Concordium

Decentralized Applications integration

Document version: 0.6 (draft, beta)

Versions history:

0.1 – initial beta

0.2 – adding serialization rules and serialized\_params attribute to protocol

0.3 – extend contract transaction parameters list with mandatory attribute, fix rules for serialization

0.4 – extend message with transaction cost attribute *nrg\_limit*

0.5 – polish some description

0.6 – added optional attributes for transaction request and response

### Main Concept and Goals

To enable Decentralized Applications development we need to integrate Web application and Wallets. 1st priority goal - Mobile Wallet.

Good approach for integration is implemented by WalletConnect

We need to build similar integration solution.

### Functional requirements:

Be able to interact with Wallet from Web application for:

- initiate Connection to Wallet

- Prove wallet ownership

- Send Transaction to Blockchain (SimpleMoneyTransfer or SmartContract transaction)

**Notes**

Protocol definitions, that not implemented yet, marked by italic text and highlighted with yellow color. Example: *nrg\_limit*

### Integration

Transport Layer

Transport layer flow is base for further communication. Responsible for:

- establish connection from Web application to Bridge/Proxy

- establish connection from Mobile wallet to Bridge/Proxy

- handle connection information

- manage connection state (reconnect/disconnect)

1) Establish connection from web to bridge:

wss://{{bridgeserver}}/ws/web/frame/{{access\_key}}/-

Where {{access\_key}} is key, obtained from Bridge Server owner. Each access key is unique for web frontend domain name.

Result of success connection is:

access#{

"connect\_string": "https://<bridgeserver>/condition/<connectionkey>/<sessionkey>"

}

2) Establish connection from mobile:

Web need to generate QR Code with encoded connection\_string value "https://<bridgeserver>/condition/<connectionkey>/<sessionkey>"

Mobile wallet:

- make empty POST request to provided url and get metadata:

{

"site": {

"title": "Web Site Name",

"description": "",

"icon\_link": "https://<web\_frontend>/favicon.svg"

},

"connection": {

"ip": "<ip>:port",

"platform": "",

"os": "",

"browser": "/",

"is\_portable": false,

"opened\_at": 1627667400

},

"meta": null,

"ws\_conn": "wss://<bridgeserver>/ws/mobile/frame/<connect\_id>/<session\_id>"

}

- prompt user for accepting requested connection.

In case Accepted is pressed, open websockets connection by url, provided in ws\_conn attribute value

As result of success connection to websocket bridge, Mobile wallet receives:

access#{

"session\_id": "<session\_id>",

"connect\_id": connect\_id

}

Mobile wallet need to save session\_id and connect\_id for further communication. These values could be used for reconnect in case some interruption occurs.

At web site, after connection, message via websocket will be received:

proxy#{

"data":"ConnectionAcceptedNotification",

"message\_type":"ConnectionAcceptNotify",

"network\_id":"<network>",

"originator":"<app\_name>",

"user\_status":"UserAccepted"

}

Its could be handled as successful communication and triggers QR Popup close.

*3) handle connection information*

*TO BE DESCRIBED*

*4) manage connection state (reconnect/disconnect)*

*TO BE DESCRIBED*



#### Flow Overview

Interaction example as sequence diagram:



Integration participants:

- Wallet – here is Mobile Wallet Application (iOS/Android) for Concordium Blockchain

- Concordium Web Bridge – here is websocket proxy server, that provides connection between web application and wallet

- Web Client – here is Web Application Frontend

- Web App Backend – backend for web application

- Wallet Proxy – intermediate service for Concordium Blockchain, that serves Wallet interaction with Concordium Blockchain Network

- Blockchain Network – here is distributed database/ledger based on blockchain technology, Concordium. Could be any node, participated in Network.

Basic Flow description:

1. Connect User of Web Application with his Mobile Wallet.

- User after sign-in, wants to connect his wallet to perform certain operations. He press something like “Connect Wallet” button.

- Web Application generate QR Code, establish connection to WebSocket Proxy (Bridge) and show QR code for scanning.

- User run his Mobile Wallet at smartphone and perform scanning of QR code

- Mobile Wallet make connection to Bridge and get additional information about originator (metadata)

- Mobile Wallet after recognizing QR code, shows “Connect Approval” dialog, that contains metadata about web application: Site Name, DNS, Icon(favicon?)… and Accept, Reject buttons

- In case User press Accept, Wallet send notification via Bridge, that connection accepted. Save connection session id, etc.

