Contract

From Bitcoin Wiki

A **distributed contract** is a method of using Bitcoin to form agreements with people via the block chain. Contracts don't make anything possible that was previously impossible, but rather, they allow you to solve common problems in a way that minimizes trust. Minimal trust often makes things more convenient by allowing human judgements to be taken out of the loop, thus allowing complete automation.

By building low trust protocols that interact with Bitcoin, entirely new products can be created:

- Smart property is property that can be atomically traded and loaned via the block chain.
- Transferable virtual property are digital items that can be traded but not duplicated.
- Agents are autonomous programs that maintain their own wallet, which they use to buy server time. Money is obtained by the agent selling services. If demand exceeds supply the agents can spawn children that either survive or die depending on whether they can get enough business.
- Distributed markets are a way to implement peer to peer bond and stock trading, allowing Bitcoin to be evolve into a full competitor to the international financial system.

This page also lists some smaller examples.

Many of the ideas underlying Bitcoin contracts were first described by Nick Szabó in his seminal paper, Formalizing and Securing Relationships on Public Networks (http://szabo.best.vwh.net/formalize.html). These pages were written by Mike Hearn (mailto:mike@plan99.net). Contact him if you have an idea for a new type of contract. You can watch **a video of a talk on contracts (https://www.youtube.com/watch?feature=player_embedded&v=mD4L7xDNCmA)** that was presented at the Bitcoin 2012 conference in London.

Contents

A warning about the mempool transaction replacement mechanism

In places this page refers to the ability to use the nSequence field for transaction mempool replacement. This mechanism was disabled in 2010 (https://github.com/bitcoin/bitcoin/commit/05454818dc7ed92f577a1a1ef6798049f17a52e7#diff-118fcbaaba162ba17933c7893247df3aR522), and more recently the code has been removed completely (https://github.com/bitcoin/bitcoin/commit/98c7c8fd1d3712e02be0e9f2eeca7e02aa54d197), due to concerns over people using it to perform DoS attacks. Implementors should take this into account and try to create contract mechanisms that do not rely on mempool replacement if they wish to have their implementations work with current implementations. If Bitcoin changes in future to allow mempool replacement once again, this page will be updated.

Theory

Every transaction in Bitcoin has one or more inputs and outputs. Each input/output has a small, pure function associated with it called a script. Scripts can contain signatures over simplified forms of the transaction itself.

Every transaction can have a lock time associated with it. This allows the transaction to be pending and replaceable until an agreed-upon future time, specified either as a block index or as a timestamp (the same field is used for both, but values less than 500 million are interpreted as a block index). If a transaction's lock time has been reached, we say it is final.

Each transaction input has a sequence number. In a normal transaction that just moves value around, the sequence numbers are all UINT_MAX and the lock time is zero. If the lock time has not yet been reached, but all the sequence numbers are UINT_MAX, the transaction is also considered final. Sequence numbers can be used to issue new versions of a transaction without invalidating other inputs signatures, e.g., in the case where each input on a transaction comes from a different party, each input may start with a sequence number of zero, and those numbers can be incremented independently.

Signature checking is flexible because the form of transaction that is signed can be controlled through the use of SIGHASH flags, which are stuck on the end of a signature. In this way, contracts can be constructed in which each party signs only a part of it, allowing other parts to be changed without their involvement. The SIGHASH flags have two parts, a mode and the ANYONECANPAY modifier:

1. SIGHASH_ALL: This is the default. It indicates that everything about the transaction is signed, except for the input scripts. Signing the input scripts as well would obviously make it impossible to construct a transaction, so they are always blanked out. Note, though, that other properties of the input, like the connected output and sequence numbers, *are* signed; it's only the scripts that are not. Intuitively, it means "I agree to put my money in, if everyone puts their money in and the outputs are this".
2. SIGHASH_NONE: The outputs are not signed and can be anything. Use this to indicate "I agree to put my money in, as long as everyone puts their money in, but I don't care what's done with the output". This mode allows others to update the transaction by changing their inputs sequence numbers.
3. SIGHASH_SINGLE: Like SIGHASH_NONE, the inputs are signed, but the sequence numbers are blanked, so others can create new versions of the transaction. However, the only output that is signed is the one at the same position as the input. Use this to indicate "I agree, as long as my output is what I want; I don't care about the others".

