KAPITEL 8

Routing in einem Netzwerk aus Zahlungskanälen

In diesem Kapitel wollen wir zeigen, wie über einen als *Routing* bezeichneten Prozess Zahlungskanäle miteinander verbunden werden können, um ein Netzwerk aus Zahlungskanälen zu bilden. Konkret sehen wir uns den ersten Teil der Routing-Schicht an, das Protokoll für »atomare und vertrauensfreie Multihop-Kontrakte«. Diese Schicht ist in Abbildung 8-1 entsprechend hervorgehoben.

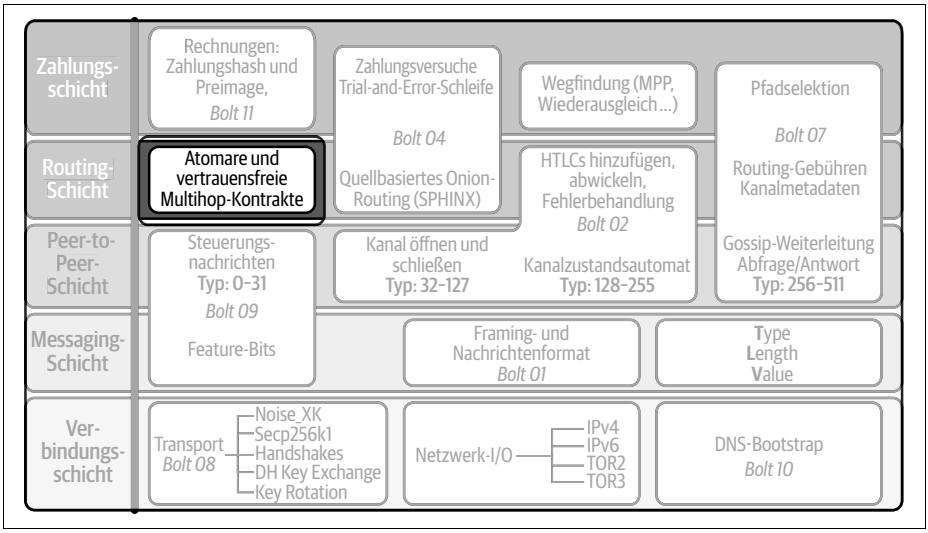

Abbildung 8-1: Atomares Routing von Zahlungen in der Lightning-Protokoll-Suite

Routing einer Zahlung

In diesem Abschnitt wollen wir das Routing aus der Perspektive von Dina untersuchen, einer Gamerin, die Spenden von ihren Fans erhält, während sie ihre Gaming-Sessions streamt.

Die Innovation gerouteter Zahlungskanäle erlaubt es Dina, Zahlungen zu empfangen, ohne einen separaten Kanal mit jedem Fan pflegen zu müssen, der ihr etwas

spenden möchte. Solange es von diesem Zuschauer einen Pfad kapitalkräftiger Kanäle zu Dina gibt, kann sie Zahlungen von diesem Fan empfangen.

In Abbildung 8-2 sehen wir ein mögliches Netzwerklayout, das von verschiedenen Zahlungskanälen zwischen Lightning-Nodes aufgebaut wurde. In diesem Diagramm kann jeder Dina eine Zahlung senden, indem er einen Pfad aufbaut. Nehmen wir an, Fan 4 möchte Dina eine Zahlung senden. Sehen Sie den Pfad, der das ermöglicht? Fan 4 könnte eine Zahlung an Dina über Fan 3, Bob und Chan routen. Ebenso könnte Alice eine Zahlung an Dina über Bob und Chan routen.

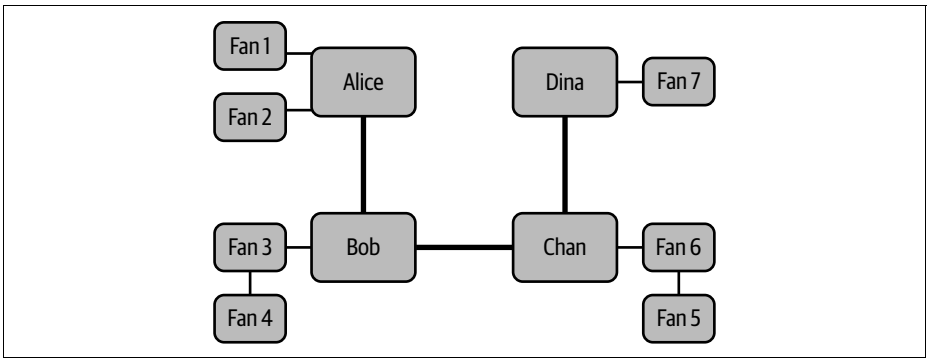

Abbildung 8-2: Mit Dina im Lightning-Netzwerk (in)direkt verbundene Fans

Die Nodes entlang des Pfads vom Fan zu Dina sind »Vermittler«, die im Kontext des Routings einer Zahlung als *Routing-Nodes* bezeichnet werden. Funktional gibt es keinen Unterschied zwischen Routing-Nodes und den Nodes, die von Dinas Fans betrieben werden. Jede Lightning-Node ist in der Lage, Zahlungen zwischen ihren Zahlungskanälen zu routen.

Ein wichtiger Punkt ist, dass Routing-Nodes kein Geld stehlen können, während sie eine Zahlung von einem Fan an Dina weiterleiten. Darüber hinaus können Routing-Nodes kein Geld verlieren, während sie am Routing-Prozess teilnehmen. Routing-Nodes können eine Routing-Gebühr dafür verlangen, dass sie als Vermittler tätig sind, doch das ist keine Pflicht. Sie können Zahlungen auch kostenlos routen.

Ein weiteres wichtiges Detail ist, dass durch die Nutzung des Onion-Routings die vermittelnden Nodes nur die vorherige und die nächste Node entlang des Pfads kennen. Sie müssen nicht wissen, wer Sender und wer Empfänger einer Zahlung ist. Das ermöglicht den Fans, vermittelnde Nodes zu nutzen, um Dina zu bezahlen, ohne private Informationen offenlegen zu müssen und ohne einen Diebstahl zu riskieren.

Dieser Prozess, bei dem eine Reihe von Zahlungskanälen mit Ende-zu-Ende-Sicherheit verbunden werden, und die für Nodes reizvolle Struktur, Zahlungen *weiterzuleiten*, sind Schlüsselinnovationen des Lightning-Netzwerks.

In diesem Kapitel tauchen wir in den Mechanismus des Routings im Lightning-Netzwerk ein und sehen uns detailliert an, wie Zahlungen durch das Netzwerk flie-

ßen. Zuerst erläutern wir das Konzept des Routings und vergleichen es mit dem der Wegfindung, weil diese häufig verwechselt und vertauscht werden. Dann sehen wir uns das Fairness-Protokoll an: ein atomares, vertrauensfreies Multihop-Protokoll zum Routing von Zahlungen. Um zu zeigen, wie das Fairness-Protokoll funktioniert, nutzen wir das physikalische Gegenstück der Übertragung von Goldmünzen zwischen vier Menschen. Abschließend sehen wir uns die Implementierung des atomaren, vertrauensfreien Multihop-Protokolls an, das momentan im Lightning-Netzwerk verwendet wird: den sogenannten *Hash Time-Locked Contract* (HTLC).

Routing versus Wegfindung

Es ist wichtig, anzumerken, dass wir das Konzept des Routings vom Konzept der Wegfindung (engl. *Pathfinding*) unterscheiden. Diese beiden Konzepte werden oft verwechselt, und der Begriff *Routing* wird häufig für beide Konzepte verwendet. Bevor wir tiefer einsteigen, wollen wir diese Mehrdeutigkeit auflösen.

Die *Wegfindung*, die in Kapitel 12 behandelt wird, ist der Prozess des Suchens und Wählens eines zusammenhängenden Wegs (Pfads) von Zahlungskanälen, die den Sender A mit dem Empfänger B verbinden. Der Sender einer Zahlung übernimmt die Wegfindung, indem er den *Kanal-Graphen* untersucht, den er aus den Kanalankündigungen aufgebaut hat, die von anderen Nodes bekannt gemacht wurden.

Das *Routing* beschreibt eine Reihe von Interaktionen im Netzwerk, die versuchen, eine Zahlung von einem Punkt A zu einem Punkt B weiterzuleiten. Dabei wird der vorher durch die Wegfindung gefundene Pfad verwendet. Das Routing ist der aktive Prozess des Sendens einer Zahlung über einen Pfad, der die Kooperation aller dazwischenliegenden Nodes entlang dieses Pfads verlangt.

Ein wichtiger Punkt ist, dass es möglicherweise einen *Pfad* zwischen Alice und Bob gibt (vielleicht sogar mehrere), aber keine aktive *Route*, über die man die Zahlung senden kann. Das ist beispielsweise der Fall, wenn alle Nodes offline sind, die Alice und Bob verbinden. In diesem Fall kann man den Kanal-Graphen untersuchen und eine Reihe von Zahlungskanälen zwischen Alice zu Bob verbinden, weil ein *Pfad* existiert. Da aber die dazwischenliegenden Nodes offline sind, kann die Zahlung nicht gesendet werden, da keine *Route* existiert.

