XYZ messenger

Abstract

Defi messenger is a powerful communication platform that allows applications to interact with their users through a messenger interface. Instead of writing and maintaining the code that sends messages to their user, process payments or organize votes, apps can just use messenger's built-in APIs. The underlying protocol is built with full respect for user's privacy and the platform can mathematically prove that it does not have access to customers' data. That allows applications to be safe that their data is not compromised by third-party companies. DeFi messenger has a decentralized structure and the platform does not authorize other apps to grant them the right to interact with users. The structure is email-alike in terms of scalability but unlike email it has a focus on privacy data protection with zero-trust factor. Unlike Slack (or similar apps), DeFi messenger aims to be open-source (both front-end and back-end) and can be forked and modified. It also allows users to get access to their data from their clients which they don't need to trust. So instead of having one Slack for all, each of us can have a personal Slack.

The Protocol

The protocol was designed as an evolution of Web 2.0 application design and a solution for building trustless decentralized applications. It is a high-level logical abstraction over the blockchain that allows to securely store and share data. It allows the creation of Web3.0 applications that are based on decentralized technologies. Unlike traditional development solutions, the protocol allows to:

1. Securely store data
2. Control data access with decentralized end-to-end encryption.
3. Broadcast data with delivery guarantee.
4. Validate the sender with the digital signature (protection from phishing attack).
5. Access data through an unlimited number of interfaces or clients from various vendors.

The protocol is blockchain-agnostic, does not use smart-contracts, is not based on crypto-economics, in alignment with regulators (GDPR, HIPAA, CCPA).

How it works

It all starts from the customer. Alice is a user of some applications. Instead of creating a login/password phrase, Alice on the client side automatically generates a key pair - the pair of a public and a private key. Those keys are generated based on asymmetric encryption algorithms like RSA.

Instead of logging in Alice can identify herself in an application with a public key, which can be disclosed to anybody. Let's imagine that Liam is a malicious user, who knows Alice's public key. He can now request the data from the application's server, and **he will get it**. For example Liam will be able to request Alice's incoming messages. Yes, he will get the messages, but they will be encrypted with Alice's public key. It means that Liam will be able to get the content of those messages only if he has the private key from Alice's public key.

Alice knows that the private key **should not be** disclosed and keeps it a secret. Decrypting data without a private key is a very complicated mathematical task. That's why Alice is not afraid that on her behalf someone will request her data. That principle works perfectly for interaction with an application and provides personalized content because the app can identify Alice by a public key (or a hash of a public key).

That is the key principle of the system. Anyone can have access to encrypted data without any limitations. But how should we store that data? Let's imagine that we store an encrypted message in separate files. How would we know the sender and the recipient? The recipient potentially could try to decrypt every message out there, but how will he find out the sender of that message? The solution for that would be a file with a structure that everyone will read and understand in the same manner. The simplest structure would be:

```
message:
  sender: Alice
  recipient: Bob
  ciphertext: ENCRYPTED_MESSAGE
```

This structure allows us to answer the question who is the sender, and who is the recipient. There is a more complicated version of this file below, which adds more proofs to the file. Proofs like verification of the sender with digital signature.

Let's take a step back. So, now we have a structured file with encrypted data. But what is the best way to store that file so we could be sure that the recipient can have access to it? That file storage system should be trustless and completely open to anyone. Fortunately there is a solution for that - an IPFS network. The network allows users to run an IPFS node and connect to other nodes in the network. Uning the gossip algorithm a node can search for a particular file that is stored on another (yet unknown) IPFS node. Every IPFS file has a unique identifier - a hash that is generated based on the content of a file. That hash will be generated again and again if we upload the same file, but will be totally different if the file has different content. For example a picture with a different pixel. That identifier (an IPFS hash) is also used as a link to the file in the network. So at the same time with one unique identifier we have both provable file content and a file address in the network.

It is important to mention that storing transaction-files and batch-files is **critical** because an application will not be able to generate the same transaction-file again. It means that the proof of file existence on a particular date will be lost.

Great. Now we have a peer-to-peer data exchange and file storage. An application can interact with Alice and store encrypted data that can be reached by anyone. We know who created the data and the recipient knows that the data is related to him (because he can decrypt it). But how can we know when the data was created?

