Blockchains for Industrial IoT - a Tutorial

Pal Varga and Ferenc Janky
Budapest University of Technology and Economics

v1.0
2019

Abstract

Utilizing Blockchains within the Internet of Things (IoT) concept is quite a recent idea. There are already a number of use cases and supporting frameworks available, which shows its potential benefits for many domains.

There are interesting, business-driven target areas within the Industrial IoT domain, including sectors such as supply chain (including manufacturing, transportation and logistics), maintenance, energy trading, grids, and even healthcare. When compared to consumer IoT, these systems have special requirements: certain level of real-time, security, engineering complexity, multi-stakeholder visibility, fast transaction and asset traceability. While the Distributed Ledger Technology (DLT) already addresses some areas of these (such as multi-stakeholder visibility or asset traceability), Blockchain Technology (BCT) provides additional value for security, building trust, and reducing cost while accelerating transactions of service agreements. This tutorial aims to reveal the opportunities and challenges as well as presenting real-life examples together with network management aspects. First it provides an overview and definitions the BCT universe – from Assets and Blocks through Consensus Mechanisms and Distributed Ledgers to Wallets. Next, it describes some special requirements of the Industrial IoT domain together with ideas of utilizing BCT to cover these needs. While discussing benefits, the tutorial reveals some drawbacks as well. These help us answering the questions: when is it beneficial to use BCT, when is it questionable, and when is it avoidable?

Furthermore, the tutorial provides insights on various use-cases of employing BCT and smart service contracts in healthcare, electricity trading, production, asset tracking or proactive maintenance. Aside from being interesting simply because they are becoming core technologies of near-future systems, IIoT and Blockchains have a network management viewpoint as well. The IIoT end-devices need on-boarding, their data needs to be secured, authenticity needs to be checked, and trust needs to be built – all of which tasks BCT can be utilized effectively. Moreover, as part of configuration management, reliable and secure firmware distribution and upgrade can be supported inherently.

The hands-on part of this tutorial includes a multi-player bidding scenario. In here, the users (consumers) can deploy smart service contracts on in an Ethereum-based blockchain, stating their need of given amount of energy at a maximum price. The energy-provider type of players in this scenario can bid on winning the contract. The various transactions related to the bidding, energy consumption and withdrawal of gains can be followed by the players either by using the GUI provided, or by issuing commands for reading transaction parameters and service contract values stored at the blockchain.

Contents

1 Introduction

This tutorial provides a generic overview of Blockchain Technology and its feasibility for the Industrial Internet of Things (IIoT) domain. The tutorial itself consist of three parts: (i) an oral lecture with its separate presentation material, (ii) the current syllabus providing support background material for (iii) the demo on using smart service contracts in a multi-player bidding scenario.

In order to get a proper grip on the tutorial exercises, some of the main concepts of the Blockchain Technology, the Distributed Ledgers, and the Smart Service Contract are described here. Beside this, some application areas of these concepts for Industrial Internet of Things (IIoT) are discussed in brief.

The presentation material is also available for this tutorial, at
http://alpha.tmit.bme.hu/~pvarga/CNSM_2019/iot_sc_tutorial_syllabus.pdf.
The cheat-sheet for the hands-on exercises is available at
http://alpha.tmit.bme.hu/~pvarga/CNSM_2019/cheat_sheet.pdf
The source code for the demo application for the multi-player bidding scenario is available at
https://github.com/fecjanky/iot_sc_tutorial.
The demo application can be pulled from DockerHub with tag `fecjanky/iot_sc_tutorial:latest`

2 Motivation: using Blockchains for Industrial IoT

2.1 General considerations

The domain of Industrial IoT is growing in terms of endpoint numbers, application functions, and multi-player usage scenarios – among others. When compared to public IoT approaches, the industrial domain proceeds with a much smoother pace, but with great mass – so its momentum (the product of mass and velocity) is huge, as well. There are several reasons why industry adopts IoT in a more conservative way, including the requirements for real-time operation, security and safety, high reliability, complexity, engineering efficiency, return of investment, and not to forget: trust. It takes time to address all these at once; and it is very risky at any industrial partner to introduce a technological approach for which not all the elements are fully tested – especially if their current systems function well, although not optimally.

The technical requirements are already available for everyone who wish to adopt IIoT solutions; and those at the forefront of Industry4.0 movement, has already started to do so. It is the non-technical terms – e.g., return of investment and trust – that are hard to justify for every paradigm shift.

Trust among new partners is currently built through the involvement of legal and financial authorities. These introduce background checks, and issue mitigation processes if a new partnership ends up on a wrong track. The involvement of authorities seems necessary, but it does slow the process down, and it has its financial price as well. There is a more traditional way of getting trust: building it slowly by partners (and their trusted ecosystems) working together from smaller towards greater ventures.

The immutable feature of distributed legders, and heir verification through blockchain technology make it possible to leave out certain authorities from the "trust-building" process. Together with smart service contracts, the use of private blockchains within industrial partnerships or ecosystems make transactions fast, secure, safe, trusted, validated and immutable throughout the various supply chains.

While the current tutorial merely provides an overview and a few hands-on demonstrations on the above issues and solutions, readers are encouraged to study the current surveys on the details of IoT blockchain technologies [1] as well as their comprehensive comparisons and applications [2].

2.2 IoT security: a brief overview on issues and solutions

There are various known security issues at each layer of the IoT infrastructure. These can be mapped to data security principles – Confidentiality, Integrity, Availability. Figure 1 depicts an approach where each layer has different security issues and measures, and operated by different players of the IoT infrastructure, so the have separated mitigation policies. Figure 2 provides an overview of these security categories and their mitigation policies.

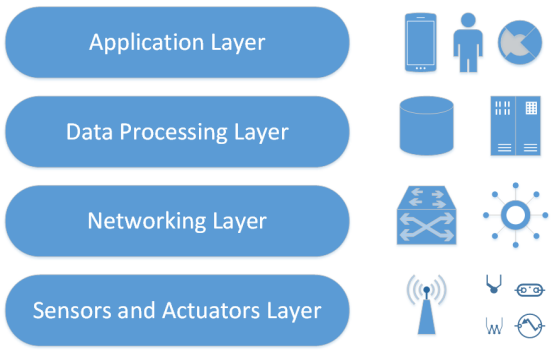

Figure 1: IoT infrastructure layers – for security assessment [3]

Layer	Threat type	Mitigation
Physical	Tampering	tamper-resistant packaging
	Eavesdropping	encryption, authorization
	Denial of Service	spread-spectrum techniques
Networking	Exhaustion	active firewalls, passive monitoring (probing), traffic admission control, bi-directional link autentication
	Collision	
	Unfairness	
	Spoofing	
	Selective forwarding	
	Sinkhole	
	Wormhole	
	Sybil	
	Flooding	
Data processing	Exhaustion	traffic monitoring
	Malware	malware detection
Application	Client app.	anti-virus filtering
	Communication	
	Integrity	testing
	Modifications	validation
	Multi-user access	process planning and design
	Data access	Traceability

Figure 2: Security issue categories and their mitigation - and overview [3]

2.3 Special features of Blockchains to be utilized by IIoT

2.3.1 Interoperability across IoT devices, IoT systems and industrial sectors

IoT data is produced and consumed at various levels of the ecosystem. Various requirements, including real-time, QoS, security, and engineering efficiency necessitate to provide borders around elements of the IoT infrastructure, within which these requirements can be satisfied when compared to the borderless scenario. Automation or IoT local clouds can be formed based on physical or logical closeness.

It was the Arrowhead framework for industrial IoT that first defined these local clouds [4], within which the interoperability of elements was based on service oriented architecture principles.

