
therless
by Red Babel.

1 Ethereum

Ethereum is a platform intended to allow users to easily write decentralized applications (Đapps) that use blockchain technology[1]. A decentralized application is a distributed application like any others (hence composed of multiple parts possibly remote to one another), with the distinguishing trait that each part is individually able to do its job without depending on the other parts. Rather than serving as a front-end for selling or providing a specific service, a Đapp is a tool for people and organizations independent of one another, to interact transactionally without the need of any centralized intermediary (as it occurs in classic client-server solutions). Intermediation functions, such as filtering and identity management[2] are either handled directly by Ethereum or left open to third-party provisions, using tools like internal token systems and reputation systems[3], to ensure that users are offered high-quality (e.g., responsive, trustworthy, available) services.

The Ethereum blockchain can be described as a blockchain **with a built-in programming language**, or as a consensus-based globally-executed virtual machine. The part of the protocol that handles the network internal state and computation is referred to as the Ethereum Virtual Machine (EVM). The EVM can be thought of as a large decentralized computer containing objects, called "accounts", each of them able to maintain an internal database, execute code and communicate.

The EVM allows code to be verified and executed on the blockchain, ensuring that it will be run the same way on any machine where a party may reside. This code is contained in "smart contracts". All nodes process smart contracts to verify the integrity of the contracts and of their outputs. The applications that use smart contracts for processing the data on the EVM are called Decentralized Applications (Đapps). Every time the EVM performs a computation (i.e.: running smart contracts, making transactions), the user of the computation must pay for that execution. The payment is calculated in Gas. A unit of Gas is paid in Ethereum cryptocurrency[4] (Ether, or, short, ETH).

1.1 Blockchain

In order to understand what Ethereum is, we should take a step back and explain blockchain and how it works. At its heart, a blockchain is a shared database, called a ledger. Much like a bank, the ledgers of simple blockchains keep track of currency (in this case, cryptocurrency) ownership, who owns what, and who owes what to whom. Unlike a centralized bank, however, everyone (every node) has a full copy of the ledger and can verify anyone's accounts. This is the distributed (or decentralized) part of the chain[5]. Each connected device with a copy of the ledger is called a node, see figure 1.

[1] https://ethereumbuilders.gitbooks.io/guide/content/en/what_is_ethereum.html
[2] It addresses the need to ensure appropriate access to resources across heterogeneous technology environments.
[3] A system that allows users to rate each other in online communities in order to build trust through reputation.
[4] https://ethereum.org/ether
[5] For the remainder of the document we will use interchangeably the terms blockchain/chain.