To answer that question we need a trusted source that can't modify timestamps and somehow prove that a particular event happened on a particular time. That's where the blockchain can help. As it was said earlier, if we save the structured transaction-file to the IPFS network we will receive the unique identifier of a transaction-file. Same file - same IPFS hash. If we attach the IPFS hash of a transaction-file to the blockchain transaction we will be able to cryptographically prove that the file of particular content existed at a particular moment. That is possible because every blockchain transaction is a part of a particular block with a particular height that has a mathematically provable timestamp.

It is known that blockchain technology has a throughput limitation. For example Bitcoin blockchain can send only 3 transactions per second, Ethereum blockchain can send about 15 transactions per second. Sure, there are blockchain solutions that deliver much higher levels of throughput but unfortunately this is done at the expense of decentralization. Yes, the speed of message broadcasting is much higher, but the blockchain itself is less trustworthy.

This creates a problem of instant message delivery. This can be solved only with centralized technologies. There are lots of possible tools for that but the most suitable are Redis or Apache Kafka. As a transport layer they could instantly deliver messages between application and customer. Each message will have confirmation status which is the same as a transaction with a related IPFS hash.

Platform architecture

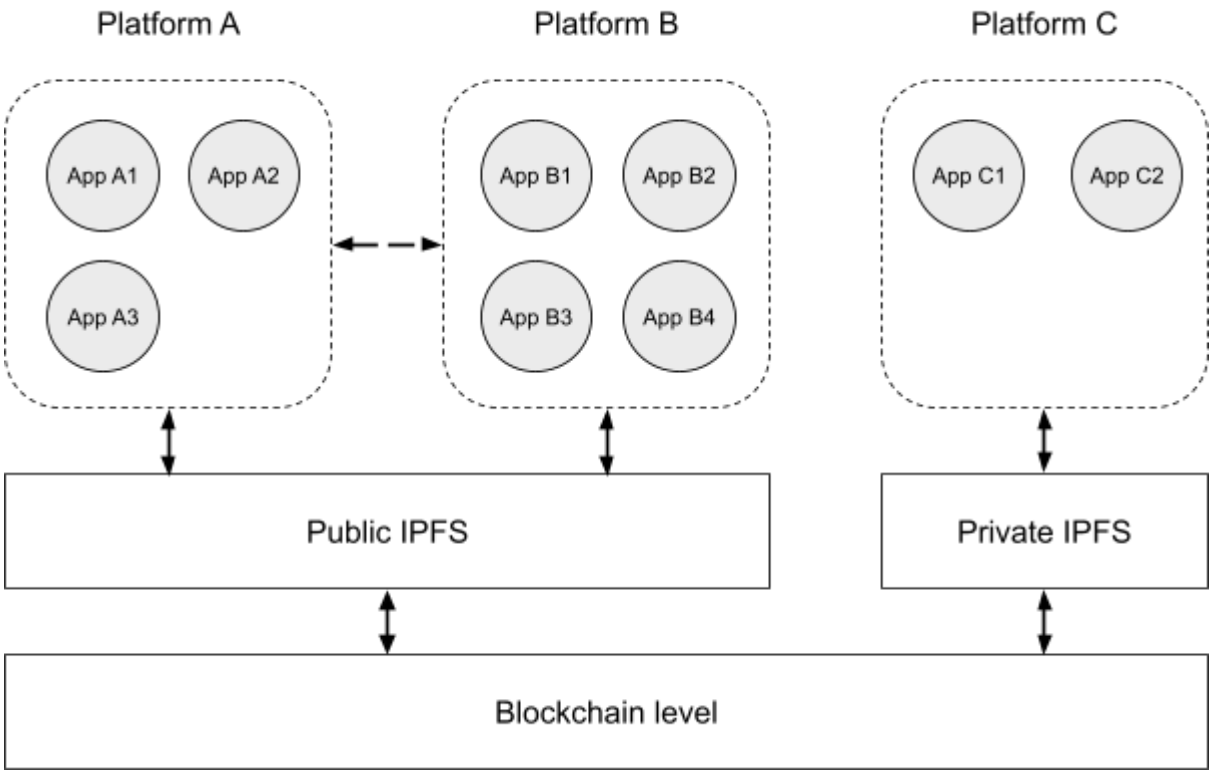
Currently the platform uses a microservice architecture which means that each service runs in a separate docker container. The structure of microservices is built in docker-compose files. Here is the list of microservices:

1. **client** - is an application client
2. **server** - is an API endpoint. It allows getting structured information about incoming and outgoing messages.
3. **parser** - is a tool for finding and matching abstract transactions. Encrypted data is saved to cache database PostgreSQL.
4. **ipfs** - is an IPFS node that allows saving and storing files based on IPFS protocol.
5. **postgresql** - is a Postgresql database that is used for storing parsed blockchain transactions. Blockchain allows not to worry about replication. In case of a database failure, the parser will recover the same information from the blockchain.
6. **redis** - instant data exchange
7. **nginx** - is a secure proxy server
8. **sponsor** - (optional) trusted third-party that pays for blockchain transactions

9. **autoheal** - is a container that checks the health of backend services every 10 seconds and restarts them if something goes wrong.

This architecture is already created and tested in the real world within a messaging application. The platform can be a separate isolated environment even with a private IPFS network. At the same time a platform could host different applications at one time. And that makes sense to connect several applications on one platform because those applications would be able to communicate between each other instantly. It is also an option to host an application on a shared platform but being isolated from other platforms.

The applications that are connected on different platforms will be able to communicate, but the messages will be sent with a delay. The delay time is defined by a particular blockchain provider. The schema below illustrates how different applications could communicate.



As we see from the schema above, Platform A connects three applications for instant communication. Applications A2 and A3 share the same platform and also can communicate instantly.

At the same time Platform B connects four applications for instant communication. Platform A and Platform B both use the public IPFS network. It means that B4 can send messages to A3 but with a delay. Platform C uses private network configuration and applications from C1 and C2 can communicate only with each other.

Data broadcasting algorithm

1. Alice is going to share some data with Bob and Carol. Alice knows Bob's and Carol's RSA public keys.
2. Using an open-source client Alice creates a group with Bob and Carol. That group has RSA key-pair and a random secret, which will be used in a hashing algorithm (hmac sha512). The group's public key (explained below), private key and random secret are encrypted for every participant, with their RSA public key.
3. Within a client Alice generates a *transaction-file* which stores data credentials.
4. That file is saved to the IPFS network. The network can be both private or public. After saving the IPFS node returns a unique hash. The hash is generated based on the file content and also becomes a link to the file in the network. That file can be reached from any IPFS node considering node discovery issues. It is assumed that every organization will have a known IPFS host server.
5. The file is delivered from peer-to-peer using a transportation layer (redis/kafka), and delivery is almost instant.
6. After receiving the file Bob and Carol parse it and decrypt the data with the group's private key. The data can be validated with the *hmac sha512* algorithm, using the group's random secret.
7. At the same time the transport layer creates a batch-file and saves it to IPFS network
8. Periodically the platform broadcasts a transaction with an attached IPFS hash to the blockchain and marks that transaction with a specific custom token. That custom token allows to filter blockchain transactions and works only with those which are related to the platform.

Transaction-file

Transaction-file is a file that is generated in the client side and contains structured information about the generated data. That file is saved to the IPFS network and can be accessed by anyone (there is an option for a private IPFS network).

