EE 374: Blockchain Foundations **Stanford, Spring 2022**

Lecture 11: Light Clients

May 6, 2022

Lecturer: Dr. Dionysis Zindros Scribes: Michael Nath, Coleman Smith

1 Light Clients

1.1 Motivation

Our current model has three main scalability limitations: storage requirements (full nodes store the entire chain, $\sim$ 1TB in Bitcoin), communication requirements (full nodes broadcast, request and download every transaction and block submitted to the network), and computation requirements (full nodes validate all incoming blocks and transactions, compute the UTXO set after each block, etc.).

We seek to design a light client (also referred to as a light node) such that it needs less storage, communicates less with the network, and requires less computation power. Our design will need only $\sim$ 100MB of storage, will only download transactions pertinent to our own address, and will not validate all transactions in the transaction graph.

Before designing such a node, we also wish to distinguish between full nodes and miners: full nodes are peers running our protocol (validating blocks and transactions, gossiping new objects, adopting longest chain, etc.), where miners perform all these tasks in addition to querying for new blocks.

1.2 Definition

To guide our discourse, we will define light clients as nodes on the blockchain network which are able to verify payments to and create transactions from a specified address, but without downloading the entire blockchain (specifically the transaction graph), and without validating said transaction graph in its entirety. Further, we will assume that light clients connect only with full node peers, rather than with other light clients, to ensure the availability of information which is available to any other full node in the network.

1.3 Header Chains

To solve the storage problem, we introduce block headers. We want our light clients to be able to interact with the network without downloading the entire blockchain, and block headers provide a way of maintaining chain virtues, while only needing to download a subset of the transaction graph.

Instead of a chain of blocks, each containing a vector of transactions, header chains are chains of block headers. Block headers are of the format: $s||x||\text{ctr}$, where s refers to the hash of the previous block header, x refers to the root of the merkle tree of all transactions in $\bar{x}$ (the transactions contained in the block), and ctr is the nonce discovered to satisfy the PoW equation.

While this solves our storage problem, it is not immediately clear how a header chain can be used to verify transactions (as the header contains no actual transaction hashes). However, as the block header contains x, light nodes may request merkle proofs π from their full node peers for the inclusion of pertinent transactions in the chain, which no PPT adversary can create for a transaction which was not included. Therefore, header chains provide the same guarantees with regards verification of transaction inclusion.

1.3.1 Block Validation

In order to verify a block in a block header chain, we modify the process slightly such that a full node must now

1. download the new block header
2. validate the header PoW and ancestry
3. download the block body
4. check the block body validity—transaction set validity and merkle tree validity.

1.3.2 Benefits

Header chains provide mitigate all three of our limitations listed above.

Unlike blocks, whose size depends on the number of transactions it contains, block headers have an lower-bounded size of 3κ, each κ coming from each hash being concatenated to make the block header. This enables the block header to be significantly more compact than a block, and solves the problem of storage for light clients which don't wish to download the entire transaction history. We say that the size is lower bounded because implementations may choose to include additional metadata in the block header. Thus, the minimum storage requirements for a light client is reduced from $O(h|\bar{x}|)$ to $O(h\kappa)$, where h is the height of the longest chain.

Since each block header is much smaller than an entire block, there are also computational benefits for both light and full nodes, as well as the network as a whole. This is because the computational complexity of calculating $H(B)$ is $O(|B|)$. As block headers are much smaller than blocks, this provides significant computational benefits in verifying PoW for both light and full nodes. This also allows light nodes to verify PoW without verifying the validity of the transactions in $\bar{x}$, which is left to the full nodes to compute. Finally, as $H(B)$ takes $O(3\kappa)$ rather than $O(|\bar{x}|)$ computational power to compute, full nodes are no longer incentivized to mine smaller blocks, which would have previously increased their hash rate.

In addition to these benefits, as light nodes only need to request the block header chain and a subset of all transactions, rather than the entire block chain and transaction graph, header chains allow light nodes to make significantly fewer requests to their full node peers, and for those requests to result in much smaller and shorter responses.

1.4 Making Payments

A core function of the light client is interacting with the block chain to make and receive payments concerning only itself. Thus, light clients interact with a sub-graph of the network and rely on full

nodes to receive the UTXOs belonging to it. A light client can then use these UTXOs to pay other addresses, sign such a transaction, and broadcast it to the network for validation and inclusion in the chain.

