Design Doc: Sponsored Operations

Authors: Lin Oshitani Pavlo Khrystenko YunYan Chi
Created: Nov 15, 2023
Latest updates: Feb 22, 2024

Summary

Enable users to execute operations while having the fee paid by another account.

Context

Motivation

To achieve mass adoption, it is necessary to reach non-crypto users who may be unfamiliar with Tezos and don't necessarily own tez. Onboarding such users has a critical hurdle: They are required to use tez to cover the fees necessary for interacting with their assets, even if they don't currently hold any tez.

Consider, for instance, a gamer who is not interested in cryptocurrencies but still wishes to engage with their in-game assets recorded on Tezos. Similarly, imagine a user who seeks to use stable coins on Tezos but does not want to hold tez for fiscal or regulatory reasons. In both scenarios, the need for tez hinders the entry of these potential users.

Permits and Gas Station Networks (GSN) have been proposed to circumvent this issue, but they require individual dApps to integrate GSN support into their contract code. This demands the redeployment of existing contracts and explicit GSN support in new contract codes, which impedes broad adoption.

To address this challenge, we propose protocol-level support for users to initiate operations to any contract while having fees covered by a third party.

Example Use-cases

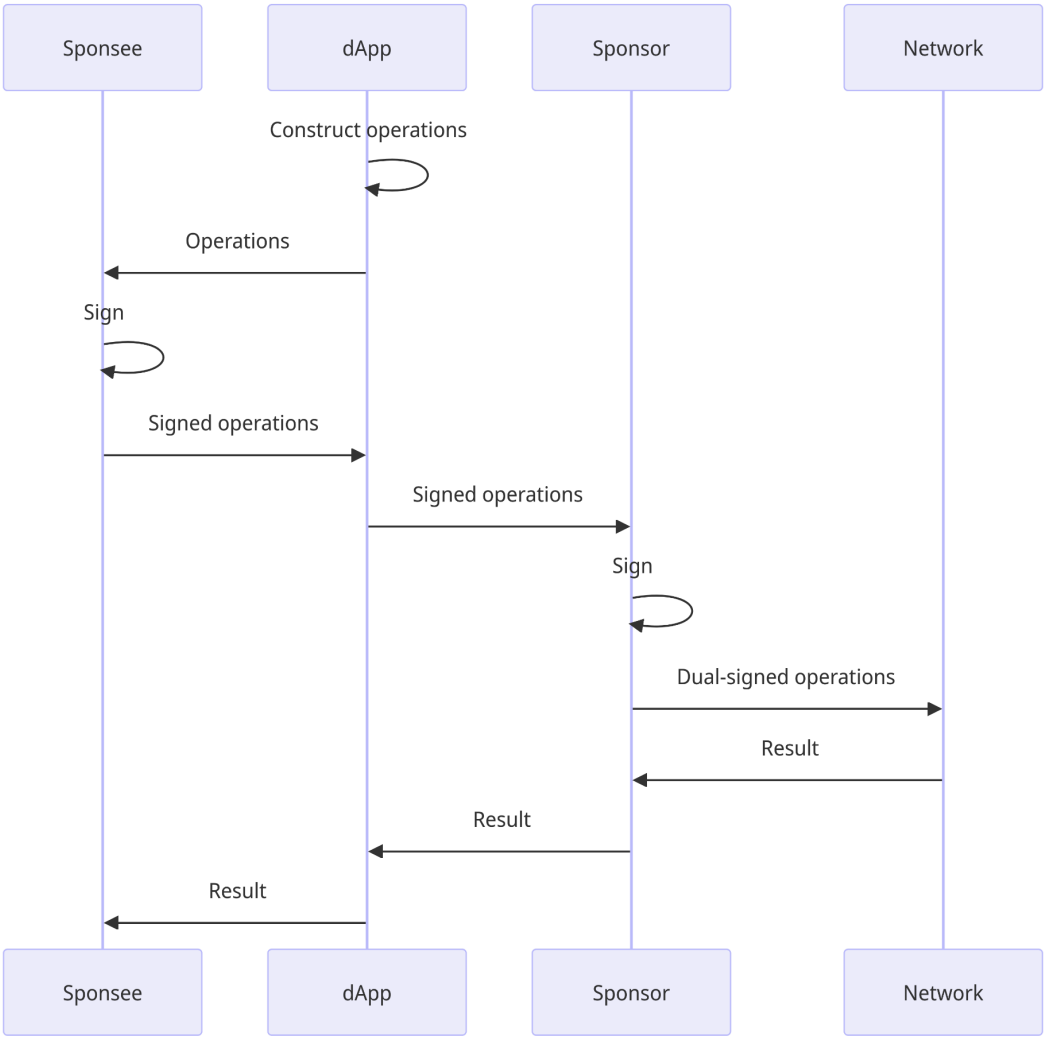
- Gaming dApps subsidize fees for interacting with their in-game assets.
- A third-party sponsor subsidizes fees in exchange for non-tez assets, such as on-chain stable coins or off-chain credit card payments.
- Organizations pay fees for specific marketing campaigns. E.g., Objkt.com gas fees paid by the objkt.com for a week, all deposits to Tezos 2.0 paid by the foundation, etc.
- Wallets subsidizes some amount of transaction fees per month as a marketing campaign.
- A third-party sponsor covers fees for withdrawal from sapling contracts. This is done to ensure that withdrawals are not linked to a user's implicit account through fee-payer information (reference).

