# Abstract

As block-chain based crypto currencies are rising, smart contracts are gaining popularity. In this project we investigate the properties a smart contracts and their applications. We attempt utilize smart contracts in order to provide a trustless environment for online shopping. The contract is designed to protect buyers and sellers from possible risks that are built in every online deal, E.g. fraud, package lost or stolen in shipment, misleading advertisement etc. We implement an application that allows users to create and configure an instance of such contract, and to interact with it intuitively and fast. We utilize Ethereum as a nearly - complete Turing machine to implement a complex contract that defines a set rules agreed by all parties involved in an online purchase. We also investigate the hacks and limitations of smart contracts, and optimize our design in terms of cost per transaction – "gas".

# Introduction to Ethereum smart contracts

## Block-chain background

hash value to some data, and every one that holds the corresponding data, hash value Block-chain, 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 different variations throughout all different block-chain 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 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

A Decentralized Application, or DApp, is an application which is mostly or entirely decentralized.

Consider all the possible aspects of an application that may be decentralized:

* Front-end software
* Back-end software (logic)
* Data storage
* Name resolution
* Message communications

There are many advantages to creating a DApp that a typical centralized architecture cannot provide:

1) Resiliency: by having the business-logic controlled by a smart contract, a DApp back-end will be fully distributed and managed on a block-chain platform. Unlike deploying an application on a centralized server, a DApp will have no downtime and will continue to persist as long as the platform is still operating.

2) Transparency: the on-chain nature of a DApp allows everyone to inspect the code and be more sure about its function. On the same note, any interaction with the the DApp will be stored forever in the blockchain.

3) Censorship Resistance: as long as a user has access to an Ethereum node (running one if necessary), the user will always be able to interact with a DApp without interference from any centralized control. No service provider, or even the owner of the smart contract, could alter the code once it is deployed on the network.

#### Smart contracts "back end"

In a DApp, smart contracts are used to store the business logic (program code) and the related state of your application. You can think of a smart contract replacing a server-side (a.k.a. "back end") component in a regular application. This is an oversimplification, of course. One of the main differences is that any computation executed in a smart contract is very expensive and so should be kept as minimal as possible. It is therefore important to identify which aspects of the application need a trusted and decentralized execution platform.

Ethereum smart contracts allow you to build almost arbitrarily complex architectures in which a network of smart contracts call and pass data between each other, reading and writing their own state variables as they go. We have to add "almost" in our description, because the amount of computation that can be done in one transaction will always be limited to some degree, as specified by the block gas

limit. After deploying your smart contract, your business logic could well be used by many other developers in the future.

One major consideration of smart contract architecture design is the inability to change the code of a smart contract once it is deployed. It can be deleted if it is programmed with an accessible SELFDESTRUCT opcode, but other than complete removal, the code cannot be changed in any way.

The second major consideration of smart contract architecture design is DApp size; a really large monolithic smart contract may cost a lot of gas to deploy and use. Therefore, some applications may choose to have off chain computation and an external data source. Keep in mind, however, that having the core business logic of the DApp be dependent on external data (e.g. from a centralized server) would mean your users will have to trust these external resources.

#### Front end (Web User Interface)

Unlike the business logic of the DApp that requires a developer to understand the EVM and new languages such as Solidity, the client side interface of a DApp can use basic web technologies (HTML, CSS, JavaScript, etc). This allows a traditional web developer to utilize the tools, libraries and frameworks they are familiar with using on a regular basis. Interactions with Ethereum, such as signing messages, sending transactions and key management are often conducted through the web browser, via an extension such as MetaMask.

The front-end is usually linked to Ethereum via the web3.js Javascript library, which is bundled with the front-end resources and served to a browser by a web server.

#### Data storage

Due to high gas costs and the currently low block gas limit, smart contracts are not suited to store or process large amounts of data. Hence, most DApps will utilize off-chain data storage services, meaning they store the bulky data off the Ethereum chain, on a data storage platform.