To build this subgraph of UTXOs, light clients must be able to make requests to full node peers for all UTXOs in the transaction graph that are owned by a given address (in this case, the address-of-interest of the light client itself). Furthermore, they must request a merkle proof from these peers for the inclusion of these transactions in the chain, which we know cannot be counterfeited by a PPT adversary. This reliance on full node peers to provide the transaction sub-graph poses no security risks under the non-eclipsing assumption, as an honest node will provide all transactions-of-interest to a requesting peer.

1.5 Block Header Validation

Light clients do not validate blocks. Instead, they validate only the block headers, and follow only valid header chains, without ever downloading the block itself. As a result, for a light client to verify and validate an incoming block header, the validation steps are a further modification (compared to the full node) of the usual process. Specifically, light clients must do the following.

1. download the new block header
2. validate the header PoW and ancestry
3. request the subset of transactions relating to their address (in case any new pertinent transactions are in this new block)
4. request proofs of inclusion for any of these relevant transactions which are in the new block (older transactions should already have been verified).

1.6 Transaction Validation

While light nodes can download header chains and verify each header (PoW and ancestry checks), they cannot validate transactions constituting the block corresponding to that block header. This is because light nodes are unaware of the entire transaction graph, and more specifically the current UTXO set. Without the UTXO set, a light node is unable to determine whether a transaction's inputs are valid. Furthermore, light nodes do not download transactions which are not pertinent to them, which means they are also unaware of the validity of said transactions. We will now see how an adversary might exploit this fact.

1.6.1 Local Chain Security

Common Prefix is still guaranteed for honest-majority networks (specifically, networks where a majority of the mining power is honest), as the PoW requirements on header chains ensure that each header in the chain is a successful query. As a result, finding a valid block header is just as difficult as finding a valid block, so no minority adversary can outpace the honest nodes in the network. However, there are some security risks for the light client in the event of a dishonest majority.

While no PPT adversary can create a proof π that some transaction is included in the chain when it is not, a dishonest *majority* might create an invalid chain (containing double spends,

invalid coinbase transactions, etc.) which grows longer than the longest honest chain. While no full node will accept the adversary's invalid chain, light clients validate only PoW and ancestry of the block header, and therefore would accept such an invalid chain. Preventing such an adversarial majority attack would require verifying the entire transaction graph. However, since light nodes do not contribute blocks to the chain, there are no security concerns for the network at large in this situation, and any invalid transactions submitted by the compromised light client would not be validated nor included in the competing honest chain. So long as the adversary's majority is transient, the light client's risks are also transient.

1.6.2 Privacy

As a light clients requests transactions relating only to its own address, it must reveal its public key to the full node(s) from whom it is receiving the transactions. This is a privacy compromise made for the sake of efficiency, although there are some ways to mitigate this risk (e.g. bloom filters, as used in the Bitcoin network).

1.6.3 Full Node Ramifications

On the topic of light clients making requests to their full node peers, honest full nodes must be able to provide all transactions which relate to the requested address. Therefore, they must store some type of mapping from public key addresses to sets of pertinent transactions, adding some complexity to the full nodes' protocol implementation.

1.7 Light Miners and the Quick Bootstrap Protocol

We will now explore whether light nodes can be miners in the network, and what modifications to our protocol are necessary for enabling this.

1.7.1 Mining as a Light Client

A naive approach to light client mining would be to bootstrap as normal, and then accept transactions into a mempool before including them in a block and broadcasting that block to the network.

However, mining as a light client is complicated by light clients' inability to validate transactions. The necessary information for validating transactions is the transaction graph of the entire blockchain, or more specifically the UTXO set after execution of the most recent block. Normally, the UTXO set is computed in the process of validating the full transaction graph of the blockchain. But, to allow for light clients (or nodes which do not download the full transaction history of the chain) to mine, we modify our block header structure to be $s||x||ctr||st$. We define st as the merkle root of the merkle tree made of all UTXOs in the UTXO set after executing the block, which we will call $\bar{st}$. That is, st is the merkle root of the merkle tree containing all necessary information for validating transactions which come after the execution of the current block.

This state commitment st should also be validated by full nodes, in the process of validating the block body $\bar{x}$. This can be done either by downloading the full state $\bar{st}$ from a peer and comparing with the result from one's own execution, or by executing the transactions $\bar{x}$ and then calculating a merkle root based on the resulting UTXO set, which is compared with st.