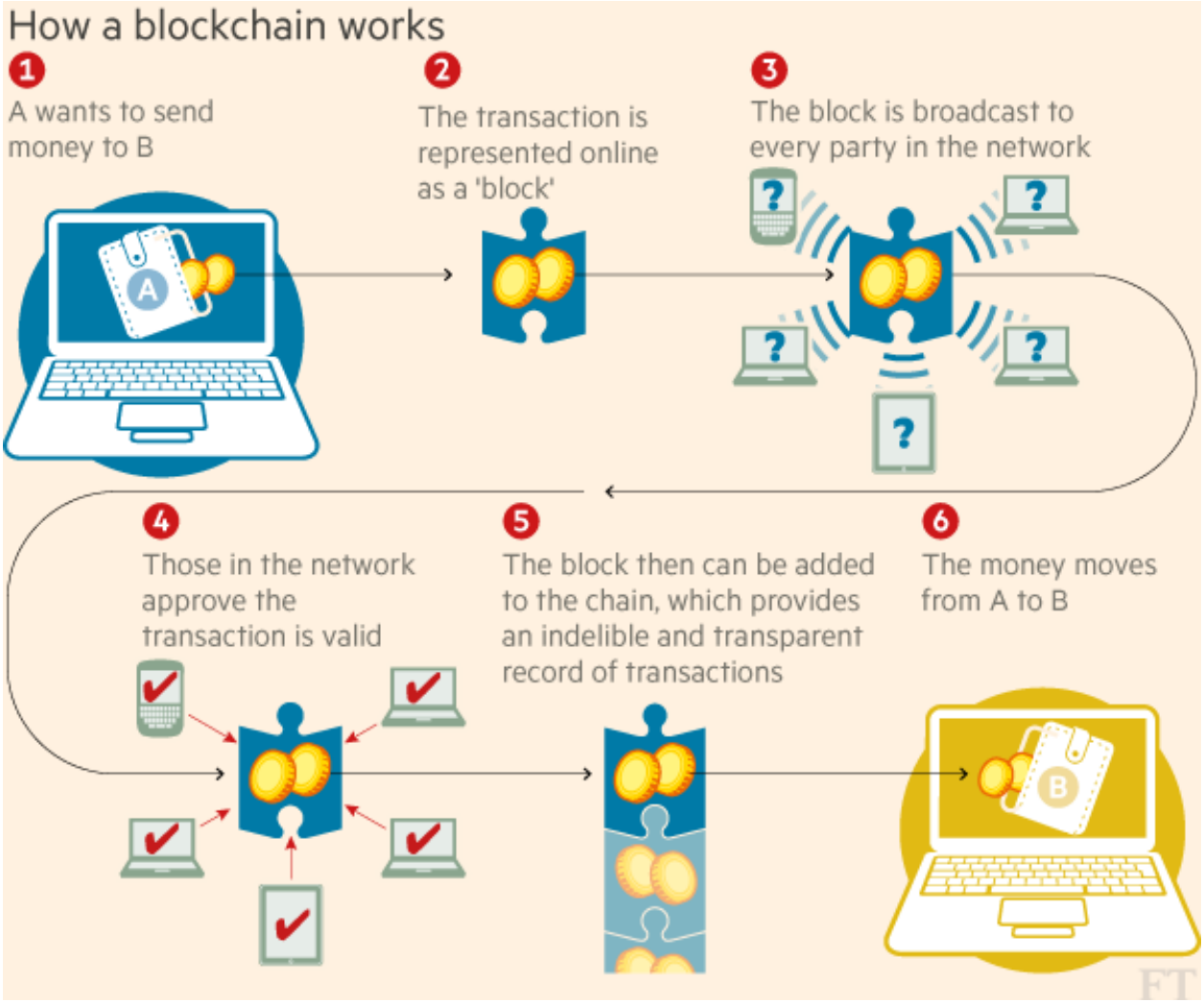


Figure 1: How blockchain works. Source: the Financial Times, "Technology: Banks seek the key to blockchain"[6].

Interactions between accounts in a blockchain network are called transactions. They can be either monetary transactions, such as sending someone some Ether, or transmissions of data items, like a comment or username. Every account on the blockchain has a unique signature, which lets everyone know which account initiated the transaction.

Blockchains eliminate the problem of trust with the following key advantages over previous databases:

- Full Decentralization: Reading/writing to the database is completely decentralized and secure. No single person or group controls a blockchain;
- Extreme Fault Tolerance (seen as the ability to handle data corruption): While fault tolerance is not unique to blockchains, they take this ability to its logical extreme of having every party that shares the database to be able to validate its changes;
- Independent Verification: Transactions can be verified individually by any single party, without the need for others. This is sometimes referred to as disintermediation.

[6] https://www.ft.com/content/eb1f8256-7b4b-11e5-a1fe-567b37f80b64

1.2 Smart Contracts

A smart contract is program code that runs on the EVM. Smart contracts can accept and store Ether, data, or a combination of both. Then, using the logic programmed into the contract, it can distribute that Ether to other accounts or even other smart contracts. Smart contracts are written in a language called Solidity[7]. Solidity is statically typed, and supports inheritance, libraries, complex user-defined types, and numerous other features.