# Problem analysis and goals

## Problem – online shopping and insured delivery

Online shopping have become increasingly popular in the past decade, consequently causing billions of packages being shipped across the globe. However, the online market is still only a relatively small share of total retail spending. We assume that the risks involved in an online purchase still discourage some potential customers and retailers from doing certain deals in that manner. Such risks could be classified into three intrinsic problems of an online deal:

1. **The shipment** - possibly from a different country. Both customer and seller must put their trust in third party, to ship the product between them. The product could get damaged, lost or stolen.
2. **Customer can't "Try before buy"** – customer buys a product based on photos or a verbal description, which increasing the chance of dissatisfaction when getting the actual product.
3. **Fraud -** Online shopping scams involve scammers pretending to be legitimate online sellers, either with a fake website or a fake ad on a genuine retailer site. The fear of fraud deters many from buying online.

We believe a solution for the problems above could significantly increase many online shops business cycle.

## Requirements document

1. **Introduction**
   1. We will develop a package delivery system – an application that will constitute a contract between all parties involved in an online purchase: the **Buyer,** the **Seller,** and the shipping company – **Carrier.**
   2. The system shall provide protection to parties in the following cases:
      1. In case of **seller**'s fraud, **buyer** will get a full refund.
      2. In case of lost/damaged/stolen package, **carrier** will pay severance seller for the merchandise value. **Carrier** will also refund **buyer** for any shipping fees paid.
      3. The system shall provide a **buyer** the option to return package to seller in case of dissatisfaction.
   3. The system shall allow each party to keep track of a package's location along the shipping process.
   4. The system shall allow **buyers** and **sellers** to choose between multiple **carriers.**
   5. The system shall enforce agreement of all parties on any other terms of shipment.
2. **Functional requirements**
   1. The system shall allow any user to create a new **package** object, and set the following parameters, as agreed prematurely :
      1. Merchandise value.
      2. Shipping fees.
      3. Maximum time for other parties to approve terms and enter as "stake holders".
      4. Maximum time of shipment.
   2. In addition, creator shall provide any additional details such as other parties' details.
   3. The terms above we'll be enforced by the system and cannot be changed by anyone.
   4. All parties should send the agreed stakes, values calculated from the parameters in 2.1., to the system.
   5. After all parties paid their stakes, **carrier** should pick up package from **seller,** starttransferring it to **buyer.**
   6. The system shall provide **carrier** the option to update the trajectory of a package, at any sorting/delivery station along the shipment route, for tracking purposes. This data will be visible to all parties.
   7. Only the involved parties can directly access the package and update trajectory data.
   8. When shipment is delivered, or terminated due to any other reason (lost, returned to seller, etc.), system shall pay each party its deserved amount, according to the reason of termination, as mentioned in 1.2.
   9. In case any party had not paid its stakes to system until the time specified in 2.1.3., all funds collected by the system will be returned to senders.
   10. In case package hadn't been delivered by the time specified in 2.1.4., it would be counted as a lost package.
   11. The system shall provide a user interface to quickly create package, send funds, update package trajectory.
3. **Non-functional requirements**
   1. The cost of using the system for sending a package should not exceed 10% of the shipping fee.
   2. Portability
      1. Application should run on user's smartphone – an android app.
   3. Performance
      1. Creating package, sending funds, and updating data should have a response time of seconds.
      2. Application should be scalable to increasing number of users - thousands.
      3. Application should require minimal storage on user device.
   4. Security
      1. System shall provide authentication of users' identity.
      2. Any communication with a server to execute the actions specified in 2. Shall be encrypted.
      3. System shall protect user's privacy – will not share personal details.

## DApp as a solution

Under the assumption that 'Ether' or similar crypto-currency maintains value as currency and will eventually become an acceptable form of payment, a decentralized application could provide a solution the minimizes the risks in section 3.1, and meet the requirements in section 3.2. We can define the terms for a package termination (requirements 1.2.\*) in a **smart contract**. That will provide users with immutability of the terms, transparency of the package state, authentication of the users, decentralization, and consequently trustlessness. A smart design of the contract would allow us to use minimal gas, hence meet requirement 3.1.

A front end UI, an android application in our case will allow users to interact with the contract intuitively according to requirement 2.11. , and would also be designed to meet requirements (3.2.-3.4.)

# 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 languages 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

-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 behavior defined under different states it is in.

-Waiting for stakes in: contract had been created and is waiting for 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.

