

네오(NEO)

해당 번역본은 이용자가 디지털자산 관련 정보를 편리하게 확인할 수 있도록 단순 참고용으로 번역된 것이며, 투자 권유를 목적으로 하지 않습니다.
그 내용이 정확하지 않을 수 있으므로 정확한 정보 습득을 위해서는 원문을 참고하시거나 원문 작성 주체 측에 문의하시기 바랍니다.
두나무는 해당 번역본 내용의 진실성 및 정확성을 보장하지 않고, 제공된 정보에 의한 투자 결과에 대하여 어떠한 책임도 지지 않으며, 투자 손실은 투자자에게 귀속됩니다.

개요

여기에서는 Neo로 개발할 수 있도록 돕는 문서를 제공합니다.
이 문서들을 통해 Neo 블록체인을 **개념적 수준**에서 학습하고, **Neo 기술 스택의 기초**를 이해하며, 더 나아가 **복잡한 애플리케이션과 활용 사례**를 다루는 심화 주제까지 탐구할 수 있습니다.
Neo는 처음부터 **오픈소스 커뮤니티의 노력**으로 발전해 왔습니다.
Neo가 지속적으로 개선되고 진화할 수 있도록 여러분의 아이디어와 지식을 기여해 주시기를 초대합니다.

- 필요하다고 생각되는 부분이 있다면 언제든지 새로운 주제를 제안할 수 있습니다.
- 직접 콘텐츠를 추가하거나 예시를 제공할 수도 있습니다.
- 모든 문서는 GitHub(<https://github.com>)을 통해 편집할 수 있습니다.

시작하기 전에

이 튜토리얼에서는 Neo 블록체인에서 **NEP17 자산을 발행**하는 예제를 다룹니다.
여기에는 다음과 같은 작업이 포함됩니다:

1. 개발 환경의 설정 및 구성
2. 스마트 컨트랙트 컴파일
3. 스마트 컨트랙트 배포
4. 프라이빗 Neo 체인에서의 호출

시스템 환경

Neo-CLI는 다음과 같이 테스트된 환경에서 실행할 것을 권장합니다:

- Windows 10
- Ubuntu 20.04
- macOS Big Sur / 버전 11.1

Neo-CLI 다운로드

Neo-CLI 설치 가이드(<https://github.com>)를 참고하여 GitHub에서 Neo-CLI 패키지를 다운로드하고 설치하십시오.

로컬 네트워크 설정

프라이빗 체인 설정

Neo는 개발, 디버깅 및 테스트 목적을 위한 테스트넷을 제공합니다.

또한, 보다 유연하게 많은 테스트 토큰을 사용할 수 있도록 **자체 프라이빗 체인**을 설정할 수도 있습니다.

아래 옵션 중 하나를 선택할 수 있습니다:

- 단일 노드로 프라이빗 체인 설정(<https://github.com>)
 - 다중 노드로 프라이빗 체인 설정(<https://github.com>)
- 위의 링크에서 제공되는 지침을 참고하여 프라이빗 체인을 설정하고, 제네시스 블록에서 NEO와 GAS를 인출하십시오.

지갑 파일 준비

이제 스마트 컨트랙트 배포에 사용할 새로운 지갑 파일을 생성해 보겠습니다:

1. `0.json`이라는 이름의 새로운 지갑 파일을 생성하고, 나중에 사용할 수 있도록 기본 주소를 복사합니다.
2. 제네시스 블록에서 NEO와 GAS를 인출한 지갑을 열고, 해당 지갑의 모든 자산을 `0.json`으로 전송한 후 거래가 확인될 때까지 기다립니다.
3. `0.json`을 열어 자산이 정상적으로 수신되었는지 확인합니다.

컨트랙트 개발

프라이빗 체인 설정과 노드 구성을 완료했습니다.

이제 환경을 구성하고, C#을 사용하여 **NEP17 컨트랙트**를 작성 및 컴파일하는 과정을 안내하겠습니다.

아래 단계는 Windows, macOS, Ubuntu 등 여러 시스템 플랫폼에서 동일하게 적용됩니다.

도구 설치

1. Visual Studio Code(<https://code.visualstudio.com/>) 다운로드 및 설치
2. .NET 8.0 SDK(<https://dotnet.microsoft.com/en-us/download/dotnet/8.0>) 다운로드 및 설치
3. 명령줄을 실행한 후, SDK가 정상적으로 설치되었는지 확인하기 위해 다음 명령어를 입력합니다:

```
dotnet --list-sdks
```

문제가 없다면 SDK 버전 번호가 표시됩니다.

컨트랙트 템플릿 설치

Neo.SmartContract.Template(<https://github.com>)은 Neo 스마트 컨트랙트 개발 시 사용하는 프로젝트 템플릿입니다. 템플릿을 설치한 후, 터미널 또는 Visual Studio를 통해 Neo 스마트 컨트랙트 프로젝트를 생성할 수 있습니다.