Figure 2 below shows an example of the working of a smart contract embedding an escrow function. An escrow is a place to store the value of a business transaction/negotiation (e.g. money), until a given condition is fulfilled. In this example, Alice wants to hire Bob to build her a patio, and they are using an escrow contract to store their respective Ether (payment vs. compensation) before the final transaction.

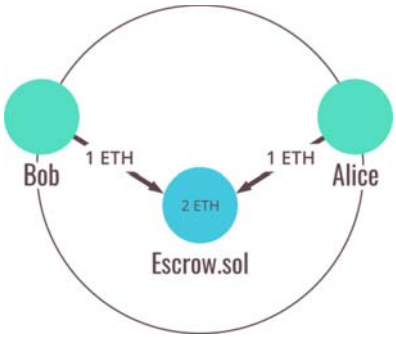


Figure 2.1

1. Alice agrees to store her payment for the patio within the escrow contract, and Bob agrees to deposit an equal amount for compensation (technically called collateral[8]) .

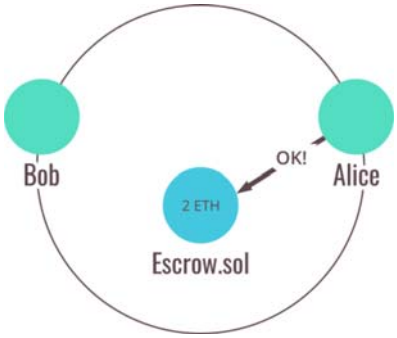


Figure 2.2

2. Bob completes the patio project and Alice gives the smart contract permission to release the funds.

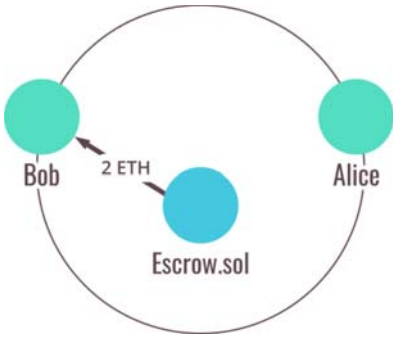


Figure 2.3

3. Bob receives Alice's payment along with his collateral.

1.3 Gas

Ethereum provides the EVM that runs on blockchain. Ether (ETH) is the fuel for the execution of programs on that virtual machine. When you send tokens, interact with a contract, send ETH, or do anything else on the blockchain, you must pay for that operation.
You are paying for the computation, regardless of whether your transaction succeeds or fails. Even if it fails, the node must validate and execute your transaction (compute) and therefore you must pay for that computation just like you would pay for a successful transaction.
When you hear someone say Gas, what is being talked about is:

- **Gas Limit**
- **Gas Price.**

[7] https://solidity.readthedocs.io
[8] A value (in this case Ether) pledged as security for the transaction. If Bob were to fail to build the patio then the collateral will be released to Alice. That rule could be written in the smart contract code.

Typically, if someone just says "*Gas*", they are talking about "*Gas Limit*". You can think of Gas limit as the amount of liters of fuel needed for a car's trip. You can think of Gas price as the cost of fuel per liter. Figure 3 summarises the analogy.

A. **Gas price**: If you want to spend less on a transaction, you can do so by lowering the amount you pay per unit of Gas. The price you pay for each unit increases or decreases depending on how quickly your transaction will be included in the ledger.
    a. During normal times (at the moment of writing, see ETH Gas station for real-time info[9]):
        - 20 GWEI, transaction fast execution time (~0.5 min)
        - 1 GWEI transaction average execution time (~1-2 mins)
        - 0.1 GWEI transaction slow execution time (~4-5 mins)
B. **Gas limit**: The Gas limit is called "the limit" because it specifies the maximum amount of units of Gas you are willing to spend on a transaction. This avoids situations where there is an error somewhere in the contract, and you spend an unlimited amount of ETH going in circles without arriving anywhere. However, the units of Gas necessary for a transaction are already defined by how much code is executed on the blockchain. If you do not want to spend as much on Gas, lowering the Gas limit won't help much. You must include enough Gas to cover the computational resources you use or your transaction will fail due to an *Out of Gas Error*. All unused Gas is refunded to you at the end of a transaction.

