Document downloaded from:

http://hdl.handle.net/10251/189085

This paper must be cited as:

Liu, X.; Muhammad, K.; Lloret, J.; Chen, Y.; Yuan, S. (2019). Elastic and cost-effective data carrier architecture for smart contract in blockchain. Future Generation Computer Systems. 100:590-599. https://doi.org/10.1016/j.future.2019.05.042

The final publication is available at

https://doi.org/10.1016/j.future.2019.05.042

Copyright Elsevier

Additional Information

Elastic and Cost-effective Data Carrier Architecture for Smart Contract in Blockchain

Xiaolong Liu[1], Khan Muhammad[2], Jaime Lloret[3*], Yu-Wen Chen[4], and Shyan-Ming Yuan[4*]

[1] College of Computer and Information Sciences, Fujian Agriculture and Forestry University, Fuzhou 350002, China; xlliu@fafu.edu.cn

[2] Department of software, Sejong University, Seoul 143-747, Republic of Korea; khan.muhammad@ieee.org

[3] Integrated Management Coastal Research Institute, Universitat Politècnica de València, C/ Paranimf nº 1, Grao de Gandía—Gandía, 46730 Valencia, Spain; jlloret@dcom.upv.es

[4] Department of Computer Science, National Chiao Tung University, Hsinchu 300, Taiwan; w369gf523@gmail.com, smyuan@cs.nctu.edu.tw

* Correspondence: Jaime Lloret (jlloret@dcom.upv.es) and Shyan-Ming Yuan (smyuan@cs.nctu.edu.tw)

Abstract—Smart contract, which could help developer deploy decentralized and secure blockchain application, is one of the most promising technologies for modern Internet of things (IoT) ecosystem today. However, Ethereum smart contract lacks of ability to communicate with outside IoT environment. To enable smart contracts to fetch off-chain data, this paper proposes a data carrier architecture that is cost-effective and elastic for blockchain-enabled IoT environment. Three components, namely Mission Manager, Task Publisher and Worker, are presented in the data carrier architecture to interact with contract developer, smart contract, Ethereum node and off-chain data sources. Selective solutions are also proposed for filtering smart contract event and decoding event log to fit different requirements. The evaluation results and discussions show the proposed system will decrease about 20USD deployment cost in average for every smart contract, and it's more efficient and elastic compared with Oraclize Oracle data carrier service.

Keywords—Blockchain, blockchain-enabled IoT, smart contract, Ethereum, off-chain data, data carrier

1. Introduction

In the day-to-day workings of information-oriented society, significant development of industrial systems has been witnessed with the convergence from wireless networks, Internet of Things (IoT) to blockchain [1, 2]. IoT is a significant component of industrial systems, which has recently attracted the interest of stakeholders [3, 4]. Meanwhile, blockchain is termed as one of the most promising technologies for IoT applications today, since not being able to modify past transactions and absence of a trusted intermediary make blockchain solution highly trustworthy [5, 6]. Several projects have examined the positive benefit of blockchain-enabled IoT applications, such as digital asset registries, peer-to-peer (P2P) energy trading, and long-tail personalized economic services [7, 8]. The most representative application of blockchain was Bitcoin proposed by Satoshi Nakamoto in 2008 [9], which is a peer-to-peer electronic cash system and a distributed ledger. It eliminates the need for trusted third party for e-commerce payment system. In 2013, blockchain developers came up with the second-generation blockchain implementation, Ethereum [10], which contains more features than Bitcoin. It provides not only a distributed ledger system but also the implementation of smart contract [11].

Ethereum smart contract is the programmable application that manages exchanges conducted online within Ethereum environment. Intelligence is built directly into the smart contract through a protocol that automatically identifies, validates, confirms, and routes transactions within the network. It allows proper, distributed, heavily automated workflows and brings more certainty and reliability to industrial systems. Recently, a great diversity of smart contract based applications have been presented, including applications for IoT, cloud computing, e-commerce and financial [12] Some researches utilize smart contracts to build access control system to overcome security and privacy issues in IoT environment [13, 14]. In terms of cloud computing, the smart contract applications could address the issues of resource management of cloud datacenters, verifiability of outsourced computation, service level agreement monitoring, negotiation and agreement establishment [15]. For e-commerce, Smart contract plays important roles in the legal implications of exchanges conducted on the blockchain [16].

Although Ethereum smart contract has now been serving an important function in the automation of transactions and multi-step processes, nowadays it lives like in a walled garden. Smart contract cannot directly communicate with external environment and fetch off-chain data, such as fed data for assets and energy trading applications in external IoT system. Because smart contracts are executed within the Ethereum Virtual Machine, whereas Ethereum Virtual Machine cannot communicate with the external systems. Every transaction processed by different Ethereum Virtual Machine spreading in same blockchain should be the same result, while fetching off-chain data are not determined, neither generating random numbers. This feature highly limits the developing of decentralized applications in Ethereum environment [17]. Practically, smart contract developers must setup an agent (i.e. data carrier) to get data off-chain, and call the contract function to pass data back to the contract.

Oracle is one of the most general solutions for the limitation mentioned above [18]. It is a new data carrier functionality that provides the connectivity of smart contract to the outside world. The idea is to fetch off-chain data that provided from more than one data source, and then execute the data-dependent action if the same answer is provided. Recently, several implementations of Oracle have been developed. The main solution is to provide an Oracle contract on the blockchain that serve off-chain data requests by other smart contracts. However, this solution requires a predefined standard on data format of smart contracts. Most important, the problem of Oracle is that it increases deployment costs for smart contract. While deploying smart contract with Oracle as data carrier, original smart contract needs to inherit extra smart contract, namely Oracle resolver, for interacting with Oracle. The inherited contract increases the deployment costs, because Ethereum charge fees when deploying smart contract is according to how much space smart contract takes, namely, the longer bytecode contract takes the higher fee Ethereum charges. Thus, if developer want to develop a service which do not depend on single contract, it takes different contract instance to service different end-user. In addition, due to the adoption of a standard interface of Oracle, the readability of smart contracts is reduced, and Oracle is not compatible with smart contracts that do not use Oracle at deployment. Therefore, this paper would propose a cost-effective data carrier for Ethereum based smart contract to solve the problems mentioned above.

The objective of this paper is to propose an elastic and cost-effective data carrier architecture for Ethereum smart contracts that minimize contract deployment costs, and monitor contract event without subscribing any filter at Ethereum node. The proposed architecture consists of three components: Mission Manager, Task Publisher and Worker. It is responsible for the interactions of contract developer register, monitor smart contract, Ethereum node callback and fetch of external data source and computation source. We also proposed selective solutions for filtering smart contract event, and decoding event log to fit different requirements. The comparison result with Oraclize Oracle service in terms of deployment cost is also presented to show the superiority of the proposed data carrier system. The main contributions of our proposed data carrier system are the following:

- Reliable: the security model is maintained in this system, users of decentralized blockchain-enabled IoT applications do not have to trust a third party.
- Elastic: the proposed data carrier system does not require a predefined standard on data format. It is not necessary for data providers to modify their services to be compatible with Ethereum protocols.
- Cost-effective: in this system, the original smart contract does not need to inherit extra smart contract for interacting with external IoT data source, which will efficiently decrease the deployment cost of every smart contract that need off-chain data carrier service.

