secret *ss* is calculated by each by multiplying their own private key with the *other's* public key, such that:

```
<ul class="simplelist">
<li><em>ss</em> = <em>aB</em> = <em>bA</em></li>
</ul>
```

But why would these two multiplications result in the same value *ss*? Follow along, as we demonstrate the math that proves this is possible:

```
<ul class="simplelist">
<li><em>ss</em></li>
<li>= <em>aB</em></li>
</ul>
```

calculated by Alice who knows both *a* (her private key) and *B* (Bob's public key)

```
<ul class="simplelist">
<li>= <em>a</em>(<em>bG</em>)</li>
</ul>
```

because we know that *B* = *bG*, we substitute

```
<ul class="simplelist">
<li> = (<em>ab</em>)<em>G</em></li>
</ul>
```

because of associativity, we can move the parentheses

```
<ul class="simplelist">
<li>= (<em>ba</em>)<em>G</em></li>
</ul>
```

because *xy* = *yx* (the curve is an abelian group)

```
<ul class="simplelist">
<li>= <em>b</em>(<em>aG</em>)</li>
</ul>
```

because of associativity, we can move the parentheses

```
<ul class="simplelist">
<li>= <em>bA</em></li>
</ul>
```

and we can substitute *aG* with *A*.

The result *bA* can be calculated independently by Bob who knows *b* (his private key) and *A* (Alice's public key).

We have therefore shown that:

```
<ul class="simplelist">
<li><em>ss</em> = <em>aB</em> (Alice can calculate this)</li>
<li><em>ss</em> = <em>bA</em> (Bob can calculate this)</li>
</ul>
```

Thus, they can each independently calculate *ss* which they can use as a shared key to symmetrically encrypt secrets between the two of them without communicating the shared secret.

A unique trait of Sphinx as a mix-net packet format is that rather than include a distinct session key for each hop in the route, which would increase the size of the mix-net packet dramatically, a clever *blinding* scheme is used to deterministically randomize the session key at each hop.

In practice, this little trick allows us to keep the onion packet as compact as possible while still retaining the desired security properties.

The session key for hop `i` is derived using the node public key, and derived shared secret of hop `i □ 1`:

```
session_key_i = session_key_{i-1} * SHA-256(node_pubkey_{i-1} || shared_secret_{i-1})
```

In other words, we take the session key of the prior hop, and multiply it by a value derived from the public key and the derived shared secret for that hop.

As elliptic curve multiplication can be performed on a public key without knowledge of the private key, each hop is able to re-randomize the session key for the next hop in a deterministic fashion.

The creator of the onion packet knows all the shared secrets (as they've encrypted the packet uniquely for each hop), and thus are able to derive all the blinding factors.

This knowledge allows them to derive all the session keys used up front during packet generation.

Note that the very first hop uses the original session key generated because this key is used to kick off the session key blinding by each subsequent hop.

Wrapping the Onion Layers

The process of wrapping the onion is detailed in BOLT #4: Onion Routing, Packet Construction.

In this section we will describe this process at a high and somewhat simplified level, omitting certain details.

Fixed-Length Onions

We've mentioned the fact that none of the "hop" nodes know how long the path is, or where they are in the path. How is this possible?

If you have a set of directions, even if encrypted, can't you tell how far you are from the beginning or end simply by looking at *where* in the list of directions you are?

The trick used in onion routing is to always make the path (the list of directions) the same length for every node. This is achieved by keeping the onion packet the same length at every step.

At each hop, the hop payload appears at the beginning of the onion payload, followed by *what seem to be* 19 more hop payloads. Every hop sees itself as the first of 20 hops.

TIP The onion payload is 1,300 bytes. Each hop payload is 65 bytes or less (padded to 65 bytes if less). So the total onion payload can fit 20 hop payloads (1300 = 20 x 65). The maximum onion routed path is therefore 20 hops.

As each layer is "peeled off," more filler data (essentially junk) is added at the end of the onion payload so the next hop gets an onion of the same size and is once again the "first hop" in the onion.

The onion size is 1,366 bytes, structured as shown in The onion packet.

1 byte
: A version byte

33 bytes
: A compressed public session key (Alice's session key) from which the per-hop shared secret (Shared secret generation) can be generated without revealing Alice's identity

1,300 bytes

The actual *onion payload* containing the instructions for each hop

32 bytes

An HMAC integrity checksum

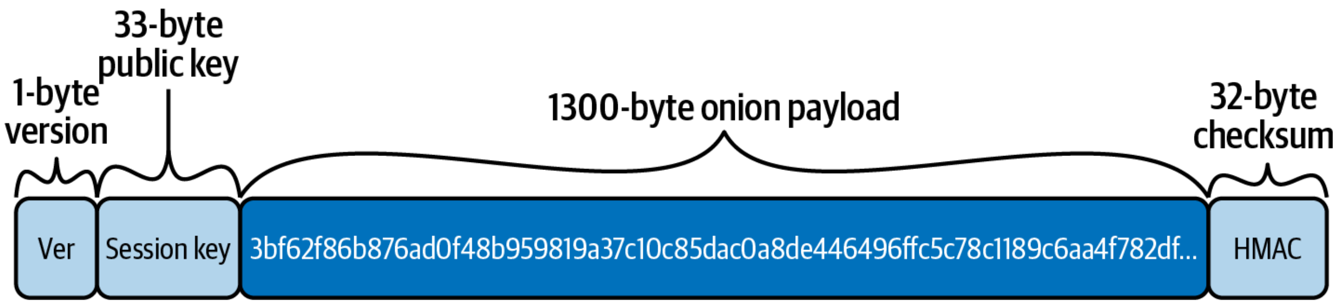


Figure 80. The onion packet

A unique trait of Sphinx as a mix-net packet format is that rather than include a distinct session key for each hop in the route, which would increase the size of the mix-net packet dramatically, instead a clever *blinding* scheme is used to deterministically randomize the session key at each hop.

In practice, this little trick allows us to keep the onion packet as compact as possible while still retaining the desired security properties.

Wrapping the Onion (Outlined)

Here is the process of wrapping the onion, outlined next. Come back to this list as we explore each step with our real-world example.

For each hop, the sender (Alice) repeats the same process:

1. Alice generates the per-hop shared secret and the rho, mu, and pad keys.
2. Alice generates 1,300 bytes of filler and fills the 1,300-byte onion payload field with this filler.
3. Alice calculates the HMAC for the hop payload (zeros for the final hop).
4. Alice calculates the length of the hop payload + HMAC + space to store the length itself.
5. Alice *right-shifts* the onion payload by the calculated space needed to fit the hop payload. The rightmost "filler" data is discarded, making enough space on the left for the payload.
6. Alice inserts the length + hop payload + HMAC at the front of the payload field in the space made from shifting the filler.
7. Alice uses the rho key to generate a 1,300-byte one-time pad.
8. Alice obfuscates the entire onion payload by XORing with the bytes generated from rho.
9. Alice calculates the HMAC of the onion payload, using the mu key.
10. Alice adds the session public key (so that the hop can calculate the shared secret).
11. Alice adds the version number.
12. Alice deterministically re-blinds the session key using a value derived by hashing the shared secret and prior hop's public key.

Next, Alice repeats the process. The new keys are calculated, the onion payload is shifted (dropping more junk), the new hop payload is added to the front, and the whole onion payload is encrypted with the rho byte stream for the next hop.

For the final hop, the HMAC included in Step #3 over the plain-text instructions is actually *all zero*. The final hop uses this signal to determine that it is indeed the final hop in the route. Alternatively, the fact that the `short_chan_id` included in the payload to denote the "next hop" is all zero can be used as well.

Note that at each phase the mu key is used to generate an HMAC over the *encrypted* (from the point of view of the node processing the payload) onion packet, as well as over the contents of the packet with a single layer of encryption removed. This outer HMAC allows the node processing the packet to verify the integrity of the onion packet (no bytes modified). The inner HMAC is then revealed during the inverse of the "shift and encrypt" routine described previously, which serves as the *outer* HMAC for the next hop.

Wrapping Dina's Hop Payload

As a reminder, the onion is wrapped by starting at the end of the path from Dina, the final node or recipient. Then the path is built in reverse all the way back to the sender, Alice.

Alice starts with an empty 1,300-byte field, the fixed-length *onion payload*. Then, Alice fills the onion payload with a pseudorandom byte stream "filler" that is generated from the pad key.

This is shown in Filling the onion payload with a random byte stream.

NOTE Random byte stream generation uses the ChaCha20 algorithm, as a cryptographic secure pseudorandom number generator (CSPRNG). Such an algorithm will generate a deterministic, long, nonrepeating stream of seemingly random bytes from an initial seed. The details are specified in BOLT #4: Onion Routing, Pseudo Random Byte Stream.

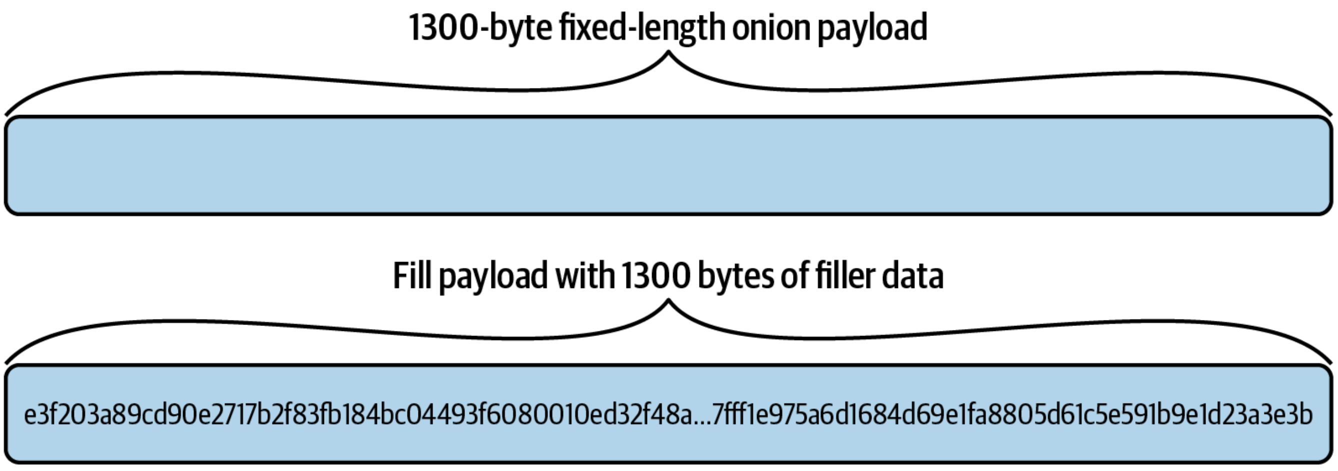

1300-byte fixed-length onion payload
Fill payload with 1300 bytes of filler data
e3f203a89cd90e2717b2f83fb184bc04493f6080010ed32f48a...7fff1e975a6d1684d69e1fa8805d61c5e591b9e1d23a3e3b

Figure 81. Filling the onion payload with a random byte stream

Alice will now insert Dina's hop payload into the left side of the 1,300-byte array, shifting the filler to the right and discarding anything that overflows. This is visualized in Adding Dina's hop payload.

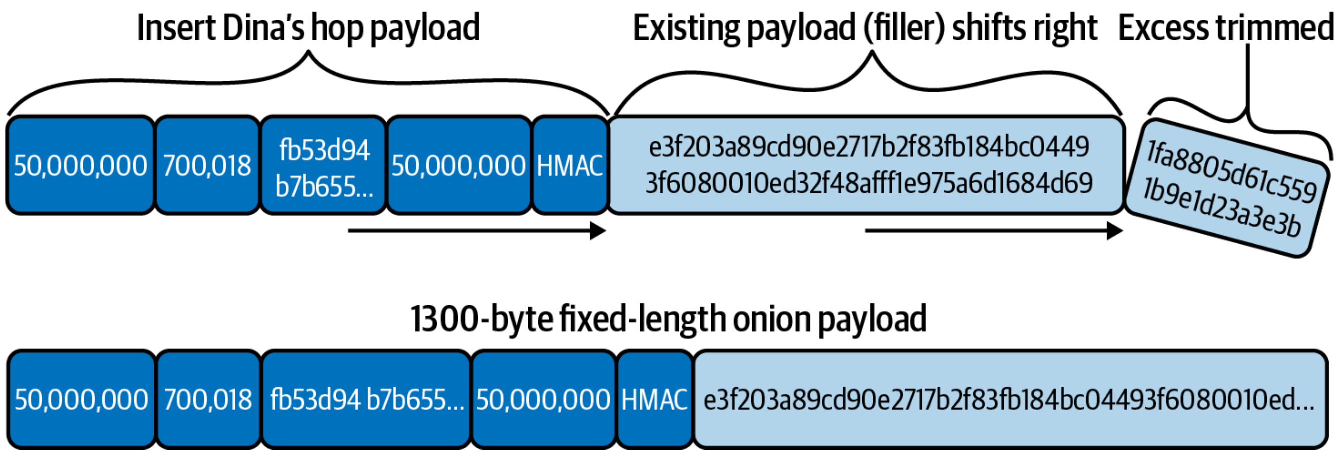

Insert Dina's hop payload
Existing payload (filler) shifts right
Excess trimmed
50,000,000
700,018
fb53d94 b7b655...
50,000,000
HMAC
e3f203a89cd90e2717b2f83fb184bc0449 3f6080010ed32f48afff1e975a6d1684d69
1fa8805d61c559 1b9e1d23a3e3b
1300-byte fixed-length onion payload
50,000,000
700,018
fb53d94 b7b655...
50,000,000
HMAC
e3f203a89cd90e2717b2f83fb184bc04493f6080010ed...

Figure 82. Adding Dina's hop payload

Another way to look at this is that Alice measures the length of Dina's hop payload, shifts the filler right to create an equal space in the left side of the onion payload, and inserts Dina's payload in that space.

Next row down we see the result: the 1,300 byte onion payload contains Dina's hop payload and then the filler byte stream filling up the rest of the space.

Next, Alice obfuscates the entire onion payload so that *only Dina* can read it.

To do this, Alice generates a byte stream using the rho key (which Dina also knows). Alice uses a bitwise exclusive or (XOR) between the bits of the onion payload and the byte stream created from rho. The result appears like a random (or encrypted) byte stream of 1,300 bytes length. This step is shown in Obfuscating the onion payload.

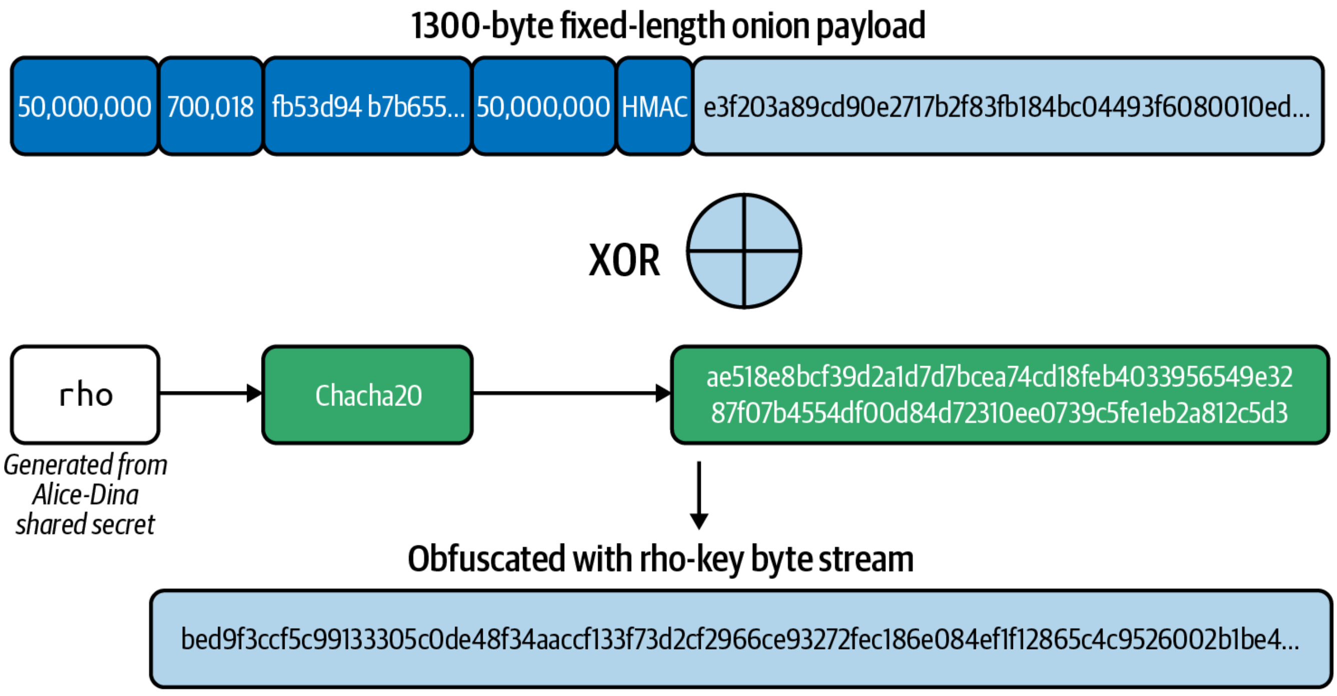

1300-byte fixed-length onion payload
50,000,000
700,018
fb53d94 b7b655...
50,000,000
HMAC
e3f203a89cd90e2717b2f83fb184bc04493f6080010ed...
XOR
rho
Generated from Alice-Dina shared secret
Chacha20
ae518e8bcf39d2a1d7d7bcea74cd18feb4033956549e32 87f07b4554df00d84d72310ee0739c5fe1eb2a812c5d3
Obfuscated with rho-key byte stream
bed9f3ccf5c99133305c0de48f34aaccf133f73d2cf2966ce93272fec186e084ef1f12865c4c9526002b1be4...

Figure 83. Obfuscating the onion payload

One of the properties of XOR is that if you do it twice, you get back to the original data. As we will see in Bob De-Obfuscates His Hop Payload, if Dina applies the same XOR operation with the byte stream generated from rho, it will reveal the original onion payload.

TIP

XOR is an *involutory* function, which means that if it is applied twice, it undoes itself. Specifically XOR(XOR(*a*, *b*), *b*) = *a*. This property is used extensively in symmetric-key cryptography.

Because only Alice and Dina have the rho key (derived from Alice and Dina's shared secret), only they can do this. Effectively, this encrypts the onion payload for Dina's eyes only.

Finally, Alice calculates a hash-based message authentication code (HMAC) for Dina's payload, which uses the mu key as its initialization key. This is shown in Adding an HMAC integrity checksum to Dina's hop payload.

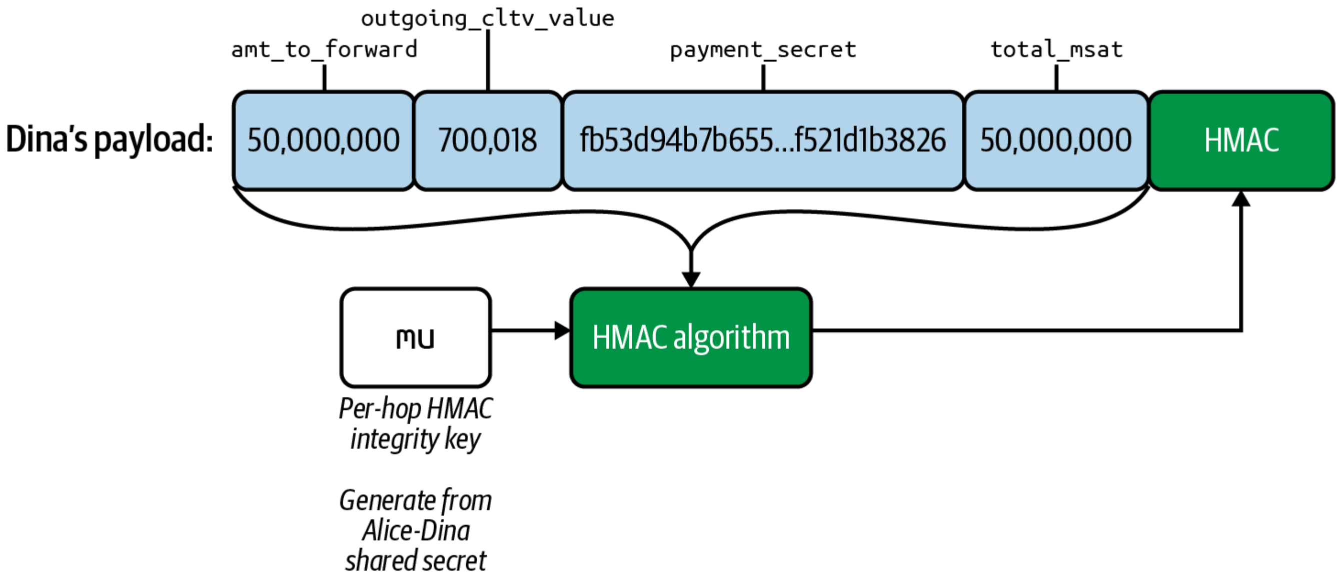


Figure 84. Adding an HMAC integrity checksum to Dina's hop payload

Onion routing replay protection and detection

The HMAC acts as a secure checksum and helps Dina verify the integrity of the hop payload. The 32-byte HMAC is appended to Dina's hop payload. Note that we compute the HMAC over the *encrypted* data rather then over the plain-text data. This is known as *encrypt-then-mac* and is the recommended way to use a MAC, as it provides both plain-text *and* ciphertext integrity.

Modern authenticated encryption also allows for the use of an optional set of plaintext bytes to also be authenticated, known as *associated data.* In practice, this is usually something like a plain-text packet header or other auxiliary information. By including this associated data in the payload to be authenticated (MAC'ed), the verifier of the MAC ensures that this associated data hasn't been tampered with (e.g., swapping out the plain-text header on an encrypted packet).

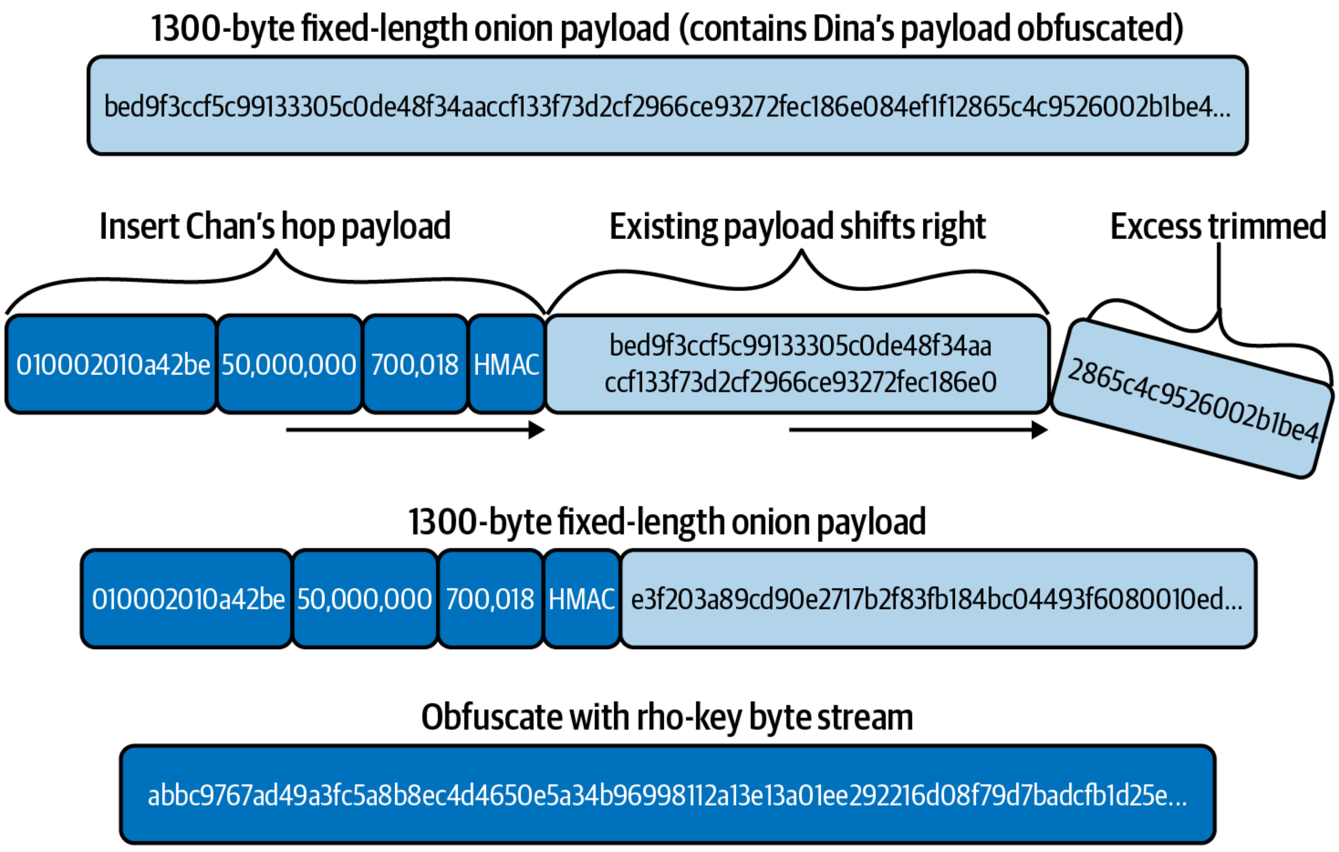
In the context of the Lightning Network, this associated data is used to *strengthen* the replay protection of this scheme. As we'll learn in the following, replay protection ensures that an attacker can't *retransmit* (replay) a packet into the network and observe its resulting path. Instead, intermediate nodes are able to use the defined replay protection measures to detect and reject a replayed packet. The base Sphinx packet format uses a log of all the ephemeral secret keys used to detect replays. If a secret key is ever used again, then the node can detect it and reject the packet.

The nature of HTLCs in the Lightning Network allows us to further strengthen the replay protection by adding an additional *economic* incentive. Remember that the payment hash of an HTLC can only ever safely be used (for a complete payment) once. If a payment hash is used again and traverses a node that has already seen the payment secret for that hash, then they can simply pull the funds and collect the entire payment amount without forwarding! We can use this fact to strengthen the replay protection by requiring that the *payment hash* is included in our HMAC computation as the associated data. With this added step, attempting to replay an onion packet also requires the sender to commit to using the *same* payment hash. As a result, on top of the normal replay protection, an attacker also stands to lose the entire amount of the HTLC replayed.

One consideration with the ever-increasing set of session keys stored for replay protection is: are nodes able to reclaim this space? In the context of the Lightning Network, the answer is: yes! Once again, due to the unique attributes of the HTLC construct, we can make a further improvement over the base Sphinx protocol. Given that HTLCs are *time-locked* contracts based on the absolute block height, once an HTLC has expired, then the contract is effectively permanently closed. As a result, nodes can use this CLTV (CHECKLOCKTIMEVERIFY operator) expiration height as an indicator to know when it's safe to garbage collect an entry in the anti-replay log.

Wrapping Chan's Hop Payload

In Wrapping the onion for Chan we see the steps used to wrap Chan's hop payload in the onion. These are the same steps Alice used to wrap Dina's hop payload.



1300-byte fixed-length onion payload (contains Dina's payload obfuscated)
bed9f3ccf5c99133305c0de48f34aaccf133f73d2cf2966ce93272fec186e084ef1f12865c4c9526002b1be4...
Insert Chan's hop payload
Existing payload shifts right
Excess trimmed
010002010a42be
50,000,000
700,018
HMAC
bed9f3ccf5c99133305c0de48f34aa ccf133f73d2cf2966ce93272fec186e0
2865c4c9526002b1be4
1300-byte fixed-length onion payload
010002010a42be
50,000,000
700,018
HMAC
e3f203a89cd90e2717b2f83fb184bc04493f6080010ed...
Obfuscate with rho-key byte stream
abbc9767ad49a3fc5a8b8ec4d4650e5a34b96998112a13e13a01ee292216d08f79d7badcfb1d25e...

Figure 85. Wrapping the onion for Chan

Alice starts with the 1,300 onion payload created for Dina. The first 65 (or fewer) bytes of this are Dina's payload obfuscated and the rest is filler. Alice must be careful not to overwrite Dina's payload.

Next, Alice needs to locate the ephemeral public key (which was generated at the very start for each hop) that will be prepended to the routing packet at this hop.

Remember that rather than include a unique ephemeral public key (that the sender and intermediate node use in an ECDH operation to generate a shared secret), Sphinx uses a single ephemeral public key that is deterministically randomized at each hop.

When processing the packet, Dina will use her shared secret and public key to derive the blinding value (`b_dina`) and use that to re-randomize the ephemeral public key, in an identical operation to what Alice performs during initial packet construction.

Alice adds an inner HMAC checksum to Chan's payload and inserts it at the "front" (left side) of the onion payload, shifting the existing payload to the right by an equal amount. Remember that there are effectively *two* HMACs used in the scheme: the outer HMAC and the inner HMAC. In this case, Chan's *inner* HMAC is actually Dina's *outer* HMAC.

Now Chan's payload is in the front of the onion. When Chan sees this, he has no idea how many payloads came before or after. It looks like the first of 20 hops always!

Next, Alice obfuscates the entire payload by XOR with the byte stream generated from the Alice-Chan rho key. Only Alice and Chan have this rho key, and only they can produce the byte stream to obfuscate and de-obfuscate the onion. Finally, as we did in the earlier step, we compute Chan's outer HMAC, which is what she'll use to verify the integrity of the encrypted onion packet.

Wrapping Bob's Hop Payload

In Wrapping the onion for Bob we see the steps used to wrap Bob's hop payload in the onion.

All right, by now this is easy!

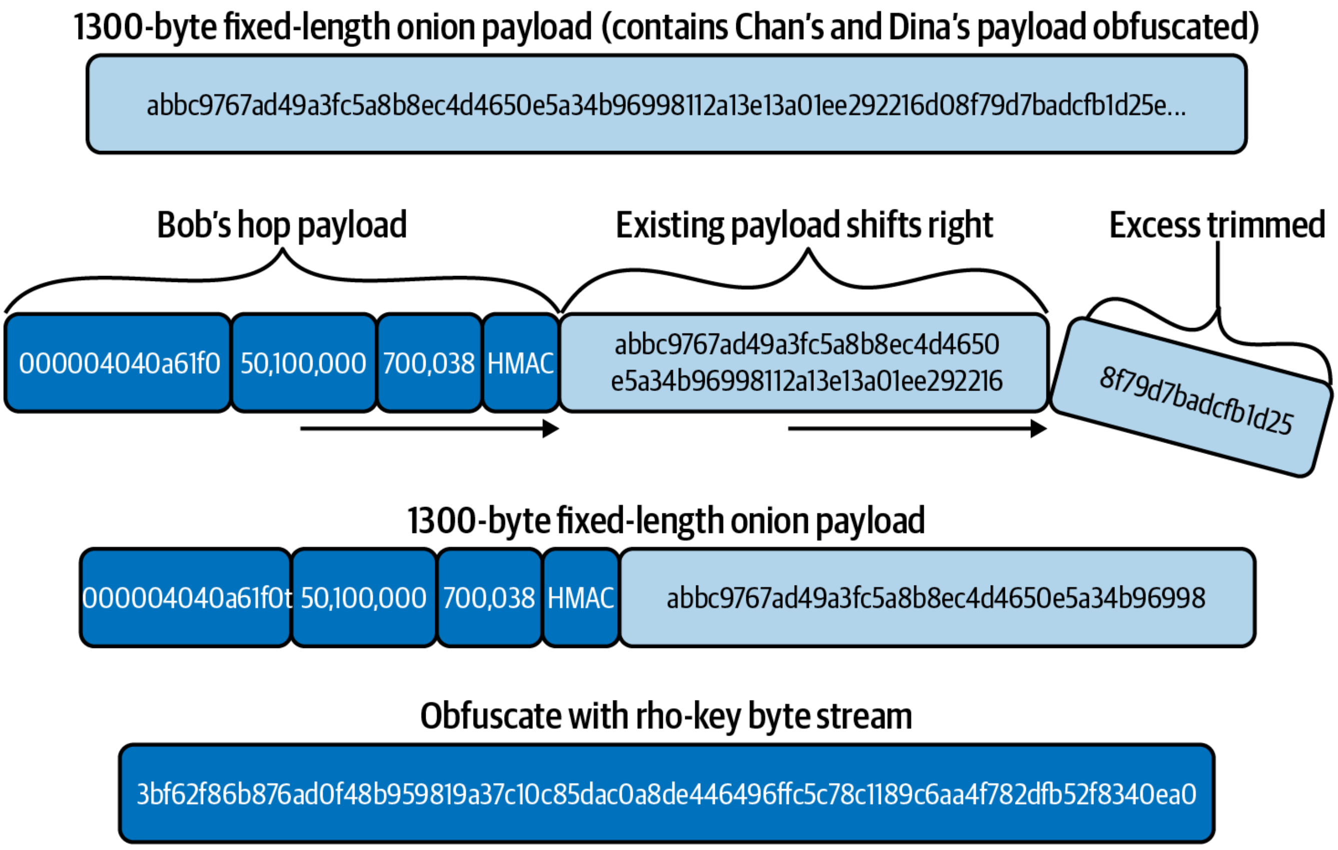

1300-byte fixed-length onion payload (contains Chan's and Dina's payload obfuscated)
abbc9767ad49a3fc5a8b8ec4d4650e5a34b96998112a13e13a01ee292216d08f79d7badcfb1d25e...
Bob's hop payload
Existing payload shifts right
Excess trimmed
000004040a61f0
50,100,000
700,038
HMAC
abbc9767ad49a3fc5a8b8ec4d4650 e5a34b96998112a13e13a01ee292216
8f79d7badcfb1d25
1300-byte fixed-length onion payload
000004040a61f0t
50,100,000
700,038
HMAC
abbc9767ad49a3fc5a8b8ec4d4650e5a34b96998
Obfuscate with rho-key byte stream
3bf62f86b876ad0f48b959819a37c10c85dac0a8de446496ffc5c78c1189c6aa4f782dfb52f8340ea0

Figure 86. Wrapping the onion for Bob

Start with the onion payload (obfuscated) containing Chan's and Dina's hop payloads.

Obtain the session key for this hop dervied from the blinding factor generated by the prior hop. Include the prior hop's outer HMAC as this hop's inner HMAC. Insert Bob's hop payload at the beginning and shift everything else over to the right, dropping a Bob-hop-payload-size chunk from the end (it was filler anyway).

Obfuscate the whole thing XOR with the rho key from the Alice-Bob shared secret so that only Bob can unwrap this.

Calculate the outer HMAC and stick it on the end of Bob's hop payload.

The Final Onion Packet

The final onion payload is ready to be sent to Bob. Alice doesn't need to add any more hop payloads.

Alice calculates an HMAC for the onion payload to cryptographically secure it with a checksum that Bob can verify.

Alice adds a 33-byte public session key that will be used by each hop to generate a shared secret and the rho, mu, and pad keys.

Finally Alice puts the onion version number (0 currently) in the front. This allows for future upgrades of the onion packet format.

The result can be seen in The onion packet.

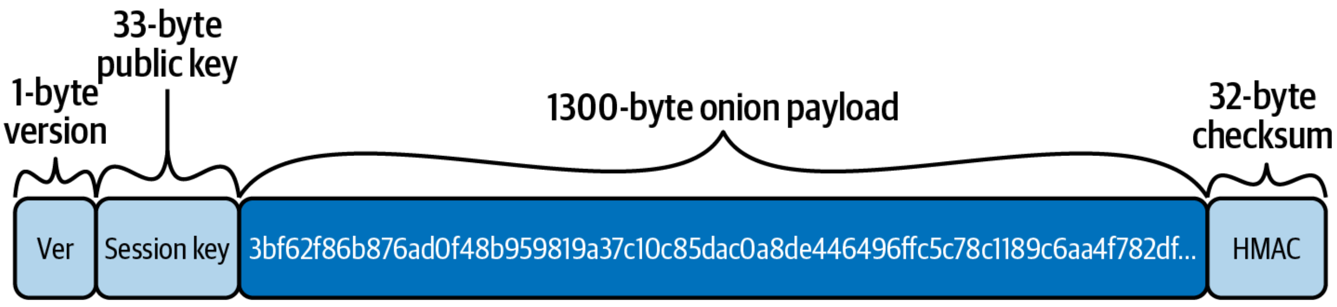


Figure 87. The onion packet

Sending the Onion

In this section we will look at how the onion packet is forwarded and how HTLCs are deployed along the path.

The update_add_htlc Message

Onion packets are sent as part of the update_add_htlc message. If you recall from The update_add_HTLC Message, in Channel Operation and Payment Forwarding, we saw the contents of the update_add_htlc message are as follows:

```
[channel_id:channel_id]
[u64:id]
[u64:amount_msat]
[sha256:payment_hash]
[u32:cltv_expiry]
[1366*byte:onion_routing_packet]
```

You will recall that this message is sent by one channel partner to ask the other channel partner to add an HTLC. This is how Alice will ask Bob to add an HTLC to pay Dina. Now you understand the purpose of the last field, onion_routing_packet, which is 1,366 bytes long. It's the fully wrapped onion packet we just constructed!

Alice Sends the Onion to Bob

Alice will send the update_add_htlc message to Bob. Let's look at what this message will contain:

channel_id

This field contains the Alice-Bob channel ID, which in our example is 0000031e192ca1 (see A detailed path constructed from gossiped channel and node information).

id

The ID of this HTLC in this channel, starting at 0.

amount_msat

The amount of the HTLC: 50,200,000 millisatoshis.

payment_hash

The RIPEMD160(SHA-256) payment hash, 9e017f6767971ed7cea17f98528d5f5c0ccb2c71.

cltv_expiry

The expiry timelock for the HTLC will be 700,058. Alice adds 20 blocks to the expiry set in Bob's payload according to Bob's negotiated cltv_expiry_delta.

onion_routing_packet

The final onion packet Alice constructed with all the hop payloads!

Bob Checks the Onion

As we saw in Channel Operation and Payment Forwarding, Bob will add the HTLC to the commitment transactions and update the state of the channel with Alice.

Bob will unwrap the onion he received from Alice as follows:

1. Bob takes the session key from the onion packet and derives the Alice-Bob shared secret.
2. Bob generates the mu key from the shared secret and uses it to verify the onion packet HMAC checksum.

Now that Bob has generated the shared key and verified the HMAC, he can start unwrapping the

1,300 byte onion payload inside the onion packet. The goal is for Bob to retrieve his own hop payload and then forward the remaining onion to the next hop.

If Bob extracts and removes his hop payload, the remaining onion will not be 1,300 bytes, it will be shorter! So the next hop will know that they are not the first hop and will be able to detect how long the path is. To prevent this, Bob needs to add more filler to refill the onion.

Bob Generates Filler

Bob generates filler in a slightly different way than Alice, but following the same general principle.

First, Bob *extends* the onion payload by 1,300 bytes and fills them with 0 values. Now the onion packet is 2,600 bytes long, with the first half containing the data Alice sent and the next half containing zeroes. This operation is shown in Bob extends the onion payload by 1,300 (zero-filled) bytes.

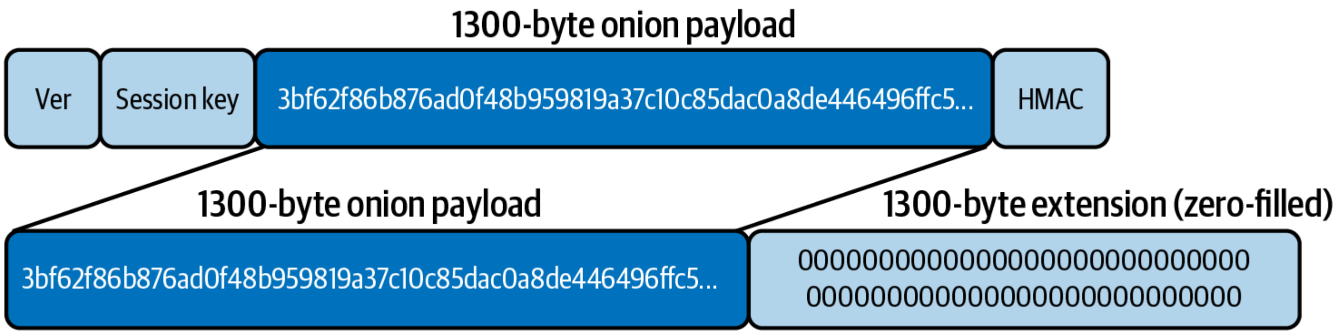

1300-byte onion payload
Ver
Session key
3bf62f86b876ad0f48b959819a37c10c85dac0a8de446496ffc5...
HMAC
1300-byte onion payload
1300-byte extension (zero-filled)
3bf62f86b876ad0f48b959819a37c10c85dac0a8de446496ffc5...
0000000000000000000000000000
000000000000000000000000000

Figure 88. Bob extends the onion payload by 1,300 (zero-filled) bytes

This empty space will become obfuscated and turn into "filler" by the same process that Bob uses to de-obfuscate his own hop payload. Let’s see how that works.

Bob De-Obfuscates His Hop Payload

Next, Bob will generate the rho key from the Alice-Bob shared key. He will use this to generate a 2,600 byte stream, using the ChaCha20 algorithm.

TIP The first 1,300 bytes of the byte stream generated by Bob are exactly the same as those generated by Alice using the rho key.

Next, Bob applies the 2,600 bytes of the rho byte stream to the 2,600-byte onion payload with a bitwise XOR operation.

The first 1,300 bytes will become de-obfuscated by this XOR operation, because it is the same operation Alice applied and XOR is involutory. So Bob will *reveal* his hop payload followed by some data that seems scrambled.

At the same time, applying the rho byte stream to the 1,300 zeroes that were added to the onion payload will turn them into seemingly random filler data. This operation is shown in Bob de-obfuscates the onion, obfuscates the filler.

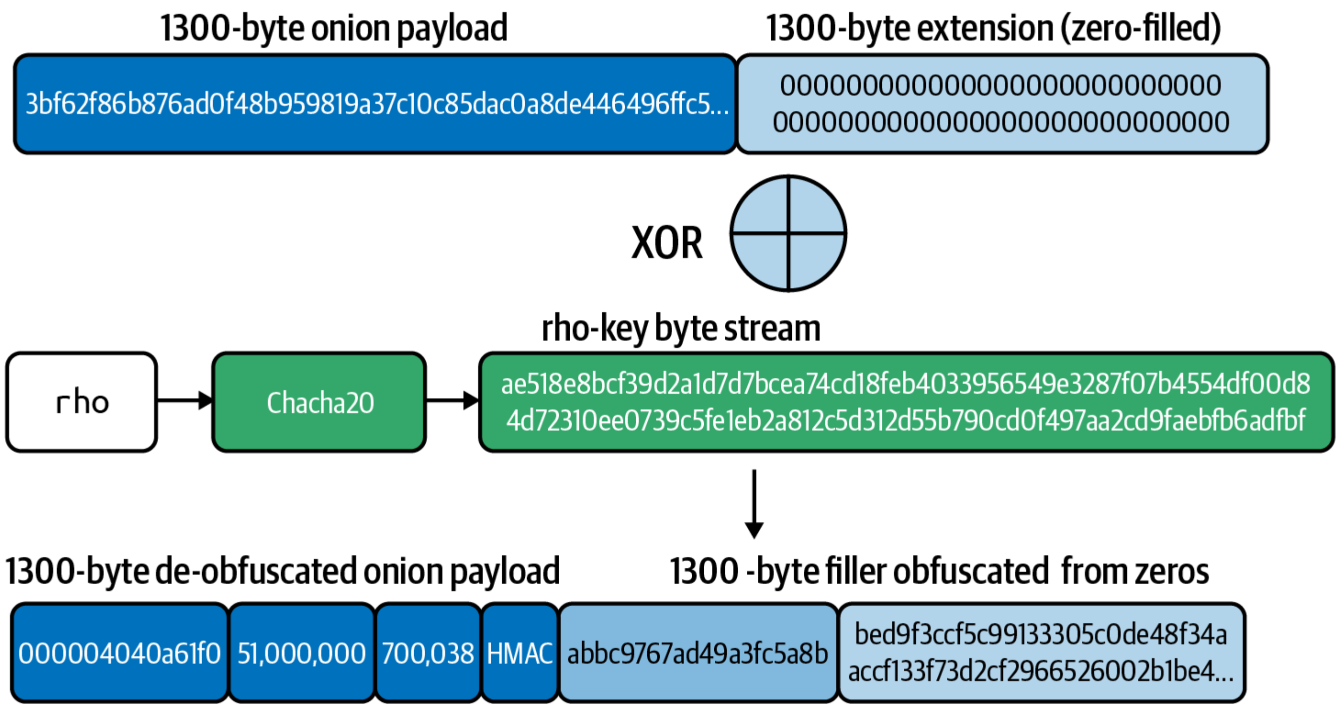

1300-byte onion payload
3bf62f86b876ad0f48b959819a37c10c85dac0a8de446496ffc5...
1300-byte extension (zero-filled)
0000000000000000000000000000
0000000000000000000000000000
XOR
rho-key byte stream
rho
Chacha20
ae518e8bcf39d2a1d7d7bcea74cd18feb4033956549e3287f07b4554df00d8
4d72310ee0739c5fe1eb2a812c5d312d55b790cd0f497aa2cd9faebfb6adfbf
1300-byte de-obfuscated onion payload
1300 -byte filler obfuscated from zeros
000004040a61f0
51,000,000
700,038
HMAC
abbc9767ad49a3fc5a8b
bed9f3ccf5c99133305c0de48f34a
accf133f73d2cf2966526002b1be4...

