Experiences Designing a Multi-Tier Architecture for a Decentralized Blockchain Application in the Energy Domain

Denis Rangelov, Nikolay Tcholtchev, Philipp Lämmel, Ina Schieferdecker

Fraunhofer Institute for Open Communication Systems (FOKUS)
Berlin, Germany
{firstname.lastname}@fokus.fraunhofer.de

***Abstract*—In recent years the emergence of the Ethereum Blockchain has introduced a new alternative perspective on how web applications can be build. More precisely, the Ethereum Blockchain allows the development of applications, where programming code can be executed in a decentralized manner with no restrictions imposed by a central authority. However, as it is the case with many emerging technologies, there is a fair amount of trade-offs that have to be considered when this technology is used as a platform for implementing decentralized applications. In this work we present two architectural designs for building decentralized applications (DApps) based on the Ethereum Blockchain technology. Within this context, we discuss the inherent strengths and weaknesses of each of the architectural designs as well as the set of challenges that we faced during the development process.**

***Index Terms*—blockchain, architectures, decentralized applications, P2P**

I. Introduction

Despite being primarily discussed within the context of cryptocurrencies, the original Bitcoin [1] paper introduced the Blockchain technology which has a wide range of applications exceeding the digital currency domain. Put simply, a Blockchain, as the name suggests, is a fully replicated data structure used in a peer-to-peer (P2P) network that consists of blocks that are linked to each other with hash pointers. Each block in the chain has a digital fingerprint, also known as "hash" and stores a set of transactions. Changing any of the transactions inside a block automatically changes its hash, which in turn invalidates all the following blocks in the chain.

In addition, adding a new block to the chain requires running a consensus algorithm (one of which is the so-called "proof-of-work") within the peer-to-peer network. This algorithm determines whether or not a peer in the network is allowed to add a new block to the chain or not. In case a new block is added to the chain, all peers in the network are notified and the new block is broadcasted to them.

With this context in mind, the Blockchain technology is considered to be tamper- and censorship-proof since no central authority or individual entity can change the transaction history or prevent transactions from happening.

These properties of the Blockchain make it a very compelling technology for a multitude of different domains with one of the most promising ones being the energy trading market. In particular, the advancement and growing availability in the field of renewable energy sources [2] has the potential to allow the users of electric energy to change their role from consumers to producers. That being the case, producing energy could potentially enable users to complete energy trades where a person who generates more energy, sells the generated amount for a small profit. With that comes the question: Is it possible to execute the above-mentioned energy transactions in a peer-to-peer manner without the need for a middleman or a central authority?

As already suggested, being tamper- and censorship-resistant makes the Blockchain a potential candidate for enabling peer-to-peer energy trading. However, not all Blockchain-related technologies are created equal. For instance, the Blockchain network that Bitcoin uses for performing cryptocurrency transactions is very unlikely to satisfy the needs of a decentralized energy trading system. Fortunately, there is a technology that allows developers to write their own business logic and deploy decentralized applications (also known as DApps) to a public Blockchain where they are running. One of the Blockchain implementations that allows this type of operations is called Ethereum [3].

This work presents multiple different ways to organize an Ethereum-based decentralized application in terms of architectural design. After introducing two available options for structuring such decentralized application, in section 3 we present a case study based on one of the described architectures. The case study encompasses an energy trading platform developed at the Fraunhofer FOKUS Institute and it is based on the Ethereum Blockchain. After describing the platform, in the sections that follow we will address the main challenge that we faced during the development process and namely - the integration of the Ethereum and standard database accounts. To conclude the paper, in the last section some closing thoughts and a future outlook will be presented.

II. Alternative Architectural Approaches

In this section, the main goal is to present two different architectural design patterns that should be considered when building a Decentralized Application (DApp) deployed on the

Ethereum Blockchain. The different approaches presented in this section are not unique to Blockchain-based applications, but when examined within that context they introduce a set of up- and downsides that deserve special attention.