File structure

As a structure syntax transaction-file uses standard YAML.

```yaml
transaction:
 version: '0.1'
 recipient:
   ciphertext: ENCRYPTED_GROUP_PUBLIC_KEY
   sha512hash: GROUP_PUBLIC_KEY_HMAC_HASH
 message:
   header:
     ciphertext: ENCRYPTED_MESSAGE_DATA
     sha512hash: HEADER_HMACH HASH
   body:
     ciphertext: ENCRYPTED_MESSAGE_DATA
     sha512hash: DATA_HMACH_HASH
```

```
  regarding:
    headerhash: ENCRYPTED_MESSAGE_HEADER
    bodyhash: ENCRYPTED_MESSAGE_DATA
  forwarded:
    headerhash: ENCRYPTED_MESSAGE_HEADER
    bodyhash: ENCRYPTED_MESSAGE_DATA
  attachment:
    - ciphertext: ENCRYPTED_IPFS_HASH
      sha512hash: ATTACHMENT_HMACH_HASH
    - ciphertext: ENCRYPTED_IPFS_HASH
      sha512hash: ATTACHMENT_HMACH_HASH
voting:
  secret:
    ciphertext: ENCRYPTED_HMAC_KEY
    sha512hash: HMAC_KEY_HMAC_HASH
  lastblock: LAST_BLOCK_HEIGHT
  option:
    - title:
        ciphertext: ENCRYPTED_OPTION_TITLE
        sha512hash: OPTION_TITLE_HMAC_HASH
      description:
        ciphertext: ENCRYPTED_DESCRIPTION_TITLE
        sha512hash: DESCRIPTION_TITLE_HMAC_HASH
    - title:
        ciphertext: ENCRYPTED_OPTION_TITLE
        sha512hash: OPTION_TITLE_HMAC_HASH
      description:
        ciphertext: ENCRYPTED_DESCRIPTION_TITLE
        sha512hash: DESCRIPTION_TITLE_HMAC_HASH
vote:
  voting: VOTING_HMAC_HASH
  option: OPTION_TITLE_HMAC_HASH
  voter: PUBLIC_KEY_HMAC_HASH
  signature: OPTION_TITLE_HMAC_HASH_SIGNATURE
blockchain:
  lastblock:
    height: LAST_BLOCK_HEIGHT
    signature: LAST_BLOCK_SIGNATURE
  trustedblock:
    height: TRUSTED_BLOCK_HEIGHT
    signature: TRUSTED_BLOCK_SIGNATURE
state:
```

```
  before: BEFORE_STATE_HASH
  after: AFTER_STATE_HASH
sender:
  - ciphertext: SENDER_PUBLIC_KEY
    sha512hash: SENDER_PUBLIC_KEY_HMAC_HASH
    signature: SIGNED_AFTER_STATE_HASH
  - ciphertext: SENDER_PUBLIC_KEY
    sha512hash: SENDER_PUBLIC_KEY_HMAC_HASH
    signature: SIGNED_AFTER_STATE_HASH
client:
  sha512hash: AFTER_STATE_HASH
  signature: SIGNED_AFTER_STATE_HASH
```

Current structure is used specifically for structured data storage, however each project can create its own structure, for example, for IoT communication and sending Operation Codes.

Let's have a closer look to the current structure:

1. **version** - the version of a protocol (required)
2. **client** - identification of client
    a. **sha512hash** - hmac sha512 hash of client's public key (required)
    b. **signature** - signed hmac sha512 hash of client's public key (required)
3. **recipient** - the recipient (group) of a message
    a. **ciphertext** - encrypted with organization's (matcher's) RSA public key recipient's RSA public key (required)
    b. **sha512hash** - hmac sha512 hash of recipient's public key (required)
4. **message**
    a. **header** - the header/subject of a message (optional)
        i. **ciphertext** - encrypted with recipient's RSA public key header
        ii. **sha512hash** - hmac sha512 hash of a header
    b. **data** - the data/body of a message (required)
        i. **ciphertex** - encrypted with recipient's RSA public key data
        ii. **sha512hash** - hmac sha512 hash of data
    c. **regarding** (optional)
        i. **headerhash** - hmac sha512 hash of a regarding header
        ii. **datahash** - hmac sha512 hash of a regarding data
    d. **forwarded** (optional)
        i. **headerhash** - hmac sha512 hash of a forwarded header
        ii. **datahash** - hmac sha512 hash of a forwarded data
    e. **attachment** (single or plural, optional)
        i. **ciphertext** - encrypted with recipient's RSA public key attachment
        ii. **sha512hash** - hmac sha512 hash of attachment
5. **voting**
    a. **secret** - a random secret that is shared between the voter and the voting organizer

    b. **lastblock** - the last block height that should contain the transaction with votes in order to count them
    c. **option** - an option that a voter can vote for
        i. **title**
            1. **ciphertext** - an encrypted title of a voting option
            2. **sha512hash** - hmac sha512 hash of a voting option title
        ii. **description**
            1. **ciphertext** - an encrypted description of a voting option
            2. **sha512hash** - hmac sha512 hash of a voting option description
6. **vote**
    a. **voting** - hmac sha512 hash of a voting (“after state” of original voting transaction)
    b. **option** - hmac sha512 hash of a voting option title
    c. **voter** - hmac sha512 hash of a voter’s public key
    d. **signature** - signed hmac sha512 hash of a voting option title
7. **blockchain** - the proof that data existed **after** some moment
    a. **lastblock** - the last mined block (required)
        i. **height** - last block height
        ii. **signature** - last block signature
    b. **trustedblock** - the trusted (minus 10 blocks) block height (required) in case of a fork
        i. **height** - trusted block height
        ii. **signature** - trusted block signature
8. **state** (required)
    a. **before** - hmac sha512 hash of a state (concatenated hashes) before sending
    b. **after** - hmac sha512 hash of a state (concatenated hashes) with current data
9. **sender** (single or plural, required)
    a. **ciphertext** - encrypted with recipient’s RSA public key sender’s RSA public key
    b. **sha512hash** - hmac sha512 hash of sender
    c. **signature** - signed “after state”

