Summary

Communication between wallets and decentralised applications.

Abstract

This document describes the communication between an app (decentralised application on Tezos) and wallet (eg. browser extension).

Motivation

A user wants to be able to interact with decentralised applications e.g. uniswap-like decentralised exchange or a game that uses NFTs, he wants to use his preferred wallet and avoid having to install a new wallet for each application he uses.

To enable adoption of decentralised applications in the Tezos ecosystem a standard for the communication between these apps and wallets is needed. Application developers shouldn't need to implement yet another wallet just for their use case and users shouldn't need a multitude of wallets just to use a certain service.

This standard should form consensus of how the types of messsages for this communication look like.

Specification

Communication Protocol Version 1.0.0 (Version code 1)

The entire protocol relies on asynchronous calls. A call consists of a request message and an asynchronous response message.

Versioning

The transport layer is responsible for the communication of the protocol version. All of the discussed messages will include the version number.

Message Types

The following chapters describe the different message types that exist between the app and wallet. This protocol assumes a secure and authenticated transport layer was established hence no messages can be spoofed.

Shared Types

```
export interface AppMetadata {
  senderId: string;
  name: string;
  icon?: string; // TODO: Should this be a URL or base64 image?
}
```

Network

For most of the request, a network has to be defined. If no network is specified, `mainnet` is used.

The `name` and `rpcUrl` can be changed for all networks. This is optional for networks `mainnet` and `carthagenet`. If no values are provided, a default will be used.

For the `custom` type, `name` and `rpcUrl` are mandatory.

```
export enum NetworkType {
  MAINNET = "mainnet",
  CARTHAGENET = "carthagenet",
  CUSTOM = "custom",
}

export interface Network {
  type: NetworkType;
  name?: string;
  rpcUrl?: string;
}
```

PartialTezosOperation

```
type omittedProperties =
  | "source"
  | "fee"
  | "counter"
  | "gas_limit"
  | "storage_limit";

// All of the operations types defined here are supported:
https://tezos.gitlab.io/api/p2p.html#operation-alpha-contents-determined-from-
data-8-bit-tag
// We remove some of the properties, which will be filled out by the wallet (eg.
fee and counter)

export type PartialTezosDelegationOperation = Omit<
  TezosDelegationOperation,
  omittedProperties
>;
export type PartialTezosOriginationOperation = Omit<
  TezosOriginationOperation,
  omittedProperties
>;
export type PartialTezosRevealOperation = Omit<
  TezosRevealOperation,
  omittedProperties
>;
export type PartialTezosTransactionOperation = Omit<
  TezosTransactionOperation,
  omittedProperties
>;
```

```
export type PartialTezosOperation =
  | TezosActivateAccountOperation
  | TezosBallotOperation
  | PartialTezosDelegationOperation
  | TezosDoubleBakingEvidenceOperation
  | TezosEndorsementOperation
  | PartialTezosOriginationOperation
  | TezosProposalOperation
  | PartialTezosRevealOperation
  | TezosSeedNonceRevelationOperation
  | PartialTezosTransactionOperation;
```

BaseMessage

```
export enum MessageType {
  PermissionRequest = "permission_request",
  SignPayloadRequest = "sign_payload_request",
  OperationRequest = "operation_request",
  BroadcastRequest = "broadcast_request",
  PermissionResponse = "permission_response",
  SignPayloadResponse = "sign_payload_response",
  OperationResponse = "operation_response",
  BroadcastResponse = "broadcast_response",
  Disconnect = "disconnect",
  Error = "error",
}

export interface BaseMessage {
  type: MessageType;
  version: string;
  id: string; // ID of the message. The same ID is used in the request and
response
  senderId: string; // ID of the sender. This is used to identify the sender of
the message
}
```

Communication Flow

After the channel between app and wallet has been opened (see Transport Layers chapter), the communication should always be initiated with a `PermissionRequest`.

`OperationRequest` and `SignPayloadRequest` always have to be addressed to a specific account in the wallet. It is not possible to send either of those requests without a prior `PermissionRequest`. If an operation is requested where no permissions have been given, or they have been rejected, a `NO_PERMISSION` error will be returned.

