A Blockchain Approach to Messaging Applications

Brooke Ryan
Department of Computer Science
University of California, Irvine
Irvine, CA
brooke.ryan@uci.edu

Yotam Golan
Department of Computer Science
University of California, Irvine
Irvine, CA
ygolan@uci.edu

Sherif Abdelkarim
Department of Computer Science
University of California, Irvine
Irvine, CA
abdelkas@uci.edu

***Abstract*—Due to an increasing demand for online privacy and security, decentralized forms of communication have become increasingly relevant. As such, blockchain presents itself as a viable networking paradigm over which to conduct such a messaging application.**

In this work, we present a prototype of a peer-to-peer messaging application that leverages blockchain technology to provide an anonymous and secure messaging application. We utilize a hybrid design of blockchain system and user coordination, while using locally-implemented encryption and message storage. The benefits of this design are two-fold—the computationally expensive tasks of message storage and encryption are handled off-chain to both reduce costs *and* minimize security vulnerabilities. As such, our application design we present is optimally both decentralized and isolated.

As demonstrated experimentally, our application design maintains low gas prices even as the number of users in the system increases, which is particularly novel. More broadly, the design that this work proposes may serve as a template for potential future applications of commonly available consumer-facing technologies, similarly seeking to build a low-cost, decentralized, blockchain-based application.

I. Introduction

The internet over the past decade has been trending towards an increasingly centralized, and increasingly monolithic, state. As of 2021, four companies were responsible for 67% of the global cloud-computing infrastructure [1]. While this centralization has provided several benefits including an easier and more cohesive experience for most users, it has exposed the initially decentralized internet to several single points of failure vulnerable from within and without.

On October 4th 2021, while performing routine updates on router firmware, a misconfigured setting resulted in Facebook disconnecting itself from the Border Gateway Protocol, rendering its networks unreachable by outside machines [2]. This led to a cascading collapse of its own network and the large portion of the internet that relied upon them. For the length of this down-time, all applications which required contacting Facebook servers ceased to function properly. These services included WhatsApp and Facebook messanger, the two largest messaging applications, which combined boast over 3.3 billion monthly users and are crucial communication tools in much of the world [3].

The increase in centralization has also presented a number of privacy issues, and increased the ability of governments to access citizen data. While WhatsApp, the most worlds most popular messaging application, purports to be end-to-end encrypted, a recent Freedom of Information request revealed it shares extensive information with third-parties when requested [4]. Despite the encryption, Whatsapp can provide Subscriber Data, Message Sender-Recieved Data, Contacts, Data/Time information, and more.

Whether by outside influences such as power outages or warrants, or internal influences such as configuration errors or malicious actions, any service which relies upon a single point of failure is inherently at risk. Instead, we argue that the future of secure and private communications lie in distributed and fault-tolerant systems that minimize single points of failure and inherently resist malicious actions.

In this work, we present a novel messaging application that leverages a hybrid blockchain-local architecture. Our architecture utilizes blockchain for user identity and coordination aspects of messaging, in order to address the aforementioned issues of centralized servers and ensuring network reliability. Working to mitigate the high cost of on-chain computation, we utilize a peer-to-peer model to encrypt, decrypt, and store messages locally. Our novel messaging application architecture integrates the advantages of privacy, security, and decentralization seen in previous blockchain-based messaging applications, while mitigating the computational complexity and high costs by simultaneously leveraging off-chain operations.

The remainder of the paper is organized as follows. In Section II, we present the relevant works. In Section III, we introduce the blockchain, peer-to-peer messaging, and encryption techniques used in this work. In Section IV, we discuss the design of the system. In Section V, we describe the experiments performed. In Section VI, we present our conclusion. In Section VII, we provide an Appendix, with information on how to access the code used to run the experiments.

II. Related Works

A. Blockchain-Based Messaging

While conducting messaging over blockchain is a relatively new endeavor, a few such applications have emerged in recent years. Bitmessage [5] is a trust-less, decentralized peer-to-peer messaging application. In this system, the users don't need to exchange any data except a 36 character address to ensures security. To prevent spam, the authors use proof-of-work, this causes the application to waste computational power and makes it less likely to be used in practice.

In Bitmessage Plus, a further iteration of Bitmessage, [6], the authors achieve privacy and anonymity by utilizing the blockchain flooding propagation mechanism as well as asymmetric encryption algorithms. To overcome the computational limitations of Bitmessage, Bitmessage Plus utilizes proof-of-space instead of proof-of-work to prevent spam. Proof-of-space requires a user to dedicate a certain amount of disk space in order to be able to send a message. This reduces the computational cost associated with proof-of-work.