	Fuel price/ Gas price	Filled up tank/ Gas limit	Rome-Milan trip/ TX fee	Saved Fuel/ Reimbursed Gas	Total cost
Car	1.5 €/litre	25L	10L	15L	15€
Ethereum	10 GWEI/Gas ($10*10^{-9}$ ETH)	48000 Gas	26000 Gas	22000 Gas	0.00026 ETH

Figure 3. Analogy between gas in Ethereum and fuel in cars.

1.4 Ethereum networks

The main Ethereum public blockchain is called **MainNet**, but other networks exist, as anyone can create their own Ethereum network. On MainNet, data on the chain—including account balances and transactions—are public, and anyone can create a node and begin verifying transactions. Ether on this network has a market value and can be exchanged for other cryptocurrency or fiat currencies like Euro.

- **Local test networks**: The Ethereum blockchain can be simulated locally for development. Local test networks process transactions instantly and Ether can be distributed as desired. One good example of local test network is *ganache-cli*[10].
- **Public test networks**: developers use public test networks (or testnet) to test Ethereum applications before final deployment to the main network. Ether on these networks is used for testing purposes only and has no value.
    - **Ropsten**: The official test network, created by The Ethereum Foundation. Its functionality is similar to the MainNet.

[9] https://ethgasstation.info
[10] https://github.com/trufflesuite/ganache-cli

1.5 Ethereum Events

Events and logs are important in Ethereum because they facilitate communication between smart contracts and their user interfaces. In traditional web development, a server response is provided in a callback to the frontend. In Ethereum, when a transaction is mined, smart contracts can emit events and write logs to the blockchain that the frontend can then process. [11]
There are 3 main use cases where logs and events can be used for:

1. An event can be the smart contract return value.
    a. Let's say we have the UI that invokes a function in a smart contract (ie: *foo*). When the transaction invoking *foo* is mined, the callback inside the UI will be triggered. This effectively allows the UI to obtain return values from *foo*.

2. Asynchronous triggers with data
    a. Events can be generally considered as asynchronous triggers with data. When a contract wants to trigger the UI, the contract emits an event. As the frontend is watching for events, it can take actions, display a message, etc. when it "sees" such events.

3. A cheaper form of storage
    a. Whenever an event is emitted, the corresponding data is written to the blockchain. The data written on the blockchain is commonly referred to as *log*. The alternative form of storage on the blockchain is *contract storage*. Just to give the idea: logs storage cost 8 gas per byte, whereas contract storage costs 20,000 gas per 32 bytes.

[11] https://media.consensys.net/technical-introduction-to-events-and-logs-in-ethereum-a074d65dd61e

2 Serverless architectures

Serverless architectures are application designs that incorporate third-party “Backend[12] as a Service” (BaaS) services, or that include custom code run in managed, ephemeral containers (those that last for a single invocation only) on a “Functions as a Service” (FaaS) execution platform. By using these ideas, and related ones like single-page applications (SPA), such architectures remove much of the need for traditional always-on server installations. Serverless architectures may benefit from significantly reduced operational cost, complexity, and engineering lead time, at a cost of increased reliance on vendor dependencies and comparatively immature (as yet) supporting services.

Serverless applications are event-driven cloud-based systems where application development relies solely on a combination of third-party services, client-side logic and cloud-hosted remote procedure calls (known as FaaS)[13]. Going serverless lets developers shift their focus from the server level to the task level. Serverless solutions let developers focus on what their application or system needs to do by taking away the complexity of also handling the backend infrastructure.[14]

Serverless computing is a cloud-computing execution model in which the cloud provider runs the server, and dynamically manages the allocation of the required computational resources. Server-side logic is still written by the application developer, but, unlike traditional architectures, it is run in stateless compute containers that are event-triggered, ephemeral, and fully managed by a third party. At present, Amazon Web Services (AWS) Lambda is one of the most popular implementations of a FaaS platform.[15]

2.1 AWS Lambda

AWS Lambda is an event-driven, serverless computing platform provided by Amazon as a part of the Amazon Web Services. It is a computing service that runs code in response to events and automatically manages the computing resources required by that code[16]. A Serverless app can simply be a couple of lambda functions to accomplish some tasks, or an entire back-end composed of hundreds of lambda functions[17]. In addition to lambda functions, an application could also use other components such as AWS API Gateway (for HTTP events), AWS DynamoDB (scalable and distributed key-value and document database), or AWS S3 (object storage).