Terminology

- **Sponsor**: The implicit account covering the fees for an operation.
- **Sponsee**: The implicit account seeking to perform operations with the fees paid by the sponsor.

Expected User-flows

Here is an example user-flow involving a sponsee, a dApp the sponsee interacts with, and a sponsor covering the fees for the interaction.



Requirements

Functional Requirements

- Support the sponsorship of at least the `Reveal` and `Transaction` manager operation.
- Support the sponsorship of batched operations.
- Support sponsees to construct operations without knowing who will sponsor them.
- Allow sponsors to sponsor operations from multiple sponsees in one block.

Non-functional Requirements

- Easy to support for wallets.
- Easy to support for indexers.
- Not break backward compatibility of existing transactions in terms of binary representation.
- Not overcomplicate the protocol code.
- Protect sponsees and sponsors from replay attacks.

Proposed Solution

Currently, when users submit operations to Tezos, they:

- Construct a batch of operations,
- sign the operations with their implicit account,
- attach the signature at the end of the binary encoded operation, and
- submit it to the L1.

Note that only *one* signature is signed in this process.

To enable sponsored operations, we need a way to

- add an additional signature, and,
- have a batch signed by *two* implicit accounts—one by the sponsee and one by the sponsor.

To maintain backward compatibility in binary encoding, we explore solutions that involve introducing a new manager operation that captures this additional signature.

Solution: The `Host` manager operation

Overview

Have the sponsor include a new `Host` operation that enables the sponsee's operation to be executed in the sponsor's batch while paying for the fees. In this approach, the sponsor will be the "host" and the sponsee will be the "guest".

An operation batch that uses `host` will look like this:

```
{
  branch = <branch>;
  contents = [
    host {
      source          = <source>;
```

```
      counter         = <counter>;
      fee             = <fee>;
      gas_limit       = <gas-limit>;
      storage_limit   = <storage-limit>;
      guest_source    = <guest-source>;
      guest_signature = <guest-signature>;
    };
    <rest-of-operations>
  ];
  signature = <signature>;
}
```

Where:

Field	Content
`<source>`	The host's public key hash. This account will pay for all fees in the batch.
`<counter>`	The host's counter.
`<fee>`	The fee for the host operation.
`<gas-limit>`	The gas limit for the `host` operation. Should cover the gas cost for the additional signature check of `<guest-signature>`.
`<storage-limit>`	The storage limit for the `host` operation. Should be zero as no storage is used.
`<guest-source>`	The guest's public key hash.
`<guest-signature>`	The signature by the guest against `<branch>+<rest-of-operations>`. When signing we use a special watermark that is unique to `guest_signature`.
`<rest-of-operations>`	The operations inside can contain operations with guest (= `<guest-source>`) as the `source`. The fees for these operations will be paid by the host (= `<source>`).
`<signature>`	The host's signature against `<branch>+contents`. When signing, the `<auth-signature>` will be zeroed out.This eliminates the necessity for the guest to sign before the host, and enables the host to sign before the guest, or even have both parties sign in

	pararell.