Ein Netzwerk von Zahlungskanälen aufbauen

Bevor wir in das Konzept atomarer, vertrauensfreier Multihop-Zahlungen eintauchen, wollen wir ein Beispiel durchgehen. Wir kehren zu Alice zurück, die in den vorherigen Kapiteln einen Kaffee bei Bob gekauft hat, mit dem sie einen offenen Kanal besitzt. Alice sieht sich nun einen Livestream von Dina, der Gamerin, an und möchte ihr 50.000 Satoshi über das Lightning-Netzwerk zukommen lassen. Doch Alice hat keinen direkten Kanal zu Dina. Was kann Alice tun?

Alice könnte einen direkten Kanal mit Dina öffnen, doch dafür ist Liquidität nötig, und die On-Chain-Gebühren könnten höher sein als die Spende selbst. Stattdessen kann Alice ihre bereits geöffneten Kanäle nutzen, um Dina eine Spende zukommen zu lassen, ohne einen direkten Kanal mit Dina öffnen zu müssen. Das ist möglich, solange es einen Pfad mit ausreichender Kapazität zwischen Alice und Dina gibt, über den die Zahlung geroutet werden kann.

Wie Sie in Abbildung 8-3 sehen, hat Alice einen Kanal mit Bob (dem Besitzer des Cafés) geöffnet. Bob hat wiederum einen Kanal mit Softwareentwickler Chan geöffnet, der ihm mit dem Kassensystem seines Cafés hilft. Chan ist Inhaber eines großen Softwareunternehmens, das das von Dina gespielte Spiel entwickelt. Die beiden haben bereits einen Kanal geöffnet, über den Dina die Spielelizenz und Ingame-Items kauft.

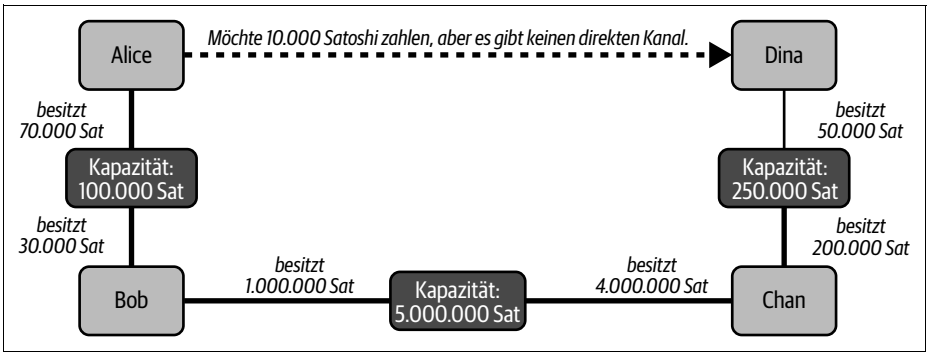

Abbildung 8-3: Ein Netzwerk aus Zahlungskanälen zwischen Alice und Dina

Ein Pfad von Alice zu Dina ist möglich. Dabei werden Bob und Chan als zwischengeschaltete Routing-Nodes verwendet. Alice kann eine *Route* entlang des gefundenen Pfads aufbauen und ein paar Tausend Satoshi an Dina senden, die von Bob und Chan *weitergeleitet* (engl. *forwarding*) werden. Im Grunde genommen bezahlt Alice Bob, der Chan bezahlt, der wiederum Dina bezahlt. Ein direkter Kanal zwischen Alice und Dina wird nicht benötigt.

Die größte Herausforderung besteht darin, zu verhindern, dass Bob und Chan das Geld stehlen, das Alice Dina senden möchte.

Ein reales Beispiel für das Routing

Um zu verstehen, wie das Lightning-Netzwerk die Zahlung während des Routings schützt, wollen wir es mit der Weitergabe von Goldmünzen im »richtigen Leben« vergleichen.

Nehmen wir an, Alice möchte Dina zehn Goldmünzen zukommen lassen, hat mit ihr aber keinen direkten Kontakt. Alice kennt Bob, der Chan kennt, der Dina kennt. Sie entscheidet sich daher, Bob und Chan um Hilfe zu bitten. Das ist in Abbildung 8-4 zu sehen.

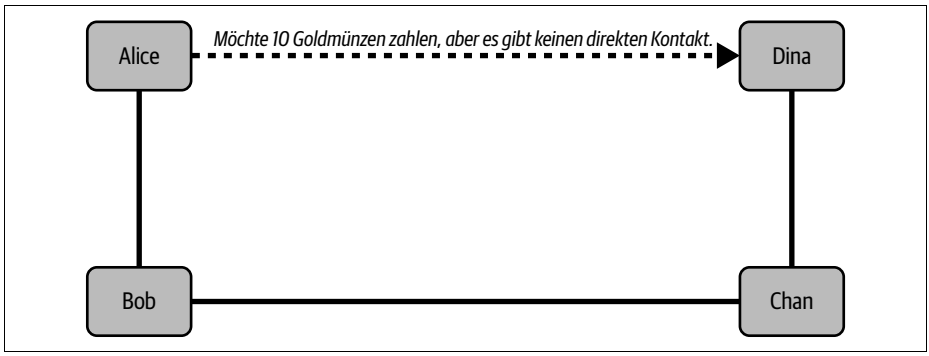

Abbildung 8-4: Alice möchte Dina zehn Goldmünzen zahlen.

Alice kann Bob bezahlen, der Chan bezahlt, der Dina bezahlt. Doch wie kann man verhindern, dass sich Bob oder Chan mit dem Geld aus dem Staub macht? Im richtigen Leben können Verträge abgeschlossen werden, um eine Reihe von Zahlungen vorzunehmen.

Alice könnte mit Bob den folgenden Vertrag vereinbaren:

> *Ich, Alice, gebe dir, Bob, zehn Goldmünzen, wenn du sie an Chan übergibst.*

Dieser Vertrag ist theoretisch schön und gut, doch im richtigen Leben läuft Alice Gefahr, dass Bob den Vertrag bricht und hofft, nicht erwischt zu werden. Selbst wenn Bob erwischt und belangt wird, steht Alice vor dem Problem, dass Bob pleite sein könnte und die zehn Goldmünzen nicht zurückgeben kann. Aber selbst wenn man diese Punkte irgendwie löst, bleibt immer noch unklar, wie man einen solchen Vertrag durchsetzen kann, um das gewünschte Ziel zu erreichen: dass die Münzen an Dina ausgezahlt werden.

Wir erweitern unseren Vertrag daher wie folgt:

> *Ich, Alice, erstatte dir, Bob, zehn Goldmünzen, wenn du mir nachweisen kannst (z. B. durch eine Quittung), dass du zehn Goldmünzen an Chan übergeben hast.*

Sie könnten sich fragen, warum Bob so einen Vertrag unterzeichnen sollte. Er muss Chan bezahlen, erhält im Gegenzug aber nichts dafür und läuft Gefahr, dass Alice ihn nicht bezahlt. Bob kann Chan einen ähnlichen Vertrag anbieten, damit er Dina bezahlt, doch auch für Chan gibt es keinen Grund, einen solchen Vertrag zu unterschreiben.

Selbst wenn man das Risiko außer Acht lässt, müssen Bob und Chan zehn Goldmünzen *besitzen*, um sie senden zu können. Andernfalls wären sie nicht in der Lage, einen solchen Vertrag zu erfüllen.

Bob und Chan würden beide ein Risiko eingehen und Opportunitätskosten tragen, wenn sie diesem Vertrag zustimmten. Man muss also einen Ausgleich schaffen, damit sie auf ihn eingehen.

Für Bob und Chan kann das interessant werden, wenn Alice beiden eine Gebühr von jeweils einer Goldmünze anbietet, falls sie die Zahlung an Dina weiterleiten.

Der Vertrag sähe dann so aus:

> *Ich, Alice, erstatte dir, Bob, zwölf Goldmünzen, wenn du mir nachweisen kannst (z. B. durch eine Quittung), dass du elf Goldmünzen an Chan übergeben hast.*

Alice verspricht Bob nun zwölf Goldmünzen: zehn für Dina und zwei für die Gebühren. Sie verspricht Bob zwölf Goldmünzen, wenn er beweisen kann, dass er elf an Chan weitergeleitet hat. Der Unterschied von einer Goldmünze ist die Gebühr, die Bob dafür erhält, dass er bei dieser Zahlung hilft. In Abbildung 8-5 sehen wir, wie durch dieses Arrangement zehn Goldmünzen über Bob und Chan an Dina weiterleitet werden.

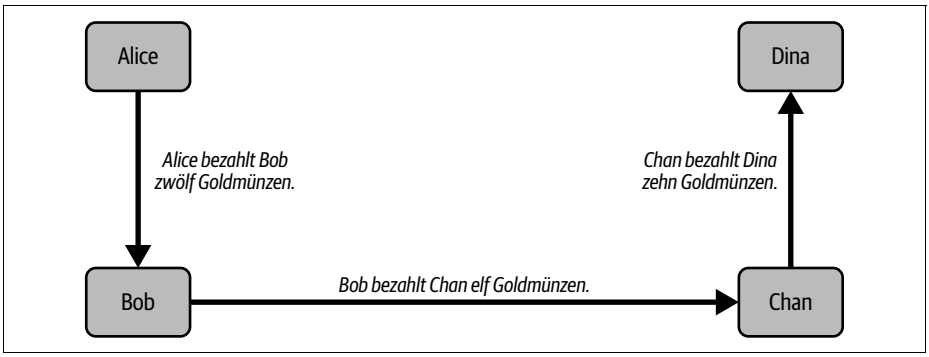

Abbildung 8-5: Alice bezahlt Bob, Bob bezahlt Chan, Chan bezahlt Dina.