The two architectural designs that the section illustrates differentiate from one another by the role of the end user in the communication flow and the location of the Ethereum account credentials. The first approach allows the client to directly communicate with the Ethereum Blockchain network. The second one utilizes a proxy which forwards the request from the client to the Ethereum Blockchain network by executing transactions on behalf of the user.

These two approaches are the subject of a comprehensive discussion that comes in the following subsections. This discussion includes not only the details about each of the architectural patterns itself, but it also examines the inherent advantages and disadvantages that each pattern represents.

Lastly, it is important to clarify that the goal of the subsections to follow is not to present an exhaustive list of all possible architectural design options. Rather, the scope of the discussion throughout this work will be limited to two approaches which have the widest range of practical applications.

1) Client-Blockchain Network Architecture: This is a pattern where the end user executes the transaction by directly communicating with the Ethereum Blockchain. Instead of connecting to an intermediary or a proxy server, the Front-End component directly connects to the Blockchain network and allows the client to execute transactions. Figure 1 illustrates this idea.

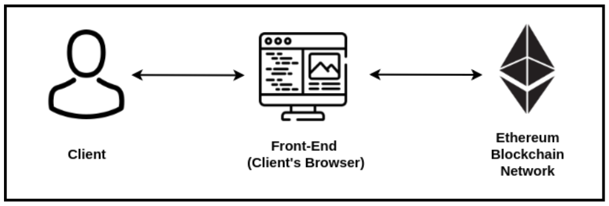

Fig. 1. Client-Blockchain Network Architecture

One of the standard ways for implementing the above-mentioned architecture is to use the MetaMask browser plugin [4]. MetaMask allows users to create new Ethereum accounts and execute Ethereum transactions within their browser. There are multiple benefits of using MetaMask. For instance, it comes with an out-of-the-box user interface that presents the transaction flow in a visual manner to the end user. In addition, there is no overhead during the setup of the plugin, since MetaMask is installed just like any other browser plugin. What is more, by using MetaMask the Ethereum wallet credentials are managed by the client (i.e. stored in the browser) instead of being managed by an intermediary service. Finally, end users need no prior knowledge about the transaction execution flow, since MetaMask provides a high level of abstraction when users interact with the Ethereum Blockchain.

At the same time, MetaMask introduces a fair share of downsides as well. Probably the most noticeable one is that in order for the user to be able to execute transactions and utilize the functions of the Front-End application, the browser needs to have MetaMask installed. This makes the application less interoperable because it is very unlikely that clients visiting the application's website will have MetaMask already pre-installed. Of course, developers can address this issue by checking if the browser already supports MetaMask and if not they can use another provider. However, this is not a very straightforward process since in most cases it requires access to a full Ethrereum node that allows incoming traffic from the client's host. Put simply, users who are visiting the DApp without having MetaMask installed remain in a disadvantageous position.

Another disadvantage is that users using the MetaMask plugin are forced to rely on the Ethereum nodes that the plugin uses under the hood. Although MetaMask is open-source[1] and changing the default Ethereum nodes that MetaMask relies on is possible, this task requires at least somewhat specialized knowledge and it is highly inapplicable for the general public.

Finally, the last drawback of using MetaMask is the storage of the Ethereum account's private key for which MetaMask uses the browser storage. As mentioned above, storing the Ethereum wallet credentials close to the client has obvious benefits and it makes the DApps that use this approach purely decentralized from the standpoint that each user is responsible for managing its own credentials. However, storing sensitive data inside the browser leaves opportunities for security exploits when visiting unknown websites with the same browser (e.g. using cross-site scripting attacks [5]). To address this issue, there are other solutions (e.g.Parity [6], MyEtherWallet [7]) that allow the user to manage and store its wallet credentials on the local disk but they have their own fair share of up- and downsides.