While these applications share the similarity of utilizing the blockchain in the context of user identity and coordination, to our knowledge there does not exist a messaging app that integrates blockchain alongside peer-to-peer in order to reduce computational complexity.

B. *Other Blockchain-Based Applications*

Beyond messaging, there exist relevant applications of blockchain in a variety of domains spanning areas such as social media, crowdfunding, and cyber-physical systems [7]. In BCOSN: A Blockchain-Based Decentralized Online Social Network [8], Jiang and Zhang propose a similar hybrid approach as we present in this paper. BCOSN utilizes blockchain for user identity and certain typical functions of social media, such as post notification or friend requests. Whereas tasks that are more difficult to achieve in a decentralized network are provided by a central server, similar to a traditional design of a social media network.

LikeStarter [9], a blockchain-based crowdfunding application, is structured as a Decentralized Autonomous Organization (DAO) that is built over the Ethereum blockchain. LikeShare certainly benefits from the aforementioned properties of blockchain (security, privacy, decentralization). The authors further exploit the similar structure of the blockchain smart contract and use it directly as the mechanism for crowdfunding. This represents a unique example of the blockchain smart contract being directly leveraged as a feature within the social network.

In cyber-physical systems, computation is often off-loaded to a cloud-based server to maximize efficiency on the embedded device [10]. In BaDS: Blockchain-Based Architecture for Data Sharing with ABS and CP-ABE in IoT, Zhang et al. explore the usage of a permissioned blockchain architecture that replaces the typical cloud model in internet of things (IoT) applications [11]. BaDS leverages a Byzantine fault tolerance mechanism, rather than Proof of Work for its consensus model, while relying on an encrypted server to store excess data.

While the previous works are not specific to messaging, the architectures employed are similar in nature to the blockchain architecture presented in this work. However, due to our minimalistic approach, our presented design is significantly less costly than the models presented.

III. Background

A. *Ethereum Blockchain*

We chose to run our application on the Ethereum blockchain because of its wide availability for open-sourcing and developing decentralized applications [12]. The Ethereum blockchain builds upon the blockchain technology introduced in the original 2008 Bitcoin paper [13]. As described by Li and He, blockchain fundamentally exhibits five key characteristics as a system: it is decentralized, trustless, tamper-proof, transparent, and secure [14]. While the Ethereum blockchain maintains these five key properties, it further evolves beyond the "distributed ledger" model found in Bitcoin to a more robust and flexible distributed state machine. Applications-wise, this means that Ethereum can be used in development for cases beyond simply managing cryptocurrency accounts and balances. This capability is employed by using a smart contract, which is a cryptographically secure function that executes automatically once its specified conditions are realized [15]. The Ethereum distributed state machine, represented by the Ethereum Virtual Machine (EVM), is fully deterministic and Turing-Complete.

Later in the experiments, we will be discussing the optimization of gas prices, which is distinct from the unit of actual cryptocurrency Ether. In Ethereum, "gas" is defined as one unit of computation [16]. Users in the Ethereum system are required to pay gas for each computational step that the execution of the smart contract necessitates. The actual price of gas is a small fraction of the cost of Ether.

B. *Peer-to-Peer Messaging*

Peer-to-Peer messaging, or P2P, is a distributed application architecture that partitions tasks or workloads between peers, wherein each "peer" acts as a server to other nodes in the network. For the purpose of messaging, it is a simple data sharing method which allows two parties to directly communicate, share, and store information without third-party involvement or a centralized server [17]. This enables functional, secure messaging applications with minimal supporting infrastructure and maximum reliability [18]. Additionally, it allows for message data to be stored entirely locally and transferred securely with encryption.

This has inherent benefits and drawbacks compared to traditional centralized messaging architectures such as Facebook Messenger. Due to the reduced necessary complexity of the architecture, no centralized point of failure, and no third-party presence, Peer-to-Peer's advantages lie in simplicity, reliability, and security. However, as data is stored entirely locally it is more vulnerable to data loss and malicious actions with physical access.

C. *Encryption*

Encryption is a process of encoding information. This process converts an original data, known as *plaintext*, to a coded version of the data, known as *ciphertext*, that makes it unintelligible to an unauthorized person. A certain key is needed to revert the *ciphertext* back into *plaintext*, this is known as decryption or deciphering [19]. A *ciphertext* can be deciphered without using the key. However, in a well designed encryption protocol, this would require significant computation, which makes it infeasible in practice.

On its own, encryption does not prevent an attacker from getting hold of the data, but it makes the data unreadable without having access to the key.