Those resources could be managed using CloudFormation, an AWS tool for deploying infrastructure. You describe your desired infrastructure in YAML or JSON, then submit your CloudFormation template for deployment. CloudFormation realizes infrastructure as code (IaC), which is the process of managing and provisioning computer data centers through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools[18].

[12] Backend in the sense of the software infrastructure hosting the execution of the application.
[13] https://hackernoon.com/what-is-serverless-architecture-what-are-its-pros-and-cons-cc4b804022e9
[14] https://readwrite.com/2012/10/15/why-the-future-of-software-and-apps-is-serverless
[15] https://martinfowler.com/articles/serverless.html
[16] https://en.wikipedia.org/wiki/AWS_Lambda
[17] https://en.wikipedia.org/wiki/Serverless_Framework
[18] https://en.wikipedia.org/wiki/Infrastructure_as_code#cite_note-AWS_in_Action,_IaC-1

2.2 Serverless framework

The Serverless Framework[19] is a free and open-source web framework written using Node.js. It provides a configuration DSL which is designed for serverless applications. It also enables IaC while removing a lot of the boilerplate required for deploying serverless applications, including permissions, event subscriptions, logging, etc. When deploying to AWS, the Serverless Framework is using CloudFormation under the hood. This means you can use the Serverless Framework's easy syntax to describe most of your Serverless Application while still having the ability to supplement with standard CloudFormation if needed.

Most importantly, the Framework assists with additional aspects of the serverless application lifecycle, including building your function package, invoking your functions for testing, and reviewing your application logs.

[19] https://serverless.com

3 Concept

Etherless is a Cloud Application Platform which allows *developers* to deploy Javascript functions in the cloud and let pay final *users* for their execution (e.g. Computation-as-a-service, CaaS), by leveraging the ethereum's smart contract technology. The platform is managed and maintained by the *Administrators*. *Developers* can deploy Javascript functions into the platform. *Users*, finally, can run those functions provided they pay the fees set by the *developers*. Such fees are partially held by the platform itself as a compensation for the expenditure of the actual function execution.

As an example, Marvin, a *developer*, wants to provide a postal code[20] lookup (PCL) service. Given a postal code, the service will return the relevant geographic information such as city, state or region. Marvin is able to implement the function but is not interested in maintaining the service itself, as he feels it may be too burdensome. He therefore decides to deploy it on *Etherless*, which, on top of the Serverless benefits of reduced operational costs, is automatically managing also the payment layer.

Norris, a *user*, can use the PCL service with the specific parameters described by the implementation. Since the usage is triggered by a smart contract, he is able to pay only for the effective usage with a granularity of a single lookup.

Etherless is built integrating the two aforementioned technologies, Ethereum and Serverless. Ethereum will be responsible for the payments and triggering the invocation of the functions. Serverless will be responsible for the execution of the functions. A command line interface (CLI) will be the bridge for actual usage for developers and users. The (virtual) communication between Ethereum and Serverless will be achieved instead by listening and emitting Ethereum events.

3.1 High level architecture

The *Etherless* architecture consists of 3 main modules:

1. *etherless-cli* is the module to which the developer interacts with *etherless*. It supports different commands with which the developer can:
    a. Configure the ethereum account;
    b. Deploy the executable code;
    c. List the functions already deployed;
    d. Run a single function. At the end of the run *etherless-cli* will return the value of the function;
    e. View the logs about the execution of a specific function.