The SIGHASH_ANYONECANPAY modifier can be combined with the above three modes. When set, only that input is signed and the other inputs can be anything.

Scripts can contain the CHECKMULTISIG opcode. This opcode provides n-of-m checking: you provide multiple public keys, and specify the number of valid signatures that must be present. The number of signatures can be less than the number of public keys. An output can require two signatures to be spent by setting it to something like this:

```
2 <pubkey1> <pubkey2> 2 CHECKMULTISIGVERIFY
```

There are two general patterns for safely creating contracts:

1. Transactions are passed around outside of the P2P network, in partially-complete or invalid forms.
2. Two transactions are used: one (the contract) is created and signed but not broadcast right away. Instead, the other transaction (the payment) is broadcast after the contract is agreed to lock in the money, and then the contract is broadcast.

This is to ensure that people always know what they are agreeing to.

Together, these features let us build interesting new financial tools on top of the block chain.

Example 1: Providing a deposit

Imagine that you open an account on a website (eg, a forum or wiki) and wish to establish your trustworthiness with the operators, but you don't have any pre-existing reputation to leverage. One solution is to buy trust by paying the website some money. But if at some point you close your account, you'd probably like that money back. You may not trust the site enough to give them a deposit that they are tempted to spend. Another risk is that the site might just disappear one day.

The goal is to prove that you made a sacrifice of some kind so the site knows you're not a spambot, but you don't want them to be able to spend the money. And if the operators disappear, you'd eventually like the coins back without needing anything from them.

We can solve this problem with a contract:

1. The user and website send each other a newly-generated public key.
2. The user creates transaction Tx1 (the payment) putting 10 BTC into an output that requires both user and website to sign, but does not broadcast it. They use the key from the previous step for the site.
3. User sends the hash of Tx1 to the website.
4. The website creates a transaction Tx2 (the contract). Tx2 spends Tx1 and pays it back to the user via the address he provided in the first step. Note that Tx1 requires two signatures, so this transaction can't be complete. nLockTime is set to some date in the future (eg, six months). The sequence number on the input is set to zero.
5. Finally, the incomplete (half-signed) transaction is sent back to the user. The user checks that the contract is as expected - that the coins will eventually come back to him - but, unless things are changed, only after six months. Because the sequence number is zero, the contract can be amended in future if both parties agree. The script in the input isn't finished though; there are only zeros where the user's signature should be. He fixes that by signing the contract and putting the new signature in the appropriate spot.
6. The user broadcasts Tx1, then broadcasts Tx2.

At this stage, the 10 BTC are in a state where neither the user nor the website can spend them independently. After six months, the contract will complete and the user will get the coins back, even if the website disappears.

What if the user wishes to close his account early? The website creates a new version of Tx2 with nLockTime set to zero and the input sequence number set to UINT_MAX, then he re-signs it. The site hands the tx back to the user, who signs it as well. The user then broadcasts the transaction, terminating the contract early and releasing the coins.

What if the six months is nearly up and the user wishes to keep his account? The same thing applies: the contract can be resigned with a newer nLockTime, a sequence number 1 higher than the previous and rebroadcast 2^32 times. No matter what happens, both parties must agree for the contract to change.

Obviously, if the user turns out to be abusive (i.e., a spammer), the website will not allow an early close of the contract. If too much abuse is getting through, the size of the deposit can be raised or the length of the contract can be increased.

Example 2: Escrow and dispute mediation

A buyer wants to trade with somebody he doesn't know or trust. In the common case where the transaction goes well, the client doesn't want any third parties involved. If something goes wrong though, he'd like a third party to decide who gets the money - perhaps a professional dispute mediation service. Note that this concept can apply to either buyer or seller. The mediator might request proof of postage from the merchant, for example.