Figure 3 depicts how the IEC IIoT reference architecture – with Edge, Platform and Enterprise Levels – can be covered by the Local Clouds, and its inter-cloud communication of the Arrowhead framework.

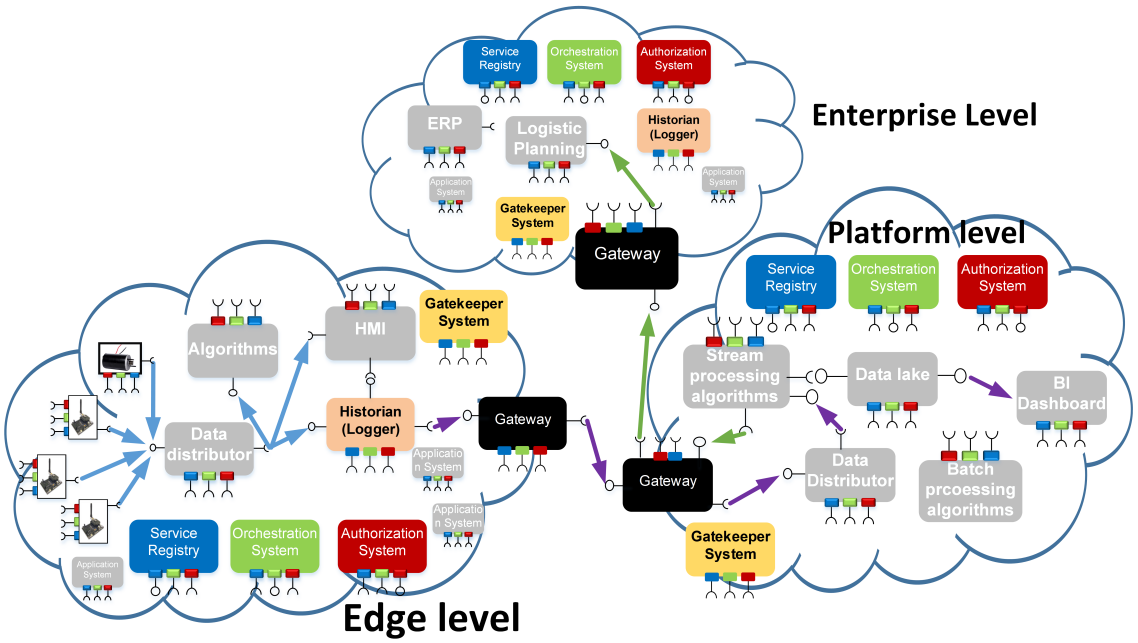

Figure 3: The IEC IIoT Reference Architecture mapped for Arrowhead Local Clouds [5]

While discovery, loose coupling and late binding are provided through the service oriented architecture approach, there are no mechanisms built in for transaction logging, smart service contract handling, or data traceability – these need to be tackled by further solutions.

2.3.2 Traceability and Reliability of IoT data

While interoperability among IoT systems can be supported by frameworks such as Arrowhead, traceability and reliability of IoT data are still issues to be solves. Distributed ledgers and blockchain technology can provide proper answers to these.

In this context, according to [1], Traceability is the capability of tracing and verifying the spatial and temporal information of a data block saved in the blockchain. Each data block saved in a blockchain is attached with a historic timestamp consequently assuring the data traceability.

Reliability of IoT data [1] is the quality of IoT data being trustworthy. It can be ensured by the integrity enforced by cryptographic mechanisms including asymmetric encryption algorithms, hash functions and digital signature, all of which are inherent in blockchains.

2.3.3 Improved Security of IoT system-of-systems through blockchains

Although it is not the basic feature of the blockchain, but its message (or transaction) initiation procedure, the data gets written to the blockchain by using PKI technologies, which covers the *Confidentiality* principle.

Due to the immutable nature of data stored at the distributed ledgers, the *Integrity* principle of data security gets covered – securing against tampering, or any sorts of modifications. Furthermore, the blockchain protects for traceability.

The *Availability* principle is supported by the distributed nature of the ledger, as a trusted database.

2.4 When is it beneficial to use BCT for IIoT?

Figure 4 depicts one of the rule-of-the-thumb decision diagrams on the topic.

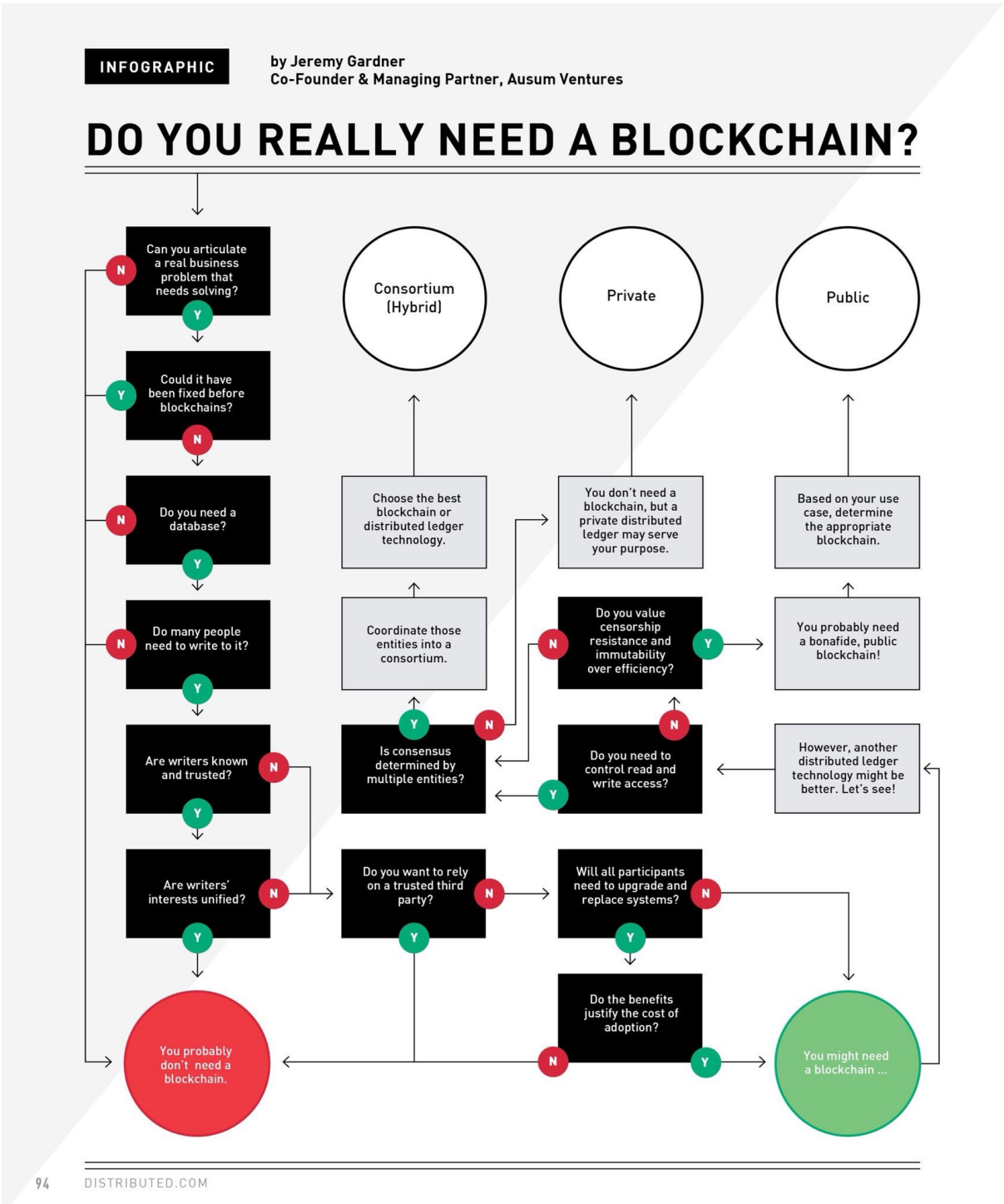

Figure 4: Do You Need to use Blockchains? (by Jeremy Gardner, 2018)

2.5 Precautions for Blockchain usage

While blockchains and their related technologies can provide answers to several issues related to security, anonymity and data privacy, trust, accelerated transactions, micropayments, cost reductions by removing middlemen, ledgers as distributed and trusted databases, etc., they also raise issues that need to be covered before their wide and unquestioned use in the Industrial IoT domain.

