Development of a payment channel over the Bitcoin network

David Lozano Jarque, *Undergraduate student, UAB.cat*

Abstract—Bitcoin is a decentralized digital cryptocurrency that allows payments between users without the need of a central authority. Despite the potential of the technology, in the past years, the scaling debate has been the main focus of development as because of the internal details of implementation of the technology, the network can't process and store the highly increasing demand of transactions in the public ledger, also called the *blockchain*. A solution for this is reducing the need of transactions with off-chain payment channels, that can be able to process thousands of micropayment transactions between two nodes so that most transactions don't appear in the blockchain but if they did would be valid, using the Bitcoin scripting language and some game theory techniques. With payment channels, only the setup and closure transactions would appear in the blockchain and all the payment transactions would be temporary and stored just by the nodes of the channel, reducing the amount of transactions the network has to handle. Our work is designing and implementing a bidirectional payment channel by using the combination of two unidirectional payment channels.

Keywords—Cryptocurrency, Bitcoin, scaling, Payment channel, Bidirectional payment channel

1 INTRODUCTION

BITCOIN is a cryptocurrency that first appeared in a cryptography mailing list [1] with a post by an anonymous user who called himself "Satoshi Nakamoto" and defined in a whitepaper [2] a decentralized cryptocurrency that allowed direct peer to peer digital currency transactions without the need of a central authority in which users trust for validating those transactions. Instead, each peer can validate those transactions using cryptography (technically validating digital signatures) and after that generate a block of transactions including them, action whose reward is retrieving newly generated currency, aiming peers to secure the network.

All the transactions ever made, grouped in a structure called *block*, are stored forming a chain, in a distributed read-write-only database each (full) network node stores called the *blockchain*, as each block is chained to the previous creating a not-modifiable chain of blocks using hash functions that link each block to a previous one using its hash.

- *E-mail: uab@davidlj95.com*
- *Specialized in Information Technologies*
- *Tutored by Joan Herrera Joancomart (dEIC.UAB.cat)*
- *Course 2016-2017*

Manuscript written on June 2017, Engineering School (UAB.cat)

1.1 The *blockchain* limits

At a high level, this is how Bitcoin works. The problem comes with the public ledger or *blockchain* that stores absolutely all transactions ever performed: with an average block size of nearly 1MB [3] (as it's the hardcoded limit size for a block) that contains approximately 2.000 transactions [4] and with a block appearing every 10 minutes, this makes this distributed database grow approximately 50GB every year [5]. The block size limit is fixed at 1MB and difficulty for solving new blocks using the proof-of-work algorithm [6] is dynamically adjusted so that new blocks appear approximately every 10 minutes. This is fixed in the protocol and therefore the code of the software nodes run, so can't be changed without everyone agreeing or could lead to a blockchain split.

1.2 The scaling problem

With Bitcoin gaining popularity among more users, more transactions are created and needed to handle and get stored in the blockchain, but due to the limits set, not all transactions can't be handled and the number of delayed transactions until the network can handle them is increasing every day. There are several active proposals [7], [8] to change those limits and allow to handle more transactions, but meanwhile a solution gets activated and agreed by all the Bitcoin ecosystem (users, developers and miners), another long term solution is being proposed: reducing the number of transactions needed to perform payments.

1.3 Payment channels

This is where payment channels appear [9], allowing to two users or more that need a constant flow of transactions to pay each other for products or services perform those payments in an instant way without waiting for the confirmation of the transaction in the blockchain by exchanging transactions privately between them that don't appear in the blockchain, also called offchain transactions. Just the opening and closure transactions of the channel would be needed to appear in the blockchain, therefore reducing the amount of transactions they need to send to the blockchain and relieving the blockchain from transactions. The trick is that privately exchanged offchain transactions could be sent to the blockchain and they would be valid, but are kept private until the channel needs to be closed and they are released to the blockchain closing the channel. Each payment transaction replaces the old one so just the last one payment needs to be kept, allowing a high rate of transactions between nodes of the payment channel. The payment channel has to be secure by design and implemented with a secure protocol so that no party of the payment channel can't steal or lock the other party funds or act maliciously.

