Lecture 8 - Ethereum

Rylan Schaeffer and Vincent Yang

May 19, 2016

Note: This lecture is based on Princeton University's BTC-Tech: Bitcoin and Cryptocurrency Technologies Spring 2015 course.

What is Ethereum

- Why the name Ethereum?

  It's a metaphor referring to ether, the hypothetical invisible medium that permeates the universe and allows light to travel

- The old way: You send a message, the message goes through servers, the server sends to client

- The new way: You send a message, the messsage goes to a set of independent computers all over the world, but nobody can access the full message The message is delivered

  Every computing computer is compensated for work

- Computer owners want to monetize their work and sell ether for real dollars, dApp developers need Ether to run their apps on the network and want to by ETH for real dollars

- Since blockchain is public, we want privacy for users

History and Motivations

- Alternative decentralized ledger protocol, not an alternative cryptocurrency

- In late 2013, Vitalik Buterin published the Ethereum white paper, and later formally announces it at the North American Bitcoin Conference in early 2014

  He researched and worked in the Bitcoin community

- Vitalik Buterin works with Dr. Gavin Wood to co-found Ethereum. In April 2014, Gavin published the Ethereum Yellow Paper

  Whitepaper is summary, Yellowpaper is formal paper of research (longer)

  Through the instructions of the yellow paper, the Ethereum client has been implemented in 7 languages: C++, Go, Python, Java, Javascript, Haskell, Rust)

- You need a large bootstrapping effort to assemble resources needed to get it up and running, so they kickstarted a large network of developers, miners, investors, and other stakeholders via. a presale of Ether tokens, the currency unit of Ethereum

- In July 2014, Ethereum distributed original allocation of Ether via. 42-day public ether presale, netting 31,591 Bitcoins, worth 18,439,086 at the time, in exchange for 60,102,216 ether

  This was used to pay legal debts and for developer effort

- Olympic: In 2014 and 2015 they hosted a set of proof of concept releases through a testnet called Olympic. The dev community was invited to test the limits of the network with prize rewards on those with records, or successfully breaking the system in one way or another
- Bounty Program: Early 2015, they offer BTC rewards for finding vulnerabilities. This is still active
- Ethereum Frontier Network launched on July 30th, 2015 where developers started writing smart contracts to deploy on the network, and miners started to join to help secure the blockchain and earn ether from mining blocks

  This was the first milestone and was to be a beta version, but was much more capable/reliable than anyone expected, so more people rushed to join
- Feb 6, 2016: First Ethereum Browser - this hosts DApps, or Decentralized Apps, where you can
  1. Transfer ownership of property
  2. Issue a local currency (transaction)
  3. Create marriage contracts
  4. Create business contracts
  5. Create rental contracts
  6. Transfer Money
  7. Buy Insurance
  8. Vote
  9. Verify information
  10. Take out a loan, etc.

Bitcoin vs. Ethereum

- Bitcoin is a stack-based language
- Ethereum is a Turing-Complete language
- Turing Complete vs. Stack Based: (Automata Theory)
  - Stack Based means it relies on a stack machine model for passing parameters

    e.g. push 2, push 4, mult

    e.g. XML, HTML, JSON, Script (Bitcoin)
  - Turing Complete means it can do anything a Turing Machine can.
  - On a basic sense, for imperative languages, Turing Complete has to have:
    1. Needs a form of conditional repetition or jump
    2. A way to read and write some form of storage (variables, tape) (with infinite space or dynamically allocated)
    3. e.g. Java, C++, Python, etc.
    4. Imperative means it uses statements to change state. The alternative would be lambda-calculus based functional programming.
- Transactions confirmed in seconds compared to minutes for Bitcoin
- Difference is purpose:

  Bitcoin is an alternative to regular money, while Ethereum is a platform that facilitates peer to peer contracts and applications via its own currency vehicle.
- Market cap of Ether is more than Ripple and Litecoin, but far behind Bitcoin

How it Works

- 3 Tier Architecture:
  1. Advanced Browser as the client
  2. Blockchain Ledger as shared resource
  3. Virtual Network of Computers
- Sandwich Complexity Model:

  Bottom architecture and Top interfaces should be simple and understandable

  Where complexity is inevitable, push into the middle
- Specialized form of Cloud Computing

Ether (Gas and Fees)