템플릿 설치

```
dotnet new install Neo.SmartContract.Template
```

모든 dotnet 템플릿 목록 확인

```
dotnet new list
```

Neo.SmartContract.Template(<https://github.com>)을 설치한 후에는 다음과 같은 기본 템플릿을 사용할 수 있습니다:

- neocontractowner - GetOwner 및 SetOwner 메서드를 포함한 소유자(Owner) 표준 컨트랙트 템플릿.
- neocontractoracle - OracleRequest를 사용하는 컨트랙트 템플릿.
- neocontractnep17 - Mint 및 Burn 메서드를 포함한 NEP-17 컨트랙트 템플릿.

참고

기존에 사용되던 *neocontract* 템플릿은 *neocontractowner*로 이름이 변경되었습니다.

터미널에서 템플릿을 사용하여 프로젝트 생성

```
dotnet new neocontractnep17
```

프로젝트 이름은 기본적으로 현재 디렉토리의 이름을 따릅니다. 또한 `-n, --name <name>` 옵션을 사용하여 프로젝트 이름을 지정할 수도 있습니다. 예: `dotnet new neocontractnep17 -n MyFirstContract`

컨트랙트 파일 컴파일

터미널 인터페이스에서 프로젝트 경로로 이동한 후, 다음 명령어를 실행하여 컨트랙트를 빌드합니다:

```
dotnet build
```

or

```
nccs
```

관련 컨트랙트 파일들은 컨트랙트 프로젝트 디렉토리의 `bin\sc` 경로 아래에 출력됩니다.

참고 자료

- 컨트랙트 작성에 대한 더 많은 정보는 기본 개념(<https://github.com>)을 참고하십시오.

- Neo N3와 Neo Legacy 컨트랙트 간의 차이점에 대한 정보는 Neo Legacy와의 차이점(<https://github.com>)을 참고하십시오.

컨트랙트 배포 및 호출

이전 섹션에서는 **NEP17 컨트랙트 파일**(NEP17.nef)과 **컨트랙트 설명 파일**(NEP17.manifest.json)을 컴파일했습니다. 이제 Neo-CLI를 사용하여 컨트랙트를 **배포 및 호출**하는 과정을 진행하겠습니다.

컨트랙트 배포

Neo-CLI에서 배포 명령어 `deploy <nefFilePath> [manifestFile]`를 입력합니다. 예시는 다음과 같습니다:

```
deploy NEP17.nef
```

Or

```
deploy NEP17.nef NEP17.manifest.json
```

명령어가 실행되면 NEP17 컨트랙트가 배포되며, 관련 수수료는 시스템에 의해 자동으로 차감됩니다.

```
neo> deploy NEP17.nef
Script hash: 0xb7f4d011241ec13db16c0e3484bdd5dd9a536f26
Gas: 10.1477624

Signed and relayed transaction with
hash=0xe03aade81fb96c44e115a1cc9cfe984a9df4a283bd10aa0aefa7ebf3e296f757
```

더 자세한 내용은 스마트 컨트랙트 배포(<https://github.com>)를 참고하십시오.

컨트랙트 호출

이제 Neo-CLI 명령어 `invoke`를 사용하여 컨트랙트를 호출할 수 있습니다:

```
invoke <scriptHash> <operation> [contractParameters=null] [witnessAddress=null]
```

예시:

```
invoke 0xb7f4d011241ec13db16c0e3484bdd5dd9a536f26 symbol
```

성공적으로 실행되면 다음과 같은 정보가 출력됩니다:

```
Invoking script with: '10c00c046e616d650c14f9f81497c3f9b62ba93f73c711d41b1eeff50c2341627d5b52'  
VM State: HALT  
Gas Consumed: 0.0103609  
Evaluation Stack: [{"type":"ByteArray","value":"VG9rZW5TeW1ib2w="}]  
  
relay tx(no|yes):
```

설명:

- **VM State:** HALT는 VM이 성공적으로 실행되었음을 의미하며, FAULT는 실행 중 예외로 인해 VM이 종료되었음을 의미합니다.
- **Evaluation Stack:** 컨트랙트 실행 결과를 나타내며, 값이 문자열 또는 ByteArray일 경우 Base64로 인코딩됩니다.
- 데이터 포맷 변환은 여기 (<https://www.base64decode.org/>)에서 수행할 수 있습니다.
VG9rZW5TeW1ib2w= ⇒ **TokenSymbol**

자세한 내용은 Invoking Smart Contract(<https://github.com>)를 참고하세요.

Neo 소개

Neo는 오픈 소스 커뮤니티 주도 블록체인 플랫폼입니다.

Neo는 탈중앙화 애플리케이션(dApp) 개발을 위한 가장 완전한 기능을 갖춘 블록체인 플랫폼입니다. Neo는 스마트 컨트랙트를 통해 자산의 디지털화와 관리 자동화를 구현할 수 있게 합니다. C#, Python, Go, TypeScript, Java 등 다중 언어 지원을 통해 개발자가 가장 익숙한 언어로 손쉽게 개발할 수 있습니다. 또한 Neo는 탈중앙화 스토리지, 오라클, 도메인 네임 서비스(DNS)와 같은 강력한 네이티브 인프라를 제공하여 차세대 인터넷(Next-Gen Internet)의 견고한 기반을 구축합니다.

