Gas Estimation

Titan의 생태계를 구성하는 어플리케이션을 개발하는 빌더들은 Titan SDK를 이용하여 L2에서 발생하는 트랜잭션 비용을 미리 산출해볼 수 있습니다.
L2 트랜잭션 비용은 'L2 execution fee + L1 security fee'로 계산할 수 있습니다. 각각에 대한 설명은 아래와 같습니다. Titan의 Optimistic Rollup은 L2에서 발생한 트랜잭션을 L1에 롤업하여 L2 네트워크의 보안 강도를 높여주는 구조를 채택하기 때문에 L2 execution fee 외에도 L1 security fee가 발생합니다.

- L2 execution fee = L2 gas price * L2 gas used (L2 gas used는 L2의 gasLimit 이하이어야 함)
- L1 security fee = L1 gas price * L1 gas used * L1 fee scalar
  - L1 fee scalar: L1 트랜잭션에 대한 가스비에 적용되는 배율 인수. 가스 가격이 급격하게 변동하는 상황에서 L1 fee scalar를 조정함으로써 현재 네트워크 조건에 동적으로 적응하고 거래 비용을 최적화할 수 있음

Gas estimation using Titan SDK

- L2 트랜잭션 수수료에 대한 더 자세한 내용은 "L2 fee" 섹션을 참고해주세요.
- 본 섹션에서는 예제 코드를 통해 Titan SDK로 L2 트랜잭션의 가스비를 산출하는 방법을 알아보겠습니다. 산출한 가스비와 실제 소모된 가스비를 서로 비교하여 얼마나 차이가 나는지 확인합니다.
- 직접 Titan SDK로 L2 트랜잭션 가스비를 계산할 수 있는 예제 코드는 여기에서 확인할 수 있습니다.
- Titan SDK와 ethers, dotenv, fs 라이브러리를 import합니다. dotenv 라이브러리는 프로젝트의 root 디렉토리 내 .env 파일을 읽어 환경 변수로 사용하는 기능을 지원합니다.

```
const ethers = require("ethers")
const titanSDK = require("@tokamak-network/titan-sdk")
require('dotenv').config()
```

- L2 네트워크에서 서명을 생성하고 트랜잭션을 전송하기 위해 ethers 라이브러리를 이용하여 provider와 wallet 인스턴스를 생성합니다.
  - `l2RpcProvider` : L2 네트워크를 위한 `ethers.providers.JsonRpcProvider` 인스턴스
  - `l2Wallet` : `ethers.Wallet` 을 사용하여 L2 네트워크의 지갑을 생성

```
const l2RpcProvider = titanSDK.asL2Provider(
  new ethers.providers.JsonRpcProvider(endpointUrl)
)
const l2Wallet = new ethers.Wallet(addHexPrefix(privateKey), l2RpcProvider)
```

- Titan SDK를 사용하여 가스 추정을 수행하는 함수 getEstimates()를 정의합니다. `provider` 는 Ethereum 공급자(provider) 객체이고, `tx` 는 추정할 트랜잭션의 객체입니다.
  - estimateTotalGasCost(): estimateL1GasCost()의 리턴값과 estimateL2GasCost()의 리턴값을 더하여 총 가스비를 Wei 단위로 산출
  - estimateL1GasCost(): L1에서 발생하는 가스비를 계산하여 Wei 단위로 산출
  - estimateL2GasCost(): L2에서 발생하는 가스비를 계산하여 Wei 단위로 산출
  - estimateL1Gas(): L1에서 사용한 가스의 양 (L1 gas used)를 계산

```
// Get estimates from the SDK
const getEstimates = async (provider, tx) => {
  return {
    totalCost: await provider.estimateTotalGasCost(tx),
    l1Cost: await provider.estimateL1GasCost(tx),
    l2Cost: await provider.estimateL2GasCost(tx),
    l1Gas: await provider.estimateL1Gas(tx)
  }
```

```
}
```

