Universitat Autònoma de Barcelona

Escola Tècnica Superior d'Enginyeria

Grau en Enginyeria Informàtica

Final degree project

Bitcoin Payment Channels

Progress report II

Author:
David Lozano Jarque
NIU 1359958
uab@davidlj95.com

Tutor:
Jordi Herrera Joancomartí

Version 0.1
June 12, 2017

Contents

A few notes about this document

Because this stage of the project was partially developed along Carlos GC[1], another student that is carrying out a very similar project, the document uses the first person pronouns in a plural form[2]. Despite that, and knowing that the final degree project has to be developed individually, all of the progress achieved has been done in an individual way, so the both of us have the exact same knowledge about the Bitcoin environment and have researched the exact same subjects until the writing of the present document and the collaboration is just to provide each other mutual help to understand the complex details that dealing with Bitcoin implies.

Also, and due to the complexity of the Bitcoin concept in depth, this document assumes some basic understanding of the Bitcoin cryptocurrency (despite it can be read by anyone and transmit the basic idea) and will skip several explanations about the some low-level details learnt during the project time lapse as the document is intended to provide a global view of the project development status rather than providing specific details of the tasks performed that include complex or difficult-to-explain Bitcoin special characteristics.

[1]Carlos González Cebrecos <carlos.gonzalezce@e-campus.uab.cat> `https://www.ccebrecos.com`

[2]The unique author of the document, notwithstanding the text is written in first person plural form, is exclusively the author that appears on this document's cover and not anyone else.

Introduction

In the present document, the second progress report of the *Bitcoin Payment Channels* project, we'll review the progress and milestones achieved during this stage of the project, to check if the project has been following the schedule, get to some conclusions and reschedule the project if necessary to change the project goals or ways to achieve them.

1.1 Project current status summary

1.1.1 Summary of the project status in the last progress report

In the last progress report, after studying and understanding the underlying concepts behind the Bitcoin technology and the details of how each of them are implemented at a low-level, we developed a Python framework to work with so we could easily create complex transactions by developing puzzle-friendly serializable objects. Despite investing time in the creation of that framework, we thought that a new library, puzzle-friendly oriented, and written for a high level of abstraction language like Python, was needed to develop things faster and would save us time rather than using the current libraries available (although we used them in our framework for critical and hard-to-code features like signatures to don't reinvent the wheel[3]).

With that framework, we tested all of our components to see if their functionality (basically serialization) matched the Bitcoin protocol using some existing Python libraries[1,2] and of course, the Bitcoin Core implementation[3], that represents the implementation used by 5989 nodes (86.49% of the total network). The latter test we performed was a P2PKH transaction fully generated by our framework[4], that was accepted by the network and mined in block 1095557 of the testnet Bitcoin blockchain.

[3]As a cryptography enthusiast, I personally tried to understand and reimplement ECDSA signatures source code from Vitalik Buterin's library[1] in my personal free time, and the reimplementation was eventually used after several tests. However, the only real improvement was a better DER ECDSA signatures codification and the knowledge adquired: a better understanding of the algorithm itself

[4]`https://tbtc.blockr.io/tx/info/258fb211724412d6ec6a531973c58233143e6ab355623658adc3164a5c70bd5b`

After that, we searched for information about Bitcoin unidirectional payment channels and designed a script to open / fund, an unidirectional payment channel, but its implementation and test were not yet performed.

1.1.2 Current project status

We found some mistakes in our script to open the Unidirectional Payment Channel, so we modified it until we got a potentially functional script to create the payment channel. Once we designed that script, we found that we had to add some functionality to our framework: be able to create and spend P2SH transactions, for example with scripts that allow multisignature transactions using `OP_CHECKMULTISIG` and also the use of the `OP_CHECKLOCKTIMEVERIFY` as described in BIP-65[4].

Problems arrived when trying to spend a P2SH multisignature script transaction once it was funded, because of how the signature was performed (we thought in signatures, the input script is replaced always with the scriptPubKey of the UTXO we are spending, but in case of P2SH, it has to be replaced with the redeemScript, and not the scriptPubKey, that contains it hash along the script standard opcodes). We had to read carefully an article explaining multisig[?] to find the issue, as the Bitcoin core error and message displayed explaining why our transaction wasn't sent to the network did not clarify anything: just said the signature was wrong.

More problems arrived when dealing with `OP_CHECKLOCKTIMEVERIFY`, another important part of our opening / funding transaction script of the Unidirectional Payment Channel. We had to search how the opcode acted, the serialization format of the number to indicate the time we are locking to, and how was the comparison of that lock time was performed (it required to modify the spending transaction locktime field, as 0, the default value was not valid if you use the lock-time feature). To solve that, we had to ask for support to our project teachers, discuss the BIP65[4] implementation, and setup an IDE[5] with the Bitcoin Core[3] C++ code to see exactly how was implemented in the latest version until we succesfully funded / opend our channel[5], performed payments, and closed it.

