Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 129 (2018) 410–419

Procedia Computer Science

www.elsevier.com/locate/procedia

2017 International Conference on Identification, Information and Knowledge in the Internet of Things

Approaches to Front-End IoT Application Development for the Ethereum Blockchain

Matevž Pustišek[a,] *, Andrej Kos[a]

[a]*University of Ljubljana, Faculty of Electrical Engineering, Tržaška 25, 1000 Ljubljana, Slovenia*

Abstract

There are several distributed ledger protocols potentially suitable for the Internet of things (IoT), including the Ethereum, Hyperledger Fabric and IOTA. This paper briefly presents and compares them from the IoT application development perspective. The IoT applications based on blockchain (BC) can incorporate the on-chain logic –the smart contracts– and Web, mobile or embedded client front-end application parts. We present three possible architectures for the IoT front-end BC applications. They differ in positioning of Ethereum blockchain clients (local device, remote server) and in positioning of key store needed for the management of outgoing transactions. The practical constraints of these architectures, which utilize the Ethereum network for trusted transaction exchange, are the data volumes, the location and synchronization of the full blockchain node and the location and the access to the Ethereum key store. Results of these experiments indicate that a full Ethereum node is not likely to reliably run on a constrained IoT devices. Therefore the architecture with remote Ethereum clients seems to be a viable approach, where two sub-options exist and differ in key store location/management. In addition, we proposed the use of architectures with a proprietary communication between the IoT device and remote blockchain client to further reduce the network traffic and enhance security. We expect it to be able to operate over low-power, low-bitrate mobile technologies, too. Our research clarifies differences in architectural approaches, but final decision for a particular ledger protocol and front-end application architecture is at strongly based on the particular intended use case.

Selection and peer-review under responsibility of the scientific committee of the 2017 International Conference on Identification, Information and Knowledge in the Internet of Things (IIKI2017).

Keywords: architecture; blockchain; Ethereum; front end application; IoT

* Corresponding author. Tel.: +386 1 4768844.
E-mail address: matevz.pustisek@fe.uni-lj.si

Selection and peer-review under responsibility of the scientific committee of the 2017 International Conference on Identification, Information and Knowledge in the Internet of Things (IIKI2017).
10.1016/j.procs.2018.03.017

1. Introduction

We can observe the initial attempts to jointly use the Internet of things (IoT) and the distributed ledger technologies. These attempts tend to study the feasibility of such an application development approach, provide proofs of concept (PoC) and explore possible business opportunities. The IoT [1] is a well-established concept referring to numerous interconnected things along with the corresponding cloud or fog based applications. It is revolutionizing the Internet and is being deployed in various applications domains. The distributed ledgers on the other hand –which are currently mostly implemented with blockchain technologies (BC)– are still emerging [2]. Nevertheless, they are likely to disrupt the field of ICT systems, services and application just as strongly as the IoT.

The scope of the existing BC systems is divergent in terms of technological features, as well as in their acceptance among the user- and developer communities. With first examples of BC based IoT solution deployments, certain inefficiencies in current BC design started appearing. Micropayments for example have become almost unrealistic in Bitcoin network due to high transaction fees and long transaction confirmation times. The scalability needed for IoT (expected billions of devices) is often limited due to the size of the blockchain and limited transaction rates. The existing BC protocols try facing some of these inefficiencies with functional extensions. In parallel, new ledger protocols are being developed, with IoT requirements built-in from the scratch.

Both developments –the IoT and the BC– are naturally seeking to be combined in common solutions, which thus provide an immense space for application development and use. However the right approach and the selection of appropriate technologies is far from being straightforward. It can crucially depend on the details of the intended use case. A seemingly small change in the foreseen use can lead to a drastic increase in complexity and additional efforts to adapt the solution, or might even be impossible.

The objective of our research is to analyze and present the practical constraints in the development of IoT applications based on Ethereum (ETH) BC. We therefore elaborate and compare the architectural approaches for the design of the front-end IoT device applications based on ETH BC. We implemented three versions of these architectures and evaluated them in terms of performance and security. The research provides directions for the IoT application developers to enable them selecting the appropriate system design and avoiding unrealistic expectations imposed to IoT devices and BC technologies. Their architectural approach can be thus shaped according to the intended use and the specifics of the planned IoT system.

In Section 2 we briefly present the state of the art including the three distributed ledger protocols that currently appear as viable candidates for IoT BC technologies and some cases of use of BC in IoT. In Section 3 we outline the principles of BC application development for IoT. Section 4 presents and compares four different architectural approaches for BC enabled IoT devices and analyzes their positive and negative sides, derived from our practical experimentation. Section 5 concludes the paper with a reflection to the expected future developments in BC for IoT.