Figure 89. Bob de-obfuscates the onion, obfuscates the filler

Bob Extracts the Outer HMAC for the Next Hop

Remember that an inner HMAC is included for each hop, which will then become the outer HMAC for the *next* hop. In this case, Bob extracts the inner HMAC (he’s already verified the integrity of the encrypted packet with the outer HMAC), and puts it aside because he’ll append it to the de-obfuscated packet to allow Chan to verify the HMAC of his encrypted packet.

Bob Removes His Payload and Left-Shifts the Onion

Now Bob can remove his hop payload from the front of the onion and left-shift the remaining data. An amount of data equal to Bob’s hop payload from the second-half 1,300 bytes of filler will now shift into the onion payload space. This is shown in Bob removes the hop payload and left-shifts the rest, filling the gap with new filler.

Now Bob can keep the first half 1,300 bytes, and discard the extended (filler) 1,300 bytes.

Bob now has a 1,300-byte onion packet to send to the next hop. It is almost identical to the onion payload that Alice had created for Chan, except that the last 65 or so bytes of filler was put there by Bob and will be different.

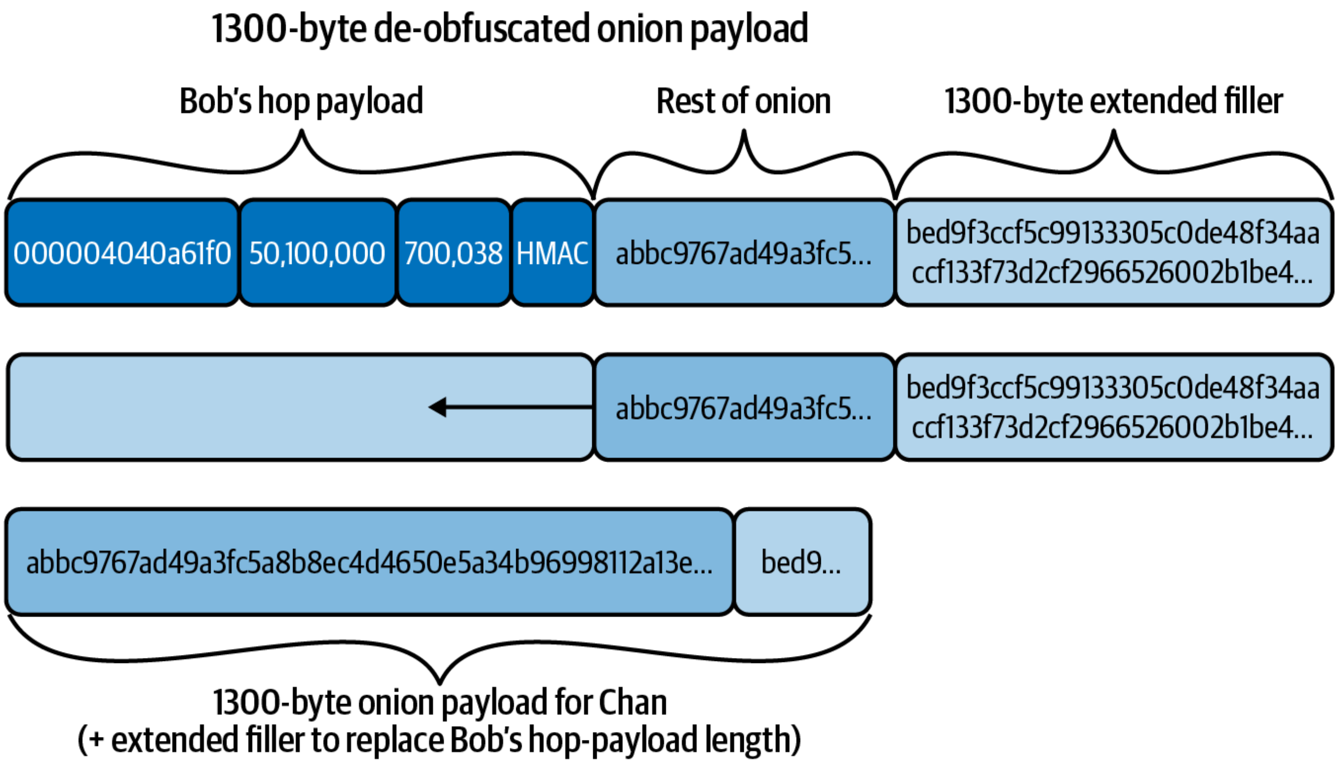

1300-byte de-obfuscated onion payload
Bob's hop payload
Rest of onion
1300-byte extended filler
000004040a61f0
50,100,000
700,038
HMAC
abbc9767ad49a3fc5...
bed9f3ccf5c99133305c0de48f34aa ccf133f73d2cf2966526002b1be4...
abbc9767ad49a3fc5...
bed9f3ccf5c99133305c0de48f34aa ccf133f73d2cf2966526002b1be4...
abbc9767ad49a3fc5a8b8ec4d4650e5a34b96998112a13e...
bed9...
1300-byte onion payload for Chan
(+ extended filler to replace Bob's hop-payload length)

Figure 90. Bob removes the hop payload and left-shifts the rest, filling the gap with new filler

No one can tell the difference between filler put there by Alice and filler put there by Bob. Filler is filler! It's all random bytes anyway. Note that if Bob (or one of Bob's other aliases) is present in the route in two distinct locations, then they can tell the difference because the base protocol always uses the same payment hash across the entire route. Atomic multipath payments (AMPs) and Point Time-Locked Contracts (PTLCs) eliminate the correlation vector by randomizing the payment identifier across each route/hop.

Bob Constructs the New Onion Packet

Bob now copies the onion payload into the onion packet, appends the outer HMAC for chan, re-randomizes the session key (the same way Alice the sender does) with the elliptic curve multiplication operation, and appends a fresh version byte.

To re-randomize the session key, Bob first computes the blinding factor for his hop, using his node public key and the shared secret he derived:

```
b_bob = SHA-256(P_bob || shared_secret_bob)
```

With this generated, Bob now re-randomizes the session key by performing an EC multiplication using his session key and the blinding factor:

```
session_key_chan = session_key_bob * b_bob
```

The `session_key_chan` public key will then be appended to the front of the onion packet for processing by Chan.

Bob Verifies the HTLC Details

Bob's hop payload contains the instructions needed to create an HTLC for Chan.

In the hop payload, Bob finds a short_channel_id, amt_to_forward, and cltv_expiry.

First, Bob checks to see if he has a channel with that short ID. He finds that he has such a channel with Chan.

Next, Bob confirms that the outgoing amount (50,100 satoshis) is less than the incoming amount (50,200 satoshis), and therefore Bob's fee expectations are met.

Similarly, Bob checks that the outgoing cltv_expiry is less than the incoming cltv_expiry, giving Bob enough time to claim the incoming HTLC if there is a breach.

Bob Sends the update_add_htlc to Chan

Bob now constructs and HTLC to send to Chan, as follows:

channel_id

This field contains the Bob-Chan channel ID, which in our example is 000004040a61f0 (see A detailed path constructed from gossiped channel and node information).

id

The ID of this HTLC in this channel, starting at 0.

amount_msat

The amount of the HTLC: 50,100,000 millisatoshis.

payment_hash

The RIPEMD160(SHA-256) payment hash, 9e017f6767971ed7cea17f98528d5f5c0ccb2c71. This is the same as the payment hash from Alice's HTLC.

cltv_expiry

The expiry timelock for the HTLC will be 700,038.

onion_routing_packet

The onion packet Bob reconstructed after removing his hop payload.

Chan Forwards the Onion

Chan repeats the exact same process as Bob:

1. Chan receives the update_add_htlc and processes the HTLC request, adding it to commitment transactions.
2. Chan generates the Alice-Chan shared key and the mu subkey.
3. Chan verifies the onion packet HMAC, then extracts the 1,300-byte onion payload
4. Chan extends the onion payload by 1,300 extra bytes, filling it with zeroes.

5. Chan uses the rho key to produce 2,600 bytes.
6. Chan uses the generated byte stream to XOR and de-obfuscate the onion payload. Simultaneously, the XOR operation obfuscates the extra 1,300 zeroes, turning them into filler.
7. Chan extracts the inner HMAC in the payload, which will become the outer HMAC for Dina.
8. Chan removes his hop payload and left-shifts the onion payload by the same amount. Some of the filler generated in the 1,300 extended bytes moves into the first-half 1,300 bytes, becoming part of the onion payload.
9. Chan constructs the onion packet for Dina with this onion payload.
10. Chan builds an update_add_htlc message for Dina and inserts the onion packet into it.
11. Chan sends the update_add_htlc to Dina.
12. Chan re-randomizes the session key as Bob did in the prior hop for Dina.

Dina Receives the Final Payload

When Dina receives the update_add_htlc message from Chan, she knows from the payment_hash that this is a payment for her. She knows she is the last hop in the onion.

Dina follows the exact same process as Bob and Chan to verify and unwrap the onion, except she doesn't construct new filler and doesn't forward anything. Instead, Dina responds to Chan with update_fulfill_htlc to redeem the HTLC. The update_fulfill_htlc will flow backward along the path until it reaches Alice. All the HTLCs are redeemed and channel balances are updated. The payment is complete!

Returning Errors

This far we've looked at the forward propagation of the onion establishing the HTLCs and the backward propagation of the payment secret unwinding the HTLCs once payment is successful.

There is another very important function of onion routing: *error return*. If there is a problem with the payment, onion, or hops, we must propagate an error backwards to inform all nodes of the failure and unwind any HTLCs.

Errors generally fall into three categories: onion failures, node failures, and channel failures. These furthermore may be subdivided into permanent and transient errors. Finally, some errors contain channel updates to help with future payment delivery attempts.

NOTE

Unlike messages in the peer-to-peer (P2P) protocol (defined in BOLT #2: Peer Protocol for Channel Management), errors are not sent as P2P messages but are wrapped inside onion return packets and follow the reverse of the onion path (back-propagating).

Error return is defined in BOLT #4: Onion Routing, Returning Errors.

Errors are encoded by the returning node (the one that discovered an error) in a *return packet* as follows:

```
    [32*byte:hmac]
    [u16:failure_len]
    [failure_len*byte:failuremsg]
    [u16:pad_len]
    [pad_len*byte:pad]
```

The return packet HMAC verification checksum is calculated with the um key, generated from the shared secret established by the onion.

TIP The um key name is the reverse of the mu name, indicating the same use but in the opposite direction (back-propagation).

Next, the returning node generates an ammag (inverse of the word "gamma") key and obfuscates the return packet using an XOR operation with a byte stream generated from ammag.

Finally the return node sends the return packet to the hop from which it received the original onion.

Each hop receiving an error will generate an ammag key and obfuscate the return packet again using an XOR operation with the byte stream from ammag.

Eventually, the sender (origin node) receives a return packet. It will then generate ammag and um keys for each hop and XOR de-obfuscate the return error iteratively until it reveals the return packet.

Failure Messages

The failuremsg is defined in BOLT #4: Onion Routing, Failure Messages.

A failure message consists of a two-byte failure code followed by the data applicable to that failure type.

The top byte of the failure_code is a set of binary flags that can be combined (with binary OR):

0x8000 (`BADONION`)
: Unparsable onion encrypted by sending peer

0x4000 (`PERM`)
: Permanent failure (otherwise transient)

0x2000 (`NODE`)
: Node failure (otherwise channel)

0x1000 (`UPDATE`)
: New channel update enclosed

The failure types shown in Onion Error Failure Types are currently defined.

Onion Error Failure Types

Table 3. Onion error failure types

Type	Symbolic name	Meaning
PERM|1	invalid_realm	The `realm` byte was not understood by the processing node
NODE|2	temporary_node_failure	General temporary failure of the processing node
PERM|NODE|2	permanent_node_failure	General permanent failure of the processing node
PERM|NODE|3	required_node_fea&#x2060;ture_&#x200b;missing	The processing node has a required feature which was not in this onion
BADONION|PERM|4	invalid_onion_version	The `version` byte was not understood by the processing node
BADONION|PERM|5	invalid_onion_hmac	The HMAC of the onion was incorrect when it reached the processing node
BADONION|PERM|6	invalid_onion_key	The ephemeral key was unparsable by the processing node
UPDATE|7	temporary_channel_&#x200b;fail&#x2060;ure	The channel from the processing node was unable to handle this HTLC, but may be able to handle it, or others, later
PERM|8	permanent_channel_&#x200b;fail&#x2060;ure	The channel from the processing node is unable to handle any HTLCs
PERM|9	required_channel_&#x200b;fea&#x2060;ture_missing	The channel from the processing node requires features not present in the onion
PERM|10	unknown_next_peer	The onion specified a `short_channel_id` which doesn't match any leading from the processing node
UPDATE|11	amount_below_minimum	The HTLC amount was below the `htlc_minimum_msat` of the channel from the processing node

Type	Symbolic name	Meaning
UPDATE|12	fee_insufficient	The fee amount was below that required by the channel from the processing node
UPDATE|13	incorrect_cltv_expiry	The `cltv_expiry` does not comply with the `cltv_expiry_delta` required by the channel from the processing node
UPDATE|14	expiry_too_soon	The CLTV expiry is too close to the current block height for safe handling by the processing node
PERM|15	incor&#x2060;rect_or_unknown_&#x200b;pay&#x2060;ment_details	The `payment_hash` is unknown to the final node, the `payment_secret` doesn't match the `payment_hash`, the amount for that `payment_hash` is incorrect, or the CLTV expiry of the HTLC is too close to the current block height for safe handling
18	final_incor&#x2060;rect_&#x200b;cltv_expiry	The CLTV expiry in the HTLC doesn't match the value in the onion
19	final_incor&#x2060;rect_&#x200b;htlc_amount	The amount in the HTLC doesn't match the value in the onion
UPDATE|20	channel_disabled	The channel from the processing node has been disabled
21	expiry_too_far	The CLTV expiry in the HTLC is too far in the future
PERM|22	invalid_onion_payload	The decrypted onion per-hop payload was not understood by the processing node or is incomplete
23	mpp_timeout	The complete amount of the multipart payment was not received within a reasonable time

Stuck payments

In the current implementation of the Lightning Network, there is a possibility that a payment attempt becomes *stuck*: neither fulfilled nor cancelled by an error. This can happen due to a bug on an intermediary node, a node going offline while handling HTLCs, or a malicious node holding HTLCs without reporting an error. In all of these cases, the HTLC cannot be resolved until it expires. The timelock (CLTV) that is set on every HTLC helps resolve this condition (among other possible HTLC routing and channel failures).

However, this means that the sender of the HTLC has to wait until expiry, and the funds committed to that HTLC remain unavailable until the HTLC expires. Furthermore, the sender *cannot retry* that same payment, because if they do, they run the risk of *both* the original and the retried payment succeeding—the recipient gets paid twice. This is because, once sent, an HTLC cannot be "cancelled" by the sender—it either has to fail or expire. Stuck payments, while rare, create an unwanted user experience, where the user's wallet cannot pay or cancel a payment.

One proposed solution to this problem is called *stuckless payments*, and it depends on Point Time-Locked Contracts (PTLCs), which are payment contracts that use a different cryptographic primitive than HTLCs (i.e., point addition on the elliptic curve instead of a hash and secret preimage). PTLCs are cumbersome using ECDSA but much easier with Bitcoin's Taproot and Schnorr signature features, which were recently locked in for activation in November 2021. It is expected that PTLCs will be implemented in the Lightning Network after these Bitcoin features become activated.

Keysend Spontaneous Payments

In the payment flow described earlier in the chapter, we assumed that Dina received an invoice from Alice "out of band," or obtained it via some mechanism unrelated to the protocol (typically copy/paste or QR code scans). This trait means that the payment process always takes two steps: first, the sender obtains an invoice, and second, uses the payment hash (encoded in the invoice) to successfully route an HTLC. The extra round trip required to obtain an invoice before making a payment may be a bottleneck in applications that involve streaming micropayments over Lightning. What if we could just "push" a payment over spontaneously, without having to obtain an invoice from the recipient first? The `keysend` protocol is an end-to-end extension (only the sender and receiver are aware) to the Lightning protocol that enables spontaneous push payments.

Custom Onion TLV Records

The modern Lightning protocol uses the TLV (Type-Length-Value) encoding in the onion to encode information that tells each node *where* and *how* to forward the payment. Leveraging the TLV format, each piece of routing information (like the next node to which to pass the HTLC) is assigned a specific type (or key) encoded as a `BigSize` variable length integer (max sized as as 64-bit integer). These "essential" (reversed values below `65536`) types are defined in BOLT #4, along with the rest of the onion routing details. Onion types with a value greater than `65536` are intended to be used by wallets and applications as "custom records."

Custom records allow payment applications to attach additional metadata or context to a payment as key/value pairs in the onion. Since the custom records are included in the onion payload itself, like all other hop contents, the records are end-to-end encrypted. As the custom records effectively consume a portion of the fixed-size 1300-bytes onion packet, encoding each key and value of each custom record reduces the amount of available space for encoding the rest of the route. In practice, this means that the more onion space used for custom records, the shorter the route can be. Given that each HTLC packet is fixed size, custom records don’t *add* any additional data to an HTLC; rather, they reallocate bytes that would have been filled with random data otherwise.

Sending and Receiving Keysend Payments

A `keysend` payment inverts the typical flow of an HTLC where the receiver reveals a secret preimage to the sender. Instead, the sender includes the preimage *within* the onion to the receiver, and routes the HTLC to the receiver. The receiver then decrypts the onion payload, and uses the included preimage (which *must* match the payment hash of the HTLC) to settle the payment. As a result, `keysend` payments can be carried out without first obtaining an invoice from the receiver, as the preimage is "pushed" over to the receiver. A `keysend` payment uses a TLV custom record type of `5482373484` to encode a 32-byte preimage value.

Keysend and Custom Records in Lightning Applications

Many streaming Lightning applications use the `keysend` protocol to continually stream satoshis to a destination identified by its public key in the network. Typically, an application will also include metadata such as a tipping/donation note or other application-level information in addition to the `keysend` record.

Conclusion

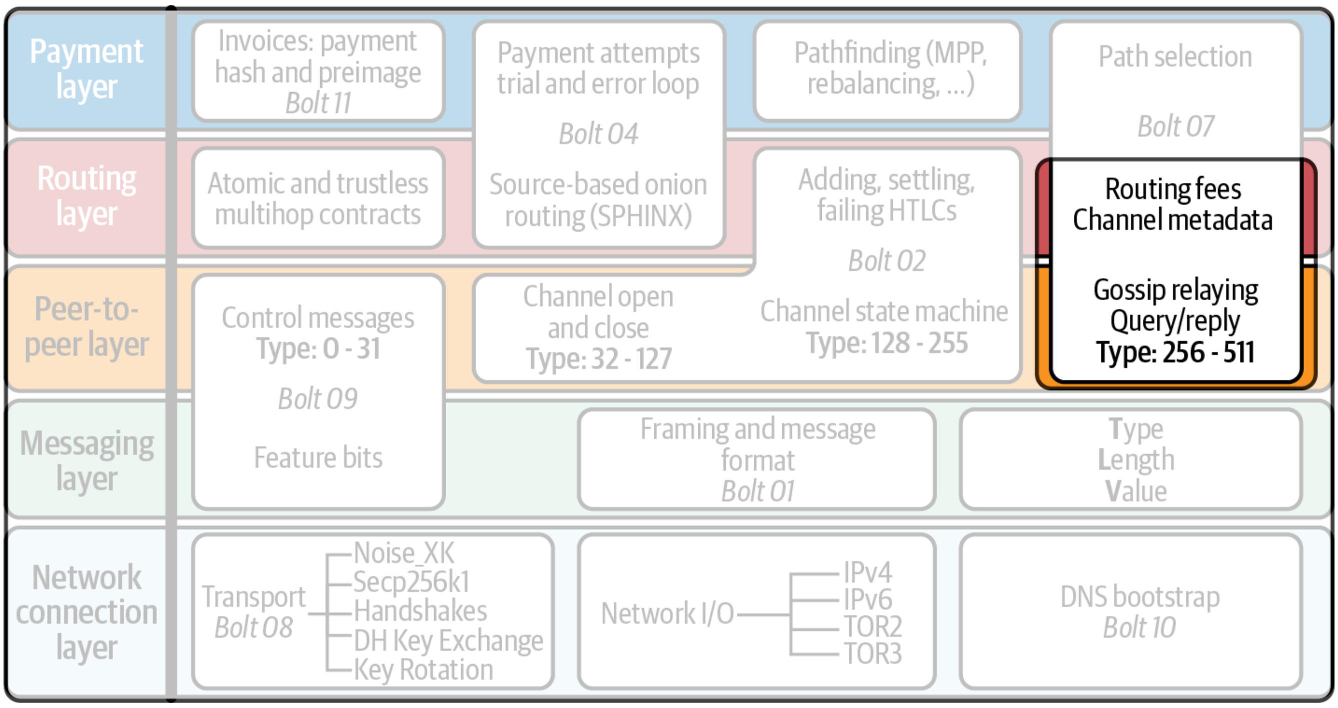
The Lightning Network’s onion routing protocol is adapted from the Sphinx protocol to better serve the needs of a payment network. As such, it offers a huge improvement in privacy and counter-surveillance compared to the public and transparent Bitcoin blockchain.

In Pathfinding and Payment Delivery we will see how the combination of source routing and onion routing is used by Alice to find a good path and route the payment to Dina. To find a path, Alice first needs to learn about the network topology, which is the topic of Gossip and the Channel Graph.

Gossip and the Channel Graph

In this chapter we will describe the Lightning Network's gossip protocol and how it is used by nodes to construct and maintain a channel graph. We will also review the DNS bootstrap mechanism used to find peers to "gossip" with.

The "Routing fees and Gossip relaying" section is highlighted by an outline spanning the routing layer and peer-to-peer layer of Gossip protocol in the Lightning protocol suite.



Payment layer
Invoices: payment hash and preimage Bolt 11
Payment attempts trial and error loop Bolt 04
Pathfinding (MPP, rebalancing, ...)
Path selection Bolt 07
Routing layer
Atomic and trustless multihop contracts
Source-based onion routing (SPHINX)
Adding, settling, failing HTLCs Bolt 02
Routing fees Channel metadata
Peer-to-peer layer
Control messages Type: 0 - 31 Bolt 09 Feature bits
Channel open and close Type: 32 - 127
Channel state machine Type: 128 - 255
Gossip relaying Query/reply Type: 256 - 511
Messaging layer
Framing and 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 91. Gossip protocol in the Lightning protocol suite

As we've learned already, the Lightning Network uses a source-based onion routing protocol to deliver a payment from a sender to the recipient. To do this, the sending node must be able to construct a path of payment channels that connects it with the recipient, as we will see in Pathfinding and Payment Delivery. Thus, the sender has to be able to map the Lightning Network by constructing a channel graph. The *channel graph* is the interconnected set of publicly advertised channels and the nodes that these channels interlink.

As channels are backed by a funding transaction that is happening on-chain, one might falsely believe that Lightning nodes could just extract the existing channels from the Bitcoin blockchain. However this is only possible to a certain extent. The funding transactions are Pay-to-Witness-Script-Hash (P2WSH) addresses, and the nature of the script (a 2-of-2 multisig) will only be revealed once the funding transaction output is spent. Even if the nature of the script were known, it's important to remember that not all 2-of-2 multisig scripts correspond to payment channels.

There are even more reasons why looking at the Bitcoin blockchain might not be helpful. For example, on the Lightning Network, the Bitcoin keys that are used for signing are rotated by the nodes for every channel and update. Thus, even if we could reliably detect funding transactions on the Bitcoin blockchain, we would not know which two nodes on the Lightning Network own that particular channel.

The Lightning Network solves this problem by implementing a *gossip protocol*. Gossip protocols are typical for peer-to-peer (P2P) networks and allow nodes to share information with the whole network with just a few direct connections to peers. Lightning nodes open encrypted peer-to-peer connections to each other and share (gossip) information that they have received from other peers. As soon as a node wants to share some information, for example, about a newly created channel, it sends a message to all its peers. Upon receiving a message, a node decides if the received message was novel and, if so, forwards the information to its peers. In this way, if the peer-to-peer network is well connected, all new information that is necessary for the operation of the network will eventually be propagated to all other peers.

Obviously, if a new peer joins the network for the first time, it needs to know some other peers on the network, so it can connect to others and participate in the network.

In this chapter, we'll explore exactly *how* Lightning nodes discover each other, discover and update their node status, and communicate with one another.

When most refer to the *network* part of the Lightning Network, they're referring to the *channel graph* which itself is a unique authenticated data structure *anchored* in the base Bitcoin blockchain.

However, the Lightning Network is also a peer-to-peer network of nodes that gossip information about payment channels and nodes. Usually, for two peers to maintain a payment channel they need to talk to each other directly, which means that there will be a peer connection between them. This suggests that the channel graph is a subnetwork of the peer-to-peer network. However, this is not true because payment channels can remain open even if one or both peers go temporarily offline.

Let's revisit some of the terminology that we have used throughout the book, specifically looking at what they mean in terms of the channel graph and the peer-to-peer network (see Terminology of the different networks).

Table 4. Terminology of the different networks

Channel graph	**Peer-to-peer network**
channel	connection
open	connect
close	disconnect
funding transaction	encrypted TCP/IP connection
send	transmit
payment	message

Because the Lightning Network is a peer-to-peer network, some initial bootstrapping is required in order for peers to discover each other. Within this chapter we'll follow the story of a new peer connecting to the network for the first time and examine each step in the bootstrapping process, from initial peer discovery to channel graph syncing and validation.

As an initial step, our new node needs to somehow *discover* at least *one* peer that is already connected to the network and has a full channel graph (as we'll see later, there's no canonical version of the channel graph). Using one of many initial bootstrapping protocols to find that first peer, after a connection is established, our new peer now needs to *download* and *validate* the channel graph. Once the channel graph has been fully validated, our new peer is ready to start opening channels and sending payments on the network.

After initial bootstrap, a node on the network needs to continue to maintain its view of the channel graph by processing new channel routing policy updates, discovering and validating new channels, removing channels that have been closed on-chain, and finally pruning channels that fail to send out a proper "heartbeat" every two weeks or so.

Upon completion of this chapter, you will understand a key component of the peer-to-peer Lightning Network: namely, how peers discover each other and maintain a local copy (perspective) of the channel graph. We'll begin by exploring the story of a new node that has just booted up and needs to find other peers to connect to on the network.

Peer Discovery

In this section, we'll begin to follow a new Lightning node that wishes to join the network through three steps:

1. Discover a set of bootstrap peers
2. Download and validate the channel graph
3. Begin the process of ongoing maintenance of the channel graph itself

P2P Bootstrapping

Before doing any thing else, our new node first needs to discover a set of peers who are already part of the network. We call this process initial peer bootstrapping, and it's something that every peer-to-peer network needs to implement properly to ensure a robust, healthy network.

Bootstrapping new peers to existing peer-to-peer networks is a very well studied problem with several known solutions, each with their own distinct trade-offs. The simplest solution to this problem is simply to package a set of *hardcoded* bootstrap peers into the packaged P2P node software. This is simple in that each new node has a list of bootstrap peers in the software they're running, but rather fragile given that if the set of bootstrap peers goes offline, then no new nodes will be able to join the network. Due to this fragility, this option is usually used as a fallback in case none of the other P2P bootstrapping mechanisms work properly.

Rather than hardcoding the set of bootstrap peers within the software/binary itself, we can instead allow peers to dynamically obtain a fresh/new set of bootstrap peers they can use to join the network. We'll call this process *initial peer discovery*. Typically we'll leverage existing internet protocols to maintain and distribute a set of bootstrapping peers. A nonexhaustive list of protocols that have been used in the past to accomplish initial peer discovery includes:

- Domain Name Service (DNS)
- Internet Relay Chat (IRC)
- Hypertext Transfer Protocol (HTTP)

Similar to the Bitcoin protocol, the primary initial peer discovery mechanism used in the Lightning Network happens via DNS. Because initial peer discovery is a critical and universal task for the network, the process has been *standardized* in BOLT #10: DNS Bootstrap.

DNS Bootstrapping

The BOLT #10 document describes a standardized way of implementing peer discovery using the DNS. Lightning's flavor of DNS-based bootstrapping uses up to three distinct record types:

- SRV records for discovering a set of *node public keys*.
- A records for mapping a node's public key to its current IPv4 address.
- AAA records for mapping a node's public key to its current IPv6 address.

Those somewhat familiar with the DNS protocol may already be familiar with the A (name to IPv4 address) and AAA (name to IPv6 address) record types, but not the SRV type. The SRV record type is used by protocols built on top of DNS to determine the *location* for a specified service. In our context, the service in question is a given Lightning node, and the location is its IP address. We need to use this additional record type because, unlike nodes within the Bitcoin protocol, we need both a public key *and* an IP address to connect to a node. As we see in Wire Protocol: Framing and Extensibility, the transport encryption protocol used in the Lightning Network requires knowledge of the public key of a node before connecting, so as to implement identity hiding for nodes in the network.

A new peer's bootstrapping workflow

Before diving into the specifics of BOLT #10, we'll first outline the high-level flow of a new node that wishes to use BOLT #10 to join the network.

First, a node needs to identify a single DNS server or set of DNS servers that understand BOLT #10 so they can be used for P2P bootstrapping.

While BOLT #10 uses *lseed.bitcoinstats.com* as the seed server, there exists no "official" set of DNS seeds for this purpose, but each of the major implementations maintains their own DNS seed, and they cross-query each other's seeds for redundancy purposes. In Table of known Lightning DNS seed servers you'll see a nonexhaustive list of some popular DNS seed servers.

Table 5. Table of known Lightning DNS seed servers

DNS server	Maintainer
lseed.bitcoinstats.com	Christian Decker
nodes.lightning.directory	Lightning Labs (Olaoluwa Osuntokun)
soa.nodes.lightning.directory	Lightning Labs (Olaoluwa Osuntokun)
lseed.darosior.ninja	Antoine Poinsot

DNS seeds exist for both Bitcoin's mainnet and testnet. For the sake of our example, we'll assume the existence of a valid BOLT #10 DNS seed at *nodes.lightning.directory*.

Next, our new node will issue an SRV query to obtain a set of *candidate bootstrap peers*. The response to our query will be a series of bech32 encoded public keys. Because DNS is a text-based protocol, we can't send raw binary data, so an encoding scheme is required. BOLT #10 specifies a bech32 encoding due to its use in the wider Bitcoin ecosystem. The number of encoded public keys returned depends on the server returning the query, as well as all the resolvers that stand between the client and the authoritative server.

Using the widely available dig command-line tool, we can query the *testnet* version of the DNS seed mentioned previously with the following command:

```
$ dig @8.8.8.8 test.nodes.lightning.directory SRV
```

We use the @ argument to force resolution via Google's nameserver (with IP address 8.8.8.8) because it does not filter large SRV query responses. At the end of the command, we specify that we only want SRV records to be returned. A sample response looks something like Querying the DNS seed for reachable nodes.

Example 8. Querying the DNS seed for reachable nodes

```
$ dig @8.8.8.8 test.nodes.lightning.directory SRV

; <<>> DiG 9.10.6 <<>> @8.8.8.8 test.nodes.lightning.directory SRV
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 43610
;; flags: qr rd ra; QUERY: 1, ANSWER: 25, AUTHORITY: 0, ADDITIONAL: 1

;; QUESTION SECTION:
;test.nodes.lightning.directory.    IN  SRV

;; ANSWER SECTION:
test.nodes.lightning.directory. 59 IN   SRV 10 10 9735 ①
ln1qfkxfad87fxx7lcwr4hvsalj8vhkwta539nuy4zlyf7hqcmrjh40xx5frs7.test.nodes.lightnin
g.directory. ②
test.nodes.lightning.directory. 59 IN   SRV 10 10 15735
ln1qtgsl3efj8verd4z27k44xu0a59kncvsarxatahm334exgnuvwhnz8dkhx8.test.nodes.lightnin
g.directory.

 [...]

;; Query time: 89 msec
;; SERVER: 8.8.8.8#53(8.8.8.8)
;; WHEN: Thu Dec 31 16:41:07 PST 2020
```

① TCP port number where the LN node can be reached.

② Node public key (ID) encoded as a virtual domain name.

We've truncated the response for brevity and show only two of the returned responses. The responses contain a "virtual" domain name for a target node, then to the left we have the *TCP port* where this node can be reached. The first response uses the standard TCP port for the Lightning Network: 9735. The second response uses a custom port, which is permitted by the protocol.

Next, we'll attempt to obtain the other piece of information we need to connect to a node: its IP address. Before we can query for this, however, we'll first *decode* the bech32 encoding of the public key from the virtual domain name:

```
ln1qfkxfad87fxx7lcwr4hvsalj8vhkwta539nuy4zlyf7hqcmrjh40xx5frs7
```

Decoding this bech32 string we obtain the following valid secp256k1 public key:

```
026c64f5a7f24c6f7f0e1d6ec877f23b2f672fb48967c2545f227d70636395eaf3
```

Now that we have the raw public key, we'll ask the DNS server to *resolve* the virtual host given so we can obtain the IP information (A record) for the node, as shown in [ex1102].

```
<div id="ex1102" data-type="example">
<h5>Obtaining the latest IP address for a node</h5>

<pre data-type="programlisting">$ dig
ln1qfkxfad87fxx7lcwr4hvsalj8vhkwta539nuy4zlyf7hqcmrjh40xx5frs7.test.nodes.lightning.di
rectory A

; &lt;&lt;&gt;&gt; DiG 9.10.6 &lt;&lt;&gt;&gt;
ln1qfkxfad87fxx7lcwr4hvsalj8vhkwta539nuy4zlyf7hqcmrjh40xx5frs7.test.nodes.lightning.di
rectory A
;; global options: +cmd
;; Got answer:
;; -&gt;&gt;HEADER&lt;&lt;- opcode: QUERY, status: NOERROR, id: 41934
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;ln1qfkxfad87fxx7lcwr4hvsalj8vhkwta539nuy4zlyf7hqcmrjh40xx5frs7.test.nodes.lightning.d
irectory. IN A

;; ANSWER SECTION:
ln1qfkxfad87fxx7lcwr4hvsalj8vhkwta539nuy4zlyf7hqcmrjh40xx5frs7.test.nodes.lightning.di
rectory. 60 IN A <em>X.X.X.X</em> <a class="co" id="comarker1" href="#c01"><img
src="callouts/1.png" alt="1"/></a>

;; Query time: 83 msec
;; SERVER: 2600:1700:6971:6dd0::1#53(2600:1700:6971:6dd0::1)
;; WHEN: Thu Dec 31 16:59:22 PST 2020
;; MSG SIZE  rcvd: 138</pre>

<dl class="calloutlist">
<dt><a class="co" id="c01" href="#comarker1"><img src="callouts/1.png"
alt="1"/></a></dt>
<dd><p>The DNS server returns an IP address <code><em>X.X.X.X</em></code>. We�ve
replaced it with X�s in the text here so as to avoid presenting a real IP
address.</p></dd>
</dl></div>
```

In the preceding command, we’ve queried the server so we can obtain an IPv4 (A record) address for our target node (replaced by __X.X.X.X__ in the preceding example). Now that we have the raw public key, IP address, and TCP port, we can connect to the node transport protocol at:

```
026c64f5a7f24c6f7f0e1d6ec877f23b2f672fb48967c2545f227d70636395eaf3@X.X.X.X:9735
```

Querying the current DNS A record for a given node can also be used to look up the *latest* set of addresses. Such queries can be used to more quickly sync the latest addressing information for a node, compared to waiting for address updates on the gossip network (see The node_announcement Message).

At this point in our journey, our new Lightning node has found its first peer and established its first connection! Now we can begin the second phase of new peer bootstrapping: channel graph synchronization and validation.

First, we'll explore more of the intricacies of BOLT #10 itself to take a deeper look into how things work under the hood.

SRV Query Options

The BOLT #10 standard is highly extensible due to its usage of nested subdomains as a communication layer for additional query options. The bootstrapping protocol allows clients to further specify the *type* of nodes they're attempting to query for versus the default of receiving a random subset of nodes in the query responses.

The query option subdomain scheme uses a series of key-value pairs where the key itself is a *single letter* and the remaining set of text is the value itself. The following query types exist in the current version of the BOLT #10 standards document:

r

The *realm* byte which is used to determine which chain or realm queries should be returned for. As is, the only value for this key is 0 which denotes "Bitcoin."

a

Allows clients to filter out returned nodes based on the *types* of addresses they advertise. As an example, this can be used to only obtain nodes that advertise a valid IPv6 address. The value that follows this type is based on a bitfield that *indexes* into the set of specified address *types* that are defined in BOLT #7. The default value for this field is 6, which represents both IPv4 and IPv6 (bits 1 and 2 are set).

l

A valid node public key serialized in compressed format. This allows a client to query for a specified node rather than receiving a set of random nodes.

n

The number of records to return. The default value for this field is 25.

An example query with additional query options looks something like the following:

```
r0.a2.n10.nodes.lightning.directory
```

Breaking down the query one key-value pair at a time, we gain the following insights:

- r0: The query targets the Bitcoin realm
- a2: The query only wants IPv4 addresses to be returned
- n10: The query requests

Try some combinations of the various flags using the dig DNS command-line tool yourself:

```
dig @8.8.8.8 r0.a6.nodes.lightning.directory SRV
```

The Channel Graph

Now that our new node is able to use the DNS bootstrapping protocol to connect to its very first peer, it can start to sync the channel graph! However, before we sync the channel graph, we'll need to learn exactly *what* we mean by the channel graph. In this section we'll explore the precise *structure* of the channel graph and examine the unique aspects of the channel graph compared to the typical abstract "graph" data structure which is well-known/used in the field of computer science.

A Directed Graph

A *graph* in computer science is a special data structure composed of vertices (typically referred to as nodes) and edges (also known as links). Two nodes may be connected by one or more edges. The channel graph is also *directed* given that a payment is able to flow in either direction over a given edge (a channel). An example of a *directed graph* is shown in A directed graph.

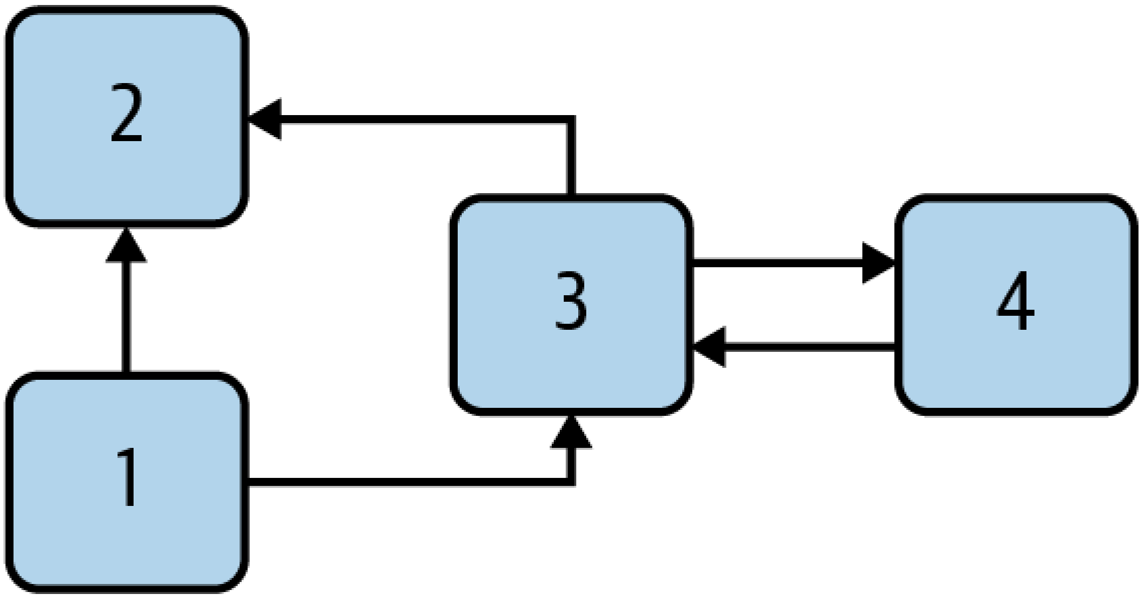


Figure 92. A directed graph

In the context of the Lightning Network, our vertices are the Lightning nodes themselves, with our edges being the payment channels connecting these nodes. Because we're concerned with *routing payments*, in our model a node with no edges (no payment channels) isn't considered to be a part of the graph since it isn't useful.

Because channels themselves are UTXOs (funded 2-of-2 multisig addresses), we can view the channel graph as a special subset of the Bitcoin UTXO set, on top of which we can add some additional information (the nodes, etc.) to arrive at the final overlay structure, which is the channel graph. This anchoring of fundamental components of the channel graph in the base Bitcoin blockchain means that it's impossible to *fake* a valid channel graph, which has useful properties when it comes to spam prevention as we'll see later.

Gossip Protocol Messages

The channel graph information is propagated across the Lightning P2P Network as three messages, which are described in BOLT #7:

node_announcement

The vertex in our graph which communicates the public key of a node, as well as how to reach the node over the internet and some additional metadata describing the set of *features* the node supports.

channel_announcement

A blockchain anchored proof of the existence of a channel between two individual nodes. Any third party can verify this proof to ensure that a *real* channel is actually being advertised. Similar to the node_announcement, this message also contains information describing the *capabilities* of the channel, which is useful when attempting to route a payment.

channel_update

A *pair* of structures that describes the set of routing policies for a given channel. channel_update messages come in a *pair* because a channel is a directed edge, so each side of the channel is able to specify its own custom routing policy.

It’s important to note that each component of the channel graph is *authenticated*, allowing a third party to ensure that the owner of a channel/update/node is actually the one sending out an update. This effectively makes the channel graph a unique type of *authenticated data structure* that cannot be counterfeited. For authentication, we use an secp256k1 ECDSA digital signature (or a series of them) over the serialized digest of the message itself. We won’t get into the specific of the messaging framing/serialization used in the Lightning Network in this chapter, as we’ll cover that information in Wire Protocol: Framing and Extensibility.

With the high-level structure of the channel graph laid out, we’ll now dive down into the precise structure of each of the three messages used to gossip the channel graph. We’ll also explain how one can also verify each message and component of the channel graph.

The node_announcement Message

First, we have the node_announcement message, which serves two primary purposes:

1. To advertise connection information so other nodes can connect to a node either to bootstrap to the network or to attempt to establish a new payment channel with that node.
2. To communicate the set of protocol-level features (capabilities) a node understands/supports. Feature negotiation between nodes allows developers to add new features independently and support them with any other node on an opt-in basis.

Unlike channel announcements, node announcements are not anchored in the base blockchain. Therefore, node announcements are only considered valid if they have propagated with a corresponding channel announcement. In other words, we always reject nodes without payment channels to ensure a malicious peer can’t flood the network with bogus nodes that are not part of the channel graph.

The node_announcement message structure

The node_announcement is comprised of the following fields:

signature
: A valid ECDSA signature that covers the serialized digest of all fields listed below. This signature must correspond to the public key of the advertised node.

features
: A bit vector that describes the set of protocol features that this node understands. We'll cover this field in more detail in Feature Bits and Protocol Extensibility on the extensibility of the Lightning protocol. At a high level, this field carries a set of bits that represent the features a node understands. As an example, a node may signal that it understands the latest channel type.

timestamp
: A Unix epoch encoded timestamp. This allows clients to enforce a partial ordering over the updates to a node's announcement.

node_id
: The secp256k1 public key that this node announcement belongs to. There can only be a single node_announcement for a given node in the channel graph at any given time. As a result, a node_announcement can supersede a prior node_announcement for the same node if it carries a higher (later) timestamp.

rgb_color
: A field that allows a node to specify an RGB color to be associated with it, often used in channel graph visualizations and node directories.

alias
: A UTF-8 string to serve as the nickname for a given node. Note that these aliases aren't required to be globally unique, nor are they verified in any way. As a result, they should not be relied on as a form of identity—they can be easily spoofed.

addresses
: A set of public internet reachable addresses that are to be associated with a given node. In the current version of the protocol, four address types are supported: IPv4 (type: 1), IPv6 (type: 2), Tor v2 (type: 3), and Tor v3 (type: 4). In the node_announcement message, each of these address types is denoted by an integer type which is included in parenthesis after the address type.

Validating node announcements

Validating an incoming node_announcement is straightforward. The following assertions should be upheld when examining a node announcement:

- If an existing node_announcement for that node is already known, then the timestamp field of a new incoming node_announcement must be greater than the prior one.
- With this constraint, we enforce a forced level of "freshness."
- If no node_announcement exists for the given node, then an existing channel_announcement

that references the given node (more on that later) must already exist in one's local channel graph.

- The included signature must be a valid ECDSA signature verified using the included node_id public key and the double–SHA-256 digest of the raw message encoding (minus the signature and frame header) as the message.
- All included addresses must be sorted in ascending order based on their address identifier.
- The included alias bytes must be a valid UTF-8 string.

The channel_announcement Message

Next, we have the channel_announcement message, which is used to *announce* a new *public* channel to the wider network. Note that announcing a channel is *optional*. A channel only needs to be announced if it is intended to be used for routing by the Lightning Network. Active routing nodes may wish to announce all their channels. However, certain nodes like mobile nodes likely don't have the uptime or desire to be an active routing node. As a result, these mobile nodes (which typically use light clients to connect to the Bitcoin P2P network) instead may have purely *unannounced* (private) channels.