The `PermissionRequest` and `BroadcastRequest` can be sent without a prior `PermissionRequest` because they are not directly related to a specific account.

1. Permission

App requests permission to access account details from wallet.

Request

```
export enum PermissionScope {
  SIGN = "sign",
  OPERATION_REQUEST = "operation_request",
  THRESHOLD = "threshold",
}

export interface PermissionRequest extends BaseMessage {
  version: string;
  id: string;
  senderId: string;
  type: MessageType.PermissionRequest;
  appMetadata: AppMetadata;
  network: Network;
  scopes: PermissionScope[];
}
```

`appMetadata`: An App should be identifiable by the user, for example with a name and icon.

`network`: Network on which the permissions are requested. Only one network can be specified. In case you need permissions on multiple networks, you need to request permissions multiple times.

`scope`: The App should ask for permissions to do certain things, for example the App can request to sign a transaction (maybe even without confirmation for micro-transactions). The following scopes are defined:

- `sign`: this indicates the app wants to be able to sign.
- `operation_request`: this indicates the app wants to perform payment requests.
- `threshold`: this indicates the app wants the wallet to show the user a threshold prompt, so that in future transactions below this threshold can be handled automatically by the wallet. The threshold is optional and does not have to be supported by wallets.

Threshold

Wallets can define a threshold and report it back to the app. The wallet needs to define an `amount` and a `timeframe`. For security reasons, no data is allowed (no contract calls).

- `amount`: indicates how much XTZ (in mutez, smallest value) can be spent per timeframe. This includes the transaction amount and transaction fees.
- `timeframe`: specifies how long the timeframe is in which the `amount` limit is enforced (in seconds)

As an example, if `amount` is set to `1000000` (1 XTZ), and the `timeframe` is set to `3600`, it means that within a timeframe of 1 hours, not more than 1 XTZ can be spent. So 2 transactions with an `amount` of `300000` (0.3 XTZ) and a `fee` of `100000` (0.1 XTZ) each would be signed automatically, without user interaction. This is

because the total `amount` spent is `800000` (0.8 XTZ). However, a third transaction would have to be confirmed by the user because it would exceed the threshold.

Response

```
export interface PermissionResponse extends BaseMessage {
  version: string;
  id: string;
  senderId: string;
  type: MessageType.PermissionResponse;
  publicKey: string; // Public Key, because it can be used to verify signatures
  network: Network; // Network on which the permissions have been granted
  scopes: PermissionScope[]; // Permissions that have been granted for this
specific address / account
  threshold?: {
    amount: string;
    timeframe: string;
  };
}
```

- `publicKey`: The public key of the account, in the format of a hex string
- `network`: The network on which the permissions have been granted. (Should usually be the same as has been requested)
- `scopes`: The permission scopes that have been granted
- `threshold`: If the threshold scope has been granted, the values the user chose will be included here.

Errors

`NO_ADDRESS_ERROR`: Will be returned if there is no address present for the protocol / network requested.

`NOT_GRANTED_ERROR`: Will be returned if the permissions requested by the App were not granted.

Notes

The `threshold` scope is an optional suggestion for the wallet. Even if the wallet does not support this, it will still be compatible with the protocol.

2. Sign Payload

App requests that an arbitrary payload / message is signed.

Request

```
export interface SignPayloadRequest extends BaseMessage {
  type: MessageType.SignPayloadRequest;
  payload: string;
  sourceAddress: string;
}
```

- `payload`: The unsigned payload as a string. It does not have to be a valid tezos transaction
- `sourceAddress`: The address that will be used to select the keypair for signing

Response

```
export interface SignPayloadResponse extends BaseMessage {
  type: MessageType.SignPayloadResponse;
  signature: string; // Signature of the payload
}
```

`signature`: The signature of the payload in the format of "edsig..."