2. State of the art

The blockchain technologies are well known fundament of cryptocurrencies, but offer many other possible applications areas, too. There are two key application domains for BC with differing business requirements [3]: the financial technologies (FinTech) and the IoT. Although both the domains require the basic common feature of a decentralized trusted ledger of transactions, substantial differences can be found in e.g. the principle use, volumes and rates of transactions, stringency of the security requirements or in cost of transactions. In FinTech the main challenge is to assure absolutely secure and trusted financial payments, with low transaction volumes and some tolerance to the transaction delays. In the IoT on the other hand, numerous devices and massive transaction volumes are expected, with micro- and nano-payments required for the IoT asset and data monetization. Transaction costs become a relevant issue here, as well as the transaction delays needed for near-real-time operation.

There are not many successful use cases of the IoT BC solutions that reach beyond simple proof-of-concept (PoC) or above the Technology readiness level (TRL) 4 and that incorporate more than just a couple of devices. This is not surprising as the BC IoT application domain still is in its infancy. The activities are directed primarily towards the clarification of the role of the BC in the IoT, testing limitations in implementation and exploring possible business opportunities. Nevertheless, some interesting use cases have been presented, primarily in the domains of smart grid and electric charging, logistics and IoT device management.

In smart grid key challenges that are being currently addressed with IoT and BC are smart meter reading, selling surplus energy in local micro grids, electric vehicle charging and demand side management [4], [5]. In logistics the role of IoT and BC is being investigated for product identification and tracking cargo shipments. In [6] a solution is presented to track containers, which measures light, temperature and other environmental parameters and then secures this information in a blockchain. This is important to prove the compliance of shipments with food or medical product regulations or even to apply automatic charging of penalties if shipping conditions are not met. In [7] a similar approach is applied in the pharma supply-chain. Zerado is focusing on access control for real estate based on NFC and BC [8] to enable the growth of the sharing economy. The IoT device management is fundamental to other application domain, because it includes access and storage of IoT data in BCs. In [9] this concept is proven in a smart-home scenario to manage home appliances and electricity consumption. Similar idea is elaborated in [10] for management of vending machines.

2.1. Divergent distributed ledger technologies

Various specifications and implementations of distributed ledgers are available, but in our opinion at the moment three have relevant prospects for the IoT. These are two BCs, the Ethereum [11] and the Hyperledger Fabric [12]. The third is the IOTA [13], which is based on a new block-less distributed ledger architecture. Although the Bitcoin (BTC) [14] is probably the most prominent BC protocol which gained reputation mostly due to the popular cryptocurrency bitcoin, its potential role in the IoT and/or distributed application development is extremely limited and is not a viable candidate for an IoT BC solution. Bitcoin protocol is namely lacking the distributed on-chain smart applications, has very big chain size and long transaction confirmation delays. Its role is thus limited more or less to supporting a cryptocurrency in FinTech applications.

Table 1. Comparison of distributed ledger technologies for the IoT.

	Bitcoin	Ethereum	Hyperledger Fabric	IOTA
Native cryptocurrency	Yes	Yes	No	Yes
Decentralized applications	No (very limited)	Yes – smart contracts: Solidity	Yes – chaincode: Go, Java - (executed in containers)	No (very limited)
Transaction fee	Yes	Yes	No	No
Network type	Public	Public (or private)	Private	Public
Network access	Permissionless	Permissionless	Permissioned	Both
Anonymous accounts	Yes	Yes	No	Both
State channels	Lightning	µRaiden, Raiden, Generalized	Not required	Not required
Suitable for IoT	No	Yes (with some constraints)	Yes	Yes
Suitable for DApps	No	Yes	Yes	No

Other practical limitation in popular BC protocols are becoming evident with new application domains of BC, as for example the IoT. The existing BC protocols try to cope with the limitations with additions that more or less successfully patch the core BC protocols. The state channels for example, combine off- and on-chain transactions to contribute to additional scalability, privacy and the reduction of confirmation delays, compared to the current BC architectures. In ETH this approach is manifested in the Raiden [15] and in BTC in the Lightning network [16]. The ETH smart contracts cannot contact external URL, which limits their integration with the "world outside of the chain". This shortcoming could be outdone by oracles [17]. These serve as intermediaries, providing data feeds along with an authenticity proof to the blockchain form/to external software (e.g. Web sites) or hardware entities. These add-ons have gained some interest, but are immature (e.g. strong mismatches between announced roadmaps and actual dates

of delivery) and with little practical acceptance. This explains why IOTA took a different approach, where the ledger technology (and entire system around it) was designed for the IoT from the very beginning.