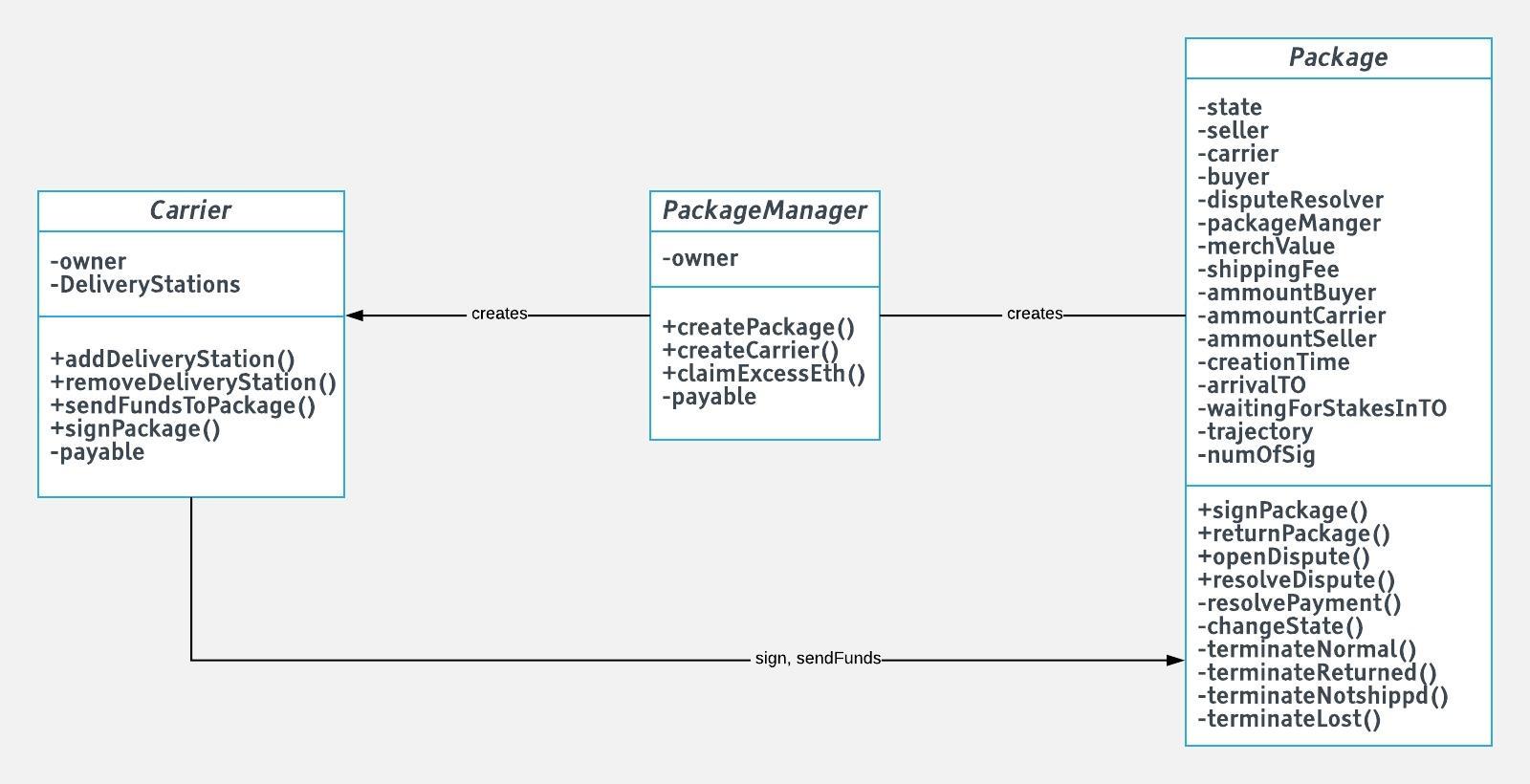
Carrier contract inherites from an “Ownable” contract (see Helper functions below for details), and his owner can add or remove Ethereum accounts which are “approved” workers of that company, the data is saved in the contract’s data. The contract has a function SignPackage() which takes 2 parameters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