Encryption is an integral part of blockchain technology [13], as it is used for generating the hashes that connect the blocks to each other. In our setup, encryption is additionally needed to protect the users' information (i.e. phone numbers, IP addresses, etc.) from being accessed without proper authorization.

Generally, there are two methodologies of encryption: symmetric and asymmetric: Symmetric encryption utilizes the same private key for both encrypting and decrypting a message; whereas asymmetric encryption uses a public key to encrypt the message, and a private key to decrypt the message [20]. Rivest-Shamir-Adleman (RSA) encryption is beneficial due to its design that prevents reverse-engineering and thus is highly secure, however the complexity of generating the key and encrypting is relatively slow [21]. RSA encryption, however, has several differing levels of security depending on if 512, 1024, 2048, or 4096 bit encryption is used. Each of these levels is a trade off of key length vs encryption quality, and represents the balance of storage costs and security. Elliptical Curve Cryptography (ECC) computes keys using elliptic curve equations [22]. It is more energy-efficient than RSA, however it increases the size of the encrypted text [23]. Ciphertext-Policy Attribute-Based Encryption (CP-ABE) is a method of symmetric encryption specifically designed for use in a distributed system [24], which grants users with specified attributes access to the key. However, it incurs a higher time cost than RSA [8]. As we will describe in the Design section of this paper, the user will be free to implement whichever kind of encryption algorithm they choose because it will be implemented locally. However, the tradeoffs of these common kinds of encryption are essential to understand if the user chooses to utilize on-chain storage, because of their effects on gas prices of the system.

IV. Design

A. System And Security Model

Our messaging system contains four major components which together allow for encrypted, log-less, distributed messaging between two users. There is the sender, the receiver, the on-chain smart contract, and the on-chain storage system.

The sender is responsible for retrieving the public encryption key of the relevant receiver from on-chain storage and encrypting their message locally. This encrypted message is then passed on to the smart contract. The smart contract then serves to relay the sent message to the appropriate end node, the receiver. The receiver must then use their locally stored private key to decrypt the message.

This system is designed to minimize the trust placed in any involved third party. The sender must interact with the smart contract and storage, and will never directly interact with the receiver. This opens the possibility for a man in the middle attack and for a malicious actor to intercept communications. To help prevent this, un-encrypted messages are only ever stored locally, and end-to-end encryption is leveraged when the messages are transmitted. The sender must, however, assume that the public key provided and the smart contract are not malicious. Because no logs are generated for message transmission, it is impossible for the sender to definitely determine who ultimately received the message or if it was decoded correctly.

The receiver interacts only with the smart contract, and must also trust it is not malicious. The smart contract provides the sender's identifier and the message data. The receiver must assume that both are correct and unaltered. To help mitigate potential attacks, the sender's identifier is included in the message data, similarly encrypted. This would prevent simple swaps to identity, but would still leave the message vulnerable to malicious action if the storage provided the wrong key, and the message was read and re-encrypted en-route. This risk is somewhat mitigated by the open-source nature of the blockchain, and the ability to inspect the code for both.

The smart contract serves to route encrypted data between nodes, and does not rely on trust for either. If the sender or receiver are malicious, then the limit of their attacks would be to send incorrect data and identifiers. While potentially problematic to the receiver node, it would have little effect on the smart contract.

Storage maintains publicly visible keys and identifiers, and must assume that provided data is non-malicious. A malicious actor could provide incorrect keys for identifiers, in a bid to allow a man-in-the-middle attack. This is mitigated by only allowing keys to be submitted by their respective identifiers.

B. System Overview

Our messaging system seeks to leverage end-to-end encrypted peer-to-peer messaging, while leveraging on-chain technology to remove the need for a centralized server and the inherent downsides that it entails. The system will thus contain four major components: the sender, the receiver, the on-chain smart contract, and the storage.

The sender and receiver will both be end-points in our system, each will be a distinct address identified by a unique marker. They will be responsible for storing, encrypting and managing their message data locally, and will only interact with the rest of the system when they wish to send messages. This will allow for the minimal usage of expensive on-chain processing, while providing a security guarantee inherent in eschewing centralized storage.

The smart contract will the be central relay hub of our network. It will handle accessing and providing public keys to message senders, as well as routing the encrypted data to its intended end point. In these ways, it will function much the same as a more traditional messaging service such as signal, but with additional redundancy due to its distributed nature.