First, there is an issue with the number of transactions per second. The current, widely used blockchains can provide a few tens of transactions per second – whereas the industrial need is more in the thousands of transactions per second range.

Second, the mixed usage of private and public blockchains are not clearly solved in practice. While the ecosystems of given supply chains can use private blockchains, they may not easily mix with other blockchains of other private ecosystems, not to mention public blockchains and their access rights. Moreover, there is a scalability dilemma between public and private blockchains as well.

Third, security. It sounds unreasonable – too costly – to tamper around blockchain data; but it is not impossible. Where the stakes are high, the weakest link of the chain can be a blockchain as well; with its known vulnerabilities: the 51% attack, the race attack, finney attacks, hidden bugs in smart contracts, just to mention a few.

Fourth, the legal issues. While it seems a clear advantage (in price, speed and flexibility) of not using central authorities in a multi-stakeholder environment, we are talking about rigid company structures with personal responsibilities, at the same time. It requires a high amount of confidence in the technology as well as the legal consequences of tampering blockchain-stored data. As they say, you cannot solve social issues by using technology.

3 Blockchain-related concepts

This subsection aims to provide some basic references related to Blockchains. This document merely provides some initial definition together with citing references that further elaborate either the whole topic or the given term. Providing a comprehensive overview of the terms is outside of the scope of this document.

Nevertheless, readers are encouraged to check out the cited further resources and targeted tutorials on these concepts. The cited references are deliberately chosen from various sources to show the wide range of tutorial materials available for readers at various levels of interest.

3.1 Blockchain

As *techterms.com* [6] defines (with slight additions):

> "A blockchain is a digital record of transactions. The name comes from its structure, in which individual records, called blocks, are linked together in single list, called a chain. Blockchains are used for recording transactions made with cryptocurrencies, such as Bitcoin, and have many other applications – including smart service contracts for IoT.
>
> Each transaction added to a blockchain is validated by multiple computers on the Internet or on intranets of private operators. These systems, which are configured to monitor specific types of blockchain transactions, form a peer-to-peer network. They work together to ensure each transaction is valid before it is added to the blockchain. This decentralized network of computers ensures a single system cannot add invalid blocks to the chain.
>
> When a new block is added to a blockchain, it is linked to the previous block using a cryptographic hash generated from the contents of the previous block. This ensures the chain is never broken and that each block is permanently recorded. It is also intentionally difficult to alter past transactions in blockchain since all the subsequent blocks must be altered first."

3.1.1 A sample workflow of Blockchain-related processes

Figure 5 depicts a sample workflow.

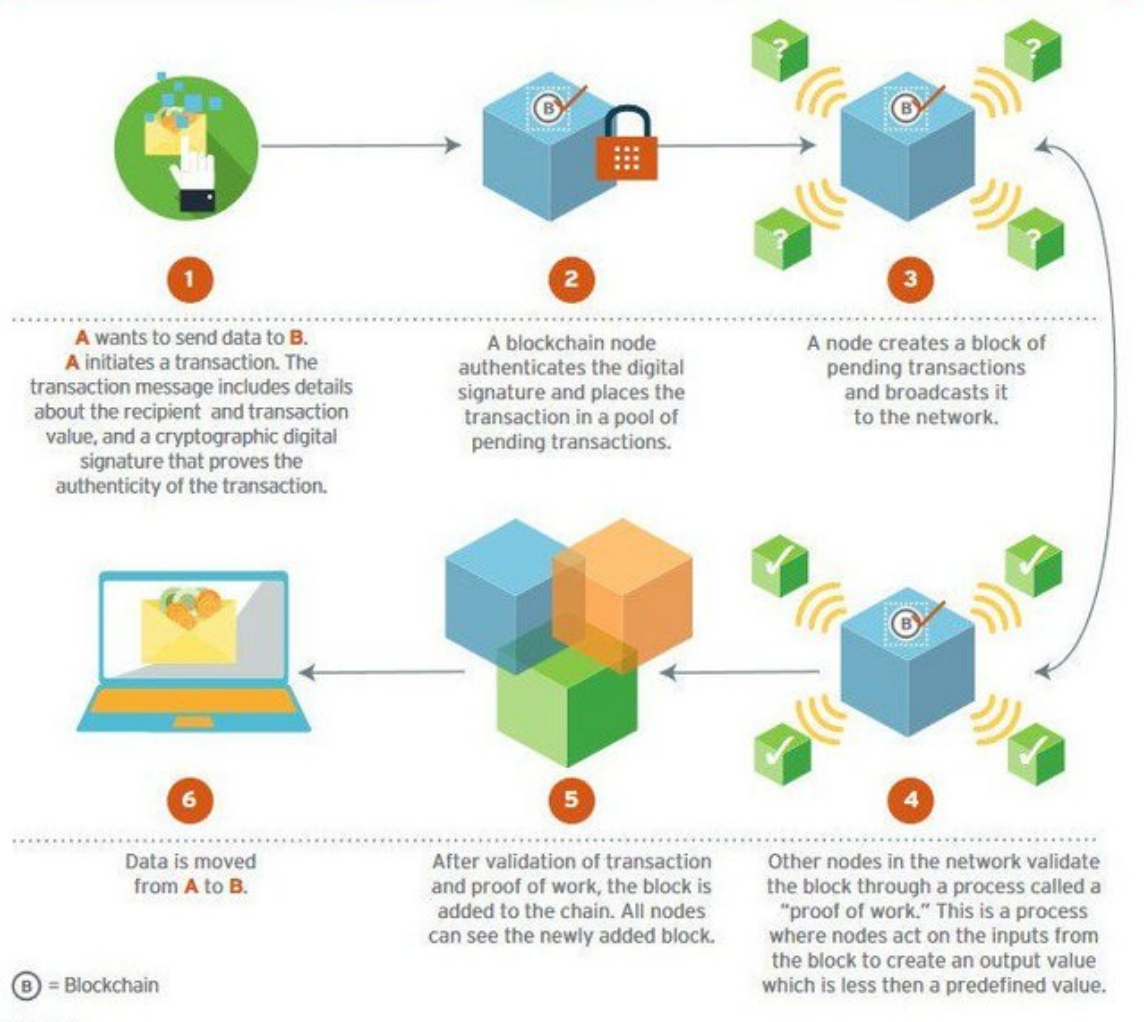

Figure 5: A typical workflow of a Blockchain-related process (https://steemit.com/blockchain/)

3.1.2 Distributed Ledger Technology

The distributed ledger database is spread across several nodes (devices) on a peer-to-peer network, where each replicates and saves an identical copy of the ledger and updates itself independently. The primary advantage is the lack of central authority. When a ledger update happens, each node constructs the new transaction, and then the nodes vote by consensus algorithm on which copy is correct. Once a consensus has been determined, all the other nodes update themselves with the new, correct copy of the ledger [7].