All in all, knowing and assessing the strengths and weaknesses of MetaMask is very important especially considering how widely used MetaMask is for building client rich applications with a direct communication link to the Ethereum Blockchain network. To summarize, regardless of the individual technology used to achieve the desired architecture, there is a set of general architectural design considerations that remain always valid.

If we take a step back from the Blockchain domain, we can logically divide the architectural pattern presented in Figure 1 into two distinct layers similar to what is commonly known as two-tiered architecture [8]. Figure 2 emphasizes this idea.

Although in practice this model is not completely accurate because the Blockchain is fully replicated, decentralized, P2P system, it is still a useful abstraction that can be used to illustrate the main advantages and disadvantages of the architectural design model. These can be summarized as follows:

Advantages:

- Reduced development overhead and cost
- Shorter development cycle
- Enhanced user experience (UX)

Disadvantages:

- Most of the business logic concentrated in one component (i.e. Front-End)

[1]https://github.com/MetaMask

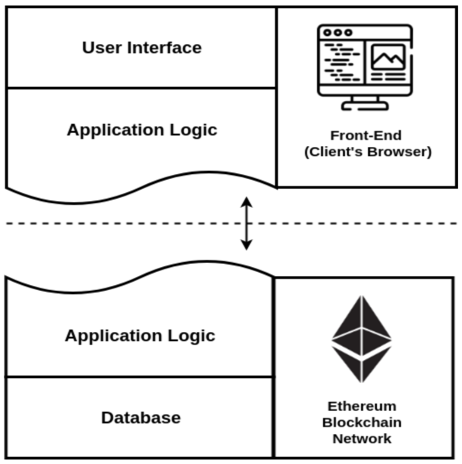

Fig. 2. 2 Tier architecture model within the context of the Ethereum Blockchain (Adapted from [8])

- Scalability limitations with the Front-End node as a limiting factor
- Limited separation of concerns

What is immediately noticeable is that most of the benefits are related to ease of use, development effort and user experience. The reason for that is that by establishing a direct communication between the client's browser and the Ethereum Blockchain network, the developers behind the decentralized application are not required to implement and manage an intermediary node. In addition, in the case where MetaMask is used, users can rely on the interfaces that the plugin itself provides, which results in even less development overhead.

However, despite the operational and user experience benefits, developing a decentralized application which directly communicates with the Ethereum Blockchain network makes developing a DApp with more complex functionality a challenging task. The reason is that most of the business logic of the application with this architectural pattern has to be handled by the Front-End which does not scale well for business logic operations.

2) Client-Proxy-Blockchain Network Architecture: In the subsection that follows the main goal will be to summarize the ideas and properties of another architectural design where instead of communicating directly with the Blockchain network, the client sends a requests to an intermediary/proxy node which in turn forwards them to the Ethereum Blockchain network.

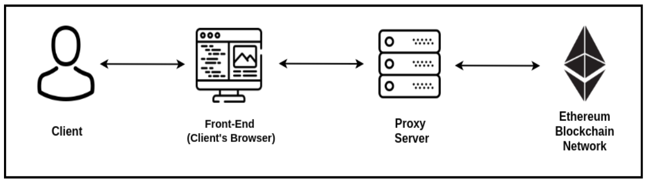

Fig. 3. Client-Proxy-Blockchain Network Architecture

This setup is depicted in Figure 3. The main idea here is that the transaction execution logic is relocated from the Front-End to the intermediary proxy server. In other words, the intermediary node serves the purpose of executing Ethreum transactions on behalf of the end user. One way to achieve this type of functionality on a server node is to use the web3.js library[2]. Web3.js supports a large number of API's that allow the proxy server to not only sign transactions[3] and create new Ethereum accounts[4] on behalf of the user, but it also provides API's for deploying Smart Contracts[5] on the Ethereum network.