Unannounced (private) channels

An unannounced channel isn't part of the known public channel graph, but can still be used to send/receive payments. An astute reader may now be wondering how a channel which isn't part of the public channel graph is able to receive payments. The solution to this problem is a set of "pathfinding helpers" that we call routing hints. As we'll see in Lightning Payment Requests, invoices created by nodes with unadvertised channels will include information to help the sender route to them, assuming the node has at least a single channel with an existing public routing node.

Due to the existence of unadvertised channels, the *true* size of the channel graph (both the public and private components) is unknown.

Locating a channel on the bitcoin blockchain

As mentioned earlier, the channel graph is authenticated due to its usage of public key cryptography, as well as the Bitcoin blockchain as a spam prevention system. To have a node accept a new channel_announcement, the advertisement must *prove* that the channel actually exists in the Bitcoin blockchain. This proof system adds an up-front cost to adding a new entry to the channel graph (the on-chain fees one must pay to create the UTXO of the channel). As a result, we mitigate spam and ensure that a dishonest node on the network can't fill up the memory of an honest node at no cost with bogus channels.

Given that we need to construct a proof of the existence of a channel, a natural question that arises is: how do we "point to" or reference a given channel for the verifier? Given that a payment channel is anchored in an unspent transaction output (see Inputs and Outputs), an initial thought might be to first attempt to advertise the full outpoint (txid:index) of the channel. Given the outpoint is globally unique and confirmed in the chain, this sounds like a good idea; however, it has a drawback: the verifier must maintain a full copy of the UTXO set to verify channels. This works fine for Bitcoin full nodes, but clients that rely on lightweight verification don't typically maintain a full UTXO set. Because we want to ensure we can support mobile nodes in the Lightning Network, we're forced to find another solution.

What if rather than referencing a channel by its UTXO, we reference it based on its "location" in the chain? To do this, we'll need a scheme that allows us to reference a given block, then a transaction within that block, and finally a specific output created by that transaction. Such an identifier is described in BOLT #7 and is referred to as a *short channel ID*, or scid. The scid is used in channel_announcement (and channel_update) as well as within the onion-encrypted routing packet included within HTLCs, as we learned in Onion Routing.

The short channel ID

Based on the preceding information, we have three pieces of information we need to encode to uniquely reference a given channel. Because we want a compact representation, we'll attempt to encode the information into a *single* integer. Our integer format of choice is an unsigned 64-bit integer, comprised of 8 bytes.

First, the block height. Using 3 bytes (24 bits) we can encode 16,777,216 blocks. That leaves 5 bytes for us to encode the transaction index and the output index, respectively. We'll use the next 3 bytes to encode the transaction index *within* a block. This is more than enough given that it's only possible to fix tens of thousands of transactions in a block at current block sizes. This leaves 2 bytes left for us to encode the output index of the channel within the transaction.

Our final scid format resembles:

```
block_height (3 bytes) || transaction_index (3 bytes) || output_index (2 bytes)
```

Using bit packing techniques, we first encode the most significant 3 bytes as the block height, the next 3 bytes as the transaction index, and the least significant 2 bytes as the output index of that creates the channel UTXO.

A short channel ID can be represented as a single integer (695313561322258433) or as a more human friendly string: 632384x1568x1. Here we see the channel was mined in block 632384, was the 1568th transaction in the block, with the channel output as the second (UTXOs are zero-indexed) output produced by the transaction.

Now that we're able to succinctly point to a given channel funding output in the chain, we can examine the full structure of the channel_announcement message, as well as see how to verify the proof-of-existence included within the message.

The channel_announcement message structure

A channel_announcement primarily communicates two things:

1. A proof that a channel exists between node A and node B with both nodes controlling the mulitsig keys in that channel output.
2. The set of capabilities of the channel (what types of HTLCs can it route, etc.).

When describing the proof, we'll typically refer to node 1 and node 2. Out of the two nodes that a channel connects, the "first" node is the node that has a "lower" public key encoding when we compare the public key of the two nodes in compressed format hex-encoded in lexicographical order. Correspondingly, in addition to a node public key on the network, each node should also control a public key within the Bitcoin blockchain.

Similar to the node_announcement message, all included signatures of the channel_announcement message should be signed/verified against the raw encoding of the message (minus the header) that follows *after* the final signature (because it isn't possible for a digital signature to sign itself).

With that said, a channel_announcement message has the following fields:

node_signature_1
: The signature of the first node over the message digest.

node_signature_2
: The signature of the second node over the message digest.

bitcoin_signature_1
: The signature of the multisig key (in the funding output) of the first node over the message digest.

bitcoin_signature_2
: The signature of the multisig key (in the funding output) of the second node over the message digest.

features
: A feature bit vector that describes the set of protocol level features supported by this channel.

chain_hash
: A 32-byte hash which is typically the genesis block hash of the blockchain (e.g., Bitcoin mainnet) the channel was opened in.

short_channel_id
: The scid that uniquely locates the given channel funding output within the blockchain.

node_id_1
: The public key of the first node in the network.

node_id_2
: The public key of the second node in the network.

bitcoin_key_1
: The raw multisig key for the channel funding output for the first node in the network.

bitcoin_key_2
: The raw multisig key for the channel funding output for the second node in the network.

Channel announcement validation

Now that we know what a channel_announcement contains, we can look at how to verify the channel's existence on-chain.

Armed with the information in the channel_announcement, any Lightning node (even one without a full copy of the Bitcoin blockchain) can verify the existence and authenticity of the payment channel.

First, the verifier will use the short channel ID to find which Bitcoin block contains the channel funding output. With the block height information, the verifier can request only that specific block from a Bitcoin node. The block can then be linked back to the genesis block by following the block header chain backward (verifying the proof-of-work), confirming that this is in fact a block belonging to the Bitcoin blockchain.

Next, the verifier uses the transaction index number to identify the transaction ID of the transaction containing the payment channel. Most modern Bitcoin libraries will allow indexing into the transaction of a block based on the index of the transaction within the greater block.

Next, the verifier uses a Bitcoin library (in the verifier's language) to extract the relevant transaction according to its index within the block. The verifier will validate the transaction (checking that it is properly signed and produces the same transaction ID when hashed).

Next, the verifier will extract the Pay-to-Witness-Script-Hash (P2WSH) output referenced by the output index number of the short channel ID. This is the address of the channel funding output. Additionally, the verifier will ensure that the size of the alleged channel matches the value of the output produced at the specified output index.

Finally, the verifier will reconstruct the multisig script from bitcoin_key_1 and bitcoin_key_2 and confirm that it produces the same address as in the output.

The verifier has now independently verified that the payment channel in the announcement is funded and confirmed on the Bitcoin blockchain!

The channel_update Message

The third and final message used in the gossip protocol is the channel_update message. Two of these are generated for each payment channel (one by each channel partner) announcing their routing fees, timelock expectations, and capabilities.

The channel_update message also contains a timestamp, allowing a node to update its routing fees and other expectations and capabilities by sending a new channel_update message with a higher (later) timestamp that supersedes any older updates.

The channel_update message contains the following fields:

signature
: A digital signature matching the node's public key, to authenticate the source and integrity of the channel update

chain_hash
: The hash of the genesis block of the chain containing the channel

short_channel_id

The short channel ID to identify the channel

timestamp

The timestamp of this update, to allow recipients to sequence updates and replace older updates

message_flags

A bit field indicating the presence of additional fields in the channel_update message

channel_flags

A bit field showing the direction of the channel and other channel options

cltv_expiry_delta

The timelock delta expectations of this node for routing (see Onion Routing)

htlc_minimum_msat

The minimum HTLC amount that will be routed

fee_base_msat

The base fee that will be charged for routing

fee_proportional_millionths

The proportional fee rate that will be charged for routing

htlc_maximum_msat (option_channel_htlc_max)

The maximum amount that will be routed

A node that receives the channel_update message can attach this metadata to the channel graph edge to enable pathfinding, as we will see in Pathfinding and Payment Delivery.

Ongoing Channel Graph Maintenance

The construction of a channel graph is not a one-time event, but rather an ongoing activity. As a node bootstraps into the network it will start receiving "gossip," in the form of the three update messages. It will use these messages to immediately start building a validated channel graph.

The more information a node receives, the better its "map" of the Lightning Network becomes and the more effective it can be at pathfinding and payment delivery.

A node won’t only add information to the channel graph. It will also keep track of the last time a channel was updated and will delete "stale" channels that have not been updated in more than two weeks. Finally, if it sees that some node no longer has any channels, it will also remove that node.

The information collected from the gossip protocol is not the only information that can be stored in the channel graph. Different Lightning node implementations may attach other metadata to nodes and channels. For example, some node implementations calculate a "score" that evaluates a node’s "quality" as a routing peer. This score is used as part of pathfinding to prioritize or deprioritize paths.

Conclusion

In this chapter, we've learned how Lightning nodes discover each other, discover and update their node status, and communicate with one another. We've learned how channel graphs are created and maintained, and we've explored a few ways that the Lightning Network discourages bad actors or dishonest nodes from spamming the network.

Pathfinding and Payment Delivery

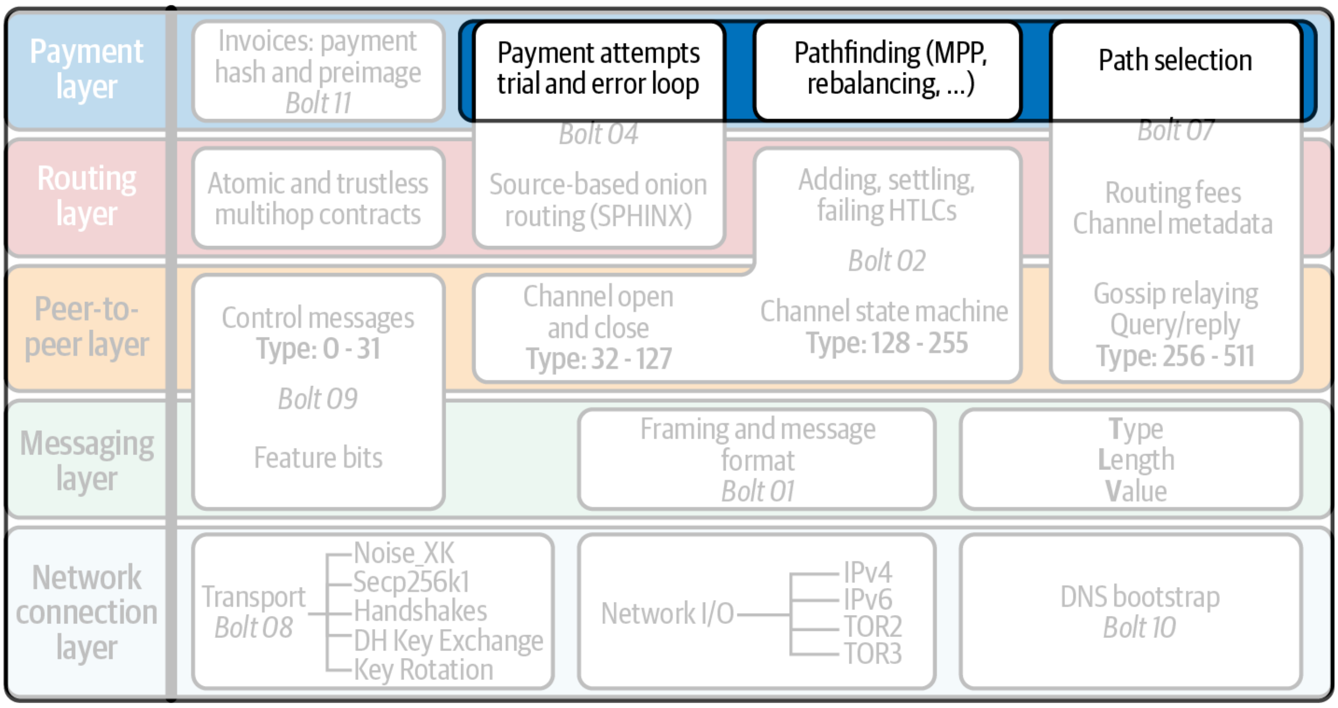
Payment delivery on the Lightning Network depends on finding a path from the sender to the recipient, a process called *pathfinding*. Since the routing is done by the sender, the sender must find a suitable path to reach the destination. This path is then encoded in an onion, as we saw in Onion Routing.

In this chapter we will examine the problem of pathfinding, understand how uncertainty about channel balances complicates this problem, and look at how a typical pathfinding implementation attempts to solve it.

Pathfinding in the Lightning Protocol Suite

Pathfinding, path selection, multipart payments (MPP), and the payment attempt trial-and-error loop occupy the majority of the payment layer at the top of the protocol suite.

These components are highlighted by an outline in the protocol suite, shown in Payment delivery in the Lightning protocol suite.



Payment layer
Invoices: payment hash and preimage Bolt 11
Payment attempts trial and error loop
Pathfinding (MPP, rebalancing, ...)
Path selection
Bolt 04
Bolt 07
Routing layer
Atomic and trustless multihop contracts
Source-based onion routing (SPHINX)
Adding, settling, failing HTLCs
Bolt 02
Routing fees Channel metadata
Peer-to-peer layer
Control messages Type: 0 - 31
Bolt 09
Feature bits
Channel open and close Type: 32 - 127
Channel state machine Type: 128 - 255
Gossip relaying Query/reply Type: 256 - 511
Messaging layer
Framing and 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 93. Payment delivery in the Lightning protocol suite

Where Is the BOLT?

So far we've looked at several technologies that are part of the Lightning Network and we have seen their exact specification as part of a BOLT standard. You may be surprised to find that pathfinding is not part of the BOLTs!

That's because pathfinding isn't an activity that requires any form of coordination or interoperability between different implementations. As we've seen, the path is selected by the sender. Even though the routing details are specified in detail in the BOLTs, the path discovery and selection are left entirely up to the sender. So each node implementation can choose a different strategy/algorithm to find paths. In fact, the different node/client and wallet implementations can even compete and use their pathfinding algorithm as a point of differentiation.

Pathfinding: What Problem Are We Solving?

The term pathfinding may be somewhat misleading because it implies a search for *a single path* connecting two nodes. In the beginning, when the Lightning Network was small and not well interconnected, the problem was indeed about finding a way to join payment channels to reach the recipient.

But, as the Lightning Network has grown explosively, the pathfinding problem's nature has shifted. In mid-2021, as we finish this book, the Lightning Network consists of 20,000 nodes connected by at least 55,000 public channels with an aggregate capacity of almost 2,000 BTC. A node has on average 8.8 channels, while the top 10 most connected nodes have between 400 and 2,000 channels *each*. A visualization of just a small subset of the LN channel graph is shown in A visualization of part of the Lightning Network as of July 2021.

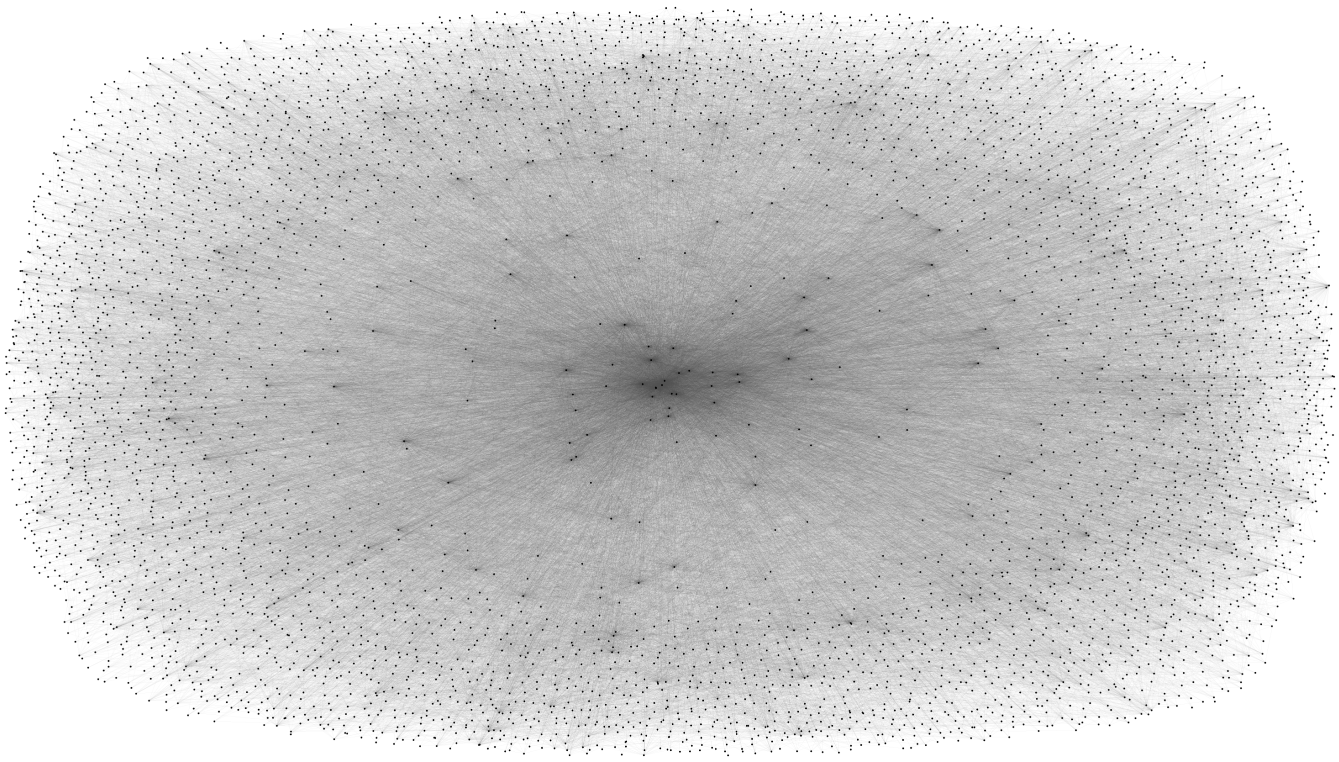


Figure 94. A visualization of part of the Lightning Network as of July 2021

NOTE The network visualization in A visualization of part of the Lightning Network as of July 2021 was produced with a simple Python script you can find in code/lngraph in the book's repository.

If the sender and recipient are connected to other well-connected nodes and have at least one channel with adequate capacity, there will be thousands of paths. The problem becomes selecting the *best* path that will succeed in payment delivery, out of a list of thousands of possible paths.

Selecting the Best Path

To select the best path, we have to first define what we mean by "best." There may be many different criteria, such as:

- Paths with enough liquidity. Obviously if a path doesn't have enough liquidity to route our payment, then it is not a suitable path.
- Paths with low fees. If we have several candidates, we may want to select ones with lower fees.
- Paths with short timelocks. We may want to avoid locking our funds for too long and therefore select paths with shorter timelocks.

All of these criteria may be desirable to some extent, and selecting paths that are favorable across many dimensions is not an easy task. Optimization problems like this may be too complex to solve for the "best" solution, but often can be solved for some approximation of the optimal, which is good news because otherwise pathfinding would be an intractable problem.

Pathfinding in Math and Computer Science

Pathfinding in the Lightning Network falls under a general category of *graph theory* in mathematics and the more specific category of *graph traversal* in computer science.

A network such as the Lightning Network can be represented as a mathematical construct called a *graph*, where *nodes* are connected to each other by *edges* (equivalent to the payment channels). The Lightning Network forms a *directed graph* because the nodes are linked *asymmetrically*, since the channel balance is split between the two channel partners and the payment liquidity is different in each direction. A directed graph with numerical capacity constraints on its edges is called a *flow network*, a mathematical construct used to optimize transportation and other similar networks. Flow networks can be used as a framework when solutions need to achieve a specific flow while minimizing cost, known as the minimum cost flow problem (MCFP).

Capacity, Balance, Liquidity

To better understand the problem of transporting satoshis from point A to point B, we need to better define three important terms: capacity, balance, and liquidity. We use these terms to describe a payment channel's ability to route a payment.

In a payment channel connecting A←→B:

Capacity

This is the aggregate amount of satoshis that were funded into the 2-of-2 multisig with the funding transaction. It represents the maximum amount of value held in the channel. The channel capacity is announced by the gossip protocol and is known to nodes.

Balance

This is the amount of satoshis held by each channel partner that can be sent to the other channel partner. A subset of the balance of A can be sent in the direction (A→B) toward node B. A subset of the balance of B can be sent in the opposite direction (A←B).

Liquidity

The available (subset) balance that can actually be sent across the channel in one direction. Liquidity of A is equal to the balance of A minus the channel reserve and any pending HTLCs committed by A.

The only value known to the network (via gossip announcements) is the aggregate capacity of the channel. Some unknown portion of that capacity is distributed as each partner’s balance. Some subset of that balance is available to send across the channel in one direction:

```
<ul class="simplelist">
<li>capacity = balance(A) + balance(B)</li>
<li>liquidity(A) = balance(A) − channel_reserve(A) − pending_HTLCs(A)</li>
</ul>
```

Uncertainty of Balances

If we knew the exact channel balances of every channel, we could compute one or more payment paths using any of the standard pathfinding algorithms taught in good computer science programs. But we don’t know the channel balances; we only know the aggregate channel capacity, which is advertised by nodes in channel announcements. In order for a payment to succeed, there must be adequate balance on the sending side of the channel. If we don’t know how the capacity is distributed between the channel partners, we don’t know if there is enough balance in the direction we are trying to send the payment.

Balances are not announced in channel updates for two reasons: privacy and scalability. First, announcing balances would reduce the privacy of the Lightning Network because it would allow surveillance of payment by statistical analysis of the changes in balances. Second, if nodes announced balances (globally) with every payment, the Lightning Network’s scaling would be as bad as that of on-chain Bitcoin transactions which are broadcast to all participants. Therefore, balances are not announced. To solve the pathfinding problem in the face of uncertainty of balances, we need innovative pathfinding strategies. These strategies must relate closely to the routing algorithm that is used, which is source-based onion routing where it is the responsibility of the sender to find a path through the network.

The uncertainty problem can be described mathematically as a *range of liquidity*, indicating the lower and upper bounds of liquidity based on the information that is known. Since we know the capacity of the channel and we know the channel reserve balance (the minimum allowed balance on each end), the liquidity can be defined as:

```
<ul class="simplelist">
<li>min(liquidity) = channel_reserve</li>
<li>max(liquidity) = capacity − channel_reserve</li>
</ul>
```

or as a range:

```
<ul class="simplelist">
<li>channel_reserve &lt;= liquidity &lt;= (capacity − channel_reserve)</li>
</ul>
```

Our channel liquidity uncertainty range is the range between the minimum and maximum possible liquidity. This is unknown to the network, except the two channel partners. However, as we will see, we can use failed HTLCs returned from our payment attempts to update our liquidity estimate and reduce uncertainty. If, for example, we get an HTLC failure code that tells us that a channel cannot fulfill an HTLC that is smaller than our estimate for maximum liquidity, that means the maximum liquidity can be updated to the amount of the failed HTLC. In simpler terms, if we think the liquidity can handle an HTLC of *N* satoshis and we find out it fails to deliver *M* satoshis (where *M* is smaller), then we can update our estimate to *M*–1 as the upper bound. We tried to find the ceiling and bumped against it, so it's lower than we thought!

Pathfinding Complexity

Finding a path through a graph is a problem modern computers can solve rather efficiently. Developers mainly choose breadth-first search if the edges are all of equal weight. In cases where the edges are not of equal weight, an algorithm based on Dijkstra's algorithm is used, such as A* (pronounced "A-star"). In our case the weights of the edges can represent the routing fees. Only edges with a capacity larger than the amount to be sent will be included in the search. In this basic form, pathfinding in the Lightning Network is very simple and straightforward.

However, channel liquidity is unknown to the sender. This turns our easy theoretical computer science problem into a rather complex real-world problem. We now have to solve a pathfinding problem with only partial knowledge. For example, we suspect which edges might be able to forward a payment because their capacity seems big enough. But we can't be certain unless we try it out or ask the channel owners directly. Even if we were able to ask the channel owners directly, their balance might change by the time we have asked others, computed a path, constructed an onion, and sent it along. Not only do we have limited information but the information we have is highly dynamic and might change at any point in time without our knowledge.

Keeping It Simple

The pathfinding mechanism implemented in Lightning nodes is to first create a list of candidate paths, filtered and sorted by some function. Then, the node or wallet will probe paths (by attempting to deliver a payment) in a trial-and-error loop until a path is found that successfully delivers the payment.

NOTE This probing is done by the Lightning node or wallet and is not directly observed by the user of the software. However, the user might suspect that probing is taking place if the payment is not completed instantly.

While blind probing is not optimal and leaves ample room for improvement, it should be noted that even this simplistic strategy works surprisingly well for smaller payments and well-connected nodes.

Most Lightning node and wallet implementations improve on this approach by ordering/weighting the list of candidate paths. Some implementations order the candidate paths by cost (fees) or some combination of cost and capacity.

Pathfinding and Payment Delivery Process

Pathfinding and payment delivery involves several steps, which we list here. Different implementations may use different algorithms and strategies, but the basic steps are likely to be very similar:

1. Create a *channel graph* from announcements and updates containing the capacity of each channel, and filter the graph, ignoring any channels with insufficient capacity for the amount we want to send.
2. Find paths connecting the sender to the recipient.
3. Order the paths by some weight (this may be part of the previous step's algorithm).
4. Try each path in order until payment succeeds (the trial-and-error loop).
5. Optionally use the HTLC failure returns to update our graph, reducing uncertainty.

We can group these steps into three primary activities:

- Channel graph construction
- Pathfinding (filtered and ordered by some heuristics)
- Payment attempt(s)

These three activities can be repeated in a *payment round* if we use the failure returns to update the graph, or if we are doing multipart payments (see Multipart Payments (MPP)).

In the next sections we will look at each of these steps in more detail, as well as more advanced payment strategies.

Channel Graph Construction

In Gossip and the Channel Graph we covered the three main messages that nodes use in their gossip: node_announcement, channel_announcement, and channel_update. These three messages allow any node to gradually construct a "map" of the Lightning Network in the form of a *channel graph*. Each of these messages provides a critical piece of information for the channel graph:

node_announcement
: This contains the information about a node on the Lightning Network, such as its node ID (public key), network address (e.g., IPv4/6 or Tor), capabilities/features, etc.

channel_announcement
: This contains the capacity and channel ID of a public (announced) channel between two nodes and proof of the channel's existence and ownership.

channel_update
: This contains a node's fee and timelock (CLTV) expectations for routing an outgoing (from that node's perspective) payment over a specified channel.

In terms of a mathematical graph, the node_announcement is the information needed to create the nodes or *vertices* of the graph. The channel_announcement allows us to create the *edges* of the graph representing the payment channels. Since each direction of the payment channel has its own balance, we create a directed graph. The channel_update allows us to incorporate fees and timelocks to set the *cost* or *weight* of the graph edges.

Depending on the algorithm we will use for pathfinding, we may establish a number of different cost functions for the edges of the graph.

For now, let's ignore the cost function and simply establish a channel graph showing nodes and channels, using the node_announcement and channel_announcement messages.

In this chapter we will see how Selena attempts to find a path to pay Rashid 1,000,000 (1M) satoshis. To start, Selena is constructing a channel graph using the information from Lightning Network gossip to discover nodes and channels. Selena will then explore her channel graph to find a path to send a payment to Rashid.

This is *Selena's* channel graph. There is no such thing as *the* channel graph, there is only ever *a channel graph*, and it is always from the perspective of the node that has constructed it (see The Map-Territory Relation).

TIP Selena does not contruct a channel graph only when sending a payment. Rather, Selena's node is *continuously* building and updating a channel graph. From the moment Selena's node starts and connects to any peer on the network it will participate in the gossip and use every message to learn as much as possible about the network.

The Map-Territory Relation

From Wikipedia's page on the Map-Territory Relation, "The map-territory relation describes the relationship between an object and a representation of that object, as in the relation between a geographical territory and a map of it."

The map-territory relation is best illustrated in "Sylvie and Bruno Concluded," a short story by Lewis Carroll which describes a fictional map that is a 1:1 scale of the territory it maps, therefore having perfect accuracy but becoming completely useless as it would cover the entire territory if unfolded.

What does this mean for the Lightning Network? The Lightning Network is the territory, and a channel graph is a map of that territory.

While we could imagine a theoretical (Platonic ideal) channel graph that represents the complete, up-to-date map of the Lightning Network, such a map is simply the Lightning Network itself. Each node has its own channel graph which is constructed from announcements and is necessarily incomplete, incorrect, and out-of-date!

The map can never completely and accurately describe the territory.

Selena listens to node_announcement messages and discovers four other nodes (in addition to

Rashid, the intended recipient). The resulting graph represents a network of six nodes: Selena and Rashid are the sender and recipient, respectively; Alice, Bob, Xavier, and Yan are intermediary nodes. Selena's initial graph is just a list of nodes, shown in Node announcements.

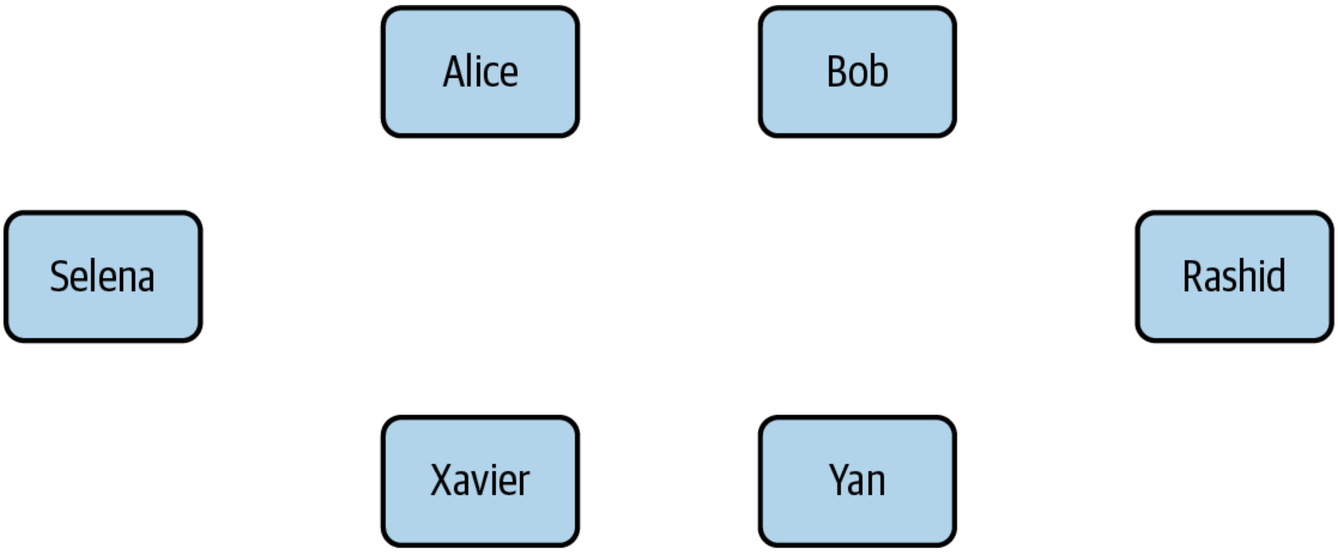


Figure 95. Node announcements

Selena also receives seven channel_announcement messages with the corresponding channel capacities, allowing her to construct a basic "map" of the network, shown in The channel graph. (The names Alice, Bob, Selena, Xavier, Yan, and Rashid have been replaced by their initials: A, B, S, X, and R, respectively.)

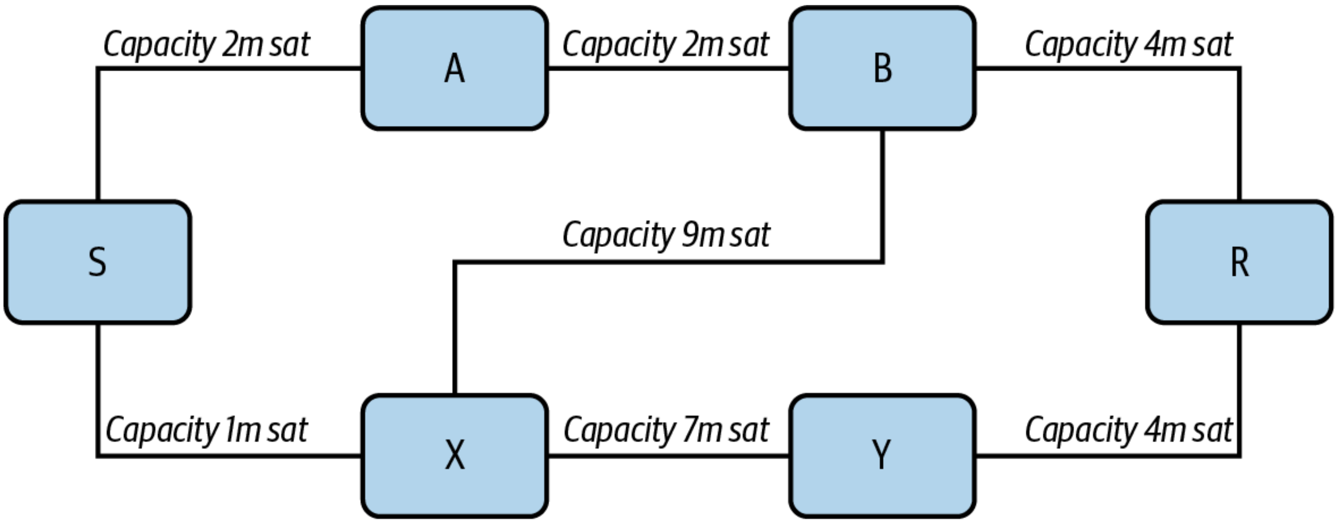


Figure 96. The channel graph

Uncertainty in the channel graph

As you can see from The channel graph, Selena does not know any of the balances of the channels. Her initial channel graph contains the highest level of uncertainty.

But wait: Selena does know *some* channel balances! She knows the balances of the channels that her own node has connected with other nodes. While this does not seem like much, it is in fact very important information for constructing a path—Selena knows the actual liquidity of her own channels. Let's update the channel graph to show this information. We will use a "?" symbol to represent the unknown balances, as shown in Channel graph with known and unknown balances.

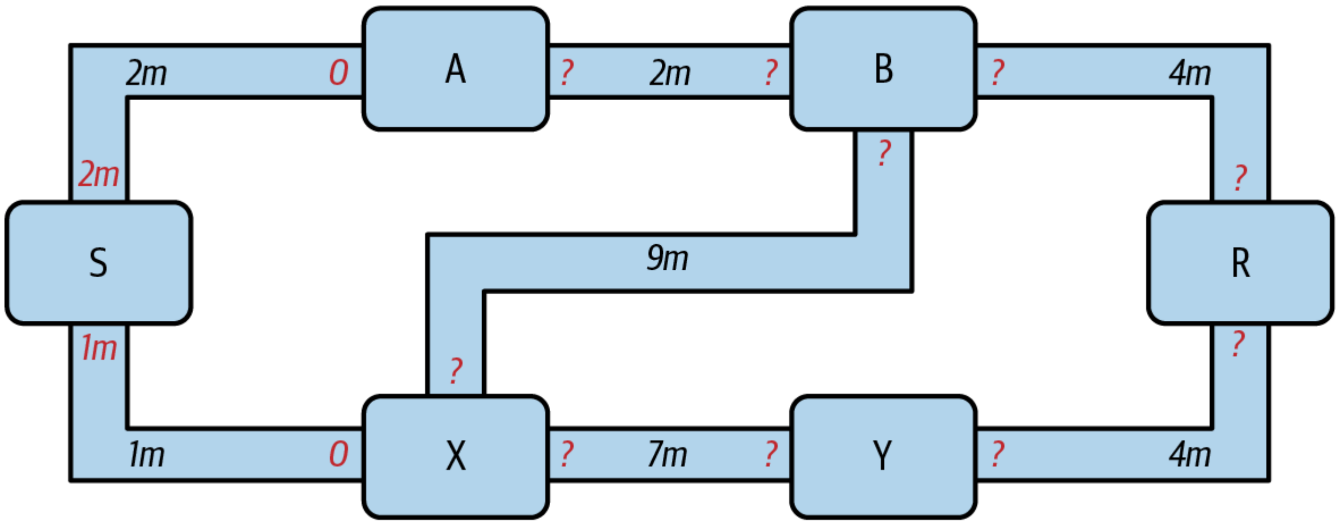


Figure 97. Channel graph with known and unknown balances

While the "?" symbol seems ominous, a lack of certainty is not the same as complete ignorance. We can *quantify* the uncertainty and *reduce* it by updating the graph with the successful/failed HTLCs we attempt.

Uncertainty can be quantified, because we know the maximum and minimum possible liquidity and can calculate probabilities for smaller (more precise) ranges.

Once we attempt to send an HTLC, we can learn more about channel balances: if we succeed, then the balance was *at least* sufficient to transport the specific amount. Meanwhile if we get a "temporary channel failure" error, the most likely reason is a lack of liquidity for the specific amount.

TIP You may be thinking, "What’s the point of learning from a successful HTLC?" After all, if it succeeded we’re "done." But consider that we may be sending one part of a multipart payment. We also may be sending other single-part payments within a short time. Anything we learn about liquidity is useful for the next attempt!

Liquidity Uncertainty and Probability

To quantify the uncertainty of a channel’s liquidity, we can apply probability theory. A basic model of the probability of payment delivery will lead to some rather obvious, but important, conclusions:

- Smaller payments have a better chance of successful delivery across a path.
- Larger capacity channels will give us a better chance of payment delivery for a specific amount.
- The more channels (hops), the lower the chance of success.

While these may be obvious, they have important implications, especially for the use of multipart payments (see Multipart Payments (MPP)). The math is not difficult to follow.

Let’s use probability theory to see how we arrived at these conclusions.

First, let’s posit that a channel with capacity *c* has liquidity on one side with an unknown value in the range of (0, *c*) or "range between 0 and *c*." For example, if the capacity is 5, then the liquidity will be in the range (0, 5). Now, from this we see that if we want to send 5 satoshis, our chance of success is only 1 in 6 (16.66%), because we will only succeed if the liquidity is exactly 5.

More simply, if the possible values for the liquidity are 0, 1, 2, 3, 4, and 5, only one of those six possible values will be sufficient to send our payment. To continue this example, if our payment amount was 3, then we would succeed if the liquidity was 3, 4, or 5. So our chances of success are 3 in 6 (50%). Expressed in math, the success probability function for a single channel is:

```
$P_c(a) = (c + 1 - a) / (c + 1)$
```

where a is the amount and c is the capacity.

From the equation we see that if the amount is close to 0, the probability is close to 1, whereas if the amount exceeds the capacity, the probability is zero.

In other words: "Smaller payments have a better chance of successful delivery" or "Larger capacity channels give us better chances of delivery for a specific amount" and "You can’t send a payment on a channel with insufficient capacity."

Now let’s think about the probability of success across a path made of several channels. Let’s say our first channel has a 50% chance of success (P = 0.5). Then if our second channel has a 50% chance of success (P = 0.5), it is intuitive that our overall chance is 25% (P = 0.25).

We can express this as an equation that calculates the probability of a payment’s success as the product of probabilities for each channel in the path(s):

```
$P_{payment} = \prod_{i=1}^n P_i$
```

Where P_i is the probability of success over one path or channel, and $P_{payment}$ is the overall probability of a successful payment over all the paths/channels.

From the equation we see that since the probability of success over a single channel is always less than or equal to 1, the probability across many channels will *drop exponentially*.

In other words, "The more channels (hops) you use, the lower the chance of success."

NOTE There is a lot of mathematical theory and modeling behind the uncertainty of the liquidity in the channels. Fundamental work about modeling the uncertainty intervals of the channel liquidity can be found in the paper "Security and Privacy of Lightning Network Payments with Uncertain Channel Balances" by (coauthor of this book) Pickhardt et al.

Fees and Other Channel Metrics

Next, our sender will add information to the graph from channel_update messages received from the intermediary nodes. As a reminder, the channel_update contains a wealth of information about a channel and the expectations of one of the channel partners.

In Channel graph fees and other channel metrics we see how Selena can update the channel graph based on channel_update messages from A, B, X, and Y. Note that the channel ID and channel direction (included in channel_flags) tell Selena which channel and which direction this update refers to. Each channel partner gossips one or more channel_update messages to announce their fee expectations and other information about the channel. For example, in the top left we see the channel_update sent by Alice for the channel A—B and the direction A-to-B. With this update, Alice tells the network how much she will charge in fees to route an HTLC to Bob over that specific channel. Bob may announce a channel update (not shown in this diagram) for the opposite direction with completely different fee expectations. Any node may send a new channel_update to change the fees or timelock expectations at any time.

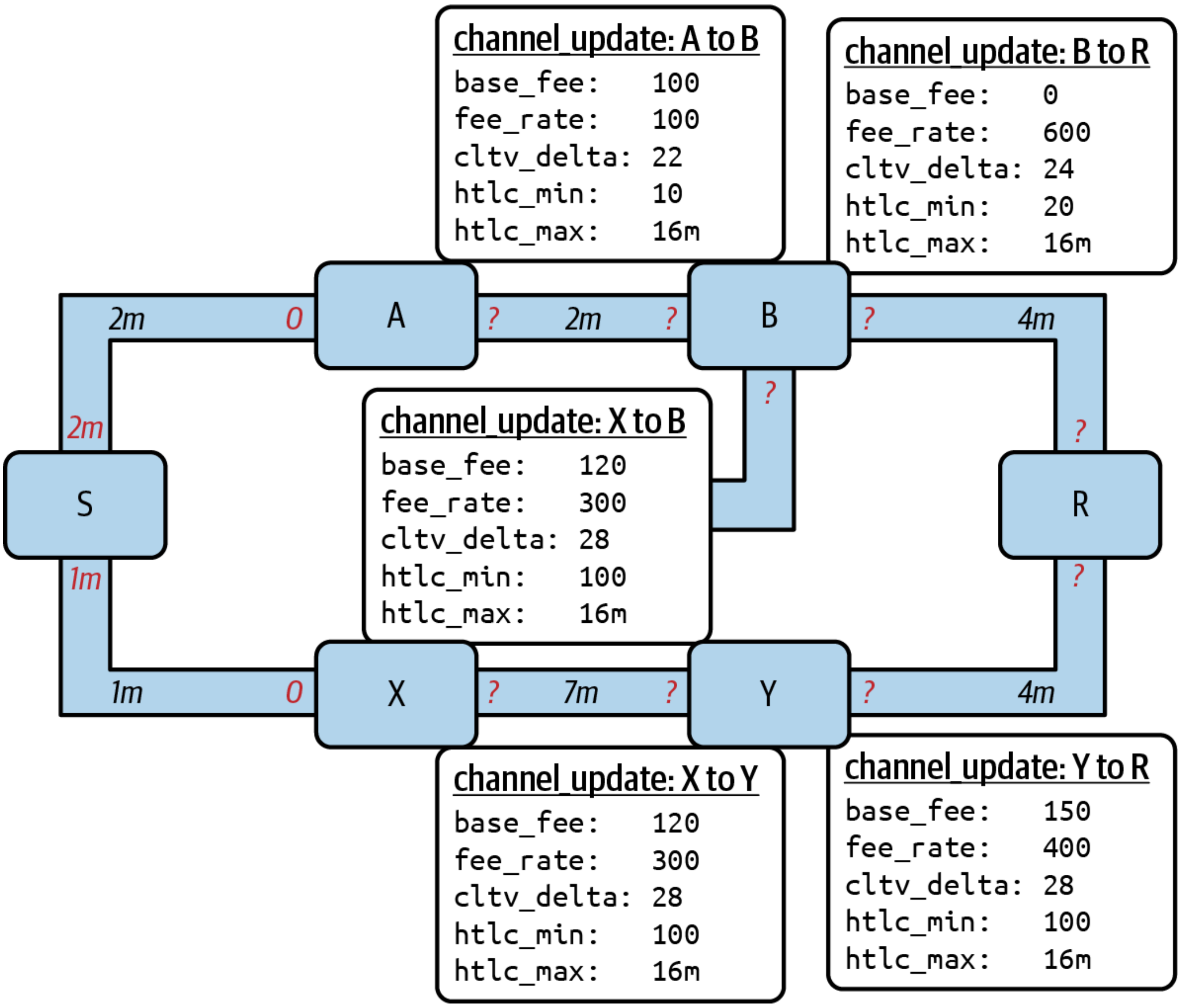


Figure 98. Channel graph fees and other channel metrics

The fee and timelock information are very important, not just as path selection metrics. As we saw in Onion Routing, the sender needs to add up fees and timelocks (cltv_expiry_delta) at each hop to make the onion. The process of calculating fees happens from the recipient to the sender *backward* along the path because each intermediary hop expects an incoming HTLC with higher amount and expiry timelock than the outgoing HTLC they will send to the next hop. So, for example, if Bob wants 1,000 satoshis in fees and 30 blocks of expiry timelock delta to send a payment to Rashid, then that amount and expiry delta must be added to the HTLC *from Alice*.

It is also important to note that a channel must have liquidity that is sufficient not only for the payment amount but also for the cumulative fees of all the subsequent hops. Even though Selena's channel to Xavier (S→X) has enough liquidity for a 1M satoshi payment, it *does not* have enough liquidity once we consider fees. We need to know fees because only paths that have sufficient liquidity for *both payment and all fees* will be considered.

Finding Candidate Paths

Finding a suitable path through a directed graph like this is a well-studied computer science problem (known broadly as the *shortest path problem*), which can be solved by a variety of algorithms depending on the desired optimization and resource constraints.

The most famous algorithm solving this problem was invented by Dutch mathematician E. W. Dijkstra in 1956, known simply as *Dijkstra's algorithm*. In addition to the original Dijkstra's algorithm, there are many variations and optimizations, such as A* ("A-star"), which is a heuristic-based algorithm.