In other words, one wants to lock up some coins so a third party has to agree in order for them to be spent:

1. Agree with the merchant on a dispute mediator (e.g., ClearCoin).
2. Ask the merchant for a public key (K1). Ask the mediator for a public key (K2). Create a new key for yourself (K3).
3. Send the merchant K2. The merchant challenges the mediator with a random nonce. The mediator signs the nonce with the private form of K2, thus proving it really belongs to merchant.
4. Create a transaction (Tx1) with an output script as follows and broadcast it:

```
2 <K1> <K2> <K3> 3 CHECKMULTISIGVERIFY
```

Now the coins are locked in such a way that they can only be spent by the following methods:

1. Client and the merchant agree (either a successful trade, or merchant agrees to reimburse client without mediation)
2. Client and the mediator agree (failed trade, mediator sides with client, like a charge-back)
3. The mediator and the merchant agree (goods delivered, merchant gets client's coins despite the dispute)

When signing an input, the contents are set to the connected output. Thus, to redeem this transaction, the client creates a scriptSig containing zeros where the other signature should be, signs it, and then sets one of the slots to his new signature. The partially-complete transaction can then be sent to the merchant or mediator for the second signature.

Example 3: Assurance contracts

An assurance contract (http://en.wikipedia.org/wiki/Assurance_contract) is a way of funding the creation of a public good (http://www.auburn.edu/~johnspm/gloss/public_goods), that is, a good that, once created, anyone can benefit from for free. The standard example is a lighthouse: whilst everyone may agree that one should be built, it’s too expensive for an individual sailor to justify building one, given that it will also benefit all his competitors.

One solution is for everyone to pledge money towards the creation of the public good, such that the pledges are only committed if the total value of all pledges is above the cost of creation. If not enough people contribute, nobody has to pay anything.

Examples where Bitcoin is superior to traditional payment methods for assurance contract fundraising include applications where frequent, small pledges need to be made automatically, for instance internet radio station funding and web page translation (https://bitcointalk.org/index.php?topic=67255.msg788110#msg788110). Consider a browser extension that you send a bit of money to. It detects the current language of the page and broadcasts a pledge for a translation into your language to be prepared. If enough users with the extension land on the same page at the same time (eg, it was linked to from somewhere high traffic), then enough pledges are broadcast to trigger a payment to a company who prepares a high quality translation. When complete it automatically loads in your browser.

We can model this in Bitcoin as follows:

1. An entrepreneur creates a new address and announces that the good will be created if at least 1000 BTC is raised. Anyone can contribute.
2. Each party wishing to pledge creates a new transaction spending some of their coins to the announced address, but they do not broadcast it. The transaction is similar to a regular transaction except for three differences: Firstly, there cannot be any change. If you don’t have any outputs of the right size, you must create one first by spending to one of your own addresses. Secondly, the input script signature is signed with SIGHASH_ALL | SIGHASH_ANYONECANPAY. Finally, the output value is set to 1000 BTC. Note that this is not a valid transaction because the output value is larger than the input value.
3. The transaction is uploaded to the entrepreneur's server, which saves it to disk and updates its count of how many coins have been pledged.
4. Once the server has enough coins, it merges the separate transactions together into a new transaction. The new transaction has a single output that simply spends to the announced address - it is the same as the outputs on each contributed transaction. The inputs to the transaction are collected from the contributed pledges.
5. The finished transaction is broadcast, sending the pledged coins to the announced address.

This scheme relies on several aspects of the protocol. The first is the SIGHASH flags used. SIGHASH_ALL is the default and means the entire contents of the transaction are signed, except for the input scripts. SIGHASH_ANYONECANPAY is an additional modifier that means the signature only covers the input it’s found in - the other inputs are not signed and thus can be anything.

By combining these flags together, we are able to create a signature that is valid even when other inputs are added, but breaks if the outputs or other properties of the transaction are changed.

The second aspect we exploit is the fact that a transaction in which the output values are larger than the input values is invalid (for obvious reasons). This means it’s safe to send the entrepreneur a transaction that spends such coins - it’s impossible for him to claim the coins unless he has other inputs that sum to the output value or more.