The remainder of this paper is organized as follows. In Section 2, we present the overview of blockchain and the analysis of related Oracle works with its limitation. The design and implementation of the proposed data carrier system is described in Section 3. Section 4 evaluates and discusses the superiority of the proposed system compared with Oraclize Oracle service. Finally, Section 5 draws conclusion and future work.

2. Related Work

2.1 Blockchain

The blockchain was first stated in the digital cryptocurrency, but its effect is being observed to be far wider than just the alternative money. Originally block chain is distributed digital transaction ledger, which is a type of database shared and synchronized among distributed network. The most representative application of blockchain was a peer-to-peer digital cash system, Bitcoin, proposed by Satoshi Nakamoto. Its effect is being witnessed to be far away than just alternative money. Nowadays, blockchain has been termed as one of the most promising technologies for business and IoT applications today [19, 20]. The blocks in blockchain record transactions among participants in peer-to-peer network, such as transaction of asset and energy trading in IoT. The key idea behind blockchain is that every block in the blockchain has a timestamp and unique cryptographic signature. Every block refers to the signature of its previous block in the chain. Therefore, all blocks can be traced, which guarantees an auditable, immutable history of all transactions in the blockchain [21]. Most importantly, there is no centralized authority or third-party is involved in the blockchain. Participants in the network conduct and agree by consensus on the updates of blocks in chain. All the confirmed and validated transaction blocks are linked and chained from the beginning of the chain to the most current block.

Ethereum is a second-generation blockchain implementation, which provides not only a distributed ledger system but also the implementation of smart contract. The purpose of Ethereum is to create an alternative protocol for building decentralized applications leverage on blockchain. The main difference between Bitcoin and Ethereum is: For the first generation distributed ledger likes Bitcoin, confirming an unconfirmed transaction only means documenting the state of digital currency transfer between two addresses; Whereas Ethereum extended the ability of transaction by adding capability of computation to blockchain [22]. It regards reaching consensus for state of program as reaching consensus for transfer. The feature of Ethereum can deal with reaching consensus for decentralized computations. To better understand the work in this paper, the following basic and foundational concepts of Ethereum should be clarified:

1) Smart contract: Smart contract enabled by Ethereum blockchain technology is a contract implemented, deployed and executed within EVM. It is a set of commitments that are defined in digital form, including the agreement on how contract participants shall fulfill these commitments. Smart contract can be regard as programmable application which consisting of functions that manage exchanges conducted online. User can create an instance of the contract and invoke functions to view and update contract data along with execution of some logic.
2) EVM: EVM is the virtual machine and runtime environment for executing code written in Ethereum smart contracts. It is the fundamental consensus mechanism for Ethereum. It is sandboxed and completely isolated from the network, file system or other processes of the host computer system. EVM implementation is run on Ethereum node in the network and executes the same instructions.
3) Ether: Ether is the currency of Ethereum. Miners of Ethereum who are successful in generating and creating a block in the chain are rewarded by Ether. It is also the medium used by Ethereum to pay transaction fees and computing service fees. Ether can be traded on the foreign exchange market easily by converting to dollars or other traditional currencies through Crypto-exchanges.

4) Gas: Gas is the internal currency of Ethereum that measurement roughly equivalent to computational operation. It determines the normal operation of the Ethereum network ecosystem. Every operation in Ethereum has Gas expenditure, and the execution cost is predetermined in terms of Gas units. Gas consists of two parts: Gas Limit and Gas Price. Gas Limit is the maximum amount of Gas that the user is willing to pay to perform an action or confirm a transaction. Gas Price is the number of Gweis that users are willing to spend on each Gas unit.

2.2 Oracle

Although integrating blockchain into the IoT is relatively recent, several proposals have already been presented to improve current IoT technology [23-25], where Ethereum is shown as the most popular platform for IoT–blockchain applications. In particular, smart contracts are presented to revolutionize many industries by replacing the need for both traditional legal agreements and centrally automated digital agreements [26]. Smart contracts in blockchain-enable environment will inevitably require high-assurance versions of off-chain data, such as smart contracts that require access to APIs reporting market prices, and need data feeds about IoT data related to energy trading. Unfortunately, Ethereum smart contracts cannot directly fetch off-chain data with the outside world, since they are executed within EVM with underlying consensus protocols. Therefore, smart contracts that with functions of random numbers, decentralized exchanges, and external information, required Oracle data carrier functionality to connect outside world [27]. Fig. 1 shows the conceptual architecture of Oracle. The concept of it is to enable smart contract to fetch off-chain data through Oracle external agent. The main solution is to provide an Oracle contract on the blockchain, which serves off-chain data requested by user smart contracts. While deploying smart contract with Oracle, original user smart contract needs to inherit extra smart contract, namely Oracle resolver, with a predefined standard on data format. The Oracle resolver is responsible for interacting with Oracle contract, which is designed to present a simple API to a relying user contract for its requests to external data source. As shown in Fig.1, Oracle contract accepts query datagram from Oracle resolver and generates event log to external agent for fetching off-chain data. At the end, external agent will launch a callback and return corresponding data for user contract.

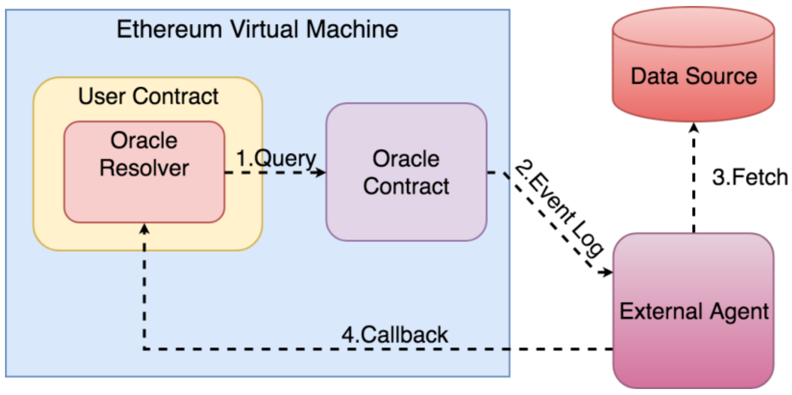

Fig. 1. The conceptual architecture of Oracle

In reality, Oracle has various ways to implement. In 2014, Ripple Labs [28] published a white paper of Smart Oracles and implemented a system of smart oracles, called Codius [29], in which rules can be written in any programming language. Codius enables smart contracts to interact with any service that accepts cryptographically signed commands. Later, Ellis et al. proposed a decentralized Oracle network named ChainLink that provides for contracts to gain external connectivity [30]. They presented both a simple on-chain contract data aggregation system, and a more efficient off-chain consensus mechanism. ChainLink can securely push data to APIs and various legacy systems on behalf of a smart contract. Recently, the leading Oracle service for smart contracts and blockchain applications is Oraclize [18], which serves thousands of requests every day on Ethereum platforms. Oraclize provides part of the infrastructure needed to build smart and useful decentralized applications, and its service guarantee the correctness of data.