Within the context of this architectural design, there are some benefits and drawbacks that become immediately noticeable:

Advantages:

- Better Separation of Concerns (SoC)
- Support for wider range of functionalities (e.g. compile[6] and deploy Smart Contracts)
- Better load distribution among the components (business logic off-loaded from the client to the proxy)

Disadvantages:

- Increased maintenance and development costs
- Scalability limitations with the intermediary node as a limiting factor
- Storing Ethereum account credentials away from the user
- Decreased performance due to network delays

The first major advantage of this architecture, as listed above, is that it allows for a better Separation of Concerns. In other words, each of the components is responsible for its own set of functionalities and there is little to no overlap with the responsibilities of the rest of the components of the decentralized application [9]. This means that the load on the system could be distributed more evenly across all components rather than overloading one single component (as it is the case with the Client-Blockchain network model presented above). In effect, there is more room for making the complexity of certain operations transparent for the end user which means that the decentralized applications supporting this architectural design are able to support more functionalities (e.g. the intermediary node can compile and deploy Smart Contracts without affecting the user experience).

In terms of disadvantages, the first noteworthy limitation of this architectural design is that it introduces more operational costs. In essence, having more components not only increases the duration of the development cycle but it also requires additional server management and maintenance. These are all factors that not only affect the time it takes for an application to be deployed in a production environment but it also increases the monetary cost of the project.

In addition to increased development and maintenance overhead, the discussed architecture presents certain performance limitations as well. In particular, the intermediary node could potentially be a bottleneck and/or a single point of failure. This downside should be considered thoroughly, since attacks

[2]https://web3js.readthedocs.io/en/1.0/
[3]https://web3js.readthedocs.io/en/1.0/web3-eth-accounts.html#signtransaction
[4]https://web3js.readthedocs.io/en/1.0/web3-eth-accounts.html#web3-eth-accounts
[5]https://web3js.readthedocs.io/en/1.0/web3-eth-contract.html#deploy
[6]https://github.com/ethereum/solc-js#readme

targeting single-point-of-failure components (e.g. Denial of Service attacks [10]) might cause server failures and disrupt the communication flow of the whole system. By the same token, due to the presence of an additional component, the performance of the decentralized application would be affected by larger network delays caused by the additional round trip times (RTT) involved in the interaction among the components.

Finally, with this architectural design the Ethereum wallet credentials are generated and managed by the proxy node. In other words, with this architecture, when a client wants to execute a Blockchain transaction, the proxy node performs the transaction on behalf of the user by using the account credentials. This strategy works only for environments where there is an established trust between the intermediary node and the rest of the participants in the system (e.g. private Blockchains). Therefore, the applications of this approach are more restricted.

III. Case Study: A Blockchain-based Energy Trading Platform

The current section aims at providing a comprehensive overview of the architecture of an energy trading platform developed at Fraunhofer FOKUS and can serve as an example of the Client-Proxy-Blockchain Network architectural model presented in section 2. The presented architecture was developed in the scope of a thesis [11] supervised in a joint effort involving the Technical University of Berlin and the Fraunhofer FOKUS institute.

The content that follows describes in detail the components that deliver the main functionality on the platform. In particular, the section opens up with an introduction of each component and its main purpose. Immediately after that, the section continues with a discussion and an overview of the main challenge caused by the architectural design decisions and the way in which this challenge was addressed.

A. Components

Figure 4 illustrates the underlying components of the energy trading platform. Fundamentally, the primary roles in terms of functionality are taken by the Front-End, the Authentication Service, the Blockchain Service and the Infrastructure Gateway. These four components are connected to each other through business logic that includes authentication as well as Blockchain-related operations.

1) Front-End: In the simplest terms, the Frontend component handles what the end users sees in the browser. This means that it is responsible for handling all interactions between the user and the energy trading platform. Creating a new user account, using a existing credentials to login or executing an energy transaction are a couple of example operations that the Frontend component handles on behalf of the user. These operations often times involve complex message exchange among the components on the platform (Figure 4) and therefore the Frontend should handle them transparently [8].