It's possible to create assurance contracts without using SIGHASH_ANYONECANPAY. Instead, a two step process is used in which pledges are collected without transactions, and once the total value is reached, a transaction with an input for each pledger is created and passed around until all signatures are collected. However, using SIGHASH_ANYONECANPAY and then merging is probably more convenient.

An assurance contract can be prepared for **funding network security** for the next block. In this way, mining can be funded even if block space is not scarce.

An elaboration of this concept is described by Tabarrok in his paper, "The private provision of public goods via dominant assurance contracts" (http://mason.gmu.edu/~atabarro/PrivateProvision.pdf). In a dominant assurance contract, if a contract fails (not enough pledges within a set time window) the entrepreneur pays a fee to those who pledged so far. This type of contract attempts to arrange incentives such that taking part is always the right strategy. A scheme for dominant assurance contracts in Bitcoin has been proposed.

Example 4: Using external state

Scripts are, by design, pure functions. They cannot poll external servers or import any state that may change as it would allow an attacker to outrun the block chain. What's more, the scripting language is extremely limited in what it can do. Fortunately, we can make transactions connected to the world in other ways.

Consider the example of an old man who wishes to give an inheritance to his grandson, either on the grandson's 18th birthday or when the man dies, whichever comes first.

To solve this, the man first sends the amount of the inheritance to himself so there is a single output of the right amount. Then he creates a transaction with a lock time of the grandson's 18th birthday that pays the coins to another key owned by the grandson, signs it, and gives it to him - but does not broadcast it. This takes care of the 18th birthday condition. If the date passes, the grandson broadcasts the transaction and claims the coins. He could do it before then, but it doesn't let him get the coins any earlier, and some nodes may choose to drop transactions in the memory pool with lock times far in the future.

The death condition is harder. As Bitcoin nodes cannot measure arbitrary conditions, we must rely on an *oracle*. An oracle is a server that has a keypair, and signs transactions on request when a user-provided expression evaluates to true.

Here is an example. The man creates a transaction spending his output, and sets the output to:

```
<hash> OP_DROP 2 <sons pubkey> <oracle pubkey> CHECKMULTISIG
```

This is the oracle script. It has an unusual form - it pushes data to the stack then immediately deletes it again. The pubkey is published on the oracle's website and is well-known. The hash is set to be the hash of the user-provided expression stating that he has died, written in a form the oracle knows how to evaluate. For example, it could be the hash of the string:

```
if (has_died('john smith', born_on=1950/01/02)) return (10.0, 1JxgRXEHBi86zYzHN2U4KMyRCg4LvwNUrp);
```

This little language is hypothetical, it'd be defined by the oracle and could be anything. The return value is an output: an amount of value and an address owned by the grandson.

Once more, the man creates this transaction but gives it directly to his grandson instead of broadcasting it. He also provides the expression that is hashed into the transaction and the name of the oracle that can unlock it.

It is used in the following algorithm:

1. The oracle accepts a measurement request. The request contains the user-provided expression, a copy of the output script, and a partially complete transaction provided by the user. Everything in this transaction is finished except for the scriptSig, which contains just one signature (the grandson's) - not enough to unlock the output.
2. The oracle checks the user-provided expression hashes to the value in the provided output script. If it doesn't, it returns an error.
3. The oracle evaluates the expression. If the result is not the destination address of the output, it returns an error.
4. Otherwise the oracle signs the transaction and returns the signature to the user. Note that when signing a Bitcoin transaction, the input script is set to the connected output script. The reason is that when OP_CHECKSIG runs, the script containing the opcode is put in the input being evaluated, _not_ the script containing the signature itself. The oracle has never seen the full output it is being asked to sign, but it doesn't have to. It knows the output script, its own public key, and the hash of the user-provided expression, which is everything it needs to check the output script and finish the transaction.
5. The user accepts the new signature, inserts it into the scriptSig and broadcasts the transaction.