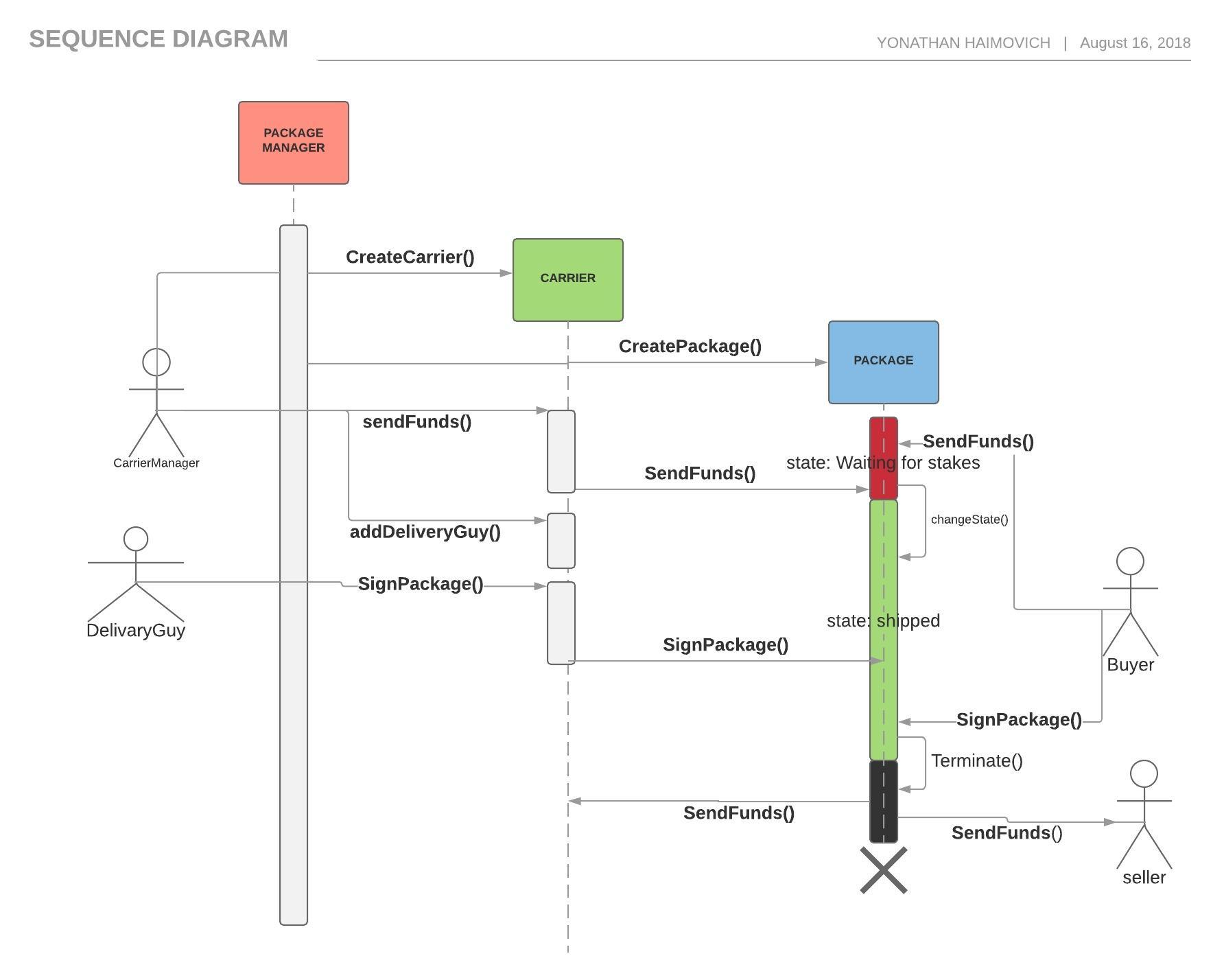
Carrier contract simply Checks if the callee address was added by the contrat’s owner, and if so, we’ll forward to call to the given package address.

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.

The backend design is visualized in a UML class diagram in figure (1), and a sequence diagram, figure (2):

**Figure (1)**



**Figure (2) **

## 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 block-chain’s main properties – 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 Test-nets (Morden, Ropsten, Rinkeby).