Errors

`NOT_GRANTED_ERROR`: Will be returned if the signature was blocked

`NO_PRIVATE_KEY_FOUND_ERROR`: Will be returned if the private key matching the sourceAddress could not be found

3. Operation Request

App sends partial Tezos operations to the wallet and the wallet will prepare the transaction and broadcast it.

"Partial" means, that only the necessary values need to be set. For example the `counter`, `gas_limit`, or `fee` will all be fetched and calculated by the wallet.

Request

```
export interface OperationRequest extends BaseMessage {
  type: MessageType.OperationRequest;
  network: Network;
  operationDetails: PartialTezosOperation[];
  sourceAddress: string;
}
```

`network`: Network on which the operation will be broadcast

`operationDetails`: Partial Tezos operations that may lack certain information like counter and fee. Those will be added by the wallet

`sourceAddress`: The address that will be used to select the keypair for signing

Response

```
export interface OperationResponse extends BaseMessage {
  type: MessageType.OperationResponse;
  transactionHash: string;
}
```

`transactionHash`: Hash of the broadcasted transaction

Errors

`PARAMETERS_INVALID_ERROR`: Will be returned if any of the parameters are invalid

`TOO_MANY_OPERATIONS`: Will be returned if too many operations are in the request and they were not able to fit into a single operation group

`BROADCAST_ERROR`: Will be returned if the user choses that the transaction is broadcast but there is an error (eg. node not available)

4. Broadcast Transactions

App requests a signed transaction to be broadcast.

Request

```
export interface BroadcastRequest extends BaseMessage {
  type: MessageType.BroadcastRequest;
  network: Network;
  signedTransaction: string;
}
```

`network`: Network on which the transaction will be broadcast

`signedTransaction`: Signed transaction that will be broadcast

Response

```
export interface BroadcastResponse extends BaseMessage {
  type: MessageType.BroadcastResponse;
  transactionHash: string;
}
```

`transactionHash`: Hash of the broadcasted transaction

Errors

`TRANSACTION_INVALID_ERROR`: Will be returned if the transaction is not parsable or is rejected by the node.

`BROADCAST_ERROR`: Will be returned if the user choses that the transaction is broadcast but there is an error (eg. node not available).

5. Error

In case of an error, the wallet will send back a dedicated message with an `errorType`. The actual error should be handled in the wallet and will not be sent back to the app.

```
export interface ErrorResponse extends BaseMessage {
  type: MessageType.Error;
  errorType: ErrorType;
}

export enum ErrorType {
  BROADCAST_ERROR = "BROADCAST_ERROR", // Broadcast | Operation Request: Will be returned if the user chooses that the transaction is broadcast but there is an error (eg. node not available).
  NETWORK_NOT_SUPPORTED = "NETWORK_NOT_SUPPORTED", // Permission: Will be returned if the selected network is not supported by the wallet / extension.
  NO_ADDRESS_ERROR = "NO_ADDRESS_ERROR", // Permission: Will be returned if there is no address present for the protocol / network requested.
  NO_PRIVATE_KEY_FOUND_ERROR = "NO_PRIVATE_KEY_FOUND_ERROR", // Sign: Will be returned if the private key matching the sourceAddress could not be found.
  NOT_GRANTED_ERROR = "NOT_GRANTED_ERROR", // Sign: Will be returned if the signature was blocked // (Not needed?) Permission: Will be returned if the permissions requested by the App were not granted.
  PARAMETERS_INVALID_ERROR = "PARAMETERS_INVALID_ERROR", // Operation Request: Will be returned if any of the parameters are invalid.
  TOO_MANY_OPERATIONS = "TOO_MANY_OPERATIONS", // Operation Request: Will be returned if too many operations were in the request and they were not able to fit into a single operation group.
  TRANSACTION_INVALID_ERROR = "TRANSACTION_INVALID_ERROR", // Broadcast: Will be returned if the transaction is not parsable or is rejected by the node.
  ABORTED_ERROR = "ABORTED_ERROR", // Permission | Operation Request | Sign Request | Broadcast: Will be returned if the request was aborted by the user or the wallet.
  UNKNOWN_ERROR = "UNKNOWN_ERROR", // Used as a wildcard if an unexpected error occured.
}
```

6. Disconnect

A message that indicates that the app or wallet wants to terminate the connection. This is a "fire and forget" message, no response is expected.

```
export interface DisconnectMessage extends BaseMessage {
  type: MessageType.Disconnect;
}
```

Transport Layers

This standard should be compatible with different transport layers. We will cover 2 common transport layers here, a `PostMessageTransport` (to talk to browser extensions) and a `P2PTransport` (communicating over a peer-to-peer network).