Da Vertrauen immer noch ein Thema ist und die Gefahr besteht, dass Alice oder Bob die Vereinbarung nicht einhält, entscheiden alle Parteien, einen Treuhänder einzuschalten. Zu Beginn des Austauschs hinterlegt Alice die zwölf Goldmünzen auf einem Treuhandkonto, das an Bob nur ausgezahlt wird, wenn er nachweisen kann, dass er elf Goldmünzen an Chan gezahlt hat.

Wir idealisieren den Treuhänder ein wenig, da er kein weiteres Risiko (z. B. Kontrahentenrisiko) darstellt. Später werden wir den Treuhänder durch einen Bitcoin Smart Contract ersetzen. Für den Augenblick nehmen wir einfach an, dass alle Seiten dem Treuhänder vertrauen.

Im Lightning-Netzwerk kann die Quittung (der Nachweis der Zahlung) ein Secret sein, das nur Dina kennt. In der Praxis ist dieses Secret eine Zufallszahl, die so groß ist, dass sie von anderen nicht erraten werden kann (üblicherweise eine *sehr, sehr* große Zahl, die in 256 Bit codiert wird).

Dina erzeugt den Secret-Wert `R` mithilfe eines Zufallszahlengenerators.

Das Secret könnte dann im Vertrag als SHA-256-Hash festgehalten werden:

$$H = \text{SHA-256}(R)$$

Wir nennen den Hash des Secrets für die Zahlung den *Zahlungshash*. Das Secret, das die Zahlung »freigibt«, ist das *Zahlungs-Secret*.

Wir wollen die Dinge einfach halten und verwenden als Dinas Secret schlicht den Text `Dinas secret`. Diese geheime Nachricht wird *Zahlungs-Secret* oder *Zahlungs-Preimage* genannt.

Um dieses Secret einzubinden (*commit*), berechnet Dina den SHA-256-Hash, der hexadezimal codiert wie folgt aussieht:

```
0575965b3b44be51e8057d551c4016d83cb1fba9ea8d6e986447ba33fe69f6b3
```

Um Alice die Zahlung zu ermöglichen, erzeugt Dina das Zahlungs-Secret und den Zahlungshash und sendet den Zahlungshash an Alice. In Abbildung 8-6 sehen wir, dass Dina den Zahlungshash über einen externen Kanal (die gestrichelte Linie) an Alice sendet, etwa per E-Mail oder Messenger.

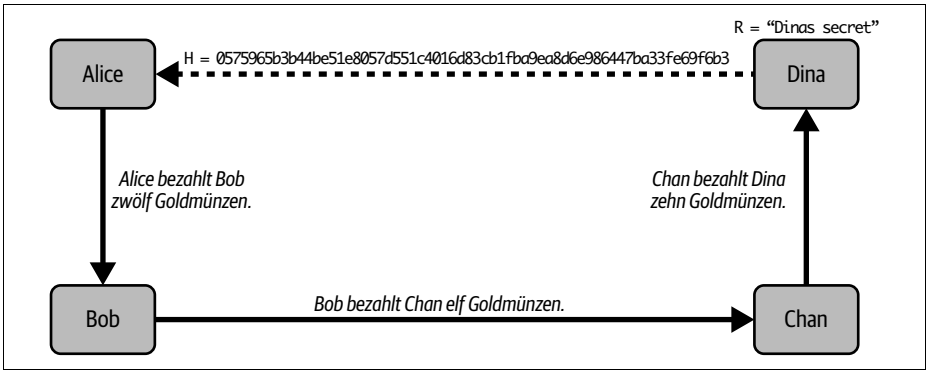

Abbildung 8-6: Dina sendet das gehashte Secret an Alice.

Alice kennt das Secret nicht, kann ihren Vertrag aber so umformulieren, dass der Hash des Secrets als Nachweis der Zahlung dient:

> *Ich, Alice, erstatte dir, Bob, zwölf Goldmünzen, wenn du mir eine gültige Nachricht vorlegst, die den Hash 057596... ergibt.*
> *Du erhältst diese Nachricht, wenn du einen vergleichbaren Vertrag mit Chan eingehst, der einen ähnlichen Vertrag mit Dina eingehen muss. Um sicherzustellen, dass die Zahlung erfolgt, hinterlege ich die zwölf Goldmünzen bei einem Treuhänder, bevor du den nächsten Vertrag abschließt.*

Dieser neue Vertrag schützt Alice davor, dass Bob das Geld nicht an Chan weiterleitet. Außerdem schützt er Bob davor, die Zahlung von Alice nicht zurückerstattet zu bekommen. Gleichzeitig stellt er durch den Hash von Dinas Secret sicher, dass Dina tatsächlich bezahlt wurde.

Nachdem Bob und Alice sich auf den Vertrag geeinigt haben und Bob die Nachricht erhält, dass Alice die zwölf Goldmünzen bei einem Treuhänder hinterlegt hat, kann Bob einen vergleichbaren Vertrag mit Chan schließen.

Da Bob eine Gebühr von einer Goldmünze erhebt, leitet er nur elf Goldmünzen an Chan weiter, sobald dieser nachweisen kann, dass er Dina bezahlt hat. Auch Chan verlangt eine Gebühr und erwartet den Eingang von elf Goldmünzen, sobald er nachgewiesen hat, dass er Dina die versprochenen zehn Goldmünzen überwiesen hat.

Bobs Vertrag mit Chan sieht wie folgt aus:

> *Ich, Bob, erstatte dir, Chan, elf Goldmünzen, wenn du mir eine gültige Nachricht vorlegst, die den Hash 057596... ergibt. Du erhältst diese Nachricht, wenn du einen vergleichbaren Vertrag mit Dina eingehst. Um sicherzugehen, dass die Erstattung erfolgt, hinterlege ich die elf Goldmünzen bei einem Treuhänder, bevor du den nächsten Vertrag abschließt.*

Sobald Chan vom Treuhänder die Nachricht erhält, dass Bob die elf Goldmünzen hinterlegt hat, setzt er einen ähnlichen Vertrag mit Dina auf:

> *Ich, Chan, erstatte dir, Dina, zehn Goldmünzen, wenn du mir eine gültige Nachricht vorlegst, die den Hash 057596... ergibt. Um sicherzustellen, dass die Erstattung nach Vorlage des Secrets erfolgt, hinterlege ich zehn Goldmünzen bei einem Treuhänder.*

Nun ist alles vorbereitet. Alice hat einen Vertrag mit Bob und hat zwölf Goldmünzen bei einem Treuhänder hinterlegt. Bob hat einen Vertrag mit Chan und hat elf Goldmünzen bei einem Treuhänder hinterlegt. Chan hat einen Vertrag mit Dina und hat zehn Goldmünzen bei einem Treuhänder hinterlegt. Dina muss nun das Secret offenlegen, also das Preimage des Hashs, das sie als Zahlungsnachweis festgelegt hat.

Dina sendet nun `Dinas secret` an Chan.

Chan prüft, ob `Dinas secret` den Hash `057596`... ergibt. Chan besitzt jetzt den Zahlungsnachweis und kann den Treuhänder anweisen, die zehn Goldmünzen an Dina auszuzahlen.

Chan gibt das Secret nun an Bob weiter. Bob prüft es und weist den Treuhänder an, die elf Goldmünzen für Chan freizugeben.

Bob gibt das Secret an Alice weiter. Alice prüft es und weist den Treuhänder an, die zwölf Goldmünzen für Bob freizugeben. Alle Verträge sind nun abgeschlossen. Alice hat insgesamt zwölf Goldmünzen bezahlt: eine für Bob, eine für Chan und zehn für Dina. Bei einer solchen Kette von Verträgen machen sich Bob und Chan nicht aus dem Staub, weil sie die Mittel zuerst bei einem Treuhänder hinterlegt haben.

Es gibt aber noch ein Problem. Wenn sich Dina weigert, das Preimage zu veröffentlichen, hängen Chans, Bobs und Alice' Goldmünzen beim Treuhänder fest und werden nicht erstattet. Das Gleiche passiert, wenn jemand innerhalb der Kette das Secret nicht weitergibt. Es kann also niemand Alice' Geld stehlen, doch alle könnten vergeblich auf ihr Geld vom Treuhänder warten.

Glücklicherweise lässt sich das lösen, indem man den Vertrag um eine Frist ergänzt.

Wir können den Vertrag dahin gehend ergänzen, dass er innerhalb einer bestimmten Frist erfüllt werden muss. Andernfalls läuft der Vertrag aus, und der Treuhänder zahlt das Geld an die Person zurück, die es ursprünglich hinterlegt hat. Wie nennen diese Frist *Timelock*.

Die Mittel werden beim Treuhänder für eine bestimmte Zeit hinterlegt und letztlich wieder freigegeben, wenn kein Zahlungsnachweis erfolgt ist.

Um das zu berücksichtigen, wird der Vertrag zwischen Alice und Bob noch einmal um eine neue Klausel ergänzt:

> *Bob hat nach der Unterzeichnung 24 Stunden Zeit, das Secret vorzulegen. Kann Bob das Secret innerhalb dieser Zeit nicht vorlegen, erhält Alice ihr Geld vom Treuhänder zurück, und der Vertrag wird ungültig.*

Bob muss jetzt natürlich sicherstellen, dass er selbst den Zahlungsnachweis innerhalb von 24 Stunden erhält. Selbst wenn er Chan ordnungsgemäß bezahlt, erhält er kein Geld, wenn er den Zahlungsnachweis nicht innerhalb dieser Frist erhält. Um dieses Risiko auszuschalten, setzt er Chan eine kürzere Frist.