3.1.3 Cryptographic Hash Function

The reason to apply "hashing" on data (or the "message") put on the blockchain is making it secret so only those with proper access rights can read it, and second, making it immutable: if the message is modified, the hash value will change. After such change, it is impossible to reconstruct the original message.

A cryptographic hash function (CHF) is a hash function that is suitable for use in cryptography. It is a mathematical algorithm that maps data of arbitrary size to a bit string of a fixed size (the "hash value", "hash", or "message digest") and is a one-way function, that is, a function which is practically infeasible to invert [8].

For "An Illustrated Guide to Cryptographic Hashes" please refer to the excellent description [9] by Steve Friedl.

3.1.4 Mining

Once a message is hashed, it gets broadcasted among all the mining nodes so it would become part of a block on the chain. Mining is the process of adding transaction records to a ledger of past transactions –, i.e., the blockchain. The blockchain serves to confirm transactions to the rest of the network as having taken place.

As the network nodes are widely distributed, every miner in the network is expected to receive multiple messages from multiple vendors at any given period of time. What the miner does is he combines these messages in a single block [10].

The primary purpose of mining is to set the history of transactions in a way that is computationally impractical to modify by any one entity. By downloading and verifying the blockchain, the nodes are able to reach consensus about the ordering of events.

Mining is intentionally designed to be resource-intensive and difficult so that the number of blocks found each day by miners remains steady. Individual blocks must contain a proof (e.g. proof of stake, proof of stake, etc.) to be considered valid. This proof is verified by other blockchain nodes each time they receive a block [11].

3.1.5 Consensus algorithms

One of the outcomes of the mining procedure is to validate the transactions of a block. As part of this, miners are working to find a *nonce* that shows that the block hash satisfies an earlier-defined, well-known target criterion.

The Proof of Work (PoW) is a popular consensus algorithm; and as such, it is aimed for resolving the problem of reliability in a network involving multiple unreliable nodes. The PoW is deliberately computationally intensive for miners. It is a piece of data which is difficult (costly, time-consuming) to produce but easy for others to verify and which satisfies certain requirements. Producing a proof of work can be a random process with low probability so that a lot of trial and error is required on average before a valid proof of work is generated [10], [11].

There are several other consensus algorithms exists [12], one of the well known ones beside PoW (used by e.g. Bitcoin) is Proof of Stake (used by e.g. Ethereum), where the creator of the next block is called "validator" (instead of miner) and is chosen via various combinations of random selection and wealth or age (i.e., the stake).

3.1.6 Merkle Tree

In order to help compressing blockchain data for those nodes who are not interested in mining, but only exchanging goods, writing messages on the blockchain and checking whether a transaction is valid that is on the blockchain, the binary Merkle Hash Tree (see Figure 6) can be used.

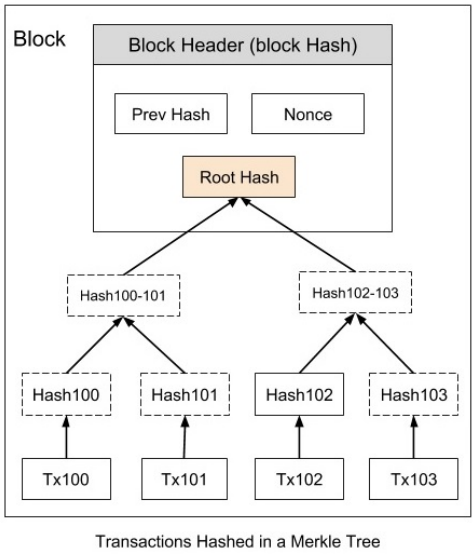

Figure 6: Merkle Tree [10]

When transactions hashed into a Merkle Tree, the client merely need to have the Previous hash, the Merkle Root, and the Nonce for Each Block header - as Figure 7 shows.

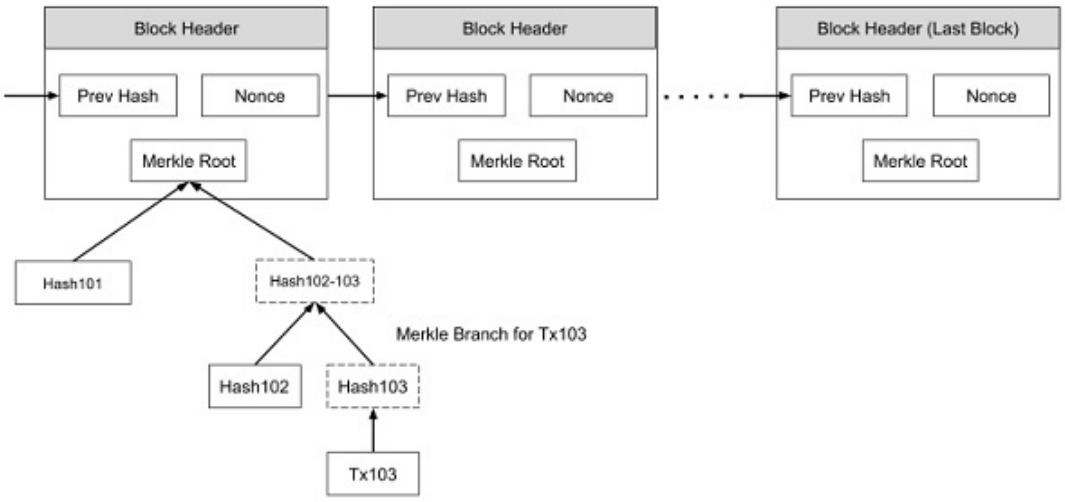

Figure 7: Transaction verification by using the Merkle Root [10]

In order to verify certain transactions made in the past, the Merkle Roots and corresponding hashes need to be used instead of all hashes for the given transactions on the block [13].

3.2 Smart Contracts

According to Wikipedia, [14] A smart contract is a computer protocol intended to digitally facilitate, verify, or enforce the negotiation or performance of a contract. Smart contracts allow the performance of credible transactions without third parties. These transactions are trackable and irreversible.

There are various kinds of contractual clauses may be made partially or fully self-executing, self-enforcing, or both. The aim of smart contracts is to provide security that is superior to traditional contract law and to reduce other transaction costs associated with contracting. See Figure 8 for some of the main features of smart contracts on blockchains.

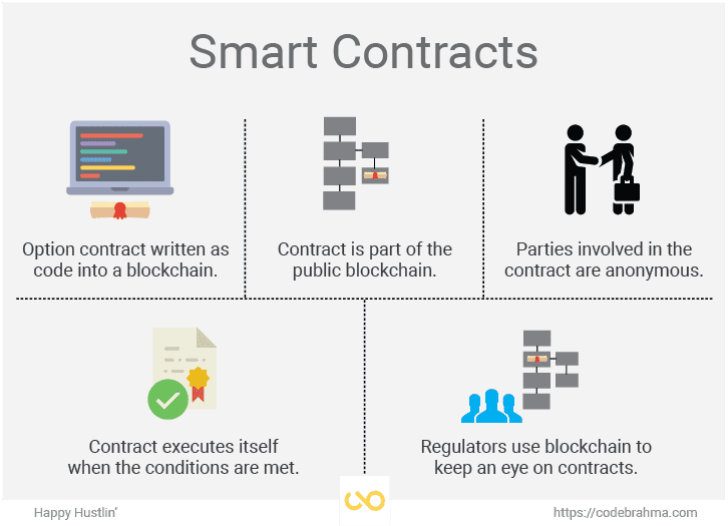

Figure 8: Smart contracts on the blockchain

The idea of smart contracts were first proposed by Nick Szabo – who, by the way, has Hungarian origins, as well.