In the example above an encryption algorithm is RSA. It is the most trusted and time-tested asymmetric encryption algorithm out there, and normally it is totally fine to use it with a strong 4096 key. However we are living in the era of quantum computing which can dramatically affect encryption strength. That is why in a real world RSA encryption will be replaced with a combination of various protocols like elliptic curve encryption, ElGamal, KDF and strong symmetric encryption.

Batch-file

The batch-file is auto-generated periodically (for example every minute). It is also saved to the IPFS network which returns the unique hash/identifier of that file. That unique hash is

attached to a blockchain transaction which allows to prove that this data existed at a particular moment - the moment of generating a block with that transaction in it.

File structure

Each batch-file contains a unique list of transaction-files that were created after the last batch.

```
batch:
 transaction:
   - QmcMg3Js6w...TLPMsGfLdiQC
   - Qmcff5e1GXM...yXYc2bFJo3JB
   - ...
```

This solves the scalability problem and allows use of slow but extremely reliable Proof-of-Work blockchains like Ergo. Also batch-files can mathematically prove that some data existed in a known time period with several minutes accuracy.

Proofs

The delivery guarantee

There are several layers of data delivery. The file storage layer (IPFS), the instant delivery layer (redis/kafka) and the backbone layer (blockchain).

The instant delivery layer is optional, but it allows sharing data as fast as possible. Every transaction-file contains the current state (hash) of the group's data for a particular client (Alice, Bob, etc.). That state has two fields for the previous state and for the current state. Those states can be easily linked through hashes which creates the **order** of file sharing. So the instant delivery layer allows to share data fast and in provable order.

The file storage layer is handled by the IPFS network. It is designed as a decentralized peer-to-peer network with unlimited number of participants. Anyone can deploy an IPFS node and connect to a public (or private) network. It allows anyone to access any file but read the data only if one has decryption keys. So this layer is for **what** data was shared.

The backbone/blockchain layer. This layer allows us to prove **when** the data was shared. if there is at least one transaction in the batch, the platform broadcasts the blockchain transaction which attaches the IPFS hash of a batch-file. The platform is open-source and does not need to be trusted. The IPFS file can’t have a different identifier/hash with the same content because the hash is calculated based on file content. The blockchain transaction is also immutable and can’t be deleted or modified. That property allows us to prove the existence of a particular file with particular content on a particular moment.

Every data is shared with groups (described below). Every group participant has access to the group's decryption keys. Every participant can monitor the blockchain and get IPFS file content and try to decrypt it with the group's private key. The correct combination of encrypted data and decryption keys will unlock the encrypted data. That allows to verify that the recipient (group or group participant) **received** the data.

Every transaction-file contains the sender's data including the digital signature. That allows us to verify the **sender(s)** and validate the uniqueness of a signature because the previous files are available. that protects the recipient(s) from phishing attacks based on pure math.

It is important to mention that every transaction-file contains the Client's signature, which allows verification of the client's client. That is important because a transaction-file is generated on the client side, and the client has access to decrypted data. Malicious client could send decrypted data to a third-party, which is unexceptable. Each client's signature is unique, but that unique signature can be accessed by anyone because it is stored in a transaction-file with public access. That puts an obligation to the platform owner to keep all signatures from day one to make sure that every signature is unique. But that is not efficient. As a solution the client could update the Key Pair from time-to-time and make a public announcement with disclosure of the new public key. That will allow to keep only the signatures for the last epoch. That will also allow the creation of LTS versions of software and obligate the customer to upgrade the client after the end of a particular LST epoch.

The protection from unauthorized access

Every customer has a unique RSA key-pair that is generated on the client side. RSA public key is shared in a decentralized manner (with classical communication tools) and is used for data encryption. The decryption keys are never stored by the platform and cannot be restored in case of loss.