Example

Consider a scenario where A (the guest) sends 10 tez to B, and C (the host) covers the transaction fees in return for 0.1 usdtz. The final operation batch injected to L1 will look like this:

```
{
  branch =<branch>;
  contents = [
    host{ source=C; counter=47; guest_source=A; guest_signature=sig_A };
    tx_AB;
    tx_AC;
   ];
  signature = sig_C;
}
```

Where

```
tx_AB  = tx{ source=A; counter=3; dst=B; 10tez }
tx_AC  = tx{ source=A; counter=4; dst=C; 0.1usdtz }
sig_A  = signature(signer=A, watermark="\006", payload=<branch>+[tx_AB; tx_AC])
sig_C  = signature(
           signer=C,
           payload=
             <branch>+
             [host{source=C; counter=47; guest_source=A; guest_signature=000};
              tx_AB;
              tx_AC])
```

Note that:

- `sig_A` by A only signs against the operations after `host`.
- `sig_C` by C signs all operations, but with `guest_signature` zeroed out.
- Both `sig_A` and `sig_C` sign payload that includes the `<branch>`.
- `guest_signature` is signed with a distinct watermark (using `\006` here) to prevent `([tx_AB; tx_AC], guest_signature)` from being submitted to the network without being sponsored.

Batching of multiple guests

Recall the requirement:

- Allow sponsors to sponsor operations from multiple sponsees in one block.

This is crucial because being limited to sponsoring only one sponsee per block would be unacceptable in terms of throughput.

Unfortunately, the following straightforward solutions will not work:

- The sponsor submits multiple independent sponsored operations in one block.
    - This is not possible due to the 1M restriction.
- The sponsor manages multiple accounts and allocates different accounts per sponsee.
    - This is not ideal as it increases the workload for sponsors who may not be experts in blockchains (e.g. Gaming companies)

Hence, to enable sponsoring multiple sponsees in one block, we allow one batch to have multiple `hosts` to represent "a batch of guest batches". When there are multiple `host` operations in one batch, each `host` operation will only take effect until the next `host` operation.

Look at the example below where sponsor C is sponsoring two sponsees, A and B, within in one batch. Here

- only `tx_1` and `tx_2` will be able to have A as the source, and
- the signature `sig_A` will only be against `tx_1` and `tx_2`.

```
{
  branch =<branch>;
  contents = [
    host{source=C; guest_source=A; guest_signature=sig_A };
    tx {source=A; ...}; // tx_1
    tx {source=A; ...}; // tx_2
    host{source=C; guest_source=B; guest_signature=sig_B };
    tx {source=B; ...}; // tx_3
    …
   ];
  signature = sig_C;
}
```

OR Semantics

Due to how batch operations are implemented in Tezos, when a transaction from A fails, the whole batch will fail, including operations from B. This is problematic as B's operation is now failing due to an operation that is completely irrelevant to B.

To solve this, we introduce what we call *OR semantics* to batches that contain host operations. With OR semantics, If one sponsee's op fails, we just ignore operations from that sponsee but keep executing the rest. More concretely, it works as follows:

- When we encounter a host operation, we save the context at that point and proceed.
- If an operation fails before the next host, instead of failing the whole batch, we rollback to the context saved above and continue executing from the next host

Security

Replay Attack

As the guest's operation will include the counter of the guest, it should not be possible to replay the operations against the guest's will.

Operation Withholding Attack

Consider the scenario in which:

1. The sponsee signs an operation,
2. hands it to the sponsor,
3. but the sponsor delays posting the sponsee's operation for a considerable length.

This situation can be problematic, especially in DeFi use cases where the timing of operation injection plays a crucial role.

This is addressed by requiring:

- The `branch` to be included in the signature inside `host.`
    - Branches are typically set as the most recent block header hash at the time the operation was constructed.
- The included `branch` to be the same as the one used for the sponsor signature.

In the protocol, we consider operations that reference branches that are older than 1 hour to be invalid, hence sponsored operations should also expire within 1 hour (or shorter if you intentionally reference an older branch).

Sponsor Bypassing Attack

In our expected user flow, the sponsee first signs their operation and then passes it to the sponsor with the expectation that the operations will be sponsored prior to injection. But if the sponsor is malicious, they might attempt to inject the sponsee's operation without sponsoring it. This is possible because the `guest_signature` is signed against `branch+operations`, hence, it can be interpreted as a signature for the non-sponsored batch of `operations.`

To prevent this, we require `guest_signature` to be signed using a unique watermark that is specific to `guest_signature.` This way, the `guest_signature` will only be a valid signature if it is within the `guest_source` operation.