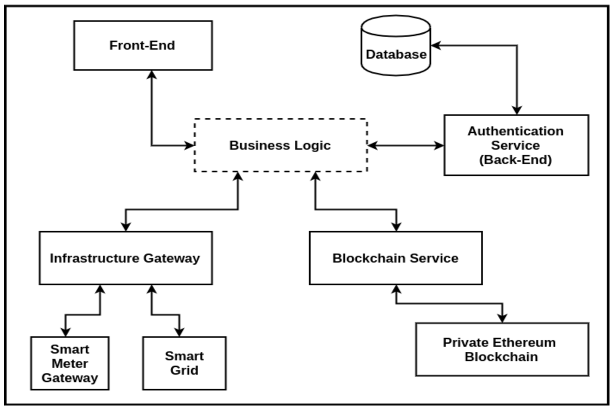

Fig. 4. Energy Trading Platform Components (As described in [11])

2) Authentication Service: The Authentication Service, as the name suggests, handles the authentication and authorization on the platform. This includes the verification of user credentials, issuing and validating authorization tokens, as well as the management of the communication with the NoSQL database where user accounts are stored. Additionally, besides validating the authorization tokens itself, the Authentication Service also maintains a REST endpoint for sharing a public key that allows other services on the platform to verify if a specific token was originally signed by the Authentication Service.

3) Blockchain Service: The Blockchain Service serves as an intermediary between all components on the platform and the private Ethereum Blockchain network hosted by Fraunhofer FOKUS. Its primary goal is to deliver proxy-like functionality, where each Blockchain-related operation is interfaced with a REST endpoint which allows the rest of the modules on the energy trading platform to seamlessly execute Blockchain transactions.

In addition to handling incoming HTTP requests from the modules on the platform, the Blockchain Service is also responsible for compiling and deploying Smart Contracts to the private Ethereum network and also for creating and managing Ethereum accounts on behalf of the users on the platform. This means that all users have two accounts - one managed by the Blockchain Service used for executing Blockchain transactions and one managed by the Authentication Service, where all the personal user data is stored. However, as mentioned earlier in the Front-End subsection, from the user's perspective the explicit creation and integration of the two accounts is handled ”silently” behind the scenes.

4) Infrastructure Gateway: The component described in the current subsection presents functionality that lies outside of the scope of this paper. However, the general ideas presented below are still valid.

Similar to how the Blockchain Service handles the Blockchain-related operations, the Infrastructure Gateway is responsible for dealing with all grid-related operations. This includes but is not limited to registering new smart meters and collecting smart meter readings. With this in mind, it is important to clarify that theoretically the platform can manage the collection of smart meter readings even without a separate component dedicated to this task. However, delegating the grid-related operations to another module might lead to complicated code base, bottlenecks, single point of failures and poor performance overall. That being the case, using the Infrastructure Gateway component as link between the Smart Grid and the Blockchain network makes a lot of sense from architectural and implementation standpoint.

B. Main Challenges

The primary purpose of this section is to describe in more detail the main challenge that was encountered with the current architecture of the energy trading platform. In particular, the main question that the current sections aims at answering is:

What approach can be used for integrating the Ethereum wallet credentials with the user accounts maintained in a traditional SQL or NoSQL store[7]*?*