Neo에 대해 더 알아보기(<https://github.com>)

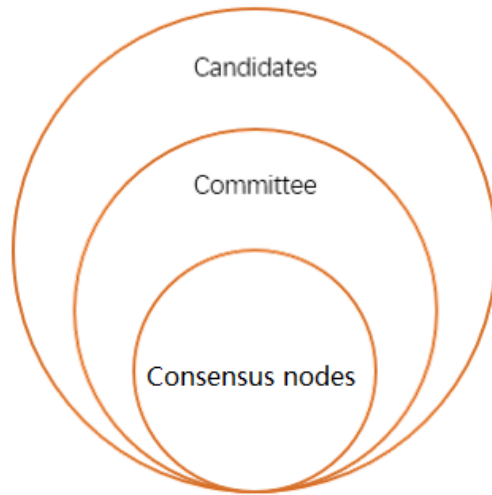
거버넌스 및 인센티브

Neo N3는 커뮤니티가 주도하는 오픈 플랫폼으로서, 온체인 거버넌스 모델을 통해 새로운 핵심 역할과 인센티브 모델을 도입하여 모든 참여자가 공정하게 보상받을 수 있도록 합니다.

거버넌스 전략

새로운 거버넌스 모델은 후보자(candidates), 위원회 구성원(committee members), 합의 노드(consensus nodes)로 구성됩니다. 이 중 위원회는 메인넷의 파라미터 조정 및 온체인 환경의 유지 관리 역할을 담당하며, 합의 노드는 트랜잭션 패키징과 블록 생성 역할을 담당합니다. 위원회 구성원과 합의 노드는 가장 많은 표를 얻은 일정 수의 후보자들 중에서 선출됩니다.

이들의 관계는 아래 그림으로 설명할 수 있습니다. 위원회 구성원과 합의 노드 간에 명시적인 관계는 없지만, 기본 위원회 구성원 수(21명)가 합의 노드 수(7명)보다 많기 때문에 일반적으로 합의 노드는 위원회 구성원의 부분집합으로 볼 수 있습니다.



후보자(Candidates)

모든 개인 또는 기관은 후보자로 등록하여 투표자들로부터 지지를 받아 위원회 구성원이 될 수 있습니다. 등록 트랜잭션이 온체인에 기록되면, NEO 보유자는 Neo 생태계의 올바른 결정을 내릴 수 있다고 판단되는 후보자에게 투표할 수 있습니다. 후보자가 받은 표 수는 지난 에포크(epoch) 동안 해당 후보자에게 투표한 모든 지갑 주소의 NEO 토큰 수를 합산하여 21블록마다 계산됩니다. 위원회 구성원이 진정으로 커뮤니티의 지지를 받았는지를 보장하기 위해, 전체 NEO 토큰의 20% 이상이 투표에 사용된 경우에만 선거가 유효합니다. 이후 가장 많은 표를 받은 일정 수의 후보자가 자동으로 위원회 구성원과 합의 노드로 선출됩니다.

위원회(Committee)

위원회 구성원은 상위 21명의 최다 득표 후보자 중에서 선출됩니다. 후보자의 득표 수는 21블록마다 계산되므로, 투표자는 위원회의 성과에 따라 유연하게 투표 전략을 조정할 수 있습니다. 어떠한 제안이 승인되기 위해서는, 위원회 구성원의 50% 이상이 동의해야 하며, 이후 블록체인 파라미터를 온체인에서 갱신하기 위한 트랜잭션에 서명합니다.

또한 위원회는 다음과 같은 다양한 노드 역할을 임명할 수 있습니다:

- 오라클(Oracle) 노드
- StateRoot 합의 노드
- NeoFS 이너링(inner ring) 노드

합의 노드(Consensus Nodes)

21명의 위원회 구성원 중 최다 득표 상위 7명은 합의 노드(consensus nodes)로서 역할을 수행하여 트랜잭션 활동을 촉진하고 Neo 블록체인의 보안을 강화합니다. 이들은 새로운 블록 제안을 시작하고 블록을 생성할 권한을 가집니다. 위원회 구성원과 마찬가지로, 합의 노드는 21블록마다 갱신됩니다.

인센티브(Incentives)

Neo N3는 Neo Legacy로부터 이어받은 이중 토큰 메커니즘을 채택하고 있으며, NEO는 거버넌스용, GAS는 지불용으로 사용됩니다.

NEO

NEO는 최대 공급량이 1억 개로 고정되어 있으며, 최소 단위가 1로 나누어지지 않습니다. NEO 보유자는 Neo 네트워크의 소유자이자 관리자로서, 네트워크 내에서 투표 트랜잭션을 생성함으로써 합의 노드 선출, 합의 전략 조정, 수수료 모델 조정 등과 같은 관리 권한을 행사할 수 있습니다. 또한 보유한 NEO의 양에 따라 해당하는 GAS를 청구할 수 있습니다.

GAS

GAS는 Neo 네트워크 자원 제어를 실현하기 위한 연료 토큰(fuel token)으로, 최소 단위는 0.00000001이며 이를 'Datoshi'라고 부릅니다.

사용자는 GAS를 청구(claim)하거나 구매(purchase)를 통해 획득할 수 있으며, 네트워크 사용 시 트랜잭션 전송, 자산 등록, 자산 발행, DApp 실행 등의 행위에 대해 일정량의 GAS를 네트워크 수수료로 지불해야 합니다.