Bob passt seinen Vertrag mit Chan daher wie folgt an:

> *Chan hat nach der Unterzeichnung 22 Stunden Zeit, das Secret vorzulegen. Kann er das Secret innerhalb dieser Zeit nicht vorlegen, erhält Bob sein Geld vom Treuhänder zurück, und der Vertrag wird ungültig.*

Wie Sie sich denken können, passt auch Chan seinen Vertrag mit Dina an:

> *Dina hat nach der Unterzeichnung 20 Stunden Zeit, das Secret vorzulegen. Legt sie das Secret nicht innerhalb dieser Zeit vor, erhält Chan sein Geld vom Treuhänder zurück, und der Vertrag wird ungültig.*

Mit einer solchen Kette von Verträgen können wir sicherstellen, dass die Zahlung von Alice über Bob über Chan an Dina innerhalb von 24 Stunden abgeschlossen ist oder fehlschlägt und jeder sein Geld zurückerhält. Der Vertrag wird entweder erfolgreich abgeschlossen oder schlägt fehl. Etwas dazwischen gibt es nicht.

Im Kontext des Lightning-Netzwerks bezeichnen wir diesen »Alles-oder-nichts-Ansatz« als *atomar*.

Solange der Treuhänder vertrauenswürdig ist und seine Aufgabe gewissenhaft erfüllt, können während dieses Prozesses keiner Partei die Einlagen gestohlen werden.

Die Grundvoraussetzung dafür, dass diese *Route* überhaupt funktioniert, ist aber, dass alle Parteien ausreichend Geld haben, um die Folge von Einzahlungen erfüllen zu können.

Das klingt zwar nach einem unbedeutenden Detail, doch Sie werden später noch sehen, dass diese Anforderung eines der größeren Probleme für LN-Nodes darstellt. Und es wird umso schwieriger, je größer die Zahlung ist. Darüber hinaus können die Parteien ihre Einlagen nicht nutzen, solange sie beim Treuhänder hinterlegt sind.

Daher entstehen Nutzern, die Zahlungen weiterleiten, durch das Sperren des Geldes Opportunitätskosten, die letztlich durch Routing-Gebühren ausgeglichen werden, wie wir im vorherigen Beispiel gesehen haben.

Nachdem wir das Routing im »richtigen Leben« betrachtet haben, wollen wir uns nun anschauen, wie es auf der Bitcoin-Blockchain ohne die Notwendigkeit eines externen Treuhänders implementiert wird. Zu diesem Zweck richten wir Kon-

trakte zwischen den Teilnehmern in Bitcoin-Skript ein. Wir ersetzen den externen Treuhänder durch *Smart Contracts*, die ein Fairness-Protokoll implementieren. Lassen Sie uns dieses Konzept aufdröseln und implementieren!

Fairness-Protokoll

Wie bereits im ersten Kapitel dieses Buchs erwähnt, besteht die Innovation von Bitcoin in der Fähigkeit, kryptografische Primitive zu verwenden, um ein Fairness-Protokoll zu implementieren, das das Vertrauen in Drittparteien (Vermittler) durch ein vertrauenswürdiges Protokoll ersetzt.

In unserem Beispiel mit den Goldmünzen mussten wir einen Treuhänder einschalten, um zu verhindern, dass eine der Parteien ihr Wort bricht. Die Innovation kryptografischer Fairness-Protokolle erlaubt es uns, den Treuhänder durch ein Protokoll zu ersetzen.

Die Eigenschaften des von uns zu entwickelnden Fairness-Protokolls sind:

Vertrauensfreier Betrieb
: Die Teilnehmer einer gerouteten Zahlung müssen weder einander noch irgendeiner dritten Partei vertrauen. Stattdessen vertrauen sie dem Protokoll, das sie vor Betrug schützt.

Atomizität
: Eine Zahlung wird entweder vollständig ausgeführt, oder sie schlägt fehl, und jeder erhält sein Geld zurück. Es ist nicht möglich, dass ein Intermediär eine geroutete Zahlung einstreicht und nicht an den nächsten Hop weiterleitet. Intermediäre können also nicht betrügen oder stehlen.

Multihop
: Die Sicherheit des Systems erstreckt sich Ende zu Ende über mehrere Zahlungskanäle hinweg, genau wie bei einer Zahlung zwischen den beiden Enden eines einzelnen Zahlungskanals.

Eine optionale, zusätzliche Eigenschaft ist die Fähigkeit, Zahlungen in mehrere Teile aufzuspalten und gleichzeitig die Atomizität der Zahlung zu erhalten. Das wird *Multipart-Zahlung* (*Multipart Payment*, kurz *MPP*) genannt und in »Multipart-Zahlungen« auf Seite 326 genauer erläutert.

Implementierung atomarer vertrauensfreier Multihop-Zahlungen

Bitcoin-Skript ist so flexibel, dass es Dutzende von Möglichkeiten gibt, ein Fairness-Protokoll zu implementieren, das die Eigenschaften Atomizität, vertrauensfreier Betrieb und Multihop-Sicherheit vereint. Die Wahl der jeweiligen Implementierung ist ein Kompromiss aus Privatsphäre, Effizienz und Komplexität.

Das heutzutage vom Lightning-Netzwerk verwendete Fairness-Protokoll heißt *Hash Time-Locked Contract* (HTLC). HTLCs nutzen ein Hash-Preimage als Secret zur Freigabe von Zahlungen wie in unserem Goldmünzenbeispiel zu Beginn dieses Kapitels. Der Empfänger einer Zahlung generiert einen zufälligen Secret-Wert und berechnet dessen Hash. Der Hash wird zur Bedingung für die Zahlung, und sobald das Secret offengelegt ist, können alle Teilnehmenden ihre eingehenden Zahlungen einlösen. HTLCs bieten Atomizität, vertrauensfreien Betrieb und Multihop-Sicherheit.

Ein weiterer vorgeschlagener Mechanismus zur Implementierung des Routings ist *Point Time-Locked Contract* (*PTLC*). PTLCs erreichen ebenfalls Atomizität, vertrauensfreien Betrieb und Multihop-Sicherheit, aber mit höherer Effizienz und besserer Privatsphäre. Die effiziente Implementierung von PTLCs hängt von einem neuen Signaturalgorithmus, den sogenannten *Schnorr-Signaturen*, ab, der Ende 2021 mit dem Taproot-Update aktiviert wurden.

Ein erneuter Blick auf das Spenden-Beispiel

Werfen wir noch einmal einen Blick auf das Beispiel aus dem ersten Teil dieses Kapitels. Alice möchte Dina über eine Lightning-Zahlung eine Spende zukommen lassen. Nehmen wir an, Alice möchte Dina 50.000 Satoshi spenden.

Damit Alice Dina etwas spenden kann, muss Dinas Node eine Rechnung für Alice erzeugen. Wir gehen darauf in Kapitel 15 im Detail ein. Für den Augenblick nehmen wir an, dass Dina eine Website betreibt, die eine Lightning-Rechnung für Spenden erzeugen kann.

Lightning-Zahlungen können über ein Feature namens *keysend* auch ohne Rechnung gesendet werden, das wir uns in »Spontane Zahlungen per keysend« auf Seite 289 etwas genauer ansehen. Hier wollen wir den einfacheren Zahlungsfluss mithilfe einer Rechnung erläutern.

Alice besucht Dinas Website, gibt den Betrag von 50.000 Satoshi in ein Formular ein, und Dinas Lightning-Node generiert daraus eine Zahlungsanforderung in Höhe von 50.000 Satoshi in Form einer Lightning-Rechnung. Diese Interaktion läuft außerhalb des Lightning-Netzwerks über das Web und ist in Abbildung 8-7 zu sehen.

Wie in den vorherigen Beispielen gehen wir auch hier davon aus, dass es keinen direkten Zahlungskanal zwischen Alice und Dina gibt. Um Dina bezahlen zu können, muss Alice einen Weg (Pfad) finden, der sie mit Dina verbindet. Diesen Schritt sehen wir uns in Kapitel 12 genauer an. Für den Augenblick nehmen wir an, dass Alice die notwendigen Informationen sammeln kann und sieht, dass es über Bob und Chan einen Pfad zu Dina gibt.

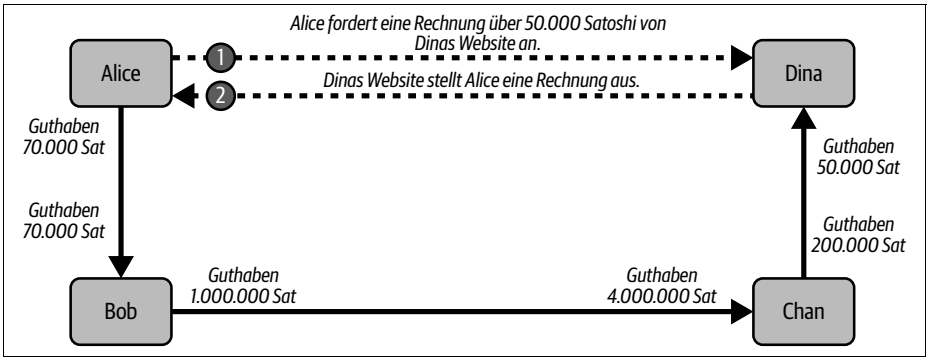

Abbildung 8-7: Alice fordert eine Rechnung über Dinas Website an.