As mentioned previously, the "search" must be applied *backward* to account for fees that are accumulated from recipient to sender. Thus, Dijkstra, A*, or some other algorithm would search for a path from the recipient to the sender, using fees, estimated liquidity, and timelock delta (or some combination) as a cost function for each hop.

Using one such algorithm, Selena calculates several possible paths to Rashid, sorted by shortest path:

1. S→A→B→R
2. S→X→Y→R
3. S→X→B→R
4. S→A→B→X→Y→R

But, as we saw previously, the channel S→X does not have enough liquidity for a 1M satoshi payment once fees are considered. So Paths 2 and 3 are not viable. That leaves Paths 1 and 4 as possible paths for the payment.

With two possible paths, Selena is ready to attempt delivery!

Payment Delivery (Trial-and-Error Loop)

Selena's node starts the trial-and-error loop by constructing the HTLCs, building the onion, and attempting delivery of the payment. For each attempt, there are three possible outcomes:

- A successful result (update_fulfill_htlc)
- An error (update_fail_htlc)
- A "stuck" payment with no response (neither success nor failure)

If the payment fails, it can be retried via a different path by updating the graph (changing a channel's metrics) and recalculating an alternative path.

We looked at what happens if the payment is "stuck" in Stuck payments. The important detail is that a stuck payment is the worst outcome because we cannot retry with another HTLC since both (the stuck one and the retry one) might go through eventually and cause a double payment.

First Attempt (Path #1)

Selena attempts the first path (S→A→B→R). She constructs the onion and sends it, but receives a failure code from Bob's node. Bob reports back a temporary channel failure with a channel_update identifying the channel B→R as the one that can't deliver. This attempt is shown in Path #1 attempt fails.

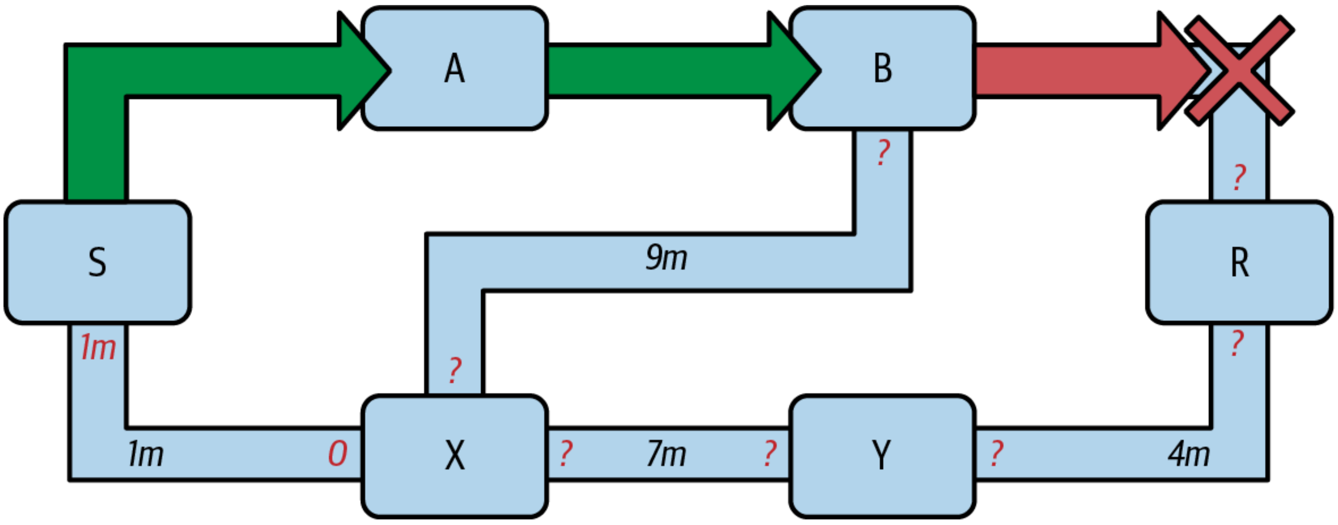


Figure 99. Path #1 attempt fails

Learning from failure

From this failure code, Selena will deduce that Bob doesn't have enough liquidity to deliver the payment to Rashid on that channel. Importantly, this failure narrows the uncertainty of the liquidity of that channel! Previously, Selena's node assumed that the liquidity on Bob's side of the channel was somewhere in the range (0, 4M). Now, she can assume that the liquidity is in the range (0, 999999). Similarly, Selena can now assume that the liquidity of that channel on Rashid's side is in the range (1M, 4M), instead of (0, 4M). Selena has learned a lot from this failure.

Second Attempt (Path #4)

Now Selena attempts the fourth candidate path (S→A→B→X→Y→R). This is a longer path and will incur more fees, but it's now the best option for delivery of the payment.

Fortunately, Selena receives an update_fulfill_htlc message from Alice, indicating that the payment was successful, as shown in Path #4 attempt succeeds.

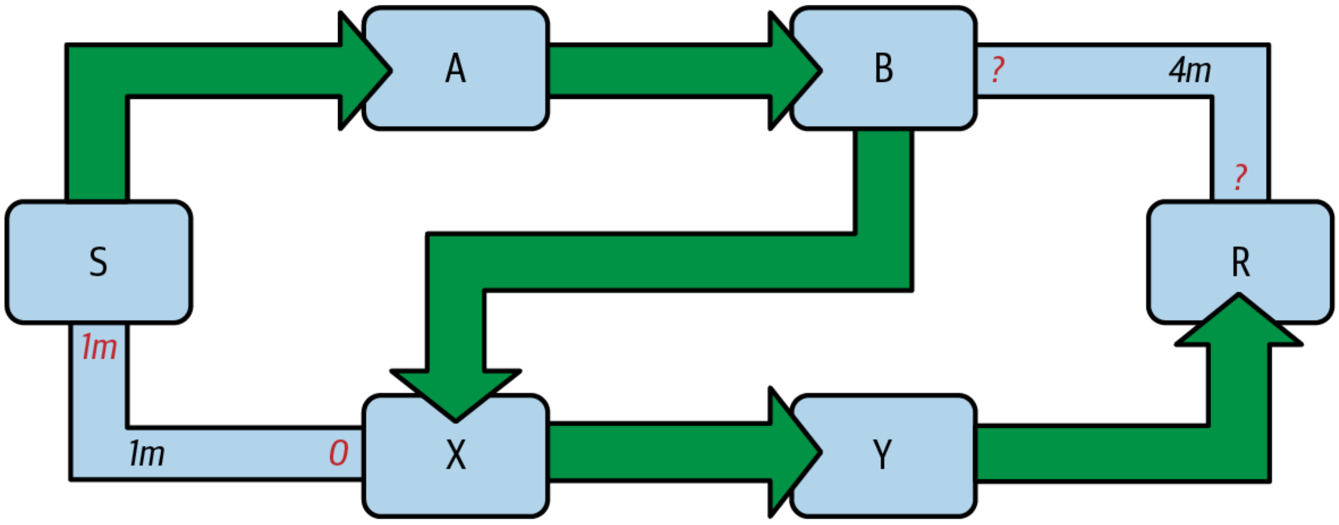


Figure 100. Path #4 attempt succeeds

Learning from success

Selena has also learnt a lot from this successful payment. She now knows that all the channels on the path S→A→B→X→Y→R had enough liquidity to deliver the payment. Furthermore, she now knows that each of these channels has moved the HTLC amount (1M + fees) to the other end of the channel. This allows Selena to recalculate the range of liquidity on the receiving side of all the channels in that path, replacing the minimum liquidity with 1M + fees.

Stale knowledge?

Selena now has a much better "map" of the Lightning Network (at least as far as these seven channels go). This knowledge will be useful for any subsequent payments that Selena attempts to make.

However, this knowledge becomes somewhat "stale" as the other nodes send or route payments. Selena will never see any of these payments (unless she is the sender). Even if she is involved in routing payments, the onion routing mechanism means she can only see the changes for one hop (her own channels).

Therefore, Selena's node must consider how long to keep this knowledge before assuming that it is stale and no longer useful.

Multipart Payments (MPP)

Multipart payments (MPP) are a feature that was introduced in the Lightning Network in 2020 and are already very widely available. Multipart payments allow a payment to be split into multiple *parts* which are sent as HTLCs over several different paths to the intended recipient, preserving the *atomicity* of the overall payment. In this context, atomicity means that either all the HTLC parts of a payment are eventually fulfilled or the entire payment fails and all the HTLC parts fail. There is no possibility of a partially successful payment.

Multipart payments are a significant improvement in the Lightning Network because they make it possible to send amounts that won't "fit" in any single channel by splitting them into smaller amounts for which there is sufficient liquidity. Furthermore, multipart payments have been shown to increase the probability of a successful payment, as compared to a single-path payment.

TIP

Now that MPP is available, it is best to think of a single-path payment as a subcategory of an MPP. Essentially, a single-path is just a multipart of size one. All payments can be considered as multipart payments unless the size of the payment and liquidity available make it possible to deliver with a single part.

Using MPP

MPP is not something that a user will select, but rather it is a node pathfinding and payment delivery strategy. The same basic steps are implemented: create a graph, select paths, and the trial-and-error loop. The difference is that during path selection we must also consider how to split the payment to optimize delivery.

In our example we can see some immediate improvements to our pathfinding problem that become possible with MPP. First, we can utilize the S→X channel that has known insufficient liquidity to transport 1M satoshis plus fees. By sending a smaller part along that channel, we can use paths that were previously unavailable. Second, we have the unknown liquidity of the B→R channel, which is insufficient to transport the 1M amount, but might be sufficient to transport a smaller amount.

Splitting payments

The fundamental question is how to split the payments. More specifically, what are the optimal number of splits and the optimal amounts for each split?

This is an area of ongoing research where novel strategies are emerging. Multipart payments lead to a different algorithmic approach than single-path payments, even though single-path solutions can emerge from a multipart optimization (i.e., a single path may be the optimal solution suggested by a multipart pathfinding algorithm).

If you recall, we found that the uncertainty of liquidity/balances leads to some (somewhat obvious) conclusions that we can apply in MPP pathfinding, namely:

- Smaller payments have a higher chance of succeeding.
- The more channels you use, the chance of success becomes (exponentially) lower.

From the first of these insights, we might conclude that splitting a large payment (e.g., 1 million satoshis) into tiny payments increases the chance that each of those smaller payments will succeed. The number of possible paths with sufficient liquidity will be greater if we send smaller amounts.

To take this idea to an extreme, why not split the 1M satoshi payment into one million separate one-satoshi parts? Well, the answer lies in our second insight: since we would be using more channels/paths to send our million single-satoshi HTLCs, our chance of success would drop exponentially.

If it’s not obvious, the two preceding insights create a "sweet spot" where we can maximize our chances of success: splitting into smaller payments but not too many splits!

Quantifying this optimal balance of size/number of splits for a given channel graph is out of the scope of this book, but it is an active area of research. Some current implementations use a very simple strategy of splitting the payment in two halves, four quarters, etc.

NOTE To read more about the optimization problem known as minimum-cost flows involved when splitting payments into different sizes and allocating them to paths, see the paper "Optimally Reliable & Cheap Payment Flows on the Lightning Network" by (coauthor of this book) René Pickhardt and Stefan Richter.

In our example, Selena's node will attempt to split the 1M satoshi payment into 2 parts with 600k and 400k satoshi, respectively, and send them on 2 different paths. This is shown in Sending two parts of a multipart payment.

Because the S→X channel can now be utilized, and (luckily for Selena), the B→R channel has sufficient liquidity for 600k satoshis, the 2 parts are successful along paths that were previously not possible.

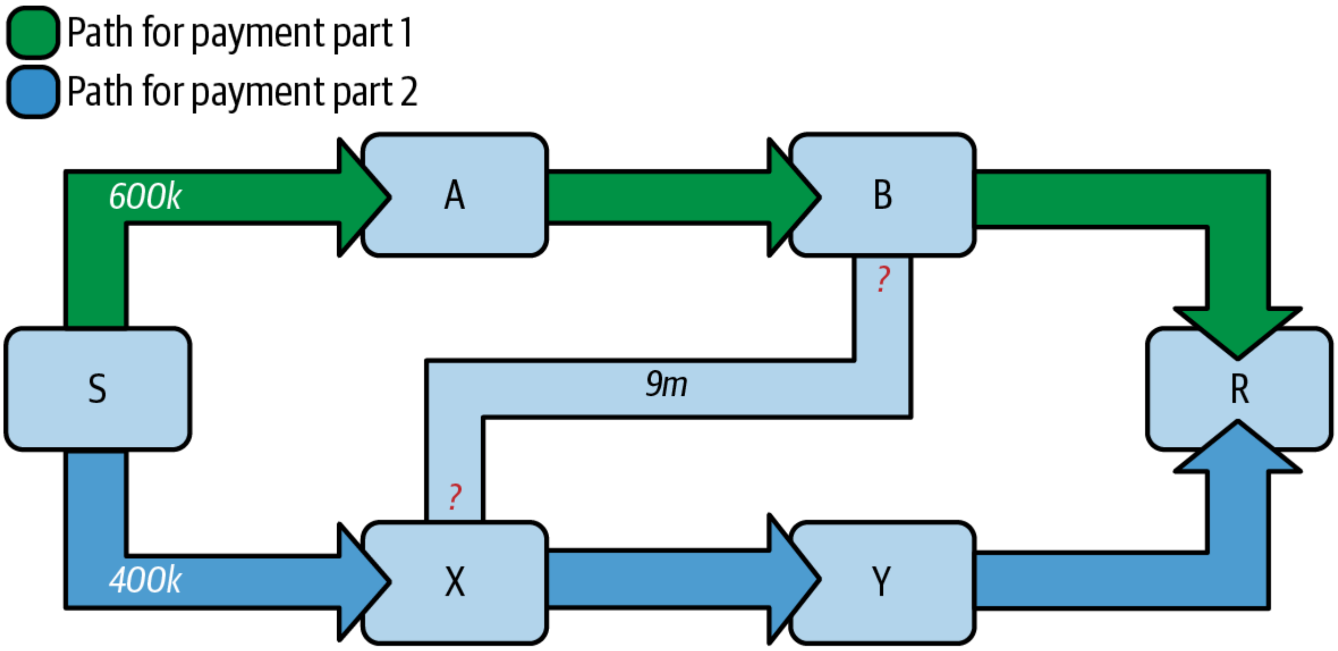


Figure 101. Sending two parts of a multipart payment

Trial and Error over Multiple "Rounds"

Multipart payments lead to a somewhat modified trial-and-error loop for payment delivery. Because we are attempting multiple paths in each attempt, we have four possible outcomes:

- All parts succeed, the payment is successful
- Some parts succeed, some fail with errors returned
- All parts fail with errors returned
- Some parts are "stuck," no errors are returned

In the second case, where some parts fail with errors returned and some parts succeed, we can now *repeat* the trial-and-error loop, but *only for the residual amount*.

Let's assume for example that Selena had a much larger channel graph with hundreds of possible paths to reach Rashid. Her pathfinding algorithm might find an optimal payment split consisting of 26 parts of varying sizes. After attempting to send all 26 parts in the first round, 3 of those parts failed with errors.

If those 3 parts consisted of, say 155k satoshis, then Selena would restart the pathfinding effort, only for 155k satoshis. The next round could find completely different paths (optimized for the residual amount of 155k), and split the 155k amount into completely different splits!

TIP While it seems like 26 split parts are a lot, tests on the Lightning Network have successfully delivered a payment of 0.3679 BTC by splitting it into 345 parts.

Furthermore, Selena's node would update the channel graph using the information gleaned from the successes and errors of the first round to find the most optimal paths and splits for the second round.

Let's say that Selena's node calculates that the best way to send the 155k residual is 6 parts split as 80k, 42k, 15k, 11k, 6.5k, and 500 satoshis. In the next round, Selena gets only one error, indicating that the 11k satoshi part failed. Again, Selena updates the channel graph based on the information gleaned and runs the pathfinding again to send the 11k residual. This time, she succeeds with 2 parts of 6k and 5k satoshis, respectively.

This multiround example of sending a payment using MPP is shown in Sending a payment in multiple rounds with MPP.

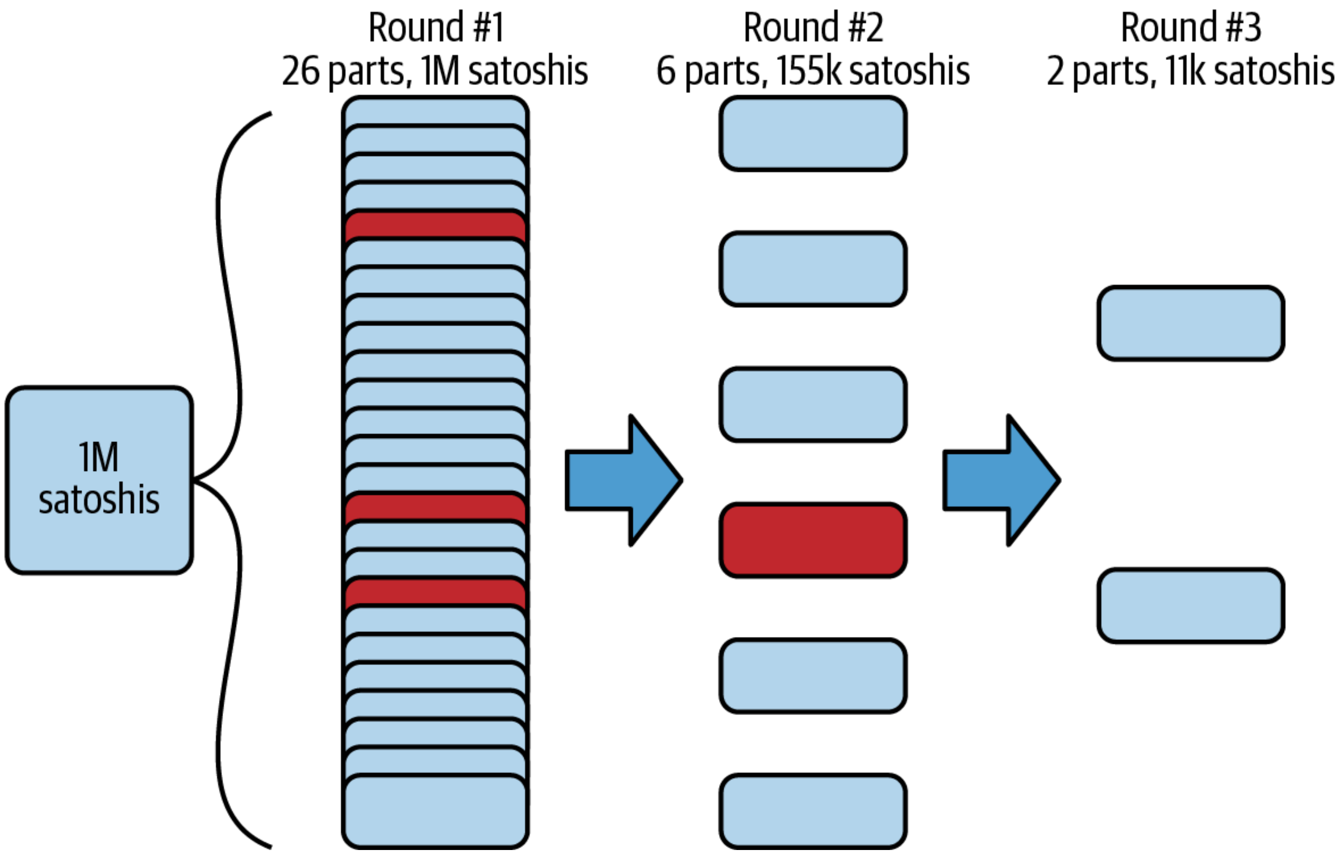


Figure 102. Sending a payment in multiple rounds with MPP

In the end, Selena's node used 3 rounds of pathfinding to send the 1M satoshis in 30 parts.

Conclusion

In this chapter we looked at pathfinding and payment delivery. We saw how to use the channel graph to find paths from a sender to a recipient. We also saw how the sender will attempt to deliver payments on a candidate path and repeat in a trial-and-error loop.

We also examined the uncertainty of channel liquidity (from the perspective of the sender) and the implications that has for pathfinding. We saw how we can quantify the uncertainty and use probability theory to draw some useful conclusions. We also saw how we can reduce uncertainty by learning from both successful and failed payments.

Finally, we saw how the newly deployed multipart payments feature allows us to split payments into parts, increasing the probability of success even for larger payments.

Wire Protocol: Framing and Extensibility

In this chapter, we dive into the wire protocol of the Lightning Network and also cover all the various extensibility levers that have been built into the protocol. By the end of this chapter, an ambitious reader should be able to write their very own wire protocol parser for the Lightning Network. In addition to being able to write a custom wire protocol parser, a reader of this chapter will gain a deep understanding of the various upgrade mechanisms that have been built into the protocol.

Messaging Layer in the Lightning Protocol Suite

The messaging layer, which is detailed in this chapter, consists of "Framing and message format," "Type-Length-Value (TLV)" encoding, and "Feature bits." These components are highlighted by an outline in the protocol suite, shown in Messaging layer in the Lightning protocol suite.

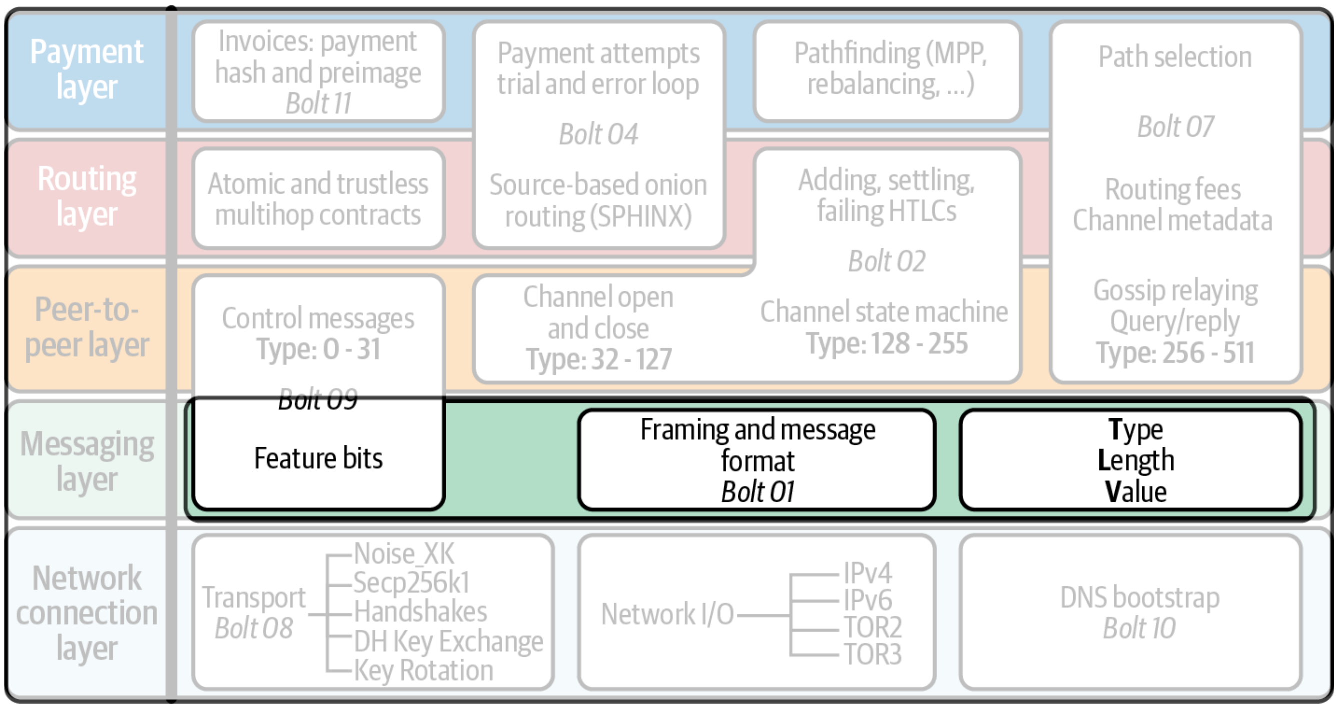

Payment layer
Invoices: payment hash and preimage Bolt 11
Payment attempts trial and error loop
Bolt 04
Source-based onion routing (SPHINX)
Pathfinding (MPP, rebalancing, ...)
Path selection
Bolt 07
Routing fees Channel metadata
Routing layer
Atomic and trustless multihop contracts
Adding, settling, failing HTLCs
Bolt 02
Channel state machine Type: 128 - 255
Peer-to-peer layer
Control messages Type: 0 - 31
Bolt 09
Channel open and close Type: 32 - 127
Gossip relaying Query/reply Type: 256 - 511
Messaging layer
Feature bits
Framing and 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 103. Messaging layer in the Lightning protocol suite

Wire Framing

We begin by describing the high-level structure of the wire *framing* within the protocol. When we say framing, we mean the way that the bytes are packed on the wire to *encode* a particular protocol message. Without knowledge of the framing system used in the protocol, a string of bytes on the wire would resemble a series of random bytes because no structure has been imposed. By applying proper framing to decode these bytes on the wire, we'll be able to extract structure and finally parse this structure into protocol messages within our higher-level language.

It's important to note that the Lightning Network is an *end-to-end encrypted* protocol, and the wire framing is itself encapsulated within an *encrypted* message transport layer. As we see in Lightning's Encrypted Message Transport, the Lightning Network uses a custom variant of the Noise Protocol to handle transport encryption. Within this chapter, whenever we give an example of wire framing, we assume the encryption layer has already been stripped away (when decoding), or that we haven't yet encrypted the set of bytes before we send them on the wire (encoding).

High-Level Wire Framing

With that said, we're ready to describe the high-level schema used to encode messages on the wire:

- Messages on the wire begin with a *2-byte* type field, followed by a message payload.
- The message payload itself can be up to 65 KB in size.
- All integers are encoded in big-endian (network order).
- Any bytes that follow after a defined message can be safely ignored.

Yep, that's it. As the protocol relies on an *encapsulating* transport protocol encryption layer, we don't need an explicit length for each message type. This is due to the fact that transport encryption works at the *message* level, so by the time we're ready to decode the next message, we already know the total number of bytes of the message itself. Using 2 bytes for the message type (encoded in big-endian) means that the protocol can have up to 2^16 – 1 or 65,535 distinct messages. Continuing, because we know all messages must be less than 65 KB, this simplifies our parsing as we can use a *fixed-size* buffer and maintain strong bounds on the total amount of memory required to parse an incoming wire message.

The final bullet point allows for a degree of *backward* compatibility because new nodes are able to provide information in the wire messages that older nodes (which may not understand them) can safely ignore. As we see subsequently, this feature, combined with a very flexible wire message extensibility format, allows the protocol to achieve *forward* compatibility as well.

Type Encoding

With this high-level background provided, we now start at the most primitive layer: parsing primitive types. In addition to encoding integers, the Lightning Protocol also allows for encoding of a vast array of types, including variable-length byte slices, elliptic curve public keys, Bitcoin addresses, and signatures. When we describe the *structure* of wire messages later in this chapter, we refer to the high-level type (the abstract type) rather than the lower-level representation of said type. In this section, we peel back this abstraction layer to ensure that our future wire parser is able to properly encode/decode any of the higher-level types.

In High-level message types, we map the name of a given message type to the high-level routine used to encode/decode the type.

Table 6. High-level message types

High-level type	Framing	Comment
`node_alias`	A 32-byte fixed-length byte slice	When decoding, reject if contents are not a valid UTF-8 string

High-level type	Framing	Comment
`channel_id`	A 32-byte fixed-length byte slice that maps an outpoint to a 32-byte value	Given an outpoint, one can convert it to a `channel_id` by taking the TxID of the outpoint and XORing it with the index (interpreted as the lower 2 bytes)
`short_chan_id`	An unsigned 64-bit integer (`uint64`)	Composed of the block height (24 bits), transaction index (24 bits), and output index (16 bits) packed into 8 bytes
`milli_satoshi`	An unsigned 64-bit integer (`uint64`)	Represents 1000th of a satoshi
`satoshi`	An unsigned 64-bit integer (`uint64`)	The base unit of bitcoin
`pubkey`	An secp256k1 public key encoded in *compressed* format, occupying 33 bytes	Occupies a fixed 33-byte length on the wire
`sig`	An ECDSA signature of the secp256k1 elliptic curve	Encoded as a *fixed* 64-byte byte slice, packed as `R || S`
`uint8`	An 8-bit integer	
`uint16`	A 16-bit integer	
`uint64`	A 64-bit integer	
`[]byte`	A variable-length byte slice	Prefixed with a 16-bit integer denoting the length of the bytes
`color_rgb`	RGB color encoding	Encoded as a series of 8-bit integers
`net_addr`	The encoding of a network address	Encoded with a 1-byte prefix that denotes the type of address, followed by the address body

In the next section, we describe the structure of each wire message, including the prefix type of the message along with the contents of its message body.

Type-Length-Value (TLV) Message Extensions

Earlier in this chapter we mentioned that messages can be up to 65 KB in size, and if while parsing a message, extra bytes are left over, then those bytes are to be ignored. At an initial glance, this requirement may appear to be somewhat arbitrary; however, this requirement allows for decoupled desynchronized evolution of the Lightning Protocol itself. We discuss this more toward the end of the chapter. But first, we turn our attention to exactly what those "extra bytes" at the end of a message can be used for.

The Protocol Buffers Message Format

The Protocol Buffers (Protobuf) message serialization format started out as an internal format used at Google and has blossomed into one of the most popular message serialization formats used by developers globally. The Protobuf format describes how a message (usually some sort of data structure related to an API) is encoded on the wire and decoded on the other end. Several "Protobuf compilers" exists in dozens of languages which act as a bridge that allows any language to encode a Protobuf that will be able to decode by a compliant decode in another language. Such cross-language data structure compatibility allows for a wide range of innovation because it’s possible to transmit structure and even typed data structures across language and abstraction boundaries.

Protobufs are also known for their flexibility with respect to how they handle changes in the underlying messages structure. As long as the field numbering schema is adhered to, then it’s possible for a newer write of Protobufs to include information within a Protobuf that may be unknown to any older readers. When the old reader encounters the new serialized format, if there are types/fields that it doesn’t understand, then it simply *ignores* them. This allows old clients and new clients to coexist because all clients can parse some portion of the newer message format.

Forward and Backward Compatibility

Protobufs are extremely popular amongst developers because they have built-in support for both forward and backward compatibility. Most developers are likely familiar with the concept of backward compatibility. In simple terms, the principle states that any changes to a message format or API should be done in a manner that doesn’t break support for older clients. Within our preceding Protobuf extensibility examples, backward compatibility is achieved by ensuring that new additions to the Protobuf format don’t break the known portions of older readers. Forward compatibility, on the other hand, is just as important for desynchronized updates; however, it’s less commonly known. For a change to be forward compatible, clients are to simply ignore any information they don’t understand. The soft fork mechanism of upgrading the Bitcoin consensus system can be said to be both forward and backward compatible: any clients that don’t update can still use Bitcoin, and if they encounter any transactions they don’t understand, then they simply ignore them as their funds aren’t using those new features.

Type-Length-Value (TLV) Format

To be able to upgrade messages in a manner that is both forward and backward compatible, in addition to feature bits (more on that later), the Lightning Network utilizes a custom message serialization format plainly called Type-Length-Value, or TLV for short. The format was inspired by the widely used Protobuf format and borrows many concepts by significantly simplifying the implementation as well as the software that interacts with message parsing. A curious reader might ask, "why not just use Protobufs?" In response, the Lightning developers would respond that we’re able to have the best of the extensibility of Protobufs while also having the benefit of a smaller implementation and thus smaller attack. As of version 3.15.6, the Protobuf compiler weighs in at over 656,671 lines of code. In comparison, LND’s implementation of the TLV message format weighs in at only 2.3k lines of code (including tests).

With the necessary background presented, we’re now ready to describe the TLV format in detail. A TLV message extension is said to be a stream of individual TLV records. A single TLV record has three components: the type of the record, the length of the record, and finally the opaque value of the record:

`type`
: An integer representing the name of the record being encoded

`length`
: The length of the record

`value`
: The opaque value of the record

Both the `type` and `length` are encoded using a variable-sized integer that's inspired by the variable-sized integer (varint) used in Bitcoin's P2P protocol, called `BigSize` for short.

BigSize Integer Encoding

In its fullest form, a `BigSize` integer can represent value up to 64 bits. In contrast to Bitcoin's varint format, the `BigSize` format instead encodes integers using a big-endian byte ordering.

The `BigSize` varint has two components: the discriminant and the body. In the context of the `BigSize` integer, the discriminant communicates to the decoder the size of the variable-sized integer that follows. Remember that the unique thing about variable-sized integers is that they allow a parser to use fewer bytes to encode smaller integers than larger ones, saving space. Encoding of a `BigSize` integer follows one of the four following options:

1. If the value is less than `0xfd` (`253`): Then the discriminant isn't really used, and the encoding is simply the integer itself. This allows us to encode very small integers with no additional overhead.
2. If the value is less than or equal to `0xffff` (`65535`): The discriminant is encoded as `0xfd`, which indicates that the value that follows is larger than `0xfd`, but smaller than `0xffff`. The number is then encoded as a 16-bit integer. Including the discriminant, we can encode a value that is greater than 253, but less than 65,535 using 3 bytes.
3. If the value is less than `0xffffffff` (`4294967295`): The discriminant is encoded as `0xfe`. The body is encoded using a 32-bit integer, including the discriminant, and we can encode a value that's less than `4,294,967,295` using 5 bytes.
4. Otherwise, we just encode the value as a full-size 64-bit integer.

TLV Encoding Constraints

Within the context of a TLV message, record types below `2^16` are said to be *reserved* for future use. Types beyond this range are to be used for "custom" message extensions used by higher-level application protocols.

The `value` of a record depends on the `type`. In other words, it can take any form because parsers will attempt to interpret it depending on the context of the type itself.

TLV Canonical Encoding

One issue with the Protobuf format is that encodings of the same message may output an entirely different set of bytes when encoded by two different versions of the compiler. Such instances of a noncanonical encoding are not acceptable within the context of Lightning, as many messages contain a signature of the message digest. If it's possible for a message to be encoded in two different ways, then it would be possible to break the authentication of a signature inadvertently by re-encoding a message using a slightly different set of bytes on the wire.

To ensure that all encoded messages are canonical, the following constraints are defined when encoding:

- All records within a TLV stream must be encoded in order of strictly increasing type.
- All records must minimally encode the `type` and `length` fields. In other words, the smallest `BigSize` representation for an integer must be used at all times.
- Each `type` may only appear once within a given TLV stream.

In addition to these encoding constraints, a series of higher-level interpretation requirements is also defined based on the *arity* of a given `type` integer. We dive further into these details toward the end of the chapter once we describe how the Lightning Protocol is upgraded in practice and in theory.

Feature Bits and Protocol Extensibility

Because the Lightning Network is a decentralized system, no single entity can enforce a protocol change or modification upon all the users of the system. This characteristic is also seen in other decentralized networks such as Bitcoin. However, unlike Bitcoin, overwhelming consensus *is not* required to change a subset of the Lightning Network. Lightning is able to evolve at will without a strong requirement of coordination because, unlike Bitcoin, there is no global consensus required in the Lightning Network. Due to this fact and the several upgrade mechanisms embedded in the Lightning Network, only the participants that wish to use these new Lightning Network features need to upgrade, and then they are able to interact with each other.

In this section, we explore the various ways that developers and users are able to design and deploy new features to the Lightning Network. The designers of the original Lightning Network knew that there were many possible future directions for the network and the underlying protocol. As a result, they made sure to implement several extensibility mechanisms within the system, which can be used to upgrade it partially or fully in a decoupled, desynchronized, and decentralized manner.

Feature Bits as an Upgrade Discoverability Mechanism

An astute reader may have noticed the various locations where feature bits are included within the Lightning Protocol. A *feature bit* is a bitfield that can be used to advertise understanding or adherence to a possible network protocol update. Feature bits are commonly assigned in pairs, meaning that each potential new feature/upgrade always defines two bits within the bitfield. One bit signals that the advertised feature is *optional*, meaning that the node knows about the feature and can use it, but doesn't consider it required for normal operation. The other bit signals that the feature is instead *required*, meaning that the node will not continue operation if a prospective peer doesn’t understand that feature.

Using these two bits (optional and required), we can construct a simple compatibility matrix that nodes/users can consult to determine if a peer is compatible with a desired feature, as shown in Feature bit compatibility matrix.

Table 7. Feature bit compatibility matrix

Bit type	**Remote optional**	**Remote required**	**Remote unknown**
Local optional	□	□	□
Local required	□	□	□
Local unknown	□	□	□

From this simplified compatibility matrix, we can see that as long as the other party knows about our feature bit, then we can interact with them using the protocol. If the party doesn’t even know about what bit we’re referring to *and* they require the feature, then we are incompatible with them. Within the network, optional features are signaled using an *odd bit number*, while required features are signaled using an *even bit number*. As an example, if a peer signals that they know of a feature that uses bit 15, then we know that this is an optional feature, and we can interact with them or respond to their messages even if we don’t know about the feature. If they instead signaled the feature using bit 16, then we know this is a required feature, and we can’t interact with them unless our node also understands that feature.

The Lightning developers have come up with an easy-to-remember phrase that encodes this matrix: "it’s OK to be odd." This simple rule allows for a rich set of interactions within the protocol, as a simple bitmask operation between two feature bit vectors allows peers to determine if certain interactions are compatible with each other or not. In other words, feature bits are used as an upgrade discoverability mechanism: they easily allow to peers to understand if they are compatible or not based on the concepts of optional, required, and unknown feature bits.

Feature bits are found in the `node_announcement`, `channel_announcement`, and `init` messages within the protocol. As a result, these three messages can be used to signal the knowledge and/or understanding of in-flight protocol updates within the network. The feature bits found in the `node_announcement` message can allow a peer to determine if their *connections* are compatible or not. The feature bits within the `channel_announcement` messages allow a peer to determine if a given payment type or HTLC can transit through a given peer or not. The feature bits within the `init` message allow peers to understand if they can maintain a connection, and also which features are negotiated for the lifetime of a given connection.

TLV for Forward and Backward Compatibility

As we learned earlier in the chapter, TLV records can be used to extend messages in a forward and backward compatible manner. Over time, these records have been used to extend existing messages without breaking the protocol by utilizing the "undefined" area within a message beyond that set of known bytes.

As an example, the original Lightning Protocol didn’t have a concept of the "largest amount HTLC" that could traverse through a channel as dictated by a routing policy. Later on, the `max_htlc` field was added to the `channel_update` message to phase in this concept over time. Peers that receive a `channel_update` that sets such a field but don't even know the upgrade existed are unaffected by the change, but have their HTLCs rejected if they are beyond the limit. Newer peers, on the other hand, are able to parse, verify, and utilize the new field.

Those familiar with the concept of soft forks in Bitcoin may now see some similarities between the two mechanisms. Unlike Bitcoin consensus-level soft forks, upgrades to the Lightning Network don't require overwhelming consensus to be adopted. Instead, at minimum, only two peers within the network need to understand a new upgrade to start using it. Commonly these two peers may be the recipient and sender of a payment, or may be the channel partners of a new payment channel.

A Taxonomy of Upgrade Mechanisms

Rather than there being a single widely utilized upgrade mechanism within the network (such as soft forks for Bitcoin), there exist several possible upgrade mechanisms within the Lightning Network. In this section, we enumerate these upgrade mechanisms and provide a real-world example of their use in the past.

Internal network upgrades

We start with the upgrade type that requires the most protocol-level coordination: internal network upgrades. An internal network upgrade is characterized by one that requires *every single node* within a prospective payment path to understand the new feature. Such an upgrade is similar to any upgrade within the internet that requires hardware-level upgrades within the core-relay portion of the upgrade. In the context of the Lightning Network, however, we deal with pure software, so such upgrades are easier to deploy, yet they still require much more coordination than any other upgrade mechanism in the network.

One example of such an upgrade within the network was the introduction of a TLV encoding for the routing information encoded within the onion packets. The prior format used a hardcoded fixed-length message format to communicate information such as the next hop. Because this format was fixed, it meant that new protocol-level upgrades weren't possible. The move to the more flexible TLV format meant that after this upgrade, any sort of feature that modified the type of information communicated at each hop could be rolled out at will.

It's worth mentioning that the TLV onion upgrade was a sort of "soft" internal network upgrade, in that if a payment wasn't using any new feature beyond that new routing information encoding, then a payment could be transmitted using a mixed set of nodes.

End-to-end upgrades

To contrast the internal network upgrade, in this section we describe the *end-to-end* network upgrade. This upgrade mechanism differs from the internal network upgrade in that it only requires the "ends" of the payment, the sender and recipient, to upgrade.

This type of upgrade allows for a wide array of unrestricted innovation within the network. Because of the onion encrypted nature of payments within the network, those forwarding HTLCs within the center of the network may not even know that new features are being utilized.

One example of an end-to-end upgrade within the network was the rollout of multipart payments

(MPP). MPP is a protocol-level feature that enables a single payment to be split into multiple parts or paths, to be assembled at the recipient for settlement. The rollout of MPP was coupled with a new `node_announcement` level feature bit that indicates that the recipient knows how to handle partial payments. Assuming a sender and recipient know about each other (possibly via a BOLT #11 invoice), then they're able to use the new feature without any further negotiation.

Another example of an end-to-end upgrade are the various types of *spontaneous* payments deployed within the network. One early type of spontaneous payments called *keysend* worked by simply placing the preimage of a payment within the encrypted onion. Upon receipt, the destination would decrypt the preimage, then use that to settle the payment. Because the entire packet is end-to-end encrypted, this payment type was safe, since none of the intermediate nodes are able to fully unwrap the onion to uncover the payment preimage.

Channel Construction-Level Updates

The final broad category of updates are those that happen at the channel construction level, but which don't modify the structure of the HTLC used widely within the network. When we say channel construction, we mean how the channel is funded or created. As an example, the eltoo channel type can be rolled out within the network using a new `node_announcement` level feature bit as well as a `channel_announcement` level feature bit. Only the two peers on the sides of the channels need to understand and advertise these new features. This channel pair can then be used to forward any payment type granted the channel supports it.

Another is the *anchor outputs* channel format which allows the commitment fee to be bumped via Bitcoin's Child-Pays-For-Parent (CPFP) fee management mechanism.

Conclusion

Lightning's wire protocol is incredibly flexible and allows for rapid innovation and interoperability without strict consensus. It is one of the reasons that the Lightning Network is experiencing much faster development and is attractive to many developers, who might otherwise find Bitcoin's development style too conservative and slow.

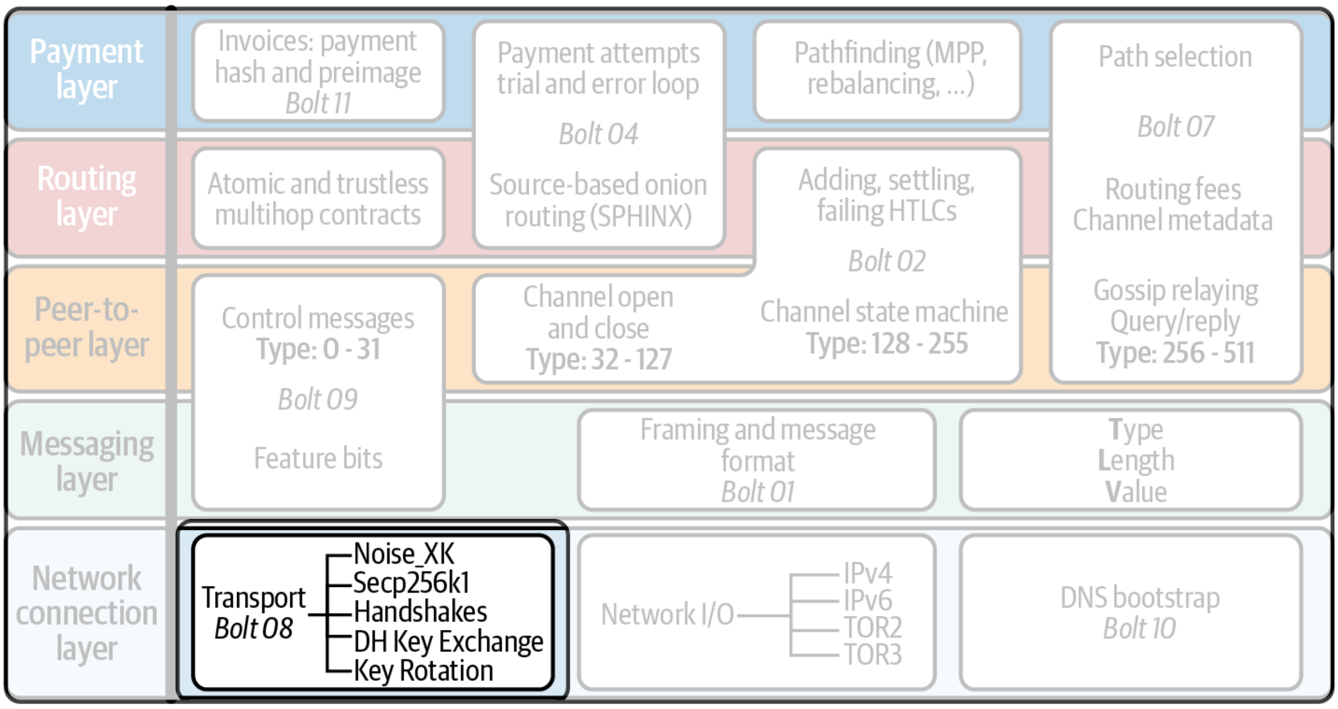
Lightning's Encrypted Message Transport

In this chapter we will review the Lightning Network's *encrypted message transport*, sometimes referred to as the *Brontide Protocol*, which allows peers to establish end-to-end encrypted communication, authentication, and integrity checking.