조금 더 구체적으로 설명하자면, GAS의 최소 단위인 Datoshi는 다음과 같이 정의됩니다:

1 Datoshi = $1e-8$ GAS.

GAS의 더 큰 단위는 Datoshi 단위로도 표현될 수 있습니다. 예를 들어,

1 Kdatoshi(천 Datoshi) = $1e-5$ GAS,

1 Mdatoshi(백만 Datoshi) = $1e-2$ GAS,

즉, 100,000,000 Datoshi = 100,000 Kdatoshi = 100 Mdatoshi = 1 GAS입니다.

이러한 단위 체계는 Neo 네트워크에서 정밀한 계산과 트랜잭션을 가능하게 합니다.

Neo Legacy와 달리, Neo N3의 GAS에는 공급 한도가 없으며 트랜잭션의 시스템 수수료는 소각(burn)됩니다.

GAS 분배 규칙(GAS Distribution Rule)

초기 설정에서, 블록당 5개의 GAS 토큰이 생성되며, 이는 Neo 위원회(합의 노드 및 후보 노드 포함), NEO 투표자, 그리고 모든 NEO 보유자에게 분배됩니다.

생성된 GAS는 아래와 같은 규칙에 따라 분배됩니다.

NEO 보유자 - 10%

Neo Legacy와 마찬가지로, 이 부분의 GAS는 NEO 보유자에게 자동으로 분배되지 않습니다.

이는 NEO 보유자가 NEO를 전송하거나 투표를 수행한 이후에, NEO 보유 기간(holding period)에 따라 계산되어 NEO 보유자의 지갑에 분배됩니다.

위원회 및 합의 노드 - 10%

남은 10%는 Neo 블록체인의 관리와 운영에 기여한 21명의 위원회 구성원에게 보상으로 지급됩니다.

매 21블록(에포크, Epoch)마다 위원회 구성원에 대한 투표가 다시 계산되며, 인센티브 비율이 새로 선출된 구성원들에게 재분배됩니다.

또한, 현재 블록에 포함된 트랜잭션의 네트워크 수수료는 스피커(speaker)에게 지급됩니다.

투표자(Voters) - 80%

생성된 GAS의 대부분(80%)은 NEO 보유자가 위원회 구성원에게 투표하도록 장려하기 위한 인센티브로 사용됩니다.

이 보상의 일부는 실제로 선출된 위원회 구성원에게 성공적으로 투표한 NEO 보유자에게만 지급되며, 각 에포크(21블록)마다 계산 및 분배됩니다.

다시 말해, 이 보상 부분은 28로 나누어집니다(위원회 구성원 21명, 합의 노드 7명 기준).

선출된 합의 노드에게 투표한 NEO 보유자는 해당 부분의 $\frac{2}{28}$ 을,

합의 노드가 아닌 선출된 위원회 구성원에게 투표한 NEO 보유자는 $\frac{1}{28}$ 을 보상으로 받습니다.

NEO 토큰 모델(NEO Token Models)

네이티브 토큰(Native Token)

Neo 시스템에는 두 가지 종류의 네이티브 토큰이 정의되어 있습니다: **NEO**와 **NeoGas(GAS)**입니다.

NEO는 거버넌스(Governing) 토큰으로, 보유자는 Neo 네트워크의 관리에 참여할 수 있습니다.

이는 합의 노드 투표, 네트워크 파라미터 수정 등의 행위를 포함합니다.

NEO의 총 발행량은 1억 개이며, 최소 단위는 1로 나누어지지 않습니다.

NEO는 제네시스 블록(Genesis block)에 등록되어 있으며, 예비 검증자(standby validators)의 다중 서명 주소(multi-signature address)에 저장됩니다.

GAS는 Neo 네트워크 자원 제어를 실현하기 위한 연료 토큰(fuel token)으로, 최소 단위는 0.00000001이며 이를 **Datoshi**라고 부릅니다. 사용자는 GAS를 청구(claim)하거나 구매(purchase)를 통해 획득할 수 있습니다. Neo 네트워크를 이용할 때는 트랜잭션 전송, 자산 등록, 자산 발행, DApp 실행 등과 같은 네트워크 사용 행위에 대해 일정량의 GAS를 수수료로 지불해야 합니다.

좀 더 구체적으로 설명하면, GAS의 최소 단위는 Datoshi이며 다음과 같이 정의됩니다:

1 Datoshi = $1e-8$ GAS.

GAS의 더 큰 단위도 Datoshi로 표현할 수 있습니다.

예를 들어,

1 Kdatoshi(천 Datoshi) = $1e-5$ GAS,

1 Mdatoshi(백만 Datoshi) = $1e-2$ GAS.

즉, 100,000,000 Datoshi = 100,000 Kdatoshi = 100 Mdatoshi = 1 GAS입니다.

이 단위 체계는 Neo 네트워크 내에서 정밀한 계산 및 트랜잭션을 가능하게 합니다.

Neo N3 제네시스 블록은, 제네시스 시점에 Neo Legacy 체인에서 유통 중인 GAS 토큰의 총량과 정확히 일치하는 양의 GAS를 발행합니다.