2 Bitcoin and Smart Contracts

As said before, Bitcoin allows to store a decentralized consensual database of transactions that transfer units of currency between users. To understand the how currency units are moved, we need to understand what a transaction is at a low level detail

2.1 Bitcoin transactions

At a low level, a transaction is just an array of bytes that specifies some inputs and some outputs, prefixed by a version field and suffixed with a field named *nLocktime* we'll talk about it later. What every transaction does is spend

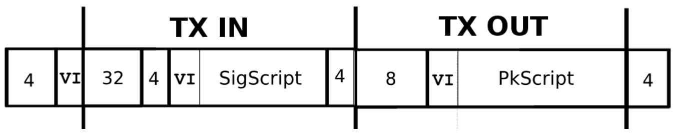

Fig. 1: Transaction binary format

a previous output by specifying an input that points to a previous output (indicated by the transaction id and number of output) and some data required to spend that output. To specify a new output to move the funds spent by the, the value of the output and the conditions to spend must be specified. This mentioned data to spend and conditions to spend is specified using a scripting language exclusive for Bitcoin technology [10]. But if all transactions have to spend a previous output, when are the first outputs generated? There's a special transaction with no inputs that is the one that generates currency, but it just valid once in a block, to spend the reward (that must mutch exactly the reward value) a node receives when creating a new block, otherwise the rest of nodes in the network won't accept it as valid.

2.2 Bitcoin scripting language

One of the powers of Bitcoin is its stack-based scripting language, as allows to specify how funds can be spent by creating a small script in the Bitcoin scripting language [10] described in the Bitcoin protocol. Therefore for a transaction to be valid, the input must refer a valid and non-spent output (also called UTXO: Unspent Transaction Output) and the input script (also called *scriptSig* followed by the refered output script (also called*scriptPubKey* must end succesfully with a non-empty stack. Also, the sum of outputs' values must be less than the sum of inputs' values[1]. This scripting language basically reads 1-byte opcodes that able to store (push into the stack) data, perform arithmetical operations over that, and some cryptographic operations like ECDSA signatures and hash functions.

The most used script is called P2PKH (pay-to-public-key-hash), that sets as the conditions for spending an output (the *scriptPubKey*) to specify a public key whose hash matches the specified one and a signature with that public key of the transaction spending the output. The input script (the *scriptSig*) therefore must contain a valid public key whose hash matches the specified in the output script and a valid signature performed with the private key paired to that public key. The hash of a public key needed in the mentioned output script to specify the spend conditions, prefixed by a version byte (that determines the network it is valid and that the type of data is a hash of a public key) suffixed with a SHA-256 4-byte checksum of that hash, all encoded in base58 is a Bitcoin address, commonly used to pay units of currency to another user who reveals their address to be paid.

But as said before, the Bitcoin scripting language allows us to code any script to specify the spend conditions and any script to specify the data to spend following those conditions. Here is when P2SH (pay-to-script-hash) comes. This method of payment allows us to create an smart contract by defining an script where we specify the conditions to spend the output (called the *redeemScript*) and create an output paying to this script hash. Therefore to spend that output, we must reveal the *reedemScript* and often specify also data that the script needs to be spent, like some signatures (multisig P2SH) or a hash preimage, or whatever we design[2].

3 Unidirectional Payment Channels

The first step, once understood how Bitcoin transactions work and how we can develop smart contracts with them, is to design unidirectional payment channels. This channels, also called simple micropayment channels were first defined by Mike Hearn and Jeremy Spilman [11] and basically define two users, one who pays (payer) incrementally some amounts to the other (payee).

3.1 The scheme

Every channel has three phases: **funding**, where the founder or channel payer put some units of currency they own into a smart contract (we use a P2SH to pay to a *redeemScript*), **payments** where the payer creates transactions that incrementally pay more to payee spending the funding output (so the payee just keeps the one that pays more to them, as just one of all the payment transactions is valid) and the **closure** where the channel is closed either because its expiry date is coming (and the payee signs and broadcasts the latest received transaction) or because a user acted maliciously. Therefore the scheme is to create a funding transaction paying to a smart contract that allows spending it using a multisig (so the payer creates transaction to pay to the payer that signs prior sending it to them) or after a certain time by the founder (as if the payee doesn't collaborate and doesn't perform the multisig funds could be locked forever). This can be achieved either by creating a smart funding transaction that includes the expiry condition or a multisig funding transaction and a refund transaction that is signed by both and allows to be spent after a certain time. We opted for a single smart transaction in order to simplify the process. Summarizing, the most important part is the funding smart

1. The difference between the sum of inputs and outputs if is greater than 0 is called the transaction **fee**, and will be rewarded along with the block reward to the node that includes the transaction in a block (in Bitcoin argot, *mines* the block)

2. Despite we could technically specify any output and input script so that if the input script followed by the output script return a valid result (succesfully executed with non-empty stack or False at the top of the stack), if we don't use either P2PKH or P2SH, our transaction would be non-standard and probably not accepted by the network nodes

contract, as must allow a refund after a certain time in case the payee does not collaborate so the payer can recover the funds and also to pay incremental amounts to the payee.