NOTE Part of this chapter includes some highly technical detail about the encryption protocol and encryption algorithms used in the Lightning encrypted transport. You may decide to skip that section if you are not interested in those details.

Encrypted Transport in the Lightning Protocol Suite

The transport component of the Lightning Network and its several components are shown in the leftmost part of the network connection layer in Encrypted message transport in the Lightning protocol suite.



Payment layer
Routing layer
Peer-to-peer layer
Messaging layer
Network connection layer
Invoices: payment hash and preimage Bolt 11
Payment attempts trial and error loop Bolt 04 Source-based onion routing (SPHINX)
Pathfinding (MPP, rebalancing, ...)
Path selection Bolt 07 Routing fees Channel metadata Gossip relaying Query/reply Type: 256 - 511
Atomic and trustless multihop contracts
Adding, settling, failing HTLCs Bolt 02 Channel state machine Type: 128 - 255
Control messages Type: 0 - 31 Bolt 09 Feature bits
Channel open and close Type: 32 - 127
Framing and message format Bolt 01
Type Length Value
Transport Bolt 08
Noise_XK
Secp256k1
Handshakes
DH Key Exchange
Key Rotation
Network I/O
IPv4
IPv6
TOR2
TOR3
DNS bootstrap Bolt 10

Figure 104. Encrypted message transport in the Lightning protocol suite

Introduction

Unlike the vanilla Bitcoin P2P network, every node in the Lightning Network is identified by a unique public key which serves as its identity. By default, this public key is used to end-to-end encrypt *all* communication within the network. Encryption by default at the lowest level of the protocol ensures that all messages are authenticated, are immune to man-in-the-middle (MITM) attacks and snooping by third parties, and ensures privacy at the fundamental transport level. In this chapter, we'll learn about the encryption protocol used by the Lightning Network in detail. Upon completion of this chapter, the reader will be familiar with the state of the art in encrypted messaging protocols, as well as the various properties such a protocol provides to the network. It's worth mentioning that the core of the encrypted message transport is *agnostic* to its usage within the context of the Lightning Network. As a result, the custom encrypted message transport that

Lightning uses can be dropped into any context that requires encrypted communication between two parties.

The Channel Graph as Decentralized Public Key Infrastructure

As we learned in Routing on a Network of Payment Channels, every node has a long-term identity that is used as the identifier for a vertex during pathfinding and also used in the asymmetric cryptographic operations related to the creation of onion encrypted routing packets. This public key, which serves as a node's long-term identity, is included in the DNS bootstrapping response, as well as embedded within the channel graph. As a result, before a node attempts to connect out to another node on the P2P network, it already knows the public key of the node it wishes to connect to.

Additionally, if the node being connected to already has a series of public channels within the graph, then the connecting node is able to further verify the identity of the node. Because the entire channel graph is fully authenticated, one can view it as a sort of decentralized public key infrastructure (PKI): to register a key, a public channel in the Bitcoin blockchain must be opened, and once a node no longer has any public channels, then they've effectively been removed from the PKI.

Because Lightning is a decentralized network, it's imperative that no one central party is designated the power to provision a public key identity within the network. In place of a central party, the Lightning Network uses the Bitcoin blockchain as a Sybil mitigation mechanism because gaining an identity on the network has a tangible cost: the fee needed to create a channel in the blockchain, as well as the opportunity cost of the capital allocated to their channels. In the process of essentially rolling a domain-specific PKI, the Lightning Network is able to significantly simplify its encrypted transport protocol as it doesn't need to deal with all the complexities that come along with TLS, the Transport Layer Security protocol.

Why Not TLS?

Readers familiar with the TLS system may be wondering at this point: why wasn't TLS used in spite of the drawbacks of the existing PKI system? It is indeed a fact that "self-signed certificates" can be used to effectively sidestep the existing global PKI system by simply asserting to the identity of a given public key amongst a set of peers. However, even with the existing PKI system out of the way, TLS has several drawbacks that prompted the creators of the Lightning Network to instead opt for a more compact custom encryption protocol.

To start with, TLS is a protocol that has been around for several decades and as a result has evolved over time as new advances have been made in the space of transport encryption. However, over time this evolution has caused the protocol to balloon in size and complexity. Over the past few decades, several vulnerabilities in TLS have been discovered and patched, with each evolution further increasing the complexity of the protocol. As a result of the age of the protocol, several versions and iterations exist, meaning a client needs to understand many of the prior iterations of the protocol to communicate with a large portion of the public internet, further increasing implementation complexity.

In the past, several memory safety vulnerabilities have been discovered in widely used implementations of SSL/TLS. Packaging such a protocol within every Lightning node would serve to increase the attack surface of nodes exposed to the public peer-to-peer network. To increase the security of the network as a whole and minimize exploitable attack surface, the creators of the Lightning Network instead opted to adopt the Noise Protocol Framework. Noise as a protocol internalizes several of the security and privacy lessons learned over time due to continual scrutiny of the TLS protocol over decades. In a way, the existence of Noise allows the community to effectively "start over," with a more compact, simplified protocol that retains all the added benefits of TLS.

The Noise Protocol Framework

The Noise Protocol Framework is a modern, extensible, and flexible message encryption protocol designed by the creators of the Signal Protocol. The Signal Protocol is one of the most widely used message encryption protocols in the world. It's used by both Signal and Whatsapp, which cumulatively are used by over a billion people around the world. The Noise framework is the result of decades of evolution both within academia as well as the industry of message encryption protocols. Lightning uses the Noise Protocol Framework to implement a *message-oriented* encryption protocol used by all nodes to communicate with each other.

A communication session using Noise has two distinct phases: the handshake phase and the messaging phase. Before two parties can communicate with each other, they first need to arrive at a shared secret known only to them which will be used to encrypt and authenticate messages sent to each other. A flavor of an authenticated key agreement is used to arrive at a final shared key between the two parties. In the context of the Noise protocol, this authenticated key agreement is referred to as a *handshake*. Once that handshake has been completed, both nodes can now being to send each other encrypted messages. Each time peers need to connect or reconnect to each other, a fresh iteration of the handshake protocol is executed, ensuring that forward secrecy is achieved (leaking the key of a prior transcript doesn't compromise any future transcripts).

Because the Noise Protocol allows a protocol designer to choose from several cryptographic primitives, such as symmetric encryption and public key cryptography, it's customary that each flavor of the Noise Protocol is referred to by a unique name. In the spirit of "Noise," each flavor of the protocol selects a name derived from some sort of "noise." In the context of the Lightning Network, the flavor of the Noise Protocol used is sometimes referred to as Brontide. A *brontide* is a low billowing noise, similar to what one would hear during a thunderstorm when very far away.

Lightning Encrypted Transport in Detail

In this section we will break down the Lightning encrypted transport protocol and delve into the details of the cryptographic algorithms and protocol used to establish encrypted, authenticated, and integrity-assured communications between peers. Feel free to skip this section if you find this level of detail daunting.

Noise XK: Lightning Network's Noise Handshake

The Noise Protocol is extremely flexible in that it advertises several handshakes, each with different security and privacy properties for a would-be protocol implementer to select from. A deep exploration of each of the handshakes and their various trade-offs is out of the scope of this chapter. With that said, the Lightning Network uses a specific handshake referred to as `Noise_XK`. The unique property provided by this handshake is *identity hiding*: in order for a node to initiate a connection with another node, it must first know its public key. Mechanically, this means that the public key of the responder is actually never transmitted during the context of the handshake. Instead, a clever series of Elliptic Curve Diffie–Hellman (ECDH) and message authentication code (MAC) checks are used to authenticate the responder.

Handshake Notation and Protocol Flow

Each handshake typically consists of several steps. At each step some (possibly) encrypted material is sent to the opposite party, an ECDH (or several) is performed, with the result of the handshake being "mixed" into a protocol *transcript*. This transcript serves to authenticate each step of the protocol and helps thwart a flavor of man-in-the-middle attacks. At the end of the handshake, two keys, `ck` and `k`, are produced which are used to encrypt messages (`k`) and rotate keys (`ck`) throughout the lifetime of the session.

In the context of a handshake, `s` is usually a long-term static public key. In our case, the public key crypto system used is an elliptic curve one, instantiated with the `secp256k1` curve, which is used elsewhere in Bitcoin. Several ephemeral keys are generated throughout the handshake. We use `e` to refer to a new ephemeral key. ECDH operations between two keys are notated as the concatenation of two keys. As an example, `ee` represents an ECDH operation between two ephemeral keys.

High-Level Overview

Using the notation laid out earlier, we can succinctly describe the `Noise_XK` as follows:

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

The protocol begins with the "pretransmission" of the responder's static key (`rs`) to the initiator. Before executing the handshake, the initiator is to generate its own static key (`s`). During each step of the handshake, all material sent across the wire and the keys sent/used are incrementally hashed into a *handshake digest*, `h`. This digest is never sent across the wire during the handshake, and is instead used as the "associated data" when AEAD (authenticated encryption with associated data) is sent across the wire. *Associated data* (AD) allows an encryption protocol to authenticate additional information alongside a cipher text packet. In other domains, the AD may be a domain name, or plain-text portion of the packet.

The existence of `h` ensures that if a portion of a transmitted handshake message is replaced, then the other side will notice. At each step, a MAC digest is checked. If the MAC check succeeds, then the receiving party knows that the handshake has been successful up until that point. Otherwise, if a MAC check ever fails, then the handshake process has failed, and the connection should be terminated.

The protocol also adds a new piece of data to each handshake message: a protocol version. The initial protocol version is `0`. At the time of writing, no new protocol versions have been created. As a result, if a peer receives a version other than `0`, then they should reject the handshake initiation attempt.

As far as cryptographic primitives, SHA-256 is used as the hash function of choice, `secp256k1` as the elliptic curve, and `ChaChaPoly-130` as the AEAD (symmetric encryption) construction.

Each variant of the Noise Protocol has a unique ASCII string used to refer to it. To ensure that two parties are using the same protocol variant, the ASCII string is hashed into a digest, which is used to initialize the starting handshake state. In the context of the Lightning Network, the ASCII string describing the protocol is `Noise_XK_secp256k1_ChaChaPoly_SHA256`.

Handshake in Three Acts

The handshake portion can be separated into three distinct "acts." The entire handshake takes 1.5 round trips between the initiator and responder. At each act, a single message is sent between both parties. The handshake message is a fixed-size payload prefixed by the protocol version.

The Noise Protocol uses an object-oriented inspired notation to describe the protocol at each step. During setup of the handshake state, each side will initialize the following variables:

`ck`

The *chaining key*. This value is the accumulated hash of all previous ECDH outputs. At the end of the handshake, `ck` is used to derive the encryption keys for Lightning messages.

`h`

The *handshake hash*. This value is the accumulated hash of *all* handshake data that has been sent and received so far during the handshake process.

`temp_k1`, `temp_k2`, `temp_k3`

The *intermediate keys*. These are used to encrypt and decrypt the zero-length AEAD payloads at the end of each handshake message.

`e`

A party's *ephemeral keypair*. For each session, a node must generate a new ephemeral key with strong cryptographic randomness.

`s`

A party's *static keypair* (`ls` for local, `rs` for remote).

Given this handshake plus messaging session state, we'll then define a series of functions that will operate on the handshake and messaging state. When describing the handshake protocol, we'll use these variables in a manner similar to pseudocode to reduce the verbosity of the explanation of each step in the protocol. We'll define the *functional* primitives of the handshake as:

`ECDH(k, rk)`

Performs an Elliptic Curve Diffie–Hellman operation using `k`, which is a valid `secp256k1` private key, and `rk`, which is a valid public key.

The returned value is the SHA-256 of the compressed format of the generated point.

`HKDF(salt,ikm)`

A function defined in `RFC 5869`, evaluated with a zero-length `info` field.

All invocations of `HKDF` implicitly return 64 bytes of cryptographic randomness using the extract-and-expand component of the `HKDF`.

`encryptWithAD(k, n, ad, plaintext)`

Outputs `encrypt(k, n, ad, plaintext)`.

Where `encrypt` is an evaluation of `ChaCha20-Poly1305` (Internet Engineering Task Force variant) with the passed arguments, with nonce `n` encoded as 32 zero bits, followed by a *little-endian* 64-bit value. Note: this follows the Noise Protocol convention, rather than our normal endian.

`decryptWithAD(k, n, ad, ciphertext)`

Outputs `decrypt(k, n, ad, ciphertext)`.

Where `decrypt` is an evaluation of `ChaCha20-Poly1305` (IETF variant) with the passed arguments, with nonce `n` encoded as 32 zero bits, followed by a *little-endian* 64-bit value.

`generateKey()`

Generates and returns a fresh `secp256k1` keypair.

Where the object returned by `generateKey` has two attributes:`.pub`, which returns an abstract object representing the public key; and `.priv`, which represents the private key used to generate the public key

Where the object also has a single method: `.serializeCompressed()`

`a || b`

This denotes the concatenation of two byte strings `a` and `b`.

Handshake session state initialization

Before starting the handshake process, both sides need to initialize the starting state that they'll use to advance the handshake process. To start, both sides need to construct the initial handshake digest `h`.

1. h = SHA-256(__protocolName__)

   Where __protocolName__ = "Noise_XK_secp256k1_ChaChaPoly_SHA256" encoded as an ASCII string.

2. `ck = h`
3. h = SHA-256(h || __prologue__)

   Where __prologue__ is the ASCII string: `lightning`.

In addition to the protocol name, we also add in an extra "prologue" that is used to further bind the protocol context to the Lightning Network.

To conclude the initialization step, both sides mix the responder's public key into the handshake digest. Because this digest is used while the associated data with a zero-length ciphertext (only the MAC) is sent, this ensures that the initiator does indeed know the public key of the responder.

- The initiating node mixes in the responding node's static public key serialized in Bitcoin's compressed format: `h = SHA-256(h || rs.pub.serializeCompressed())`
- The responding node mixes in their local static public key serialized in Bitcoin's compressed format: `h = SHA-256(h || ls.pub.serializeCompressed())`

Handshake acts

After the initial handshake initialization, we can begin the actual execution of the handshake process. The handshake is composed of a series of three messages sent between the initiator and responder, henceforth referred to as "acts." Because each act is a single message sent between the parties, a handshake is completed in a total of 1.5 round trips (0.5 for each act).

The first act completes the initial portion of the incremental triple Diffie–Hellman (DH) key exchange (using a new ephemeral key generated by the initiator) and also ensures that the initiator actually knows the long-term public key of the responder. During the second act, the responder transmits the ephemeral key they wish to use for the session to the initiator, and once again incrementally mixes this new key into the triple DH handshake. During the third and final act, the initiator transmits their long-term static public key to the responder and executes the final DH operation to mix that into the final resulting shared secret.

Act One

```
    -> e, es
```

Act One is sent from initiator to responder. During Act One, the initiator attempts to satisfy an implicit challenge by the responder. To complete this challenge, the initiator must know the static public key of the responder.

The handshake message is *exactly* 50 bytes: 1 byte for the handshake version, 33 bytes for the compressed ephemeral public key of the initiator, and 16 bytes for the `poly1305` tag.

Sender actions:

1. `e = generateKey()`
2. `h = SHA-256(h || e.pub.serializeCompressed())`

   The newly generated ephemeral key is accumulated into the running handshake digest.

3. `es = ECDH(e.priv, rs)`

   The initiator performs an ECDH between its newly generated ephemeral key and the remote node's static public key.

4. `ck, temp_k1 = HKDF(ck, es)`

   A new temporary encryption key is generated, which is used to generate the authenticating MAC.

5. `c = encryptWithAD(temp_k1, 0, h, zero)`

   Where `zero` is a zero-length plain text.

6. `h = SHA-256(h || c)`

   Finally, the generated ciphertext is accumulated into the authenticating handshake digest.

7. Send `m = 0 || e.pub.serializeCompressed() || c` to the responder over the network buffer.

Receiver actions:

1. Read *exactly* 50 bytes from the network buffer.
2. Parse the read message (`m`) into `v`, `re`, and `c`:
   - Where `v` is the *first* byte of `m`, `re` is the next 33 bytes of `m`, and `c` is the last 16 bytes of `m`.
   - The raw bytes of the remote party's ephemeral public key (`re`) are to be deserialized into a point on the curve using affine coordinates as encoded by the key's serialized composed format.
3. If `v` is an unrecognized handshake version, then the responder must abort the connection attempt.
4. `h = SHA-256(h || re.serializeCompressed())`

   The responder accumulates the initiator's ephemeral key into the authenticating handshake digest.

5. `es = ECDH(s.priv, re)`

   The responder performs an ECDH between its static private key and the initiator's ephemeral public key.

6. `ck, temp_k1 = HKDF(ck, es)`

   A new temporary encryption key is generated, which will shortly be used to check the authenticating MAC.

7. `p = decryptWithAD(temp_k1, 0, h, c)`

   If the MAC check in this operation fails, then the initiator does *not* know the responder's static public key. If this is the case, then the responder must terminate the connection without any further messages.

8. `h = SHA-256(h || c)`

   The received ciphertext is mixed into the handshake digest. This step serves to ensure the payload wasn't modified by a MITM.

Act Two

```
    <- e, ee
```

Act Two is sent from the responder to the initiator. Act Two will *only* take place if Act One was successful. Act One was successful if the responder was able to properly decrypt and check the MAC of the tag sent at the end of Act One.

The handshake is *exactly* 50 bytes: 1 byte for the handshake version, 33 bytes for the compressed ephemeral public key of the responder, and 16 bytes for the `poly1305` tag.

Sender actions:

1. `e = generateKey()`
2. `h = SHA-256(h || e.pub.serializeCompressed())`

   The newly generated ephemeral key is accumulated into the running handshake digest.

3. `ee = ECDH(e.priv, re)`

   Where `re` is the ephemeral key of the initiator, which was received during Act One.

4. `ck, temp_k2 = HKDF(ck, ee)`

   A new temporary encryption key is generated, which is used to generate the authenticating MAC.

5. `c = encryptWithAD(temp_k2, 0, h, zero)`

   Where `zero` is a zero-length plain text.

6. `h = SHA-256(h || c)`

   Finally, the generated ciphertext is accumulated into the authenticating handshake digest.

7. Send `m = 0 || e.pub.serializeCompressed() || c` to the initiator over the network buffer.

Receiver actions:

1. Read *exactly* 50 bytes from the network buffer.
2. Parse the read message (`m`) into `v`, `re`, and `c`:

   Where `v` is the *first* byte of `m`, `re` is the next 33 bytes of `m`, and `c` is the last 16 bytes of `m`.

3. If `v` is an unrecognized handshake version, then the responder must abort the connection attempt.
4. `h = SHA-256(h || re.serializeCompressed())`
5. `ee = ECDH(e.priv, re)`

   Where `re` is the responder's ephemeral public key.

The raw bytes of the remote party's ephemeral public key (`re`) are to be deserialized into a point on the curve using affine coordinates as encoded by the key's serialized composed format.

6. `ck, temp_k2 = HKDF(ck, ee)`

   A new temporary encryption key is generated, which is used to generate the authenticating MAC.

7. `p = decryptWithAD(temp_k2, 0, h, c)`

   If the MAC check in this operation fails, then the initiator must terminate the connection without any further messages.

8. `h = SHA-256(h || c)`

   The received ciphertext is mixed into the handshake digest. This step serves to ensure the payload wasn't modified by a MITM.

Act Three

```
  -> s, se
```

Act Three is the final phase in the authenticated key agreement described in this section. This act is sent from the initiator to the responder as a concluding step. Act Three is executed *if and only if* Act Two was successful. During Act Three, the initiator transports its static public key to the responder encrypted with *strong* forward secrecy, using the accumulated `HKDF` derived secret key at this point of the handshake.

The handshake is *exactly* 66 bytes: 1 byte for the handshake version, 33 bytes for the static public key encrypted with the `ChaCha20` stream cipher, 16 bytes for the encrypted public key's tag generated via the AEAD construction, and 16 bytes for a final authenticating tag.

Sender actions:

1. `c = encryptWithAD(temp_k2, 1, h, s.pub.serializeCompressed())`

   Where `s` is the static public key of the initiator.

2. `h = SHA-256(h || c)`
3. `se = ECDH(s.priv, re)`

   Where `re` is the ephemeral public key of the responder.

4. `ck, temp_k3 = HKDF(ck, se)`

   The final intermediate shared secret is mixed into the running chaining key.

5. `t = encryptWithAD(temp_k3, 0, h, zero)`

   Where `zero` is a zero-length plain text.

6. `sk, rk = HKDF(ck, zero)`

   Where `zero` is a zero-length plain text, `sk` is the key to be used by the initiator to encrypt messages to the responder, and `rk` is the key to be used by the initiator to decrypt messages sent by the responder.

   The final encryption keys, to be used for sending and receiving messages for the duration of the session, are generated.

7. `rn = 0, sn = 0`

   The sending and receiving nonces are initialized to 0.

8. Send `m = 0 || c || t` over the network buffer.

Receiver actions:

1. Read *exactly* 66 bytes from the network buffer.
2. Parse the read message (`m`) into `v`, `c`, and `t`:

   Where `v` is the *first* byte of `m`, `c` is the next 49 bytes of `m`, and `t` is the last 16 bytes of `m`.

3. If `v` is an unrecognized handshake version, then the responder must abort the connection attempt.
4. `rs = decryptWithAD(temp_k2, 1, h, c)`

   At this point, the responder has recovered the static public key of the initiator.

5. `h = SHA-256(h || c)`
6. `se = ECDH(e.priv, rs)`

   Where `e` is the responder's original ephemeral key.

7. `ck, temp_k3 = HKDF(ck, se)`
8. `p = decryptWithAD(temp_k3, 0, h, t)`

   If the MAC check in this operation fails, then the responder must terminate the connection without any further messages.

9. `rk, sk = HKDF(ck, zero)`

   Where `zero` is a zero-length plain text, `rk` is the key to be used by the responder to decrypt the messages sent by the initiator, and `sk` is the key to be used by the responder to encrypt messages to the initiator.

   The final encryption keys, to be used for sending and receiving messages for the duration of the session, are generated.

10. `rn = 0, sn = 0`

    The sending and receiving nonces are initialized to 0.

Transport message encryption

At the conclusion of Act Three, both sides have derived the encryption keys, which will be used to encrypt and decrypt messages for the remainder of the session.

The actual Lightning Protocol messages are encapsulated within AEAD ciphertexts. Each message is prefixed with another AEAD ciphertext, which encodes the total length of the following Lightning message (not including its MAC).

The *maximum* size of *any* Lightning message must not exceed 65,535 bytes. A maximum size of 65,535 simplifies testing, makes memory management easier, and helps mitigate memory-exhaustion attacks.

To make traffic analysis more difficult, the length prefix for all encrypted Lightning messages is also encrypted. Additionally a 16-byte `Poly-1305` tag is added to the encrypted length prefix to ensure that the packet length hasn't been modified when in flight and also to avoid creating a decryption oracle.

The structure of packets on the wire resembles the diagram in Encrypted packet structure.

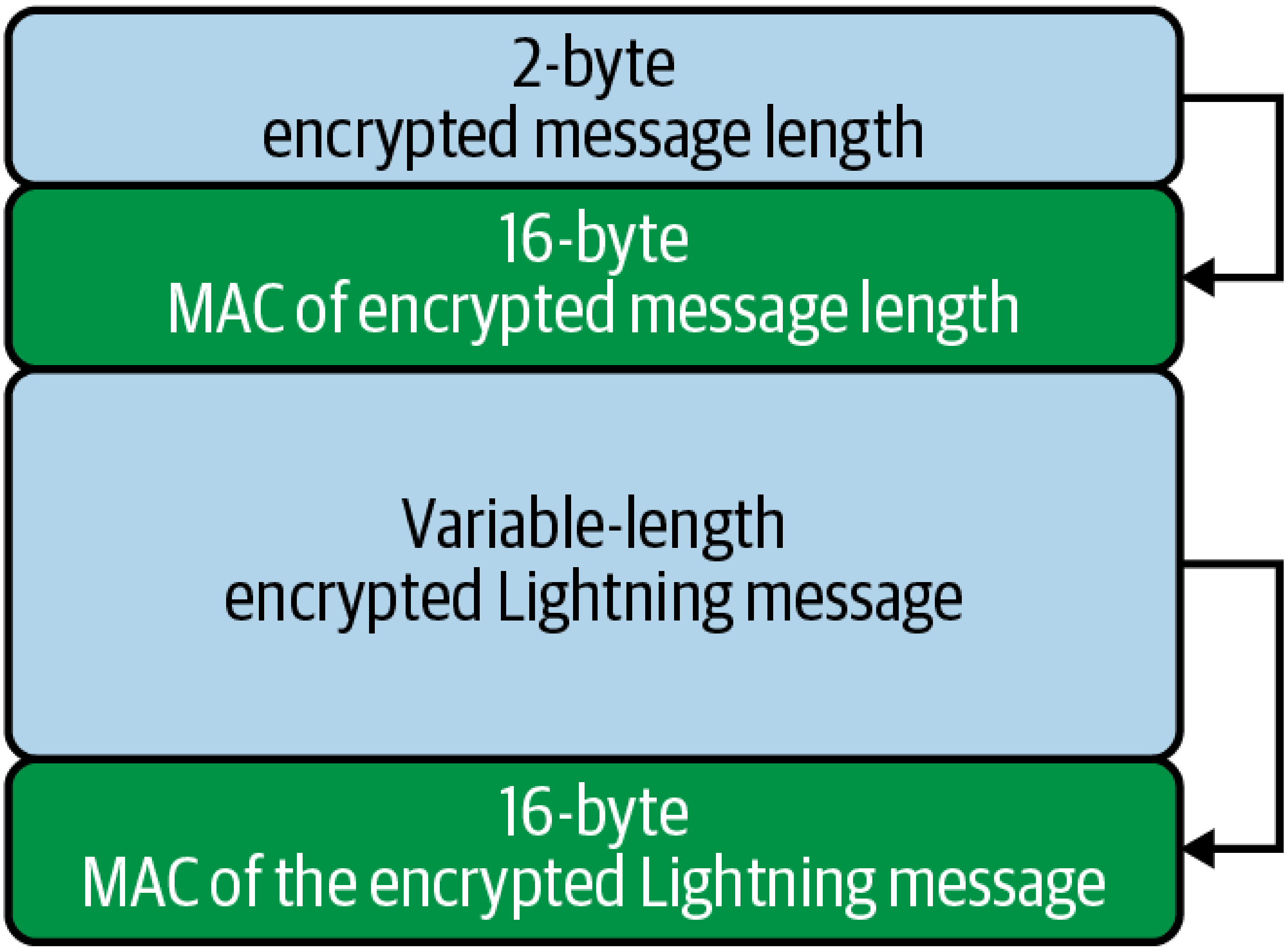

2-byte
encrypted message length
16-byte
MAC of encrypted message length
Variable-length
encrypted Lightning message
16-byte
MAC of the encrypted Lightning message

Figure 105. Encrypted packet structure

The prefixed message length is encoded as a 2-byte big-endian integer, for a total maximum packet length of 2 + 16 + 65,535 + 16 = 65,569 bytes.

Encrypting and sending messages

To encrypt and send a Lightning message (`m`) to the network stream, given a sending key (`sk`) and a nonce (`sn`), the following steps are completed:

1. Let `l = len(m)`.

   Where `len` obtains the length in bytes of the Lightning message.

2. Serialize `l` into 2 bytes encoded as a big-endian integer.
3. Encrypt `l` (using `ChaChaPoly-1305`, `sn`, and `sk`), to obtain `lc` (18 bytes).
   - The nonce `sn` is encoded as a 96-bit little-endian number. As the decoded nonce is 64 bits, the 96-bit nonce is encoded as 32 bits of leading zeros followed by a 64-bit value.
   - The nonce `sn` must be incremented after this step.
   - A zero-length byte slice is to be passed as the AD (associated data).
4. Finally, encrypt the message itself (`m`) using the same procedure used to encrypt the length prefix. Let this encrypted ciphertext be known as `c`.

   The nonce `sn` must be incremented after this step.

5. Send `lc || c` over the network buffer.

Receiving and decrypting messages

To decrypt the *next* message in the network stream, the following steps are completed:

1. Read *exactly* 18 bytes from the network buffer.
2. Let the encrypted length prefix be known as `lc`.
3. Decrypt `lc` (using `ChaCha20-Poly1305`, `rn`, and `rk`) to obtain the size of the encrypted packet `l`.
   - A zero-length byte slice is to be passed as the AD (associated data).
   - The nonce `rn` must be incremented after this step.
4. Read *exactly* `l + 16` bytes from the network buffer, and let the bytes be known as `c`.
5. Decrypt `c` (using `ChaCha20-Poly1305`, `rn`, and `rk`) to obtain decrypted plain-text packet `p`.

   The nonce `rn` must be incremented after this step.

Lightning message key rotation

Changing keys regularly and forgetting previous keys is useful to prevent the decryption of old messages, in the case of later key leakage (i.e., backward secrecy).

Key rotation is performed for *each* key (`sk` and `rk`) *individually*. A key is to be rotated after a party encrypts or decrypts 1,000 times with it (i.e., every 500 messages). This can be properly accounted for by rotating the key once the nonce dedicated to it exceeds 1,000.

Key rotation for a key `k` is performed according to the following steps:

1. Let `ck` be the chaining key obtained at the end of Act Three.
2. `ck', k' = HKDF(ck, k)`
3. Reset the nonce for the key to `n = 0`.
4. `k = k'`
5. `ck = ck'`

Conclusion

Lightning's underlying transport encryption is based on the Noise Protocol and offers strong security guarantees of privacy, authenticity, and integrity for all communications between Lightning peers.

Unlike Bitcoin where peers often communicate "in the clear" (without encryption), all Lightning communications are encrypted peer-to-peer. In addition to transport encryption (peer-to-peer), in the Lightning Network, payments are *also* encrypted into onion packets (hop-to-hop) and payment details are sent out-of-band between the sender and recipient (end-to-end). The combination of all these security mechanisms is cumulative and provides a layered defense against de-anonymization, man-in-the-middle attacks, and network surveillance.

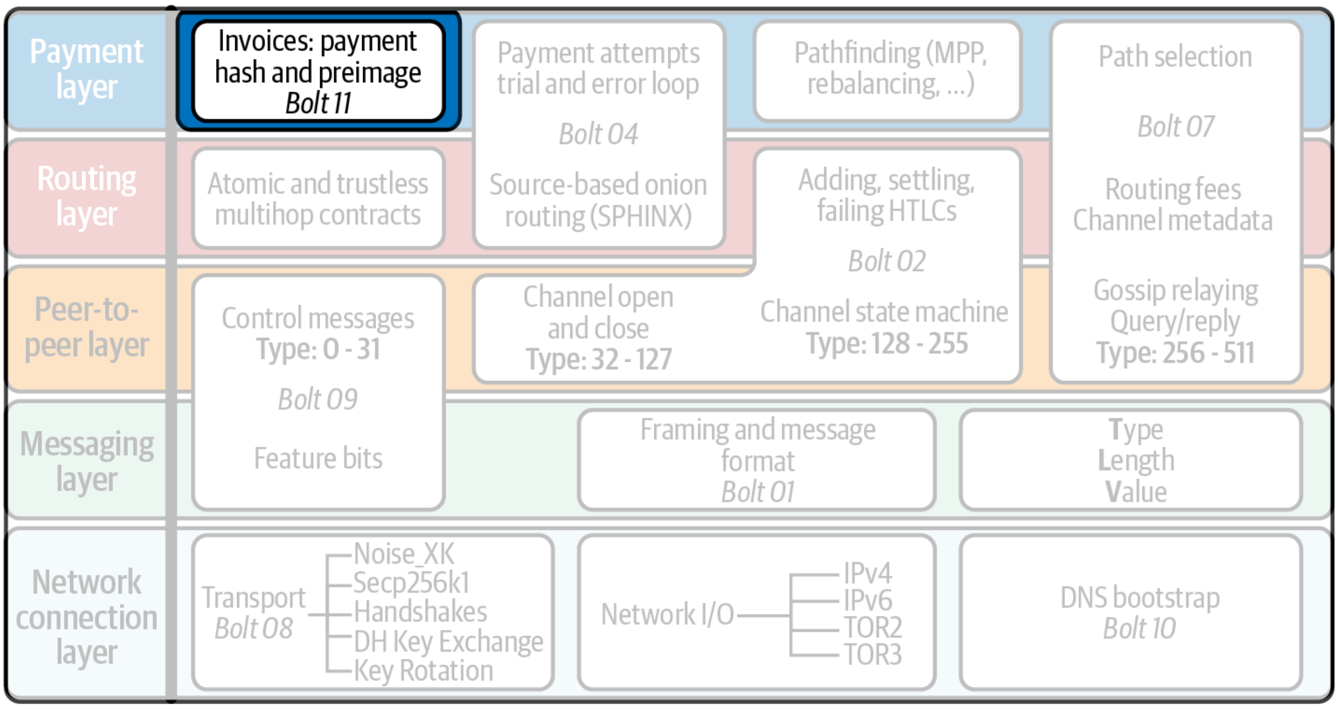
Of course, no security is perfect and we will see in Security and Privacy of the Lightning Network that these properties can be degraded and attacked. However, the Lightning Network significantly improves upon the privacy of Bitcoin.

Lightning Payment Requests

In this chapter we will look at *Lightning payment requests*, or as they are more commonly known, *Lightning invoices*.

Invoices in the Lightning Protocol Suite

Payment requests, aka *invoices*, are part of the payment layer and are shown in the upper left of Payment requests in the Lightning protocol suite.



Payment layer
Routing layer
Peer-to-peer layer
Messaging layer
Network connection layer
Invoices: payment hash and preimage Bolt 11
Payment attempts trial and error loop Bolt 04 Source-based onion routing (SPHINX)
Pathfinding (MPP, rebalancing, ...)
Path selection Bolt 07 Routing fees Channel metadata Gossip relaying Query/reply Type: 256 - 511
Atomic and trustless multihop contracts
Adding, settling, failing HTLCs Bolt 02 Channel state machine Type: 128 - 255
Control messages Type: 0 - 31 Bolt 09 Feature bits
Channel open and close Type: 32 - 127
Framing and message format Bolt 01
Type Length Value
Transport Bolt 08
Noise_XK
Secp256k1
Handshakes
DH Key Exchange
Key Rotation
Network I/O
IPv4
IPv6
TOR2
TOR3
DNS bootstrap Bolt 10

Figure 106. Payment requests in the Lightning protocol suite

Introduction

As we've learned throughout the book, minimally two pieces of data are required to complete a Lightning payment: a payment hash and a destination. As SHA-256 is used in the Lightning Network to implement HTLCs, this information requires 32 bytes to communicate. Destinations, on the other hand, are simply the `secp256k1` public key of the node that wishes to receive a payment. The purpose of a payment request in the context of the Lightning Network is to communicate these two pieces of information from sender to receiver. The QR-code-friendly format for communicating the information required to complete a payment from receiver to sender is described in BOLT #11: Invoice Protocol for Lightning Payments. In practice, more than just the payment hash and destination are communicated in a payment request to make the encoding more fully featured.

Lightning Payment Requests versus Bitcoin Addresses

A commonly asked question when people first encounter a Lightning Payment request is: why can't a normal static address format be used instead?

To answer this question, you must first internalize how Lightning differs from base layer Bitcoin as a payment method. Compared to a Bitcoin address which may be used to make a potentially unbounded number of payments (though reusing a Bitcoin address may degrade one’s privacy), a Lightning payment request should only ever be used *once*. This is due to the fact that sending a payment to a Bitcoin address essentially uses a public key cryptosystem to "encode" the payment in a manner that only the true "owner" of that Bitcoin address can redeem it.

In contrast, to complete a Lightning payment, the recipient must reveal a "secret" to the entire payment route, including the sender. This can be interpreted as usage of a kind of domain-specific symmetric cryptography because the payment preimage is for practical purposes a nonce (number only used once). If the sender attempts to make another payment using that identical payment hash, then they risk losing funds because the payment may not actually be delivered to the destination. It’s safe to assume that after a preimage has been revealed, all nodes in the path will keep it around forever, then rather than forward the HTLC to collect a routing fee if the payment is completed, they can simply settle the payment at that instance and gain the entire payment amount in return. As a result, it’s unsafe to ever use a payment request more than once.

New variants of the original Lightning payment request exist that allow the sender to reuse them as many times as they want. These variants flip the normal payment flow as the sender transmits a preimage within the encrypted onion payload to the receiver, who is the only one that is able to decrypt it and settle the payment. Alternatively, assuming a mechanism that allows a sender to typically request a new payment request from the receiver, then an interactive protocol can be used to allow a degree of payment request reuse.

BOLT #11: Lightning Payment Request Serialization and Interpretation

In this section, we’ll describe the mechanism used to encode the set of information required to complete a payment on the Lightning Network. As mentioned earlier, the payment hash and destination is the minimum amount of information required to complete a payment. However, in practice, more information such as timelock information, payment request expiration, and possibly an on-chain fallback address are also communicated. The full specification document is BOLT #11: Invoice Protocol for Lightning Payments.

Payment Request Encoding in Practice

First, let’s examine what a real payment request looks like in practice. The following is a valid payment request that could have been used to complete a payment on the mainnet Lightning Network at the time it was created:

```
lnbc2500u1pvjluezpp5qqqsyqcyq5rqwzqfqqqsyqcyq5rqwzqfqqqsyqcyq5rqwzqfqypqdq5xysxxatsyp3
k7enxv4jsxqzpuaztrnwngzn3kdzw5hydlzf03qdgm2hdq27cqv3agm2awhz5se903vruatfhq77w3ls4evs3c
h9zw97j25emudupq63nyw24cg27h2rspfj9srp
```

The Human-Readable Prefix

Looking at the string, we can tease out a portion that we can parse with our eyes, while the rest of it just looks like a random set of strings. The part that is somewhat parsable by a human is referred to as the *human-readable prefix*. It allows a human to quickly extract some relevant information from a payment request at a glance. In this case, we can see that this payment is for the mainnet instance of the Lightning Network (`lnbc`), and is requesting 2,500 uBTC (microbitcoin), or 25,000,000 satoshis. The latter potion is referred to as the data portion and uses an extensible format to encode the information required to complete a payment.

Each version of instance of the Lightning Network (mainnet, testnet, etc.) has its own human-readable prefix (see BOLT #11 network prefixes). This allows client software and also humans to quickly determine if a payment request can be satisfied by their node or not.

Table 8. BOLT #11 network prefixes

Network	BOLT #11 prefix
mainnet	`lnbc`
testnet	`lntb`
simnet/regtest	`lnbcrt`

The first portion of the human-readable prefix is a *compact* expression of the amount of the payment request. The compact amount is encoded in two parts. First, an integer is used as the *base* amount. This is then followed by a multiplier that allows us to specify distinct order of magnitude increases offset by the base amount. If we return to our initial example, then we can take the `2500u` portion and decrease it by a factor of 1,000 to instead use `2500m` or (2,500 mBTC). As a rule of thumb, to ascertain the amount of an invoice at a glance, take the base factor and multiply it by the multiplier.

A full list of the currently defined multipliers is given in BOLT #11 amount multipliers.

Table 9. BOLT #11 amount multipliers

Multiplier	Bitcoin unit	Multiplication factor
`m`	milli	0.001
`u`	micro	0.000001
`n`	nano	0.000000001
`p`	pico	0.000000000001

bech32 and the Data Segment

If the "unreadable" portion looks familiar, then that's because it uses the very same encoding scheme as SegWit compatible Bitcoin addresses use today, namely bech32. Describing the bech32 encoding scheme is outside the scope of this chapter. In brief, it's a sophisticated way to encode short strings that has very good error correction as well as detection properties.

The data portion can be separated into three sections:

- The timestamp
- Zero or more tagged key-value pairs
- The signature of the entire invoice

The timestamp is expressed in seconds since the year 1970, or the Unix Epoch. This timestamp allows the sender to gauge how old the invoice is, and as we'll see later, allows the receiver to force an invoice to only be valid for a period of time if they wish.

Similar to the TLV format we learned about in Type-Length-Value (TLV) Format, the BOLT #11 invoice format uses a series of extensible key-value pairs to encode information needed to satisfy a payment. Because key-value pairs are used, it's easy to add new values in the future if a new payment type or additional requirement/functionality is introduced.

Finally, a signature is included that covers the entire invoice signed by the destination of the payment. This signature allows the sender to verify that the payment request was indeed created by the destination of the payment. Unlike Bitcoin payment requests which aren't signed, this allows us to ensure that a particular entity signed the payment request. The signature itself is encoded using a recovery ID, which allows a more compact signature to be used that allows public key extraction. When verifying the signature, the recovery ID extracts the public key, then verifies that against the public key included in the invoice.

Tagged invoice fields

The tagged invoice fields are encoded in the main body of the invoice. These fields represent different key-value pairs that express either additional information that may help complete the payment or information which is *required* to complete the payment. Because a slight variant of bech32 is utilized, each of these fields are actually in the "base 5" domain.

A given tag field is comprised of three components:

- The `type` of the field (5 bits)
- The `length` of the data of the field (10 bits)
- The `data` itself, which is `length * 5 bytes` in size

A full list of all the currently defined tagged fields is given in BOLT #11 tagged invoice fields.

Table 10. BOLT #11 tagged invoice fields

Field tag	**Data length**	**Usage**
`p`	`52`	The SHA-256 payment hash.
`s`	`52`	A 256-bit secret that increases the end-to-end privacy of a payment by mitigating probing by intermediate nodes.
`d`	Variable	The description, a short UTF-8 string of the purpose of the payment.
`n`	`53`	The public key of the destination node.

Field tag	Data length	Usage
`h`	`52`	A hash that represents a description of the payment itself. This can be used to commit to a description that's over 639 bytes in length.
`x`	Variable	The expiry time, in seconds, of the payment. The default is 1 hour (3,600) if not specified.
`c`	Variable	The `min_cltv_expiry` to use for the final hop in the route. The default is 9 if not specified.
`f`	Variable	A fallback on-chain address to be used to complete the payment if the payment cannot be completed over the Lightning Network.
`r`	Variable	One or more entries that allow a receiver to give the sender additional ephemeral edges to complete the payment.
`9`	Variable	A set of 5-bit values that contain the feature bits that are required in order to complete the payment.

The elements contained in the field `r` are commonly referred to as *routing hints*. They allow the receiver to communicate an extra set of edges that may help the sender complete their payment. The hints are usually used when the receiver has some/all private channels, and they wish to guide the sender into this "unmapped" portion of the channel graph. A routing hint encodes effectively the same information that a normal `channel_update` message does. The update is itself packed into a single value with the following fields:

- The `pubkey` of the outgoing node in the edge (264 bits)
- The `short_channel_id` of the "virtual" edge (64 bits)
- The base fee (`fee_base_msat`) of the edge (32 bits)
- The proportional fee (`fee_proportional_millionths`) (32 bits)
- The CLTV expiry delta (`cltv_expiry_delta`) (16 bits)

The final portion of the data segment is the set of feature bits that communicate to the sender the functionality needed to complete a payment. As an example, if a new payment type is added in the future that isn't backward compatible with the original payment type, then the receiver can set a *required* feature bit to communicate that the payer needs to understand that feature to complete the payment.

Conclusion

As we have seen, invoices are a lot more than just a request for an amount. They contain critical information about *how* to make the payment, such as routing hints, the destination node's public key, ephemeral keys to increase security, and much more.

Security and Privacy of the Lightning Network

In this chapter, we look at some of the most important issues related to the security and privacy of the Lightning Network. First, we'll consider privacy, what it means, how to evaluate it, and some things you can do to protect your own privacy while using the Lightning Network. Then we'll explore some common attacks and mitigation techniques.

Why Is Privacy Important?

The key value proposition of cryptocurrency is censorship resistant money. Bitcoin offers participants the possibility of storing and transferring their wealth without interference by governments, banks, or corporations. The Lightning Network continues this mission.

Unlike trivial scaling solutions like custodial Bitcoin banks, the Lightning Network aims to scale Bitcoin without compromising on self custody, which should lead to greater censorship resistance in the Bitcoin ecosystem. However, the Lightning Network operates under a different security model, which introduces novel security and privacy challenges.

Definitions of Privacy

The question, "Is Lightning private?" has no direct answer. Privacy is a complex topic; it is often difficult to precisely define what we mean by privacy, particularly if you are not a privacy researcher. Fortunately, privacy researchers use processes to analyze and evaluate the privacy characteristics of systems, and we can use them too! Let's look at how a security researcher might seek to answer the question, "Is Lightning private?" in two general steps.

First, a privacy researcher would define a *security model* that specifies what an adversary is capable of and aims to achieve. Then, they would describe the relevant properties of the system and check whether it conforms to the requirements.

Process to Evaluate Privacy

A security model is based on a set of underlying *security assumptions*. In cryptographic systems, these assumptions are often centered around the mathematical properties of the cryptographic primitives, such as ciphers, signatures, and hash functions. The security assumptions of the Lightning Network are that the ECDSA signatures, SHA-256 hash function, and other cryptographic functions used in the protocol behave within their security definitions. For example, we assume that it is practically impossible to find a preimage (and second preimage) of a hash function. This allows the Lightning Network to rely on the HTLC mechanism (which uses the preimage of a hash function) for the atomicity of multihop payments: nobody except the final recipient can reveal the payment secret and resolve the HTLC. We also assume a degree of connectivity in the network, namely that Lightning channels form a connected graph. Therefore, it is possible to find a path from any sender to any receiver. Finally, we assume network messages are propagated within certain timeouts.