GAS 생성 및 분배에 대한 자세한 내용은 [거버넌스 및 인센티브](#)를 참고하세요.

NEP17 자산(토큰)

NEP17 토큰은 스마트 컨트랙트를 통해 발행 및 관리되어야 합니다.

이들의 정보는 스마트 컨트랙트의 스토리지에 저장되며, 계정(Account) 모델이 사용됩니다.

자세한 내용은 NEP-17(<https://github.com>)을 참고하세요.

지갑(Wallets)

지갑은 Neo의 기본 구성 요소이자 사용자가 Neo 네트워크에 접근하기 위한 인터페이스입니다.

지갑은 트랜잭션 전송, 컨트랙트 배포, 자산 등록 등과 같은 트랜잭션 관련 작업을 담당합니다.

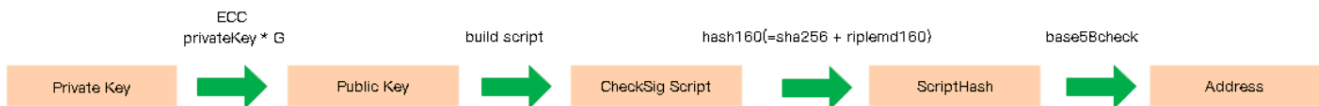
개발자는 Neo 지갑을 자신만의 아이디어에 맞게 재설계하고 수정할 수 있지만,

아래의 규칙과 패턴은 반드시 따라야 합니다.

계정(Accounts)

Neo에서 계정(Account)은 스마트 컨트랙트이며, 주소(Address)는 컨트랙트 스크립트를 나타냅니다.

다음 순서도는 개인 키(Private Key)에서 공개 키(Public Key)를, 그리고 주소(Address)를 도출하는 과정을 보여줍니다:



개인 키(Private Key)

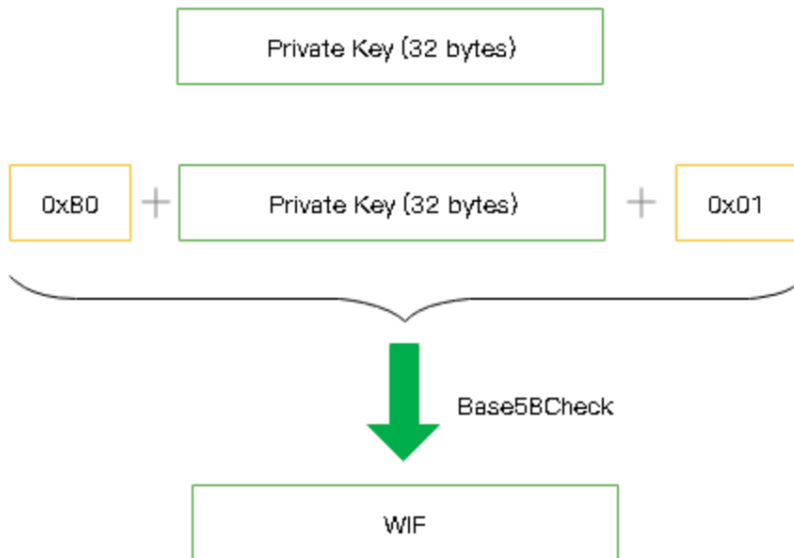
개인 키는 1과 n 사이의 임의의 값으로 생성됩니다.

여기서 n은 상수이며, 2^{256} 보다 약간 작은 값입니다.

일반적으로 개인 키는 256비트(32바이트) 숫자로 표현됩니다.

Neo에서 개인 키는 다음 두 가지 주요 인코딩 형식을 가집니다:

- **Hexstring 형식**
Hexstring 형식은 16진수 문자를 사용하여 바이트 배열을 문자열로 표현하는 방식입니다.
- **WIF 형식**
WIF 형식은 원래의 32바이트 데이터에 접두사 0x80과 접미사 0x01을 추가한 뒤, Base58Check 인코딩을 수행하여 문자열을 생성하는 방식입니다.



예시:

형식 (Format)	값 (Value)
byte[]	[0xc7,0x13,0x4d,0x6f,0xd8,0xe7,0x3d,0x81,0x9e,0x82,0x75,0x5c,0x64,0xc9,0x37,0x88,0xdb,0x09,0x61,0x92,0x9e,0x02,0x5a,0x53,0x36,0x3c,0x4c,0xc0,0x2a,0x69,0x62]
hexstring	c7134d6fd8e73d819e82755c64c93788d8db0961929e025a53363c4cc02a6962
wif	L3tgppXLgdaeqSGSFw1Go3skBiy8vQAM7YMXvTHsKQtE16PBncSU

공개 키(Public Key)

공개 키는 개인 키를 기반으로 ECC(Elliptic Curve Cryptography, 타원 곡선 암호) 알고리즘을 통해 계산된 점 (X, Y)입니다. X, Y 좌표는 각각 32바이트 데이터로 표현할 수 있습니다. Neo는 비트코인과 달리 ECC 알고리즘의 곡선으로 **secp256r1**를 사용합니다. Neo에는 다음 두 가지 공개 키 형식이 존재합니다:

- **비압축 공개 키(Uncompressed Public Key)**
0x04 + X (32 bytes) + Y (32 bytes)
- **압축 공개 키(Compressed Public Key)**
0x02/0x03 + X (32 bytes)

예시:

형식 (Format)	값 (Value)
개인 키 (Private Key)	c7134d6fd8e73d819e82755c64c93788d8db0961929e025a53363c4cc02a6962
공개 키 (압축형)	035a928f201639204e06b4368b1a93365462a8ebbf0b8818151b74faab3a2b61a
공개 키 (비압축형)	045a928f201639204e06b4368b1a93365462a8ebbf0b8818151b74faab3a2b61a35dfabcb79ac492a2a88588d2f2e73f045cd8af58059282e09d693dc340e113f

주소(Address)

주소는 공개 키가 일련의 변환 과정을 거쳐 생성된 문자와 숫자의 조합 문자열입니다. 이 절에서는 공개 키에서 Neo 주소로 변환되는 과정을 설명합니다.



참고 (NOTE)

Neo N3에서는 주소 스크립트가 변경되었습니다.

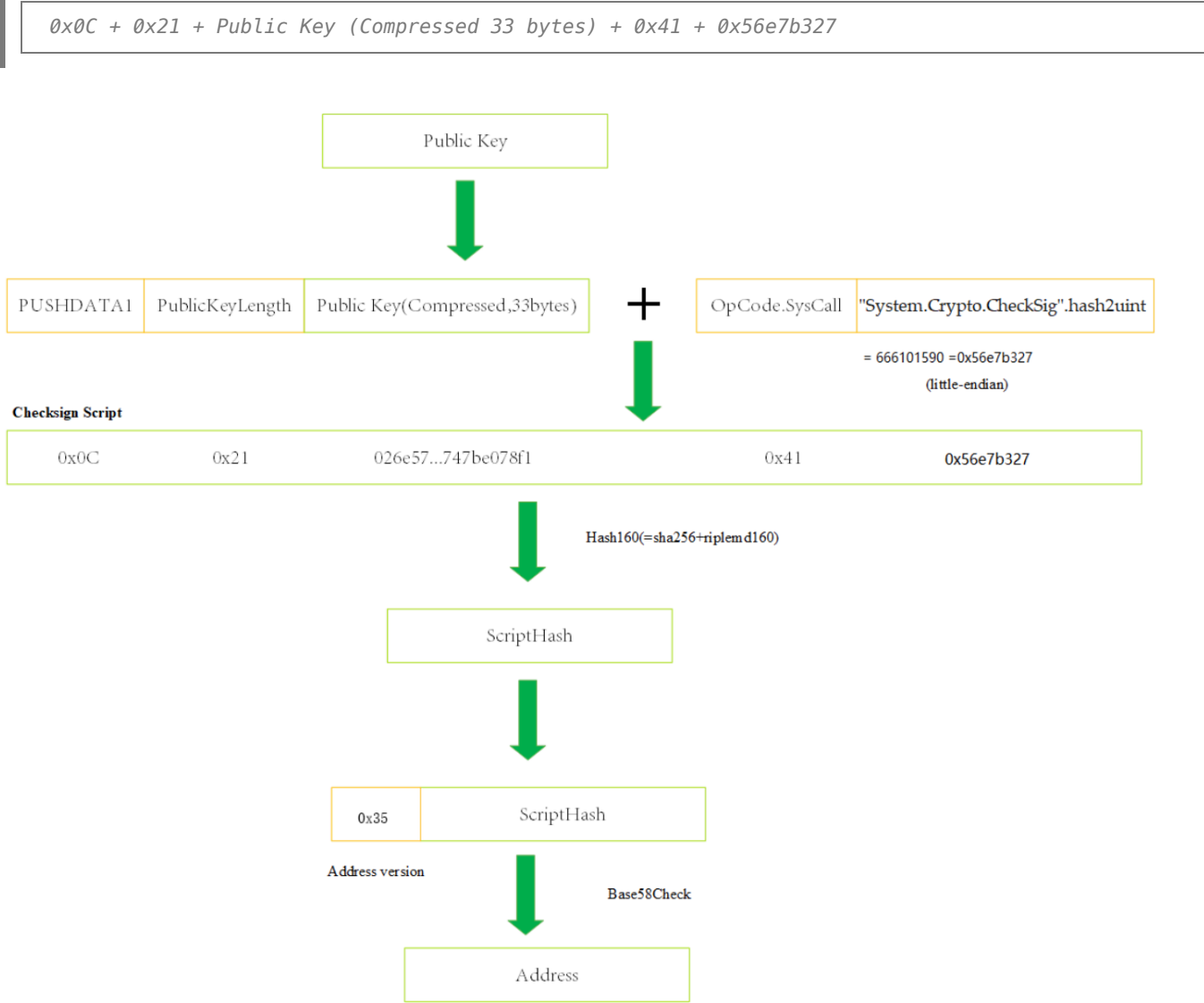
더 이상 `Opcode.CheckSig` 및 `Opcode.CheckMultiSig`를 사용하지 않으며, 대신 상호 운용 서비스 호출인

`SysCall "Neo.Crypto.CheckSig".hash2uint,`

`SysCall "Neo.Crypto.CheckMultisig".hash2uint`를 사용합니다.