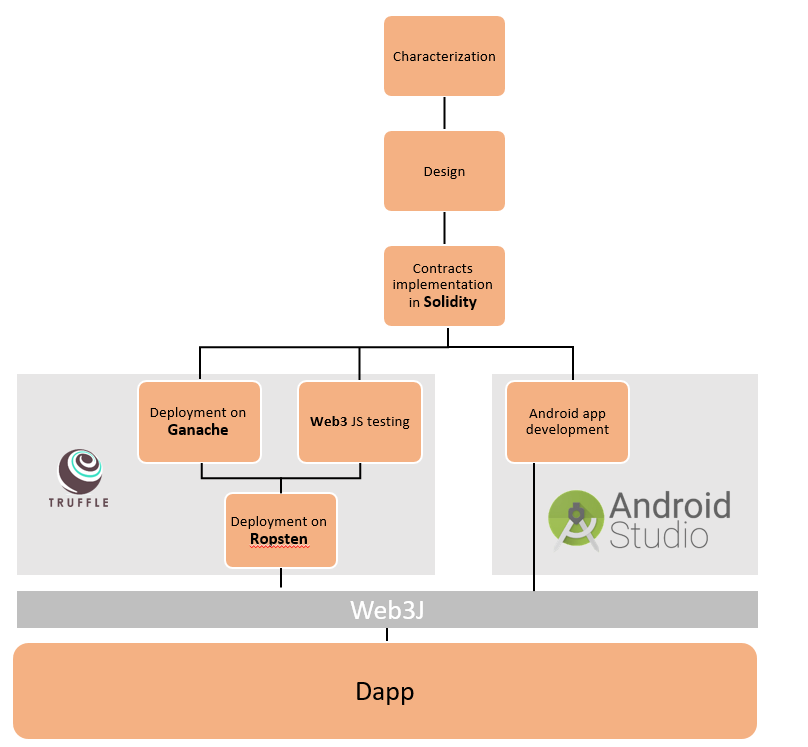
Web3j and truffle offers a built-in support for using INFURA and other remote nodes through the use of http service class.

# Development process

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

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

Figure (3) – development process



## 5.1. 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.

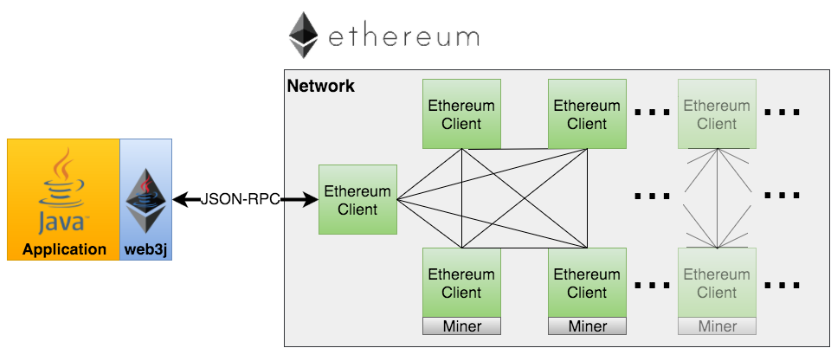
## compiling, testing and deployment using truffle framework

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



## Developer instructions – README

**Setting up the development environment**

There are a few technical requirements. Please install the following:

* [Node.js v6+ LTS and npm](https://nodejs.org/en/)
* [Git](https://git-scm.com/)

Next, install truffle by executing one command in your Node.js console:

**npm install –g truffle**

To deploy contracts on local block – chain node, for debugging and testing purposes (optional) download and install[ganache](http://truffleframework.com/ganache).

For developing the front end side of the application, download and install [android studio](https://developer.android.com/studio/).

**Clone project**

**git clone** [**https://github.com/ykurtser/Project2SmartContract.git**](https://github.com/ykurtser/Project2SmartContract.git)

**Contract developing, testing and deployment**

In the project directory, navigate to **./truffle**. This directory contains:

* **Contracts:** contains the solidity source files for our smart contracts.
* **Migrations:** Truffle uses a migration system to handle smart contract deployments. The migrations contract is an additional special smart contract that keeps track of changes to avoid unintentional spending.
* **Test:** contains both javascript and solidity tests for our contracts
* **Truffle.js:** a truffle configuration file.