Generally, the first benefit of Oracle is that if users have multiple contracts that need external data, traditionally, they should program responder and launch one responder for each smart contract. But if users take the architecture of Oracle, the only event emitted by contract that needs off-chain data would be Oracle contract, which makes Oracle become the agent of all contracts that needs off-chain data. The second benefit is that Oracle does not need to manage contract's application binary interface. In general, anyone wants to interact with specific contract, two elements will be required, i.e. contract address and application binary interface. However, Oracle users do not need to provide any application binary interface for Oracle provider. Because the Oracle data carrier system, such as Oraclize, contains a virtual function used for callback, user needs to inherit standard callback function to receive external data.

However, the feature that Oracle does not need application binary interface is a double-edged sword. Its shortcoming is everyone can easily decode transaction event, even trigger the callback function when contract programmer does not limit the message sender of callback function. Appropriately, although the purpose of application binary interface does not encrypt the transaction, it still increases the risk of smart contract [31]. Moreover, the original smart contract needs to inherit extra smart contract of Oracle for interacting with external data source. The inherited contract increases the original smart contract content. Since the more content contract takes the higher fee Ethereum charges, the deployment costs will be increased while deploying Ethereum smart contract with Oracle as data carrier. To solve the problems mentioned above, in this paper, we will propose an elastic and cost-effective data carrier for smart contract to interact with the outside data source.

3 System Design and Implementation

This section will introduce the propose data carrier architecture for smart contract in blockchain-enabled IoT environment to interact with the outside data sources. The overview of the proposed architecture and details of each component will be illustrated accordingly.

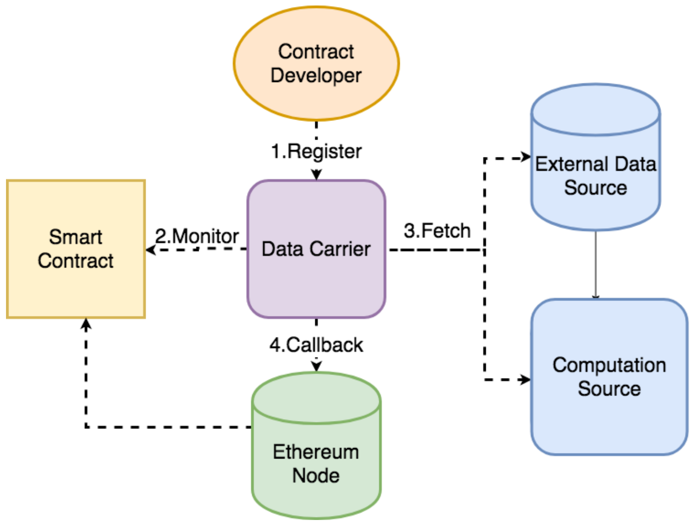

Fig. 2. The interactions of the proposed data carrier

Fig. 2 shows the interactions of the proposed data carrier system. At the very beginning, it is responsible for the register of original smart contract developer, including constructing mission and registering mission. After that, the data carrier system would monitor the corresponding Ethereum smart contract that needs external off-chain data. Once being activated by any transaction, the managed smart contract will fetch off-chain data from external environment and callback the fetched results to smart contract through Ethereum node. Generally, there are two kinds of off-chain data that need to be handled in the data carrier service for smart contract. The first one is the general data that provided by external data source, and the other one is the results that computed in computation source.

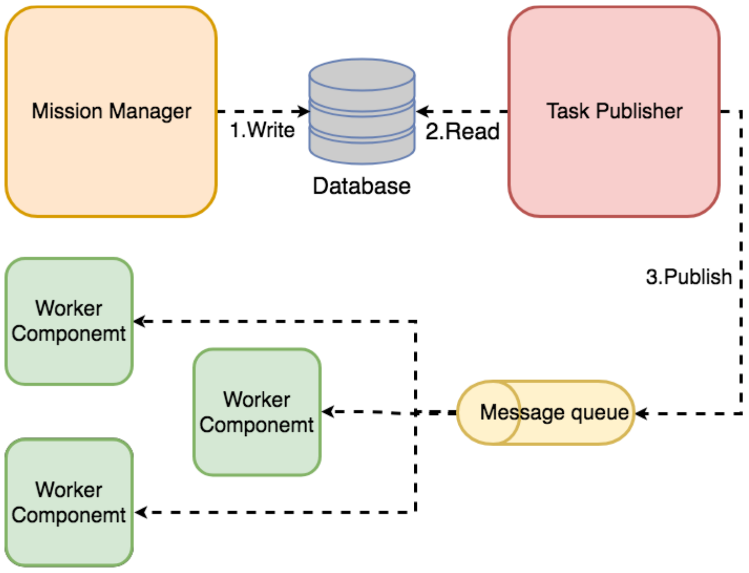

Fig. 3. The architecture of the proposed data carrier

Fig. 3 shows the conceptual architecture of the proposed data carrier system. Basically, it contains three components: Mission Manager, Task Publisher and Worker. Mission Manager is used to receive mission registered by system user. A mission contains event hash, contract address, ways to respond event, and the queue topic response for event. Task Publisher will perform four phases action for each block pended, including, collect transactions on new block, filter out unconcerned transaction, fetch argument in event, and send generated task to specific Worker. Worker will retrieve data according to the task, encode data with application binary interface, and make function call transaction as event's callback. While using the proposed system, users need to do only two things. The first thing is to register in the system, which is part of the Mission Manager. The second thing is to build Worker themselves if they are not using the features provided by the system maintainer, and connected this Worker to Task Publisher.

3.1 Mission Manager

Mission Manager consists of front-end and back-end. The front-end is mainly responsible for constructing mission and registering mission to back-end. The data source information collected by frontend of Mission Manager is mission requisition template (MRT), which is described in Table 1. Front-end will transform MRT into mission and send it to back-end via http post to register mission.

Table 1
The standard format of mission requisition template (MRT)

Key	**Format**	**Description**
contractAddress	String	The contract address which is monitored
eventName	String	Target event's name
contract_interface	JSON Array	Contract application binary interface
command	String	Command template, executing command after filling in parameter part.
callbackFunctionName	String	Name of function which is used to receive external data or computation result.
messageQueueChannel	String	Used to identify worker connection port.

To build service back-end, the Express web framework [32] is used. It is designed for building web applications and APIs, and hosted within the node.js runtime environment. We use Express to set up a RESTful API for users to register mission, and store it in MongoDB. The standard format of mission that is sent from front-end is described in Table 2. The stored mission provides the necessary information for monitoring Ethereum blockchain, how to send external data back to the smart contract, and how does worker retrieve the external data.