일반 주소(Ordinary Address)

1. 공개 키로 CheckSig 스크립트를 구성하며, 형식은 다음과 같습니다:



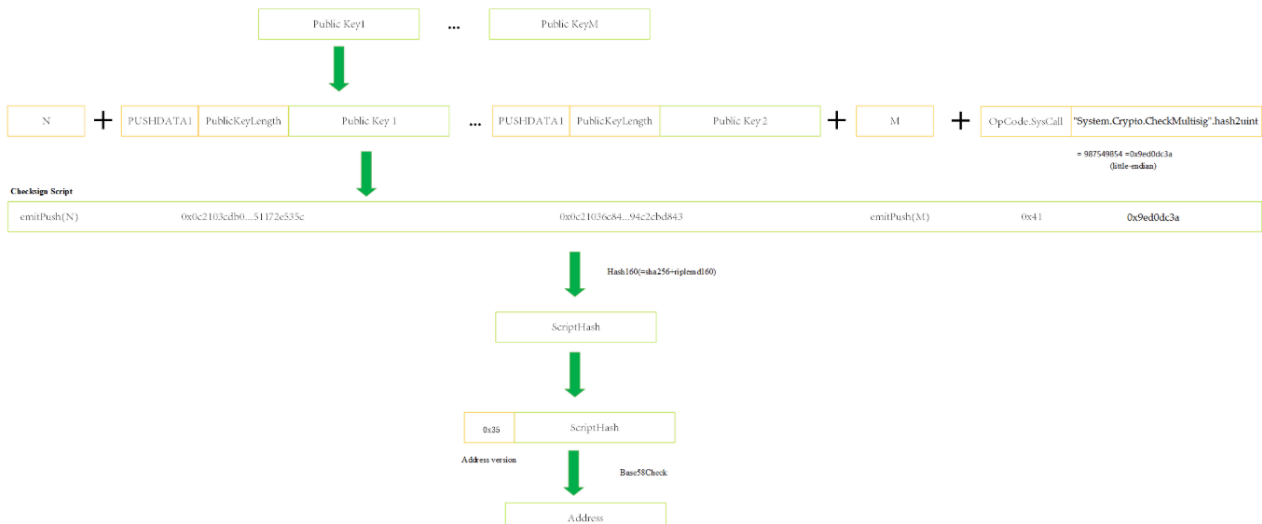
- 2. 컨트랙트의 스크립트 해시(20바이트)를 계산합니다(스크립트에 대해 SHA256 한 번과 RIPEMD160 한 번을 수행).
 - 3. 해시에 버전 프리픽스를 추가합니다(현재 Neo 프로토콜 버전은 53이며 해당 바이트는 0x35).
 - 4. 위의 바이트 데이터를 Base58Check 인코딩합니다.
- 예시:

형식 (Format)	값 (Value)
Private Key	087780053c374394a48d685aacf021804fa9fab19537d16194ee215e825942a0
Public Key (Compressed)	03cdb067d930fd5adaa6c68545016044aadddec64ba39e548250eaea551172e535c
Script	0c2103cdb067d930fd5adaa6c68545016044aadddec64ba39e548250eaea551172e535c4156e7b327
Address	NNLi44dJNXtDNSBkofB48aTVYtb1zZrNEs

다중 서명 주소(Multi-Signature Address)

1. 여러 주소로 N-of-M CheckMultiSig 스크립트를 구성합니다. 스크립트 형식은 다음과 같습니다:

$$\text{emitPush}(N) + 0x0C + 0x21 + \text{Public Key 1 (Compressed 33 bytes)} + \dots + 0x0C + 0x21 + \text{Public Key } m \text{ (Compressed 33 bytes)} + \text{emitPush}(M) + 0x41 + 0x9ed0dc3a$$



2. 컨트랙트의 스크립트 해시(20바이트)를 계산합니다(스크립트에 대해 SHA256 한 번과 RIPEMD160 한 번을 수행).
 3. 해시에 버전 프리픽스를 추가합니다(현재 Neo 프로토콜 버전은 53이며 해당 바이트는 0x35).
 4. 위의 바이트 데이터를 Base58Check 인코딩합니다.
- 예시:

형식 (Format)	값 (Value)
Private Key	087780053c374394a48d685aacfo21804fa9fab19537d16194ee215e825942a0 9a973a470b5fd7a2c12753a1ef55db5a8c8dde42421406a28c2a994e1a1dcc8a
Public Key (Compressed)	03cdb067d930fd5adaa6c6854501604aadddec64ba39e548250eaea551172e535c 036c8431cc78b33177a60b4bcc02baf60d05fee5038e7339d3a688e394c2cbd843
Script	110c21036c8431cc78b33177a60b4bcc02baf60d05fee5038e7339d3a688e394c2cbd8430c2103cdb067d930fd5adaa6c6854501604aadddec64ba39e548250eaea551172e535c12419ed0dc3a
Address	NZ3pqnc1hMN8EHW55ZnCnu8B2wooXJHCyr

참고 (NOTE)

`emitPush(number)` 사용 시 숫자의 구간(interval)에 유의하십시오. 다음은 `number`가 `BigInteger`인 경우의 예로, 다음과 같습니다:

```
data = number.ToByteArray();
```