- Ether is to be treated as a "crypto-fuel", a token whose purpose is to pay for computation, and is not intended to be used as or considered a currency, asset, share or anything else
- All transactions in Bitcoin are roughly the same, since they can be modeled to the same unit (BTC).
- Since Ethereum is Turing-complete, it needs to take into account

  cost of bandwidth

  cost of storage

  cost of computation
- Halting problem cannot be reliably predicted ahead of time
- Prevent denial-of-service via infinite loops?
- Basic mechanism for transaction fees:
  - Every transaction must specify a quantity of "gas" that it is willing to consume ($startgas$) and a fee it is willing to pay per unit gas ($gasprice$). At the start of execution, $startgas \ \cdot gasprice$ ether are removed from the transaction sender's account
  - All operations, including database reads/writes and computational steps consume gas
  - If a transaction execution finishes (returns), consuming less gas than needed, then the transaction executes and the sender receives a refund of $gas_rem \ \cdot gasprice$ and miner of block receives $(startgas - gas_rem) \ \cdot gasprice$
  - If a transaction runs out of gas, then all execution reverts, but the transaction is valid, and the sum $startgas \ \cdot gasprice$ is given to the miner

  Prevents shitty code
- Proof of necessity:
  - If transaction don't specify a gas limit, then a malicious user could have an infinite loop, but miners can't tell beforehand
  - Alternative to strict gas-counting, time-limiting doesn't work because it is subjecetive - some machines are faster than others
  - Entire value $startgas \ \cdot gasprice$ has to be removed at beginning so account doesn't run out half way.

    Balance checking isn't enough because the account can send it somewhere else

  - If execution didn't revert, then contracts need strong security measures to ensure that not only part a program is executed, or else some changes in contract execution could be executed but not others

- Further features:
  - 21000 gas is charged as base fee. This covers sending sender address from signature and disk/bandwith space of storing transaction
  - Transactions can have unlimited data, so the gas fee is 1 gas per zero byte and 5 gas per nonzero byte (encourage compression)
  - Memory is infinite, but gas is 1 per 32 bytes of expansion

- Economics of Gas Pricing

  Bitcoin has voluntary fees, relying on miners to act as gatekeeprs.

  The equivalent is to let senders set arbitrary gas costs. Unfortunately, since every transaction a miner includes has to be processed by the network, the vast majority of cost of transaction processing is by 3rd parties and not the miner, so tragedy of the commons might occur

  The solution is a voting system. Miners have the right to set a gas limit to be within 0.0975% (1/1024) of the gas limit of the gas block, so the gas limit should be around the median of miner's preferences.

  The hope is that this can be soft-forked into a more precise algorithm

Patricia Trees

- Merkle Trees can let us efficiently prove what happened. However, there are limitations:
  - CAN prove inclusion of transactions
  - CAN'T prove anything about current state

    e.g. Digital Asset holdings, name registrations, status of financial contracts, etc.
  - In order to tell anything about the next transaction, you need to know the authenticity of every single prior transaction.

- Every block header has not 1 Merkle Tree, but 3 trees for 3 kinds of objects:
  1. Transactions
  2. Receipts (data showing effect of transaction)
  3. State (Patricia)

  This gives capabilities of:

  1. Has this transaction been included in a past block?

     Transaction Tree
  2. Tell me all instances of an event of type X (e.g. crowdfunding contract reaching a goal from this address in the past Y time)

     Receipt Tree
  3. What is my balance?

     State Tree
  4. Does this account exist?