3.2 The smart contract

In order to create a transaction that spends some funds of the payer and the output pays to a smart contract that requires a multisig for being spent (and will be used to perform payments) or just a signature after certain time (to prevent from funds being locked if payee doesn't collaborate), our proposal[3] was to create a transaction funding this *redeemScript*:

```
OP_IF <time> OP_CHECKLOCKTIMEVERIFY
 OP_DROP <PubKeyPayer1> OP_CHECKSIG
    OP_ELSE OP_2 <PubKeyPayer2>
<PubKeyPayee> OP_2 OP_CHECKMULTISIG
             OP_ENDIF
```

Note that the payer both owns private key of `<PubKeyPayer(1/2)>` and the payee holds the private key of `<PubKeyPayee>`.

3.3 Channel operations

With this smart contract script, we could create and test after that all the transactions for the channel:

- **Funding**: A transaction spending a payer's P2PKH *scriptSig* and with a P2SH output paying to the previously mentioned redeem script hash
- **Payment**: A transaction signed by both parties (firstly signed by the payer and then sent to the payee missing its signature to be valid) spending the redeem script with the `OP_CHECKMULTISIG` statement specifying an `OP_FALSE` and whose outputs are P2PKH *scriptPubKeys* to the payed and to the payee as a return. Each payment transaction must pay more than the previous one to the payer, as the payer will always hold the one that pays more to them. In case of wanting the payer to receive less than the previous transaction, we need a bidirectional payment channel.
- **Refund**: A transaction signed by the payer, and with `nLocktime`[4] field set after the `<time>` field specified in the script, spending the transaction with just its signature as specifies to pay with the first block of the redeem script with an `OP_TRUE` and whose output is a P2PKH output to an address the founder owns.
- **Closure**: The same transaction as the payment can act as a closure if broadcasted to the network previously signed by the payee. It has to be sent by the payed user before the expiry time or the payer could use the refund transaction so all payment transactions would be invalid as those funds would be already spent by the refund transaction.

3.4 The protocol

All this transactions must be created following a secure protocol that ensures all users are secure creating and operating the channel without any of them trusting the other. The protocol to establish a unidirectional payment channel between Alice (the payer) and Bob (the payee) would be the following: Alice requests opening a channel and specifies the funds of the channel (maximum amount Alice can pay to Bob) and the expiry date of it. If Bob agrees on the channel creation, he sends its public key so that Alice can create the funding transaction. Once the funding transaction is created, Alice sends the transaction to Bob along with the redeem script, so he can trace it and verify the contract is correct. Bob sends an acknowledge to Alice if wants to proceed with the channel opening (a signed one with Bob's key so Alice can verify the acknowledge is real). Alice eventually can broadcast the funding transaction to the Bitcoin network. Once the transaction is confirmed, Alice can create payment transactions, sign them and send them to Bob privately. When the expiry date is getting over, Bob will close the channel by broadcasting the latest received

3. Along with Carlos Gonzalez Cebrecos

4. We require the use of the `nLocktime` transaction field in order to make the script `OP_CHECKLOCKTIMEVERIFY` work as specified in BIP-65 [12]

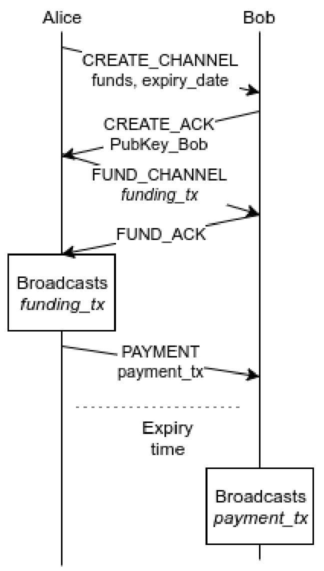

Fig. 2: Unidirectional payment channel protocol

payment transaction to the Bitcoin network (after signing it with its key).

4 Bidirectional payment channels

The problem with above channels is that just the payer can pay incrementally amounts of currency unit to the payee. What if we want the channel to be duplex so that both parties can send amounts of currency in both ways? In this work we researched following the solution proposed by Christian Decker and Roger Wattenhofer [13] that is to basically to create a duplex payment channel by using two unidirectional payment channels linked together, one in each direction. Another popular solution proposed is the Lightning Network, that uses a more complex structure to build a duplex payment channel [14]

4.1 The scheme

As said previously, the idea is to use two unidirectional payment channels, one in each direction, so that we can pay in both directions. To do that, in the funding transaction, there must be two inputs and two outputs. One input and one output per user. The Alice input spent value minus fees will be the first output value, where the output will pay to the same redeem script as the unidirectional channel. This will be the channel used by Alice to pay to Bob. The second input and output will be constructed using the same scheme for Bob to pay Alice. The rest of the payment channel would work the same way that in a unidirectional channel, where each transaction spends an output or another depending if Alice is paying to Bob or viceversa.