Integration of Accounts

One of the most notable challenges faced during the development of the energy trading platform described above was the integration of the accounts stored in the database[8] and the Ethereum accounts used for executing Blockchain transactions. The reason why this type of account integration is challenging is because to sign a Blockchain transaction the user needs access to the private key of the Ethereum account. However, the only entity that has access to the private key of the Ethereum account on the energy trading platform is the Blockchain Service. Since the private key cannot be transmitted over the network or shared with any of the rest of the components on the platform, there should be a mechanism that allows the end user to initiate Blockchain transactions by sending a request to the Blockchain Service, however without transfering any Ethereum account credentials. In other words, when a client makes a request from the Front-End (i.e. a web application accessed over the browser) the Blockchain Service must be able to verify the identity of the user before executing the requested transaction. That being the case, the identity verification process by default should require a *claim* transfer from the Front-End component to the Blockchain Service. More precisely, when a user sends a request to the Blockchain Service the request should contain a verifiable piece of data (i.e. *the claim*) that contains information about the user identity. Since the Blockchain Service executes a transaction on behalf of the user *only* after its identity is confirmed, it is safe to say that the integration of the two types of accounts is an integral part of the transaction initiation process.

With this in mind, the communication flow during transaction initiation can be divided logically into two distinct phases: *user sign-up* (where the user creates a new Ethereum wallet and a new account with personal data) and *transaction execution process* (where the Blockchain Service verifies the user credentials and executes the Blockchain transaction on their behalf). These two phases are the focus of the discussion that follows:

1) Phase 1 (Sign-up): At a first glance, there is not an obvious reason for linking the user sign-up to the transaction initiation process. However, as mentioned above, executing Blockchain transactions with the current architecture requires the Front-End component to provide a form of identification with each transaction initiation request it sends. There is variety of ways for generating pieces of data that can be used for authorization purposes. The one used throughout the development of the energy trading platform is JSON Web Token (JWT) [12]. In the most simplest terms, JWTs are tokens encoded as a Base64 strings that contain a header, a payload and a signature [13]. The function that generate¡s the signature uses the payload as an input which makes it nearly impossible to successfully tamper with the data inside the token. Additionally, JWTs can be signed both using symmetric and asymmetric approaches [14].

Figure 5 depicts the process of creating a new account on the energy trading platform and the generation of the aforementioned JSON Web Token.

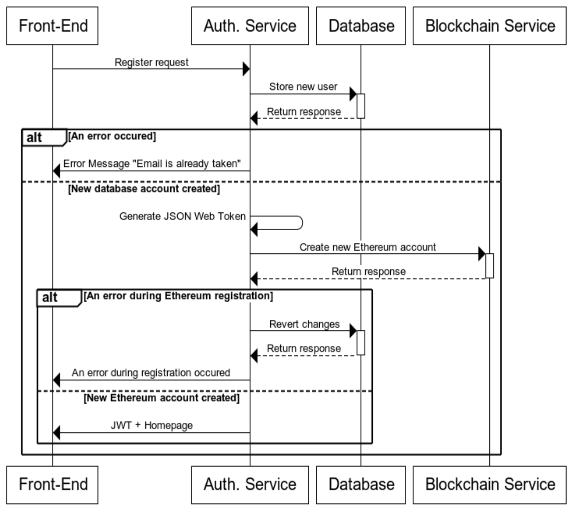

Fig. 5. Sign-up communication flow on the energy trading platform (Adapted from ...)

In the first step of the registration flow, the user sends a request to the Authentication Service component. Immediately after the request is received, the Authentication Service forwards a query to the database which in turn creates a new user account. After the new account is stored in the database, the Authentication Service generates a JSON Web Token (JWT). The JWT's payload contains the username, email and a database id that uniquely identifies the user. The newly generated token is then signed with the private key of the Authentication Service. It is important to mention here that the energy trading platform uses asymmetric cryptography (in particular the RSA algorithm [15]) to sign and verify the tokens.

Going back to Figure 5, in the step called "Create new Ethereum account" the Authentication Service sends a request to the Blockchain Service to generate new Ethereum account

[7]The corresponding store is architecturally integrated with the Authentication Service component

[8]The NoSQL database integrated on the platform is used for storing off-chain personal user account information. This information is used primarily by the Authentication Service and the Front-End components

credentials. In turn, Blockchain Service stores a key-value entry in an in-memory database[9]. The key-value pair uses the unique database id generated by the database as a key, and the Ethereum account credentials as a value. Therefore, by providing the user's database *id*, the Blockchain Service can retrieve the Ethereum account credentials.