If, and only if, the oracle agrees that the man is dead, the grandson can broadcast the two transactions (the contract and the claim) and take the coins.

Oracles can potentially evaluate anything, yet the output script form in the block chain can always be the same. Consider the following possibilities:

```
today() == 2011/09/25 && exchange_rate(mtgoxUSD) >= 12.5 && exchange_rate(mtgoxUSD) <= 13.5
Require exchange rate to be between two values on a given date

google_results_count(site:www.google.com/hostednews 'Mike Hearn' olympic gold medal) > 0
A bet on me doing something that I will never actually do

// Choose between one of two winners of a bet on the outcome of the Eurovision song contest.
if (eurovision_winner() == 'Azerbaijan')
```

```
  return 1Lj9udBVDwptFffGSJSC2sohCfudQgSTPD;
else
  return 1JxgRXEHBi86zYzHN2U4KMyRCg4LvwNUrp;
```

The conditions that control whether the oracle signs can be arbitrarily complex, but the block chain never needs to contain more than a single hash.

The *Early Temple* project (http://earlytemple.com/) has implemented a prototype of an oracle that looks for a key phrase in a web page.

Trust minimization: challenges

There are various ways to reduce the level of required trust in the oracle.

Going back to our first example, the oracle has not seen the transaction the grandson is trying to unlock, as it was never broadcast, thus, it cannot hold the grandson to ransom because it does not know whether the transaction it's signing for even exists. People can, and should, regularly **challenge** the oracle in an automated fashion to ensure it always outputs what is expected. The challenges can be done without spending any coins because the tx to be signed can be invalid (ie, connected to transactions that don't exist). The oracle has no way to know whether a request to be signed is random or real. How to challenge the oracle with conditions that are not yet true is however an open research question.

Trust minimization: multiple independent oracles

The number of keys in the CHECKMULTISIG can be increased to allow for **n-of-m** oracles if need be. Of course, it is vital to check that the oracles really are independent and not in collusion. An implementation of such a system, and explanation was published in the **Orisi whitepaper (http://github.com/orisi/wiki/wiki/Orisi-White-Paper)**.

The idea, in short, is that independent oracles could be run by a set of trustworthy independent actors. Contract participants would first settle upon a set of oracles they both are comfortable using - for example by trusting a dedicated site (http://oracles.lo/) - and then sign a contract requiring most of the oracles signatures to resolve.

One of the interesting properties of such a system is that it can possibly implement very complicated sets of scripts and external inputs, while being based on a set of standard bitcoin scripts. Another property is that the oracles can possibly replace each ones by forwarding the funds to new multisig addresses (https://bitcoinmagazine.com/11108/multisig-future-bitcoin/) when they discover that one of the oracles is faulty/corrupted. Finally, such oracles could be used to implement sidechains without dedicated blockchain opcodes.

Trust minimization: trusted hardware

Using commodity hardware, you can use **trusted computing** in the form of Intel TXT or the AMD equivalent (SKINIT) to set up a sealed hardware environment and then use the TPM chip to attest that fact to a third party. That third party can verify the hardware was in the required state. Defeating this requires someone to be able to interfere with the execution of a program that may run entirely on the CPU, even in extreme cases without any data traversing the memory bus (you can run entirely using on-die cache if the program is small enough).

Trust minimization: Amazon AWS oracles

Finally, perhaps the most practical approach currently is to use Amazon Web Services. As of November 2013, the closest we have to a working oracle is this recipe for creating a trusted computing environment using AWS (https://bitcointalk.org/index.php?topic=301538.0), built in support of this project for doing selective SSL logging and decryption (https://bitcointalk.org/index.php?topic=173220.0). The idea is that an oracle, which can be proven trustworthy using the Amazon APIs with Amazon as the root of trust, records encrypted SSL sessions to an online banking interface in such a way that later if there is a dispute about a person-to-person exchange, the logs can be decrypted and the dispute settled.