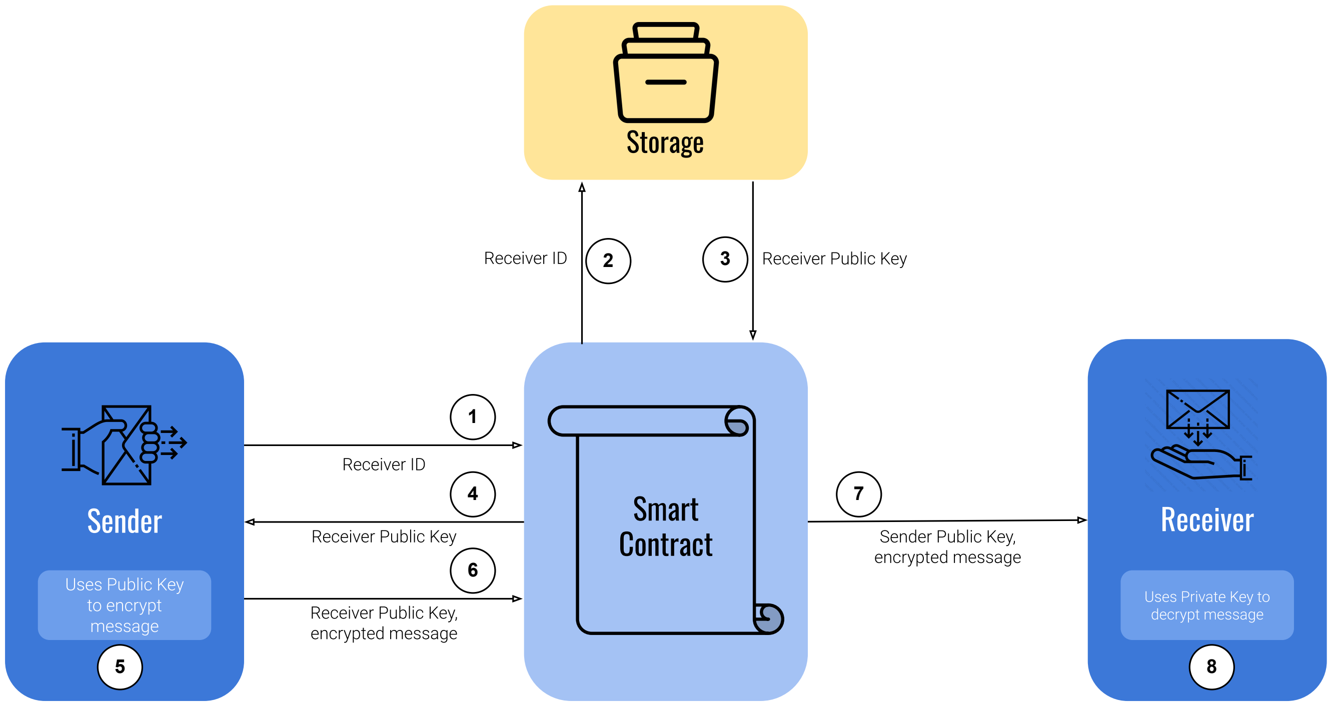
Storage
Receiver ID
2
3
Receiver Public Key
Sender
Uses Public Key to encrypt message
5
1
Receiver ID
4
Receiver Public Key
6
Receiver Public Key, encrypted message
Smart Contract
7
Sender Public Key, encrypted message
Receiver
Uses Private Key to decrypt message
8

Fig. 1. This illustration shows the use cases of sending & receiving a message. The steps of operation are numbered from 1 to 8.

The storage will be responsible for maintaining public keys and the encrypted identifiers used to mark unique users so as to allow message routing. This can either be on or off chain.

To process a message, the sender will first query the smart contract for the public key of the desired receiver, the smart contract will then retrieve the public key from storage, and return it to the sender. The sender will use the provided key to encrypt the message locally, before again querying the smart contract. This will involve passing the encrypted message and receiver identification to the smart contract, which will then route the encrypted message and the sender identification to the receiver. The receiver may then decrypt the message locally.

C. Design Details

In this section we will go over the system design details. This section is divided into two parts: Components, where we shall delineate each component in our system in detail, and Use Cases, where we demonstrate the main use cases our system will be able to handle.

Components:

This system consists of the following four main components: sender, receiver, smart contract, and storage.

Sender: the sender is a user that wants to *send a message*. A sender initiates the message-sending operation by sending a request to the smart contract. This request includes the receiver's unique identifier, which represents the receiving address of the message. This will be either a phone number or an email address. Next, the sender receives a response from the smart contract, which includes the receiver's public key. The sender then uses the receiver's public key to encrypt the message. After the message is encrypted, the sender sends the encrypted message along with the receiver's public key to the smart contract.

Receiver: the receiver is the node that *receives the message* from the sender through the smart contract. The receiver receives the encrypted message along with the sender's public key. After that, it decrypts the message using its own private key, which would reveal the original message sent by the sender.