The three distributed ledger technologies are compared in Table 1. Particular technologies are presented in more detail in the following subsections.

2.1.1. The Ethereum

The Ethereum white paper [11], which is the initial document describing ETH, explains that the Ethereum protocol was originally conceived as an upgraded version of a cryptocurrency, providing advanced features such as on-blockchain escrow, withdrawal limits, financial contracts, gambling markets and the like via a highly generalized programming language. The ETH protocol, which is being developed by The Ethereum Foundation, is specified in the Yellow paper [18].

ETH protocol is a BC protocol. New transactions form blocks which are validated by mining nodes. The miners use proof-of-work (PoW) consensus algorithm. Miners are rewarded for their work by mining fees, paid by transaction issuers. The ETH nodes can participate in one of two public networks, the mainnet or the test network Ropsten. Both run the same BC protocols, but the cryptocurrency in Ropsten has no real value. Private networks are possible, too.

The key innovation in ETH compared to BTC is the support of smart contracts (SC). These are not some formal requirements or obligations, but can be more adequately explained as autonomous agents, whose behavior is determined by their contract code. This code is executed every time this account receives a message, which is a transaction addressed to it. To develop smart contracts and thus the distributed applications (DApps) a computationally universal (i.e. Turing complete) language is provided. The fundamental SC language is the low-level bytecode language and ETH network provides a virtual machine (i.e. Ethereum virtual machine, EVM) which executes such code. Several high(er) level languages are available for application development. The current flagship is Solidity [19] – a JavaScript like language, but other languages have been used in the past. Higher level code is compiled to bytecode prior to execution in the EVM.

For these reasons Ethereum emerged as the platform for (i) financial (currencies, token systems), (ii) semi-financial (e.g. crowd sensing) and (iii) non-financial applications (on-line voting, decentralized governance). It is the leading BC protocol in terms of innovation. Most of the BC projects aiming beyond simple value transactions and coin offerings are based on ETH. The ETH cryptocurrency –ether– has second largest market capitalization and is preceded only by bitcoin.

2.1.2. The Hyperledger Fabric

The Hyperledger Project is a collaborative effort to create an enterprise-grade, open-source distributed ledger framework and code base. Established as a project of the Linux Foundation in early 2016, the Hyperledger Project currently has more than 130 members, including leaders in finance, banking, in the internet of things, supply chain, manufacturing and technology.

The Hyperledger Fabric (HLF) [12], one of the multiple projects currently in incubation under the Hyperledger Project, is a permissioned blockchain platform aimed predominantly at business use. It is open-source and based on standards, runs arbitrary smart contracts (called chaincode), supports strong security, identity features, basic REST APIs, CLIs and uses a modular architecture with pluggable consensus protocols (currently an implementation of Byzantine fault-tolerant consensus using the PBFT protocol is supported). The distributed ledger protocol of the fabric is run by peers. The fabric distinguishes between two kinds of peers: (i) validating peer is a node on the network responsible for running consensus, validating transactions, and maintaining the ledger and (ii) a non-validating peer which is a node that functions as a proxy to connect to validating peers [20].

The HLF protocol is implemented in e.g. IBM Watson IoT™ platform [21]. This a BC-as-a-Service platform (BaaS), which enables the IoT devices to send data to private blockchain ledgers for inclusion in shared transactions with tamper-resistant records. HLF in IBM Watson IoT is predominantly suitable for private blockchains in enterprise settings, because it is using a different consensus algorithm than e.g. ETH. It is distinguished by a well-documented HTTPS REST API for all blockchain related functions. Web developers can thus benefit from BC features, but continue to utilize API technologies they are already familiar blocks and transactions as well as peers and networks, monitoring of chain status, and registrations and management of BC users.

2.1.3. IOTA

The IOTA [13] development was initiated by the IOTA Foundation in 2015. The key distinguishing feature of IOTA distributed ledger compared to BC technologies is that it was initially designed to cope with the challenges imposed by IoT. These include scalability (the IOTA ledger does no limit the number or rated of transactions), zero transaction fees (different consensus principle) and rapid transaction confirmations. The new options provided to IoT in this way include machine-to-machine nano-payments, secure (sensor) data feeds and identification of IoT devices.