Example 5: Trading across chains

The Bitcoin technology can be adapted to create multiple, independent currencies. NameCoin is an example of one such currency that operates under a slightly different set of rules, and can also be used to rent names in a namespace. Currencies that implement the same ideas as Bitcoin can be traded freely against each other with limited trust.

For example, imagine a consortium of companies that issue EURcoins, a crypto-currency that is backed 1:1 by deposits in the consortium's bank accounts. Such a currency would have a different set of tradeoffs to Bitcoin: more centralized, but without FX risk. People might wish to trade Bitcoins for EURcoins back and forth, with the companies only getting involved when cashing in/out of the regular banking system.

To implement this, a protocol proposed by TierNolan (https://bitcointalk.org/index.php?topic=193281.msg2224949#msg2224949) can be used:

1. Party 'A' generates some random data, x (the secret).
2. Party 'A' generates Tx1 (the payment) containing an output with the chain-trade script in it. See below for this script and a discussion of it. It allows coin release either by signing with the two keys (key 'A' and key 'B') or with (secret 'x', key 'B'). This transaction is not broadcast. The chain release script contains hashes, not the actual secrets themselves.
3. Party 'A' generates Tx2 (the contract), which spends Tx1 and has an output going back to key 'A'. It has a lock time in the future and the input has a sequence number of zero, so it can be replaced. 'A' signs Tx2 and sends it to 'B', who also signs it and sends it back.
4. 'A' broadcasts Tx1 and Tx2. Party 'B' can now see the coins but cannot spend them because it does not have an output going to him, and the tx is not finalized anyway.

5. 'B' performs the same scheme in reverse on the alternative chain. The lock time for 'B' should be much larger than the lock time for 'A'. Both sides of the trade are now pending but incomplete.
6. Since 'A' knows the secret, 'A' can claim his coins immediately. However, 'A', in the process of claiming his coin, reveals the secret 'x' to 'B', who then uses it to finish the other side of the trade with ('x', key 'B').

This protocol makes it feasible to do trades automatically in an entirely peer-to-peer manner, which should ensure good liquidity.

The chain-trade script could look like this:

```
IF
  2 <key A> <key B> 2 CHECKMULTISIGVERIFY
ELSE
  <key B> CHECKSIGVERIFY SHA256 <hash of secret x> EQUALVERIFY
ENDIF
```

The contract input script looks like either:

```
<sig A> <sig B> 1
```

or

```
<secret x> <sig B> 0
```

i.e., the first data element selects which phase is being used.

See Atomic cross-chain trading for details.

Note that whilst EURcoins is a natural idea, there are other ways to implement peer-to-peer currency exchange (Bitcoin to fiat and vice versa), see the Ripple currency exchange article for more information.

Sergio Demian-Lerner proposed P2PTradeX (https://bitcointalk.org/index.php?topic=91843.0), a solution requiring the validation rules for one blockchain to be effectively encoded into the validation rules for the other.

Example 6: Pay-for-proof contracts: buying a solution to any pure function

In example 4, we saw how to make payments conditional on the output of some arbitrary program. Those programs are very powerful and can do anything a regular program can do, like fetching web pages. The downside is that a third party is required (an oracle). Although there are techniques that can help reduce the trust needed in the oracle, none can reduce it to zero.

For a restricted class of programs, pure functions, new cryptographic techniques are starting to become available that can actually reduce the trust needed all the way to zero with no third parties. These programs cannot perform any I/O, but in many cases this restriction turns out to be unimportant or can be worked around in other ways, like by giving the program a signed/timestamped document as an input instead of having the program download it.

Gregory Maxwell has invented a protocol for doing this, which you can read about here: Zero Knowledge Contingent Payment

Example 7: Rapidly-adjusted (micro)payments to a pre-determined party

Bitcoin transactions are very cheap relative to traditional payment systems, but still have a cost due to the need for it to be mined and stored. There are some cases in which you want to rapidly and cheaply adjust the amount of money sent to a particular recipient without incurring the cost of a broadcast transaction.

