# abstract

# introduction to Ethereum smart contracts

## blockchain background

Blockchain, the technology behind Bitcoin, Ethereum, and many other crypto-currency systems, seems to be the driving technology behind the next generation of Internet, also referred to as the Decentralized Web. Blockchain is a novel solution to the age-old human problem of trust. It provides an architecture for so-called trustless trust. It allows us to trust the outputs of the system without trusting any actor within it.

A blockchain system is comprised of several key components which appear in defferent variations through out all diffierent blockchain products.

Transaction – A transaction is the basic message / action that can be done in the system, it’s comprised of the transaction’s content (in Ethereum: value and/or data) and some meta data.

The transaction’s content can vary and can mean a simple transfer of the system’s underlying currency (content holds value only) , or more complex meaning like calling some function in the system (content holds data only).

Examples of transaction’s meaning in Ethereum:

- Yuval paid Yonathan 10 ETH

- Alex called function X with parameter Y in contract which is identified by address 0x123

Transactions in Ethereum are being encrypted (signed) using an elliptic curve digital signatures algorithm (ECDSA), which means that the private key holder can generate hash value to some data, and every one that holds the corresponding data, hash value and private key’s associated public address (which can also be calculated easily from the private key), can easily verify if those 3 match.

On the contrary, it will take the same verifier nearly infinite amount of resources, and some additional data to induce the private key which generated the transaction.

Block and block chain – A block is just a list of transactions bundled together, with some additional meta data.

In the blocks meta-data, hides 3 main components which hold a key rule in Proof-Of-Work block chains.

Nonce – In

## Ethereum - the world computer

## smart contracts

## decentralized app

# problem analysis and goals

## 3.1. problem – online shopping and insured delivery

## 3.2. DApp as a solution

## 3.3. requirements

# application and design

Our Dapp is based on 4 different user types: Buyer, Seller, Carrier, Carrier manager (TODO check phrases).

## Backend architecture & design

**A word about Dapp’s backend architecture**

When thinking about building a centralized app, there are some choices to be made regarding which servers & cloud providers to use.

The beauty of Ethereum based Dapps, is that that although having some choices in tools and languages when developing smart contracts, when moving to production, the choices collapse to one, since every node in the Ethereum network is doing your Dapp’s backend calculations.

Contract language

There are quite few languges to choose from which have an EVM (Ethereum Virtual Machines) compilers:

TODO – list some languages

Every language has its pros and cons, we chose to use solidity based on the fact that its the most commonly used contract language, and its community is the most broad and active.

That was a key factor when using a technology which its main issue is that its still under on-going development and the change rate is very rapid.

Moreover, one of Ethereum’s community main guidelines is that, if a change is significant enough, it will be implemented, even with the high cost of backward-**In**compatibility.

**Design**

Contract oriented design is very similar in concept to object-oriented, and the same SOLID and GRASP principles are the ones leading choices.

**Contracts**:

**Package manager contract:**

Package manager is the only contract (not including helper contracts) which we deployed to the network, and his sole purpose is to use as an easy interface for other contracts (Carrier / package) to be created.

The other contracts are created by the package manager as result of users’ transactions to the package manager.

This is a common design pattern when using smart contracts because:

-It allows the front-end to handle only functions calls and not contract deployment.

-Version control - Package manager binary contains package and carrier binaries in it (and any other imports done), and once a binary is deployed to the system, its immutable. meaning that all contracts that was created from the same contract, have the same version

**Package contract:**

A new package contract is created for every new package tracked and insured in the system.

Package is created with the following parameters:

Relevant accounts’ Public addresses:

-Buyer

-Seller

-Carrier

-Dispute resolver

Predetermined agreed details:

-Merchandise value (Amount buyer pays for the goods) [units: WEI]

-Shipping fee [units: WEI]

-time for all sides to transfer money to the account [units: DAYS]

-maximum shipment time from the moment all sides transferred money [units: DAYS]

Package contract code defines the pricing policy, and his being terminated (SUICIDE opcode) when either there is a time-out, the buyer signs the package, the seller receives the returned package, or the dispute resolver determines how to split the refund on a disputed package.

**Package states:**

The package contract has different beheviour defined under different states it is in.

-Waiting for stakes in: contract had been created and is witing until all sides transferred all their predefined entry-stakes funds. This state is time limited, refunds according to pricing policy.

-Shipped: all funds had been transferred to the contract, and the package is either waiting for pickup by carrier or carrier shipping the package to the buyer. This state is time limited, refunds according to pricing policy.

-Returned: buyer decided to return the package to the seller, before signing the package himself. This state is time limited, refunds according to pricing policy.