Unlike the ETH or BTC, which are to some extent trapped by their huge success as cryptocurrencies and try to cope with the challenges of IoT with add-ons and extensions of their protocols (state channels, considerations of alternative consensus mechanisms), the IOTA is based on a new blockless distributed ledger architecture. This is implemented with directed acyclic graphs (DAG) and called the tangle. It provides drastically different approach to transaction validation than mining of blocks in the ETH and BTC. For each issued transaction an IOTA node has to approve two transactions from other nodes in the tangle at no fee. Contrary to today's BCs, consensus is no-longer decoupled but instead an intrinsic part of the system, leading to decentralized and self-regulating peer-to-peer network. One of key limitations of IOTA is the lack of transaction order as a consequence of DAG. This practically limits the possibility for smart contracts in IOTA.

IOTA application environment consists of IOTA nodes. These are composed of a core or a light (for resource restricted edge devices) client communicating with the tangle, and application part. There reference implementation of the clients is Java, but other are included in the development roadmap [22]. A JavaScript library with complete API coverage is available for NodeJS and browser applications. All API calls are to be sent via a POST HTTP request in JSON format. The clients provide core IOTA functionality, which can be extended with modules through the IOTA eXtension Interface (IXI). One of IOTA modules is Masked Authenticated Messaging (MAM) for secure, encrypted and authorized data stream services.

There is a tradeable cryptocurrency available in the public IOTA network.

3. Blockchain application development for the IoT

Distributed ledgers provide a trusted environment for exchange of transaction. In terms of application development for the IoT two paradigms can be combined—front-end and on-chain. Depending on the intended use both application parts can be combined for one solution. **Front-end application parts** are Web, mobile and embedded applications, which use BC via client APIs that are exposed by the BC clients. Front-end application parts are required for user interfaces and for IoT devices to utilize the BC. **On-chain business logic** refers to smart contracts (i.e. chaincode in HLF), which are programs deployed and executed in the BC network. Executions of smart contracts are validated in the BC. BC thus provides a decentralized and trusted virtual machine for smart contract executions. The on-chain logic is not absolutely required for IoT.

3.1. On-chain application part

The decentralized environment for trusted transactions, which eliminates the need for trusted central authorities, is e.g. fundament for e.g. cryptocurrencies. However, some BC technologies go beyond in providing smart contracts – the truly revolutionizing feature of BC, not present in the traditional Web, cloud, mashup architectures. Smart contracts are on-chain business logic that is executed within the blockchain network. The execution can be verified by any network participant and thus trusted in the same way as any other transaction in BC network is.

Smart contracts code is written a corresponding programming language (e.g. in Solidity for ETH, in Go (or Java) for HLF), it is compiled to the bitcode suitable for particular BC, and deployed to the network.

Once deployed in the BC network a smart contract is address by its unique address, similarly to the regular BC accounts. A smart contract exposes functions, which can be used by other blockchain accounts. These functions represent a kind of an on-chain API for other BC accounts, and are accessible via blockchain. Smart contract receives transactions addressed to it, with parameter required by a specific function in SC embedded in the transaction. The smart contract processes the incoming request according to its programming logic and optionally launches events.

3.2. Front-end application part

The Web, mobile or embedded applications combine the regular application logic (e.g. for the operation of the user interfaces, acquisition of sensor data, local data processing) with BC capabilities. The latter can be a simple transaction exchange in the BC network or communications with on-chain application part, i.e. the smart contract. The front-end applications use the BC via BC client API libraries and BC client APIs that are exposed by the BC clients. These functional blocks of a front-end application part are described in more detail in the continuation of this Section.

The front-end application parts are required for user interfaces and IoT devices to utilize the BC. The front-end application programs can be run in the BC-enabled Web browsers/wallets, e.g. Mist [23] or Chrome and Firefox with Metamask browser plugin [24]. This enables a very efficient implementation of Web based user interfaces (UI). A browser is selected as the application execution environment, when the front-end part requires a UI and the developers want to rely on known Web UI technologies (e.g. HTML5, JS), but still apply the BC in their solutions. Beside the common browser features (HTTP/HTTPs protocols, HTML rendering), a BC-enabled Web browser implements BC client API libraries needed for the application programs to use the BC, as well as the key wallet to securely store and manage users' BC accounts. BC-enabled Web browser is just the simplest, but of course not the only option to implement UI. Advanced UIs can be built e.g. as cross platform desktop apps, but the developers need to import libraries for BC communications and implement key wallets on their own.

