How to send multiple user operations in parallel

permissionless.js library lets you to send multiple transactions in parallel. This can be useful if you want to batch multiple calls in a single user operation or if you want to send user operations concurrently.

Batching Multiple Calls

To batch multiple calls in a single user operation, you can use thesendTransactions method.

```

consttransactionHash=awaitsmartAccountClient.sendTransaction({ calls: [ { to:"0xd8da6bf26964af9d7eed9e03e53415d37aa96045", value:parseEther("0.1"), data:"0x", }, { to:"0xd8da6bf26964af9d7eed9e03e53415d37aa96045", value:parseEther("0.1"), data:"0x", }, ], })

```

Sending Multiple User Operations in Parallel

In an Externally Owned Account (EOA), thenonce is a simple incrementing number. However, in smart accounts, thenonce consists of two components: akey and asequence :

- 192-bit “key”
- 64-bit “sequence”

For each uniquekey , the sequence must be incremented by 1 for each transaction. This means that if you send multiple transactions in parallel, you can use differentkeys for the parallel transactions.

Parallel Transactions Ordering

Important thing to note is that parallel transaction's ordering is not guaranteed. So execution of randomkey on chain could be:

- [key-C][sequence-0]
- [key-A][sequence-0]
- [key-B][sequence-0]

While execution ofsequence for a specifickey will always be in order. So the following is a valid order of execution:

1. [key-C][sequence-0]
2. [key-A][sequence-0]
3. [key-C][sequence-1]
4. [key-B][sequence-0]
5. [key-A][sequence-1]
6. [key-B][sequence-1]

Note: Pimlico bundler currently only support up to 10 parallel transactions. If you need to send more than 10 transactions in parallel feel free to contact us atsupport@pimlico.io .

In the example below, we use the current timestamp as thekey to send parallel transactions. You can use any other value as thekey .

```

import{ encodeNonce }from"permissionless/utils"

constparallelNonce1=encodeNonce({ key:BigInt(Date.now()), sequence:0n, })

consttransaction1=smartAccountClient.sendTransaction({ calls: [ { to:"0xd8da6bf26964af9d7eed9e03e53415d37aa96045", value:parseEther("0.1"), data:"0x", }, { to:"0xd8da6bf26964af9d7eed9e03e53415d37aa96045", value:parseEther("0.1"), data:"0x", }, ], nonce: parallelNonce1, })

constparallelNonce2=encodeNonce({ key:BigInt(Date.now()), sequence:0n, })

consttransaction2=smartAccountClient.sendTransaction({ calls: [ { to:"0xd8da6bf26964af9d7eed9e03e53415d37aa96045", value:parseEther("0.1"), data:"0x", }, { to:"0xd8da6bf26964af9d7eed9e03e53415d37aa96045", value:parseEther("0.1"), data:"0x", }, ], nonce: parallelNonce2, })

consthashes=awaitPromise.all([transaction1, transaction2])
```

```

Note: The flow described in this section is only applicable to smart accounts that are already deployed, if your account is not deployed, you may run into AA10 errors as multiple userOperations will contain the factory/factoryData fields which will cause conflicts.

This way, you can efficiently manage multiple transactions either sequentially or concurrently.
```