After generating the Ethereum account, the Blockchain Service sends back a response to the Authentication Service, which then returns the newly generated JWT to the client and that serves as an indication that the registration process was successful. Executing the sign-up flow in the described manner, serves as an initial setup used for performing Blockchain transactions later on in phase 2.

2) Phase 2 (Executing Blockchain Transactions): Figure 6 illustrates the steps involved in the execution of Blockchain transactions on the energy trading platform. It is important to clarify here that the following description ¡assumes that at the moment of the first issued request, the Blockchain Service has already acquired the public key used for the verification of JWT from the Authentication Service.

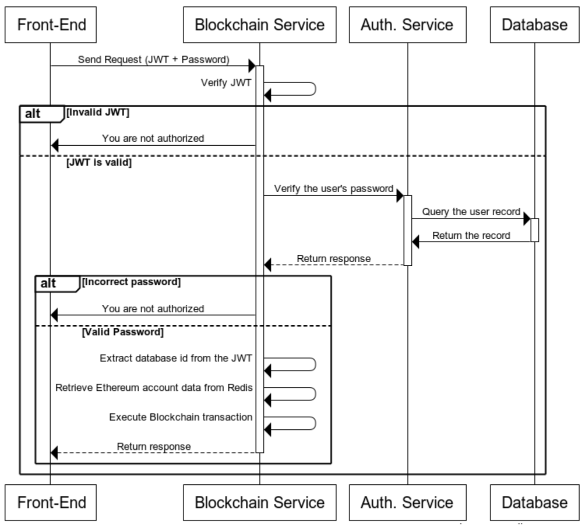

Fig. 6. Execution of Blockchain Transactions on the energy trading platform

In the first step of the flow presented in Figure 6 the Front-End component sends a request to the Blockchain Service for performing a Blockchain transaction. Attached to the request header, the client sends a JSON Web Token and attached to the request body, the client sends the account password. In turn, the Blockchain Service verifies the JWT by using the public key of the Authentication Service. In addition to the validity check of the token, which proves that the JWT was signed by the Authentication Service, the Blockchain Service executes a second authorization check. In particular, the Blockchain Service asks the Authentication Service if the received password belongs to the owner of the JWT. If that is the case, the user has successfully presented two types of proof of identity - on one hand the signed JSON Web Token that includes the identity claim and on the other the account password which only the account holder should have access to. After successfully verifying the authorization rights of the client, the Blockchain Service extracts the database id from the JWT's payload. This id, as mentioned in *"Phase 1"*, serves as the key for retrieving the Ethereum account data stored in the in-memory database. After querying the Ethereum account credentials, the Blockchain Service uses them to sign the transaction on behalf of the user. Furthermore, after performing the Ethereum transaction, the Blockchain Service returns a response to the end user with the result of the initial request.

[9]The main reason behind using an in-memory database is maintaining control over the Ethereum account credentials on one machine while at the same time achieving a high level of performance

C. Discussion

In summary, by using the two phases described above, the user can successfully perform Ethereum transactions in an architectural model where an intermediary node connects the Front-End component to the Ethereum Blockchain network. What is more, because of the two-phase setup, there is a secure integration between the standard database account and the Ethereum wallet of the end users. That being the case, even though the described process includes two phases, in general, *"Phase 1"* is executed only once (during the user sign-up), and it sets up the authorization mechanisms that *"Phase 2"* utilizes during the execution of Ethereum transactions.

The analysis of the main outcomes of this work clearly show that when it comes to the development of decentralized applications which integrate the Ethereum Blockchain in the technology stack, there are inherent architectural trade-offs that should be considered. Some of the areas that are strongly affected by the architectural design of the decentralized application are scalability, performance, security and user experience. That being the case, choosing the right architecture for a DApp is not a trivial task and requires careful and meticulous analysis [11].