Table 2
The standard format of mission

Key	Format	Description
contractAddress	String	The contract address which is monitored
eventHash	String	Hash of the event name and parameter type.
eventInterface	JSON Array	A part of contract application binary interface, used to decode event.
command	String	Command template, executing command after filling in parameter part.
callbackInterface	JSON Array	A part of contract application binary interface, used to encode transaction data.
messageQueueChannel	String	Used to identify worker connection port.

3.2 Task Publisher

Task Publisher will perform four phases action for each block pended, including collect transactions on Ethereum node, filter out unconcerned transaction, fetch argument in event, and send generated task to specific Worker. Fig. 4 shows the general architecture of task publisher, which is mainly consist of filter module, decoding module and publishing module. The transaction information is retrieved from Ethereum node, which can be practically parsed from public Ethereum block explorer website directly, or retrieve data via website provided APIs. The Task Publisher is implement by Node.js with the characteristic of event-driven and non-blocking I/O model. The following subsections will illustrate each module of Task Publisher in details.

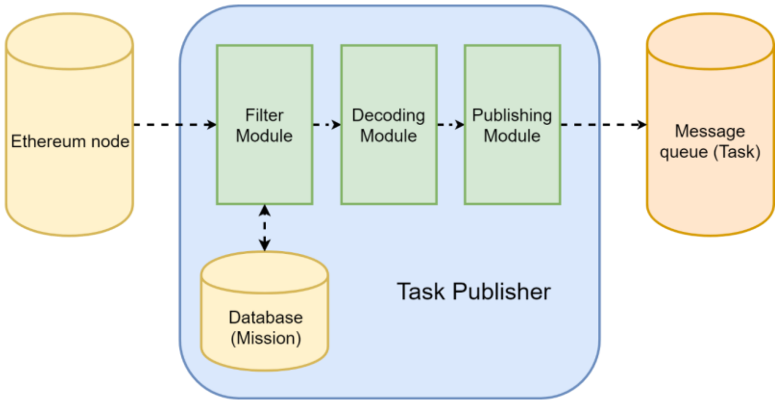

Fig. 4. The architecture of Task Publisher

3.2.1 Filter module

Filter module is triggered when every new block header comes in to find out whether managed Ethereum smart contract is activated by any address. In order to know when can we check new block on Ethereum blockchain, it subscribes Ethereum node with web3 package. Block header information retrieved from Ethereum node can be used as timer to check for changes on the blockchain. After retrieving block header, we could know the number of newest block pended to blockchain. After that we could get the detailed block information by web3 method “web3.eth.getBlock”. The block information contains transaction hash, which is the hash of the signed transaction object. It is unique for each transaction, and by which Ethereum user can trace their transaction. With this transaction hash, the “target address” of the transaction will be retrieved from Ethereum node. The target address is the address of the transaction receiver, if the transaction is used for trigger smart contract function, the target address will be the smart contract address.

After that, we can already know which smart contract in Ethereum is triggered, and could know whether the smart contract hosted by our system has been triggered or not by checking Key “contractAddress” in the mission database. Since each transaction can trigger multiple events in smart contract, we can further use the key “eventHash” store in mission database to filter out the event we are responsible for.

3.2.2 Decoding module

The goal of decoding module is to decode the arguments in the filtered event from filter module, and generate task for Worker. Since we don't qualify the user's event record arguments sequence and type, we need the event interface to perform decode event. Fig. 5 shows the workflow of task generating. At the very beginning decoding module will obtains the filtered event from filter module and the mission from the database. In mission, we can additionally know the message queue channel that is responsible for the event. The key “eventInterface” of mission, which is part of contract application binary interface, will be used to decode the event’s log arguments. After that, the decoded content will replace the “command” of mission into an actual comment of task.

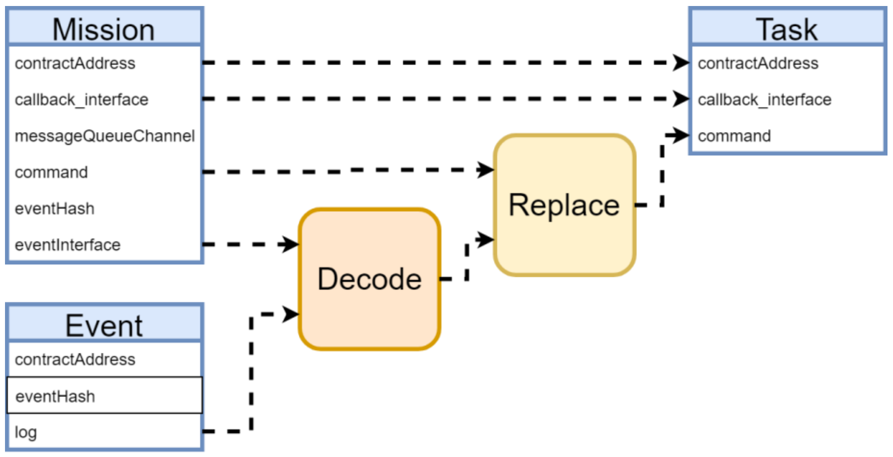

Fig. 5. The workflow of Task Generating

Fig. 6 shows an example of how decoding module replace command template. In this example user supposes to generate random numbers range from 10 to 100 for the smart contract. As shown in Fig. 6, the command contains a python command template with variables "$lowerBound" and "$upperBound". The decoded log contains the log arguments decoded by the decoding module, where the values of lowerBound is 10 and upperBound is 100. During command replacement, log arguments 10 and 100 will be used to replace the variables "$lowerBound" and "$upperBound" in the python command template, respectively. Therefore, after replacing the variables in command by log arguments, decoding module will get the actual command, which will be "command: python random.py 10 100" in this example. The final task will be generated by combining the "replaced command" with "contractAddress" and "callback_interface" of mission.

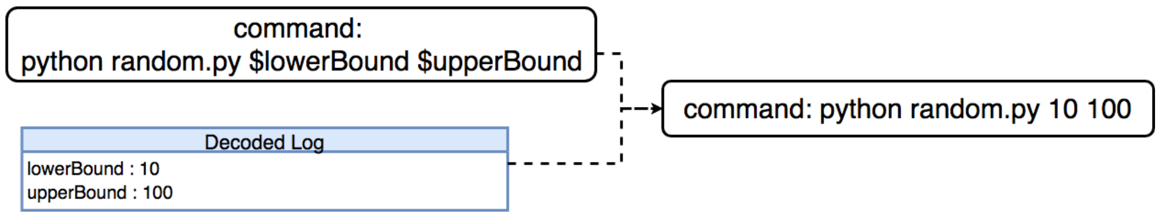
command:
python random.py $lowerBound $upperBound
Decoded Log
lowerBound : 10
upperBound : 100
command: python random.py 10 100

Fig. 6. An example of command replacement

3.2.3 Publishing module

After replacing command and generating task, we should push the task to the message queue according to “messageQueueChannel” of mission in the database. In practice, rabbitMQ [33] is used to implement through the Rabbit.js package, which provides a simple, socket-oriented API for messaging in Node.JS. The message queue mode we use is PUSH/WORKER mode. WORKER socket will receive a share of the messages, and require calling of acknowledgement function to acknowledge that each message has been processed. Any messages left unacknowledged when the socket closes, or crashes, will be requeued and delivered to another connected socket. A WORKER socket is read-only, and has the additional method which acknowledges the oldest unacknowledged message, and must be called once only for each message.