After solved this issue, the project development continued separately, as the Bidirectional Payment Channel development stage started and each of us will implement it individually after studying the white papers describing each of them. I studied the multinode Bidirectional Payment Channel proposed by Christian Decker and Roger Wattenhofer[6] and in the time of writing this report, I'm in the stage of development of the opening script for the channel. This means, a 2-weeks delay of the project schedule, mainly due to the extra research performed to create P2SH spending transactions and using `OP_CHECKLOCKTIMEVERIFY` and extra delay due to personal timing issues.

[5] `https://tbtc.blockr.io/tx/info/f730e8b212b1f6e48e6c99c0071dee11353dffae50192a1a3ada6aef13b7c818`

Unidirectional payment channels

2.1 The funding script

In the previous progress report, we designed the following redeem script to create the funding transaction of an unidirectional payment channel[6]:

```
OP_0 <sigA> <sigB> | OP_2 <pubKeyA> <pubKeyB> OP_2 OP_CHECKMULTISIG
OP_NOTIF <time> OP_CHECKLOCKTIMEVERIFY OP_DROP <PubKeyFounder> OP_CHECKSIG
OP_ENDIF
```

Note that the character `"|"` separes the data needed to pay the redeem script from the redeem script itself. Therefore the first part will be the data (along with the redeem script itself) to be used in the scriptSig that spends the script and the hash of the second part the hash to put in the P2SH output to fund the channel.

What we wanted to achieve was that, after a transaction was sent paying to this redeem script (technically speaking paying to its hash, with a P2SH scriptPubKey), the payments of the channel could be performed by using a multisig, until the specified time (the expiry date of the channel) where the founder could refund his funds in case of no cooperation of the second party. Therefore, the founder owns both `<pubKeyA>` and `<pubKeyFounder>` private keys.

Summarizing, there are two ways to spend this script:

- **Using the payment channel:** The founder, when want to send funds to the second party, creates a transaction spending this script before the expiry time arrives and sends it to the second party (with founder's signature included) sending some amount to the second party and some amount to themself as return. If the second party wants to close the channel, signs to complete the multisig and sends the last transaction to the blockchain (because as more transactions are given by the founder, the last is

[6]Because of a typo error, `OP_CHECKMULTISIG and OP_ENDIF` did not appear in the 0.1 version of the report

supposed to give more amount than the previous ones, so the previous ones are discarded). In this case, the signatures wil be correct so `OP_CHECKMULTISIG` will return `OP_TRUE`, the `OP_NOTIF` statements block including the `OP_CHECKLOCKTIMEVERIFY` won't be executed and the transactions will be valid.

- **Waiting for expiry time:** If the second party does not cooperate, there has to be a way for the founder to get back his funds if the payment channel is not used before the expiry time. Because of that, it exists the `OP_CHECKLOCKTIME` verify code. It will allow to spend the funds after the time specified has been reached, just needing one signature, the founder's signature. But to do that, we have to reach `OP_NOTIF` and the only way is to make `OP_CHECKMULTISIG` fail, for example, using in `<sigA>,<sigB>` the same signature, created with private key matching `<pubKeyA>`, that the founder owns. And previous to that `OP_0`, put the signature created with the private key matching `<pubKeyFounder>` public key to spend it. This is not optimal, as to select payment method, we have to make `OP_CHECKMULTISIG` fail and append data (two bad signatures) that occupies unnecessary space. Despite that, the transaction would be spendable after a certain period of time.

Therefore, and after consulting with our teachers, we got a valid script to operate a unidirectional channel, despite it could be optimized by creating a non-standard transaction.

2.1.1 Implementing the script

To implement the script using our framework, we had to implement new opcodes (`OP_0, OP_2, OP_CHECKMULTISIG, OP_NOTIF, OP_ENDIF`, compose them all together and create a transaction to fund the script and then spend that script.

First problem: P2SH signatures

Creating a P2PKH transaction spending an UTXO with a P2PKH scriptPubKey with a P2PKH scriptSig with just an output to pay to a P2SH scriptPubKey with the hash of the redeem script implemented was not a problem, as we had all the tools needed and the transaction was accepted without any problems.

The problems came when wanting to spend that P2SH UTXO created, as the signature did not match according to the errors the Bitcoin Core implementation told us when trying to send the raw transaction. We checked again all the methods that created signatures, but did not know where the source of the problem was, as we succesfully had tested those methods creating P2PKH transactions. It was not until we reviewed an article describing how a simple P2SH worked (just a multisig P2SH)[7] and reverse engineered the Bitcoin Core code to check where the error was triggered[3] that we found the mistake: **when**

signing a P2SH UTXO, and creating the pseudo-transaction to be signed, the input script being signed has to be replaced by the redeem script, instead of the scriptPubkey when dealing with P2PKH outputs. AFter that, we created a transaction to fund and spend a simple multisig redeem script and continued with our development.