4 Smart contracts using Ethereum

Within the Ethereum framework Smart Contracts are implemented in the form of Decentralized Applications - DApp. These DApps are defined using a higher level programming language that is either:

- Solidity, that is a statically typed programming language showing similarities to Javascript, C++ and other high level, object oriented languages [15]
- Vyper, which is a Python based language still under development [16]

Solidity is an object-oriented, high-level language for implementing smart contracts. Smart contracts are programs which govern the behaviour of accounts within the Ethereum state. Solidity 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. With Solidity you can create contracts for uses such as voting, crowdfunding, blind auctions, and multi-signature wallets.[15]

The process of how a smart contract is synthesized into "executable code" is somewhat similar to other well known paradigms of other high level programming languages. The contract code is compiled to low-level, assembly like code that is executed by the Ethereum Virtual Machine (EVM). The data and code is stored on the blockchain, the code can be called using a well known ABI that corresponds to an API that can also be used in the higher level programming language. etc.

Smart contracts define the data and the protocol on how that data can be changed. Actually since one key feature is that blockchains are immutable, change might not be the appropriate term for that but more like how the data should be transformed to get in to a next valid state. This is analogous to various well established immutable algorithms and data structures used in pure functional programming, such as persistent list, m-trie, etc.

In the following figures we summarize a simplified concept of Smart Contract operation.

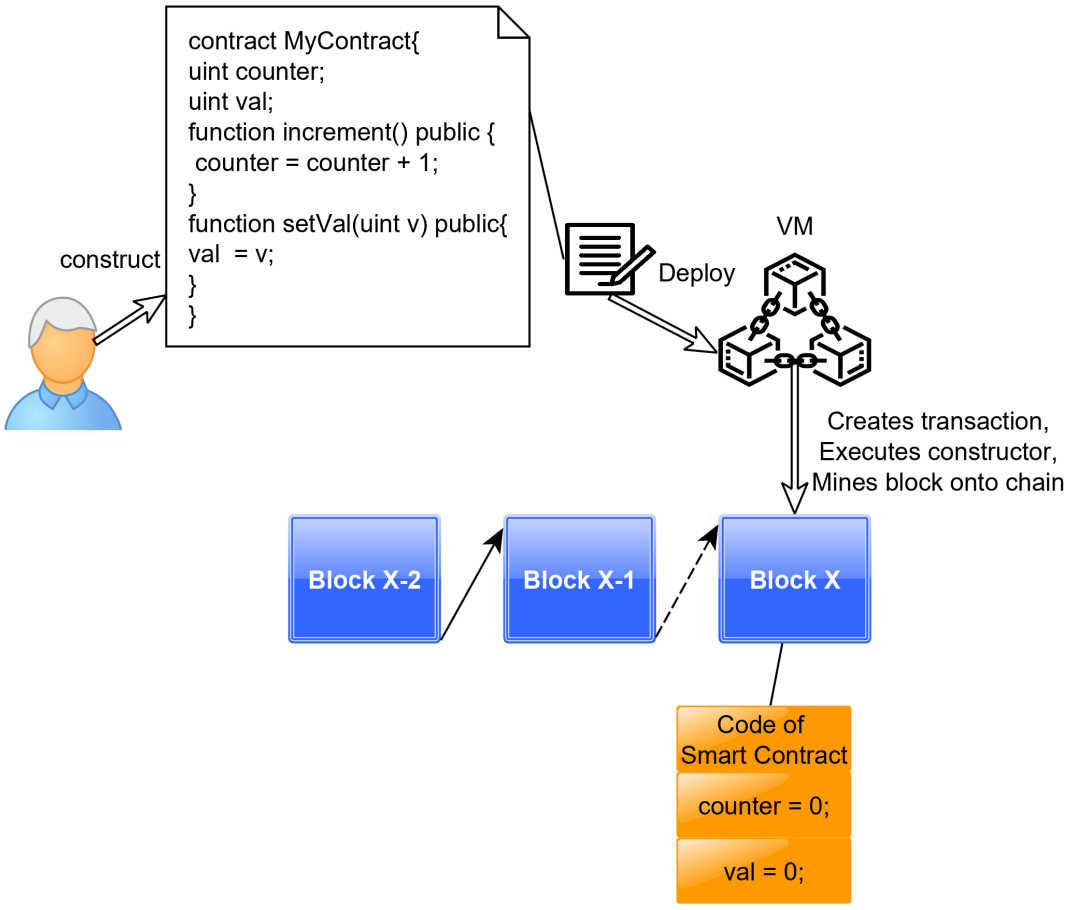

Figure 9: User constructs smart contract, VM deploys it onto the chain

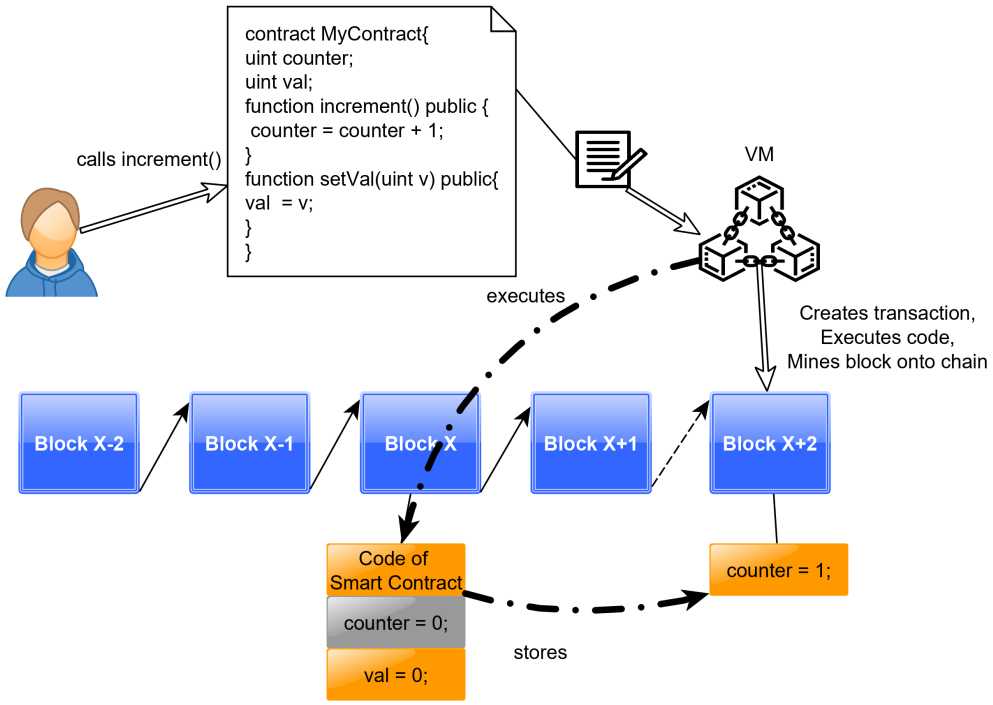

Figure 10: User calls a function of the smart contract, VM executes code, creates transaction, mines block onto the chain