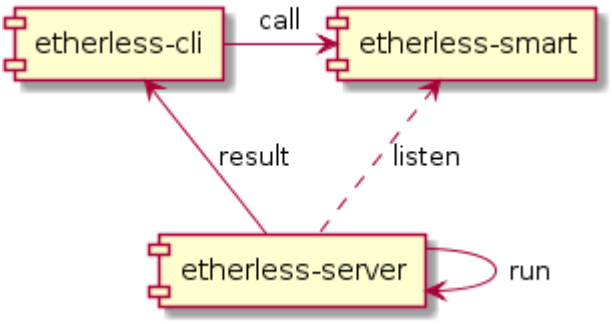
    Depending on the command, the *etherless-cli* can:
    - emit events in Ethereum, by executing smart contracts;
    - listen to events to be emitted by the Ethereum.

2. e*therless-smart* consists of a set of ethereum smart contracts that handle the communication between *etherless-cli* and *etherless-server* as well as the payment of this work using ETH as currency.

3. *etherless-server* listens to events emitted by *etherless-smart* for deploying and running lambda functions. Once the lambda function returns or raises an exception, *etherless-server* emits an event in the blockchain containing the corresponding information. The event is listened to by *etherless-cli*, which shows the result to the developer.

[20] C.A.P., Zip code, or postcode

Nota bene: A central part of this architecture relies on the concept of **event**. Solidity events provide an abstraction of it, on top of the EVM's logging functionality. Applications can subscribe and listen to these events through the RPC interface of an Ethereum client[21].



Etherless component diagram

3.2 Example of usage

Marvin developed a PCL service as a Javascript function contained in the file '*pcl.js*'. To deploy the service, he will run:

```
etherless deploy pcl.js zclookup
```

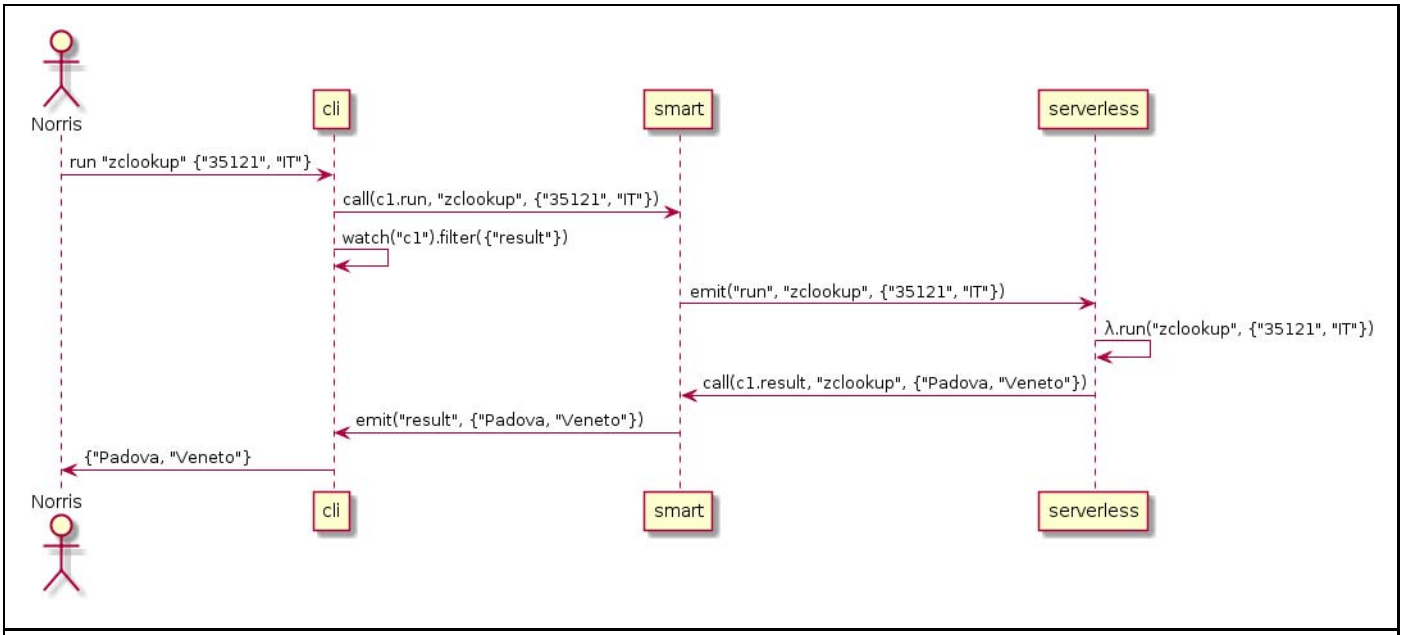
This command assumes that Marvin has already initialized *Etherless* and the Ethereum address he is using has enough ethers to deploy the function correctly.