- 본 예제에서는 L2에서 발생하는 가스비를 산출하기 위해 Greeter 컨트랙트를 배포하고 컨트랙트 내 함수를 호출합니다.
  - Greeter 컨트랙트에 대한 설명은 아래와 같습니다.
    1. `constructor(string memory _greeting)` : Greeter의 생성자. `_greeting` 이라는 string 타입의 인자를 받아서 `greeting` 변수에 저장하며 생성자는 계약을 배포할 때 한 번만 실행
    2. `function greet() public view returns (string memory)` : `greeting` 변수의 값을 반환하는 함수. `view` 키워드는 이 함수가 상태를 변경하지 않는다는 것을 나타냄
    3. `function setGreeting(string memory _greeting) public` : `greeting` 변수의 값을 변경하는 `setGreeting` 함수. `_greeting` 이라는 인자를 받아서 `greeting` 변수를 업데이트

```
// Greeter.sol
// SPDX-License-Identifier: Unlicense
pragma solidity ^0.8.0;

contract Greeter {
  string greeting;

  constructor(string memory _greeting) {
//    console.log("Deploying a Greeter with greeting:", _greeting);
    greeting = _greeting;
  }

  function greet() public view returns (string memory) {
    return greeting;
  }

  function setGreeting(string memory _greeting) public {
//    console.log("Changing greeting from '%s' to '%s'", greeting, _greeting);
    greeting = _greeting;
  }
}
```

- Greeter 컨트랙트의 abi와 bytecode와 L2 signer를 파라미터로 `new ethers.ContractFactory` 함수를 실행하여 팩토리 인스턴스를 생성합니다. 그리고 Greeter 스마트 계약을 배포합니다. "Hi!" 라는 초기 인사말을 생성자에 전달하여 배포될 인스턴스를 초기화합니다.

```
const factoryGreeter = new ethers.ContractFactory(greeterJSON.abi, greeterJSON.bytecode

const greeter = await factoryGreeter.deploy("Hi!")
await greeter.deployed()
console.log(`Greeter address: ${greeter.address}`)
```

- greeter의 setGreeting 함수를 호출하여 “Hi!”를 “Hello!”로 변경합니다. 이전에 정의한 `getEstimates` 함수를 호출하여 L1과 L2에서 발생한 가스비를 추정하여 계산합니다. `signer.provider` 는 이더리움 공급자(provider)를 나타내며, `fakeTx` 는 추정할 트랜잭션 객체입니다. 추정 결과는 `estimated` 변수에 할당됩니다. `greeter` 컨트랙트 `setGreeting` 함수를 호출하여 실제 가스 소비량을 추정합니다. 추정된 가스 소비량은 `estimated.l2Gas` 에 할당됩니다.

```
const greeting = "Hello!"
```

```
const fakeTxReq = await greeter.populateTransaction.setGreeting(greeting)
const fakeTx = await signer.populateTransaction(fakeTxReq)
console.log("About to get estimates")
let estimated = await getEstimates(signer.provider, fakeTx)
estimated.l2Gas = await greeter.estimateGas.setGreeting(greeting)
```

- `greeter` 인스턴스의 `setGreeting` 함수를 호출하여 `greeting` 을 전달하여 실제 트랜잭션을 생성합니다. 트랜잭션의 gas price를 설정하고 `wait()` 함수를 실행합니다. 트랜잭션이 처리될 때까지 대기하고, 처리가 완료되면 트랜잭션 실행 결과를 반환합니다.

```
console.log("About to create the transaction")
realTx = await greeter.setGreeting(greeting)
realTx.gasPrice = realTx.maxFeePerGas;
console.log("Transaction created and submitted")
realTxResp = await realTx.wait()
```

- 예상 가스비와 실제 가스비는 예시로 아래와 같이 나타낼 수 있습니다. SDK로 예상한 가스비와 실제 가스비를 각각 총 가스비, L1 가스비, L2 가스비로 나눠서 확인하고, L1과 L2에서 각각 소모된 가스의 양과 그 차이도 체크해볼 수 있습니다.

```
Estimates:
   Total gas cost:           8842620676 wei
      L1 gas cost:               620676 wei
      L2 gas cost:           8842000000 wei

Real values:
   Total gas cost:           8839120640 wei
      L1 gas cost:               620676 wei
      L2 gas cost:           8838499964 wei

L1 Gas:
      Estimate:       4926
          Real:       4926
    Difference:          0

L2 Gas:
      Estimate:      35368
          Real:      35354
    Difference:        -14
```