Sie erinnern sich, dass Bob und Chan eine kleine Vergütung für das Routing der Zahlung erwarten? Alice möchte Dina 50.000 Satoshi zahlen, doch wie Sie in den folgenden Abschnitten noch sehen werden, sendet sie Bob 50.200 Satoshi. Die 200 zusätzlichen Satoshi werden zwischen Bob und Chan als Routing-Gebühr (also jeweils 100 Satoshi) aufgeteilt.

Alice' Node kann nun eine Lightning-Zahlung erzeugen. In den nächsten Abschnitten sehen wir, wie Alice' Node einen HTLC erzeugt, um Dina zu bezahlen, und wie dieser HTLC über den Pfad zwischen Alice und Dina weitergeleitet wird.

On-Chain- versus Off-Chain-Abwicklung von HTLCs

Das Lightning-Netzwerk soll *Off-Chain*-Transaktionen ermöglichen, die genauso vertrauenswürdig sind wie *On-Chain*-Transaktionen, da niemand betrügen kann. Der Grund dafür, dass niemand betrügen kann, ist, dass jeder Teilnehmer zu jedem Zeitpunkt seine Off-Chain-Transaktion On-Chain bringen kann. Jede Off-Chain-Transaktion kann jederzeit in der Bitcoin-Blockchain veröffentlicht werden. Die Bitcoin-Blockchain fungiert also bei Bedarf als Mechanismus der Konfliktlösung und der endgültigen Abrechnung.

Allein die Tatsache, dass jede Transaktion jederzeit On-Chain gebracht werden kann, ist der Grund dafür, dass all diese Transaktionen Off-Chain gehalten werden können. Solange Sie sich des Zugriffs auf Ihre Einlagen sicher sind, können Sie mit den anderen Teilnehmenden weiter kooperieren und die On-Chain-Abwicklung samt zusätzlichen Gebühren vermeiden.

In allen folgenden Beispielen gehen wir davon aus, dass jegliche Transaktion zu jeder Zeit On-Chain gebracht werden kann. Die Teilnehmer halten sie Off-Chain, doch die Funktionalität des Systems unterscheidet sich nur durch die höheren Gebühren und die Verzögerungen durch das On-Chain-Mining der Transaktionen. Die Beispiele funktionieren alle gleich, egal ob On-Chain oder Off-Chain.

Hash Time-Locked Contracts

In diesem Abschnitt erläutern wir, wie HTLCs funktionieren.

Der erste Teil eines HTLC ist der *Hash*, also die Verwendung eines kryptografischen Hashalgorithmus für ein zufällig generiertes Secret. Die Kenntnis des Secrets erlaubt das Einlösen der Zahlung. Die kryptografische Hashfunktion garantiert, dass der Hash für jeden leicht zu verifizieren ist, während es gleichzeitig unmöglich ist, das Secret-Preimage zu erraten. Darüber hinaus gibt es nur ein Preimage, das die (Aus-)Zahlungsbedingung erfüllt.

In Abbildung 8-8 sehen wir, dass Alice eine Lightning-Rechnung von Dina erhält. Innerhalb dieser Rechnung hat Dina einen *Zahlungshash* codiert, also den kryptografischen Hash des von Dinas Node erzeugten Secrets. Dinas Secret wird als *Zahlungs-Preimage* bezeichnet. Der Zahlungshash dient als Kennung (ID), der zum Routing der Zahlung an Dina verwendet werden kann. Sobald die Zahlung abgeschlossen ist, dient das Zahlungs-Preimage als Quittung und Nachweis der Zahlung.

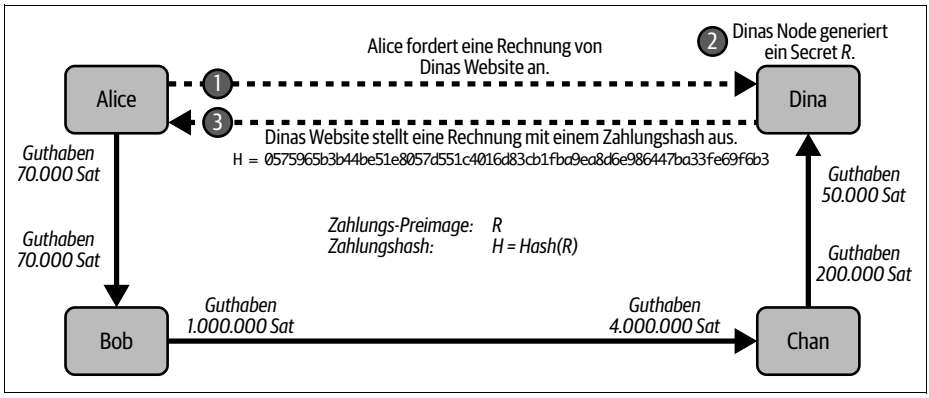

Abbildung 8-8: Alice erhält einen Zahlungshash von Dina.

Im Lightning-Netzwerk ist Dinas Zahlungs-Preimage kein einfacher Text wie `Dinas secret`, sondern eine durch Dinas Node erzeugte Zufallszahl. Wir nennen diese Zufallszahl *R*.

Dinas Node berechnet den kryptografischen Hash von *R* wie folgt:

$$H = \text{SHA-256}(R)$$

In dieser Gleichung ist *H* der Hash (oder *Zahlungshash*) und *R* das Secret (oder *Zahlungs_Preimage*). Der Einsatz einer kryptografischen Hashfunktion ist ein Element, das den *vertrauensfreien Betrieb* garantiert. Die Zahlungsvermittler müssen einander nicht vertrauen, weil sie wissen, dass niemand das Secret erraten oder fälschen kann.

HTLCs in Bitcoin-Skript

In unserem Goldmünzenbeispiel besaß Alice einen treuhänderisch abgesicherten Vertrag:

> *Alice erstattet Bob zwölf Goldmünzen, wenn er eine gültige Nachricht vorweisen kann, die den Hash 0575…f6b3 ergibt. Bob hat 24 Stunden nach Unterzeichnung des Vertrags Zeit, das Secret vorzulegen. Kann Bob das Secret innerhalb dieser Zeit nicht vorlegen, erhält Alice ihr Geld vom Treuhänder zurück, und der Vertrag wird ungültig.*

Sehen wir uns an, wie man das als HTLC in Bitcoin-Skript implementiert. In Beispiel 8-1 sehen Sie ein HTLC-Bitcoin-Skript, wie es momentan im Lightning-Netzwerk verwendet wird. Sie finden diese Definition in BOLT #3, Transactions (*https://github.com/lightning/bolts/blob/master/03-transactions.md#offered-htlc-outputs*).

Beispiel 8-1: In Bitcoin-Skript implementierter HTLC (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, das sieht kompliziert aus! Doch keine Sorge, wir schlüsseln das Schritt für Schritt auf.

Das im Lightning-Netzwerk verwendete Bitcoin-Skript ist recht komplex, da es für den On-Chain-Platzbedarf optimiert ist. Das Skript ist daher sehr kompakt, aber auch entsprechend schwierig zu lesen.

In den folgenden Abschnitten konzentrieren wir uns auf die Hauptelemente des Skripts und stellen vereinfachte Skripten vor, die sich ein wenig von der aktuellen Lightning-Implementierung unterscheiden.

Den Hauptteil des HTLC bildet Zeile 10 von Beispiel 8-1. Wir wollen das von Grund auf selbst entwickeln!

Zahlungs-Preimage und Hashverifikation

Den Kern eines HTLC bildet der Hash, über denn die Zahlung erfolgt, wenn der Empfänger das Zahlungs-Preimage kennt. Alice sichert die Zahlung durch einen bestimmten Zahlungshash ab, und Bob zeigt das Zahlungs-Preimage vor, um die Mittel einzulösen. Das Bitcoin-System kann die Korrektheit von Bobs Zahlungs-Preimage prüfen, indem es dessen Hash berechnet und mit dem Zahlungshash vergleicht, den Alice zur Sicherung der Geldmittel verwendet hat.

Dieser Teil eines HTLC kann in Bitcoin-Skript wie folgt implementiert werden:

```
OP_SHA256 <H> OP_EQUAL
```

Alice kann einen Transaktions-Output erzeugen, der 50.200 Satoshi über das obige Locking-Skript auszahlt, wobei `<H>` durch den von Dina bereitgestellten Hashwert `0575...f6b3` ersetzt wird. Alice signiert die Transaktion dann und bietet sie Bob an:

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

Bob kann diesen HTLC nicht ausgeben, solange er Dinas Secret nicht kennt, d.h., die Freigabe des HTLC bedingt, dass Bob seinen Teil der Zahlung erfüllt und diese Dina erreicht.

Sobald Bob Dinas Secret kennt, kann er diesen Output über das Unlocking-Skript entsperren, das den geheimen Preimage-Wert *R* enthält.

Die Kombination aus Entsperren (Unlocking-Skript) und Sperren (Locking-Skript) sieht so aus:

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

Die Bitcoin-Skript-Engine würde dieses Skript wie folgt evaluieren:

1. `R` wird auf dem Stack abgelegt.
2. Der Operator `OP_SHA256` nimmt den Wert von `R` vom Stack und wendet die Hashfunktion auf ihn an. Das Ergebnis H_R wird wieder auf dem Stack abgelegt.
3. `H` wird auf dem Stack abgelegt.
4. Der Operator `OP_EQUAL` vergleicht `H` mit H_R. Sind beide gleich, ist das Ergebnis wahr (`TRUE`), das Skript ist abgeschlossen und die Zahlung verifiziert.