Now, for editing or creating new contracts, go to ./**contracts** and edit .sol files.

To deploy new added contracts, you will have to update ./migrations/2\_deploy\_contracts.js for more details, go to <https://truffleframework.com/docs/truffle/getting-started/running-migrations>.

To deploy project's **PackageManager** developed by us, run command:

truffle migrate –reset –network <network name>

The –**reset** flag is for a new deployment of the contract, if not used the migrations contract will allow contract to be deployed only once, to avoid ether spending by mistake.

The –**network** flag specifies the network contract will be deployed to. Truffle provides a system for managing the compilation and deployment artifacts for each network, and does so in a way that simplifies final application deployment. **The network must be configured in ./truffle.js.**  the currently configured networks we have used are **ganache** (on local node), **rinkeby** and **ropsten.** For more details go to:[**https://truffleframework.com/docs/truffle/advanced/networks-and-app-deployment**](https://truffleframework.com/docs/truffle/advanced/networks-and-app-deployment)

For testing, all test files should be located in the ./test directory. Truffle will only run test files with the following file extensions: .js, .es, .es6, and .jsx, and .sol. All other files are ignored. Run command:

**truffle test .pathToTestFile/testFile.js –network <network name>**

<https://truffleframework.com/docs/truffle/testing/writing-tests-in-javascript>

**Android app**

**To build and run the application, open ./android/app in android studio.**

In order to interact with contracts with java, we use web3j library. Web3j provide a command line tool to create java adaptors to solidity contracts. If you wish to edit existing contracts or creating new ones, you will have to re generate this adaptors.

**Install web3j command line tool:**

[**https://github.com/web3j/web3j/releases/tag/v3.5.0**](%20https:/github.com/web3j/web3j/releases/tag/v3.5.0)

**For creating contract Java abstraction files:**

1. solc <.sol file path> --bin --abi --optimize -o (ex. solc ../truffle/contracts/P2PackageManager.sol --bin --abi --optimize -o ./)
2. web3j solidity generate <.bin file path> <.abi file path> -o <project src/main path> -p <project's package name> (ex. web3j solidity generate ./P2Carrier.bin ./P2Carrier.abi -o ./../android/app/src/main/ -p com.example.chaimovy.myapplication)

For a better understanding on how to interact with contracts go to [web3j documentation](https://docs.web3j.io/index.html%20)

# Summary and conclusions

## Summery

We have created a feasible solution that meets the requirements in chapter 3.2. However, a Dapp has its own problems and limitations, that have to be considered when developing block-chain based application:

* **Performance and scalability**

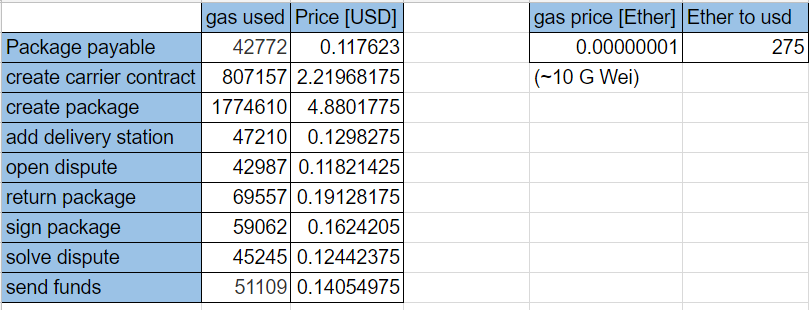
A lot of research is currently done on the way Ethereum balances gas price and block size according to the state of network. Also scalability to a large number of users depends on the state of the EVM, which is still being developed. This underlying an even more vast issue than just performance – because of Ethereum’s exponentially growing block size, the bottleneck is not regulated by  [computational performance](https://en.wikipedia.org/wiki/Computer_performance) or [network performance](https://en.wikipedia.org/wiki/Network_performance), which results in a shrinking and more centralized network due to network demands that **increasingly** exceed the average users hardware and bandwidth. This undermines a major fundamental of a Dapp – decentralization. In our project we have neglected those issues, and assumed the network is decentralized enough for our needs. However we will briefly discuss the main issues and how they have affected our design.