The same assumption holds for Norris, who will use the following command to run the service:

```
etherless run zclookup 35121 IT
```

The matching contract will register the payment, emit an event that will be listened by the *etherless-server*. *Etherless-server* will perform the computation, register the result and invoke a response contract which result will emit another event. Such event is listened by the command line that will finally give back the result to Norris.

[21] https://solidity.readthedocs.io/en/latest/contracts.html#events



Etherless sequence diagram

3.3 Environments

A best practice in industry is to divide software development across different environments. Each environment provides a set of resources (e.g. networks, servers, file storage, others) needed to run an instance of the system. Every environment is capable of running the whole system, but it is entirely independent of the others. Therefore, using one resource in one environment (e. g. store a file or write in a database) does not conflict with the usage of resources in another. Environments are used to test the system before deployment to production. They also allow developers to run the system locally.

In a simple scenario, we have at least 4 environments encompassing the following flow across them. The developer builds some features and runs them in her local computer. When satisfied, she will build a suite of (verification) tests. Such tests could be run locally or as part of a continuous integration[22] process. The features will be then deployed on a staging environment so that other people can use/test such features. Finally, the release cycle will determine when the final deployment to production will occur.

This project **must** use the following minimal number of environments: Local, Test and Staging. Production is not required for the scope of this project.

1. Local: in the local environment, the Ethereum testrpc network provided by Truffle could be used. Truffle provides also a local web server to serve the front end files.

2. Test: the same network and web server could be used for the test environment.

3. Staging: must be publicly accessible. For it, Ethereum network Ropsten[23] shall be used. Ethers could be found online (e.g. https://faucet.metamask.io).

4. Production: not required, but the project should be production-ready. For production, the deployment will be done against the Ethereum main network. To interact with the main network, real Ether needs to be used. We will explore the possibility of such deployment in due time.

[22] https://docs.microsoft.com/en-us/azure/devops/learn/what-is-continuous-integration
[23] https://ropsten.etherscan.io

4 Requirements

4.1 Minimum

Etherless-cli can be installed in the system using the node package manager[24] (npm). The developer installs *etherless-cli* launching the following command via bash (assuming the name of the package is: etherless-cli):

```
npm install -g etherless-cli
```

Once installed the developer must be able to perform the following commands by CLI:

1. `etherless init`
2. `etherless deploy [file.js] [func-name]`
3. `etherless list`
4. `etherless run [func-name] [param1] … [paramN]`
5. `etherless delete [func-name]`

4.1.1 init

```
etherless init
```

is the first command the developer execute in order to access to Ethereum. The command creates a new ethereum account or the developer can decide to use an account already existing.

4.1.2 deploy

```
etherless deploy file.js myFunction
```

allows the developer to deploy to *etherless-server*, the javascript function exported in `file.js` under the name `myFunction`. `myFunction` is also the handler of which the function will be available to the user once deployed.
`deploy` is a synchronous command. Once launched, it returns only when the deploy is successful or an exception if it encounters some problems.