     State Tree

5. Pretend to run this transaction. What's the output?

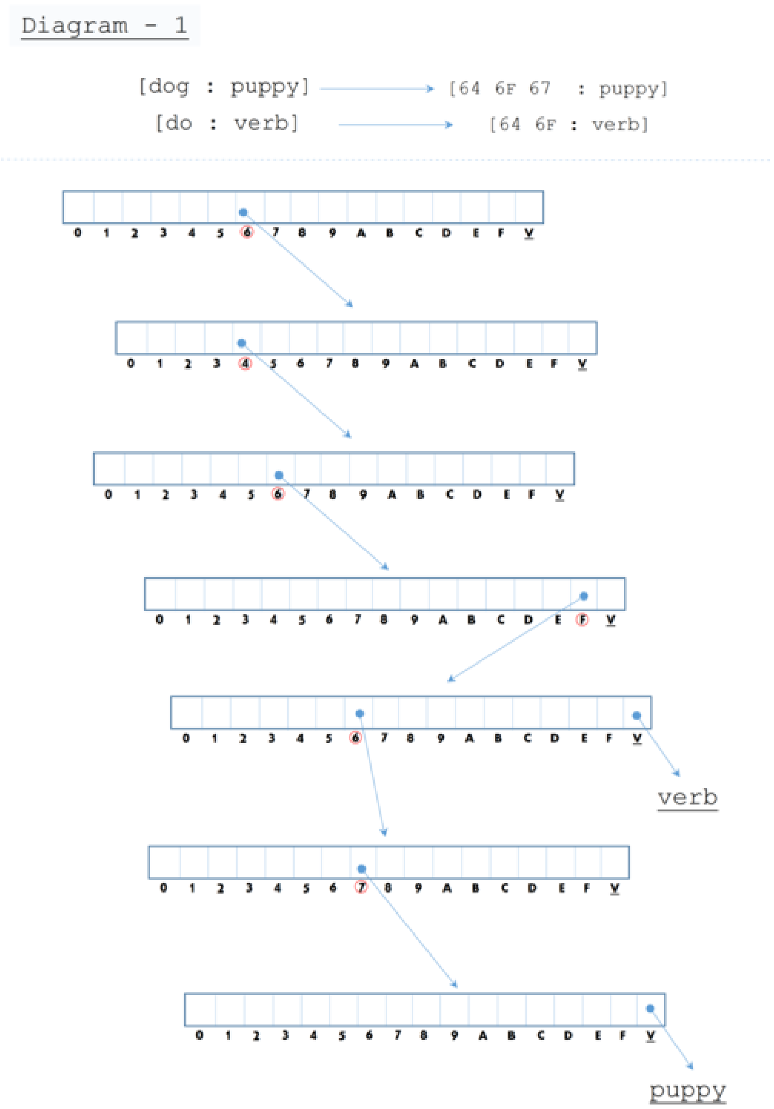
   State Tree, but more complex:

   You need a Merkle *state transition proof*

   Make claim "If you run transaction T on state with root S, the result will be a state with root S', with log L and output O". (Output is the return statement).

   To do this, the server locally creates a fake block, and pretends to be a light client while applying the transaction. It 'responds' to all of its own queries, then keeps track of returned data. Then, the server sends the client the combined data. using provided proof as database. If it reaches the same result, then it works.

- Patricia Tree:
  - Merkle trees are good data structures for transactions, because it doesn't matter how much time it takes to edit, since it's created once then forever frozen.
  - However, the state tree is different. The *State* in *Ethereum* has a key value map, where keys are addresses and values are account declarations, listing the balance, nonce, code, and storage for each account.
  - Unlike transaction history, the state has to be frequently changed - balance and nonce is often changed, and new accounts are often made.
  - We want a data structure
    - Where we can quickly find the new tree root after insert, update edit or delete, without recomputing everything.
    - Depth of tree is bounded, so an attacker can't perform a denial of service attack by making the tree super deep.
    - Root depends on data, not order.
  - Radix Tree + Merkle Patricia Tree modification:
    - Given a key/value pair that should be inserted, the key will be split so its sub-parts (e.g. characters and each value) will be a node in a tree, so if we want to retrieve a reference value we should jump from node to node representing part of the key until we find the end and there we will have the value.
    - Each node on the diagram has 17 elements which are representing slots for *[0.. f]* digits of encoding, and one more slot - the last one, for the value (if it exists)
    - In our case we will split the key to a nibbles of it's ASCII code:
    - Let's take *dog* string as a key, in order to get it's nibbles, we check its ASCII code which is [$0x64$ $0x6f$ $0x67$], now the nibbles of the key are 4 bits of each byte: $6, 4, 6, f, 6, 7$.
    - Another example is the key "do" which obviously will be encoded to $6, 4, 6, f$. ("dog" with no "g").
    - Now, let's see how to encode the key value ["do" : "$verb$"] and ["dog" : "$puppy$"] into a data structure: (Diagram 1)



-
- Problem with Radix Trees: You need over a kb to store a key that's a few hundred characters long - very inefficient
- How to improve it?? Define a too-long-path
- Insert another key/value: *"doggiestan":"awesome_place"*.