-Under dispute: There is some dispute between carrier and seller when returning the package, waiting for dispute resolver to decide how to split refund.

Signing a package:

In order to track a package and to know who holds responsibility for the package at a given time, each time the package “switches hands” it needs to be signed by the receiving party.

Taking responsibility over a package has a meaning only when the side that signed it has a stake in the package. This is why only the corresponding package’s buyer, seller and carrier can sign the package.

When signing the package one needs to add a location string parameter, to give some descriptive info regarding the transfer.

Signing a package can happen either when the carrier picks up the package from the seller, when carrier switches hands, when the buyer receives the package, or when a seller receives a returned package.

**Carrier contract:**

As explained below, a package can be signed only by one of the parties’ accounts that it initiated with.

As a carrier company manager, we wouldn’t want to have an account which is shared to all of our carriers, because this account will need to have enough ether in it for gas, while all of our workers will have access to the account’s private key in order for them to sign a transaction such as a SignPackage() function call.

Carrier contract comes to solve this problem and serves as a proxy between the carriers and the packages.

Carrier contract inherites from an “Ownable” contract (see Helper functions below for details), and his owner can add or remove Ethereum accounts which are “approved” workers of that company, the data is saved in the contract’s data. The contract has a function SignPackage() which takes 2 parameters – Address of the package that needs to be signed and the “location” string needed by Package’s SignPackage() function.

Carrier contract simply Checks if the callee address was added by the contrat’s owner, and if so, we’ll forward to call to the given package address.

The contract also has a sendFundsToPackage() function which can be called by the contract owner only, and simply transfer funds to the given package address in order to activate it.

## Frontend architecture & design

## Middleware

At the moment of writing this, most of the mobile phones surely doesn’t have enough resources to run a local light node on them, and surely not a full node.

As we discussed previously, there is no need for even a light node in order to sign a transaction and send it to a transaction pool to be mined. Following that, one might ask why is that a necessity, the answer to that is based on one of any blockchain’s main characaristic – consensus.

The problem arises when needing to read data from the block-chain, to solve that we used INFURA.

The mission of INFURA is to provide the world with secure, stable, robust, balanced, fault tolerant and easily scalable Ethereum node. By doing so, it eliminates a burden for developers to maintain their own infrastructure.

INFURA presents end-points for Ethereum through a JSON-RPC API including the web3 and eth methods. You can also use INFURA to broadcast signed raw transactions to the Ethereum blockchain.

INFURA includes a service layer called ‘Ferryman’ that provides intelligent routing of incoming requests allowing for incoming requests to be directed to Geth, Strato, EthereumH, Nethereum, Parity or any other back-end client that may be optimized for certain operations.

INFURA is available for the Ethereum Mainnet (ETH), and various Testnets (Morden, Ropsten, Rinkeby).

Web3j and truffle offers a built-in support for using infura and other remote nodes through the use of http service class.

# development process

As any software project, the process involces the usage of different languages, tools and frameworks.

In the following section we will discuss the main, smart contract oriented tools we used.

## writing smart contracts in solidity

Solidity is a contract-oriented, high-level language for implementing smart contracts. It was influenced by C++, Python and JavaScript and is designed to target the Ethereum Virtual Machine (EVM).

Solidity is statically typed, supports inheritance, libraries and complex user-defined types among other features.

Using solidity is extremely intuitive and easy to start for people who have experience with programming and understands the basics of Ethereum.

Instead of classes you have “contract” and “library” entities.

The language data-types are pretty standard and include the known: bool, integer, fixed-point, enum, arrays, strings, etc.

A new and interesting data type that is worth mentioning is “Address” which holds a 20 byte address value. Address type has a useful member functions such as balance() and transfer().

Solidity was invented to help developers to create smart contracts in the most secure, fast, and intuitive way, and it does that in a great manner.

Solidity written contracts can be easily compiled using solc.

## compiling, testing and deployment using truffle framework

Truffle is probably the most broad and easy to use development environment, testing framework and asset pipeline for blockchains using the Ethereum Virtual Machine (EVM), aiming to make life as a developer easier. With Truffle, you get:

* Built-in smart contract compilation, linking, deployment and binary management.
* Automated contract testing for rapid development.
* Scriptable, extensible deployment & migrations framework.
* Network management for deploying to any number of public & private networks.
* Package management with EthPM & NPM, using the [ERC190](https://github.com/ethereum/EIPs/issues/190) standard.
* Interactive console for direct contract communication.
* Configurable build pipeline with support for tight integration.
* External script runner that executes scripts within a Truffle environment.

## ganache, testnets

## web3, web3j

# summary an conclusions

## summery

## security issues

## problems and solutions

## future work

# references