Smart Contract: the smart contract manages the interactions between users and each other and between users and storage. This smart contract can receive three kinds of requests: *create_a_user* request, *get_public_key* request, and *send_a_message* request.

create_a_user: A properly formatted *create_a_user* request will include the user's unique identifier and the user's public key. First the smart contract verifies that the request is properly formatted. If the request is not properly the smart contract ignores it. If it is properly formatted the smart contract takes the ID, public key pair and passes it on to the storage, so the data can later be retrieved.

get_public_key: A properly formatted *get_public_key* request will include the receiver's unique ID. Similar to *create_a_user* request, if the request is not properly formatted, the smart contract ignores the request. If it is properly formatted the smart contract uses the provided receiver ID to query the storage in order to get the receiver's public key. After the storage responds back with the receiver's public key the smart contract responds to the sender's request with the receiver's

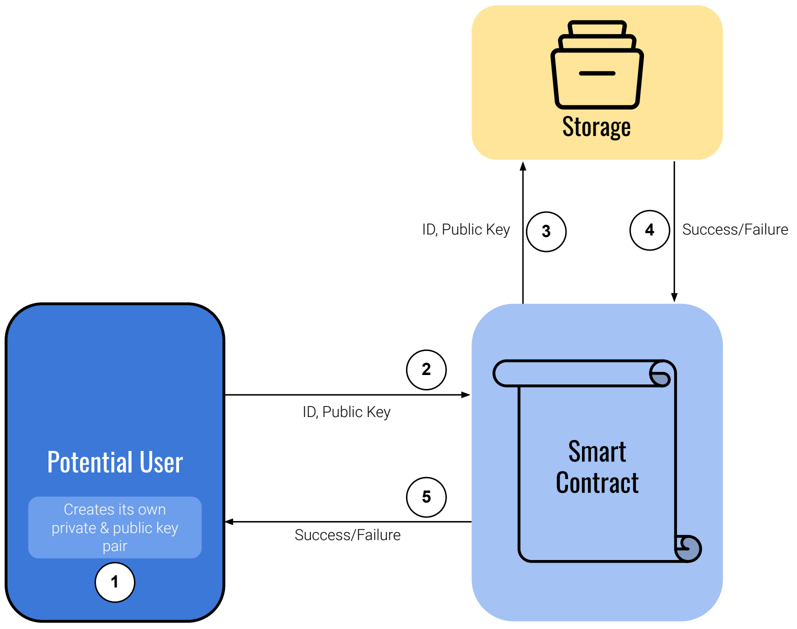

Fig. 2. This illustration shows the use cases of creating a new user. The steps of operation are numbered from 1 to 5.

public key.

send_a_message: A properly formatted *send_a_message* request will include the receiver's public key and the encrypted message. Similar to the other two requests, if the request is not properly formatted the smart contract ignores it. If it is properly formatted, the smart contract sends the encrypted message to the receiver along with the sender's public key.

Storage: The main purpose behind the storage is to store data that matches users' identifiers with users' public keys, without having to store it on-chain, which significantly reduces the cost. The storage is only accessible by the smart contract, meaning the users have no direct access to the storage. The storage can receive two types of request: *POST*, and *GET*. *POST*: the *POST* request is received when a new user is created by the smart contract. The smart contract send a *POST* request to the storage including the user's ID and public key pair. The storage saves this data. *GET*: the *GET* request is used when the smart contract has the user's ID and it wants to retrieve the user's public key. So it sends a *GET* request to the storage and the storage uses user's ID as a query to get the user's public key, and then it sends it back to the smart contract.

Use Cases:

Now that we understand in more details each component in our system along with its job, we can now go over how these components interact in each use case in our system.

Sending & Receiving a message: This use case is of a user (sender) sending a message to another user (receiver). It goes as follows:

1) The sender sends a request to the smart contract containing the receiver's ID. This request is basically the sender asking the smart contract for the receiver's public key.
2) The smart contract receives the request and uses the receiver's ID to query the storage for the receiver's public key.
3) The storage replies back with the receiver's public key.
4) The smart contract responds back to the sender with the receiver's public key.
5) The sender uses the receiver's public key to encrypt the message it wishes to send.
6) The sender sends a request to the smart contract containing the receiver's public key and the encrypted message.
7) The smart contract sends the encrypted message along with the sender's public key to the receiver.
8) The receiver uses its own private key to decrypt the message.

Figure 1 illustrates this use case with in visual form.