4.2 The protocol

In order to create the duplex payment channel, the following protocol must be followed in order for the channel to be secure: We can

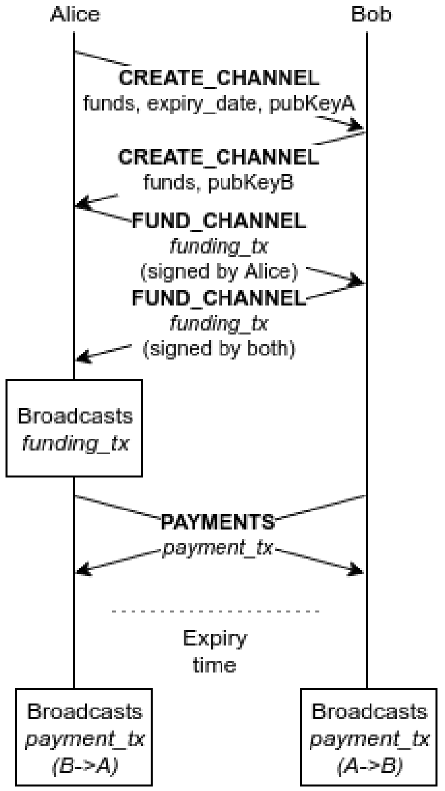

Fig. 3: Bidirectional payment channel protocol

see the protocol is similar to the unidirectional payment channel. In this example, Alice starts the request for the creation of the payment channel, but Bob could also send the request, inverting the communications until the payments section. The basic protocol consists in Alice sending the request to Bob for the channel creation, as happened with the unidirectional channel with the funds Alice desires to pay Bob, but also including his public key so that

Bob can verify the funding transaction created. If Bob agrees with the channel creation, replies with his public key to create the output for Alice paying to Bob and the funds that Bob wants to use to fund his channel to pay Alice. Once Alice has all data, can create the funding transaction with the two outputs and his input, and sign her input (indicating `SIGHASH_ALL` meaning that signs the transaction containing the two outputs). Alice sends the partially completed transaction (along with the redeem scripts) to Bob. Bob checks the transaction is correct and adds his input signed, returning the fully signed transaction to Alice as a final acknowledge for creating the channel. Once Alice receives the transaction, checks that is valid and broadcasts the transaction to the network. Now payments spending the Alice output to pay to Bob and the Bob output to pay to Alice can be performed creating offchain transactions. To close the channel, both Bob and Alice have to release the latest received payment transactions before the channel expiry to close the channel.

4.3 Channel operations

The same operations applied for the unidirectional payment channel would be valid (despite the transaction for funding being slightly different with an added input and output for the second way channel).

4.4 Channel reset

One thing that can happen is that either Alice or Bob spends all the funds they owned paying to the other user. In that case, the channel needs to be reset, so that the received funds from the other party can be used to continue paying to them. To do this, a solution is also described by C. Decker and R. Wattenhofer [13] and is called the invalidation tree using what it's called atomic multiparty opt-in transactions.

4.4.1 Atomic multiparty opt-in

This kind of meta-transactions are a model for creating transactions to fund smart contracts (one or more outputs) that instead of being funded by one or more inputs with a P2PKH *scriptSig* owned by a user, they claim a multisig *P2SH* output that has not been signed yet. This allows to first design the smart contract and once all parties agree, they sign a transaction spending one or more P2PKH to fund the multisig output claimed by the opt-in transaction and now both transactions have funded the smart contract in a secure way no matter the order of signatures.

In the case of channel resetting, this transactions are not necessary for a simple duplex payment channel, but can be used if we wish the channel to be reseted, as creating another smart contract with different conditions (like specifying different amounts) spending the opt-in transaction but with a lower locktime than the previous smart contract transaction would make the new transaction the current as the old one would have a larger locktime and therefore the current one can be spent before. This just works when specifying transactions with a lower locktime to replace the previous ones, so that renewing the expiry time couldn't be done with this kind of transactions. We can also chain opt-in transactions forming what is called an invalidation tree, where the invalidation is performed by specifying lower timelocks on each new transactions to invalidate previous ones.
`(-- OPTIONAL PROJECT FEATURE --)`

4.5 Multihop payment channels

Using HTLC (Hash time locked smart contracts) we can create channels that are able to route payments from A to C by using two existent channels, from A to B and B to C.
`(-- OPTIONAL PROJECT FEATURE --)`

5 THE IMPLEMENTATION

In order to implement the bidirectional payment channel, a research was performed to check what Python libraries where available to develop smart contracts and therefore transactions with smart contracts. What we found is that no object-oriented and well documented library was available to create non-typical transactions (P2SH with a custom `redeemScript`).