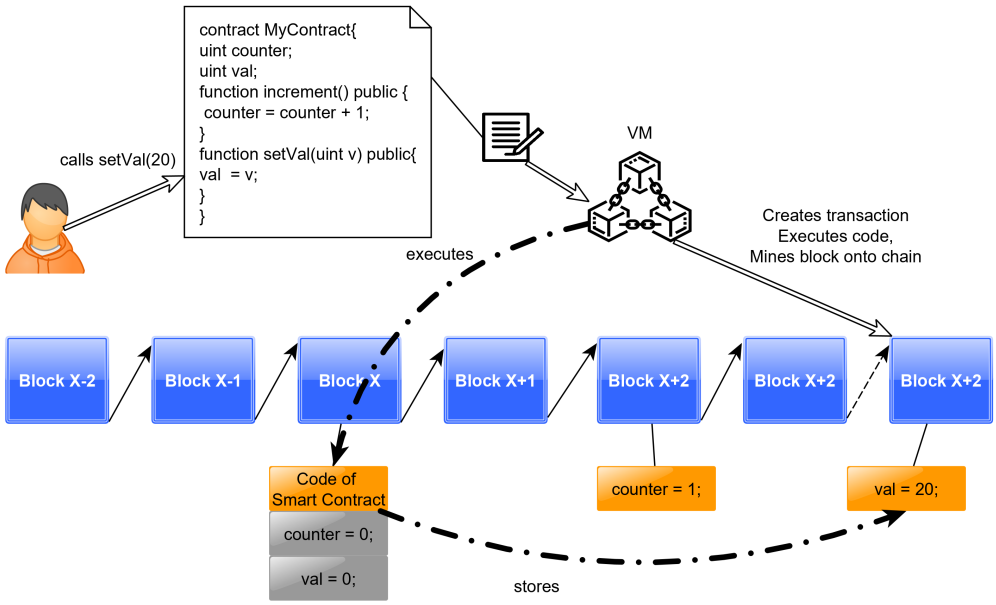

Figure 11: Another user calls a different function, at every point of time, orange data members indicate a valid state of the contract object

5 Introduction to the demo environment

The following tools are **required** in order to effectively participate in the tutorial

- Web browser supporting ECMAScript 2015
- SSH client (e.g. *putty* on windows, *openssh* on linux
- Arbitrary text editor for Solidity smart contract editing

The demo environment - as illustrated by Figure 12 - consists of a Web UI and back-end for managing the training sessions and allowing that the participants can easily interact with each other's deployed contracts.The back-end interacts with the Etherum based blockchain network using web3 API. The blockchain network consists only one node for now, but that doesn't greatly affects how the endpoints would interact with the blockchain network as long as connection is available to a node.

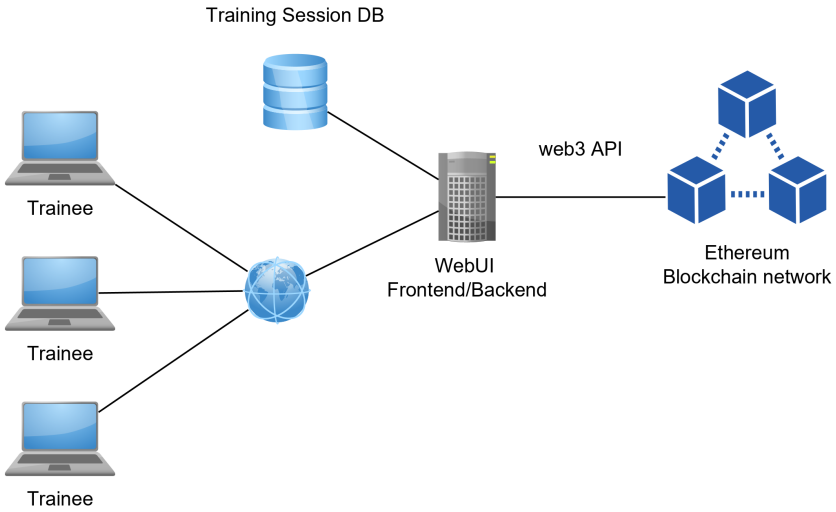

Figure 12: The tutorial environment

5.1 Login details

- Demo environment: `https://blockchain.cnsm2019-tutorial.com/`
  - username: `user<X>` , where is `<X>` is a number, e.g.: `user11`, the number allocation will happen at the time of starting the hands-on exercises
  - password: same as the username, e.g `user11`
- SSH access details:
  - host: blockchain.cnsm2019-tutorial.com
  - port: 2222
  - username: tutorial
  - password: cnsm2019

5.2 Attaching to *geth* node

- SSH onto host specified above
- Attach to *geth* blockchain node by issueing command `sudo ./admin/geth_attach`
- the *tutorial* user's password has to be given again

5.3 Most frequently used *geth* commands

- the *geth* console is a javascript REPL, any valid Javascript code is accepted. The commands below are actually *web3* API calls, for all available commands see [17]
- `<variable name> = <expression>` assigns the result of expression `<expression>` to the variable named `<variable>`, e.g. `block34 = eth.getBlock(34)` that means assign the block object of block 34 to the variable named `block34`
- to define any custom function define a Javascript function and assign it to a variable:

```
myFunction = function(param1,param2,...) {
 // body of the function
}

// call the function with arguments
myFunction(arg1,arg2,...)
```

- `eth.accounts` an array of all accounts managed by this node
- After startup create this function :

```
findAddress = function(addr){ return eth.accounts.filter(function(account){
    return new RegExp(addr+".*").test(account) }) }
```

  this can be used to find an account starting with `addr` e.g. `findAccount("0x10db7")`

- Note: scripts can be preloaded with the `--preload path/to/script.js` argument when attaching to geth if needed
- `eth.getBalance(<account>)` for checking the balance of an account, where `<account>` is a string literal having the value of the address of an Ethereum account or a deployed smart contract, e.g. `eth.getBalance("0x9afeD102A10D54Cc6C0E5153752c69B4876A7419")`
- `web3.toWei(<value>,<dimension>)` for getting the equivalent value in *Wei*s of the specified amount, where `<value>` is a number literal and `<dimension>` is a string literal having the name of a valid etherum metic, e.g. `web3.toWei(3.14,"ether")`
- `eth.getBlock(<block Number>)` for displaying the given block's data
- `eth.getTransaction(<transaction hash>)` for displaying the given transaction's data
- `eth.getGasPrice(console.log)` will return the median *gasPrice* based on the recently mined transaction or the default *gasPrice* if there hasn't been any yet
- `eth.getTransactionCount(<account>,[<block specifier>])`, get the total transaction count for acount indicated by `<account>`, optionally a block specifier can be supplied, e.g.: `eth.getTransactionCount("0x9afeD102A10D54Cc6C0E5153752c69B4876A7419","pending")` will get the transaction count for the given address including the pending transactions waiting in queue to be mined onto the blockchain
- `personal.signTransaction(<Transaction object>, <password>)` to sign a transaction with the senders private key represented by `<Transaction object>` object where the sender's account has `<password>` (string literal) as password, where a transaction has the following parameters
  - `from` : the sender's address
  - `to` : the receiver's address
  - `value` : the amount of *Wei*s to be transferred to the receiver from the sender with this transaction
  - `gas` : the amount of gas that can be consumed by the EVM for executing this transaction
  - `gasPrice` : the price to be paid by the sender per units of gas consumed, the maximum price of the transaction will be `value + gas*gasPrice`, if only fraction of the specified gas has been used by EVM, only the proportional price has to be paid while in contrast if the supplied gas was not enough the transaction will be reverted, but the gas price still has to be paid
  - `nonce` : this is different from the block nonce in case of Proof-of-Work. This one indicates the total transaction count of the user in order to prevent double spending
  - `data` : optional field, the additional data for this transaction, the code and data members are stored in this section of the transaction
  - Example:

```
signedTransaction = personal.signTransaction({
    from: "0x9afeD102A10D54Cc6C0E5153752c69B4876A7419",
    to: "0x3FecF304285303Fba1C34124889Ea1256e9BB0de",
    value: web3.toWei(1,"ether"),
    gas: 200000,
    gasPrice: 10,
    nonce: eth.getTransactionCount("0
        x9afeD102A10D54Cc6C0E5153752c69B4876A7419")
    },"user1" )
```

- `eth.sendRawTransaction(<Raw Transaction data>)` sends and execute the provided transaction where
  `<Raw Transaction data>` is the raw transaction data in a hexadecimal string format, using the signed transaction object the raw transaction can be accessed by the `raw` member variable, e.g.: `eth.sendRawTransaction(signedTransaction.raw)`

5.4 PowerBid game

5.4.1 Description of the contract

As part of familiarizing ourselves with the concepts of smart contracts and their potential use cases in IoT environments we are going to use a game. The game consists of a pre-implemented and simple smart contract that is modelling an electrical power auction. The game has 2 roles: supplier and consumer. The contract is created by the consumer where he specifies the required energy and the approximate time window where he intends to consume the required power. The creator - i.e. the consumer - of the contract also specifies a maximum price to pay for this amount of energy at that time window the he also has to pay upfront as a collateral to the deal. The suppliers essentially then monitor these contracts whether they can satisfy them while maximizing their profit then bid accordingly. This model is a sell auction i.e. a smaller price is considered more aggressive.