HTLCs von Alice zu Dina propagieren

Alice will nun den HTLC über das Netzwerk propagieren, bis er Dina erreicht.

In Abbildung 8-9 sehen wir, wie der HTLC über das Netzwerk von Alice an Dina propagiert wird. Alice hat Bob ein HTLC mit 50.200 Satoshi übergeben. Bob kann nun ein HTLC mit 50.100 Satoshi erzeugen und an Chan übergeben.

Bob weiß, dass Chan Bobs HTLC ohne das Secret nicht einlösen kann. Bob kann dieses Secret seinerseits verwenden, um Alice' HTLC einzulösen. Das ist ein wirklich wichtiger Punkt, da er die Ende-zu-Ende-Atomizität des HTLC sicherstellt.

Um den HTLC einlösen zu können, muss das Secret veröffentlicht werden, was dann auch den anderen ermöglicht, ihre HTLCs einzulösen. Entweder können alle HTLCs eingelöst werden oder keiner: *Atomizität*!

Da Alice' HTLC 100 Satoshi höher ausfällt als der HTLC von Bob für Chan, erhält Bob 100 Satoshi als Routing-Gebühr, sobald die Zahlung abgeschlossen ist.

Bob geht kein Risiko ein und vertraut weder Alice noch Chan. Stattdessen vertraut Bob darauf, dass eine signierte Transaktion zusammen mit dem Secret in der Bitcoin-Blockchain eingelöst werden kann.

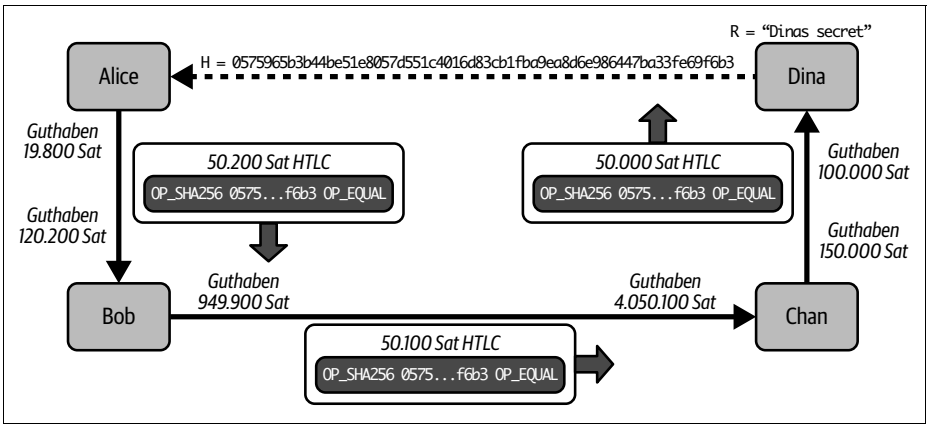

Abbildung 8-9: Einen HTLC durch das Netzwerk propagieren

Einen vergleichbaren HTLC über 50.000 Satoshi schickt Chan an Dina. Er riskiert nichts und muss weder Bob noch Dina vertrauen. Um den HTLC einzulösen, muss Dina das Secret veröffentlichen, das Chan nutzen kann, um Bobs HTLC einzulösen. Auch Chan erhält 100 Satoshi als Routing-Gebühr.

Rückpropagation des Secrets

Sobald Dina einen 50.000-HTLC von Chan erhalten hat, kann sie bezahlt werden. Dina könnte diesen HTLC einfach On-Chain bestätigen und einlösen, indem sie das Secret in der Spending-Transaktion veröffentlicht. Alternativ kann Dina ihr Kanalguthaben mit Chan aktualisieren, indem sie ihm das Secret mitteilt. Es gibt keinen Grund, die Transaktionsgebühr zu übernehmen und On-Chain zu gehen. Darum sendet Dina das Secret einfach an Chan, und sie einigen sich darauf, das Kanalguthaben zu aktualisieren, sodass es die Lightning-Zahlung von 50.000 Satoshi an Dina widerspiegelt. In Abbildung 8-10 sehen wir, dass Dina das Secret an Chan übergibt und damit den HTLC erfüllt.

Beachten Sie, dass Dinas Kanalguthaben von 50.000 Satoshi auf 100.000 Satoshi steigt. Chans Kanalguthaben sinkt von 200.000 Satoshi auf 150.000 Satoshi. Die Kanalkapazität ändert sich nicht, aber 50.000 wurden von Chans Seite des Kanals auf Dinas Seite des Kanals verschoben.

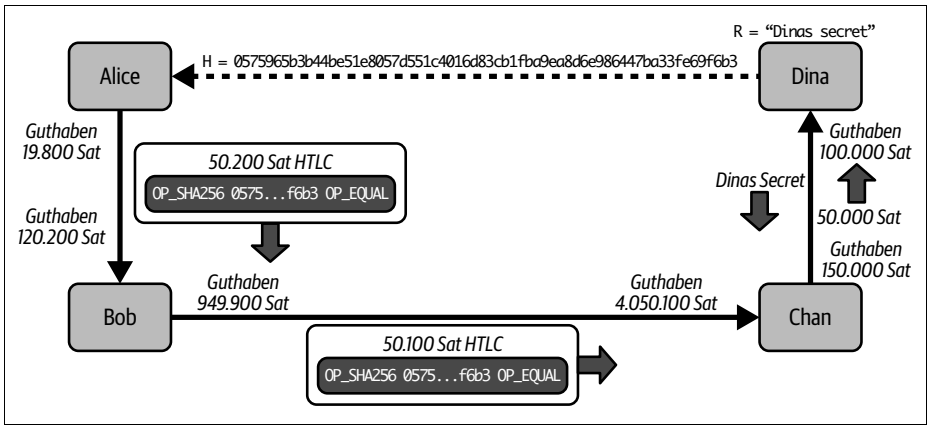

Abbildung 8-10: Dina schließt Chans HTLC Off-Chain ab.

Chan kennt nun das Secret und hat Dina 50.000 Satoshi gezahlt. Er kann das ohne Risiko tun, weil das Secret es ihm erlaubt, Bobs HTLC im Wert von 50.100 Satoshi einzulösen. Chan hat die Möglichkeit, den HTLC On-Chain zu stellen und das Secret in der Bitcoin-Blockchain zu veröffentlichen. Doch genau wie Dina will er die Transaktionsgebühren vermeiden. Stattdessen sendet er das Secret an Bob, um die Kanalguthaben so zu aktualisieren, dass sie die Lightning-Zahlung von 50.100 Satoshi von Bob an Chan widerspiegeln. In Abbildung 8-11 sendet Chan das Secret an Bob und empfängt im Gegenzug die Zahlung.

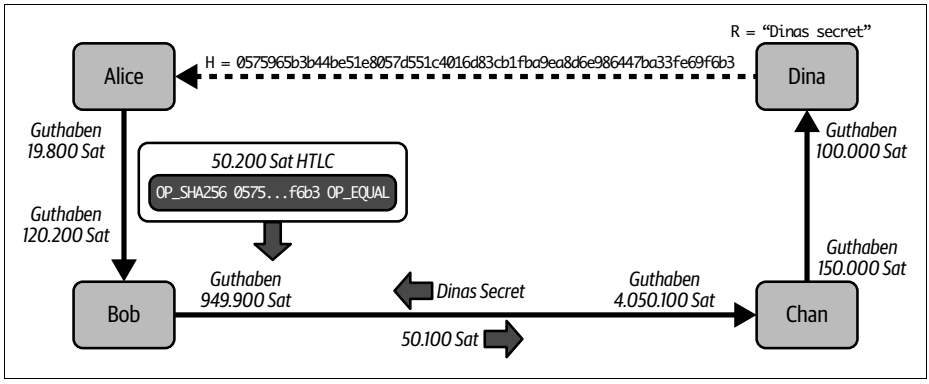

Abbildung 8-11: Chan schließt Bobs HTLC Off-Chain ab.

Chan hat Dina 50.000 Satoshi überwiesen und 50.100 Satoshi von Bob empfangen. Chans Kanalguthaben hat sich also um 100 Satoshi erhöht, die er als Routing-Gebühr erhalten hat.

Bob besitzt nun ebenfalls das Secret. Er kann es nutzen, um Alice' HTLC On-Chain einzulösen, oder er vermeidet die Transaktionsgebühren, indem er den HTLC im Kanal mit Alice abschließt. In Abbildung 8-12 sendet Bob das Secret an Alice, und beide aktualisieren das Kanalguthaben, damit es die Lightning-Zahlung von 50.200 Satoshi von Alice an Bob widerspiegelt.