Because of that, we implemented a new library / framework to create easily customized transactions using the Bitcoin protocol information and it's implementation details [15], [16].

5.1 Our Bitcoin framework

To implement our framework, we decided to create a series of modules and classes oriented towards to the puzzle-friendlyness property: all objects / classes must be able to be serialized / deserialize into / from an array of bytes compatible with the Bitcoin protocol. We just implemented to save time, but, the strictly necessary modules and classes needed for this project development, but setting the base for developing a well designed, usable and easy to understand Bitcoin Python library that aims new developers to create smart contracts in the Bitcoin network.

5.2 Developing progress

The framework started with the ability to create an empty valid transaction and after that implementing all the fields necessary, composing each field of another subfields to allow the mentioned puzzle-friendlyness. The latest developed part of the framework was part of the Bitcoin scripting language (that is in constant development) to implement the needed opcodes to create the smart contracts.

Once the framework allowed to create valid transactions (that required special focus on cryptography functions and its serialization), we tested basic P2PKH transactions created with the framework and a P2SH multisig transaction. After that, the OP_CHECKLOCKTIME verify was implemented and tested and a unidirectional payment channel was created.

After all development and testing finished for the creation of valid and functional unidirectional payment channels, I started developing the Bidirectional Payment Channel as specified in the previous chapter of this document.

5.3 The duplex channel implementation

`(-- TODO --)`

6 Conclusion

```
(-- TODO (Main point: switch
to Ethereum, also flippening is
happening :) --)
```

Acknowledgments

This work couldn't have been possible without the collaboration with Carlos Gonzalez Cebrecos, another student performing a similar project that coworked developing our Bitcoin framework and solving some Bitcoin and related technologies doubts along with our tutors Jordi Herrera Joancomarti, Sergi Delgado and Cristina Perez who introduced us in the Bitcoin world.

References

[1] S. Nakamoto, "Bitcoin p2p e-cash paper." http://www.mail-archive.com/cryptography@metzdowd.com/msg09959.html, November 2008. (Accessed on 06/09/2017).

[2] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system." https://bitcoin.org/bitcoin.pdf, November 2008. (Accessed on 06/10/2017).

[3] B. L. S.A.R.L., "Average block size - blockchain." https://blockchain.info/es/charts/avg-block-size. (Accessed on 06/10/2017).

[4] B. L. S.A.R.L., "Average number of transactions per block - blockchain." https://blockchain.info/es/charts/n-transactions-per-block. (Accessed on 06/10/2017).

[5] B. L. S.A.R.L., "Blockchain size - blockchain." https://blockchain.info/charts/blocks-size. (Accessed on 06/10/2017).

[6] "Proof of work - bitcoin wiki." https://en.bitcoin.it/wiki/Proof_of_work, November 2011. (Accessed on 06/10/2017).

[7] "Segwit resources." https://segwit.org/. (Accessed on 06/10/2017).

[8] "Bitcoin unlimited." https://www.bitcoinunlimited.info/. (Accessed on 06/10/2017).

[9] "Payment channels - bitcoin wiki." https://en.bitcoin.it/wiki/Payment_channels. (Accessed on 06/10/2017).

[10] "Script - bitcoin wiki." https://en.bitcoin.it/wiki/Script. (Accessed on 06/10/2017).

[11] M. Hearn and J. Spilman, "Contract - bitcoin wiki." https://en.bitcoin.it/wiki/Contract. (Accessed on 06/11/2017).

[12] P. Todd, "bips/bip-0065.mediawiki at master bitcoin/bips." https://github.com/bitcoin/bips/blob/master/bip-0065.mediawiki, October 2014. (Accessed on 06/11/2017).

[13] C. Decker and R. Wattenhofer, "A fast and scalable payment network with bitcoin duplex micropayment channels," in *Symposium on Self-Stabilizing Systems*, pp. 3–18, Springer, 2015.

[14] J. Poon and T. Dryja, "The bitcoin lightning network: Scalable off-chain instant payments," 2015.

[15] "Developer guide - bitcoin." https://bitcoin.org/en/developer-guide. (Accessed on 06/12/2017).

[16] "Protocol documentation - bitcoin wiki." https://en.bitcoin.it/wiki/Protocol_documentation. (Accessed on 06/12/2017).