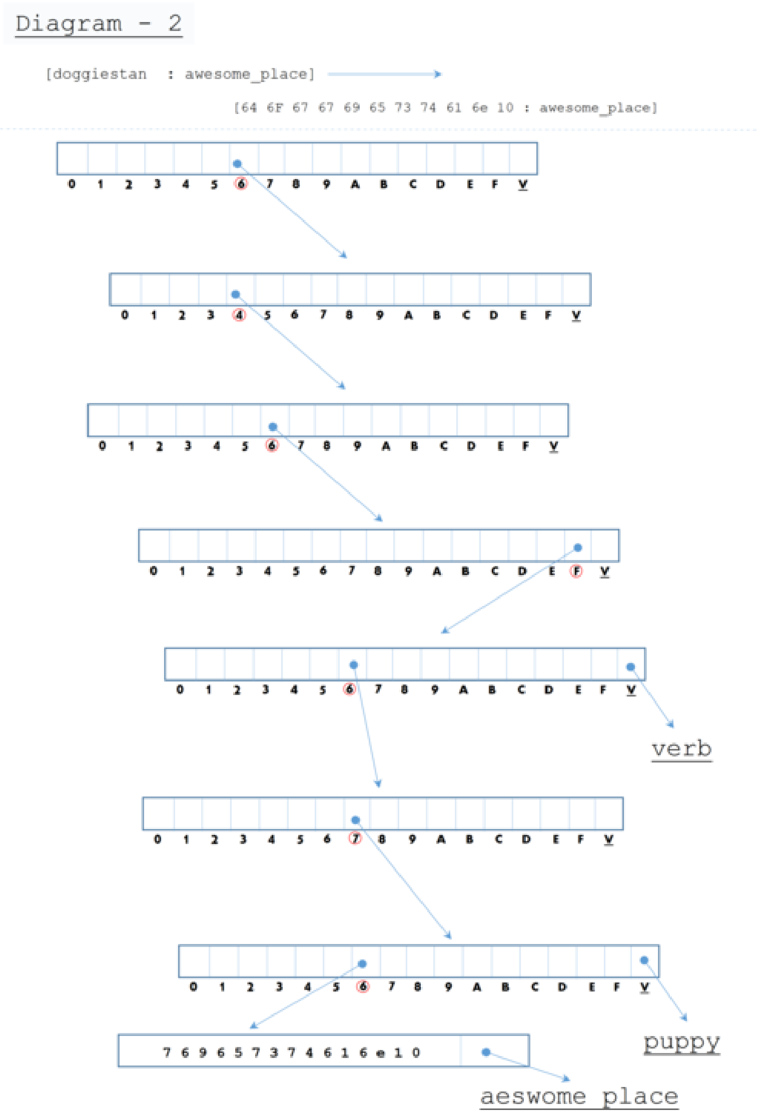
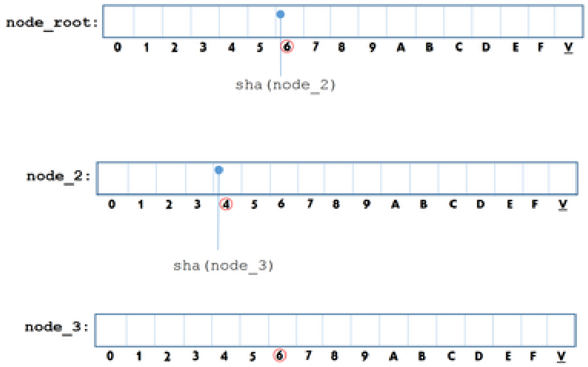


Diagram - 2
[doggiestan : awesome_place]
[64 6F 67 67 69 65 73 74 61 6e 10 : awesome_place]
0 1 2 3 4 5 6 7 8 9 A B C D E F V
0 1 2 3 4 5 6 7 8 9 A B C D E F V
0 1 2 3 4 5 6 7 8 9 A B C D E F V
0 1 2 3 4 5 6 7 8 9 A B C D E F V
0 1 2 3 4 5 6 7 8 9 A B C D E F V
verb
0 1 2 3 4 5 6 7 8 9 A B C D E F V
0 1 2 3 4 5 6 7 8 9 A B C D E F V
puppy
7 6 9 6 5 7 3 7 4 6 1 6 e 1 0
aeswome place

∗

∗ How to encode the finger print? *sha3(root_node)* function.