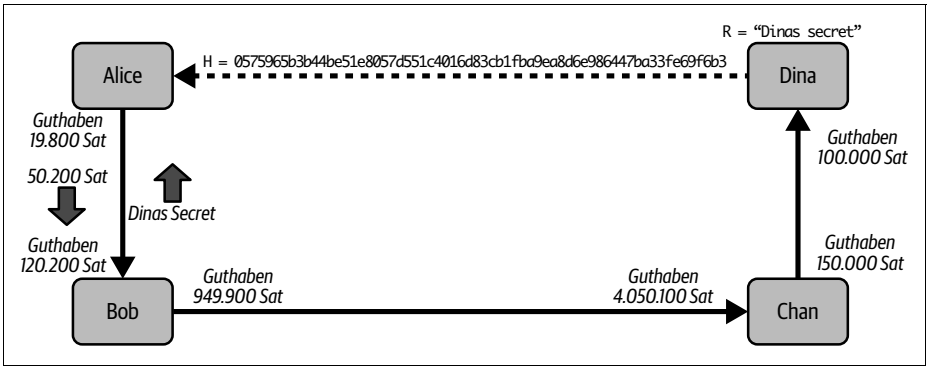

Abbildung 8-12: Bob schließt Alice' HTLC Off-Chain ab.

Bob hat 50.200 Satoshi von Alice erhalten und 50.100 Satoshi an Chan gezahlt. Sein Kanalguthaben weist nun 100 Satoshi mehr auf, die er als Routing-Gebühr erhalten hat.

Alice hat das Secret empfangen und den HTLC über 50.200 Satoshi abgeschlossen. Das Secret kann nun als *Quittung* verwendet werden, d.h., Alice kann nachweisen, dass Dina Geld für diesen Zahlungshash erhalten hat.

Die abschließenden Kanalguthaben in Abbildung 8-13 spiegeln Alice' Zahlung an Dina wider sowie die Routing-Gebühren, die für jeden Hop bezahlt wurden.

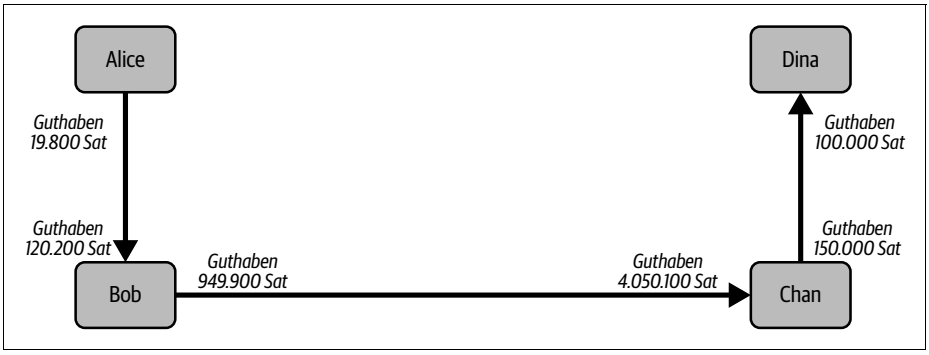

Abbildung 8-13: Kanalguthaben nach der Zahlung

Signaturbindung: Diebstahl von HTLCs verhindern

Es gibt einen Haken. Haben Sie ihn bemerkt?

Wenn Alice, Bob und Chan wie in Abbildung 8-13 gezeigt die HTLCs erzeugen, gibt es ein kleines, aber nicht unerhebliches Verlustrisiko. Jeder dieser HTLCs kann von jedem eingelöst (ausgegeben) werden, der das Secret kennt. Anfänglich kennt nur Dina das Secret. Dina soll nur den HTLC von Chan einlösen. Doch Dina könnte alle drei HTLCs gleichzeitig einlösen, sogar in einer einzigen Transaktion!

Schließlich kennt Dina das Secret vor allen anderen. Auch Chan soll eigentlich nur Bobs HTLC einlösen, sobald er das Secret kennt. Doch was wäre, wenn Chan auch Alice' HTLC einlöst?

Das ist nicht *vertrauensfrei*! Das wichtigste Sicherheitsmerkmal ist ausgehebelt. Wir müssen das korrigieren.

Das HTLC-Skript muss eine zusätzliche Bedingung enthalten, die jeden HTLC an einen bestimmten Empfänger bindet. Wir verlangen daher eine digitale Signatur, die dem öffentlichen Schlüssel des jeweiligen Empfängers entspricht. Auf diese Weise verhindern wir, dass eine andere Person den HTLC einlösen kann. Da nur der designierte Empfänger eine digitale Signatur für den fraglichen öffentlichen Schlüssel erzeugen kann, ist auch nur er in der Lage, den HTLC einzulösen.

Mit dieser Modifikation im Hinterkopf wollen wir uns die Skripte noch einmal ansehen. Alice' HTLC für Bob wird um Bobs öffentlichen Schlüssel und den `OP_CHECKSIG`-Operator ergänzt.

Hier das modifizierte HTLC-Skript:

```
OP_SHA256 <H> OP_EQUALVERIFY <Bobs öffentlicher Schlüssel> OP_CHECKSIG
```

Beachten Sie, dass zusätzlich `OP_EQUAL` in `OP_EQUALVERIFY` geändert wurde. Hat ein Operator die Endung `VERIFY`, gibt er weder `TRUE` noch `FALSE` auf dem Stack zurück. Stattdessen wird die Ausführung angehalten, und das Skript schlägt bei einem falschen Ergebnis fehl. Bei Erfolg wird weitergemacht, ohne dass etwas auf dem Stack abgelegt wird.

Um diesen HTLC einzulösen, muss Bob ein Unlocking-Skript präsentieren, das eine Signatur aus Bobs privatem Schlüssel sowie dem Preimage enthält:

```
<Bobs Signatur> <R>
```

Die Unlocking- und die Locking-Skripte werden kombiniert und wie folgt von der Skripting-Engine ausgewertet:

```
<Bobs Signatur> <R> OP_SHA256 <H> OP_EQUALVERIFY <Bobs öffentlicher Schlüssel> OP_CHECKSIG
```

1. `<Bobs Signatur>` wird auf dem Stack abgelegt.
2. `R` wird auf dem Stack abgelegt.
3. `OP_SHA256` nimmt `R` vom Stack und wendet die Hashfunktion darauf an. Das Ergebnis H_R wird auf dem Stack abgelegt.
4. `H` wird auf dem Stack abgelegt.
5. `OP_EQUALVERIFY` nimmt `H` und H_R vom Stack und vergleicht sie. Sind sie nicht gleich, wird die Ausführung angehalten. Andernfalls wird die Ausführung fortgesetzt, ohne etwas auf dem Stack abzulegen.
6. `<Bobs öffentlicher Schlüssel>` wird auf dem Stack abgelegt.
7. `OP_CHECKSIG` entfernt `<Bobs Signatur>` sowie `<Bobs öffentlichen Schlüssel>` und vergleicht die Signatur. Das Ergebnis (`TRUE/FALSE`) wird auf dem Stack abgelegt.

Wie Sie sehen, ist das etwas komplizierter, behebt aber unser HTLC-Problem und stellt sicher, dass nur der vorgesehene Empfänger ihn einlösen kann.

Hashoptimierung

Sehen wir uns an, wie der erste Teil des HTLC-Skripts bisher aussieht:

```
OP_SHA256 <H> OP_EQUALVERIFY
```

In der obigen symbolischen Darstellung sieht es so aus, als würden die `OP_`-Operatoren den größten Raum einnehmen. Doch das ist nicht der Fall. Bitcoin-Skripte sind binär codiert, und jeder Operator wird durch ein Byte repräsentiert. Gleichzeitig ist der `<H>`-Wert, den wir als Platzhalter für den Zahlungshash verwenden, 32 Byte (256 Bit) lang. Eine Liste aller Operatoren von Bitcoin-Skript samt binärer und hexadezimaler Codierung finden Sie bei Bitcoin Wiki: Script (*https://en.bitcoin.it/wiki/Script*) oder in Appendix D, »Transaction Script Language Operators, Constants, and Symbols,« in *Mastering Bitcoin* (*https://github.com/bitcoinbook/bitcoinbook/blob/develop/appdx-scriptops.asciidoc*).

Stellen wir es hexadezimal dar, sieht unser HTLC-Skript wie folgt aus:

```
a8 0575965b3b44be51e8057d551c4016d83cb1fba9ea8d6e986447ba33fe69f6b3 88
```

In hexadezimaler Codierung hat `OP_SHA256` den Wert `a8` und `OP_EQUALVERIFY` den Wert `88`. Die Gesamtlänge des Skripts beträgt 34 Byte, davon 32 Byte für den Hash.

Wie bereits erwähnt, muss jeder Teilnehmer im Lightning-Netzwerk in der Lage sein, eine gehaltene Off-Chain-Transaktion On-Chain zu bringen, um seinen Anspruch auf seine Einlage durchsetzen zu können. Um eine Transaktion On-Chain zu bringen, muss er die Transaktionsgebühr an die Miner zahlen, die sich proportional zur Größe der Transaktion in Bytes berechnet.

Daher wollen wir Wege finden, das »Gewicht« der On-Chain-Transaktionen zu verringern, indem wir das Skript so weit wie möglich optimieren. Eine Möglichkeit besteht hier darin, eine weitere Hashfunktion über den SHA-256-Algorithmus zu legen, die kleinere Hashes erzeugt. Bitcoin-Skript stellt den Operator `OP_HASH160` bereit, der ein Preimage »doppelt hasht«: Zuerst wird das Preimage mit SHA-256 gehasht, und der daraus resultierende Hash wird dann mit dem Hashalgorithmus RIPEMD160 noch einmal gehasht. Der von RIPEMD160 erzeugte Hash ist 160 Bit bzw. 20 Byte lang, also wesentlich kompakter. Bei Bitcoin-Skript ist das eine weitverbreitete Optimierung, die für viele gängige Adressformate verwendet wird.