Creating a user: This use case is of a node wishing to become a user. It goes as follows:

1) The user creates its own public and private key pair locally.
2) The user sends a *create_a_user* request to the smart contract containing its ID and its public key.
3) The smart contract sends a POST request to the storage to add this new ID, public key pair to the database.
4) The storage responds with a success message if the operation was successful
5) If the smart contract receives a success message from the storage within the allotted timeout period, it then responds to the user with a success message, otherwise it will send a failure message.

Figure 2 illustrates this use case.

V. Evaluation

In order to evaluate the effectiveness of the proposed application design, we performed several experiments. The experiments were evaluated in a cluster for each key aspect of our proposed design. The goal of the experiments is to analyze both verify the correctness of the design as well as to evaluate cost metrics.

To implement these experiments, we utilized the solidity programming language and developed on the Remix IDE provided by Ethereum. Solidity is a programming language that is designed for writing smart contracts [25]. The code can be viewed on GitHub, and is also provided in the Appendix.

A. Cost Evaluation

One key aspect of the proposed design is the **optimization of gas prices**. In the following experiments, we show how varying flexible aspects of the design such as on-chain vs. off-chain storage or the number of users in the system vary the number of computational steps involved in execution, and thus the cost. Ultimately, our goal is to optimize gas costs and also demonstrate the trade-offs different aspects of the system have on the overall price involved for a potential user.

In the first experiment, we evaluate the relationship between the number of users in the system (x-axis) and the cost of gas (y-axis). In order to test this, we registered 100 test users with the *Sender.create_a_user* smart contract. For the purposes of this experiment, all the test users will have the same generated key 2048-bit RSA public key. We iterated the following steps until all test users had been registered with the system: We registered 1 user using the *Sender.create_a_user* smart contract request with the public key. For each user registered, we then recorded the gas price. In Figure 3, we show how the number of users registered with the system affects the gas prices. Because the user's public key is stored in the system, we suspect that there always will be a small constant gas price associated with this computational cost. We expect the prices to remain consistent with use, as address and key sizes will not change. The address is a consistent 4 bytes long, and the 2048-bit RSA public key is 398 bytes, with a combined size of 440 bytes to store for each new user. As the combined storage space of both values is 14*32 bytes, using the established cost of 20,000 gas per 32 byte storage we expected to see a gas cost of 280,000 plus additional charges for run-time execution. This was an assumption that proved true, with each user creation costing a consistent 376,923 Gas [Fig. 3], at a current exchange rate of approximately $1.05.

In the second experiment, we evaluate the effect of on-chain vs. off-chain message storage on gas prices. In order to test this, we configured two different environments of our system: one with on-chain storage (provided in the `Storage` smart contract), and another with no provided storage. The environment that provided no message storage is equivalent to off-chain storage for the purposes of this experiment, since the burden of message storage would be on the user. To implement the environment that provided on-chain message storage, we wrote code for a version of the smart contract which took the additional step to store the encrypted message on-chain. For each environment, we performed the following steps iteratively: We utilized the *create_a_user* smart contract in order to create 100 test accounts, 50 senders and 50 receivers each. We then registered 1 "sender" user and 1 "receiver" user each with the application along with their public key (for the purposes of the experiment, each user is assigned the same public/private key pair). Then the sender user sent two identical messages to the designated receiver account. The message sent throughout the course of this experiment is a small, 250 character identical message. For each message sent, we recorded the gas price. We repeated these steps until all 50 test senders had sent their messages. Then, we repeated the steps again, so that each sender test user had ultimately sent 2 messages. In this experiment, the x-axis is represented by number of active users, active being defined as having successfully sent the message. The y-axis represents the cost. In Figure 4, we demonstrate and analyze how this affects gas prices. Our hypothesis for this experiment was that it would cost significantly less to store messages off-chain. We found that storing messages off-chain was significantly cheaper then storing on-chain for all initial messaging, but once on-chain storage was purchased the cost delta was significantly decreased. While the first message required purchasing additional on-chain storage, which is the primary cost sink, all subsequent messages could then leverage this existing storage for a reduced cost.

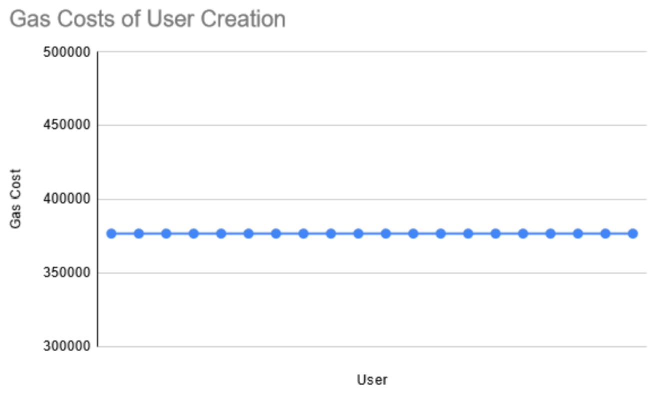

Fig. 3. Cost evaluation experiment 1, analyzing the effect of the number of users in the system (x-axis) and the cost of gas (y-axis). Each point is the cost average of 50 users.