The contract has been described in Solidity language and it's latest source that is used during this tutorial can be studied here: https://raw.githubusercontent.com/fecjanky/iot_sc_tutorial/master/src/powerbid.sol.

The contract has the following main phases:

1. Auction Phase, where the suppliers participate in the sell auction
2. Consumption Phase, where the consumer consumes the power paid and requested
3. Withdraw Phase, where the consumer can withdraw his gain (that is the difference between the max. price and the best price) and the supplier can withdraw the funds from the contract

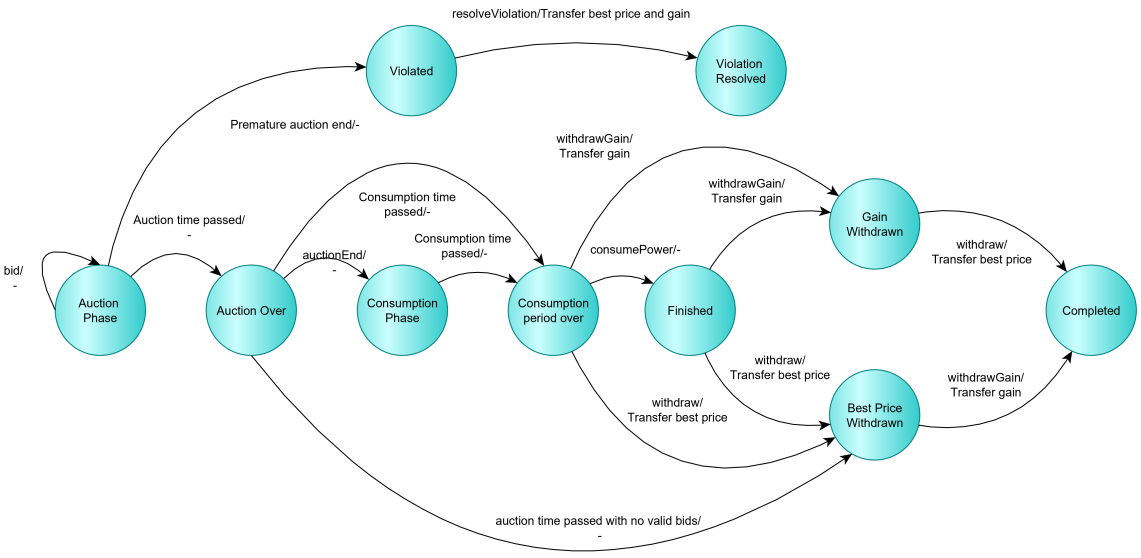

Figure 13: The state diagram of the PowerBid smart contract

5.4.2 Deployment, bidding and evaluation

The deployment of contract is done by supplying the required constructor parameters in the row where the *Deploy* button is present. The transaction options are sourced from the corresponding *TX Options* inputs and mapped to the transaction parameters as supplied in one of the previous tasks (see listing 5.3. The deployment is initiated by clicking the *Deploy* button. After successful deployment (see Figure 14) the new contract will appear under the deployed contracts list on the left hand side and will be selected as the active contract. In this case the transaction receipt will be logged in the bottom grey log area while in case of error the error message has to be inspected to see what went wrong with the deployment.

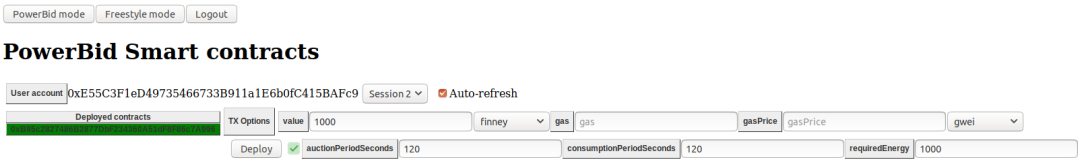

Figure 14: Deployment of a Powerbid contract

The *TX Options* are not mandatory in case of an arbitrary smart contract but as one inspects the constructor code (see listing 1) it can be seen that the message value must be greater than 0 *Wei* i.e. the creator of this smart contract must transfer some money upfront that is in line how the auction is modelled. The *gas* and *gasPrice* options are not mandatory, and in that case those are estimated automatically using the web3 API.

```
constructor(
    uint auctionPeriodSeconds,
    uint consumptionPeriodSeconds,
    uint requiredEnergy
) public payable {
    require(requiredEnergy > 0, "required energy must be greater than 0 wh");
    require(msg.value > 0 ,"max price must be greater than 0");
    require(auctionPeriodSeconds > 0, "auction period must be greater than 0");
    require(consumptionPeriodSeconds > 0, "consumption period must be greater than
        0");
    consumer = msg.sender;
    ...
}
```

Listing 1: Excerpt from the constructor of the Powerbid smart contract

As described before the *gas* determines the amount of operations that can be performed by the virtual machine while executing a give smart contract API call and the *gasPrice* times the used *gas* will be the reward of the individual who successfully places the transaction in a block onto the canonical chain. For these exercises the *gas* field can be left empty, however the *gasPrice* should be specified as if it is missing it will be estimated based on the last few blocks median gas price and depending on the exchange values it can happen easily that the transaction costs more than the gained value resulting in a net balance decrease.

It's also worth mentioning that in case of a `require` expression failure in the code when the transaction would be normally reverted the estimation will always fail with the same error message *'gas required exceeds allowance (8000000) or always failing transaction'*, while if no estimation is needed - i.e. all options are specified, then the EVM will try to create the transaction, but it will result in a reversion due to the failed assertion with error *'Transaction has been reverted by the EVM'* (see Figures 16 and 15). In both cases the contract's source code has to be studied for further troubleshooting. As of the time of this writing the `require` expression's string parameter is not propagated to the user, but that can change in the future editions of the tool-chain as a consequence it is good practice to give meaningful diagnostics there.