The `PostMessageTransport` should always be preferred over the `P2PTransport`, if available. If not, then it should fall back to the `P2PTransport`. The reason for this is that this might be the users preferred wallet, or that the browser extension has already established a P2P connection to his mobile/desktop wallet. In that case, the browser extension can be used as a relay between the App and the mobile/desktop wallet.

Serialisation

The serialisation used for the transport layer messages requires an algorithm which is well known and fast to encode/decode.

We decided on using `JSON` as the underlying structure and use `base58check` encoding. The encoding is well known in the tezos ecosystem and produces URL compatible strings.

```
const str = JSON.stringify(message);

return bs58check.encode(Buffer.from(str));
```

In the future, we might replace `JSON` with Tezos' Data Encoding. The reason we decided against using it is because there are no libraries available, and this could hinder adoption on new platforms.

Handshake (opening a channel)

Communication between app and wallet requires a secure setup. This is the proposed handshake process:

1. We define the communication/transport layer as 'channel'
2. App generates and stores (e.g. local storage) a channel asymmetric keypair. This keypair will be reused for all channels
3. App serialises public key, app name, app id and app icon in an handshake uri [1]
4. Handshake uri is either sent to the wallet (PostMessage) or shown as QR / clickable link (P2PTransport)
5. Wallet receives handshake uri
6. Wallet generates and stores (e.g. local storage) a channel asymmetric keypair. This keypair will be reused for all channels
7. Wallet serialises its own public key and info in a handshake uri
8. Wallet uses PostMessage or P2P network layer to reach out to app and send its own handshake uri
9. Wallet and app compute DH symmetric channel key
10. Symmetric encrypted acknowledgment message is sent
11. Channel is open

The handshake messages are different depending on the transport and will be described in the respective chapter.

Why not simply use directly the crypto address in the wallet?
There would be one big upside namely the wallet-app channel could be easily restored on any wallet that imported the crypto private key. This would also allow the user to accept/confirm messages coming from the app on different devices. The reason for the decision against such a solution is because it would imply that if an app sets up a channel once with a wallet it will always be able to send messages to it in future and since security is of utmost importance for this standard the decision has been made to have random channel keys which can be destroyed/ignored when an app turns rogue.

[1] app id and app icon can be spoofed by any dapp, however the browser extension should make sure that spoofing is avoided. Also the user always has to check him/herself if the shown URL matches the one in his/her address bar.

Closing a channel

Closing a channel will require one of the parties to send a channel close operation. After closing the channel the party will no longer listen to the channel (fire and forget).

Use Cases

The following use cases should be covered

DApp	Wallet	Important Use Case	Communication	Handshake
Desktop	Mobile	✅	P2P	QR scanning
Desktop	Desktop	✅	P2P	Desktop Deeplink
Desktop	Browser	✅	P2P (Possibly iFrame/WebRTC)	Browser Deeplink
Desktop	Extension	✅	PostMessage	PostMessage
Mobile	Mobile	✅	P2P	Mobile Deeplink
Mobile	Desktop	❌	P2P	Copy Paste
Mobile	Browser	❌	P2P	Copy Paste
Mobile	Extension	❌	P2P	Copy Paste

P2P Transport

The peer-to-peer transport should cover the following use cases:

- Same Device Desktop to Desktop (e.g. Galleon)
- Same Device Desktop to Hardware Wallet (e.g. Ledger)
- Same Device Mobile to Mobile (e.g. Cortez)
- Two Device Desktop to online Mobile (e.g. Wetez)
- Two Device Desktop to offline Mobile (e.g. AirGap)

The transport layer used for app-wallet communication we propose is using a decentralised messaging service. We are proposing to use the matrix protocol. Other projects in the blockchain space like Raiden have been using matrix successfully to cover very similar requirements. The long term vision is that the Tezos

reference node will eventually include such a messaging service by default, replacing the matrix network in the long term.