Second problem: `OP_CHECKLOCKTIMEVERIFY` operation

The second problem we found is dealing with `OP_CHECKLOCKTIMEVERIFY`. To deal with that opcode, we read the document where it was proposed[4] (as it didn't exist before in the original code, it was an improvement that reimplemented a no operation opcode). Reading the document, we found that previously to understand this opcode, we had to fully understand the `nLocktime` field meaning in a transaction (until now, we used `0` in that field as was the default, recommended value).

After reading several questions on a QA site about Bitcoin[8,9], we understood the meaning of the field. When specified in a transaction, the field makes the transaction **invalid**[7]until the lock time (can be specified using either an absolute block number or a UNIX timestamp) is reached.

We thought we had understood everything until we started testing with some scripts that appear as an example in the opcode definition document[4]. At first, the document does not clearly specify how the `<time>` field must be indicated (little-endian, big-endian, size...), rather than referencing to a class named `CScriptNum` in the Bitcoin Core implementation and saying it can take up to 5 bytes and use the minimum bytes possible.

After consulting one of our teachers Sergi, who refers us to a question in the same QA site[10] and looking in the Bitcoin Core implementation C++ code[8], we couldn't find that it was a little-endian encoded number that takes the minimum bytes possible.

But once specifying the time in the correct format, we couldn't still fund and spend the script: we didn't know how the check against the time was performed. After reviewing the Bitcoin Core implemenation of the opcode, [9] we didn't understand how the check was performed. The comparison of the time specified in the script is against the field `nLocktime` of the transaction, so if we don't specify a higher `nLocktime` in the transaction than the time in the script, the operation `OP_CHECKLOCKTIMEVERIFY` will fail. For this reason, the

[7]It's important to know that when speaking in Bitcoin terms, an **invalid transaction** means the transaction won't be accepted by the network (won't be accepted in the mempool either and therefore neither mined) and an **unspendable transaction** will be accepted and can be mined, but won't be spendable. The invalidity or unspendability can be tepmoral, as in case of the locktime transactions, or permanent

[8]`https://github.com/bitcoin/bitcoin/blob/master/src/script/script.h#L358-L360`

[9]`https://github.com/bitcoin/bitcoin/blob/master/src/script/interpreter.cpp#L1272-L1306`

`nLocktime` field and the time in the script must be in the same format (or both specify UNIX timestamps, or both specify an absolute block number).

After that, we could spend a P2SH transaction that uses a `OP_CHECKMULTISIG` and `OP_CHECKLOCKTIMEVERIFY` opcode[10]

Redesigning the script:

After digging into the Bitcoin Core implementation[3], and reading carefully the BIP-65 document[4], we designed a model for time locked scripts, that contain a script that can be spent just after a certain time and a script that can always be used to spend it. We called that in our framework a `TimeLockedScript`, whose model is the next:

```
OP_IF <time> OP_CHECKLOCKTIMEVERIFY OP_DROP <timelocked_script> OP_ELSE
              <unlocked_script> OP_ENDIF <lifetime_script>
```

What this scripts allows us is:

- **Specify how are we spending it**: When we are spending, we can select if we're paying to the script using the locktime condition (therefore, the locktime has been reached), or if we're paying to the script before the locktime arrives, just by specifying `OP_0` or `OP_1` in the data to pay the script. If you specify `OP_1`, you'll enter in the locktime condition and with `OP_0` you'll be able to pay before the locktime.3
- **Specify a script for each condition**: We can specify a script to be used to pay after the locktime, and the script to pay before the locktime arrives (or at anytime). This way with this model more scripts can be designed rather than an opening unidirectional channel script (escrow script for instance).

2.1.2 The unidirectional channel implementation

Finally our redeem script for the channel was:

```
OP_IF <time> OP_CHECKLOCKTIMEVERIFY OP_DROP <PubKeyFounder> OP_CHECKSIG
    OP_ELSE OP_2 <PubKeyA> <PubKeyB> OP_2 OP_CHECKMULTISIG OP_ENDIF
```

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

- **Funding**: A transaction spending a P2PKH scriptSig and with a P2SH output paying to the previously mentioned redeem script hash

[10]https://tbtc.blockr.io/tx/info/8dc10f058a0c6ee6ba481cfdb8cd350a5b406f76a024cdcbd96a87931372cb46

- **Payment**: A transaction signed by both parties (firstly signed by the founder) 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 founder as a return.
- **Refund**: A transaction signed by the founder, and with `nLocktime` 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 sent to the network. It has to be sent by the payed user before the expiry time or the founder could use the refund transaction so payment transactions would be invalid.