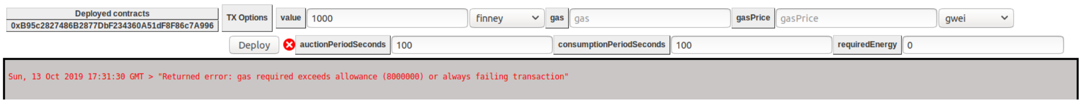

Figure 15: Invalid parameters with estimation

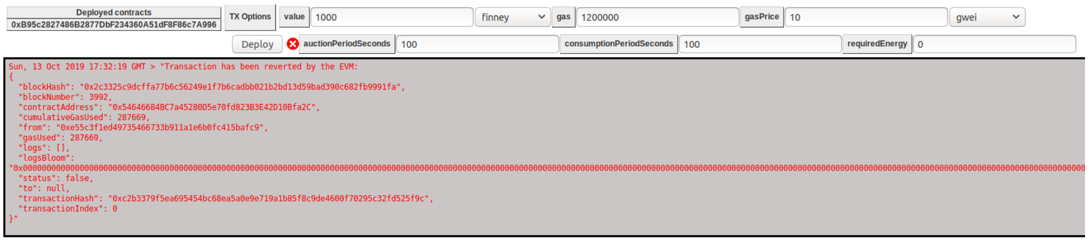

Figure 16: Invalid parameters without estimation

After successful deployment the contract will show up on the left hand side listing. By clicking on a deployed contract it's controls will become visible. The auction and consumption time left fields are updated automatically, but the other fields have to be queried manually by either calling the corresponding API function (exposed as the adjacent button) or by clicking on the phase getter, that will call all non-mutable API functions. The functions that mutate the contract are marked red. The suppliers can use the *bid* function to place their bids.

After the auction period has ended the consumption phase begins. The consumer can end the auction prematurely but in that case a different state transition sequence begins (see Figure 13).

After the consumption period has passed or the consumer explicitly marked the contract as consumed by clicking on the *consumePower* button the contract is open for withdrawals. The amounts can be withdrawn in any order driving the contract into the *Completed* state which is a terminal state.

5.4.3 Game-play summary

The high-level game-play for the two different roles can be summarized as:

- Consumer role:
  1. Deploy contract after specifying the constructor parameters
  2. Monitor the sell auction until the auction is ongoing.
  3. Consume the power (virtually, calling the *'consumePower'* API by pressing the corresponding button)
  4. withdraw gains (Max. price - best price)
  5. evaluate relative gain
- Producer/Supplier role:
  1. Bid on the deployed Smart contracts
  2. monitor bid state based on the visual feedback and/or polling the contracts
  3. place further bids as desired
  4. wait for consumer to consume the power
  5. withdraw the price amount from the contracts that has been win by the user
  6. evaluate relative gain

5.4.4 Typical values to use

- *gasPrice* : set to 10 Wei, otherwise the balance could drain quickly
- Consumer role (contract creation):
  - *value*: somewhere betwee 900-1000 finneys

  - *auctionPeriodSeconds*: 300 (seconds) so that will be enough time for bidding
  - *consumptionPeriodSeconds*: 60 (seconds) is enough
  - *requiredEnergy*: set it to 100000

- Supplier role:
  - when bidding don't go lower than 80% of the *maxPrice*

5.5 Implement a number guessing game

5.5.1 Specification

The game participants are guessing numbers. The winner is who choose the smallest number. If a number has been chosen by more than 1 person it is not considered as a valid guess. 1 person can only guess once. The creator offers a prize that (greater than 0 Wei) is to be won by the winner. If at the end of the game there are no valid guesses the creator is considered the winner. Only the winner must be able to withdraw the prize. At construction time the owner should be able to specify the total number of guesses that will limit the number of individuals who are able to play the game. The prize mustn't be possible to be withdrawn before the game ends. The results should only be possible to be queried through the programmatic API when the game is over.

The contract template can be downloaded from: https://raw.githubusercontent.com/fecjanky/iot_sc_tutorial/master/src/NumberGuessing_template.sol

One can use the *Remix IDE* for developing smart contracts that includes the solidity compiler and also a test network to deploy and interact with the contract. It is available under http://remix.ethereum.org/

5.5.2 Potential solution

Since Solidity is nearly a Turing-complete programming language, several higher level abstractions can be built therefore there is no single solution for the specification above. A reference solution can be found here:
https://raw.githubusercontent.com/fecjanky/iot_sc_tutorial/master/src/NumberGuessing.sol

A key element of this solution is to provide iterability over the built-in *mapping* type through a linear index to lookup key mapping. Furthermore, this sample solution illustrates the usage of *storage* and *memory* variable references.

Acknowledgements

This Tutorial has been initially compiled for the audience of the
15th International Conference on Network and Service Management – technically co-sponsored by IFIP, IEEE Communication Society, IEEE Computer Society, and ACM.

In connection to this work, the authors have received funding from the EU ECSEL JU under the H2020 Framework Programme, JU grant nr. 737459 (Productive4.0 project, www.productive40.eu). Part of this work was supported by the Higher Education Excellence Program of the Ministry of Human Capacities, Hungary, in the frame of Artificial Intelligence research area of Budapest University of Technology and Economics (BME FIKP-MI/SC).

References

[1] H.-N. Dai, Z. Zheng, and Y. Zhang, "Blockchain for Internet of Things: A Survey," *IEEE Internet of Things Journal*, vol. 6, no. 5, pp. 8076–8094, 2019.

[2] M. S. Ali, M. Vecchio, M. Pincheira, K. Dolui, F. Antonelli, and M. H. Rehmani, "Applications of Blockchains in the Internet of Things: A Comprehensive Survey," *IEEE Communications Surveys and Tutorials*, vol. 21, no. 2, pp. 1676–1717, 2018.

[3] P. Varga and S. Plosz and G. Soos and Cs. Hegedus, "Security Threats and Issues in Automation IoT," in *IEEE International Workshop on Factory Communication Systems - a Conference (WFCS 2017)*, (Trondheim, Norway), 2017.

[4] P. Varga, F. Blomstedt, L. L. Ferreira, J. Eliasson, M. Johansson, J. Delsing, , and I. M. de Soria, "Making System of Systems Interoperable - the Core Components of the Arrowhead Technology Framework," *Journal of Network and Computer Applications*, vol. 80, pp. 85–95, 2017.

[5] Cs. Hegedus and P. Varga and A. Franko, "Secure and Trusted Inter-Cloud Communications in the Arrowhead Framework," in *1st IEEE International Conference on Industrial Cyber-Physical Systems (ICPS)*, (St.Petersburg, Russia), 2018.

[6] TechTerms.com, "Blockchain." https://techterms.com/definition/blockchain, 2019.

[7] R. Maull, P. Godsiff, C. Mulligan, A. Brown, and B. Kewell, "Distributed ledger technology: Applications and implications," *Briefings in Entrepreneurial Finance*, vol. 26, no. 5, pp. 481–489, 2017.

[8] Wikipedia, "Cryptographic hash function." https://en.wikipedia.org/wiki/Cryptographic_hash_function, 2019.

[9] Steve Friedl, "Unixwiz.net tech tips: An illustrated guide to cryptographic hashes." http://www.unixwiz.net/techtips/iguide-crypto-hashes.html, 2005.

[10] Tutorialspoint, "Blockchain - bitcoin - mining." https://techterms.com/definition/blockchain, 2019.

[11] Bitcoin Wiki, "Mining." https://en.bitcoin.it/wiki/Mining, 2019.

[12] L. M. Bach and B. Mihaljevic and M. Zagar, "Comparative analysis of blockchain consensus algorithms," in *41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, (Opatija, Croatia), 2018.

[13] IOTA Ecosystem, "Iota tutorial 18 - merkle tree." https://ecosystem.iota.org/tutorials/iota-tutorial-18, 2018.

[14] Wikipedia, "Smart contract." https://en.wikipedia.org/wiki/Smart_contract, 2019.

[15] Solidity, "Solidity documentation." https://solidity.readthedocs.io/en/v0.5.12/, 2019.

[16] Vyper, "Vyper documentation." https://vyper.readthedocs.io/en/latest/installing-vyper.html, 2019.

[17] web3.js, "web3.js API documentation." https://web3js.readthedocs.io/en/v1.2.1/getting-started.html, 2019.