The main reasons why we propose to use a decentralised messaging service as the transport layer between app and wallet are:

- Decentralization
- Enables modern user experience
- Covers all of the scenarios described above
- Easy to integrate with all kind of wallets
- Minimal effort to "join" for a wallet
- Support for oracles (e.g. push service)

P2PTransport handshake

The following request has to be serialized using `JSON.stringify`, `base58check` encoded and displayed as a QR code or converted into a clickable link.

```
export interface P2PPairingRequest {
  name: string;
  icon?: string; // TODO: Should this be a URL or base64 image?
  appUrl?: string;
  publicKey: string;
  relayServer: string;
}

export interface P2PPairingResponse {
  name: string;
  icon?: string; // TODO: Should this be a URL or base64 image?
  appUrl?: string;
  publicKey: string;
}
```

The response is sent pack in the following format as a string (over the matrix protocol).

```
["@channel-open", recipient, encryptedMessage].join(":");
```

P2PTransport communication

The serialized message described above will be encrypted and sent through the channel to the recipient.

PostMessage Transport (Browser Extension)

Browser extension detection

The detection of a compatible chrome extension should be done by using a ping-pong approach.

After loading, the app will send a message containing `ping` via postMessage interface. The browser extension will reply with a `pong` as soon as possible. The app should wait for at least `200ms` until it can assume that there is no extension installed and fall back to the P2P Transport described above.

```
// App: Sending ping

const message = {
  target: ExtensionMessageTarget.EXTENSION,
  payload: "ping",
};

window.postMessage(message as any, "*");

// Extension: Sending pong

const message = {
  target: ExtensionMessageTarget.PAGE,
  payload: "pong",
};
```

Browser extension handshake

As described above, a secure connection channel has to be established. For this, the app will first send a `PairingRequest` to the extension. The extension will store the information it receives and send back its own information in the `PairingResponse`

```
export interface PostMessagePairingRequest {
  name: string;
  icon?: string; // TODO: Should this be a URL or base64 image?
  appUrl?: string;
  publicKey: string;
}

export interface PostMessagePairingResponse {
  name: string;
  icon?: string; // TODO: Should this be a URL or base64 image?
  appUrl?: string;
  publicKey: string;
}
```

After this handshake, the public keys have been exchanged and the subsequent communication is encrypted. This makes sure that no messages can be spoofed.

Browser extension communication

The serialized message described above will be encrypted and wrapped in an object before being sent via PostMessage.

```
export enum ExtensionMessageTarget {
  BACKGROUND = "toBackground",
  PAGE = "toPage",
  EXTENSION = "toExtension",
}

export interface ExtensionMessage<T, U = unknown> {
  target: ExtensionMessageTarget;
  encryptedPayload: T;
}
```

Deeplinks

Deeplinks allow us to cover more combinations of DApp (Desktop), DApp (mobile) with web-wallets, mobile wallets and desktop wallets. This is only used for the handshake, after the initial handshake the P2P Transport described above is used.

Deeplink Handshake

> TODO: This section is still under construction while we coordinate with the tzip-8 proposal.

We propose the same scheme that is used in tzip-8.

The only difference is that the payload is "wrapped" and an additional type parameter is added. This way the different tzip messages can be differentiated and wallets can handle the ones they support.

```
web+tezos://?type=tzip10&data=<data>
```

Test Cases

TBD: Test cases for an implementation are strongly recommended as are any proofs of correctness via formal methods.

Implementations

Beacon SDK

Apendix

Wallets implementing the standard

- Beacon Extension
- AirGap

Applications using the standard

- Beacon Landing Page
- Example DApp

Libraries working with implementations

- Taquito

Copyright