Since there may be an error task at the worker side, it requires to ensure that each task will be processed correctly. Worker cannot immediately acknowledge the queue after obtaining the data. If the worker obtains the task that cannot be processed, it will be corrupted before worker informs the system that the task cannot be executed. In addition, on both publisher and worker side, we set option “persistent” in RabbitMQ to be true. The option “persistent” could govern the lifetime of messages, and setting it to be true means RabbitMQ will keep messages over restarts by writing them to disk. This is an option for all sockets, and crucially, sockets connected to the same address must agree on persistence. The “persistent” feature ensures that even if the system crashes, tasks that have not yet been executed by the Worker will not be lost.

3.3 Worker

Worker is responsible for retrieving data according to the task sent by Task Publisher, encoding data with application binary interface, and making function call transaction as event’s callback. Fig. 7 shows the general architecture of Worker, which consists of execution module, transaction module, fetching agent that could fetch external data, and computing agent that provide external computation.

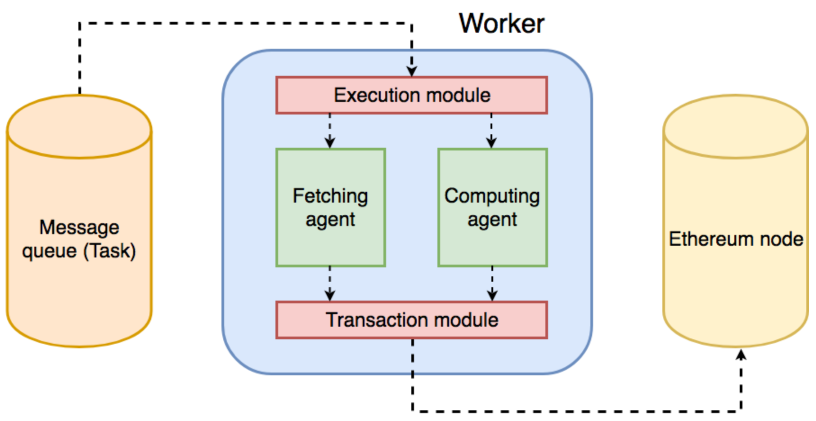

Fig. 7. The architecture of Worker

The Execution module would execute receiving “command” from task in message queue to obtain data. It uses the child_process package of Node.js to generate an external execution program. This program can be fetching agent or computing agent, the working scenarios are shown in Fig. 8. The execution can be processed as external computation or simply fetch data from external data source, which makes Worker highly flexible. After execution, both working scenarios require to output the parameters of smart contract to standard output as a JSON array. The worker will obtain this output for the transaction module as callback parameter.

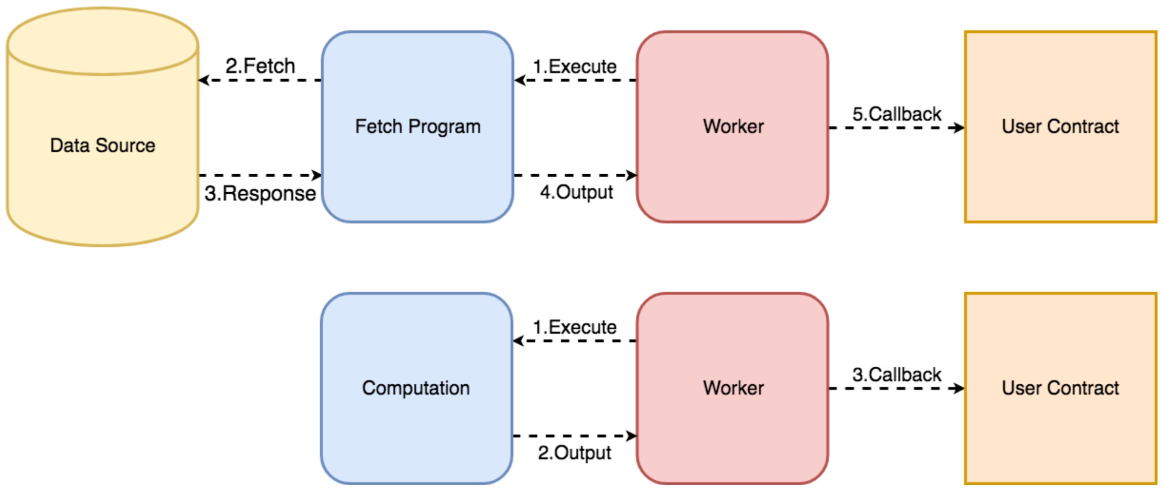

Fig. 8. The working scenarios of Worker

The transaction module is responsible for passing the results generated by the execution module back to the smart contract via function calls. The functions responsible for receiving external data are data callback functions. To interact with smart contract in Ethereum node, we only need to manage the parameters to be transmitted and use the callback function interface to encode transaction. Therefore, with help of the proposed data carrier system, external data and computation source can be efficiently fetched for smart contract to interact with the outside world.

4 Evaluation Results and Discussions

4.1 Evaluation results

The comparison results of the proposed data carrier system with Oraclize Oracle service are presented in this section. The main difference between the data carrier architecture proposed in this paper and the Oracle system is the deployment costs of smart contracts. Fig. 9 shows the default deployment scenario of the proposed system. While deploying smart contract with the proposed data carrier system, the contract developers only require to compose their original smart contract and assign corresponding mission. The proposed system will automatically response transaction events, and no extra smart contract deployment is required. On the contrary, while deploying smart contract with Oracle as data carrier, original smart contract needs to inherit extra smart contract for interacting with Oracle. The inherited contract increases the deployment costs, because Ethereum charge fees when deploying smart contract is depended on how much space smart contract takes, namely, the longer bytecode contract takes the higher fee Ethereum charges.

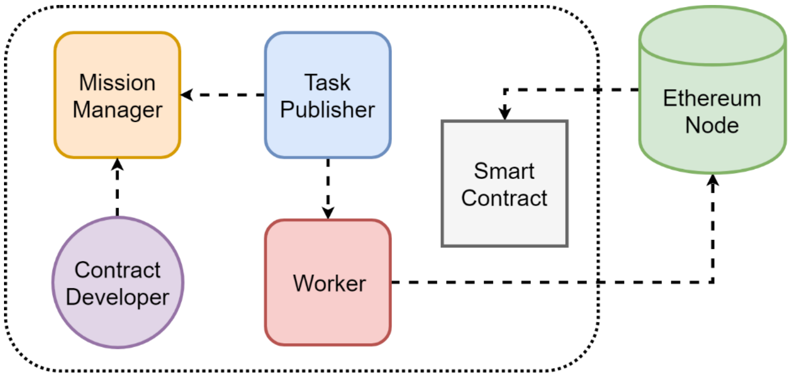

Fig. 9. Default deployment scenario of the proposed data carrier system

The evaluation is presented to demonstrate that the proposed system can accurately decrease deployment costs of smart contracts compared with Oracle. The example smart contract used in the evaluation is KrakenPriceTicker.sol [34], which is a smart contract that fetch Bitcoin price at Kraken digital asset trading platform. To KrakenPriceTicker smart contract to fetch external data in Oracle service, user should deploy the Oracle contract at the same time.