Diagram - 3


node_root:
0 1 2 3 4 5 6 7 8 9 A B C D E F V
sha (node_2)
node_2:
0 1 2 3 4 5 6 7 8 9 A B C D E F V
sha (node_3)
node_3:
0 1 2 3 4 5 6 7 8 9 A B C D E F V

*

- "Merkle" part comes from the fact that a cryptographic hash of a node is used as a pointer to a node, rather than a memory location
- Fully deterministic - a Patricia tree with the same (key, value) bindings is guaranteed to be exactly the same down to the last byte, with the same root hash
- O(log(n)) for inserts, lookups, and deletes
- The solution: A node in a Merkle Patricia Tree is one of:
  1. NULL (empty string)
  2. A two-item array $[key, v]$ (aka kv node)
  3. A 17-item array [$v0$ ...$v15, vt$] (aka diverge or branch node)

  In the event that there is a long path of nodes, each with only one element, we shortcut the descent by setting up a kv node $[key, value]$ where the key is the path to descend, and the value is the hash of the node.

  Also, internal nodes can no longer have values, only leaves with no children can

  To be fully generic, if we want to store "dog" and "doge" at the same time, we use a terminator symbol to the alphabet so you know it's a value.
- For a kv node, a two-item array $[key, v]$, v can be a value or a node.

  When v is a value, the key must end with the terminator

  When v is a node, the key must not have a terminator no longer have
- http://ethereumj.io/blog/2015/07/05/Ethereum-Trie/

- Given the 3 types of nodes, let's say we have a tree with the pairs *("dog", "puppy"), ("horse", "stallion"), ("do", "verb"), ("doge", "coin").*

  Convert this to hex:

  ```
  [ 6, 4, 6, 15, 16 ] : 'verb'
  [ 6, 4, 6, 15, 6, 7, 16 ] : 'puppy'
  [ 6, 4, 6, 15, 6, 7, 6, 5, 16 ] : 'coin'
  [ 6, 8, 6, 15, 7, 2, 7, 3, 6, 5, 16 ] : 'stallion'
  ```

  Build the tree:

```
ROOT: [ '\x16', A ]
A:['', '', '', '', B, '', '', '', C, '', '', '', '', '', '', '', '']
B:['\x00\x6f', D ]
D:['', '', '', '', '', '', E, '', '', '', '', '', '', '', '', '', 'verb']
E:['\x17', F ]
F:['', '', '', '', '', '', G, '', '', '', '', '', '', '', '', '', 'puppy']
G:['\x35', 'coin']
C:['\x20\x6f\x72\x73\x65', 'stallion']
```

SHA-2 vs. SHA-3

- Bitcoin uses HashCash, which uses SHA-256, which is in the SHA-2 family.
- SHA-2's predecessors were HD5 and SHA-1, which were broken. NIST (National Institute of Standards and Technology) decided to start creating a replacement for SHA-2, but SHA-2 turned out to be surprisingly robust
- Regardless, Ethereum uses SHA-3

Mining

- A block is only valid if it contains Proof of Work of a given difficulty

  In Ethereum 1.1, this is likely going to be switched for a proof of stake model
- The proof of work used is called Ethash, which is memory hard

  This way, it is ASIC resistant

  This is achieved with a POW algorithm that requires choosing subsets of a fixed resource dependent on the nonce and block header.

  This resource is called a DAG.

  Solve a contraint on a subset of the DAG, but has large memory requirements with little super-linear benefit. This is such that pooling has no advantage

  Currently CPUs are basically worthless, GPU's are dominant
- Difficulty is set such that every 12 seconds, there is a new block
- Successful PoW miner receives

  static block reward of 5.0 Ether

  All gas expended within the block, or all gas consumed by execution of all transactions in the block submitted by winner for miners. Over time, this should dwarf the static award

  Extra reward for including Uncles of the block, in the form of 1/32 per Uncle included
- Uncles are stale blocks, and are rewarded to neutralize effect of network lag on dispersion of mining rewards, to increase security

  Receive 7/8 of static block reward, max of 2 uncles per block

Casper - Proof of Stake

- Security deposit based concensus protocol - Nodes, called bonded validators, have to place a security deposit (bonding) to serve consensus
- https://blog.ethereum.org/2015/08/01/introducing-casper-friendly-ghost/

Light Clients

- Let users in low-capacity environments maintain assurance/use some features
- Know the state of an account at some time
- Check transaction confirmation
- Download a subset of information, no mining

UTXO vs. Accounts

https://github.com/ethereum/wiki/wiki/Design-Rationale