Auch wir wollen diese Optimierung nutzen. Unser SHA-256-Hash lautet `057596...69f6b3`. Lassen wir ihn durch die Hashfunktion `RIPEMD160` laufen, erhalten wir das folgende Ergebnis:

```
R = "Dinas secret"
H256 = SHA256(R)
H256 = 0575965b3b44be51e8057d551c4016d83cb1fba9ea8d6e986447ba33fe69f6b3
H160 = RIPEMD160(H256)
H160 = 9e017f6767971ed7cea17f98528d5f5c0ccb2c71
```

Alice kann den RIPEMD160-Hash des von Dina bereitgestellten Zahlungshashs erzeugen und diesen kürzeren Hash in ihrem HTLC nutzen, genau wie Bob und Chan!

Das »optimierte« HTLC-Skript sieht dann wie folgt aus:

```
OP_HASH160 <H160> OP_EQUALVERIFY
```

Hexadezimal codiert, ergibt das:

```
a9 9e017f6767971ed7cea17f98528d5f5c0ccb2c71 88
```

Dabei hat OP_HASH160 den Wert a9 und OP_EQUALVERIFY den Wert 88. Dieses Skript ist nur 22 Byte lang! Wir sparen 12 Byte bei jeder Transaktion, die einen HTLC On-Chain einlöst.

Mit dieser Optimierung enden wir bei dem HTLC-Skript, das wir in Zeile 10 von Beispiel 8-1 gesehen haben:

```
...
    # To local node via HTLC-success transaction.
    OP_HASH160 <RIPEMD160(payment_hash)> OP_EQUALVERIFY...
```

Kooperations- und Timeout-Fehler

Bisher haben wir uns nur den »Hash«-Teil des HTLC angesehen und wie er funktioniert, wenn alle kooperieren und zum Zeitpunkt der Zahlung online sind.

Was passiert aber, wenn jemand offline geht oder nicht kooperiert? Was passiert, wenn die Zahlung nicht abgeschlossen werden kann?

Wir müssen sicherstellen, dass solche Fehler »in Würde« abgefangen werden, weil gelegentliche Routing-Fehler unvermeidbar sind. Es gibt zwei Arten von Fehlern: Kooperation und Rückerstattung mit Timelock.

Kooperationsfehler sind relativ einfach zu beheben: Der HTLC wird von jedem Teilnehmer entlang der Route rückabgewickelt, d.h., die HTLC-Outputs werden aus ihren Commitment-Transaktionen gelöscht, ohne das Guthaben zu verändern. Wie das genau funktioniert, erklären wir in Kapitel 9.

Sehen wir uns an, wie wir einen HTLC rückgängig machen können, ohne mit einem oder mehreren Teilnehmern kooperieren zu müssen. Wir müssen sicherstellen, dass die Mittel in einem HTLC nicht *für immer* gesperrt werden, falls einer der Teilnehmer nicht kooperiert. Andernfalls wäre es möglich, von anderen Teilnehmern Lösegeld zu fordern: »Dein Guthaben bleibt für immer eingefroren, wenn du mir kein Lösegeld zahlst.«

Um das zu verhindern, enthält jedes HTLC-Skript eine Rückerstattungsklausel, die an einen Timelock gekoppelt ist. Erinnern Sie sich an unseren ursprünglichen Treuhändervertrag? »Bob hat nach Unterzeichnung des Vertrags 24 Stunden Zeit, das Secret vorzulegen. Kann Bob das Secret innerhalb dieser Zeit nicht vorlegen, wird Alice' Einlage zurückerstattet.«

Die zeitgesteuerte Rückerstattung ist ein wichtiger Teil des Skripts, der die *Atomizität* sicherstellt. Die Ende-zu-Ende-Zahlung wird entweder erfolgreich abgeschlossen oder sauber rückabgewickelt. Es gibt kein »halb bezahlt«, um das man sich Gedanken machen müsste. Bei einem Fehler kann jeder Teilnehmer den HTLC kooperativ mit seinem Kanalpartner rückabwickeln oder einseitig eine zeitbasierte Rückerstattungstransaktion On-Chain stellen, um sein Geld zurückzuerhalten.

Um diese Rückerstattung in Bitcoin-Skript zu implementieren, nutzen wir den speziellen Operator `OP_CHECKLOCKTIMEVERIFY` oder kurz `OP_CLTV`. Hier das Skript, das wir aus Zeile 13 von Beispiel 8-1 kennen:

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

Der Operator `OP_CLTV` erwartet ein Ablaufdatum, definiert als Blockhöhe ab der Gültigkeit dieser Transaktion. Ist der Transaktions-Timelock nicht mit `<cltv_expiry>` identisch, schlägt die Evaluierung des Skripts fehl, und die Transaktion ist ungültig. Andernfalls läuft das Skript weiter, ohne einen Output auf dem Stack abzulegen. Denken Sie daran, dass die Endung `VERIFY` bedeutet, dass der Operator kein `TRUE` oder `FALSE` ausgibt, sondern einfach angehalten oder, ohne einen Wert auf dem Stack abzulegen, fortgesetzt wird.

`OP_CLTV` fungiert im Grunde genommen als »Torwächter«, der das Skript daran hindert, fortzufahren, wenn die Blockhöhe `<cltv_expiry>` auf der Bitcoin-Blockchain noch nicht erreicht wurde.

Der Operator `OP_DROP` entfernt einfach das oberste Element vom Skript-Stack. Das ist zu Beginn notwendig, weil noch ein »Überbleibsel« der vorherigen Skriptzeile übrig ist. Es ist *nach* `OP_CLTV` nötig, den `<cltv_expiry>`-Wert vom Stack zu entfernen, weil er nicht länger benötigt wird.

Nachdem der Stack bereinigt ist, sollten ein öffentlicher Schlüssel und eine Signatur übrig sein, die `OP_CHECKSIG` verifizieren kann. Wie in »Signaturbindung: Diebstahl von HTLCs verhindern« auf Seite 230 gesehen, muss sichergestellt werden, dass nur der rechtmäßige Besitzer die Mittel abrufen kann. Zu diesem Zweck wird der Output an dessen öffentlichen Schlüssel gebunden, und eine Signatur wird benötigt.

Kürzere Timelocks

Während die HTLCs von Alice zu Dina wandern, erhält die zeitgesteuerte Rückerstattungsklausel jedes HTLC einen anderen `cltv_expiry`-Wert. Wir erläutern das in Kapitel 10 genauer. Für den Moment reicht es, zu wissen, dass für eine ordentliche Rückabwicklung einer fehlgeschlagenen Zahlung jeder Hop ein kleines bisschen kürzer auf seine Rückerstattung warten muss. Der Unterschied zwischen den

Timelocks der einzelnen Hops wird `cltv_expiry_delta` genannt. Er wird von jeder Node gesetzt und im Netzwerk verbreitet, wie wir in Kapitel 11 sehen werden.

Alice setzt beispielsweise den Rückerstattungs-Timelock des ersten HTLC auf die Blockhöhe »Aktuell + 500 Blöcke« (wobei »Aktuell« für die aktuelle Blockhöhe steht). Bob würde `cltv_expiry` im HTLC mit Chan auf »Aktuell + 450« setzen, und Chan würde seinen Timelock mit »Aktuell + 400« angeben. Auf diese Weise erhält Chan seine Rückerstattung für den HTLC mit Dina, *bevor* Bob eine Rückerstattung für seinen HTLC an Chan bekommt. Bob erhält eine Rückerstattung für den HTLC mit Chan, *bevor* Alice ihre Rückerstattung für den HTLC mit Bob erhält. Der verkürzte Timelock verhindert Race-Conditions und stellt sicher, dass die HTLC-Kette vom Ziel zur Quelle rückabgewickelt wird.

Fazit

In diesem Kapitel haben Sie gesehen, wie Alice Dina bezahlen kann, auch wenn es keinen direkten Zahlungskanal gibt. Alice kann einen Weg finden, der sie mit Dina verbindet, und eine Zahlung über mehrere Zahlungskanäle routen, damit sie Dina erreicht.

Damit die Zahlung über mehrere Hops atomar und vertrauensfrei bleibt, muss Alice mit allen zwischengeschalteten Nodes des Pfads ein Fairness-Protokoll implementieren. Das Fairness-Protokoll ist momentan als HTLC implementiert, das Mittel an einen Zahlungshash bindet, der aus einem geheimen Zahlungs-Preimage abgeleitet wird.

Jeder Teilnehmer der Zahlungsroute leitet einen HTLC an den nächsten Teilnehmer weiter, ohne sich um Diebstahl oder eingefrorene Guthaben Gedanken machen zu müssen. Der HTLC kann eingelöst werden, indem man das geheime Zahlungs-Preimage vorlegt. Sobald ein HTLC Dina erreicht, legt sie das Preimage offen, das rückwärts laufend alle verwendeten HTLCs auflöst.

Abschließend haben wir den HTLC mit einer zeitgesteuerten Rückerstattungsklausel vervollständigt. Sie stellt sicher, dass der HTLC rückabgewickelt wird und jeder Teilnehmer sein Geld zurückerhält, wenn die Zahlung (aus welchem Grund auch immer) fehlschlägt oder wenn einer der Teilnehmenden nicht kooperiert. Da man immer die Möglichkeit hat, für eine Rückerstattung On-Chain zu gehen, erreicht der HTLC sein Fairness-Ziel eines atomaren und vertrauensfreien Betriebs.