Now that we’ve identified some of our underlying assumptions, let’s consider some possible adversaries.

Here are some possible models of adversaries in the Lightning Network. An "honest-but-curious" forwarding node can observe payment amounts, the immediately preceding and following nodes, and the graph of announced channels with their capacities. A very well-connected node can do the same but to a larger extent. For example, consider the developers of a popular wallet who maintain a node that their users connect to by default. This node would be responsible for routing a large share of payments to and from the users of that wallet. What if multiple nodes are under adversarial control? If two colluding nodes happen to be on the same payment path, they would understand that they are forwarding HTLCs belonging to the same payment because HTLCs have the same payment hash.

NOTE Multipart payments (see Multipart Payments (MPP)) enable users to obfuscate their payment amounts given their nonuniform split sizes.

What may be the goals of a Lightning attacker? Information security is often described in terms of three main properties: confidentiality, integrity, and availability.

Confidentiality
The information only gets to intended recipients.

Integrity
The information does not get altered in transit.

Availability
The system is functioning most of the time.

The important properties of the Lightning Network are mostly centered around confidentiality and availability. Some of the most important properties to protect include:

- Only the sender and the receiver know the payment amount.
- No one can link senders and receivers.
- An honest user cannot be blocked from sending and receiving payments.

For each privacy goal and security model, there is a certain probability that an attacker succeeds. This probability depends on various factors, such as the size and structure of the network. Other things being equal, it is generally easier to successfully attack a small network rather than a large one. Similarly, the more centralized the network is, the more capable an attacker can be if "central" nodes are under their control. Of course, the term centralization must be defined precisely to build security models around it, and there are many possible definitions of how centralized a network is. Finally, as a payment network, the Lightning Network depends on economic stimuli. The size and structure of fees affect the routing algorithm, and therefore can either aid the attacker by forwarding most payments through their nodes or prevent this from happening.

Anonymity Set

What does it mean to de-anonymize someone? In simple terms, de-anonymization implies linking some action with a person’s real-world identity, such as their name or physical address. In privacy research, the notion of de-anonymization is more nuanced. First, we are not necessarily talking about names and addresses. Discovering someone’s IP address or telephone number may also be considered de-anonymization. A piece of information that allows linking a user’s action to their previous actions is referred to as *identity*. Second, de-anonymization is not binary; a user is neither fully anonymous nor completely de-anonymized. Instead, privacy research looks at anonymity compared to the anonymity set.

The *anonymity set* is a central notion in privacy research. It refers to the set of identities such that, from an attacker’s viewpoint, a given action could correspond to anyone in the set. Consider a real-life example. Imagine you meet a person on a city street. What is their anonymity set from your point of view? If you don’t know them personally, and without any additional information, their anonymity set roughly equals the city’s population, including travelers. If you additionally consider their appearance, you may be able to roughly estimate their age and exclude the city residents who are obviously older or younger than the person in question from the anonymity set. Furthermore, if you notice that the person walks into the office of Company X using an electronic badge, the anonymity set shrinks to the number of Company X’s employees and visitors. Finally, you may notice the license number of the car they used to arrive at the place. If you are a casual observer, this doesn’t give you much. However, if you are a city official and have access to the database that matches license plate numbers with names, you can narrow down the anonymity set to just a few people: the car owner and any close friends and relatives that may have borrowed the car.

This example illustrates a few important points. First, every bit of information may bring the adversary closer to their goal. It may not be necessary to shrink the anonymity set to the size of one. For instance, if the adversary plans a targeted denial-of-service (DoS) attack and can take down 100 servers, the anonymity set of 100 suffices. Second, the adversary can cross-correlate information from different sources. Even if a privacy leak looks relatively benign, we never know what it can achieve in combination with other data sources. Finally, especially in cryptographic settings, the attacker always has the "last resort" of a brute-force search. Cryptographic primitives are designed so that it is practically impossible to guess a secret such as a private key. Nevertheless, each bit of information brings the adversary closer to this goal, and at some point, it becomes attainable.

In terms of Lightning, de-anonymizing generally means deriving a correspondence between payments and users identified by node IDs. Each payment may be assigned a sender anonymity set and a receiver anonymity set. Ideally, the anonymity set consists of all the users of the network. This assures that the attacker has no information whatsoever. However, the real network leaks information that allows an attacker to narrow down the search. The smaller the anonymity set, the higher the chance of successful de-anonymization.

Differences Between the Lightning Network and Bitcoin in Terms of Privacy

While it’s true that transactions on the Bitcoin network do not associate real-world identities with Bitcoin addresses, all transactions are broadcast in cleartext and can be analyzed. Multiple companies have been established to de-anonymize users of Bitcoin and other cryptocurrencies.

At first glance, Lightning provides better privacy than Bitcoin because Lightning payments are not broadcast to the whole network. While this improves the privacy baseline, other properties of the Lightning protocol may make anonymous payments more challenging. For instance, larger payments may have fewer routing options. This may allow an adversary who controls well-capitalized nodes to route most large payments and discover payment amounts and probably other details. Over time, as the Lightning Network grows, this may become less of a problem.

Another relevant difference between Lightning and Bitcoin is that Lightning nodes maintain a permanent identity, whereas Bitcoin nodes do not. A sophisticated Bitcoin user can easily switch nodes used to receive blockchain data and broadcast transactions. A Lightning user, on the contrary, sends and receives payments through the nodes they have used to open their payment channels. Moreover, the Lightning protocol assumes that routing nodes announce their IP address in addition to their node ID. This creates a permanent link between node IDs and IP addresses, which may be dangerous, considering that an IP address is often an intermediary step in anonymity attacks linked to the user's physical location and, in most cases, real-world identity. It is possible to use Lightning over Tor, but many nodes do not use this functionality, as can be seen from statistics collected from node announcements.

A Lightning user, when sending a payment, has its neighbors in its anonymity set. Specifically, a routing node only knows the immediately preceding and following nodes. The routing node does not know whether its immediate neighbors in the payment route are the ultimate sender or receiver. Therefore, the anonymity set of a node in Lightning roughly equals its neighbors (see The anonymity set of Alice and Bob constitutes their neighbors).

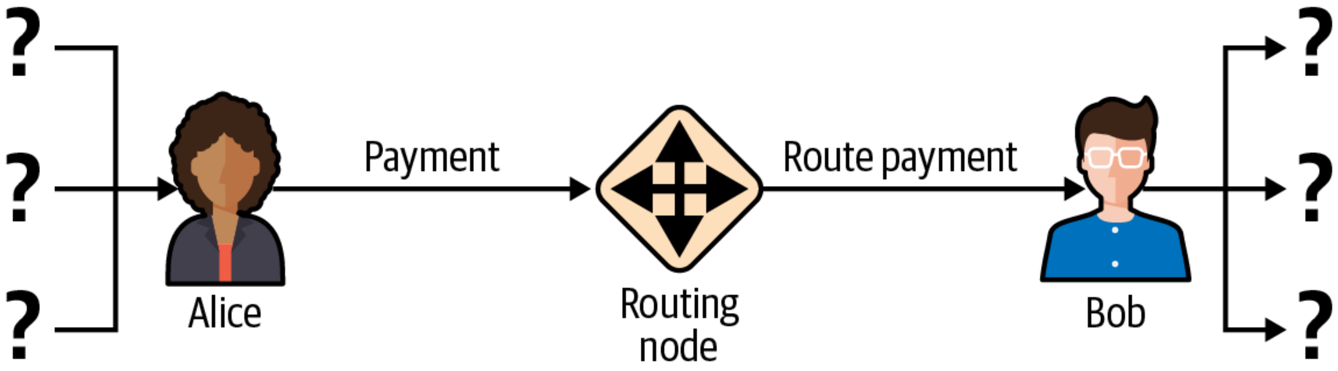


Figure 107. The anonymity set of Alice and Bob constitutes their neighbors

Similar logic applies to payment receivers. Many users open only a handful of payment channels, therefore limiting their anonymity sets. Moreover, in Lightning, the anonymity set is static or at least slowly changing. In contrast, one can achieve significantly larger anonymity sets in on-chain CoinJoin transactions. CoinJoin transactions with anonymity sets larger than 50 are quite frequent. Typically, the anonymity sets in a CoinJoin transaction correspond to a dynamically changing set of users.

Finally, Lightning users can also be denied service, having their channels blocked or depleted by an attacker. Forwarding payments requires capital—a scarce resource!—to be temporarily blocked in HTLCs along the route. An attacker may send many payments but fail to finalize them, occupying honest users' capital for long periods. This attack vector is not present (or at least not as obvious) in Bitcoin.

In summary, while some aspects of the Lightning Network's architecture suggest that it is a step forward in terms of privacy compared to Bitcoin, other properties of the protocol may make attacks on privacy easier. Thorough research is needed to evaluate what privacy guarantees the Lightning Network provides and improve the state of affairs.

The issues discussed in this part of the chapter summarize research available in mid-2021. However, this area of research and development is growing quickly. We are happy to report that the authors are aware of multiple research teams currently working on Lightning privacy.

Now let's review some of the attacks on LN privacy that have been described in academic literature.

Attacks on Lightning

Recent research describes various ways in which the security and privacy of the Lightning Network may be compromised.

Observing Payment Amounts

One of the goals for a privacy-preserving payment system is to hide the payment amount from uninvolved parties. The Lightning Network is an improvement over Layer 1 in this regard. While Bitcoin transactions are broadcast in cleartext and can be observed by anyone, Lightning payments only travel through a few nodes along the payment path. However, intermediary nodes do see the payment amount, although this payment amount might not correspond to the actual total payment amount (see Multipart Payments (MPP)). This is necessary to create a new HTLC at every hop. The availability of payment amounts to intermediary nodes do not present an immediate threat. However, an *honest-but-curious* intermediary node may use it as a part of a larger attack.

Linking Senders and Receivers

An attacker might be interested in learning the sender and/or the receiver of a payment to reveal certain economic relationships. This breach of privacy could harm censorship resistance, as an intermediary node could censor payments to or from certain receivers or senders. Ideally, linking senders to receivers should not be possible to anyone other than the sender and the receiver.

In the following sections, we will consider two types of adversaries: the off-path adversary and the on-path adversary. An off-path adversary tries to assess the sender and the receiver of a payment without participating in the payment routing process. An on-path adversary can leverage any information it might gain by routing the payment of interest.

First, consider the *off-path adversary*. In the first step of this attack scenario, a potent off-path adversary deduces the individual balances in each payment channel via probing (described in a subsequent section) and forms a network snapshot at time t_1. For simplicity's sake, let's make t_1 equal 12:05. It then probes the network again at sometime later at time t_2, which we'll make 12:10. The attacker would then compare the snapshots at 12:10 and 12:05 and use the differences between the two snapshots to infer information about payments that took place by looking at paths that have changed. In the simplest case, if only one payment occurred between 12:10 and 12:05, the adversary would observe a single path where the balances have changed by the same amounts.

Thus, the adversary learns almost everything about this payment: the sender, the recipient, and the amount. If multiple payment paths overlap, the adversary needs to apply heuristics to identify such overlap and separate the payments.

Now, we turn our attention to an *on-path adversary*. Such an adversary might seem convoluted. However, in June 2020, researchers noted that the single most central node observed close to 50% of all LN payments, while the four most central nodes observed an average of 72% payments. These findings emphasize the relevance of the on-path attacker model. Even though intermediaries in a payment path only learn their successor and predecessor, there are several leakages that a malicious or honest-but-curious intermediary might use to infer the sender and the receiver.

The on-path adversary can observe the amount of any routed payment as well as timelock deltas (see Onion Routing). Hence, the adversary can exclude any nodes from the sender's or the receiver's anonymity set with capacities lower than the routed amount. Therefore, we observe a trade-off between privacy and payment amounts. Typically, the larger the payment amount is, the smaller the anonymity sets are. We note that this leakage could be minimized with multipart payments or with large capacity payment channels. Similarly, payment channels with small timelock deltas could be excluded from a payment path. More precisely, a payment channel cannot pertain to a payment if the remaining time the payment might be locked for is larger than what the forwarding node would be willing to accept. This leakage could be evicted by adhering to the so-called shadow routes.

One of the most subtle and yet powerful leakages an on-path adversary can foster is the timing analysis. An on-path adversary can keep a log for every routed payment, along with the amount of time it takes for a node to respond to an HTLC request. Before starting the attack, the attacker learns every node's latency characteristics in the Lightning Network by sending them requests. Naturally, this can aid in establishing the adversary's precise position in the payment path. Even more, as it was recently shown, an attacker can successfully determine the sender and the receiver of a payment from a set of possible senders and receivers using time-based estimators.

Finally, it's important to recognize that unknown or unstudied leakages probably exist that could aid de-anonymizing attempts. For instance, because different Lightning wallets apply different routing algorithms, even knowing the applied routing algorithm could help exclude certain nodes from being a sender and/or receiver of a payment.

Revealing Channel Balances (Probing)

The balances of Lightning channels are supposed to be hidden for privacy and efficiency reasons. A Lightning node only knows the balances of its adjacent channels. The protocol provides no standard way to query the balance of a remote channel.

However, an attacker can reveal the balance of a remote channel in a *probing attack*. In information security, probing refers to the technique of sending requests to a targeted system and making conclusions about its private state based on the received responses.

Lightning channels are prone to probing. Recall that a standard Lightning payment starts with the receiver creating a random payment secret and sending its hash to the sender. Note that for the intermediary nodes, all hashes look random. There is no way to tell whether a hash corresponds to a real secret or was generated randomly.

The probing attack proceeds as follows. Say the attacker Mallory wants to reveal Alice's balance of a public channel between Alice and Bob. Suppose the total capacity of that channel is 1 million satoshis. Alice's balance could be anything from zero to 1 million satoshis (to be precise, the estimate is a bit tighter due to channel reserve, but we don't account for it here for simplicity). Mallory opens a channel with Alice with 1 million satoshis and sends 500,000 satoshis to Bob via Alice using a *random number* as the payment hash. Of course, this number does not correspond to any known payment secret. Therefore, the payment will fail. The question is: how exactly will it fail?

There are two scenarios. If Alice has more than 500,000 satoshis on her side of the channel to Bob, she forwards the payment. Bob decrypts the payment onion and realizes that the payment is intended for him. He looks up his local store of payment secrets and searches for the preimage that corresponds to the payment hash, but does not find one. Following the protocol, Bob returns the "unknown payment hash" error to Alice, who relays it back to Mallory. As a result, Mallory knows that the payment *could have succeeded* if the payment hash was real. Therefore, Mallory can update her estimation of Alice's balance from "between zero and 1 million" to "between 500,000 and 1 million." Another scenario happens if Alice's balance is lower than 500,000 satoshis. In that case, Alice is unable to forward the payment and returns the "insufficient balance" error to Mallory. Mallory updates her estimation from "between zero and 1 million" to "between zero and 500,000."

Note that in any case, Mallory's estimation becomes twice as precise after just one probing! She can continue probing, choosing the next probing amount such that it divides the current estimation interval in half. This well-known search technique is called *binary search*. With binary search, the number of probes is *logarithmic* in the desired precision. For example, to obtain Alice's balance in a channel of 1 million satoshis up to a single satoshi, Mallory would only have to perform $\log_2$ (1,000,000) about 20 probings. If one probing takes 3 seconds, one channel can be precisely probed in only about a minute!

Channel probing can be made even more efficient. In its simplest variant, Mallory directly connects to the channel she wants to probe. Is it possible to probe a channel without opening a channel to one of its endpoints? Imagine Mallory now wants to probe a channel between Bob and Charlie but doesn't want to open another channel, which requires paying on-chain fees and waiting for confirmations of the funding transactions. Instead, Mallory reuses her existing channel to Alice and sends a probe along the route Mallory → Alice → Bob → Charlie. Mallory can interpret the "unknown payment hash" error in the same way as before: the probe has reached the destination; therefore, all channels along the route have sufficient balances to forward it. But what if Mallory receives the "insufficient balance" error? Does it mean that the balance is insufficient between Alice and Bob or between Bob and Charlie?

In the current Lightning protocol, error messages report not only *which* error occurred but also *where* it happened. So, with more careful error handling, Mallory now knows which channel failed. If this is the target channel, she updates her estimates; if not, she chooses another route to the target channel. She even gets *additional* information about the balances of intermediary channels, on top of that of the target channel.

The probing attack can be further used to link senders and receivers, as described in the previous section.

At this point, you may ask: why does the Lightning Network do such a poor job at protecting its users' private data? Wouldn’t it be better to not reveal to the sender why and where the payment has failed? Indeed, this could be a potential countermeasure, but it has significant drawbacks. Lightning has to strike a careful balance between privacy and efficiency. Remember that regular nodes don’t know balance distributions in remote channels. Therefore, payments can (and often do) fail because of insufficient balance at an intermediary hop. Error messages allow the sender to exclude the failing channel from consideration when constructing another route. One popular Lightning wallet even performs probing internally to check whether a constructed route can really handle a payment.

There are other potential countermeasures against channel probing. First, it is hard for an attacker to target unannounced channels. Second, nodes that implement just-in-time (JIT) routing may be less prone to the attack. Finally, as multipart payments make the problem of insufficient capacity less severe, the protocol developers may consider hiding some of the error details without harming efficiency.

Denial of Service

When resources are made publicly available, there is a risk that attackers may attempt to make that resource unavailable by executing a denial-of-service (DoS) attack. Generally, this is achieved by the attacker bombarding a resource with requests, which are indistinguishable from legitimate queries. The attacks seldom result in the target suffering financial loss, aside from the opportunity cost of their service being down, and are merely intended to aggrieve the target.

Typical mitigations for DoS attacks require authentication for requests to separate legitimate users from malicious ones. These mitigations incur a trivial cost to regular users but will act as a sufficient deterrent to an attacker launching requests at scale. Anti-denial-of-service measures can be seen everywhere on the internet—websites apply rate limits to ensure that no one user can consume all of their server’s attention, film review sites require login authentication to keep angry r/prequelmemes (Reddit group) members at bay, and data services sell API keys to limit the number of queries.

DoS in bitcoin

In Bitcoin, the bandwidth that nodes use to relay transactions and the space that they avail to the network in the form of their mempool are publicly available resources. Any node on the network can consume bandwidth and mempool space by sending a valid transaction. If this transaction is mined in a valid block, they will pay transaction fees, which adds a cost to using these shared network resources.

In the past, the Bitcoin network faced an attempted DoS attack where attackers spammed the network with low-fee transactions. Many of these transactions were not selected by miners due to their low transaction fees, so the attackers could consume network resources without paying the fees. To address this issue, a minimum transaction relay fee that set a threshold fee that nodes require to propagate transactions was set. This measure largely ensured that the transactions that consume network resources will eventually pay their chain fees. The minimum relay fee is acceptable to regular users but would hurt attackers financially if they tried to spam the network. While some transactions may not make it into valid blocks within high-fee environments, these measures have largely been effective at deterring this type of spam.

DoS in Lightning

Similarly to Bitcoin, the Lightning Network charges fees for the use of its public resources, but in this case, the resources are public channels, and the fees come in the form of routing fees. The ability to route payments through nodes in exchange for fees provides the network with a large scalability benefit—nodes that are not directly connected can still transact—but it comes at the cost of exposing a public resource that must be protected against DoS attacks.

When a Lightning node forwards a payment on your behalf, it uses data and payment bandwidth to update its commitment transaction, and the amount of the payment is reserved in their channel balance until it is settled or failed. In successful payments, this is acceptable because the node is eventually paid out its fees. Failed payments do not incur fees in the current protocol. This allows nodes to costlessly route failed payments through any channels. This is great for legitimate users, who wouldn't like to pay for failed attempts, but also allows attackers to costlessly consume nodes' resources—much like the low-fee transactions on Bitcoin that never end up paying miner fees.

At the time of writing, a discussion is ongoing on the lightning-dev mailing list as to how best address this issue.

Known DoS attacks

There are two known DoS attacks on public LN channels which render a target channel, or a set of target channels, unusable. Both attacks involve routing payments through a public channel, then holding them until their timeout, thus maximizing the attack's duration. The requirement to fail payments to not pay fees is fairly simple to meet because malicious nodes can simply reroute payments to themselves. In the absence of fees for failed payments, the only cost to the attacker is the on-chain cost of opening a channel to dispatch these payments through, which can be trivial in low-fee environments.

Commitment Jamming

Lightning nodes update their shared state using asymmetric commitment transactions, on which HTLCs are added and removed to facilitate payments. Each party is limited to a total of 483 HTLCs in the commitment transaction at a time. A channel jamming attack allows an attacker to render a channel unusable by routing 483 payments through the target channel and holding them until they time out.

It should be noted that this limit was chosen in the specification to ensure that all the HTLCs can be swept in a single justice transaction. While this limit *may* be increased, transactions are still limited by the block size, so the number of slots available is likely to remain limited.

Channel Liquidity Lockup

A channel liquidity lockup attack is comparable to a channel jamming attack in that it routes payments through a channel and holds them so that the channel is unusable. Rather than locking up slots on the channel commitment, this attack routes large HTLCs through a target channel, consuming all the channel's available bandwidth. This attack's capital commitment is higher than the commitment jamming attack because the attacking node needs more funds to route failed payments through the target.

Cross-Layer De-Anonymization

Computer networks are often layered. Layering allows for separation of concerns and makes the whole system manageable. No one could design a website if it required understanding all the TCP/IP stack up to the physical encoding of bits in an optical cable. Every layer is supposed to provide the functionality to the layer above in a clean way. Ideally, the upper layer should perceive a lower layer as a black box. In reality, though, implementations are not ideal, and the details *leak* into the upper layer. This is the problem of leaky abstractions.

In the context of Lightning, the LN protocol relies on the Bitcoin protocol and the LN P2P network. Up to this point, we only considered the privacy guarantees offered by the Lightning Network in isolation. However, creating and closing payment channels are inherently performed on the Bitcoin blockchain. Consequently, for a complete analysis of the Lightning Network's privacy provisions, one needs to consider every layer of the technological stack users might interact with. Specifically, a de-anonymizing adversary can and will use off-chain and on-chain data to cluster or link LN nodes to corresponding Bitcoin addresses.

Attackers attempting to de-anonymize LN users may have various goals, in a cross-layer context:

- Cluster Bitcoin addresses owned by the same user (Layer 1). We call these Bitcoin entities.
- Cluster LN nodes owned by the same user (Layer 2).
- Unambiguously link sets of LN nodes to the sets of Bitcoin entities that control them.

There are several heuristics and usage patterns that allow an adversary to cluster Bitcoin addresses and LN nodes owned by the same LN users. Moreover, these clusters can be linked across layers using other powerful cross-layer linking heuristics. The last type of heuristics, cross-layer linking techniques, emphasizes the need for a holistic view of privacy. Specifically, we must consider privacy in the context of both layers together.

On-Chain Bitcoin Entity Clustering

Lightning Network blockchain interactions are permanently reflected in the Bitcoin entity graph. Even if a channel is closed, an attacker can observe which address funded the channel and where the coins are spent after closing it. For this analysis, let's consider four separate entities. Opening a channel causes a monetary flow from a *source entity* to a *funding entity*; closing a channel causes a flow from a *settlement entity* to a *destination entity*.

In early 2021, Romiti et al. identified four heuristics that allow the clustering of these entities. Two of them capture certain leaky funding behavior and two describe leaky settlement behaviors.

Star heuristic (funding)

If a component contains one source entity that forwards funds to one or more funding entities, these funding entities are likely controlled by the same user.

Snake heuristic (funding)

If a component contains one source entity that forwards funds to one or more entities, which themselves are used as source and funding entities, then all these entities are likely controlled by the same user.

Collector heuristic (settlement)

If a component contains one destination entity that receives funds from one or more settlement entities, these settlement entities are likely controlled by the same user.

Proxy heuristic (settlement)

If a component contains one destination entity that receives funds from one or more entities, which themselves are used as settlement and destination entities, then these entities are likely controlled by the same user.

It is worthwhile pointing out that these heuristics might produce false positives. For instance, if transactions of several unrelated users are combined in a CoinJoin transaction, then the star or the proxy heuristic can produce false positives. This could happen if users are funding a payment channel from a CoinJoin transaction. Another potential source of false positives could be that an entity could represent several users if clustered addresses are controlled by a service (e.g., exchange) or on behalf of their users (custodial wallet). However, these false positives can effectively be filtered out.

Countermeasures

If outputs of funding transactions are not reused for opening other channels, the snake heuristic does not work. If users refrain from funding channels from a single external source and avoid collecting funds in a single external destination entity, the other heuristics would not yield any significant results.

Off-Chain Lightning Node Clustering

LN nodes advertise aliases, for instance, *LNBig.com*. Aliases can improve the usability of the system. However, users tend to use similar aliases for their own different nodes. For example, *LNBig.com Billing* is likely owned by the same user as the node with alias *LNBig.com*. Given this observation, one can cluster LN nodes by applying their node aliases. Specifically, one clusters LN nodes into a single address if their aliases are similar with respect to some string similarity metric.

Another method to cluster LN nodes is applying their IP or Tor addresses. If the same IP or Tor addresses correspond to different LN nodes, these nodes are likely controlled by the same user.

Countermeasures

For more privacy, aliases should be sufficiently different from one another. While the public announcement of IP addresses may be unavoidable for those nodes that wish to have incoming channels in the Lightning Network, linkability across nodes of the same user can be mitigated if the clients for each node are hosted with different service providers and thus IP addresses.

Cross-Layer Linking: Lightning Nodes and Bitcoin Entities

Associating LN nodes to Bitcoin entities is a serious breach of privacy that is exacerbated by the fact that most LN nodes publicly expose their IP addresses. Typically, an IP address can be considered as a unique identifier of a user. Two widely observed behavior patterns reveal links between LN nodes and Bitcoin entities:

Coin reuse

Whenever users close payment channels, they get back their corresponding coins. However, many users reuse those coins in opening a new channel. Those coins can effectively be linked to a common LN node.

Entity reuse

Typically, users fund their payment channels from Bitcoin addresses corresponding to the same Bitcoin entity.

These cross-layer linking algorithms could be foiled if users possess multiple unclustered addresses or use multiple wallets to interact with the Lightning Network.

The possible de-anonymization of Bitcoin entities illustrates how important it is to consider the privacy of both layers simultaneously instead of one at a time.

Lightning Graph

The Lightning Network, as the name suggests, is a peer-to-peer network of payment channels. Therefore, many of its properties (privacy, robustness, connectivity, routing efficiency) are influenced and characterized by its network nature.

In this section, we discuss and analyze the Lightning Network from the point of view of network science. We are particularly interested in understanding the LN channel graph, its robustness, connectivity, and other important characteristics.

How Does the Lightning Graph Look in Reality?

One could have expected that the Lightning Network is a random graph, where edges are randomly formed between nodes. If this was the case, then the Lightning Network's degree distribution would follow a Gaussian normal distribution. In particular, most of the nodes would have approximately the same degree, and we would not expect nodes with extraordinarily large degrees. This is because the normal distribution exponentially decreases for values outside of the interval around the average value of the distribution. The depiction of a random graph (as we saw in A visualization of part of the Lightning Network as of July 2021) looks like a mesh network topology. It looks decentralized and nonhierarchical: every node seems to have equal importance. Additionally, random graphs have a large diameter. In particular, routing in such graphs is challenging because the shortest path between any two nodes is moderately long.

However, in stark contrast, the LN graph is completely different.

Lightning graph today

Lightning is a financial network. Thus, the growth and formation of the network are also influenced by economic incentives. Whenever a node joins the Lightning Network, it may want to maximize its connectivity to other nodes in order to increase its routing efficiency. This phenomenon is called preferential attachment. These economic incentives result in a fundamentally different network than a random graph.

Based on snapshots of publicly announced channels, the degree distribution of the Lightning

Network follows a power-law function. In such a graph, the vast majority of nodes have very few connections to other nodes, while only a handful of nodes have numerous connections. At a high level, this graph topology resembles a star: the network has a well-connected core and a loosely connected periphery. Networks with power-law degree distribution are also called scale-free networks. This topology is advantageous for routing payments efficiently but prone to certain topology-based attacks.

Topology-based attacks

An adversary might want to disrupt the Lightning Network and may decide its goal is to dismantle the whole network into many smaller components, making payment routing practically impossible in the whole network. A less ambitious, but still malicious and severe goal might be to only take down certain network nodes. Such a disruption might occur on the node level or on the edge level.

Let’s suppose an adversary can take down any node in the Lightning Network. For instance, it can attack them with a distributed denial of service (DDoS) attack or make them nonoperational by any means. It turns out that if the adversary chooses nodes randomly, then scale-free networks like the Lightning Network are robust against node-removal attacks. This is because a random node lies on the periphery with a small number of connections, therefore playing a negligible role in the network’s connectivity. However, if the adversary is more prudent, it can target the most well-connected nodes. Not surprisingly, the Lightning Network and other scale-free networks are *not* robust against targeted node-removal attacks.

On the other hand, the adversary could be more stealthy. Several topology-based attacks target a single node or a single payment channel. For example, an adversary might be interested in exhausting a certain payment channel’s capacity on purpose. More generally, an adversary can deplete all the outgoing capacity of a node to knock it down from the routing market. This could be easily obtained by routing payments through the victim node with amounts equalling the outgoing capacity of each payment channel. After completing this so-called node isolation attack, the victim cannot send or route payments anymore unless it receives a payment or rebalances its channels.

To conclude, even by design, it is possible to remove edges and nodes from the routable Lightning Network. However, depending on the utilized attack vector, the adversary may have to provide more or fewer resources to carry out the attack.

Temporality of the Lightning Network

The Lightning Network is a dynamically changing, permissionless network. Nodes can freely join or leave the network, they can open and create payment channels anytime they want. Therefore, a single static snapshot of the LN graph is misleading. We need to consider the temporality and ever-changing nature of the network. For now, the LN graph is growing in terms of the number of nodes and payment channels. Its effective diameter is also shrinking; that is, nodes become closer to each other, as we can see in The steady growth of the Lightning Network in nodes, channels, and locked capacity (as of September 2021).

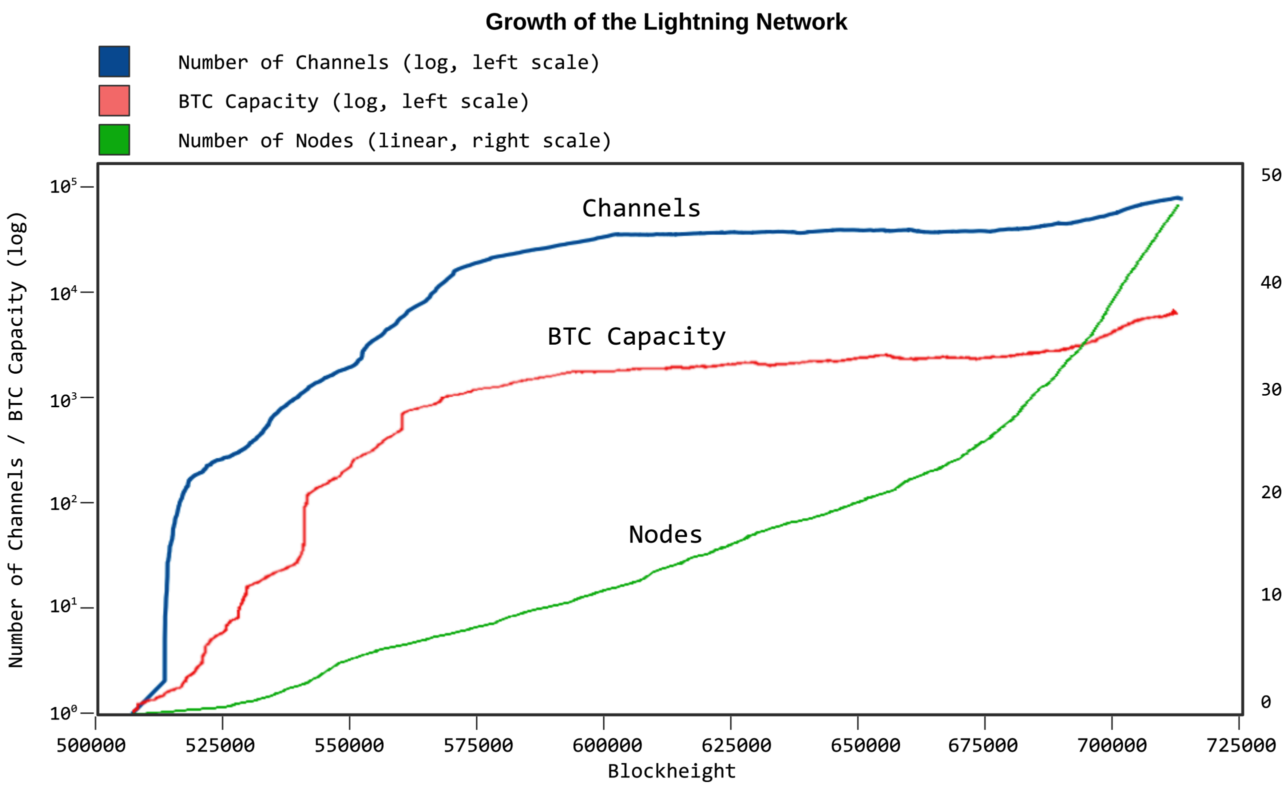


Figure 108. The steady growth of the Lightning Network in nodes, channels, and locked capacity (as of September 2021)

In social networks, triangle closing behavior is common. Specifically, in a graph where nodes represent people and friendships are represented as edges, it is somewhat expected that triangles will emerge in the graph. A triangle, in this case, represents pairwise friendships between three people. For instance, if Alice knows Bob and Bob knows Charlie, then it is likely that at some point Bob will introduce Alice to Charlie. However, this behavior would be strange in the Lightning Network. Nodes are simply not incentivized to close triangles because they could have just routed payments instead of opening a new payment channel. Surprisingly, triangle closing is a common practice in the Lightning Network. The number of triangles was steadily growing before the implementation of multipart payments. This is counterintuitive and surprising given that nodes could have just routed payments through the two sides of the triangle instead of opening the third channel. This may mean that routing inefficiencies incentivized users to close triangles and not fall back on routing. Hopefully, multipart payments will help increase the effectiveness of payment routing.

Centralization in the Lightning Network

A common metric to assess the centrality of a node in a graph is its *betweenness centrality*. Central point dominance is a metric derived from betweenness centrality, used to assess the centrality of a network. For a precise definition of central point dominance, the reader is referred to Freeman's work.

The larger the central point dominance of a network is, the more centralized the network is. We can observe that the Lightning Network has a greater central point dominance (i.e., it is more centralized) than a random graph (Erdős–Rényi graph) or a scale-free graph (Barabási–Albert graph) of equal size.

In general, our understanding of the dynamic nature of the LN channel graph is rather limited. It is fruitful to analyze how protocol changes like multipart payments can affect the dynamics of the Lightning Network. It would be beneficial to explore the temporal nature of the LN graph in more depth.

Economic Incentives and Graph Structure

The LN graph forms spontaneously, and nodes connect to each other based on mutual interest. As a result, incentives drive graph development. Let’s look at some of the relevant incentives:

- Rational incentives:
  - Nodes establish channels to send, receive, and route payments (earn fees).
  - What makes a channel more likely to be established between two nodes that act rationally?
- Altruistic incentives:
  - Nodes establish channels "for the good of the network."
  - While we should not base our security assumptions on altruism, to a certain extent, altruistic behavior drives Bitcoin (accepting incoming connections, serving blocks).
  - What role does it play in Lightning?

In the early stages of the Lightning Network, many node operators have claimed that the earned routing fees do not compensate for the opportunity costs stemming from liquidity lock-up. This would indicate that operating a node may be driven mostly by altruistic incentives "for the good of the network." This might change in the future if the Lightning Network has significantly larger traffic or if a market for routing fees emerges. On the other hand, if a node wishes to optimize its routing fees, it would minimize the average shortest path lengths to every other node. Put differently, a profit-seeker node will try to locate itself in the *center* of the channel graph or close to it.

Practical Advice for Users to Protect Their Privacy

We’re still in the early stages of the Lightning Network. Many of the concerns listed in this chapter are likely to be addressed as it matures and grows. In the meantime, there are some measures that you can take to guard your node against malicious users; something as simple as updating the default parameters that your node runs with can go a long way in hardening your node.

Unannounced Channels

If you intend to use the Lightning Network to send and receive funds between nodes and wallets you control, and have no interest in routing other users' payments, there is little need to announce your channels to the rest of the network. You could open a channel between, say, your desktop PC running a full node and your mobile phone running a Lightning wallet, and simply forgo the channel announcement discussed in How the Lightning Network Works. These are sometimes called "private" channels; however, it is more correct to refer to them as "unannounced" channels because they are not strictly private.

Unannounced channels will not be known to the rest of the network and won’t normally be used to route other users' payments. They can still be used to route payments if other nodes are made aware of them; for example, an invoice could contain routing hints which suggests a path with an unannounced channel. However, assuming that you've only opened an unannounced channel with yourself, you do gain some measure of privacy. Since you are not exposing your channel to the network, you lower the risk of a denial-of-service attack on your node. You can also more easily manage the capacity of this channel, since it will only be used to receive or send directly to your node.

There are also advantages to opening an unannounced channel with a known party that you transact with frequently. For example, if Alice and Bob frequently play poker for bitcoin, they could open a channel to send their winnings back and forth. Under normal conditions, this channel will not be used to route payments from other users or collect fees. And since the channel will not be known to the rest of the network, any payments between Alice and Bob cannot be inferred by tracking changes in the channel's routing capacity. This confers some privacy to Alice and Bob; however, if one of them decides to make other users aware of the channel, such as by including it in the routing hints of an invoice, then this privacy is lost.

It should also be noted that to open an unannounced channel, a public transaction must be made on the Bitcoin blockchain. Hence it is possible to infer the existence and size of the channel if a malicious party is monitoring the blockchain for channel opening transactions and attempting to match them to channels on the network. Furthermore, when the channel is closed, the final balance of the channel will be made public once it's committed to the Bitcoin blockchain. However, since the opening and commitment transactions are pseudonymous, it will not be a simple matter to connect it back to Alice or Bob. In addition, the Taproot update of 2021 makes it difficult to distinguish between channel opening and closing transactions and other specific kinds of Bitcoin transactions. Hence, while unannouned channels are not completely private, they do provide some privacy benefits when used carefully.

Routing Considerations

As covered in Denial of Service, nodes that open public channels expose themselves to the risk of a series of attacks on their channels. While mitigations are being developed on the protocol level, there are many steps that a node can take to protect against denial of service attacks on their public channels:

Minimum HTLC size

On channel open, your node can set the minimum HTLC size that it will accept. Setting a higher value ensures that each of your available channel slots cannot be occupied by a very small payment.

Rate limiting

Many node implementations allow nodes to dynamically accept or reject HTLCs that are forwarded through your node. Some useful guidelines for a custom rate limiter are as follows:

- Limit the number of commitment slots a single peer may consume
- Monitor failure rates from a single peer, and rate limit if their failures spike suddenly

Shadow channels

Nodes that wish to open large channels to a single target can instead open a single public channel to the target and support it with further private channels called shadow channels. These channels can still be used for routing but are not announced to potential attackers.

Accepting Channels

At present, Lightning nodes struggle with bootstrapping inbound liquidity. While there are some paid solutions to acquiring inbound liquidity, like swap services, channel markets, and paid channel opening services from known hubs, many nodes will gladly accept any legitimate looking channel opening request to increase their inbound liquidity.

Stepping back to the context of Bitcoin, this can be compared to the way that Bitcoin Core treats its incoming and outgoing connections differently out of concern that the node may be eclipsed. If a node opens an incoming connection to your Bitcoin node, you have no way of knowing whether the initiator randomly selected you or is specifically targeting your node with malicious intent. Your outgoing connections do not need to be treated with such suspicion because either the node was selected randomly from a pool of many potential peers or you intentionally connected to the peer manually.

The same can be said in Lightning. When you open a channel, it is done with intention, but when a remote party opens a channel to your node, you have no way of knowing whether this channel will be used to attack your node or not. As several papers note, the relatively low cost of spinning up a node and opening channels to targets is one of the significant factors that make attacks easy. If you accept incoming channels, it is prudent to place some restrictions on the nodes you accept incoming channels from. Many implementations expose channel acceptance hooks that allow you to tailor your channel acceptance policies to your preferences.

The question of accepting and rejecting channels is a philosophical one. What if we end up with a Lightning Network where new nodes cannot participate because they cannot open any channels? Our suggestion is not to set an exclusive list of "mega-hubs" from which you will accept channels, but rather to accept channels in a manner that suits your risk preference.

Some potential strategies are:

No risk

Do not accept any incoming channels.

Low risk

Accept channels from a known set of nodes that you have previously had successful channels open with.

Medium risk

Only accept channels from nodes that have been present in the graph for a longer period and have some long-lived channels.

Higher risk

Accept any incoming channels, and implement the mitigations described in Routing Considerations.

Conclusion

In summary, privacy and security are nuanced, complex topics, and while many researchers and developers are looking for network-wide improvements, it’s important for everyone participating in the network to understand what they can do to protect their own privacy and increase security on an individual node level.

References and Further Reading

In this chapter, we used many references from ongoing research on Lightning security. You may find these useful articles and papers listed by topic in the following lists.

Privacy and probing attacks

- Jordi Herrera-Joancomarti□ et al. "On the Difficulty of Hiding the Balance of Lightning Network Channels". *Asia CCS '19: Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security* (July 2019): 602–612.
- Utz Nisslmueller et al. "Toward Active and Passive Confidentiality Attacks on Cryptocurrency Off-Chain Networks." arXiv preprint, https://arxiv.org/abs/2003.00003 (2020).
- Sergei Tikhomirov et al. "Probing Channel Balances in the Lightning Network." arXiv preprint, https://arxiv.org/abs/2004.00333 (2020).
- George Kappos et al. "An Empirical Analysis of Privacy in the Lightning Network." arXiv preprint, https://arxiv.org/abs/2003.12470 (2021).
- Zap source code with the probing function.

Congestion attacks

- Ayelet Mizrahi and Aviv Zohar. "Congestion Attacks in Payment Channel Networks." arXiv preprint, https://arxiv.org/abs/2002.06564 (2020).

Routing considerations

- Marty Bent, interview with Joost Jager, *Tales from the Crypt*, podcast audio, October 2, 2020, https://anchor.fm/tales-from-the-crypt/episodes/197-Joost-Jager-ekghn6.

Conclusion

In just a few years, the Lightning Network has gone from a whitepaper to a rapidly growing global network. As Bitcoin's second layer, it has delivered on the promise of fast, inexpensive, and private payments. Additionally, it has started a tsunami of innovation, as it unleashes developers from the constraints of lockstep consensus that exist in Bitcoin development.

Innovation in the Lightning Network is happening in several different levels:

- At Bitcoin's Core protocol, providing use and demand for new Bitcoin Script opcodes, signing algorithms, and optimizations
- At the Lightning protocol level, with new features deployed rapidly network wide
- At the payment channel level, with new channel constructs and enhancements
- As distinct opt-in features deployed end-to-end by independent implementations that senders and recipients can use if they want
- With new and exciting Lightning Applications (LApps) build on top of the clients and protocols

Let's look at how these innovations are changing Lightning now and in the near future.

Decentralized and Asynchronous Innovation

Lightning isn't bound by lockstep consensus, as is the case with Bitcoin. That means that different Lightning clients can implement different features and negotiate their interactions (see Feature Bits and Protocol Extensibility). As a result, innovation in the Lightning Network is occurring at a much faster rate than in Bitcoin.

Not only is Lightning advancing rapidly, but it is creating demand for new features in the Bitcoin system. Many recent and planned innovations in Bitcoin are both motivated and justified by their use in the Lightning Network. In fact, the Lightning Network is often mentioned as an example use case for many of the new features.

Bitcoin Protocol and Bitcoin Script Innovation

The Bitcoin system is, by necessity, a conservative system that has to preserve compatibility with consensus rules to avoid unplanned forks of the blockchain or partitions of the P2P network. As a result, new features require a lot of coordination and testing before they are implemented in mainnet, the live production system.

Here are some of the current or proposed innovations in Bitcoin that are motivated by use cases in the Lightning Network:

Neutrino

A lightweight client protocol with improved privacy features over the legacy SPV protocol. Neutrino is mostly used by Lightning clients to access the Bitcoin blockchain.

Schnorr signatures

Introduced as part of the *Taproot* soft fork, Schnorr signatures will enable flexible Point Time-

Locked Contracts (PTLCs) for channel construction in Lightning. This might in particular make use of revocable signatures instead of revocable transactions.

Taproot

Also part of the November 2021 soft fork that introduces Schnorr signatures, Taproot allows complex scripts to appear as single-payer, single-payee payments, and indistinguishable from the most common type of payment on Bitcoin. This will allow Lightning channel cooperative (mutual) closure transactions to appear indistinguishable from simple payments and increase privacy for LN users.

Input rebinding

Also known by the names SIGHAS_NOINPUT or SIGHASH_ANYPREVOUT, this planned upgrade to the Bitcoin Script language is primarily motivated by advanced smart contracts such as the eltoo channel protocol.

Covenants

Currently in the early stages of research, covenants allow transactions to create outputs that constrain future transactions which spend them. This mechanism could increase security for Lightning channels by making it possible to enforce address whitelisting in commitment transactions.

Lightning Protocol Innovation

The Lightning P2P protocol is highly extensible and has undergone a lot of change since its inception. The "It's OK to be odd" rule used in feature bits (see Feature Bits and Protocol Extensibility) ensures that nodes can negotiate the features they support, enabling multiple independent upgrades to the protocol.