IV. Conclusion and Future Work

In this work, we present two architectural designs relevant for the development of decentralized applications (DApp) using the Ethereum Blockchain technology. The practical application of one of the approaches is illustrated by examining the architecture of an energy trading platform developed at Fraunhofer FOKUS. Within this context, we address one of the main challenges faced during the implementation of the platform and namely - integrating the standard database accounts with the Ethereum wallet credentials.

With regards to future work, one of the most significant challenges in terms of impact and severity of consequences is security. Although we briefly touched this topic, it is one that deserves a more in-depth analysis and has a large impact on the system. Therefore, security and its architectural implications on DApps are an area that will require further future evaluation. Within this context, it is an ongoing effort to implement and support mechanisms that ensure that the energy trading platform presented in this work is protected against security attacks (e.g. DoS attacks [10], Sybil attacks [16], etc.). Working in this direction would provide us with better insights into the potential ways in which certain security mechanisms affect the performance, the user experience and the structure of most decentralized applications.

Acknowledgement

This work was funded by the Federal Ministry for Economic Affairs and Energy under the funding number 03SIN514. The authors are responsible for the content of this publication.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," https://bitcoin.org/bitcoin.pdf, 2008.

[2] M. Liserre, T. Sauter, and J. Y. Hung, "Future energy systems: Integrating renewable energy sources into the smart power grid through industrial electronics," *IEEE Industrial Electronics Magazine*, vol. 4, no. 1, pp. 18–37, March 2010.

[3] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform," https://github.com/ethereum/wiki/wiki/White-Paper, 2013.

[4] Metamask. brings ethereum to your browser. Accessed 22-May-2019. [Online]. Available: https://metamask.io/

[5] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic, "Noxes: A client-side solution for mitigating cross-site scripting attacks," in *Proceedings of the 2006 ACM Symposium on Applied Computing*, ser. SAC '06. New York, NY, USA: ACM, 2006, pp. 330–337. [Online]. Available: http://doi.acm.org/10.1145/1141277.1141357

[6] Parity ethereum. the fastest and most advanced ethereum client. Accessed 23-May-2019. [Online]. Available: https://www.parity.io/ethereum/

[7] Ethereum's original wallet. Accessed 23-May-2019. [Online]. Available: https://www.myetherwallet.com/

[8] A. S. Tanenbaum and M. V. Steen, *Distributed Systems: Principles and Paradigms*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001.

[9] M. Patterns and P. Team, *Microsoft Application Architecture Guide, 2nd Edition (Patterns and Practices).* Microsoft Press; Second edition, 2009, ch. Key Principles of Software Architecture.

[10] K. Elleithy, D. Blagovic, W. Cheng, and P. Sideleau, "Denial of service attack techniques: Analysis, implementation and comparison," *Journal of Systemics, Cybernetics and Informatics*, vol. 3, pp. 66–71, 01 2006.

[11] D. Rangelov, "Design and development of a front-end and business logic components for blockchain utilization in the energy domain," Master's thesis, Technical University Berlin, Berlin, 2 2019.

[12] M.Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," Internet Requests for Comments, RFC 7519, 5 2015. [Online]. Available: https://tools.ietf.org/html/rfc7519

[13] Introduction to json web tokens. Accessed 22-May-2019. [Online]. Available: https://jwt.io/introduction/

[14] H. Delfs and H. Knebl, *Introduction to Cryptography. Principles and Applications.* Springer, Berlin, Heidelberg, 2015.

[15] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978. [Online]. Available: http://doi.acm.org/10.1145/359340.359342

[16] J. R. Douceur, "The sybil attack," in *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, ser. IPTPS '01. London, UK, UK: Springer-Verlag, 2002, pp. 251–260. [Online]. Available: http://dl.acm.org/citation.cfm?id=646334.687813