As all written data is publicly available on the Ethereum chain, we encrypt it using public-private RSA encryption to ensure only the intended recipient can access it. In this experiment, we will analyse the gas costs inherent with each level of RSA security (512-bit, 1024-bit, 2048-bit, 4096-bit) by creating 10 users with on-chain storage for each level.

For each of the 10 users at each security level we find the gas cost of user creation, and then sending two identical 50 character messages encrypted with the associated RSA key size. By sending two identical messages we can determine

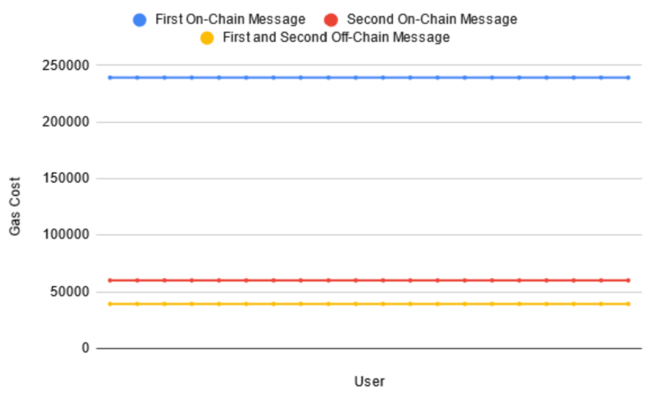

Fig. 4. Cost evaluation experiment 2, analyzing the effect of the off-chain/ on-chain storage (x-axis) and the cost of gas (y-axis).

what the initial costs are to send to a user, and what recurring costs will be for all following communications that take equal or lesser space.

As shown in 5, costs rise significantly with increased key-size. 512-bit user creation cost an average of 181,377 in gas fees, 1024-bit cost 230,264, 2048-bit cost 376,923, 4096-bit cost 645,802. This is to be expected as increasing key sizes increases the size of both on-chain storage demands, and message size. While increasing the initial data footprint and upfront costs, recurring message costs for additional sends remain relatively low.

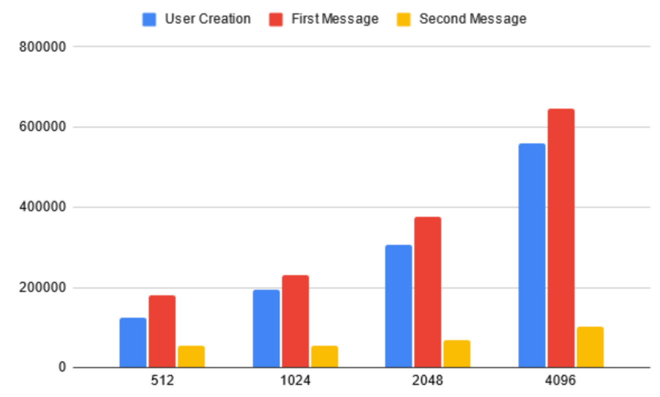

Fig. 5. Cost evaluation experiment 3, studying the effects of RSA key-size (x-axis) and its effect on application gas costs (y-axis).

VI. Conclusion

In this article, we have presented a minimalist, blockchain-based, decentralized design for a messaging application which runs on the Ethereum Virtual Machine. In conjunction with smart contracts, we have used the blockchain as a decentralized mechanism to implement functionalities of messaging applications that are traditionally provided by a centralized server. By implementing a minimalistic design, wherein many of the functions of typical messaging applications such as encryption, decryption, and message storage are provided off-chain, we are able to provide a novel, highly cost-efficient design for a blockchain-based application.

Despite the relatively limited existing options for blockchain-based messaging applications, we believe this to be a significant contribution in this space. Compared to existing blockchain-based consumer-facing technologies, such as those presented in the Works Related section, our application provides a significantly lower cost model. Perhaps the most novel and significant contribution of this presented work in and of itself is it being a cost-accessible blockchain-based application.

In future works, more research should be done on further cost optimizations that can be performed. Preliminary explorations we have performed on altering the structure of the code, such as utilizing Solidity keywords like `public`, `internal`, `calldata` etc., have shown promising potential for further optimization of cost. Future works could explore further optimizations to reduce assembly-line code and thus gas cost. Additionally, more work should be done on the potential user base for such an application, to further inform design decisions and tradeoffs.