Table 3
Deployment Costs of KrakenPriceTicker and Oracle contracts

Optimization	KrakenPriceTicker (Gas)	Oracle (Gas)	Total (Gas)
No	433,800	2,563,800	2,997,600
Yes	393,000	1,719,200	2,112,200

Table 3 shows the deployment costs of krakenPriceTicker and Oracle contracts, where optimization refers to whether the smart contract functions are optimized by the smart contract developer or not. The deployment of original krakenPriceTicker smart contract costs 433,800Gas before optimization and 393,000Gas after optimization. On the other hand, Oracle contract costs about 2,563,800Gas before optimization, and the deployment cost is 1,719,200Gas even if the optimization is conducted. It indicates that the deployment of krakenPriceTicker smart contract in Oracle would cost about 2,112,200Gas after optimization. As shown in Table 3, the deployment cost of Oracle contract may even be several times higher than the original krakenPriceTicker smart contract. This is because Oracle provides a lot of additional functions that are redundant for users, which results in a very large storage consume and deployment cost during deploying smart contract in Oracle. Table 4 shows the Ethereum fee schedule during deployment. According to Table 4, we can find that the main cost of smart contract deployment is charge for placing code in smart contract creation and carried data size in transaction. Therefore, the more data carried by smart contract and transaction, the higher the fee Ethereum charged.

Table 4
Ethereum fee schedule during smart contract deployment.

Name	Value (Gas)	Description
Code deposit	32000	Paid for a CREATE operation
Create	200	Paid per byte for a CREATE operation to place code into state
Transaction	21000	Paid for every transaction.
Txdatazero	4	Paid for every zero byte of data or code for a transaction.
Txdatanonzero	68	Paid for every non-zero byte of data or code for a transaction.

In this evaluation, we intent to calculate how much cost can be saved by our data carrier system compared with Oracle for every smart contract. Basically, the difference between our data carrier system and Oracle is that additional Oracle contract should be inherited and deployed in Oracle environment. Therefore, at lease 1,719,200Gas can be saved in our data carrier system according to Table 3. The equation for calculating the actual cost of deploying a smart contract list is defined as follows:

$$\text{Deployment Cost} = \text{Gas Used} * \text{Gas Price} * \text{Ether Price}, \qquad (1)$$

where Gas Used is the total Gas used to deploy the smart contract, which is decided after compiling the smart contract (more specifically, it was decided during the deployment). When deploying smart contracts, the cost mainly comes from the size of the original data, the space occupied by the smart contract after deployment, and the constructor operating costs. Gas price refers to the amount of Ether users are willing to pay for every unit of Gas, and is usually measured in "Gwei", which equals to 10^{-9} Ether. Ether price is the actual exchange rate of Ether to dollar in real world.

To evaluate the actual deployment cost, we list the actual price of Ethereum Gas and Ether from February to May 2018 [35, 36], which are shown in Fig. 10 and Fig. 11, respectively. After calculation, the results in Table 5 show that the average Gas and Ether Prices are 17.15973Gwei and 668.7079USD, respectively. Since Oracle contract takes about 1,719,200Gas as shown in Table 3, the actual cost of deploying Oracle contract is 1,719,200Gas * 17.15973Gwei* 668.7079USD≈ 19.72USD. It indicates that 20USD deployment cost can be saved for averagely by our data carrier system compared with Oracle.

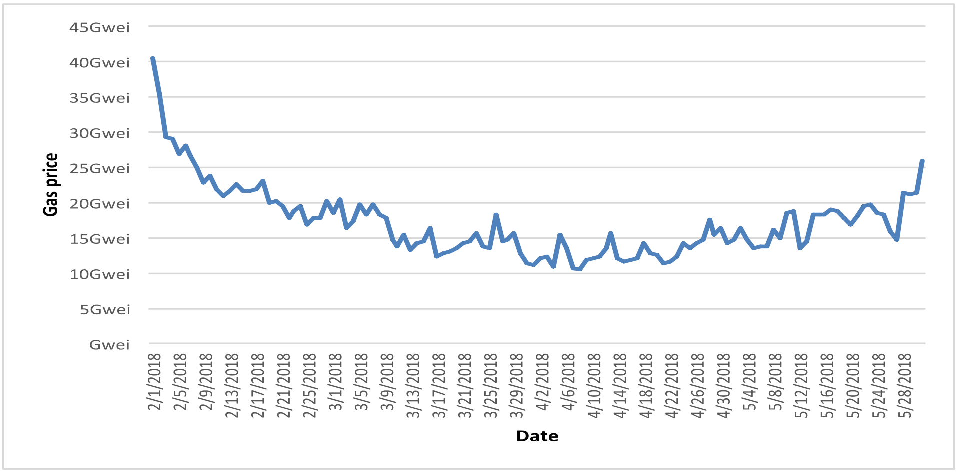

Fig. 10. Gas Price (2018.2-2018.5)

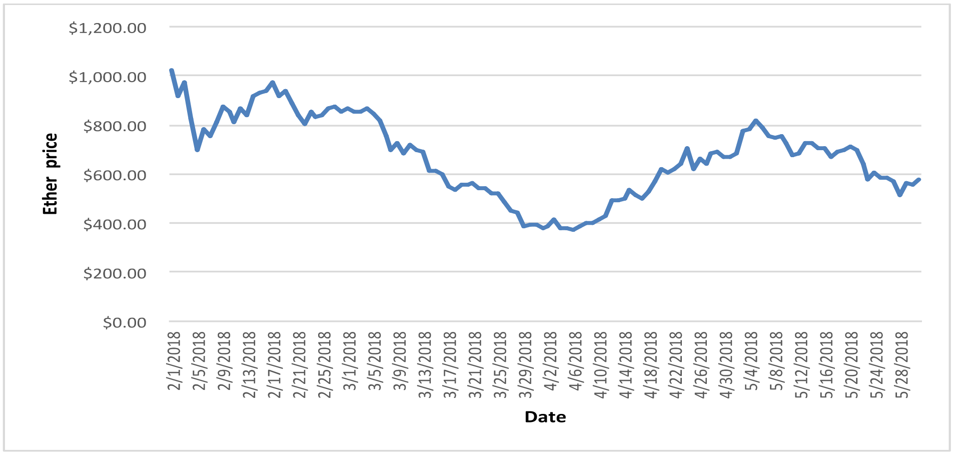

Fig. 11. Ether price (2018.2-2018.5)

Table 5
Average Gas and Ether Prices from February to May 2018

Average Gas Price	**Average Ether Price**
17.15973 Gwei	668.7079 USD

4.2 Discussions

The evaluation results have demonstrated that the proposed system can accurately decrease deployment costs of smart contracts compared with Oracle. The other problem in Oracle is it requires a predefined standard on data format. It is not compatible with the smart contracts that do not use Oracle standard during deployment. In Decoding module of Task Publisher in the proposed architecture, event interface is used to decode the event logs. Although we can use the standard interface of Oracle, the system would not be compatible with smart contracts that didn't have automatic callbacks, since Oracle standard interface is pre-specified. In addition, malicious attackers can easily decode the event in Oracle standard interface, even trigger the callback function when contract programmer does not limit the message sender of callback function. Although the purpose of application binary interface does not encrypt the transaction, it still increases the risk of smart contract. Moreover, if the standard interface of Oracle is used, this interface may be compatible with different types of data. Therefore, the readability of smart contracts will reduce, which will also cause users writing null data in the event log and incur additional costs inevitably.