An unmanned embedded IoT system operates without direct user interventions, so a browser is not the appropriate environment for application execution. In that case the application is usually executed in some serve side runtime environment (e.g. NodeJS for JS) and the appropriate BC client API libraries need to be imported in the environment for proper operation. This is the fundament for an IoT device with BC support. There are two key modes of operation for the BC-enabled IoT devices to work with and react upon the changes in the BC. In the first case a device is identified by a BC address/account. The BC transactions can be sent to and from this address. For the outgoing transactions to be properly signed by the issuers, location of and secure access to the account key store is needed (see Section 4 for details). In this mode the device/application can e.g. autonomously record its status in the chain. In the second case, the IoT device does not have its own BC account. But even without it, a device can intercept the transactions or the events created by the smart contracts in the BC network. In this way the application can execute certain actions (e.g. toggle on a relay), if a corresponding transaction or event was recorded in the chain (e.g. transaction of some value to a specified BC address). This mode of operation is passive, IoT devices/application cannot create transactions (operates as a sniffer), but is much simpler in terms of secure key store management. Although rather distinct in their scopes, both modes of operation have practical value for front-end application.

There are four key functional blocks present in a front-end application to provide the desired functionality and communicate properly with the BC:

- The BC client is responsible for running the BC protocols and thus the entire communication with BC network. This includes the management of blocks (keeping the local chain up-to-date) and transactions (e.g. sending outgoing transactions), listening to events, management of peers and network, monitoring of chain status, managing the accounts or mining blocks. There are several ETH BC client implementations available, but *geth* [25] usually serves as the reference, because it is being developed by Ethereum Foundation developers. A popular alternative is the parity [26]. In IOTA the BC client is IOTA Core [27].
- The BC client API is a part of the BC client that exposes the clients' capabilities. Through this client API the entire functionality of BC client can be exploited. The API can be accessed through the common programing and communication interfaces, usually the inter-process communication (IPC), HTTP POST or Websockets (WS). IPC can be applied if the application and BC client run on the same machine (local communication), HTTP and WS on the other hand enable also the remote access to the BC client. The data passing through one of this channels is usually structured as JSON.

  The BC client API libraries facilitate application development and use of the BC client API. There are various implementations of these libraries available, for different programming languages and by different developers. In ETH such a library is the web3.js [28] (current version 0.20.x, with 1.0.0 beta already announced), in IOTA is the iota.lib.js [29] - all for JavaScript. Other implementations may vary in their maturity. These libraries are included in the application projects. Apart from interfacing the BC client API, these libraries can provide additional features. One this is the local wallet, which keeps and secures access to user accounts and keys, and enables

signing the outgoing transactions. This functionality for example, is not supported in the ETH web3.js 0.x.x, but is added in the 1.0.0 beta. This is of outmost importance for the IoT BC application development as now the application code can manage the accounts easily, securely and without user interaction. Dedicated ETH wallets/browsers require user confirmation for a new outgoing transaction, which is not applicable in embedded devices and applications. The BC client API libraries provide functions for signing the transactions and passing it to the BC network. In the programming language of the front-end application a transaction is presented as a data structure, which is passed then to the corresponding functions. This data structure has no signature filed, but therefore usually defines the source address. In JS programing with web3.js 1.0.0 library for example, the function *sendTransaction()* receives such a structure in JSON format, creates appropriate signature, encodes the results in RLP to build the raw transaction and broadcasts it to the BC network peers. The *signTransaction()* on the other hand only creates a raw transaction that can be then later passed to the network by e.g. *sendSignedTransaction()*. For a transaction to be signed, the *sendTransaction()* and *signTransaction()* require access to an unlocked ETH account.

- The application implements the desired functionality and utilizes the BC through the BC client API libraries. Application programming code is in case of ETH and IOTA frequently written in JavaScript. The reasons for this are twofold. First, in both cases the JS BC client API libraries are the most advanced and proven, and second, it is suitable for browser based in IoT device applications.

4. Outline of the architectures for front-end applications in Ethereum based IoT devices

The architecture of the front-end applications parts heavily depends on the capabilities and limitations of the IoT devices, where the front-ends are deployed. The IoT devices demonstrate a wide range of communication (bitrate, persistence of connectivity) and computation (CPU, storage) capabilities, ranging from dumb sensor nodes to fully equipped computers. It is therefore necessary to know these capabilities in advance, to properly select where particular functional blocks (Section 3.2) can run and how they are configured. The ETH client *geth* for example, can be run with various BC synchronization options (full client – entire chain data (block headers and block bodies) is downloaded and stored in the device; light client – only gets the current state of the chain, faster synchronization, however, unreliable event filtering).

All the considerations about the architecture and configurations are aiming to provide a reliable execution of the front-end application logic and of the workflow for the ETH transactions (creation, signing, submitting, monitoring) related to a particular device.