Security

Classical data security principals are based on protecting some secret with different layers of protections like private VPN, firewalls etc. The problem is that secret data should be protected from **every** attack out there but the hacker should find only **one** vulnerability. That leads to endless improvement of data protection schemes. The more complicated a security system is, the higher will be the cost, and of-course the harder it will be to maintain and modify the security system. The advanced security is available only to enterprise-level companies with endless budgets. The small and medium size businesses can't afford it, or just don't consider security risks at all.

The technology allows applications that are protected from data leakages and unauthorized access **by default**. Fundamentally there is only one thing that an attacker should hack - the encryption. Hacking cryptography is a mathematically hard operation which normally takes billions of years.

Technological advantages

The main advantages are auditability and data access control. The protocol allows creating zero-trust applications because the database does not know what is stored in it.

Better messaging

Current messaging solutions like **Email** or **WhatsApp** are (a) centralized, (b) unsecure and (c) mutable.

There is no way to send a message from Alice to Bob without a central server. It means that such a server is a ‘trustee’ in the process of transmission. Users may not be 100% sure that a message was delivered. Also if servers go down, no transmission will be possible as such.

There are reasons why services like Telegram or ProtonMail are banned in Russia, for one, to allow (unauthorized) access by third parties. There is also a question as to why Gmail, Mail.ru, Yandex or WhatsApp are not banned in Russia. Also, users’ data is an asset that can be sold to big internet companies (whether users can actually prevent that or not).

All messages and emails are stored on centralized servers. The data can leak, be deleted or lost. Also data can be modified, users may not be 100% sure about data immutability.

The protocol guarantees sending and receiving of messages with protection from unauthorized access and protection from data leakages.

Better security

The classical way to secure data is to build cardons of security layers like VPNs or FireWalls. But the problem is that security teams must protect services and customer’s data from **every** potential threat. At the same time the hacker needs to find only **one** vulnerability to hack the system.

The technology guarantees protection of data by delegating all security threats to math. To hack the system the hacker will have to hack the cryptography under the hood, which is a mathematically complicated task. The applications will significantly reduce both the risks and the cost of security protection.

Every message is accompanied with a digital signature which allows to verify the message sender. That provides built-in **protection from phishing attacks**.

Total auditability

The main idea of Web 3.0 is to reduce the trust factor to zero. That can be achieved by providing a transparent environment which allows customers to control and verify data flow. That concept allows to be sure that the message was sent with particular encrypted content. That is important for applications that use DeFi Messenger to interact with their customers. That also makes it possible to provide Messenger-as-a-service and charge for each message.

Monetization

Messenger-as-a-service

As it was said earlier DeFi Messenger connects applications with their customers. The same problem is solved with email. DeFi can charge for a batch of sent messages (text, media, votes, purchases etc.). The best model is freemium, with a free set of messages that could be delivered by an app per month. For example, the same monetization model was used by Mailchimp and was extremely successful.

A corporate cluster

The software is open-source and under MIT license. Any organization can deploy it on their infrastructure. In this case they will have to handle software updates, fix the bugs, backup the data, configure data replication, and handle the crypto-economics.

In case of corporate policy or regulatory requirements, the platform can be deployed on corporate servers. In this case the configuration will be discussed individually and in case of agreement our team will have access to corporate servers. This will allow us to monitor health-check, upgrade the software, fix the bugs and handle the crypto-economics.

Customization options:

- Issuing custom token
- Protocol version (sponsored or personal transaction broadcasting)
- Access policy / Block the resigned employees
- Cross-platform communication (sending data instantly from platform to platform)
- Custom client configuration (branding, access policy, etc)
- Etc.

The fact that our team is the creators of the protocol and the software, that will add a lot of credibility even if there will be competitors.

Data storage as a service

As a company we will use our own software and deploy it on our servers. We will grant other developers access to our platform and let them use our hardware to run their applications. Every application will generate data that will require space on the hard drive and also require instant-messaging capacity. As it was said earlier, protecting IPFS files is critical. As a service provider we will be able to charge both for data and for every outgoing message. The numbers are fully transparent and can be proved, which means that we know exactly how many messages were sent by a particular organization.

Applications within one platform will be able to exchange data instantly which will create a network effect. It would be great to be able to provide access to the platform for the low, affordable for SMB price. Potentially that will have a great effect on the growth-factor.

Services

It is totally possible to create a centralized payment gateway that will allow to make in-app purchases. That service can be extended with a **virtual credit card**, the same as an Apple card. That card would protect customers from unexpected purchases (even periodical) and will require a two-factor-authentication (2FA) for any external purchase request.