Caveats and Notes

Zero-tez accounts

The protocol requires the allocation of sources of manager operations before executing any operations with that source. This is problematic as we expect to sponsor accounts that do not hold any tez.

There are several ways to solve this problem:

- Have the sponsor deposit 1 mutez as a separate transaction before sponsoring any operations.
    - E.g. At the account creation of the game.
- Introduce “ghost accounts” where we enable operations to be executed for unallocated (or temporary allocated) accounts.

As the latter requires a lot of care and work, we propose to start with requiring the sponsor to deposit and consider “ghost accounts” in the future if needed.

Gas

The gas for the additional checking of sponsee signatures should be accounted for. I.e. each host operation should cost at least the gas cost of a signature check.

Replace-by-fee

Tezos supports managers to replace their previous manager operation by submitting an operation with a higher (in terms of fee/gas) fee. With sponsored operations, who should be able to do this? It would be weird if a sponsee can nullify the sponsor’s operation batch, which may contain ops from other sponsees. Hence, only the sponsor should be able to replace the sponsored batch via replace-by-fee.

Who Sets the Fees?

Suppose we have a sponsored operation (sponsor=A, sponsee=B) that looks like this:

```
[
  host {source=A; guest_source=B; fee=fee_0;};
```

```
  tx_1 {source=B; fee=fee_1;};
  tx_2 {source=B; fee=fee_2;};
]
```

There are two ways to set the fees (`fee_0, fee_1, and fee_2`):

1. The sponsee sets the fees `fee_1` and `fee_2` based on local simulation. The sponsor accepts it if it looks legitimate.
2. The sponsee sets the fees `fee_1` and `fee_2` to zero. The sponsor compensates for both fees inside `fee_0`.

The latter sounds more reasonable as it is the sponsor who will pay the fee, but both use cases are supported in this proposal.

Additionally, note that while the sponsee may not be aware of the fee and can set it to zero, the same approach cannot be taken for gas and storage limits. This is because gas and storage limits are accounted for per operation, not batch. Consequently, setting these values to zero would likely result in operation failure. Although there have been proposals to account for gas and storage per batch, such effort falls outside the scope of this document.

Multi-party Atomic Operations

The proposed solution supports atomic operations that involve two parties. This can have a broader use case than just sponsoring operations.

For example, an atomic transfer between of tez and usdtz between A and B can be done with:

```
[host(source=A; guest_source=B);
 tx{source=A; dst=B; 10tez};
 tx{source=B; dst=A; 10usdtz}]
```

Note that the proposed solution only supports two-party atomic operations and not general multi-party atomic operations. This is due to how we handle `hosts` in batches - Each guest only signs operations up until the next `host`, so operations can only be signed by up to two accounts (the sponsor and the sponsee specified in the previous `host`) and not more.

Having operations signed by three or more accounts is possible if we enable `host` to take a list of `pkh * signature` instead of just one pkh and one signature. However, it is unclear if it is worth the added complexity.

Related Work

Past work within Tezos under the name "Metatransactions":

- Document:
    - https://hackmd.io/9Y4hWIP4T1-FUvbREfBC0w
- Milestone:
    - https://gitlab.com/metastatedev/tezos/-/issues/107
- MR:
    - https://gitlab.com/tezos/tezos/-/merge_requests/2391

Arthur B mentions the idea of multiple managers signing a single batch here:

- Improving manager transaction batches - Research and Development - Tezos Agora Forum

Similar proposals/implementation in other chains:

- MultiverseX:
    - Relayed Transactions • MultiversX Docs
- Ethereum:
    - EIP-2711: Sponsored, expiring and batch transactions.
    - EIP-2733: Transaction Package
- Polkadot:
    - https://github.com/paritytech/polkadot-sdk/issues/266

Notes from design call

- Sponsee injects op that conflicts with sponsored batch.
    - Allow conflic, but not consider sponsee as manager
        - What happens if a sponsored op empties the account?
    - How bad?
        - When open, cannot have whitelist
    - In Mempool, prioritize sponsored , but multiple sponsored op can include objected
- OR sematics in the batch.
- Sponsee signs the host op (incl the sponsor's counter)