We will use this new block header definition in outlining our quick bootstrap protocol for light miners.

1.7.2 Quick Bootstrap Protocol

We can construct a quick bootstrap protocol by making use of the state commitment st in each block header. This commitment in a given header can be used to download and verify the UTXO set after the execution of the corresponding block. By relying on st, a light miner can avoid downloading $\bar{x}$ for all blocks which were mined before they joined the network, while still being able to validate transactions based off of the most recent state.

Therefore, our quick bootstrap protocol differs from the regular full node bootstrap protocol in that it does not have the light miner validate the entirety of the transaction graph, but only the portions of the graph which are included after the light miner boots. Further, only transactions included in blocks after the light miner boots need to be executed, greatly simplifying the boot process.

The protocol is as follows:

1. download and validate the header chain
2. download the state $\bar{st}$ of the chain tip and validate the merkle tree
3. accept transactions into the mempool and form valid transactions into a block (validated using the state $\bar{st}$)
4. mine the header of this block off of the chain tip of the longest chain
5. when new blocks are announced, validate as normal (download the block header, validate, download $\bar{x}$, validate, etc.).

We can see that this protocol does not require the downloading of any portion of the transactions $\bar{x}$ of any block already included in the chain at the time of booting, in keeping with our usage of "light" with respect to light clients.

1.7.3 Light Miner Security

Since light miners do not validate any transactions that were on the blockchain before they booted, they are liable to begin mining off of an invalid chain tip, potentially contributing their hashing power to the adversary's chain. We rely upon the Common Prefix guarantee of the chain to avoid this, by starting with block $C[-k]$ to begin our full validation of the blockchain—where we accept $C[-k]$ as being the tip of a valid chain—and validating the k most recent blocks (downloading and validating $\bar{x}$) of the longest chain which follow. This ensures that the light miner accepts only valid longest chains, by starting with the end of the commonly valid portion of the chain.

Properties of Full Node, Full Miner, and Light Node			
Properties	Miner	Full Node	Light Node
Download Header	✓	✓	✓
Download Body	✓	✓	
Create Blocks	✓		
PoW Check	✓	✓	✓
Check Tx Validity	✓	✓	
Size	$\sim$ 1TB	$\sim$ 1TB	$\sim$ 100MB
Honest Majority	$\frac{n-t}{n}$		

2 Security in Earnest: Part 0

2.1 Random Oracle

So far, we have defined our random oracle such that $Pr[H(B) \leq T] = p = \frac{T}{2^k}$. The guarantees of this definition are actually stronger than collision resistance of a hash function, but we will ignore that for now. Instead, we will focus on making our random oracle more concrete.

2.1.1 Naive Random Oracle

A first pass at the implementation of our random oracle would be the following algorithm.

Algorithm 1 Naive Random Oracle

```
procedure H(B)
    y ←$ {0,1}^κ
    return y
end procedure
```

While this satisfies the statistical properties of our random oracle, this algorithm is not a hash function, and is not consistent, i.e. it does not return the same output if it is called with the same input. This inconsistency renders it useless for our purposes, so we modify it in the next section to better represent our true random oracle.

2.1.2 Our Random Oracle

Our modification of the random oracle above involves the inclusion of a state $\mathcal{T}$.

Algorithm 2 Random Oracle

```
T ← {}
procedure H(B)
    if B ∉ T then
        y ←$ {0,1}^κ
        T[B] = y
    end if
    return T[B]
end procedure
```

We see that this achieves consistency, i.e. $H(x)$ returns the same output every time across different calls with the same input x, for any x. However, our storage symbol $\mathcal{T}$ must be consistent over all parties in the network, and therefore does not represent a local mapping, but rather turns our oracle into a so-called "network function".

2.2 Synchrony

Until now, we considered time as continuous. We will now follow a synchronous model where time is broken up into rounds, each round lasting a discrete time Δ. Additionally, we will consider the

network delay to be precisely Δ. This implies that any message broadcast by an honest party at some point during the round r will be received by **every honest party** at the start of round $r+1$, all at exactly the same moment.

This model synchronizes the arrival of messages such that they all arrive at the boundary of each round, and let's us discretize time by eliminating complexities of network distance and speed. Further, the execution of our network entities is simplified to a "lockstep execution" model, where we can easily predict when any node will receive a given message, which will be precisely one round after it was broadcast.