References

[1] "Amazon leads 150-billion cloud market." https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/. Accessed: 2022-01-26.

[2] "Why facebook and instagram went down for hours on monday." https://www.npr.org/2021/10/05/1043211171/facebook-instagram-whatsapp-outage-business-impact. Accessed: 2022-01-26.

[3] "Most popular global mobile messenger apps as of october 2021, based on number of monthly active users." https://www.statista.com/statistics/258749/most-popular-global-mobile-messenger-apps/. Accessed: 2022-01-26.

[4] "Jan. 2021 fbi infographic re lawful access to secure messaging apps data." https://propertyofthepeople.org/document-detail/?doc-id=21114562. Accessed: 2022-01-26.

[5] J. Warren, "Bitmessage: A peer-to-peer message authentication and delivery system," *white paper (27 November 2012), https://bitmessage. org/bitmessage. pdf*, 2012.

[6] L. Shi, Z. Guo, and M. Xu, "Bitmessage plus: A blockchain-based communication protocol with high practicality," *IEEE Access*, vol. 9, pp. 21618–21626, 2021.

[7] F. Casino, T. K. Dasaklis, and C. Patsakis, "A systematic literature review of blockchain-based applications: Current status, classification and open issues," *Telematics and Informatics*, vol. 36, pp. 55–81, 2019.

[8] L. Jiang and X. Zhang, "Bcosn: A blockchain-based decentralized online social network," *IEEE Transactions on Computational Social Systems*, vol. 6, no. 6, pp. 1454–1466, 2019.

[9] M. Zichichi, M. Contu, S. Ferretti, and G. D'Angelo, "Likestarter: a smart-contract based social DAO for crowdfunding," *CoRR*, vol. abs/1905.05560, 2019.

[10] S. Sedigh and A. Hurson, "Chapter one - introduction and preface," vol. 87 of *Advances in Computers*, pp. 1–6, Elsevier, 2012.

[11] Y. Zhang, D. He, and K.-K. R. Choo, "Bads: Blockchain-based architecture for data sharing with abs and cp-abe in iot," *Wireless Communications and Mobile Computing*, vol. 2018, pp. 1–9, 11 2018.

[12] R. A. Canessane, N. Srinivasan, A. Beuria, A. Singh, and B. M. Kumar, "Decentralised applications using ethereum blockchain," in *2019 Fifth International Conference on Science Technology Engineering and Mathematics (ICONSTEM)*, vol. 1, pp. 75–79, 2019.

[13] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Cryptography Mailing list at https://metzdowd.com*, 03 2009.

[14] W. Li and M. He, “Comparative analysis of bitcoin, ethereum, and libra,” in *2020 IEEE 11th International Conference on Software Engineering and Service Science (ICSESS)*, pp. 545–550, 2020.
[15] N. Szabo *et al.*, “Smart contracts,” 1994.
[16] V. Buterin, “Ethereum whitepaper,” 2013.
[17] R. Schollmeier, “A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications,” *Proceedings of the First International Conference on Peer-to-Peer Computing, IEEE*, 2002.
[18] S. Ahson and M. Ilyas, “Services, technologies, and security of session initiation protoco,” *SIP Handbook*, 2008.
[19] “Data encryption.” https://www.britannica.com/technology/data-encryption. Accessed: 2022-02-11.
[20] S. Chandra, S. Paira, S. S. Alam, and G. Sanyal, “A comparative survey of symmetric and asymmetric key cryptography,” in *2014 International Conference on Electronics, Communication and Computational Engineering (ICECCE)*, pp. 83–93, 2014.
[21] “Rsa.” www.encryptionanddecryption.com/algorithms/asymmetric_algorithms.html.
[22] V. Miller, “Use of elliptic curves in cryptography.,” pp. 417–426, 01 1985.
[23] D. Mahto and D. YADAV, “Rsa and ecc: A comparative analysis,” *International Journal of Applied Engineering Research*, vol. 12, pp. 9053–9061, 01 2017.
[24] J. Bethencourt, A. Sahai, and B. Waters, “Ciphertext-policy attribute-based encryption,” in *2007 IEEE Symposium on Security and Privacy (SP ’07)*, pp. 321–334, 2007.
[25] https://docs.soliditylang.org/en/v0.4.23/.

VII. Appendix

All code for this project can be viewed in the Blockchain-Project GitHub repository. Deployment instructions and documentation on using the smart contracts can be found on the README. Additionally, the code for the preliminary experiments that were mentioned in the conclusion can be found in the `messaging_app_optimizations.sol` file.