Bidirectional payment channels

3.1 The whitepaper

After studying the whitepaper[6] that describes a bidirectional payment channel that we'll try to implement, I decided to go simple and start by implementing a single node bidirectional payment channel implementation. The whitepaper goes further and implements a multi-node bidirectional payment channels, but depending on how the project develops, we may not be able to implement it before the project deadline arrives.

At the moment of writing this document, I've reviewed the paper and started implementing the funding script, but without having tested the script on the network yet.

Results and conclusions

4.1 Project status and schedule

After reviewing the Gantt's diagram of the previous progress report, we can see that according to the the current status (testing the funding transaction of a bidirectional payment channel), we are two weeks on delay compared with the estimated schedule proposed on the previous document.

The problems that surged in order to reach that delay were:

- **More research time than expected:** We thought that with our knowledge acquired until the last progress report we would spend less time in each iteration performing research tasks and more time doing testing would be required. What happened is that the more advanced or non-standard scripts you are developing, the less information you find. It was true that we also required more test time, and less development as we developed a great and usable framework, but the research task time couldn't be reduced because of the lack of sources. To avoid long searches, I decided to use the Bitcoin Core implementation code as the main source of information as it's the code that will be executed to check our transactions and despite being hard to understand because of the advanced C++ syntax used, the efforts of understanding the code are paid as the knowledge obtained will be valid without any doubt.
- **Personal timing issues:** Some weeks we couldn't complete the iterations because I started an scholarship that takes more time than expected and therefore each iteration was delayed more than necessary.

To reschedule the project, we skipped implementing the two transactions (refund and fund) method to open a channel, as the one-transaction works and is more secure and I will use the Bitcoin Core implementation as my main source of information to ensure the validity and accuracy of the knowledge obtained. With this changes, I expect to be on-schedule after testing the bidirectional payment channel funding transaction.

4.2 About the development

Since last progress report, I've learned how P2SH transactions are created and spent, what a BIP exactly is, how it is created, developed, tested and requested to be introduced as a feature in the network, and how the BIP-65 works implying that I understand the `nLocktime` exact meaning and operation, and also how `nSequence` works because it also was a requirement for BIP-65.

After reading the whitepaper, I think no extra knowledge should be necessary to develop the payment channel rather than creating and testing the transaction with its scripts, but it's very optimistic to say it. Despite that, and with the changes performed to the research tasks (look first the Bitcoin Core implementation, as I'm know more familiarized with it), I think I'll be able to develop a bidirectional payment channel as on schedule, despite maybe I'll have to skip optional features like the channel automation so the channel can be used by any user with a minimal knowledge about Bitcoin.

Bibliography

[1] "vbuterin/pybitcointools: Simple, common-sense bitcoin-themed python ecc library." `https://github.com/vbuterin/pybitcointools`. (Accessed on 05/16/2017).

[2] "petertodd/python-bitcoinlib: Python2/3 library providing an easy interface to the bitcoin data structures and protocol.." `https://github.com/petertodd/python-bitcoinlib`. (Accessed on 05/16/2017).

[3] "bitcoin/bitcoin: Bitcoin core integration/staging tree." `https://github.com/bitcoin/bitcoin`. (Accessed on 05/16/2017).

[4] "bips/bip-0065.mediawiki at master · bitcoin/bips." `https://github.com/bitcoin/bips/blob/master/bip-0065.mediawiki`, October 2014. (Accessed on 05/17/2017).

[5] "How to set up an ide for developing bitcoin core on linux." `https://medium.com/@lopp/how-to-set-up-an-ide-for-developing-bitcoin-core-on-linux-193bd313acb1`. (Accessed on 05/17/2017).

[6] 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.

[7] "Bitcoin multisig the hard way: Understanding raw p2sh multisig transactions." `http://www.soroushjp.com/2014/12/20/bitcoin-multisig-the-hard-way-understanding-raw-multisignature-bitcoin-transactio` December 2014. (Accessed on 05/17/2017).

[8] "locktime - nlocktime transactions, how do they persist? are they broadcast before they are valid? - bitcoin stack exchange." `https://bitcoin.stackexchange.com/questions/26937/nlocktime-transactions-how-do-they-persist-are-they-broadcast-before-they-are`, June 2014. (Accessed on 05/18/2017).

[9] "sequence - is my understanding of locktime correct? - bitcoin stack exchange." `https://bitcoin.stackexchange.com/questions/40764/`

is-my-understanding-of-locktime-correct, October 2015. (Accessed on 05/18/2017).

[10] "How is time encoded (bip65) in scripts? - bitcoin stack exchange." https://bitcoin.stackexchange.com/questions/39119/how-is-time-encoded-bip65-in-scripts, August 2015. (Accessed on 05/18/2017).