There are various options how to design the front-end application architecture of an ETH-enabled IoT device. The options differ in computation, communication and security requirements imposed to the IoT device. Computation and communication constraints are predominantly related to the synchronization of the chain data (huge amounts). Communication can as well become an issue for transaction exchange, if very low bit rate channels are applied in the device (e.g. Low-Power Wide-Area Network (LPWAN)). Security in respect of architecture refers to the location of and the access to the key store, needed for transaction signatures and to the access control of HTTP and WS channels for JSON-RPC.

The options for the architectures for front-end applications in ETH-enabled IoT devices are the following:

- ***Stand-alone node*** with *geth* client and the application part running on the same physical device and with a local key store.
- ***Remote geth client*** where the JS application and BC client API libraries run on a constrained IoT device, but the BC client (*geth*) on another, more capable computer/server. In this case two sub-options exist:
  - Local key store in the IoT device or
  - Remote key store in the server where *geth* is running.

4.1. Stand-alone IoT node architecture

In the stand-alone node architecture, which is depicted in Fig. 2a, all the functional blocks run in the same physical device. As the BC client (*geth*) is running there as well, it imposes high demand on CPU and memory. If the full BC

synchronization is enabled, one has to count to several GB of ETH chain data to be transferred to and stored at the device.

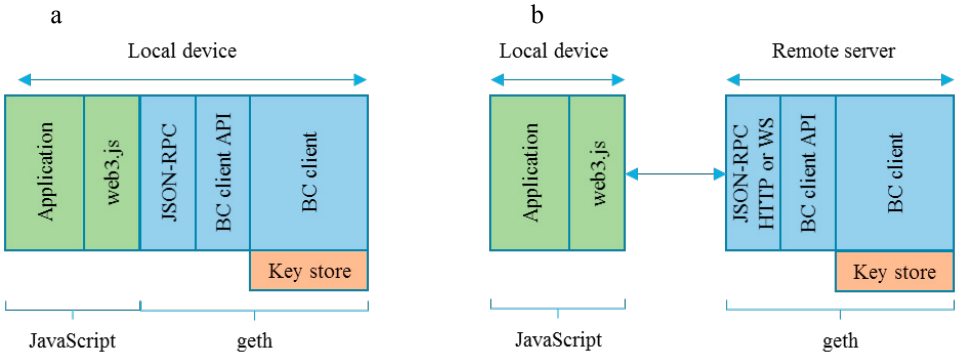

Fig. 1. (a) Stand-alone IoT node and (b) remote *geth* client architectures

Key store in this case is placed locally and is unlocked by *geth* upon the BC client initialization. The key risk in this case is the hardware security (stolen keys, if the physical device privacy is violated).

Our experience with such a setup showed that it is suitable only for the most powerful (IoT) devices. We tried to run the full client on RPi v3B embedded system with a wired internet connection. The syncing of the chain proved to be highly unreliable. We experienced unusual long synchronizations (syncing lasted for several days and was still not completed), unexpected interruptions in synchronization, etc. While conducting these tests we had a reference client running on a regular computer (same IP network capacities) and syncing there was unproblematic. It is important to know, that a not synced BC prevents the application part from using BC services. We tried running the *geth* in light mode, too. The syncing was more successful, however we experienced severe problems with filtering the events, which were launched by our smart contract. Some events were lost due to uncomplete data information provided, despite the corresponding transactions being dully recorded, and chain synced.

4.2. Remote geth client with remote key store

With remote *geth* client and remote key store we run *geth* on a separate, constraint-less server. The JavaScript application part of course remains in the local IoT device. In this way the most resource demanding part is moved from the IoT device. A remote server exposes *geth* functionality over JSON-RPC API, with HTTP or WS as the transport channel. In this setup the key store remained at the server and was applied at the *geth* initialization, just as in the case of the stand-alone node. The remote *geth* client with remote key store architecture is depicted in Fig. 1b.

This architecture actually proved to have a practical value. A local device was successful in running the application part, while a remote server seamlessly run the *geth*. We analyzed the traffic between the local device and the remote server with Wireshark. A typical transaction submitted by the application to the *geth* in form of JSON-RPC over HTTP comprised of one HTTP POST request and response. The size of the request messages was about 800 B, with the typical transaction in JSON format about 450 B. The rest were message TCP/IP protocol headers. The response message was smaller, 280 B. If WS was used instead of HTTP, the messages were roughly 200 B smaller. This does not seem much, but can still exceed communication limits of low-bit-rate devices. This is especially true, if not only a limited number of transactions is passed over HTTP/WS, but as well some event filtering form web3.js is applied, which utilized polling principle, generating constant network load.