- In case User press Reject, Wallet send notification via Bridge, that connection rejected and Close connection to Bridge

2. Prove user wallet ownership

… To Be described. In general, need to Sign some message by Wallet and than check sign at Web Application

3. Get Account Information

Web Application make request via bridge for Wallet Address and Balance: after user connect his wallet or when user open his profile with Wallet, or when Web Application need to show wallet balance.

- Web Client send request AccountInfo to Wallet via Bridge

- Wallet respond with Account Addresses and correspond unshielded balances

4. Make transaction

When user wants to make some action at Web Application, that require transaction at Blockchain: He pressed button (e.g. “Make Bid”).

- Web Client get transaction parameters from Backend (e.g. Contract Address, Amount/Price, Method Name, Method parameters… )

- Web Client prepare transaction request and send it to Wallet via Bridge

- Wallet shows Transaction request dialog to User in Mobile Wallet Application. Dialog contains Transaction Data (Originator Info, Amount, Receiver, Contract Method, Parameters) and Accept, Reject buttons.

- In case User press Accept, Wallet send notification via Bridge, that transaction accepted; encode Transaction, Sign, send via wallet proxy to Blockchain.

- In case User press Reject, Wallet send notification via Bridge, that transaction rejected.

- when Wallet get notification from Blockchain, it shows information about transaction status to User. And send notification about transaction status to Web Client via Bridge.

- when Backend service got notification about transaction from Blockchain (Smartcontract event), it save/update information about some item, related to this transaction.

5. Simple Money Transfer

When Web application have some functionality (e.g. “Donate to charity fund”), that allows to send money to certain Address/Account, user could press button for this action:

- Web Client send SimpleTransfer request to Wallet via Bridge. Request contains Receiver, Amount

- Wallet shows SimpleMoneyTransfer request dialog to User in Mobile Wallet Application. Dialog contains Transaction Data (Originator Info, Amount, Receiver) and Accept, Reject buttons.

- In case User press Accept, Wallet send notification via Bridge, that Transfer sent; encode Transfer, Sign, send via wallet proxy to Blockchain.

- In case User press Reject, Wallet send notification via Bridge, that transfer rejected.

- when Wallet get notification from Blockchain, it shows information about Transfer status to User. And send notification about Transfer status to Web Client via Bridge.

- when Backend service got notification about transfer (e.g. related to Address), it save/update information about some item, related to this transfer (e.g. updates list of donates). (???)

#### Messages design

Here is list messages (payload data) per interaction

All messages **request** have following format (json):

{

originator: "",

network\_id: "",

message\_type: “<one of from set>”,

description: “Create NFT”,

data: {payload}

}

originator – “some user agent/app info”

network\_id – “network identifier e.g. stage, mainnet, etc.”

description – “some user friendly transaction description”; could be used to show at wallet application

message\_type – “one of supported request”. Could be AccountInfo|SimpleTransfer|Transaction|*SignRequest*

Each request payload:

{ <request\_data\_depended\_from\_message\_type>

}

Each response have followed format (example for Connection Acceptance):

proxy#{

"data":"ConnectionAcceptedNotification",

"message\_type":"ConnectionAcceptNotify",

"network\_id":"stage",

"originator":"Android app",

"user\_status":"UserAccepted"}

Where

proxy# - prefix, that added by Web Bridge before actual json message.

Message itself:

originator is User Agent Info,

network\_id = "connected network id",

user\_status: UserRejected - user press decline button in wallet or UserAccepted - user press Accept button - in case user\_status:UserAccepted, “data”: “ConnectionAcceptedNotification” ; in case user\_status:UserRejected, "data":"" will in response be empty

**Supported message\_type**:

requests: "**AccountInfo**|SimpleTransfer|**Transaction**|*SignRequest*"

responses: “ConnectionAcceptNotify|AccountInfoResponse|SimpleTransferReponse|TransactionResponse|SignResponse”

Data section per message\_type:

1. Get Account info from wallet. message\_type:"AccountInfo"

{"message\_type":"AccountInfo","data":{}}

Data is empty, as we request list of Accounts from wallet

Response from wallet:

proxy#{"data":[{"address":"4MD6pAAz62XXXXXXXXXXXXXXvnFraM41TEAetAUU","balance":"2000000000"}],"message\_type":"AccountInfoResponse","network\_id":"stage","originator":"Android app","user\_status":""}

2. Send Money from wallet address to another: message\_type:"SimpleTransfer"

"data":

"{

from:"WalletAddress",

to:"RecieverAddress",

"nrg\_limit":"",

nonce:"",

amount:""

}"

Response from wallet:

"data":{TxHash:"",TxStatus":""}

Where TxStatus one of "Accepted"|"Rejected"

Accepted - mean accepted by Network

Rejected - mean rejected by Network

3. Send transaction to smartcontract. message\_type: "Transaction"

Data must contain smartcontract method and parameters definition, joined with supported data types

"data":

{

"from":"WalletAddres",

"nonce":"",

"amount":"",

"expiry":"",

"nrg\_limit":"",

"contract\_title":"",

"contract\_address":{"address":"","index":"","subindex":""},

"contract\_name":"name",

"contract\_method":"method",

"contract\_params":[

{"param\_name":"param1", "param\_type":"param1Type", "param\_value":"param1Value", "mandatory":"1"},

{"param\_name":"param2", "param\_type":"param2Type", "param\_value":"param2Value", "mandatory":"1"},

{"param\_name":"param3", "param\_type":"param3Type", "param\_value":"param3Value", "mandatory":"0"}

],

"serialized\_params":"serializedParams"

}

All attributes in json must be sent as text, enclosed in ""

Attributes description:

"from" - address, which sent transaction (note, that Wallet could have several address under management, so this attribute specify which to use)

"nonce" – empty attribute, as in general case, Wallet need to get nonce by request to blockchain

"amount" – amount in microCCD, that will be withdrawn from address with transaction

"expiry" – expiry time in transaction (timestamp), after this, transaction become invalid

"nrg\_limit" – ENERGY limit, specified in NRG unit – maximum value of transaction fee, that customer ready to spent while transaction execution. This value is validated at blockchain before accepting transaction. If wallet account balance is lower, than nrg\_limit, converted in CCD, + amount , transaction will be dropped. Limit value is maximum fee amount, that allowed to be spent.

"contract\_title" – descriptive, optional attribute, that used by Wallet to show user-friendly transaction description. Value could be any text. Response from wallet will contain this value in "action" attribute

"contract\_address" – contract address in format {"address":"","index":"","subindex":""} - index and subindex is only valuable values.

"contract\_name" – name of contract to call

"contract\_method" – name contract method to call

"contract\_params" – array of contract method parameters (could be used by wallet to show transaction details for user):

"param\_name" – name of parameter (informative),

"param\_type" – values is one of [uint8, uint16, uint32, uint64, AccountAddress, ContractAddress, amount, string] – this value is used to make correct serialization

"param\_value" – value, that actually serialized

"mandatory" – attribute, that indicates parameter is optional or mandatory, value is one of 1 or 0

"serialized\_params" – all parameters, serialized by formal rules. Value will be sent to blockchain “as is”. All parameters must be serialized in same sequence, as it defined in SmartContract method in blockchain

Example:

{

"message\_type":"Transaction",

"data":{

"from":"4MD6pAAz629ZhXKXXXXXXXXXXXXXFraM41TEAetAUU",

"nonce":"",

"amount":"8",

"expiry":"1650283529",

"contract\_address":{

"address":"",

"index":"276",

"sub\_index":"0"

},

"contract\_name":"trader",

"contract\_method":"buy",

"contract\_params":[

{

"param\_name":"lot\_id",

"param\_type":"uint64",

"param\_value":"2189"

},

{

"param\_name":"value",

"param\_type":"uint64",

"param\_value":"1"

}

],

"nrg\_limit":"60000",

"serialized\_params":"8d080000000000000100000000000000"

}

}

Response from wallet:

"data":{"TxHash":"","TxStatus":"", "action":""}

Where TxStatus one of "Accepted"|"Rejected"

Accepted - mean accepted by Network

Rejected - mean rejected by Network

action – descriptive attribute from wallet, to help recognizing completed transaction: Wallet could set it equal <”contract\_title”> or <“contract\_name”.”contract\_method”>

*4. Sign message. message\_type: "SignRequest"*

*"data":{*

*from:"WalletAddress",*

*message:"Message to sign"*

*}*

*Response:*

*"data":{*

*signerAddress:"",*

*sign: ""*

*}*

==========

Examples for smart contract transactions (NFT ERC721):

{

...