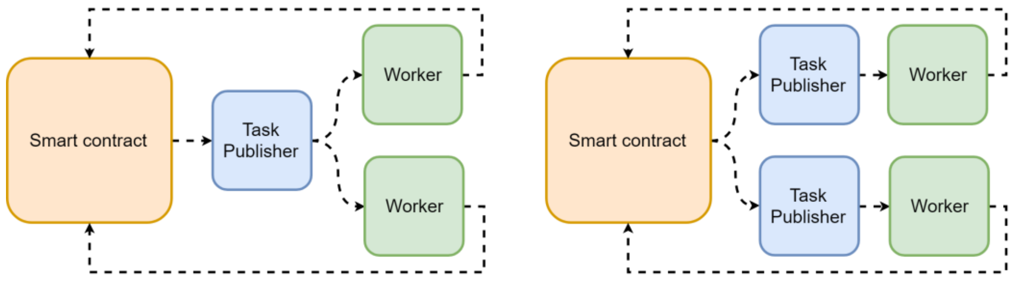

Fig. 12. The frameworks of double source consensus solutions

On the other hand, in data carrier service for smart contract, if the external data is off-chain data the data source consensus problem will arise. Although both the proposed system and Oracle service can guarantee that the source of the data is original source of the request and the calculated results have not been tampered, the use of data source cannot guarantee the correctness itself. If the smart contract is a type of contract like insurance, the information imported from the off-chain should be based on mutual agreement from different data resource parties. Generally, to solve data source consensus issue in Oracle service, the traditional way is that both parties should upload data and manage it by smart contract, and the other way is to verify it by decentralized computation solution, such as TrueBit [37]. However, in the proposed data carrier architecture, the off-chain data resource consensus problem can be solved more efficiently by using different Workers instead of managing mutual data in smart contract. The frameworks of double source consensus solutions in the proposed architecture are shown in Fig.12, the Workers could be simply assigned by single Task Publisher or assigned by different Task Publishers, respectively.

In terms of security, Oracle provides authenticity proofs for smart contract developer, but they are optional functions in Oracle, which means the use of authenticity proofs in Oracle requires the payment of an extra fee. The cost depends from the data source type used and by the authenticity proof requested. In addition, not all proofs are compatible with all data source types. Therefore, smart contract developer not only needs to cover the gas consumption of inheriting the basic data fetching functions in Oracle, but also needs to pay extra fees if authentication proofs are required. Compared with Oracle, the most significant contribution of the proposed system is it can help smart contract developers reduce their gas consumption during smart contract deployment. However, since the proposed system is deployed in off-chain environment, some users may doubt the transaction security in our system. Since the system components are designed elastic, there are different deployment scenarios of the proposed data carrier system. For users who require more trustworthy data fetching service, we provide a more trustworthy deployment scenario, as shown in Fig. 13. In this scenario, Worker component is suggested to deployed on user-side and maintained by themselves. Because Worker component contains transaction module which is the key module for passing fetched results to the smart contract. The data carrier system will be more trustworthy if users deploy Worker in their local sides.

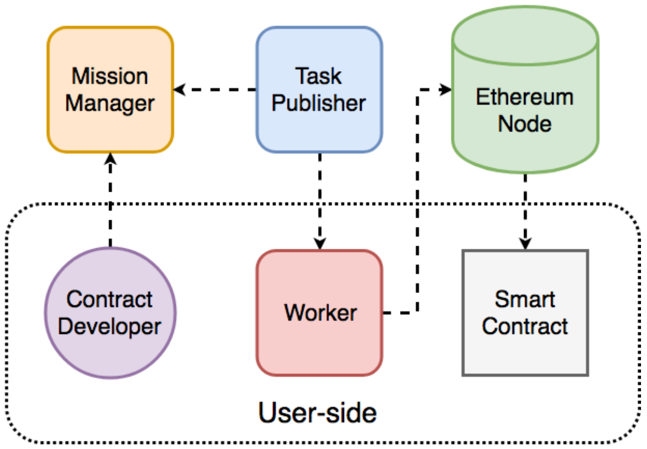

Fig. 13. More trustworthy deployment scenario of the proposed data carrier system

In addition to deploying Worker in user-side, an alternative for enhancing security of proposed system is adopting trusted hardware, such as Intel‘s Software Guard Extensions (SGX). SGX is a set of new instructions that confer hardware protections on user-level code, which has been used in Town Crier (TC) [38] for scraping HTTPS- enabled websites and serving source-authenticated data to relying smart contracts. In the future, we will try to incorporate Town Crier’s design, such as integrating Worker component with Intel SGX instruction set, to present a more cost-effective and trustworthy data carrier system in server-side. trustworthy data feeds.

5 Conclusions

This work proposes an elastic and cost-effective data carrier architecture for smart contracts in blockchain-enabled IoT environment that requires communication with external off-chain data. The proposed architecture consists of three components: Mission

Manager, Task Publisher and Worker. Selective solutions for filtering smart contract event and decoding event log to fit different requirements are presented. The proposed system is designed to minimize contract deployment costs and monitor contract event without subscribing any filter at Ethereum node. In the evaluation, we show that it will save about 20USD deployment cost for average by our data carrier system compared with Oracle service. We also discuss the deployments of solving data resource consensus problem caused by fetching off-chain data, and trustworthy scenario for users who require more secure data fetching service. Compared with Oracle, the proposed data carrier system is demonstrated more efficient, elastic and cost-effective. In the future, to make a great deal of improvement in security, we will try to combine the proposed components with Intel SGX instruction set and decentralized technologies, such as Raiden Network [39], to present a more cost-effective and secure data carrier system.

ACKNOWLEDGMENTS

This work was supported by the fund of National Natural Science Foundation of China (Grants No. 61702102), Natural Science Foundation of Fujian Province, China (Grant No. 2018J05100), Foundation for Distinguished Young Scholars of Fujian Agriculture and Forestry University (Grant No. xjq201809), and in part by the MOST of Taiwan (Grant No. 107-2623-E-009-006-D).

REFERENCES

[1] R. Li, T. Song, B. Mei, H. Li, X. Cheng, L. Sun, Blockchain for large-scale internet of things data storage and protection, IEEE Transactions on Services Computing, 2018, DOI: 10.1109/TSC.2018.2853167.