However this architecture has potential security risks we need to understand. If *geth* is run with the key store unlocked, than anyone accessing the *geth* with JSON-RPC over HTTP or WS can create transactions signed with this key. There is no access control to HTTP or WS built in the *geth*, so we need to plan the IP network layer security very carefully in this case. This risks are relevant only in the case where the IoT device acts as an active transaction creator. If it runs in passive mode (sniffing the chain for transactions and events), no key store is required, so there are no risks in this respect.

4.3. Remote geth client with local key store

The remote *geth* client with local key store architecture and the one with remote key store differ in the positioning of the key store. As this is no longer placed at the server, security risk of sharing the same *geth* server by several devices diminish. In this case however, the application has to (create and) submit raw transactions, including the signature and apply proper serialization. To be able to do that, new/additional JavaScript libraries are required. Signing of transactions is not supported in web3.js version 0.x.x. In this case additional libraries like LightWallet or Ethereumjs-tx have to be applied. Fortunately, web3.js version 1.0.0.beta announced all the functions needed to keep and manage local wallets. We see this as the most recommendable approach to implement the remote *geth* client with local key store architecture.

Interestingly, passing the raw transactions instead of JSON transaction objects over the HTTP/WS, did not result in smaller message sizes, which were expected due to the more efficient RLP encoding. The raw transaction namely includes the signature and transaction hash (not present in JSON transaction object), resulting in about 40 B larger messages in this particular case.

4.4. Proprietary local-device to remote-server communication

As the last option we see a proprietary communication protocol between the IoT local device end the remote (*geth*) server, discarding the existing JSON or RLP data formats, too. We see two benefits in it. First, the communication bandwidth requirements can be reduced to the minimum, and be thus able to communicate over the low-bit-rate channels, too. Second, we can apply advanced server access controls to minimize security risks described in the remote *geth* client with remote key store architecture.

This architecture to some extent digresses from the decentralized peer-to-peer philosophy which is fundamental to the distributed ledgers and BCs. On the other hand, similar approaches are taken e.g. in the most of the mobile BC clients (mobile app has to trust the server which provides it the BC functionality). The recent fog computing developments and 4/5G network architectures also indicate, that network edge nodes could serve as the application gateways, providing functionality such as this, to the end nodes.

5. Conclusions

With our research we expect to clarify how to match the requirements and constraints of the IoT devices with the appropriate architectural approach to develop the front-end IoT applications for ETH. We are currently developing tools to generate and monitor transactions in the IoT devices automatically, which will enable a systematic performance testing of the abovementioned architectures. We will continue with the network load measurements and traffic profiling, to conduct emulation and simulation studies of the ETH IoT nodes connected over the low-power, low-bitrate mobile technologies.

Acknowledgements

The authors wish to acknowledge the support of the research program “Algorithms and Optimization Procedures in Telecommunications”, financed by the Slovenian Research Agency.

References

[1] Dechamps A, Duda A, Skarmeta A, Lathouwer BD, Agostinho C, Cosgrove-Sacks C, et al. Internet of things applications - from research and innovation to market deployment. [Internet]. Vermesan O, Friess P, editors. Delft: River Publishers; 2014. (River Publishers Series in Communications). Available from: http://www.internet-of-things-research.eu/pdf/IoT-From%20Research%20and%20Innovation%20to%20Market%20Deployment_IERC_Cluster_eBook_978-87-93102-95-8_P.pdf

[2] Kasey Panetta. Gartner’s Top 10 Strategic Technology Trends for 2017 - Smarter With Gartner [Internet]. 2016 [cited 2018 Jan 25]. Available from: http://www.gartner.com/smarterwithgartner/gartners-top-10-technology-trends-2017/