**Response time** - while reading and verifying on-chain data is fast (milliseconds), any state changing activity, a transaction, depends on the miners and can take seconds or even minutes. This can be frustrating to users and also eliminate the possibility to use Dapps for any real-time or fast rate communication applications.

**Storage –** In orderto achieve complete trustlessness, one would want to have a whole ethereum node on personal device, meaning keep entire ledger of transactions and being able to verify them without relying on a third party. However storing the entire ledger is impossible – currently it would require more than 1TB. To solve the storage problem we have used INFURA, as detailed in chapter 5, this of course is a security issue, as user's private key is being sent to a third party. Another way to approach this issue would be using a light node. Light node download just the header of the blocks and it does not download all the blocks. The light client protocol allows light nodes processing about 1KB of data per 2 minutes to receive data from the network about the parts of the state that are of concern to them, and be sure that the data is correct provided that the majority of miners are correctly following the protocol. For all kind of processing, light node depends on full node peers.

* **Gas price**

As described in chapter 2, each changing state transaction requires gas. In our design we wanted to make sure gas costs remain low, a neglect able part of the shipping fees of a package. That way usage remain affordable. Although we have not made an actual business plan, that has guided us to an efficient design that minimizes gas cost. We have tried to separate transaction to specific, single responsibility methods, as well as minimize storage of contracts. We have also left some activities and features to be implemented off-chain, such as public addresses sharing. In the table blow, average gas prices of transactions are presented, both in wei and in usd:



* **Smart contracts as an alternative to legal contracts –** We have triedto cover as many cases and define them in code, to minimize any external factors of human behavior. However we could not get complete coverage against all kind of fraud or mal **–** usage of the system. For example if a buyer refuses to sign a package and "steel" it from carrier, or if seller refuses to sign a returned package. For those cases we use the "dispute resolver" as an authority accepted by all parties. That way of solving disputes is used today in online shopping platforms such as Ebay. While not perfectly trustless, we believe our Dapp still provide needed protection, especially for buyers. In general we conclude that smart contracts, designed properly, are a better tool than having to trust a piece of paper signed by a person.

## Future work

* We believe further work can be done designing contract more specific to the product, or type of deal. Perhaps not only packages but services too. This can be done by inheritance from contract "Package".
* Further optimization in terms of gas cost, as network scales up.

# References and resources

**Literature**

* 1. Mastering bitcoin - 2nd edition:

<https://github.com/bitcoinbook/bitcoinbook>

* 1. Mastering Ethereum:

<https://github.com/ethereumbook/ethereumbook>

* 1. Official Ethereum white paper:

<https://github.com/ethereum/wiki/wiki/White-Paper>

* 1. Official Ethereum yellow paper:

<https://ethereum.github.io/yellowpaper/paper.pdf>

* 1. Solidity full documentation:

<https://solidity.readthedocs.io/en/latest/>

* 1. Upgradeability-using-unstructured-storage –

[https://blog.zeppelinos.org/upgradeability-using-unstructured-storage/](https://blog.zeppelinos.org/upgradeability-using-unstructured-storage/%20%20%20%20)

**Blogs**

* 1. <https://hackernoon.com/the-ethereum-blockchain-size-has-exceeded-1tb-and-yes-its-an-issue-2b650b5f4f62>
  2. <https://hackernoon.com/how-to-not-destroy-millions-in-smart-contracts-pt-1-bdefac3656b7>

**Tools and frameworks**

1. Truffle Framework - documentation and tutorials:

<http://truffleframework.com/>

1. Open zeppelin - framework of reusable and secure smart contracts in Solidity

<https://openzeppelin.org/>

1. Ropsten testnet virtual network block explorer –

<http://ropsten.etherscan.io>

1. **Light-client-protocol**

<https://github.com/ethereum/wiki/wiki/Light-client-protocol>

**Online IDEs:**

1. Remix

[https://remix.ethereum.org/#optimize=false&version=soljson-v0.4.19+commit.c4cbbb05.js](https://remix.ethereum.org/%23optimize=false&version=soljson-v0.4.19+commit.c4cbbb05.js)

1. Ethfiddle
2. <https://ethfiddle.com/>

**Gas statistics**

1. <https://ethgasstation.info/>

**Android**

1. web3j library:

<https://web3j.io/>