Ethereum and smart contracts

Prof. Raluca Ada Popa
Sept 12, 2018

Material based on the Ethereum white paper
Cryptocurrencies we cover in this class

Four very different cryptocurrencies each introducing different and powerful notions

- **Bitcoin**: the first one, consensus via proof of work, blockchain, miners, etc.
- **Ethereum**: smart contracts
- **Algorand**: no proof of work, committee selection, no forks, scalable
- **Zcash**: encrypted ledger, transactions via zero-knowledge proofs
Ethereum

- Ethereum extends blockchain capabilities with **smart contracts**

- **Smart contract** = code running as part of transactions
  - don’t think of it as something that needs to be complied with, but more generally as code

- Currency is **Ether**
The ledger

- The ledger is the same as in Bitcoin: same proof of work, same idea of mining and competition on extending the blockchain, same consensus criteria that the longest chain wins

- The main innovations are in how the ledger is used, e.g., smart contracts

- All participants in Ethereum replay the blockchain as in Bitcoin to verify transactions, which here also means that they are running the code of the smart contracts
Ethereum notions

- Accounts
- Transactions
- Messages
Account

Identified by a 20-byte address

Is a tuple consisting of:

- **nonce**: counter used to identify transactions
- **ether balance**
- **contract code**: written in EVM (Ethereum Virtual Machine Code), a low-level language that is Turing complete. Or you can use a higher-level language Solidity
- **storage**
Two types of accounts

- **Externally owned account**: controlled by private keys
  - It can create and send a message to another account by signing a transaction

- **Contract accounts**: controlled by a contract code
  - It gets activated when receiving a message, the smart contract code executes, can read and write from storage and send messages to other accounts
Transaction

A transaction creates state changes, and consists of:

• **recipient** of the message

• **signature of sender**

• **amount of ether** to transfer from sender to recipient

• **optional data**

• **start gas value:** the max number of steps the transaction is allowed to execute for

• **gas price:** fee the sender pays for computational step

Why do we need a max gas value?

For countering a potential denial-of-service attack, so a contract cannot stall all nodes by making them run an infinite loop.

Why do we need gas when we have ether?

Computation vs financial are different resources.
• Same as a transaction, contains all fields a transaction contains except for the gas price. The message is sent by a contract, not by an external account.
Spending gas

- Gas allowance assigned by a transaction or contract applies to the total gas consumed by that transaction and all sub-executions.

- For example:
  - an external actor A sends a transaction to B with 1,000 gas;
  - B’s contract code consumes 600 gas before sending a message to C;
  - C’s contract code consumes 300 gas before returning;

Then B can spend another 100 gas before running out of gas.
All execution happens at all the participants

- There is no central place where accounts live, every Ethereum participant keeps track of the accounts by playing the entire blockchain.

- Each Ethereum participant runs each transaction to verify it, transfers the messages it generates and runs the corresponding smart contract codes.
Ethereum state transition function

Each transaction transitions the state

<table>
<thead>
<tr>
<th>State</th>
<th>Transaction</th>
<th>State'</th>
</tr>
</thead>
<tbody>
<tr>
<td>14c5f8ba:</td>
<td>From: 14c5f8ba</td>
<td>14c5f8ba:</td>
</tr>
<tr>
<td>- 1024 eth</td>
<td>To: bb75a980</td>
<td>- 1014 eth</td>
</tr>
<tr>
<td></td>
<td>Value: 10</td>
<td>bb75a980:</td>
</tr>
<tr>
<td></td>
<td>Data: 2, CHARLIE</td>
<td>- 5212 eth</td>
</tr>
<tr>
<td></td>
<td>Sig: 30452fbedb3f7959f2ce8a1</td>
<td>contract.storage[tx.data[0]] = tx.data[1]</td>
</tr>
</tbody>
</table>
| bb75a980:        | send(tx.value / 3, contract.storage[0]) send(tx.value / 3, contract.storage[1]) send(tx.value / 3, contract.storage[2]) | [0, 235235, CHARLIE, ALICE ...
| - 5202 eth       | [0, 235235, 0, ALICE ....]  | [0, 235235, CHARLIE, ALICE .. |
| 892bf92f:        | send(tx.value / 3, contract.storage[0]) send(tx.value / 3, contract.storage[1]) send(tx.value / 3, contract.storage[2]) | [ALICE, BOB, CHARLIE ] |
| - 0 eth          | [ALICE, BOB, CHARLIE ]      | 892bf92f:        |
| 4096ad65:        | send(tx.value / 3, contract.storage[0]) send(tx.value / 3, contract.storage[1]) send(tx.value / 3, contract.storage[2]) | - 77 eth          |
| - 77 eth         | [ALICE, BOB, CHARLIE ]      | 4096ad65:        |

account


1. Check if the transaction is well-formed, the signature is valid, and the nonce matches the nonce in the sender's account. If not, return an error.

2. Calculate the transaction fee as \( \text{STARTGAS} \times \text{GASPRICE} \), and determine the sending address from the signature. Subtract the fee from the sender's account balance and increment the sender's nonce. If there is not enough balance to spend, return an error.

3. Initialize \( \text{GAS} = \text{STARTGAS} \), and take off a certain quantity of gas per byte to pay for the bytes in the transaction. The number of bytes is given by the lines of code and data info.

4. Transfer the transaction value from the sender's account to the receiving account. If the receiving account does not yet exist, create it. If the receiving account is a contract, run the contract's code either to completion or until the execution runs out of gas.

5. If the value transfer failed because the sender did not have enough money, or the code execution ran out of gas, revert all state changes except the payment of the fees, and add the fees to the miner's account.

6. Otherwise, refund the fees for all remaining gas to the sender, and send the fees paid for gas consumed to the miner.
Example

Transaction:
• 10 ether value, 2000 gas, 0.001 ether gasprice
• data 64bytes: 0-31 represents number 2, and 32-63 represents string CHARLIE

Contract code:

```python
if !self.storage[calldataload(0)]:
    self.storage[calldataload(0)] = calldataload(32)
```

Process for state transition function:
1. Check that the transaction is valid and well formed.
2. Check that the transaction sender has at least $2000 \times 0.001 = 2$ ether. If they do, then subtract 2 ether from the sender's account.
3. Initialize gas = 2000; assuming the transaction is 170 bytes long and the byte-fee is 5, subtract 850 so that there is 1150 gas left.
4. Subtract 10 more ether from the sender's account, and add it to the contract's account.
5. Run the code. It sets the storage at index 2 to the value CHARLIE. Suppose this takes 187 gas, so the remaining amount of gas is 1150 - 187 = 963.
6. Add $963 \times 0.001 = 0.963$ ether back to the sender's account, and return the resulting state.
Applications

- **Cryptocurrency**: a database with one operation: subtract X units from A and give X units to B, with the provision that (i) A had at least X units before the transaction and (ii) the transaction is approved by A.

```python
def send(to, value):
    if self.storage[msg.sender] >= value:
        self.storage[msg.sender] = self.storage[msg.sender] - value
        self.storage[to] = self.storage[to] + value
```

- **DNS**: register names and transfer ownership, no one can spoof a name

- **Decentralized organization**: an organization where members decide who can spend how much of the funds of the company, and is enforced cryptographically
Questions

• So who runs the code?

Everyone who verifies transactions in the blockchain

• How would you store a lot of data in a transaction?

Merkle trees, but they use Patricia trees (better for delete and insert)

• Other questions?