[3] Mulholland A. Blockchain or Distributed Ledger? Defining the requirement, not the technology [Internet]. Constellation Research Inc. 2017 [cited 2018 Jan 25]. Available from: https://www.constellationr.com/blog-news/blockchain-or-distributed-ledger-defining-requirement-not-technology-0
[4] Share&Charge - Charging Station Network - Become part of the Community! [Internet]. [cited 2018 Jan 25]. Available from: https://shareandcharge.com/en/
[5] Brainbot Technologies AG [Internet]. Smart Contract and Blockchain Consulting for Enterprises. [cited 2018 Jan 25]. Available from: http://www.brainbot.com/
[6] Ford N. IoT Application Using Watson IoT & IBM Blockchain [Internet]. Mendix. 2017 [cited 2018 Jan 25]. Available from: https://www.mendix.com/blog/built-iot-application-10-days-using-watson-iot-ibm-blockchain/
[7] Bocek T, Rodrigues BB, Strasser T, Stiller B. Blockchains everywhere - a use-case of blockchains in the pharma supply-chain. In: 2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM). 2017. p. 772–7.
[8] Zerado Access Control [Internet]. Zerado. [cited 2018 Jan 25]. Available from: http://zerado.com/en/products-services/access-control/
[9] Huh S, Cho S, Kim S. Managing IoT devices using blockchain platform. In: 2017 19th International Conference on Advanced Communication Technology (ICACT). 2017. p. 464–7.
[10] IBM Watson Internet of Things. Blockchain and IoT: Vending Machine with eSIM Demo [Internet]. 2017 [cited 2018 Jan 25]. Available from: https://www.youtube.com/watch?v=T9kYuBcOnjI
[11] Viktor Trón, Felix Lange. Ethereum Specification [Internet]. 2015 [cited 2018 Jan 25]. Available from: https://github.com/ethereum/go-ethereum/wiki/Ethereum-Specification
[12] IBM Blockchain based on Hyperledger Fabric from the Linux Foundation [Internet]. 2017 [cited 2018 Jan 25]. Available from: https://www.ibm.com/blockchain/hyperledger.html
[13] IOTA Developer Hub [Internet]. [cited 2018 Jan 25]. Available from: https://dev.iota.org/
[14] Protocol documentation - Bitcoin Wiki [Internet]. [cited 2018 Jan 25]. Available from: https://en.bitcoin.it/wiki/Protocol_documentation
[15] The Raiden Network [Internet]. High speed asset transfers for Ethereum. 2016 [cited 2018 Jan 25]. Available from: http://raiden.network/
[16] Lightning Network [Internet]. Scalable, Instant Bitcoin/Blockchain Transactions. [cited 2017 May 5]. Available from: http://lightning.network/
[17] Oraclize Documentation [Internet]. Overview. [cited 2018 Jan 25]. Available from: http://docs.oraclize.it/#overview
[18] Gavin Wood. The “Yellow Paper”: Ethereum’s formal specification [Internet]. 2017 [cited 2017 Aug 6]. Available from: https://ethereum.github.io/yellowpaper/paper.pdf
[19] Solidity — Solidity 0.4.19 documentation [Internet]. [cited 2018 Jan 25]. Available from: http://solidity.readthedocs.io/en/latest/index.html
[20] Cachin C. Architecture of the Hyperledger Blockchain Fabric. In: Workshop on Distributed Cryptocurrencies and Consensus Ledgers [Internet]. Chicago, Illinois, USA: ACM; 2016 [cited 2017 Jun 21]. Available from: https://www.zurich.ibm.com/dccl/#program
[21] IBM Watson Internet of Things. Blockchain and IoT: Vending Machine with eSIM Demo [Internet]. 2017 [cited 2018 Jan 25]. Available from: https://www.youtube.com/watch?v=T9kYuBcOnjI
[22] D. Sonstebo, “IOTA Development Roadmap,” IOTA, 31-Mar-2017. [Online]. Available: https://blog.iota.org/iota-development-roadmap-74741f37ed01. [Accessed: 02-Aug-2017].
[23] Ethereum/mist: Mist [Internet]. Browse and use ?apps on the Ethereum network. [cited 2018 Jan 25]. Available from: https://github.com/ethereum/mist/#mist-browser
[24] MetaMask [Internet]. Brings Ethereum to your browser. [cited 2018 Jan 25]. Available from: https://metamask.io/
[25] Viktor Trón, Felix Lange. Geth [Internet]. ethereum/go-ethereum Wiki · GitHub. 2017 [cited 2018 Jan 25]. Available from: https://github.com/ethereum/go-ethereum/wiki/geth
[26] Parity Ethereum Browser [Internet]. Parity Technologies. [cited 2018 Jan 25]. Available from: https://parity.io/
[27] IOTA Reference Implementation - iri [Internet]. GitHub. [cited 2018 Jan 25]. Available from: https://github.com/iotaledger/iri
[28] web3.js - Ethereum JavaScript API [Internet]. GitHub. [cited 2018 Jan 25]. Available from: https://github.com/ethereum/web3.js
[29] iota.lib.js - IOTA Javascript Library [Internet]. GitHub. [cited 2018 Jan 25]. Available from: https://github.com/iotaledger/iota.lib.js