TLV Extensibility

The Type-Length-Value (see Type-Length-Value (TLV) Format) mechanism for extending the messaging protocol is extremely powerful and has already enabled the introduction of several new capabilities in Lightning while maintaining both forward and backward compatibility. A prominent example, which is currently being developed and makes use of this, is path blinding and trampoline payments. This allows a recipient to hide itself from the sender, but also allows mobile clients to send payments without the necessity of storing the full channel graph on their devices by using a third party to which they don't need to reveal the final recipient.

Payment Channel Construction

Payment channels are an abstraction that is operated by two channel partners. As long as those two are willing to run new code, they can implement a variety of channel mechanisms simultaneously. In fact, recent research suggests that channels could even be upgraded to a new mechanism dynamically, without closing the old channel and opening a new channel type.

eltoo

A proposed channel mechanism that uses input-rebinding to significantly simplify the operation of payment channels and remove the need for the penalty mechanism. It needs a new Bitcoin signature type before it can be implemented

Opt-In End-to-End Features

Point Time-Locked Contracts (PTLCs)

A different approach to HTLCs, PTLCs can increase privacy, reduce information leaked to intermediary nodes, and operate more efficiently than HTLC-based channels.

Large channels

Large or *Wumbo* channels were introduced in a dynamic way to the network without requiring coordination. Channels that support large payments are advertized as part of the channel announcement messages and can be used in an opt-in manner.

Multipart payments (MPP)

MPP was also introduced in an opt-in manner, but even better only requires the sender and recipient of a payment to be able to do MPP. The rest of the network simply routes HTLCs as if they are single-part payments.

JIT routing

An optional method that can be used by routing nodes to increase their reliability and to protect themselves from being spammed.

Keysend

An upgrade introduced independently by Lightning client implementations, it allows the sender to send money in an "unsolicited" and asynchronous way without requiring an invoice first.

HODL invoices[11]

Payments where the final HTLC is not collected, committing the sender to the payment, but allowing the recipient to delay collection until some other condition is satisfied, or cancel the invoice without collection. This was also implemented independently by different Lightning clients and can be used in an opt-in manner.

Onion routed message services

The onion routing mechanism and the underlying public key database of nodes can be used to send data that is unrelated to payments, such as text messages or forum posts. The use of Lightning to enable paid messaging as a solution to spam posts and Sybil attacks (spam) is another innovation that was implemented independently of the core protocol.

Offers

Currently proposed as BOLT #12 but already implemented by some nodes, this is a communication protocol to request (recurring) invoices from remote nodes via Onion messages.

Lightning Applications (LApps)

While still in their infancy, we are already seeing the emergence of interesting Lightning Applications. Broadly defined as an application that uses the Lightning Protocol or a Lightning client as a component, LApps are the application layer of Lightning. A prominent example is LNURL, which provides a similar functionality as BOLT #12 offers, but just over HTTP and

Lightning addresses. It works on top of offers to provide users with an email-style address to which others can send funds while the software in the background requests an invoice against the LNURL endpoint of the node. Further LApps are being built for simple games, messaging applications, microservices, payable APIs, paid dispensers (e.g., fuel pumps), derivative trading systems, and much more.

Ready, Set, Go!

The future is looking bright. The Lightning Network is taking Bitcoin to new unexplored markets and applications. Equipped with the knowledge in this book, you can explore this new frontier or maybe even join as a pioneer and forge a new path.

Appendix A: Bitcoin Fundamentals Review

The Lightning Network is capable of running above multiple blockchains, but is primarily anchored on Bitcoin. To understand the Lightning Network, you need a fundamental understanding of Bitcoin and its building blocks.

There are many good resources that you can use to learn more about Bitcoin, including the companion book *Mastering Bitcoin*, 2nd Edition, by Andreas M. Antonopoulos, which you can find on GitHub under an open source license. However, you do not need to read a whole other book to be ready for this one!

In this chapter, we've collected the most important concepts you need to know about Bitcoin and explained them in the context of the Lightning Network. This way you can learn exactly what you need to know to grasp the Lightning Network without any distractions.

This chapter covers several important concepts from Bitcoin, including:

- Keys and digital signatures
- Hash functions
- Bitcoin transactions and their structure
- Bitcoin transaction chaining
- Transaction outpoints
- Bitcoin Script: locking and unlocking scripts
- Basic locking scripts
- Complex and conditional locking scripts
- Timelocks

Keys and Digital Signatures

You may have heard that bitcoin is based on *cryptography*, which is a branch of mathematics used extensively in computer security. Cryptography can also be used to prove knowledge of a secret without revealing that secret (digital signature), or prove the authenticity of data (digital fingerprint). These types of cryptographic proofs are the mathematical tools critical to Bitcoin and used extensively in Bitcoin applications.

Ownership of bitcoin is established through *digital keys*, *bitcoin addresses*, and *digital signatures*. The digital keys are not actually stored in the network, but are instead created and stored by users in a file, or simple database, called a *wallet*. The digital keys in a user's wallet are completely independent of the Bitcoin Protocol and can be generated and managed by the user's wallet software without reference to the blockchain or access to the internet.

Most Bitcoin transactions require a valid digital signature to be included in the blockchain, which can only be generated with a secret key; therefore, anyone with a copy of that key has control of the bitcoin. The digital signature used to spend funds is also referred to as a *witness*, a term used in cryptography. The witness data in a bitcoin transaction testifies to the true ownership of the funds being spent. Keys come in pairs consisting of a private (secret) key and a public key. Think of the public key as similar to a bank account number and the private key as similar to the secret PIN.

Private and Public Keys

A private key is simply a number, picked at random. In practice, and to make managing many keys easy, most Bitcoin wallets generate a sequence of private keys from a single random *seed* using a deterministic derivation algorithm. Simply put, a single random number is used to produce a repeatable sequence of seemingly random numbers that are used as private keys. This allows users to only back up the seed and be able to *derive* all the keys they need from that seed.

Bitcoin, like many other cryptocurrencies and blockchains, uses *elliptic curves* for security. In Bitcoin, elliptic curve multiplication on the *secp256k1* elliptic curve is used as a *one-way function*. Simply put, the nature of elliptic curve math makes it trivial to calculate the scalar multiplication of a point but impossible to calculate the inverse (division, or discrete logarithm).

Each private key has a corresponding *public key*, which is calculated from the private key, using scalar multiplication on the elliptic curve. In simple terms, with a private key *k*, we can multiply it with a constant *G* to produce a public key *K*:

```
<ul class="simplelist">
<li><em>K</em> = <em>k</em>*<em>G</em></li>
</ul>
```

It is impossible to reverse this calculation. Given a public key *K*, one cannot calculate the private key *k*. Division by *G* is not possible in elliptic curve math. Instead, one would have to try all possible values of *k* in an exhaustive process called a *brute-force attack*. Because *k* is a 256-bit number, exhausting all possible values with any classical computer would require more time and energy than available in this universe.

Hashes

Another important tool used extensively in Bitcoin, and in the Lightning Network, are *cryptographic hash functions*, and specifically the SHA-256 hash function.

A hash function, also known as a *digest function*, is a function that takes arbitrary length data and transforms it into a fixed length result, called the *hash*, *digest*, or *fingerprint* (see The SHA-256 cryptographic hash algorithm). Importantly, hash functions are *one-way* functions, meaning that you can't reverse them and calculate the input data from the fingerprint.

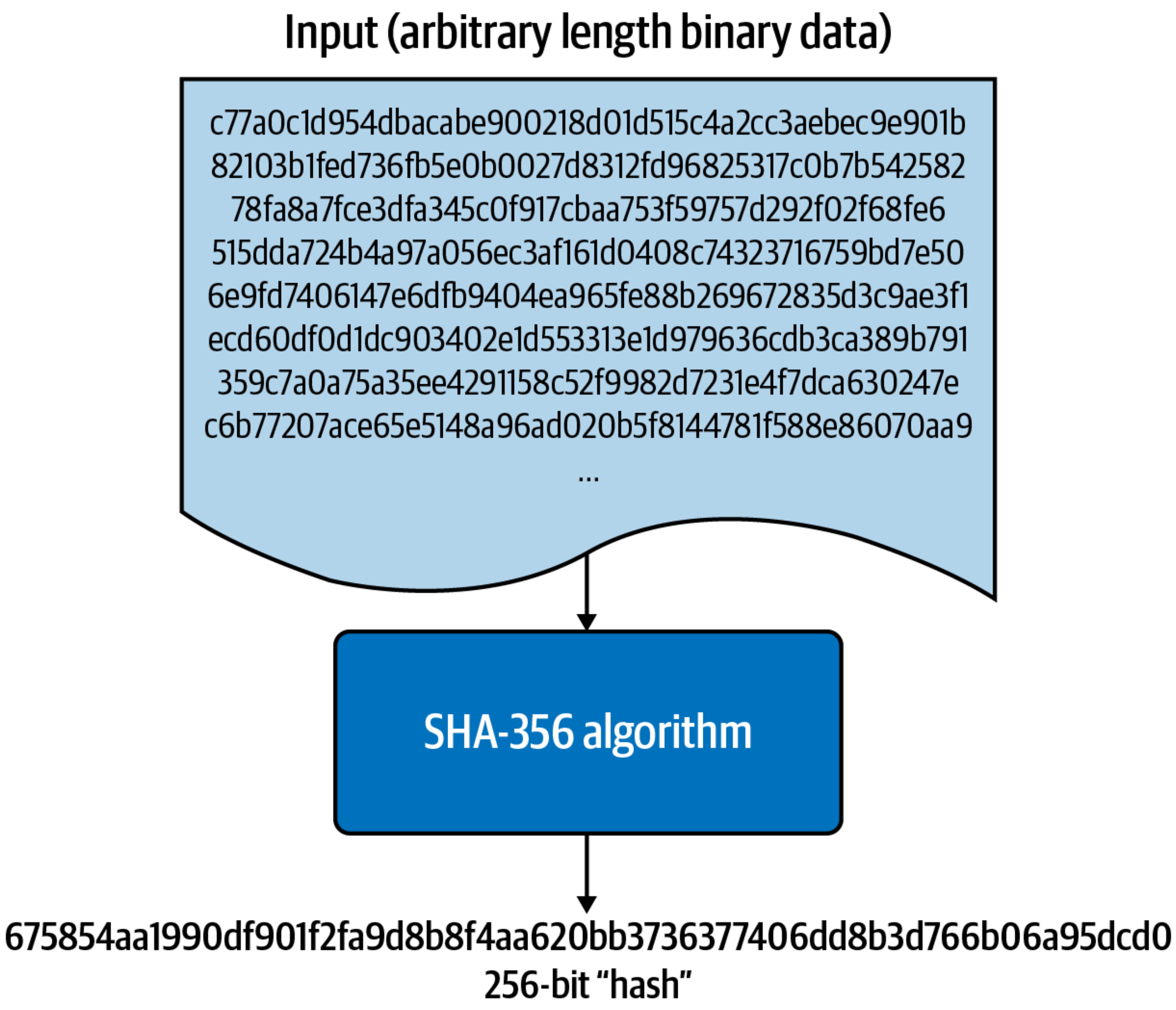

Input (arbitrary length binary data)
c77a0c1d954dbacabe900218d01d515c4a2cc3aebec9e901b
82103b1fed736fb5e0b0027d8312fd96825317c0b7b542582
78fa8a7fce3dfa345c0f917cbaa753f59757d292f02f68fe6
515dda724b4a97a056ec3af161d0408c74323716759bd7e50
6e9fd7406147e6dfb9404ea965fe88b269672835d3c9ae3f1
ecd60df0d1dc903402e1d553313e1d979636cdb3ca389b791
359c7a0a75a35ee4291158c52f9982d7231e4f7dca630247e
c6b77207ace65e5148a96ad020b5f8144781f588e86070aa9
...
SHA-356 algorithm
675854aa1990df901f2fa9d8b8f4aa620bb3736377406dd8b3d766b06a95dcd0
256-bit "hash"

Figure 109. The SHA-256 cryptographic hash algorithm

For example, if we use a command-line terminal to feed the text "Mastering the Lightning Network" into the SHA-256 function, it will produce a fingerprint as follows:

```
$ echo -n "Mastering the Lightning Network" | shasum -a 256

ce86e4cd423d80d054b387aca23c02f5fc53b14be4f8d3ef14c089422b2235de  -
```

TIP The input used to calculate a hash is also called a *preimage*.

The length of the input can be much bigger, of course. Let's try the same thing with the PDF file of the Bitcoin whitepaper from Satoshi Nakamoto:

```
$ wget http://bitcoin.org/bitcoin.pdf
$ cat bitcoin.pdf | shasum -a 256
b1674191a88ec5cdd733e4240a81803105dc412d6c6708d53ab94fc248f4f553  -
```

While it takes longer than a single sentence, the SHA-256 function processes the 9-page PDF, "digesting" it into a 256-bit fingerprint.

Now at this point you might be wondering how it is possible for a function that digests data of unlimited size to produce a unique fingerprint that is a fixed-size number?

In theory, since there is an infinite number of possible preimages (inputs) and only a finite number of fingerprints, there must be many preimages that produce the same 256-bit fingerprint. When two preimages produce the same hash, this is known as a *collision*.

In practice, a 256-bit number is so large that you will never find a collision on purpose. Cryptographic hash functions work on the basis that a search for a collision is a brute-force effort that takes so much energy and time that it is not practically possible.

Cryptographic hash functions are broadly used in a variety of applications because they have some useful features. They are:

Deterministic
: The same input always produces the same hash.

Irreversible
: It is not possible to compute the preimage of a hash.

Collision-proof
: It is computationally infeasible to find two messages that have the same hash.

Uncorrelated
: A small change in the input produces such a big change in the output that the output seems uncorrelated to the input.

Uniform/random
: A cryptographic hash function produces hashes that are uniformly distributed across the entire 256-bit space of possible outputs. The output of a hash appears to be random, though it is not truly random.

Using these features of cryptographic hashes, we can build some interesting applications:

Fingerprints
: A hash can be used to fingerprint a file or message so that it can be uniquely identified. Hashes can be used as universal identifiers of any data set.

Integrity proof
: A fingerprint of a file or message demonstrates its integrity because the file or message cannot be tampered with or modified in any way without changing the fingerprint. This is often used to ensure software has not been tampered with before installing it on your computer.

Commitment/nonrepudiation
: You can commit to a specific preimage (e.g., a number or message) without revealing it by publishing its hash. Later, you can reveal the secret, and everyone can verify that it is the same thing you committed to earlier because it produces the published hash.

Proof-of-work/hash grinding

You can use a hash to prove you have done computational work by showing a nonrandom pattern in the hash which can only be produced by repeated guesses at a preimage. For example, the hash of a Bitcoin block header starts with a lot of zero bits. The only way to produce it is by changing a part of the header and hashing it trillions of times until it produces that pattern by chance.

Atomicity

You can make a secret preimage a prerequisite of spending funds in several linked transactions. If any one of the parties reveals the preimage in order to spend one of the transactions, all the other parties can now spend their transactions too. All or none become spendable, achieving atomicity across several transactions.

Digital Signatures

The private key is used to create signatures that are required to spend bitcoin by proving ownership of funds used in a transaction.

A *digital signature* is a number that is calculated from the application of the private key to a specific message.

Given a message m and a private key k, a signature function F_{sign} can produce a signature S:

```
$ S = F_{sign}(m, k) $
```

This signature S can be independently verified by anyone who has the public key K (corresponding to private key k), and the message:

```
$ F_{verify}(m, K, S) $
```

If F_{verify} returns a true result, then the verifier can confirm that the message m was signed by someone who had access to the private key k. Importantly, the digital signature proves the possession of the private key k at the time of signing, without revealing k.

Digital signatures use a cryptographic hash algorithm. The signature is applied to a hash of the message, so that the message m is "summarized" to a fixed-length hash $H(m)$ that serves as a fingerprint.

By applying the digital signature on the hash of a transaction, the signature not only proves the authorization, but also "locks" the transaction data, ensuring its integrity. A signed transaction cannot be modified because any change would result in a different hash and invalidate the signature.

Signature Types

Signatures are not always applied to the entire transaction. To provide signing flexibility, a Bitcoin digital signature contains a prefix called the signature hash type, which specifies which part of the transaction data is included in the hash. This allows the signature to commit or "lock" all, or only some of, the data in the transaction. The most common signature hash type is SIGHASH_ALL which locks everything in the transaction by including all the transaction data in the hash that is signed. By comparison, SIGHASH_SINGLE locks all the transaction inputs, but only one output (more about inputs and outputs in the next section). Different signature hash types can be combined to produce six different "patterns" of transaction data that are locked by the signature.

More information about signature hash types can be found in the section "Signature Hash Types" in Chapter 6 of *Mastering Bitcoin*, Second Edition.

Bitcoin Transactions

Transactions are data structures that encode the transfer of value between participants in the bitcoin system.

Inputs and Outputs

The fundamental building block of a bitcoin transaction is a transaction output. *Transaction outputs* are indivisible chunks of bitcoin currency, recorded on the blockchain, and recognized as valid by the entire network. A transaction spends inputs and creates outputs. Transaction *inputs* are simply references to outputs of previously recorded transactions. This way, each transaction spends the outputs of previous transactions and creates new outputs (see A transaction transfers value from inputs to outputs).

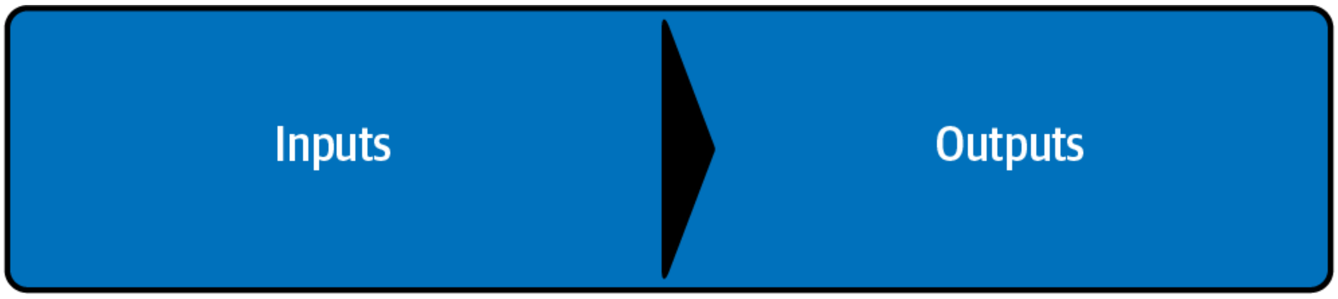


Figure 110. A transaction transfers value from inputs to outputs

Bitcoin full nodes track all available and spendable outputs, known as *unspent transaction outputs* (UTXOs). The collection of all UTXOs is known as the UTXO set, which currently numbers in the millions of UTXOs. The UTXO set grows as new UTXOs are created and shrinks when UTXOs are consumed. Every transaction represents a change (state transition) in the UTXO set, by consuming one or more UTXOs as *transaction inputs* and creating one or more UTXOs as its *transaction outputs*.

For example, let’s assume that a user Alice has a 100,000 satoshi UTXO that she can spend. Alice can pay Bob 100,000 satoshi by constructing a transaction with one input (consuming her existing 100,000 satoshi input) and one output that "pays" Bob 100,000 satoshi. Now Bob has a 100,000 satoshi UTXO that he can spend, creating a new transaction that consumes this new UTXO and spends it to another UTXO as a payment to another user, and so on (see Alice pays 100,000 satoshis to Bob).

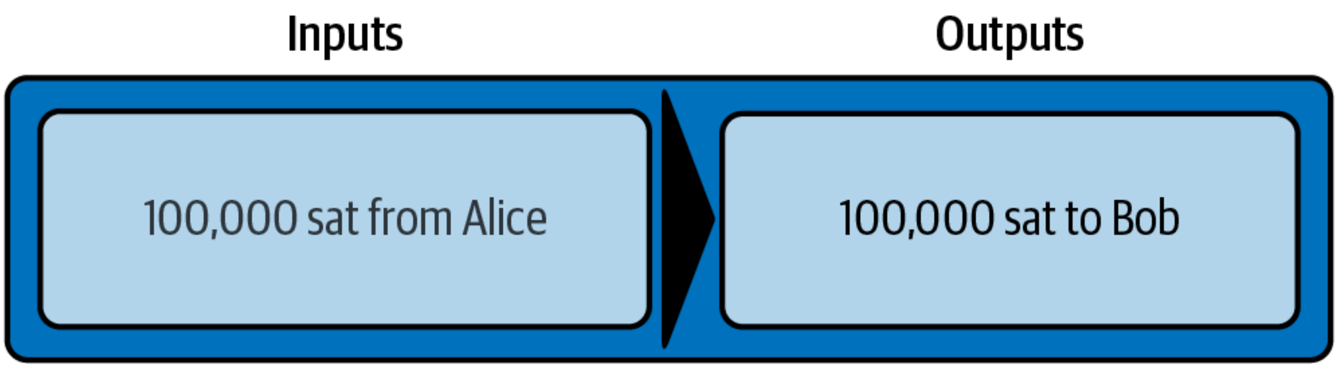


Figure 111. Alice pays 100,000 satoshis to Bob

A transaction output can have an arbitrary (integer) value denominated in satoshis. Just as dollars can be divided down to two decimal places as cents, bitcoin can be divided down to eight decimal places as satoshis. Although an output can have any arbitrary value, once created it is indivisible. This is an important characteristic of outputs that needs to be emphasized: outputs are discrete and indivisible units of value, denominated in integer satoshis. An unspent output can only be consumed in its entirety by a transaction.

So what if Alice wants to pay Bob 50,000 satoshi, but only has an indivisible 100,000 satoshi UTXO? Alice will need to create a transaction that consumes (as its input) the 100,000 satoshi UTXO and has two outputs: one paying 50,000 satoshi to Bob and one paying 50,000 satoshi *back* to Alice as "change" (see Alice pays 50k sat to Bob and 50k sat to herself as change).

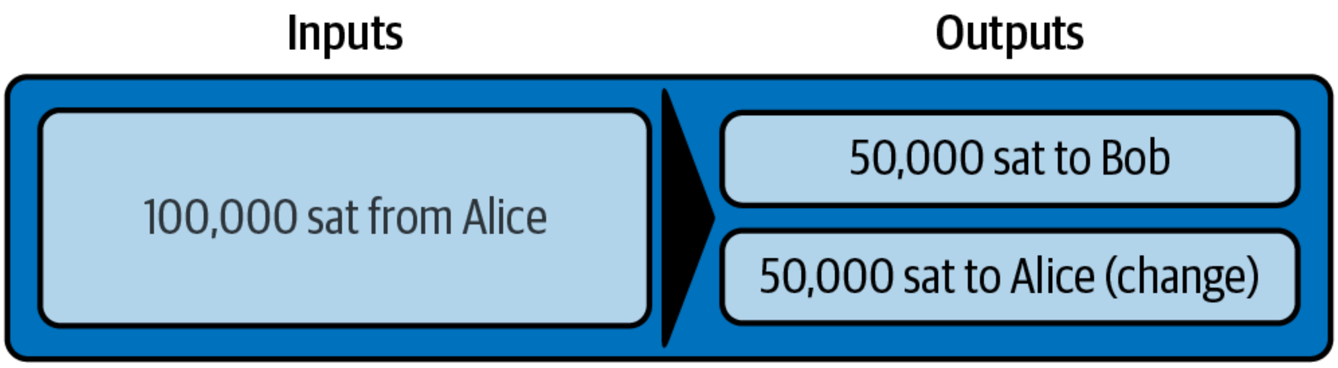


Figure 112. Alice pays 50k sat to Bob and 50k sat to herself as change

TIP There’s nothing special about a change output or any way to distinguish it from any other output. It doesn’t have to be the last output. There could be more than one change output, or no change outputs. Only the creator of the transaction knows which outputs are to others and which outputs are to addresses they own and therefore "change."

Similarly, if Alice wants to pay Bob 85,000 satoshi but has two 50,000 satoshi UTXOs available, she has to create a transaction with two inputs (consuming both her 50,000 satoshi UTXOs) and two outputs, paying Bob 85,000 and sending 15,000 satoshi back to herself as change (see Alice uses two 50k inputs to pay 85k sat to Bob and 15k sat to herself as change).

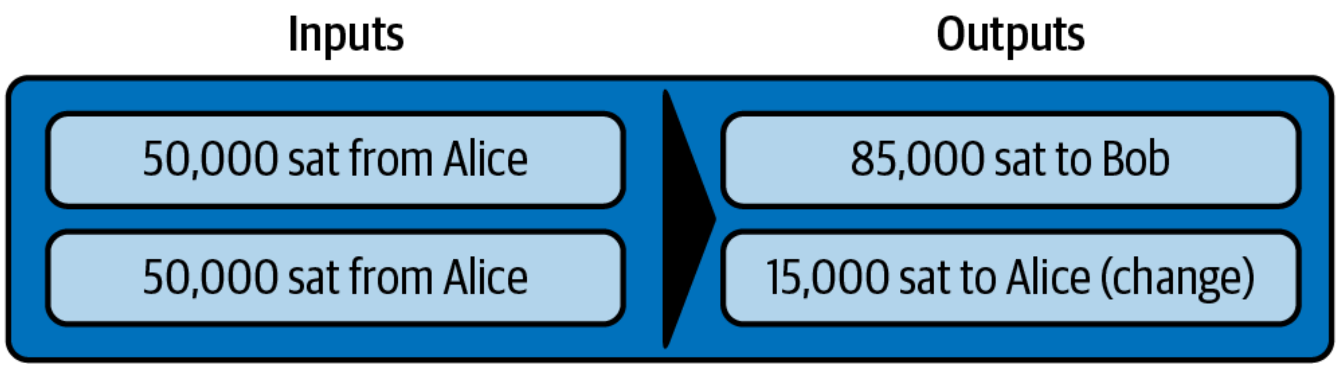


Figure 113. Alice uses two 50k inputs to pay 85k sat to Bob and 15k sat to herself as change

The preceding illustrations and examples show how a Bitcoin transaction combines (spends) one or more inputs and creates one or more outputs. A transaction can have hundreds or even thousands of inputs and outputs.

TIP While the transactions created by the Lightning Network have multiple outputs, they do not have "change" per se, because the entire available balance of a channel is split between the two channel partners.

Transaction Chains

Every output can be spent as an input in a subsequent transaction. So, for example, if Bob decided to spend 10,000 satoshi in a transaction paying Chan, and Chan spent 4,000 satoshi to pay Dina, it would play out as shown in Alice pays Bob who pays Chan who pays Dina.

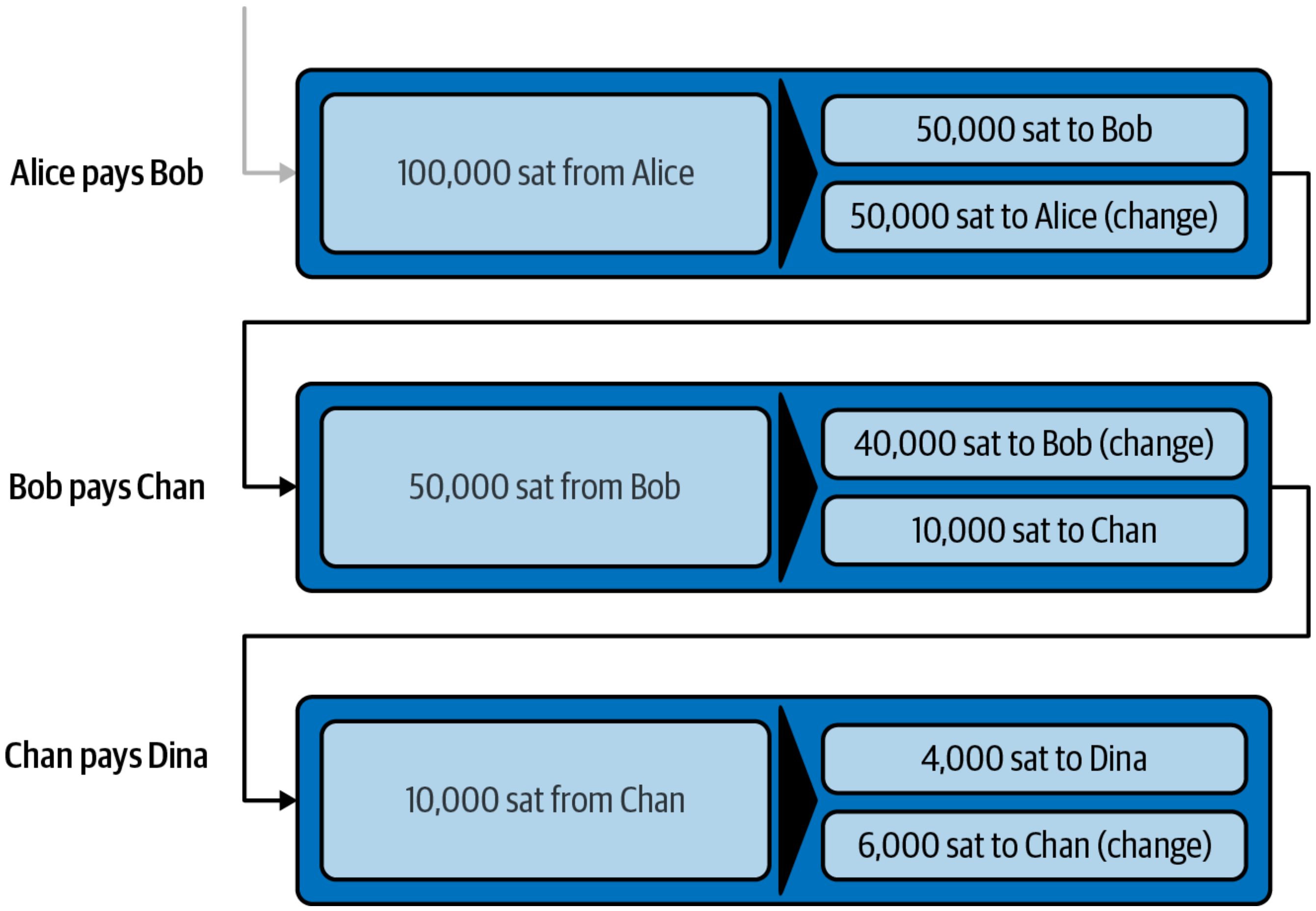


Figure 114. Alice pays Bob who pays Chan who pays Dina

An output is considered *spent* if it is referenced as an input in another transaction that is recorded on the blockchain. An output is considered *unspent* (and available for spending) if no recorded transaction references it.

The only type of transaction that doesn't have inputs is a special transaction created by Bitcoin miners called the *coinbase transaction*. The coinbase transaction has only outputs and no inputs because it creates new bitcoin from mining. Every other transaction spends one or more previously recorded outputs as its inputs.

Since transactions are chained, if you pick a transaction at random, you can follow any one of its inputs backward to the previous transaction that created it. If you keep doing that, you will eventually reach a coinbase transaction where the bitcoin was first mined.

TxID: Transaction Identifiers

Every transaction in the Bitcoin system is identified by a unique identifier (assuming the existence of BIP-0030), called the *transaction ID* or *TxID* for short. To produce a unique identifier, we use the SHA-256 cryptographic hash function to produce a hash of the transaction's data. This "fingerprint" serves as a universal identifier. A transaction can be referenced by its transaction ID, and once a transaction is recorded on the Bitcoin blockchain, every node in the Bitcoin network knows that this transaction is valid.

For example, a transaction ID might look like this:

A transaction ID produced from hashing the transaction data

```
e31e4e214c3f436937c74b8663b3ca58f7ad5b3fce7783eb84fd9a5ee5b9a54c
```

This is a real transaction (created as an example for the *Mastering Bitcoin* book) that can be found on the Bitcoin blockchain.

Try to find it by entering this TxID into a block explorer:

```
<ul class="simplelist">
<li><a
href="https://blockstream.info/tx/e31e4e214c3f436937c74b8663b3ca58f7ad5b3fce7783eb84fd
9a5ee5b9a54c"><em>https://blockstream.info/tx/e31e4e214c3f436937c74b8663b3ca58f7ad5b3f
ce7783eb84fd9a5ee5b9a54c</em></a></li></ul>
```

or use the short link (case-sensitive):

```
<ul class="simplelist">
<li><a href="http://bit.ly/AliceTx"><em>http://bit.ly/AliceTx</em></a></li>
</ul>
```

Outpoints: Output Identifiers

Because every transaction has a unique ID, we can also identify a transaction output within that transaction uniquely by reference to the TxID and the output index number. The first output in a transaction is output index 0, the second output is output index 1, and so on. An output identifier is commonly known as an *outpoint*.

By convention we write an outpoint as the TxID, a colon, and the output index number:

A outpoint: identifying an output by TxID and index number

```
7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18:0
```

Output identifiers (outpoints) are the mechanisms that link transactions together in a chain. Every transaction input is a reference to a specific output of a previous transaction. That reference is an outpoint: a TxID and output index number. So a transaction "spends" a specific output (by index number) from a specific transaction (by TxID) to create new outputs that themselves can be spent by reference to the outpoint.

Transaction inputs refer to outpoints forming a chain presents the chain of transactions from Alice to Bob to Chan to Dina, this time with outpoints in each of the inputs.

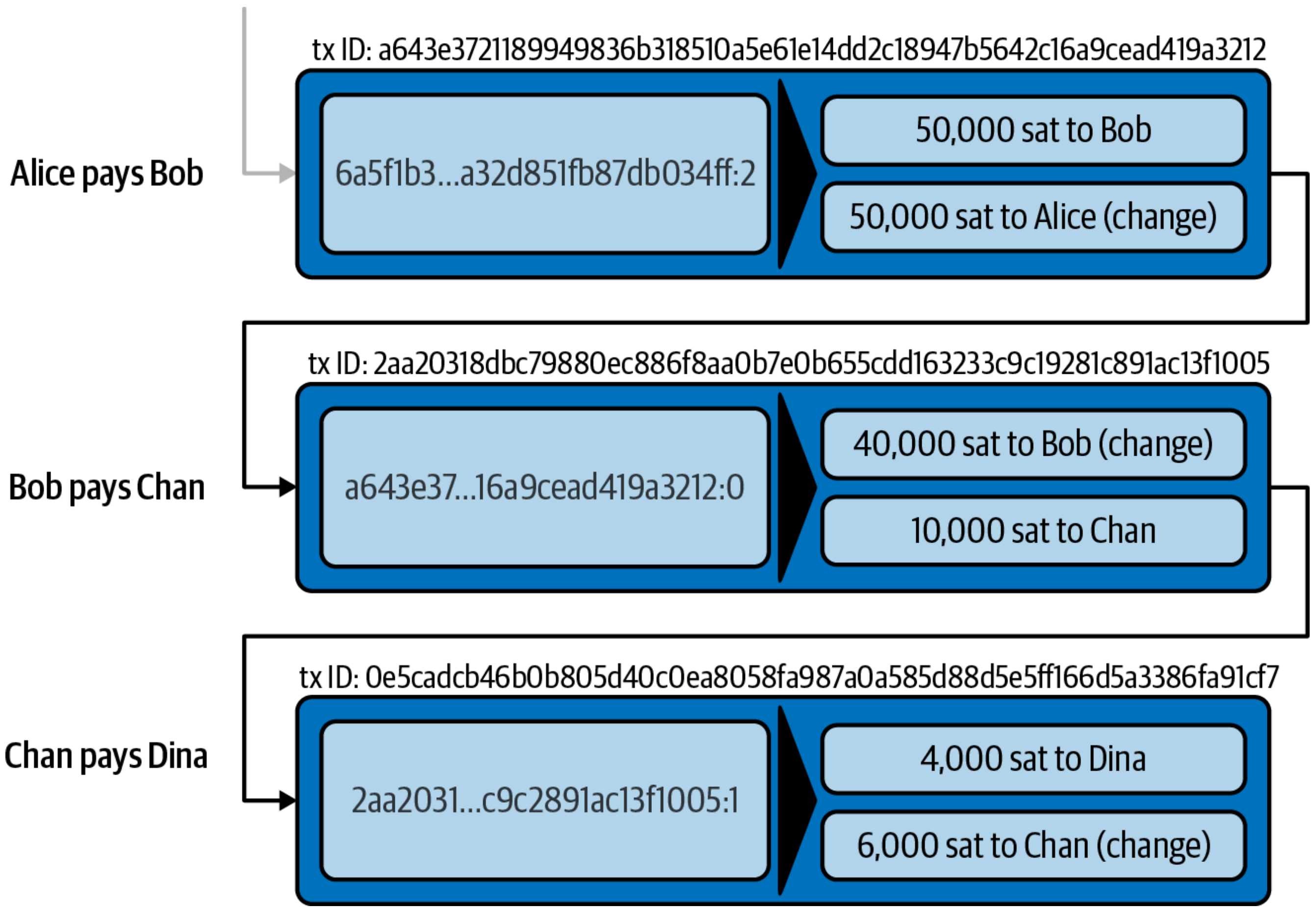

Alice pays Bob
tx ID: a643e3721189949836b318510a5e61e14dd2c18947b5642c16a9cead419a3212
6a5f1b3...a32d851fb87db034ff:2
50,000 sat to Bob
50,000 sat to Alice (change)
Bob pays Chan
tx ID: 2aa20318dbc79880ec886f8aa0b7e0b655cdd163233c9c19281c891ac13f1005
a643e37...16a9cead419a3212:0
40,000 sat to Bob (change)
10,000 sat to Chan
Chan pays Dina
tx ID: 0e5cadcb46b0b805d40c0ea8058fa987a0a585d88d5e5ff166d5a3386fa91cf7
2aa2031...c9c2891ac13f1005:1
4,000 sat to Dina
6,000 sat to Chan (change)

Figure 115. Transaction inputs refer to outpoints forming a chain

The input in Bob's transaction references Alice's transaction (by TxID) and the 0 indexed output.

The input in Chan's transaction references Bob's transaction's TxID and the first indexed output, because the payment to Chan is output #1. In Bob's payment to Chan, Bob's change is output #0.[12]

Now, if we look at Alice's payment to Bob, we can see that Alice is spending an outpoint that was the third (output index #2) output in a transaction whose ID is 6a5f1b3[…]. We don't see that referenced transaction in the diagram, but we can deduce these details from the outpoint.

Bitcoin Script

The final element of Bitcoin that is needed to complete our understanding is the scripting language that controls access to outpoints. So far, we've simplified the description by saying "Alice signs the transaction to pay Bob." Behind the scenes, however, there is some hidden complexity that makes it possible to implement more complex spending conditions. The simplest and most common spending condition is "present a signature matching the following public key." A spending condition like this is recorded in each output as *locking script* written in a scripting language called *Bitcoin Script*.

Bitcoin Script is an extremely simple stack-based scripting language. It does not contain loops or recursion and therefore is *Turing incomplete* (meaning it cannot express arbitrary complexity and has predictable execution). Those familiar with the (now ancient) programming language FORTH will recognize the syntax and style.

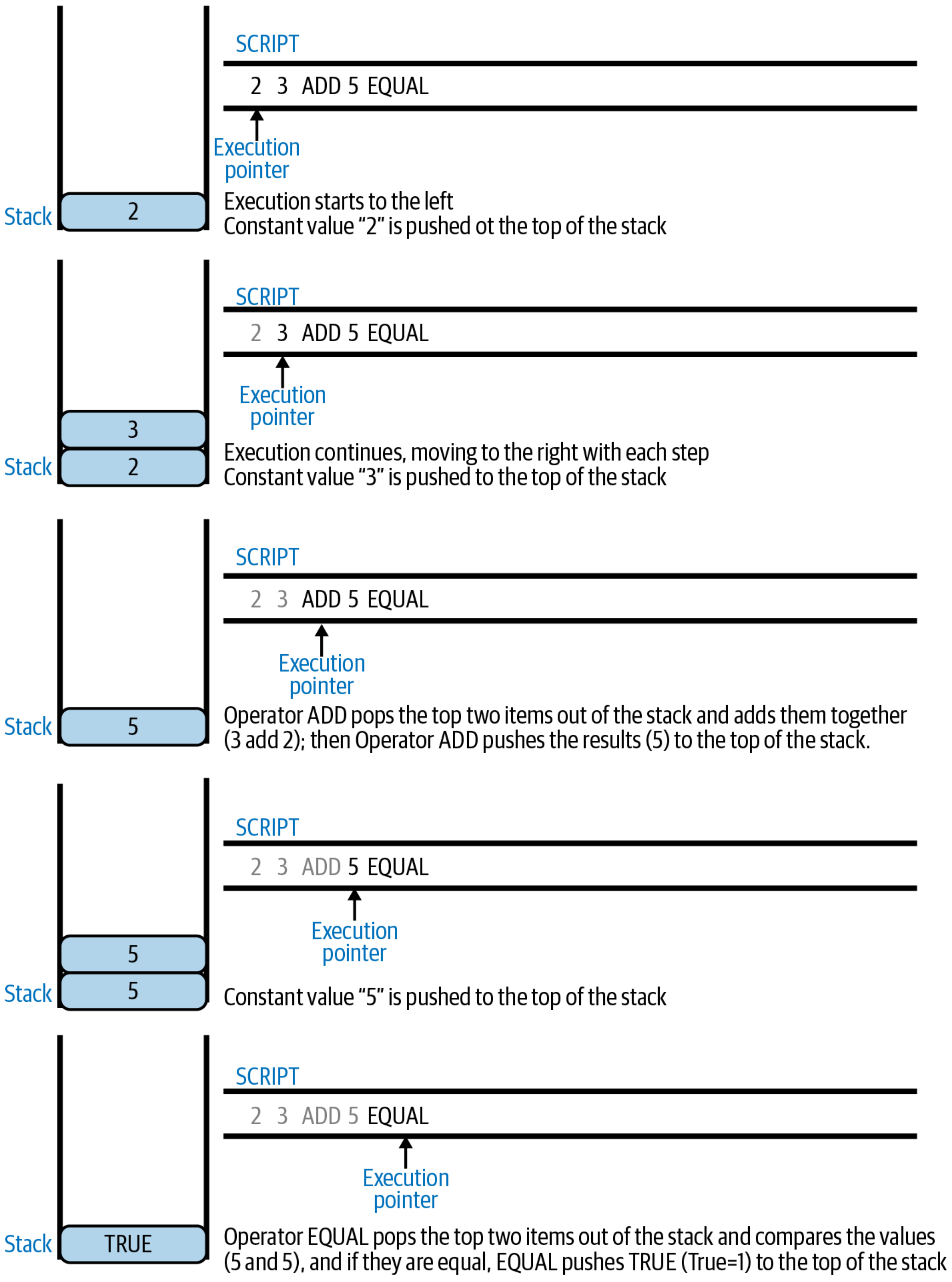
Running Bitcoin Script

In simple terms, the Bitcoin system evaluates Bitcoin Script by running the script on a stack; if the final result is TRUE, it considers the spending condition satisfied and the transaction valid.

Let's look at a very simple example of Bitcoin Script, which adds the numbers 2 and 3 and then compares the result to the number 5:

```
2 3 ADD 5 EQUAL
```

In Example of Bitcoin Script execution, we see how this script is executed (from left to right).


SCRIPT
2 3 ADD 5 EQUAL
Execution pointer
Stack
2
Execution starts to the left
Constant value "2" is pushed ot the top of the stack
SCRIPT
2 3 ADD 5 EQUAL
Execution pointer
Stack
3
2
Execution continues, moving to the right with each step
Constant value "3" is pushed to the top of the stack
SCRIPT
2 3 ADD 5 EQUAL
Execution pointer
Stack
5
Operator ADD pops the top two items out of the stack and adds them together (3 add 2); then Operator ADD pushes the results (5) to the top of the stack.
SCRIPT
2 3 ADD 5 EQUAL
Execution pointer
Stack
5
5
Constant value "5" is pushed to the top of the stack
SCRIPT
2 3 ADD 5 EQUAL
Execution pointer
Stack
TRUE
Operator EQUAL pops the top two items out of the stack and compares the values (5 and 5), and if they are equal, EQUAL pushes TRUE (True=1) to the top of the stack
Figure 116. Example of Bitcoin Script execution

Locking and Unlocking Scripts

Bitcoin Script is made up of two parts:

Locking scripts

These are embedded in transaction outputs, setting the conditions that must be fulfilled to spend that output. For example, Alice's wallet adds a locking script to the output paying Bob, that sets the condition that Bob's signature is required to spend it.

Unlocking scripts

These are embedded in transaction inputs, fulfilling the conditions set by the referenced output's locking script. For example, Bob can unlock the preceding output by providing an unlocking script containing a digital signature.

Using a simplified model, for validation, the unlocking script and locking script are concatenated and executed (P2SH and SegWit are exceptions). For example, if someone locked a transaction output with the locking script "3 ADD 5 EQUAL", we could spend it with the unlocking script "2" in a transaction input. Anyone validating that transaction would concatenate our unlocking script (2) and the locking script (3 ADD 5 EQUAL) and run the result through the Bitcoin Script execution engine. They would get TRUE and we would be able to spend the output.

Obviously, this simplified example would make a very poor choice for locking an actual Bitcoin output because there is no secret, just basic arithmetic. Anyone could spend the output by providing the answer "2." Most locking scripts therefore require demonstrating knowledge of a secret.

Locking to a Public Key (Signature)

The simplest form of a locking script is one that requires a signature. Let's consider Alice's transaction that pays Bob 50,000 satoshis. The output Alice creates to pay Bob will have a locking script requiring Bob's signature and would look like this:

A locking script that requires a digital signature from Bob's private key

```
<Bob Public Key> CHECKSIG
```

The operator `CHECKSIG` takes two items from the stack: a signature and a public key. As you can see, Bob's public key is in the locking script, so what is missing is the signature corresponding to that public key. This locking script can only be spent by Bob, because only Bob has the corresponding private key needed to produce a digital signature matching the public key.