[2] D. Liu, Y. Xu, X. Huang, Identification of location spoofing in wireless sensor networks in non-line-of-sight conditions, IEEE Transactions on Industrial Informatics, 2018, 14(6), 2375-2384.

[3] Z. Yan, J. Liu, A. V. Vasilakos, L. T. Yang, Trustworthy data fusion and mining in internet of things, Future Generation Computer Systems, 2015, 49(C), 45-46.

[4] K. Muhammad, R. Hamza, J. Ahmad, J. Lloret, H. H. G. Wang, S. W. Baik, Secure surveillance framework for IoT systems using probabilistic image encryption, IEEE Transactions on Industrial Informatics, 2018, 14(8), 3679-3689.

[5] A. Reyna, C. Martín, J. Chen, E. Soler, M. Díaz, On blockchain and its integration with IoT. challenges and opportunities, Future Generation Computer Systems, 2018, 88, 173-190.

[6] J. Kang, R. Yu, X. Huang, S. Maharjan, Y. Zhang, E. Hossain, Enabling localized peer-to-peer electricity trading among plug-in hybrid electric vehicles using consortium blockchains, IEEE Transactions on Industrial Informatics, 2017, 13(6), 3154-3164.

[7] M. A. Khan, K. Salah, IoT security: review, blockchain solutions, and open challenges, Future Generation Computer Systems. 2018, 82, 395-411.

[8] Y. Zhang, J. Wen, The IoT electric business model: using blockchain technology for the internet of things, Peer-to-Peer Networking and Applications, 2017, 10(4), 983-994.

[9] S. Nakamoto, Bitcoin: a peer-to-peer electronic cash system. Consulted, 2008, Available: https://bitcoin.org/bitcoin.pdf.

[10] V. Buterin, A next-generation smart contract and decentralized application platform, Etherum 2014, 1–36. https://doi.org/10.5663/aps.v1i1.10138

[11] D. Magazzeni, P. Mcburney, W. Nash, Validation and verification of smart contracts: a research agenda, Computer, 2017, 50(9), 50-57.

[12] M.Alharby, A. Aldweesh, A. V. Moorsel, Blockchain-based Smart Contracts: A Systematic Mapping Study of Academic Research, ICCBB 2018: International Conference on Cloud Computing, Big Data and Blockchain, 1-6.

[13] M. Y. Afanasev, Y. V. Fedosov, A. A. Krylova, S. A. Shorokhov, An application of blockchain and smart contracts for machine-to-machine communications in cyber-physical production systems, IEEE Industrial Cyber-Physical Systems (ICPS), 2018, DOI: 10.1109/ICPHYS.2018.8387630.

[14] A. Kosba, A. Miller, E. Shi, Z. Wen, C. Papamanthou, Hawk: the blockchain model of cryptography and privacy-preserving smart contracts, IEEE Security & Privacy, 2016, pp.839-858.

[15] C. Dong, Y. Wang, A. Aldweesh, P. McCorry, and A. van Moorsel, Betrayal, distrust, and rationality: Smart counter-collusion contracts for verifiable cloud computing, in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 211–227, ACM, 2017.

[16] P. Ryan, Smart contract relations in e-commerce: legal implications of exchanges conducted on the blockchain, Technology Innovation Management Review, 2017, 7(10), 14-21.

[17] K. Christidis, M. Devetsikiotis, Blockchains and smart contracts for the internet of things, *IEEE Access*, 2016, 4, 2292-2303.

[18] Oraclize, API documentation. Available online: http://docs.oraclize.it/ (22 08 2018).

[19] C. Prybila, S. Schulte, C. Hochreiner, I. Weber, Runtime verification for business processes utilizing the bitcoin blockchain, Future Generation Computer Systems.2018, DOI: 10.1016/j.future.2017.08.024.

[20] T. T. A. Dinh, R. Liu, M. Zhang, G. Chen, B. C. Ooi, J. Wang, Untangling blockchain: a data processing view of blockchain systems, *IEEE Transactions on Knowledge & Data Engineering*, 2018, 30(7), 1366-1385.

[21] M. Swan, Anticipating the economic benefits of blockchain, Technology Innovation Management Review, 2017, 7(10), 6-13.

[22] Y. Yuan, F. Y.Wang, Blockchain and cryptocurrencies: model, techniques, and applications, IEEE Transactions on Systems Man & Cybernetics Systems, 2018, 48(9), 1421-1428.

[23] P. Veena, S. Panikkar, S. Nair, P. Brody, Empowering the edge-practical insights on a decentralized internet of things, IBM Institute for Business Value, 2015, 17.

[24] G. Prisco, Slock it to introduce smart locks linked to smart ethereum contracts, decentralize the sharing economy, Available online: https://bitcoinmagazine.com/articles/slock-it-to-introduce-smart-locks-linked-to-smart-ethereum-contracts-decentralize-the-sharing-economy-1446746719/. (22 08 2018).

[25] Chain of things, Available online: https://www.chainofthings.com/. (22 08 2018).

[26] J. Liu, W. Li, G. O. Karame, N. Asokan, Toward fairness of cryptocurrency payments, IEEE Security & Privacy, 2018, 16(3), 81-89.

[27] Understanding-oracles, Available online: https://blog.oraclize.it/understanding-oracles-99055c9c9f7b (22 08 2018).

[28] Ripple Labs, Available online: https://ripple.com (22 08 2018).

[29] Codius, Available online: https://github.com/codius/codius-wiki/wiki/White-Paper (22 08 2018).

[30] S. Ellis, A. Juels, S. Nazarov, Chain link a decentralized oracle network, Chainlink, 2017, 1-38.

[31] N. Atzei, M. Bartoletti, T. Cimoli, A survey of attacks on ethereum smart contracts (SoK), International Conference on Principles of Security and Trust, Springer, Berlin, Heidelberg, 2017, 164-186.

[32] Express, Available online: http://expressjs.com (22 08 2018).

[33] Rabbitmq, Available online: www.rabbitmq.com (22 08 2018).

[34] KrakenPriceTicker, Available online: https://dapps.oraclize.it/browser-solidity/#version=soljson-v0.4.19+commit.c4cbbb05.js&optimize=false&gist=ad3d1f6007942b727f5909b55e6445d2 (22 08 2018).

[35] Ether Historical Prices, Available online: https://etherscan.io/chart/etherprice (22 08 2018).

[36] Transaction Fees, Available online: https://etherscan.io/chart/transactionfee (22 08 2018).

[37] T. Jason, R. Christian, A scalable verification solution for blockchains, Ethereum, 2017, 1-50.

[38] F. Zhang, E. Cecchetti, K. Croman, Town Crier: An Authenticated Data Feed for Smart Contracts, Acm Conference on Computer & Communications Security, 2016, 1-20.

[39] R. Joseph, D. Thaddeus, The bitcoin lightning network: Scalable off-chain instant payments, 2016, https://lightning.network/lightning-network-paper.pdf.