4.1.3 list

```
etherless list
```

list all the functions deployed to *etherless-server.*

[24] https://www.npmjs.com

4.1.4 run

```
ethereless run myFunction [param1] … [paramN]
```

this command runs `myFunction`, on *etherless-server*, passing `param1, ...paramN` as parameters. The parameter order follows the order of the function signature.
`run` is a synchronous command. Once launched it returns the result of the function or, in case the functions has some execution problems, an exception.

```
etherless run add 3 4
```

4.1.5 delete

```
etherless delete add
```

allows the developer to delete the function from *etherless-server*.
`delete` is a synchronous command. Once launched, it returns only when the deploy is successful or an exception if it encounters some problems.

4.1.6 Example

```
// myfile.js
exports.add = (a, b) => a + b;
```

```
etherless deploy myFile.js add
```

```
etherless run add 3 4
```

```
etherless delete add
```

4.2 Optional

By implementing *Etherless'* basic requirements, the developer can execute the deployed code only *without* using external libraries, hence limiting the possible applications of the functions.
We want to improve over that limitation, to allow developers to leverage *npm* to import libraries into the function's source code. Npm read the dependencies versions and packages from two files *package.json* and *package-lock.json*
With the aforementioned requirement, the `etherless deploy myFile.js` command will deploy the *package.json* and *package-lock.json* files. A building phase (if needed) will be performed by the serverless counterpart based on those two files.

4.2.1 Example

```
// myfile.js
const math = require('mathjs');
exports.add = (a, b) => math.add(a,b);
```

```
// package.json
{
  "name": "add",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "author": "",
  "license": "ISC",
  "dependencies": {
    "mathjs": "^6.2.1"
  }
}
```

```
// package-lock.json
{ … }
```

```
etherless deploy myFile.js add
```

```
etherless run add 3 4
```

The developer expects *7* as result.

4.3 Technology

1. Smart contracts must be upgradable;
2. *Etherless* will be developed using Typescript 3.6 using a promise[25]/async-await[26] centric approach;
3. *typescript-eslint* must be used and enforced using ESLint[27] throughout the development process;
4. The *etherless-server* must be implemented using the Serverless Framework[28].

The source code of *Etherless* should be published and versioned using either GitHub or GitLab.
Along with the source code, the necessary documentation should be provided for the end user to use the system and for the developer to run and deploy the modules.

[25] https://basarat.gitbooks.io/typescript/docs/promise.html
[26] https://basarat.gitbooks.io/typescript/docs/async-await.html
[27] https://github.com/eslint/eslint
[28] https://serverless.com/

4.2 Warranty and maintenance

The vendor has to demonstrate at the RA *(Revisione di accettazione)* that the product works correctly and according to requirements. Fixing bugs, flaws and any not-compliance with the requirements are entirely at the expense of the vendor.

4.3 Credits and License

RedBabel holds interests in this project as **proof of concept** of the productivity of the technologies specified above. The system will be distributed under the MIT license, the developer will be mentioned in the copyright credits. Redbabel will be credited to, under the section credits in the README file. Please refer to the Italian law about the management of public bids for what is not specified in this technical specifications document.

4.4 Useful links

- https://blog.zeppelin.solutions: best practices for smart contract development;
- Truffle Framework: development framework for Ethereum;
- Etherscan.io: blockchain explorer. A search engine that allows users to easily lookup, confirm and validate transactions that have taken place on the Ethereum Blockchain;
- Ethgasstation.info: dashboard that shows information about Gas on the ethereum mainnet.
- https://truffleframework.com/tutorials/ethereum-overview;
- https://www.ethnews.com/glossary.

5 The proponent

RedBabel is a webcraft consulting firm based in Amsterdam and formed by Alessandro Maccagnan and Milo Ertola. The firm operates in Amsterdam and Italy where it holds close relationships with the respective startup ecosystems.
Contacts:

- Alessandro Maccagnan: alessandro@redbabel.com
- Milo Ertola: milo@redbabel.com

To allow for a better and seamless communication between us, the proponent, and you, the vendor, we provide a Slack group available to all group members. To access it: https://slackin-tutcjuiyym.now.sh.