For example, consider an untrusted internet access point, like a WiFi hotspot in a cafe you never visited before. You'd like to pay 0.001 BTC per 10 kilobytes of usage, without opening an account with the cafe. A zero-trust solution means it could be fully automatic, so you could just pre-allocate a budget on your phone's mobile wallet at the start of the month, and then your device automatically negotiates and pays for internet access on demand. The cafe also wants to allow anyone to pay them without the fear of being ripped off.

To do this, the following protocol can be used. This protocol relies upon a **different** behavior of nLockTime to the original design. Starting around 2013 time-locked transactions were made non standard and no longer enter the memory pool, thus cannot be broadcast before the timelock expires. When the behaviour of nLockTime is restored to the original design from Satoshi, a variant of this protocol is required which is discussed below.

We define the client to be the party sending value, and the server to be the party receiving it. This is written from the clients perspective.

1. Create a public key (K1). Request a public key from the server (K2).
2. Create and sign but do not broadcast a transaction (T1) that sets up a payment of (for example) 10 BTC to an output requiring both the server's private key and one of your own to be used. A good way to do this is use OP_CHECKMULTISIG. The value to be used is chosen as an efficiency tradeoff.
3. Create a refund transaction (T2) that is connected to the output of T1 which sends all the money back to yourself. It has a time lock set for some time in the future, for instance a few hours. Don't sign it, and provide the unsigned transaction to the server. By convention, the output script is "2 K1 K2 2 CHECKMULTISIG"
4. The server signs T2 using its private key associated with K2 and returns the signature to the client. Note that it has not seen T1 at this point, just the hash (which is in the unsigned T2).

5. The client verifies the servers signature is correct and aborts if not.
6. The client signs T1 and passes the signature to the server, which now broadcasts the transaction (either party can do this if they both have connectivity). This locks in the money.
7. The client then creates a new transaction, T3, which connects to T1 like the refund transaction does and has two outputs. One goes to K1 and the other goes to K2. It starts out with all value allocated to the first output (K1), ie, it does the same thing as the refund transaction but is not time locked. The client signs T3 and provides the transaction and signature to the server.
8. The server verifies the output to itself is of the expected size and verifies the client's provided signature is correct.
9. When the client wishes to pay the server, it adjusts its copy of T3 to allocate more value to the server's output and less to its own. It then re-signs the new T3 and sends the signature to the server. It does not have to send the whole transaction, just the signature and the amount to increment by is sufficient. The server adjusts its copy of T3 to match the new amounts, verifies the signature and continues.

This continues until the session ends, or the 1-day period is getting close to expiry. The AP then signs and broadcasts the last transaction it saw, allocating the final amount to itself. The refund transaction is needed to handle the case where the server disappears or halts at any point, leaving the allocated value in limbo. If this happens then once the time lock has expired the client can broadcast the refund transaction and get back all the money.

This protocol has been implemented in bitcoinj (https://code.google.com/p/bitcoinj/wiki/WorkingWithMicropayments).

The lock time and sequence numbers avoid an attack in which the AP provides connectivity, and then the user double-spends the output back to themselves using the first version of TX2, thus preventing the cafe from claiming the bill. If the user does try this, the TX won't be included right away, giving the access point a window of time in which it can observe the TX broadcast, and then broadcast the last version it saw, overriding the user's attempted double-spend.

The latter protocol that relies on transaction replacement is more flexible because it allows the value allocated to go down as well as up during the lifetime of the channel as long as the client receives signatures from the server, but for many use cases this functionality is not required. Replacement also allows for more complex configurations of channels that involve more than two parties. Elaboration on such use cases is a left as an exercise for the reader.

Example 8: Multi-party decentralised lotteries

Using some of the techniques from example 6 and some very advanced scripting, it becomes possible to build a multi-party lottery with no operator. The exact protocol used is explained in the paper "Secure multiparty computations on Bitcoin" (http://eprint.iacr.org/2013/784). A shorter explanation of how it works may be added to this wiki at some point in the future.

See Also

- Script

Retrieved from "https://en.bitcoin.it/w/index.php?title=Contract&oldid=59172"

Category: Technical