To unlock this locking script, Bob would provide an unlocking script containing only his digital signature:

An unlocking script containing (only) a digital signature from Bob's private key

```
<Bob Signature>
```

In A transaction chain showing the locking script (output) and unlocking script (input) you can see the locking script in Alice's transaction (in the output that pays Bob) and the unlocking script (in the input that spends that output) in Bob's transaction.

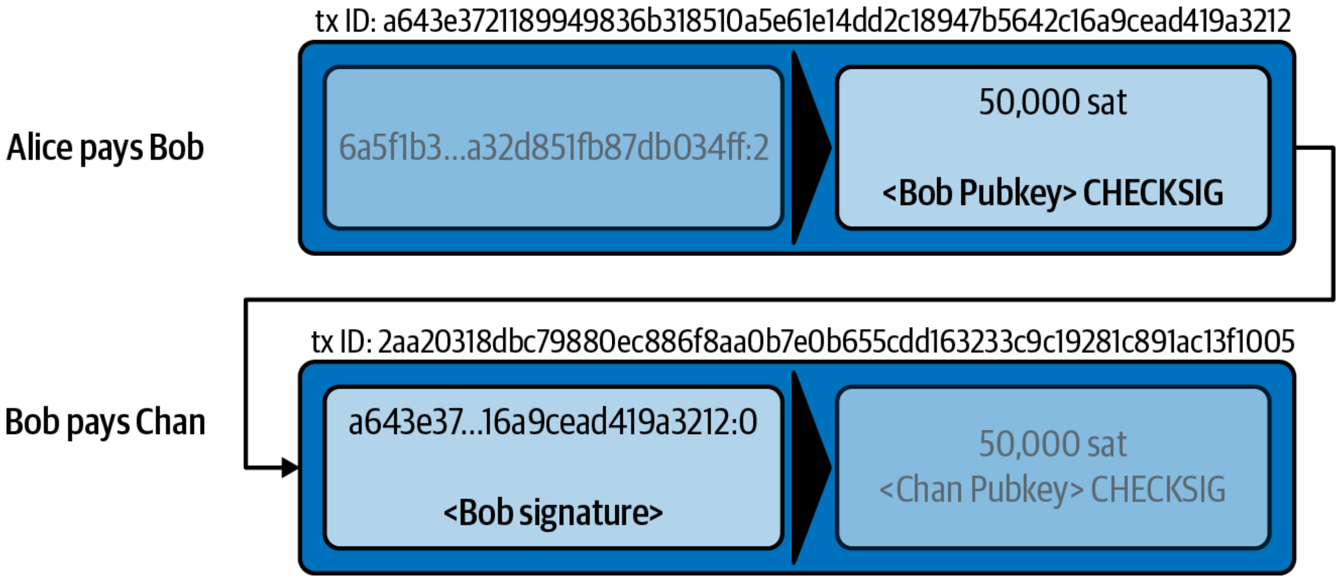

tx ID: a643e3721189949836b318510a5e61e14dd2c18947b5642c16a9cead419a3212
Alice pays Bob
6a5f1b3...a32d851fb87db034ff:2
50,000 sat
<Bob Pubkey> CHECKSIG
tx ID: 2aa20318dbc79880ec886f8aa0b7e0b655cdd163233c9c19281c891ac13f1005
Bob pays Chan
a643e37...16a9cead419a3212:0
<Bob signature>
50,000 sat
<Chan Pubkey> CHECKSIG

Figure 117. A transaction chain showing the locking script (output) and unlocking script (input)

To validate Bob's transaction, a Bitcoin node would do the following:

1. Extract the unlocking script from the input (<Bob Signature>).
2. Look up the outpoint it is attempting to spend (a643e37...3213:0). This is Alice's transaction and would be found on the blockchain.
3. Extract the locking script from that outpoint (<Bob PubKey> CHECKSIG).
4. Concatenate into one script, placing the unlocking script in front of the locking script (<Bob Signature> <Bob PubKey> CHECKSIG).
5. Execute this script on the Bitcoin Script execution engine to see what result is produced.
6. If the result is TRUE, deduce that Bob's transaction is valid because it was able to fulfill the spending condition to spend that outpoint.

Locking to a Hash (Secret)

Another type of locking script, one that is used in the Lightning Network, is a *hashlock*. To unlock it, you must know the secret *preimage* to the hash.

To demonstrate this, let's have Bob generate a random number R and keep it secret:

```
R = 1833462189
```

Now, Bob calculates the SHA-256 hash of this number:

```
H = SHA256(R) =>
H = SHA256(1833462189) =>
H = 0ffd8bea4abdb0deafd6f2a8ad7941c13256a19248a7b0612407379e1460036a
```

Now, Bob gives the hash H we calculated previously to Alice, but keeps the number R secret. Recall that because of the properties of cryptographic hashes, Alice can't "reverse" the hash calculation and guess the number R.

Alice creates an output paying 50,000 satoshi with the locking script:

```
HASH256 H EQUAL
```

where H is the actual hash value (0ffd8...036a) that Bob gave to Alice.

Let's explain this script:

The HASH256 operator pops a value from the stack and calculates the SHA-256 hash of that value. Then it pushes the result onto the stack.

The H value is pushed onto the stack, and then the EQUAL operator checks if the two values are the same and pushes TRUE or FALSE onto the stack accordingly.

Therefore, this locking script will only work if it is combined with an unlocking script that contains R, so that when concatenated, we have:

```
R HASH256 H EQUAL
```

Only Bob knows R, so only Bob can produce a transaction with an unlocking script revealing the secret value R.

Interestingly, Bob can give the R value to anyone else, who can then spend that Bitcoin. This makes the secret value R almost like a bitcoin "voucher," since anyone who has it can spend the output Alice created. We'll see how this is a useful property for the Lightning Network!

Multisignature Scripts

The Bitcoin Script language provides a multisignature building block (primitive), that can be used to build escrow services and complex ownership configurations between several stakeholders. An arrangement that requires multiple signatures to spend Bitcoin is called a *multisignature scheme*, further specified as a *K-of-N* scheme, where:

- *N* is the total number of signers identified in the multisignature scheme, and
- *K* is the *quorum* or *threshold*: the minimum number of signatures to authorize spending.

The script for an *K*-of-*N* multisignature is:

```
K <PubKey1> <PubKey2> ... <PubKeyN> N CHECKMULTISIG
```

where *N* is the total number of listed public keys (Public Key 1 through Public Key *N*) and *K* is the threshold of required signatures to spend the output.

The Lightning Network uses a 2-of-2 multisignature scheme to build a payment channel. For example, a payment channel between Alice and Bob would be built on a 2-of-2 multisignature like this:

```
2 <PubKey Alice> <PubKey Bob> 2 CHECKMULTISIG
```

The preceding locking script can be satisfied with an unlocking script containing a pair of signatures:[13]

```
0 <Sig Alice> <Sig Bob>
```

The two scripts together would form the combined validation script:

```
0 <Sig Alice> <Sig Bob> 2 <PubKey Alice> <PubKey Bob> 2 CHECKMULTISIG
```

A multisignature locking script can be represented by a Bitcoin address, encoding the hash of the locking script. For example, the initial funding transaction of a Lightning payment channel is a transaction that pays to an address that encodes a locking script of a 2-of-2 multisig of the two channel partners.

Timelock Scripts

Another important building block that exists in Bitcoin and is used extensively in the Lightning Network is a *timelock*. A timelock is a restriction on spending that requires that a certain time or block height has elapsed before spending is allowed. It is a bit like a postdated check drawn from a bank account that can't be cashed before the date on the check.

Bitcoin has two levels of timelocks: transaction-level timelocks and output-level timelocks.

A *transaction-level timelock* is recorded in the transaction `nLockTime` field of the transaction and prevents the entire transaction from being accepted before the timelock has passed. Transaction-level timelocks are the most commonly used timelock mechanism in Bitcoin today.

An *output-level timelock* is created by a script operator. There are two types of output timelocks: absolute timelocks and relative timelocks.

Output-level *absolute timelocks* are implemented by the operator CHECKLOCKTIMEVERIFY, which is often shortened in conversation as *CLTV*. Absolute timelocks implement a time constraint with an absolute timestamp or blockheight, expressing the equivalent of "not spendable before block 800,000."

Output-level *relative timelocks* are implemented by the operator CHECKSEQUENCEVERIFY, often shortened in conversation as *CSV*. Relative timelocks implement a spending constraint that is relative to the confirmation of the transaction, expressing the equivalent of "can't be spent until 1,024 blocks after confirmation."

Scripts with Multiple Conditions

One of the more powerful features of Bitcoin Script is flow control, also known as conditional clauses. You are probably familiar with flow control in various programming languages that use the construct IF...THEN...ELSE. Bitcoin conditional clauses look a bit different, but are essentially the same construct.

At a basic level, bitcoin conditional opcodes allow us to construct a locking script that has two ways of being unlocked, depending on a TRUE/FALSE outcome of evaluating a logical condition. For example, if x is TRUE, the locking script is A ELSE the locking script is B.

Additionally, bitcoin conditional expressions can be *nested* indefinitely, meaning that a conditional clause can contain another within it, which contains another, etc. Bitcoin Script flow control can be used to construct very complex scripts with hundreds or even thousands of possible execution paths. There is no limit to nesting, but consensus rules impose a limit on the maximum size, in bytes, of a script.

Bitcoin implements flow control using the IF, ELSE, ENDIF, and NOTIF opcodes. Additionally, conditional expressions can contain boolean operators such as BOOLAND, `BOOLOR`, and NOT.

At first glance, you may find Bitcoin's flow control scripts confusing. That is because Bitcoin Script is a stack language. The same way that the arithmetic operation `1 + 1` looks "backward" when expressed in Bitcoin Script as 1 1 ADD, flow control clauses in Bitcoin also look "backward."

In most traditional (procedural) programming languages, flow control looks like this:

Pseudocode of flow control in most programming languages

```
if (condition):
  code to run when condition is true
else:
  code to run when condition is false
code to run in either case
```

In a stack-based language like Bitcoin Script, the logical condition comes *before* the IF, which makes it look "backward," like this:

Bitcoin Script flow control

```
condition
IF
  code to run when condition is true
ELSE
  code to run when condition is false
ENDIF
code to run in either case
```

When reading Bitcoin Script, remember that the condition being evaluated comes *before* the IF opcode.

Using Flow Control in Scripts

A very common use for flow control in Bitcoin Script is to construct a locking script that offers multiple execution paths, each a different way of redeeming the UTXO.

Let’s look at a simple example, where we have two signers, Alice and Bob, and either one is able to redeem. With multisig, this would be expressed as a 1-of-2 multisig script. For the sake of demonstration, we will do the same thing with an IF clause:

```
IF
 <Alice's Pubkey> CHECKSIG
ELSE
 <Bob's Pubkey> CHECKSIG
ENDIF
```

Looking at this locking script, you may be wondering: "Where is the condition? There is nothing preceding the IF clause!"

The condition is not part of the locking script. Instead, the condition will be *offered in the unlocking script*, allowing Alice and Bob to "choose" which execution path they want.

Alice redeems this with the unlocking script:

```
<Alice's Sig> 1
```

The 1 at the end serves as the condition (TRUE) that will make the IF clause execute the first redemption path for which Alice has a signature.

For Bob to redeem this, he would have to choose the second execution path by giving a FALSE value to the IF clause:

```
<Bob's Sig> 0
```

Bob’s unlocking script puts a 0 on the stack, causing the IF clause to execute the second (ELSE) script, which requires Bob’s signature.

Because each of the two conditions also requires a signature, Alice can’t use the second clause and Bob can’t use the first clause; they don’t have the necessary signatures for that!

Since conditional flows can be nested, so can the TRUE / FALSE values in the unlocking script, to navigate a complex path of conditions.

In A complex script used in the Lightning Network you can see an example of the kind of complex script that is used in the Lightning Network, with multiple conditions.[14] The scripts used in the Lightning Network are highly optimized and compact, to minimize the on-chain footprint, so they are not easy to read and understand. Nevertheless, see if you can identify some of the Bitcoin Script concepts we learned about in this chapter.

Example 9. A complex script used in the Lightning Network

```
# To remote node with revocation key
DUP HASH160 <RIPEMD160(SHA256(revocationpubkey))> EQUAL
IF
    CHECKSIG
ELSE
    <remote_htlcpubkey> SWAP SIZE 32 EQUAL
    NOTIF
        # To local node via HTLC-timeout transaction (timelocked).
        DROP 2 SWAP <local_htlcpubkey> 2 CHECKMULTISIG
    ELSE
        # To remote node with preimage.
        HASH160 <RIPEMD160(payment_hash)> EQUALVERIFY
        CHECKSIG
    ENDIF
ENDIF
```

Appendix B: Docker Basic Installation and Use

This book contains a number of examples that run inside Docker containers for standardization across different operating systems.

This section will help you install Docker and familiarize yourself with some of the most commonly used Docker commands, so that you can run the book's example containers.

Installing Docker

Before we begin, you should install the Docker container system on your computer. Docker is an open system that is distributed for free as a *Community Edition* for many different operating systems including Windows, macOS, and Linux. The Windows and Macintosh versions are called *Docker Desktop* and consist of a GUI desktop application and command-line tools. The Linux version is called *Docker Engine* and is comprised of a server daemon and command-line tools. We will be using the command-line tools, which are identical across all platforms.

Go ahead and install Docker for your operating system by following the instructions to "Get Docker" from the Docker website.

Select your operating system from the list and follow the installation instructions.

TIP If you install on Linux, follow the post-installation instructions to ensure you can run Docker as a regular user instead of user root. Otherwise, you will need to prefix all docker commands with sudo, running them as root like: sudo docker.

Once you have Docker installed, you can test your installation by running the demo container hello-world like this:

```
$ docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

[...]
```

Basic Docker Commands

In this appendix, we use Docker quite extensively. We will be using the following Docker commands and arguments.

Building a Container

```
<pre data-type="programlisting">docker build [-t <em>tag</em>]
[<em>directory</em>]</pre>
```

__tag__ is how we identify the container we are building, and __directory__ is where the container's context (folders and files) and definition file (Dockerfile) are found.

Running a Container

```
<pre data-type="programlisting">docker run -it [--network <em>netname</em>] [--name
<em>cname</em>] <em>tag</em></pre>
```

__netname__ is the name of a Docker network, __cname__ is the name we choose for this container instance, and __tag__ is the name tag we gave the container when we built it.

Executing a Command in a Container

```
<pre data-type="programlisting">docker exec <em>cname command</em></pre>
```

__cname__ is the name we gave the container in the run command, and __command__ is an executable or script that we want to run inside the container.

Stopping and Starting a Container

In most cases, if we are running a container in an *interactive* as well as *terminal* mode, i.e., with the i and t flags (combined as -it) set, the container can be stopped by simply pressing Ctrl-C or by exiting the shell with exit or Ctrl-D. If a container does not terminate, you can stop it from another terminal like this:

```
<pre data-type="programlisting">docker stop <em>cname</em></pre>
```

To resume an already existing container, use the `start` command like so:

```
<pre data-type="programlisting">docker start <em>cname</em></pre>
```

Deleting a Container by Name

If you name a container instead of letting Docker name it randomly, you cannot reuse that name until the container is deleted. Docker will return an error like this:

```
docker: Error response from daemon: Conflict. The container name "/bitcoind" is
already in use...
```

To fix this, delete the existing instance of the container:

```
<pre data-type="programlisting">docker rm <em>cname</em></pre>
```

__cname__ is the name assigned to the container (bitcoind in the example error message).

Listing Running Containers

```
docker ps
```

This command shows the current running containers and their names.

Listing Docker Images

```
docker image ls
```

This command shows the Docker images that have been built or downloaded on your computer.

Conclusion

These basic Docker commands will be enough to get you started and will allow you to run all the examples in this book.

Appendix C: Wire Protocol Messages

This appendix lists all the currently defined message types used in the Lightning P2P protocol. Additionally, we show the structure of each message, grouping the messages into logical groupings based on the protocol flows.

NOTE Lightning Protocol messages are extensible and their structure may change during network-wide upgrades. For the authoritative information, consult the latest version of the BOLTs found in the GitHub Lightning-RFC repository.

Message Types

Currently defined message types are listed in Message types.

Table 11. Message types

Type integer	Message name	Category
16	`init`	Connection Establishment
17	`error`	Error Communication
18	`ping`	Connection Liveness
19	`pong`	Connection Liveness
32	`open_channel`	Channel Funding
33	`accept_channel`	Channel Funding
34	`funding_created`	Channel Funding
35	`funding_signed`	Channel Funding
36	`funding_locked`	Channel Funding + Channel Operation
38	`shutdown`	Channel Closing
39	`closing_signed`	Channel Closing
128	`update_add_htlc`	Channel Operation
130	`update_fulfill_hltc`	Channel Operation
131	`update_fail_htlc`	Channel Operation
132	`commit_sig`	Channel Operation
133	`revoke_and_ack`	Channel Operation
134	`update_fee`	Channel Operation
135	`update_fail_malformed_htlc`	Channel Operation
136	`channel_reestablish`	Channel Operation
256	`channel_announcement`	Channel Announcement
257	`node_announcement`	Channel Announcement

Type integer	Message name	Category
258	`channel_update`	Channel Announcement
259	`announce_signatures`	Channel Announcement
261	`query_short_chan_ids`	Channel Graph Syncing
262	`reply_short_chan_ids_end`	Channel Graph Syncing
263	`query_channel_range`	Channel Graph Syncing
264	`reply_channel_range`	Channel Graph Syncing
265	`gossip_timestamp_range`	Channel Graph Syncing

In High-level message types, the `Category` field allows us to quickly categorize a message based on its functionality within the protocol itself. At a high level, we place a message into one of eight (nonexhaustive) buckets including:

Connection Establishment

Sent when a peer-to-peer connection is first established. Also used to negotiate the set of features supported by a new connection.

Error Communication

Used by peers to communicate the occurrence of protocol level errors to each other.

Connection Liveness

Used by peers to check that a given transport connection is still live.

Channel Funding

Used by peers to create a new payment channel. This process is also known as the channel funding process.

Channel Operation

The act of updating a given channel off-chain. This includes sending and receiving payments, as well as forwarding payments within the network.

Channel Announcement

The process of announcing a new public channel to the wider network so it can be used for routing purposes.

Channel Graph Syncing

The process of downloading and verifying the channel graph.

Notice how messages that belong to the same category typically share an adjacent *message type* as well. This is done on purpose to group semantically similar messages together within the specification itself.

Message Structure

We now detail each message category in order to define the precise structure and semantics of all defined messages within the LN protocol.

Connection Establishment Messages

Messages in this category are the very first message sent between peers once they establish a transport connection. At the time of writing this chapter, there exists only a single message within this category, the `init` message. The `init` message is sent by *both* sides of the connection once it has been first established. No other messages are to be sent before the `init` message has been sent by both parties.

The init message

The structure of the `init` message is defined as follows:

- Type: 16
- Fields:
  - `uint16`: `global_features_len`
  - `global_features_len*byte`: `global_features`
  - `uint16`: `features_len`
  - `features_len*byte`: `features`
  - `tlv_stream_tlvs`

Structurally, the `init` message is composed of two variable size bytes slices that each store a set of *feature bits*. As we see in Feature Bits and Protocol Extensibility, feature bits are a primitive used within the protocol to advertise the set of protocol features a node either understands (optional features) or demands (required features).

Note that modern node implementations will only use the `features` field, with items residing within the `global_features` vector for primarily *historical* purposes (backward compatibility).

What follows after the core message is a series of Type-Length-Value (TLV) records that can be used to extend the message in a forward- and backward-compatible manner in the future. We'll cover what TLV records are and how they're used later in this appendix.

An `init` message is then examined by a peer to determine if the connection is well-defined based on the set of optional and required feature bits advertised by both sides.

An optional feature means that a peer knows about a feature, but they don't consider it critical to the operation of a new connection. An example of one would be something like the ability to understand the semantics of a newly added field to an existing message.

On the other hand, required features indicate that if the other peer doesn't know about the feature, then the connection isn't well defined. An example of such a feature would be a theoretical new channel type within the protocol: if your peer doesn't know of this feature, then you don't want to keep the connection because they're unable to open your new preferred channel type.

Error Communication Messages

Messages in this category are used to send connection level errors between two peers. Another type of error exists in the protocol: an HTLC forwarding level error. Connection level errors may signal things like feature bit incompatibility or the intent to *force close* (unilaterally broadcast the latest signed commitment).

The error message

The sole message in this category is the `error` message.

- Type: 17
- Fields:
  - `channel_id` : `chan_id`
  - `uint16` : `data_len`
  - `data_len*byte` : `data`

An `error` message can be sent within the scope of a particular channel by setting the `channel_id` to the `channel_id` of the channel undergoing this new error state. Alternatively, if the error applies to the connection in general, then the `channel_id` field should be set to all zeroes. This all zero `channel_id` is also known as the connection level identifier for an error.

Depending on the nature of the error, sending an `error` message to a peer you have a channel with may indicate that the channel cannot continue without manual intervention, so the only option at that point is to force close the channel by broadcasting the latest commitment state of the channel.

Connection Liveness

Messages in this section are used to probe to determine if a connection is still live or not. Because the LN protocol somewhat abstracts over the underlying transport being used to transmit the messages, a set of protocol level `ping` and `pong` messages are defined.

The ping message

- Type: 18
- Fields:
  - `uint16` : `num_pong_bytes`
  - `uint16` : `ping_body_len`
  - `ping_body_len*bytes` : `ping_body`

Next its companion, the `pong` message.

The pong message

- Type: 19
- Fields:

    - uint16 : pong_body_len
    - ping_body_len*bytes : pong_body

A ping message can be sent by either party at any time.

The ping message includes a num_pong_bytes field that is used to instruct the receiving node with respect to how large the payload it sends in its pong message is. The ping message also includes a ping_body opaque set of bytes which can be safely ignored. It only serves to allow a sender to pad out ping messages they send, which can be useful in attempting to thwart certain de-anonymization techniques based on packet sizes on the wire.

A pong message should be sent in response to a received ping message. The receiver should read a set of num_pong_bytes random bytes to send back as the pong_body field. Clever use of these fields/messages may allow a privacy conscious routing node to attempt to thwart certain classes of network de-anonymization attempts because they can create a "fake" transcript that resembles other messages based on the packet sizes sent across. Remember that by default the Lightning Network uses an *encrypted* transport, so a passive network monitor cannot read the plain-text bytes and thus only has timing and packet sizes to go off of.

Channel Funding

As we go on, we enter into the territory of the core messages that govern the functionality and semantics of the Lightning Protocol. In this section, we explore the messages sent during the process of creating a new channel. We'll only describe the fields used, as we leave an in-depth analysis of the funding process to Payment Channels.

Messages that are sent during the channel funding flow belong to the following set of five messages: open_channel, accept_channel, funding_created, funding_signed, and funding_locked.

The detailed protocol flow using these messages is described in Payment Channels.

The open_channel message

- Type: 32
- Fields:
    - chain_hash : chain_hash
    - 32*byte : temp_chan_id
    - uint64 : funding_satoshis
    - uint64 : push_msat
    - uint64 : dust_limit_satoshis
    - uint64 : max_htlc_value_in_flight_msat
    - uint64 : channel_reserve_satoshis
    - uint64 : htlc_minimum_msat
    - uint32 : feerate_per_kw
    - uint16 : to_self_delay

    - `uint16` : `max_accepted_htlcs`
    - `pubkey` : `funding_pubkey`
    - `pubkey` : `revocation_basepoint`
    - `pubkey` : `payment_basepoint`
    - `pubkey` : `delayed_payment_basepoint`
    - `pubkey` : `htlc_basepoint`
    - `pubkey` : `first_per_commitment_point`
    - `byte` : `channel_flags`
    - `tlv_stream` : `tlvs`

This is the first message sent when a node wishes to execute a new funding flow with another node. This message contains all the necessary information required for both peers to construct both the funding transaction as well as the commitment transaction.

At the time of writing this chapter, a single TLV record is defined within the set of optional TLV records that may be appended to the end of a defined message:

- Type: 0
- Data: `upfront_shutdown_script`

The `upfront_shutdown_script` is a variable-sized byte slice that must be a valid public key script as accepted by the Bitcoin network's consensus algorithm. By providing such an address, the sending party is able to effectively create a "closed loop" for their channel, as neither side will sign off an cooperative closure transaction that pays to any other address. In practice, this address is usually one derived from a cold storage wallet.

The `channel_flags` field is a bitfield of which, at the time of writing, only the *first* bit has any sort of significance. If this bit is set, then this channel is to be advertised to the public network as a routable channel. Otherwise, the channel is considered to be unadvertised, also commonly referred to as a private channel.

The accept_channel message

The `accept_channel` message is the response to the `open_channel` message.

- Type: 33
- Fields:
    - `32*byte` : `temp_chan_id`
    - `uint64` : `dust_limit_satoshis`
    - `uint64` : `max_htlc_value_in_flight_msat`
    - `uint64` : `channel_reserve_satoshis`
    - `uint64` : `htlc_minimum_msat`
    - `uint32` : `minimum_depth`

    - `uint16` : `to_self_delay`
    - `uint16` : `max_accepted_htlcs`
    - `pubkey` : `funding_pubkey`
    - `pubkey` : `revocation_basepoint`
    - `pubkey` : `payment_basepoint`
    - `pubkey` : `delayed_payment_basepoint`
    - `pubkey` : `htlc_basepoint`
    - `pubkey` : `first_per_commitment_point`
    - `tlv_stream` : `tlvs`

The `accept_channel` message is the second message sent during the funding flow process. It serves to acknowledge an intent to open a channel with a new remote peer. The message mostly echoes the set of parameters that the responder wishes to apply to their version of the commitment transaction. In Payment Channels, when we go into the funding process in detail, we explore the implications of the various parameters that can be set when opening a new channel.

The funding_created message

In response, the initiator will send the `funding_created` message.

- Type: 34
- Fields:
    - `32*byte` : `temp_chan_id`
    - `32*byte` : `funding_txid`
    - `uint16` : `funding_output_index`
    - `sig` : `commit_sig`

Once the initiator of a channel receives the `accept_channel` message from the responder, they have all the materials they need to construct the commitment transaction, as well as the funding transaction. As channels by default are single funder (only one side commits funds), only the initiator needs to construct the funding transaction. As a result, to allow the responder to sign a version of a commitment transaction for the initiator, the initiator only needs to send the funding outpoint of the channel.

The funding_signed message

To conclude, the responder sends the `funding_signed` message.

- Type: 34
- Fields:
    - `channel_id` : `channel_id`
    - `sig` : `signature`

To conclude after the responder receives the `funding_created` message, they now own a valid signature of the commitment transaction by the initiator. With this signature they're able to exit the channel at any time by signing their half of the multisig funding output and broadcasting the transaction. This is referred to as a force close. Conversely, to give the initiator the ability to close the channel, the responder also signs the initiator's commitment transaction.

Once this message has been received by the initiator, it's safe for them to broadcast the funding transaction because they're now able to exit the channel agreement unilaterally.

The funding_locked message

Once the funding transaction has received enough confirmations, the `funding_locked` message is sent.

- Type: 36
- Fields:
  - `channel_id` : `channel_id`
  - `pubkey` : `next_per_commitment_point`

Once the funding transaction obtains a `minimum_depth` number of confirmations, then the `funding_locked` message is to be sent by both sides. Only after this message has been received and sent can the channel begin to be used.

Channel Closing

Channel closing is a multistep process. One node initiates by sending the `shutdown` message. The two channel partners then exchange a series of `channel_closing` messages to negotiate mutually acceptable fees for the closing transaction. The channel funder sends the first `closing_signed` message, and the other side can accept by sending a `closing_signed` message with the same fee values.

The shutdown message

- Type: 38
- Fields:
  - `channel_id` : `channel_id`
  - `u16` : `len`
  - `len*byte` : `scriptpubkey`

The closing_signed message

- Type: 39
- Fields:
  - `channel_id` : `channel_id`
  - `u64` : `fee_satoshis`

    - `signature` : `signature`

Channel Operation

In this section, we briefly describe the set of messages used to allow nodes to operate a channel. By operation, we mean being able to send, receive, and forward payments for a given channel.

To send, receive, or forward a payment over a channel, an HTLC must first be added to both commitment transactions that comprise a channel link.

The update_add_htlc message

The `update_add_htlc` message allows either side to add a new HTLC to the opposite commitment transaction.

- Type: 128
- Fields:
    - `channel_id` : `channel_id`
    - `uint64` : `id`
    - `uint64` : `amount_msat`
    - `sha256` : `payment_hash`
    - `uint32` : `cltv_expiry`
    - `1366*byte` : `onion_routing_packet`

Sending this message allows one party to initiate either sending a new payment or forwarding an existing payment that arrived via an incoming channel. The message specifies the amount (`amount_msat`) along with the payment hash that unlocks the payment itself. The set of forwarding instructions of the next hop are onion encrypted within the `onion_routing_packet` field. In Onion Routing, on multihop HTLC forwarding, we cover the onion routing protocol used in the Lightning Network in detail.

Note that each HTLC sent uses an automatically incrementing ID which is used by any message which modifies an HTLC (settle or cancel) to reference the HTLC in a unique manner scoped to the channel.

The update_fulfill_hltc message

The `update_fulfill_hltc` message allows redemption (receipt) of an active HTLC.

- Type: 130
- Fields:
    - `channel_id` : `channel_id`
    - `uint64` : `id`
    - `32*byte` : `payment_preimage`

This message is sent by the HTLC receiver to the proposer to redeem an active HTLC. The message references the `id` of the HTLC in question, and also provides the preimage (which unlocks the HLTC).

The update_fail_htlc message

The `update_fail_htlc` message is sent to remove an HTLC from a commitment transaction.

- Type: 131
- Fields:
  - `channel_id` : `channel_id`
  - `uint64` : `id`
  - `uint16` : `len`
  - `len*byte` : `reason`

The `update_fail_htlc` message is the opposite of the `update_fulfill_hltc` message in that it allows the receiver of an HTLC to remove the very same HTLC. This message is typically sent when an HTLC cannot be properly routed upstream and needs to be sent back to the sender to unravel the HTLC chain. As we explore in Failure Messages, the message contains an *encrypted* failure reason (`reason`) which may allow the sender to either adjust their payment route or terminate if the failure itself is a terminal one.

The commitment_signed message

The `commitment_signed` message is used to stamp the creation of a new commitment transaction.

- Type: 132
- Fields:
  - `channel_id` : `channel_id`
  - `sig` : `signature`
  - `uint16` : `num_htlcs`
  - `num_htlcs*sig` : `htlc_signature`

In addition to sending a signature for the next commitment transaction, the sender of this message also needs to send a signature for each HTLC that's present on the commitment transaction.

The revoke_and_ack message

The `revoke_and_ack` is sent to revoke a dated commitment.

- Type: 133
- Fields:
  - `channel_id` : `channel_id`
  - `32*byte` : `per_commitment_secret`
  - `pubkey` : `next_per_commitment_point`

Because the Lightning Network uses a replace-by-revoke commitment transaction, after receiving a new commitment transaction via the `commit_sig` message, a party must revoke their past commitment before they're able to receive another one. While revoking a commitment transaction, the revoker then also provides the next commitment point that's required to allow the other party to send them a new commitment state.

The update_fee message

The `update_fee` is sent to update the fee on the current commitment transactions.

- Type: 134
- Fields:
  - `channel_id` : `channel_id`
  - `uint32` : `feerate_per_kw`

This message can only be sent by the initiator of the channel; they're the ones that will pay for the commitment fee of the channel as along as it's open.

The update_fail_malformed_htlc message

The `update_fail_malformed_htlc` message is sent to remove a corrupted HTLC.

- Type: 135
- Fields:
  - `channel_id` : `channel_id`
  - `uint64` : `id`
  - `sha256` : `sha256_of_onion`
  - `uint16` : `failure_code`

This message is similar to the `update_fail_htlc` message, but it's rarely used in practice. As mentioned previously, each HTLC carries an onion encrypted routing packet that also covers the integrity of portions of the HTLC itself. If a party receives an onion packet that has somehow been corrupted along the way, then it won't be able to decrypt the packet. As a result, it also can't properly forward the HTLC; therefore, it'll send this message to signify that the HTLC has been corrupted somewhere along the route back to the sender.

Channel Announcement

Messages in this category are used to announce components of the channel graph authenticated data structure to the wider network. The channel graph has a series of unique properties due to the condition that all data added to the channel graph must also be anchored in the base Bitcoin blockchain. As a result, to add a new entry to the channel graph, an agent must be an on-chain transaction fee. This serves as a natural spam deterrent for the Lightning Network.

The channel_announcement message

The `channel_announcement` message is used to announce a new channel to the wider network.

- Type: 256
- Fields:
  - `sig` : `node_signature_1`
  - `sig` : `node_signature_2`
  - `sig` : `bitcoin_signature_1`
  - `sig` : `bitcoin_signature_2`
  - `uint16` : `len`
  - `len*byte` : `features`
  - `chain_hash` : `chain_hash`
  - `short_channel_id` : `short_channel_id`
  - `pubkey` : `node_id_1`
  - `pubkey` : `node_id_2`
  - `pubkey` : `bitcoin_key_1`
  - `pubkey` : `bitcoin_key_2`

The series of signatures and public keys in the message serves to create a *proof* that the channel actually exists within the base Bitcoin blockchain. As we detail in The short channel ID, each channel is uniquely identified by a locator that encodes its *location* within the blockchain. This locator is called this `short_channel_id` and can fit into a 64-bit integer.

The node_announcement message

The `node_announcement` message allows a node to announce/update its vertex within the greater channel graph.

- Type: 257
- Fields:
  - `sig` : `signature`
  - `uint64` : `flen`
  - `flen*byte` : `features`
  - `uint32` : `timestamp`
  - `pubkey` : `node_id`
  - `3*byte` : `rgb_color`
  - `32*byte` : `alias`
  - `uint16` : `addrlen`
  - `addrlen*byte` : `addresses`

Note that if a node doesn't have any advertised channel within the channel graph, then this message is ignored to ensure that adding an item to the channel graph bears an on-chain cost. In this case, the on-chain cost will be the cost of creating the channel to which this node is connected.

In addition to advertising its feature set, this message also allows a node to announce/update the set of network `addresses` where it can be reached.

The channel_update message

The `channel_update` message is sent to update the properties and policies of an active channel edge within the channel graph.

- Type: 258
- Fields:
  - `signature` : `signature`
  - `chain_hash` : `chain_hash`
  - `short_channel_id` : `short_channel_id`
  - `uint32` : `timestamp`
  - `byte` : `message_flags`
  - `byte` : `channel_flags`
  - `uint16` : `cltv_expiry_delta`
  - `uint64` : `htlc_minimum_msat`
  - `uint32` : `fee_base_msat`
  - `uint32` : `fee_proportional_millionths`
  - `uint16` : `htlc_maximum_msat`

In addition to being able to enable/disable a channel, this message allows a node to update its routing fees as well as other fields that shape the type of payment that is permitted to flow through this channel.

The announce_signatures message

The `announce_signatures` message is exchanged by channel peers to assemble the set of signatures required to produce a `channel_announcement` message.

- Type: 259
- Fields:
  - `channel_id` : `channel_id`
  - `short_channel_id` : `short_channel_id`
  - `sig` : `node_signature`
  - `sig` : `bitcoin_signature`

After the `funding_locked` message has been sent, if both sides wish to advertise their channel to the network, then they'll each send the `announce_signatures` message which allows both sides to emplace the four signatures required to generate an `announce_signatures` message.

Channel Graph Syncing

The `query_short_chan_ids` message allows a peer to obtain the channel information related to a series of short channel IDs.

The query_short_chan_ids message

- Type: 261
- Fields:
  - `chain_hash` : `chain_hash`
  - `u16` : `len`
  - `len*byte` : `encoded_short_ids`
  - `query_short_channel_ids_tlvs` : `tlvs`

As we learn in Gossip and the Channel Graph, these channel IDs may be a series of channels that were new to the sender or were out-of-date, which allows the sender to obtain the latest set of information for a set of channels.

The reply_short_chan_ids_end message

The `reply_short_chan_ids_end` message is sent after a peer finishes responding to a prior `query_short_chan_ids` message.

- Type: 262
- Fields:
  - `chain_hash` : `chain_hash`
  - `byte` : `full_information`

This message signals to the receiving party that if they wish to send another query message, they can now do so.

The query_channel_range message

The `query_channel_range` message allows a node to query for the set of channels opened within a block range.

- Type: 263
- Fields:
  - `chain_hash` : `chain_hash`
  - `u32` : `first_blocknum`
  - `u32` : `number_of_blocks`
  - `query_channel_range_tlvs` : `tlvs`

As channels are represented using a short channel ID that encodes the location of a channel in the chain, a node on the network can use a block height as a sort of *cursor* to seek through the chain in order to discover a set of newly opened channels.

The reply_channel_range message

The `reply_channel_range` message is the response to the `query_channel_range` message and includes the set of short channel IDs for known channels within that range.

- Type: 264
- Fields:
  - `chain_hash` : `chain_hash`
  - `u32` : `first_blocknum`
  - `u32` : `number_of_blocks`
  - `byte` : `sync_complete`
  - `u16` : `len`
  - `len*byte` : `encoded_short_ids`
  - `reply_channel_range_tlvs` : `tlvs`

As a response to `query_channel_range`, this message sends back the set of channels that were opened within that range. This process can be repeated with the requester advancing their cursor further down the chain to continue syncing the channel graph.

The gossip_timestamp_range message

The `gossip_timestamp_range` message allows a peer to start receiving new incoming gossip messages on the network.

- Type: 265
- Fields:
  - `chain_hash` : `chain_hash`
  - `u32` : `first_timestamp`
  - `u32` : `timestamp_range`

Once a peer has synced the channel graph, they can send this message if they wish to receive real-time updates on changes in the channel graph. They can also set the `first_timestamp` and `timestamp_range` fields if they wish to receive a backlog of updates they may have missed while they were down.

Appendix D: Sources and License Notices

This appendix contains attribution and license notices for material included by permission granted via open licenses.

Sources

Material was sourced from various public and open-licensed sources:

- ION Lightning Network Wiki
- "Lightning 101: What Is a Lightning Invoice?" by Suredbits
- Lightning Network In-Progress Specifications GitHub; Creative Commons Attribution (CC-BY 4.0)
- Wikipedia page, "Elliptic-curve Diffie–Hellman"
- Wikipedia page, "Digital signature"
- Wikipedia page, "Cryptographic hash function"
- Wikipedia page, "Onion routing"
- Wikimedia Commons, "Lightning Network Protocol Suite"
- Wikimedia Commons, "Introduction to the Lightning Network Protocol and the Basics of Lightning Technology"

BTCPay Server

BTCPay Server logo, screenshots, and other images used with permission under the MIT License:

Lamassu Industries AG

Images of the *Gaia* Bitcoin ATM seen in A Lamassu Bitcoin ATM are used with permission of Lamassu Industries AG. The use of these images is not an endorsement of the product or company, but are provided as a visual example of a Bitcoin ATM.

Onion Error Failure Types

Table 12. Onion error failure types

Type	Symbolic name	Meaning
PERM|1	invalid_realm	The `realm` byte was not understood by the processing node
NODE|2	temporary_node_failure	General temporary failure of the processing node
PERM|NODE|2	permanent_node_failure	General permanent failure of the processing node
PERM|NODE|3	required_node_fea&#x2060;ture_&#x200b;missing	The processing node has a required feature which was not in this onion
BADONION|PERM|4	invalid_onion_version	The `version` byte was not understood by the processing node
BADONION|PERM|5	invalid_onion_hmac	The HMAC of the onion was incorrect when it reached the processing node
BADONION|PERM|6	invalid_onion_key	The ephemeral key was unparsable by the processing node
UPDATE|7	temporary_channel_&#x200b;fail&#x2060;ure	The channel from the processing node was unable to handle this HTLC, but may be able to handle it, or others, later
PERM|8	permanent_channel_&#x200b;fail&#x2060;ure	The channel from the processing node is unable to handle any HTLCs
PERM|9	required_channel_&#x200b;fea&#x2060;ture_missing	The channel from the processing node requires features not present in the onion
PERM|10	unknown_next_peer	The onion specified a `short_channel_id` which doesn't match any leading from the processing node
UPDATE|11	amount_below_minimum	The HTLC amount was below the `htlc_minimum_msat` of the channel from the processing node

Type	Symbolic name	Meaning
UPDATE|12	fee_insufficient	The fee amount was below that required by the channel from the processing node
UPDATE|13	incorrect_cltv_expiry	The `cltv_expiry` does not comply with the `cltv_expiry_delta` required by the channel from the processing node
UPDATE|14	expiry_too_soon	The CLTV expiry is too close to the current block height for safe handling by the processing node
PERM|15	incor&#x2060;rect_or_unknown_&#x200b;pay&#x2060;ment_details	The `payment_hash` is unknown to the final node, the `payment_secret` doesn't match the `payment_hash`, the amount for that `payment_hash` is incorrect, or the CLTV expiry of the HTLC is too close to the current block height for safe handling
18	final_incor&#x2060;rect_&#x200b;cltv_expiry	The CLTV expiry in the HTLC doesn't match the value in the onion
19	final_incor&#x2060;rect_&#x200b;htlc_amount	The amount in the HTLC doesn't match the value in the onion
UPDATE|20	channel_disabled	The channel from the processing node has been disabled
21	expiry_too_far	The CLTV expiry in the HTLC is too far in the future
PERM|22	invalid_onion_payload	The decrypted onion per-hop payload was not understood by the processing node or is incomplete
23	mpp_timeout	The complete amount of the multipart payment was not received within a reasonable time

Colophon

The animals on the cover of *Mastering the Lightning Network* are wood ants (*Formica rufa*). Commonly used to describe a broad group of ants, “wood ants” are those that either construct nests in forested areas or infest wood in a home. However, *Formica rufa* specifically refers to the mound-building red wood ants that are mainly found across southern Britain, northern-to-middle Europe, the Pyrenees mountain range, and Siberia. Sometimes, they are also found in North America in both coniferous and broad-leaf broken woodlands and parklands.

Also known as the southern wood ant, this subspecies of wood ants are aggressive, active, and large. The wood ant queens are typically 12–15 mm in size and can live up to 15 years. Worker ants, on the other hand, are slightly smaller at 8–10 mm and have a lifespan of anywhere between a few weeks to seven years depending on whether they’re male or female (males die soon after mating).

Capable of producing formic acid in their abdomens, red wood ants can eject it up to a few feet away when threatened by predators. Their nests are often conspicuous mounds of grass, twigs, or conifer needles, often built against a rotting tree stump in an area that the sunlight can easily reach. Wood ants live in large colonies that may have 100,000 to 400,000 workers and 100 queens. Red wood ants are very territorial, and often attack and remove other ant species from the area.

Red wood worker ants forage up to 50 meters from their nest to collect a natural resin found dripping from pine trees. In a behavior unique to wood ants, individual ants walk over the resin to disinfect themselves from bacteria and fungi. Additionally, they also eat aphid honeydew, small insects, and arachnids. Red wood ants are commonly used in forestry and often introduced into an area as a form of pest management.

The red wood ants are currently a protected species and are categorized as “near threatened” by the IUCN. Many of the animals on O’Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to animals.oreilly.com.

The cover illustration is by Karen Montgomery, based on a black-and-white engraving from a loose plate, origin unknown. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag’s Ubuntu Mono.

[1] The Wikipedia entry on game theory provides more information.

[2] Joseph Poon and Thaddeus Dryja. "The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments." DRAFT Version 0.5.9.2. January 14, 2016. https://lightning.network/lightning-network-paper.pdf.

[3] Andreas M. Antonopoulos, *Mastering Bitcoin*, 2nd Edition, Chapter 1 (O'Reilly)

[4] ACINQ: Developers of the Eclair Mobile Lightning wallet (https://acinq.co).

[5] It is generally not advisable to reuse the same Bitcoin address for multiple payments because all Bitcoin transactions are public. A nosy person passing by could scan Alice's QR code and see how many tips Alice has already received to this address on the Bitcoin blockchain. Fortunately, the Lightning Network offers more private solutions to this, discussed later in the book!

[6] The Eclair wallet doesn't offer an option to automatically calculate the necessary fees and allocate the maximum amount of funds to a channel, so Alice has to calculate this herself.

[7] While the original Lightning whitepaper described channels funded by both channel partners, the current specification, as of 2020, assumes that just one partner commits funds to the channel. As of May 2021, dual-funded lightning channels are experimental in the c-lightning LN implementation.

[8] George Danezis and Ian Goldberg, "Sphinx: A Compact and Provably Secure Mix Format," in *IEEE Symposium on Security and Privacy* (New York: IEEE, 2009), 269–282.

[9] The term "onion" was originally used by the Tor project. Moreover, the Tor network is also called the Onion network and the project uses an onion as its logo. The top-level domain name used by Tor services on the internet is *onion*.

[10] George Danezis and Ian Goldberg, "Sphinx: A Compact and Provably Secure Mix Format," in *IEEE Symposium on Security and Privacy* (New York: IEEE, 2009), 269–282.

[11] The word *HODL* comes from an excited misspelling of the word "HOLD" shouted in a forum to encourage people not to sell bitcoin in a panic.

[12] Recall that change doesn't have to be the last output in a transaction and is in fact indistinguishable from other outputs.

[13] The first argument (0) does not have any meaning but is required due to a bug in Bitcoin's multisignature implementation. This issue is described in *Mastering Bitcoin*, Chapter 7.

[14] From BOLT #3.