"message\_type": "Transaction",

"description": "Create NFT",

"data":

{

"from":"owner\_address", //from user profile, address of connected wallet

"nonce":"", // empty, value from user wallet Account Info

"amount":"0", // no money send for mint

"expiry":, // now() + 30 minutes

"nrg\_limit":"150000",

"contract\_title":"mint NFT",

"contract\_address": {"address": "","index": "51","sub\_index": "0"},

"contract\_name":"inventory", //

"contract\_method":"create", //

"contract\_params": [

{ "param\_name":"token\_id", "param\_type": "uint64", "param\_value": "token\_id", "mandatory": "1" },

{ "param\_name":"owner\_address', "param\_type": "accountaddress", "param\_value": <empty>, mandatory: "0" }, //direct inventory create call, must be empty

{ "param\_name":"royalty", "param\_type": "uint64", "param\_value": "roaltypercent", "mandatory": "1" }, //from web form, backend (0-99 values accepted)

]

"serialized\_params":"HEXENCODED"

}

}

S**ma**rt Contract Parameters serialization rules

Javascript library with serialization function is available for use. For other language you need to implement by rules, described below.

Parameters must be serialized in same order, as defined in Smart Contract

Rule of serialization depends from parameter type:

**uint64** : hex from Little Endian uint64

**address** : base58

**string** : concatenated hex of string length (LE uint32) and hex of string

**bool** : hex from uint8

**amount**: hex Little Endian from uint64

**timestamp**: hex Little Endian from uint64

**duration**: hex Little Endian from uint64

If parameter **mandatory** attribute is **0**, then before encoded value of parameter , encoded boolean value need to be set: 01 in case parameter value exist; 00 in case value is empty

If parameter **mandatory** attribute is **1**, then **no boolean prefix** need before parameter value

After encoding, all parameters need to be concatenated in string

Example

Parameters for inventory.create ERC721 SmartContract transaction:

{ "message\_type":"Transaction",

"description": "Create NFT",

"data":{

"from":"4MD6pAAz629ZhXKLNxJgfhjdfufhdBTQvnFraM41TEAetAUU",

"nonce":"",

"amount":"0",

"expiry":"1635416004",

"nrg\_limit":"44000",

"contract\_address":{

"address":"",

"index":"70",

"sub\_index":"0"

},

"contract\_name":"inventory",

"contract\_method":"create",

"contract\_params":[

{ "param\_name":"token\_id",

"param\_type":"uint64",

"param\_value":"305455883003887619",

mandatory: "1"},

{ "param\_name":"accountaddress\_from",

"param\_type":"accountaddress",

"param\_value":"",

mandatory: "0"},

{ "param\_name":"royalties",

"param\_type":"uint64",

"param\_value":"0",

mandatory: "1"},

{ "param\_name":"url",

"param\_type":"string",

"param\_value":"ipfs://",

mandatory: "1"}

],

"serialized\_params": "030000d981323d0400000000000000000007000000697066733a2f2f"

}

}

Concatenation of next strings:

token\_id(uint64). First item in contract\_params list in transaction request: Hex in Little Endian

accountaddress\_from(accountaddress), Creator missing address identifier: Hex from 0 (uint8)

royalties(uint64). 3rd item in contract\_params list in transaction request: Hex in Little Endian

url (string). Concatenation of hex in LE from URL length in uint8 and hex of URL content

Example:

"params": "030000d981323d0400000000000000000007000000697066733a2f2f"

- 030000d981323d04 = hex from 305455883003887619 (uint64) in little endian, token id

- 00 = hex from 0 (uint8), **creator missing** **address** identifier

- 0000000000000000 = hex from 0 (uint64) in little endian, royalty

- URL value “ipfs://”

hex in LE from URL string length in uint32 == “07000000”

Hex of URL == “697066733a2f2f”

Concatenation will be: “07000000697066733a2f2f”