For example, if a customer would like to buy flowers, after picking a perfect bouquet, he will receive a message with shopping details and a request to authenticate the payment. After that the payment processing company will charge the money from the credit card and send another message to the customer, that the payment is processed. The flower store could send another message that the payment was successfully processed.

Another example. The customer is going to rent a car, and pays with his virtual credit card. That creates a purchase request that is sent to the customer. That request asks to validate some deposit and a rental fee. Some time later the customer can get another request that asks to pay for the toll road. The customer just signs the requests and makes payments without disclosing the credit card data to the car rental office. That virtual credit card can be connected to any existing credit card. The bank itself will be able to authenticate the payments and send an extra 2FA message to the customer.

Due to the full transparency of data flow and the digital signatures, every organization or the customer will be able to prove that the message was sent and received.

Value for the customer’s customer

The main benefit for a company's customers is provable interaction with the company which leads to a much higher level of trust. In complicated cases where the company acts maliciously, the protocol will protect the customer. The data that is related to the customer is stored on his device. As an option, we will be able to backup customer’s data for an extra fee.

Company’s customers will be able to see their interaction with the company, same as interactions of other customers. Of course they will see only the fact of interaction without knowing any details. The protocol will protect every message and will be able to mathematically prove the timestamps and the content of the messages.

A great example would be the communication of state citizens with authorities. Again every fact of communication has a timestamp and is protected by digital signatures. It means that every official will be able to validate the citizen by a digital signature. The important thing is that digital signature can be generated in a decentralized manner which makes it even more secure.

That digital signature is a great tool for making digital contracts as well. That will allow to avoid paperwork and care about every contract within the app.

In-app voting

Abstract

It’s a chess gaming platform where teams can compete with each other by means of collective decision. The decision is made with a voting mechanism which is handled by a DeFi messenger application. The voting protocol solves the problem of unfair votes counting and end result manipulation. It does not solve the problem of a malicious voter that “sells” his vote for any reason.

The protocol

The voting protocol is similar to a message-sending protocol. It is based on a combination of IPFS and Blockchain technologies. The vote is organized by a central authority that handles collection of votes. The protocol does not use smart-contracts and is fully auditable.

Voters can be sure that their vote is counted and counted correctly	Yes
Voters can be sure that all votes counted correctly	Yes
Voters can double-check that other voters voted once	Yes, but only if the voters addresses and secrets will be revealed
The voter can prove how he voted	Yes
Intermediary results are hidden	Optional (yes, by default)
Nobody knows how the voter has voted	Optional (yes, by default)
Voter can’t be de-anonymized	Yes, but central authority knows every voter
Voter can vote only once	Yes (application level restriction)
Validated time-stamping	Yes
Unlimited number of voters per voting	Yes, for voters on the same platform
Voting organization in minutes	Yes, for voters on the same platform
Voter KYC	Optional (application level restriction)
Voting has time limit	

How it works

1. The organizer (central authority) sends a message to all voters with a voting description and all available options. To send a message the organizer uses a DeFi messenger APIs (described above).
2. The voter receives the message with available voting options and signes one of them. The DeFi messenger application will display all options as separate buttons
3. The voter pushes part/;'icular button and creates a transaction file which then is broadcasted to the blockchain network
4. Every voting has a time limit which is set as a particular blockchain height. The voting organizer receives all votes until the preset blockchain height is exceeded.
5. After the organizer stops receiving votes, the results are published in a Votes file (described below) which can be sent to all voters or publicly announced.

Votes file

Votes file is a XML file that contains all collected votes. Every voter can examine it and be sure that all votes counted correctly.

```
<?xml version="1.0"?>
<votes>
   <vote>
       <option>OPTION_HASH</option>
       <publickey>PUBLIC_KEY_HMAC_HASH</publickey>
       <signature>SIGNATURE</signature>
   </vote>
   <vote>
       <option>OPTION_HASH</option>
       <publickey>PUBLIC_KEY_HMAC_HASH</publickey>
       <signature>SIGNATURE</signature>
   </vote>
   <vote>
       <option>OPTION_HASH</option>
       <publickey>PUBLIC_KEY_HMAC_HASH</publickey>
       <signature>SIGNATURE</signature>
   </vote>
</votes>
```

The end result file is also published to IPFS and broadcasts to the blockchain network. That protects the end results from manipulation in future because it proves the particular content of a file on a particular moment.