Number	Emit OpCode	Value
$-1 \leq number \leq 16$	<code>OpCode.PUSHo + (byte)(int)number</code>	<code>0x10 + number</code>
<code>data.Length == 1</code>	<code>OpCode.PUSHINT8 + data</code>	<code>0x00 + data</code>
<code>data.Length == 2</code>	<code>OpCode.PUSHINT16 + data</code>	<code>0x01 + data</code>
<code>data.Length <= 4</code>	<code>OpCode.PUSHINT32 + data</code>	<code>0x02 + PadRight(data, 4)</code>
<code>data.Length <= 8</code>	<code>OpCode.PUSHINT64 + data</code>	<code>0x03 + PadRight(data, 8)</code>
<code>data.Length <= 16</code>	<code>OpCode.PUSHINT128 + data</code>	<code>0x04 + PadRight(data, 16)</code>
<code>data.Length <= 32</code>	<code>OpCode.PUSHINT256 + data</code>	<code>0x05 + PadRight(data, 32)</code>

지갑 주소 스크립트해시(The wallet address scripthash)

Neo 블록체인에서 지갑을 생성할 때, 개인 키, 공개 키, 지갑 주소, 그리고 관련 스크립트해시가 생성됩니다.

아래에서는 표준 지갑 주소와 이에 대응하는 스크립트해시 문자열을 빅엔디안과 리틀엔디안 형식으로 살펴봅니다.

형식 (Format)	문자열 (String)
Address	NUnLWXALK2G6gYa7RadPLRiQYunZHnncxg
Big-endian Scripthash	0xed7cc6f5f2dd842d384f254bc0c2d58fb69a4761
Little-endian Scripthash	61479ab68fd5c2c04b254f382d84ddf2f5c67ced
Base64 Scripthash	YUeato/VwsBLJU84LYTd8vXGfOo=

지갑 주소와 스크립트해시 간, 또는 빅엔디안과 리틀엔디안 바이트 순서 간 변환에는 Data Convertor(<https://github.com>) 도구를 사용하십시오.

지갑 파일(Wallet Files)

db3 파일(db3 files)

db3 지갑은 대량의 계정 정보 저장 및 조회를 용이하게 하기 위해 거래소의 지갑에서 일반적으로 사용됩니다.

db3 지갑 파일은 SQLite를 사용해 데이터를 저장하며, 파일 확장자는 .db3입니다. 하나의 db3 파일에는 네 개의 테이블이 생성됩니다:

- Account

필드 (Field)	유형 (Type)	isRequired	비고 (Note)
PrivateKeyEncrypted	VarBinary(96)	Yes	AES256로 암호화됨
PublicKeyHash	Binary(20)	Yes	기본 키(Primary Key)

- Address

필드 (Field)	유형 (Type)	isRequired	비고 (Note)
ScriptHash	Binary(20)	Yes	기본 키(Primary Key)

- Contract

필드 (Field)	유형 (Type)	isRequired	비고 (Note)
RawData	VarBinary	Yes	
ScriptHash	Binary(20)	Yes	기본 키(Primary Key), 외래 키(Foreign Key), Address 테이블과 연관
PublicKeyHash	Binary(20)	Yes	인덱스(Index), 외래 키(Foreign Key), Account 테이블과 연관

- Key

필드 (Field)	유형 (Type)	isRequired	비고 (Note)
Name	VarChar(20)	Yes	기본 키(Primary Key)
Value	VarBinary	Yes	

Key 테이블에는 주로 AES256 관련 속성들이 저장됩니다:

- **PasswordHash**: 비밀번호의 해시 값으로, SHA256 해시 연산을 통해 생성됩니다.
- **IV**: AES의 초기화 벡터(Initial Vector)로, 무작위로 생성됩니다.
- **MasterKey**: PasswordKey와 IV 파라미터를 사용하여 AES256 알고리즘으로 개인 키를 암호화해 얻은 암호문(ciphertext)입니다.
- **Version**: 지갑의 버전 정보를 나타냅니다.

db3 지갑은 암호화 및 복호화 방식으로 **AES(대칭키 암호화, symmetrical encryption)** 를 사용합니다.

NEP6 파일

NEP6 지갑 파일은 NEP6 표준을 준수하며, 파일 확장자는 `.json`입니다.
JSON 형식은 다음과 같습니다:

```
{
  "name": null,
  "version": "3.0",
  "script": {
    "n": 16384,
    "r": 8,
    "p": 8
  },
  "accounts": [
    {
      "address": "Nf8iN8CABre87oDaDrHSnMAyVoU9jYa2FR",
      "label": null,
      "isdefault": false,
      "lock": false,
      "key": "6PYM9DxRY8RMhKHp512xExRVLeB9DSkW2cCKCe65oXgL4tD2kaJX2yb9vD",
      "contract": {
        "script": "DCEDYgBftumtbwC64LbngHbZPDVrSMrEuHXNP0tJzPl0dL5BdHR2qg==",
        "parameters": [
          {
            "name": "signature",
            "type": "Signature"
          }
        ],
        "deployed": false
      },
      "extra": null
    }
  ],
  "extra": null
}
```

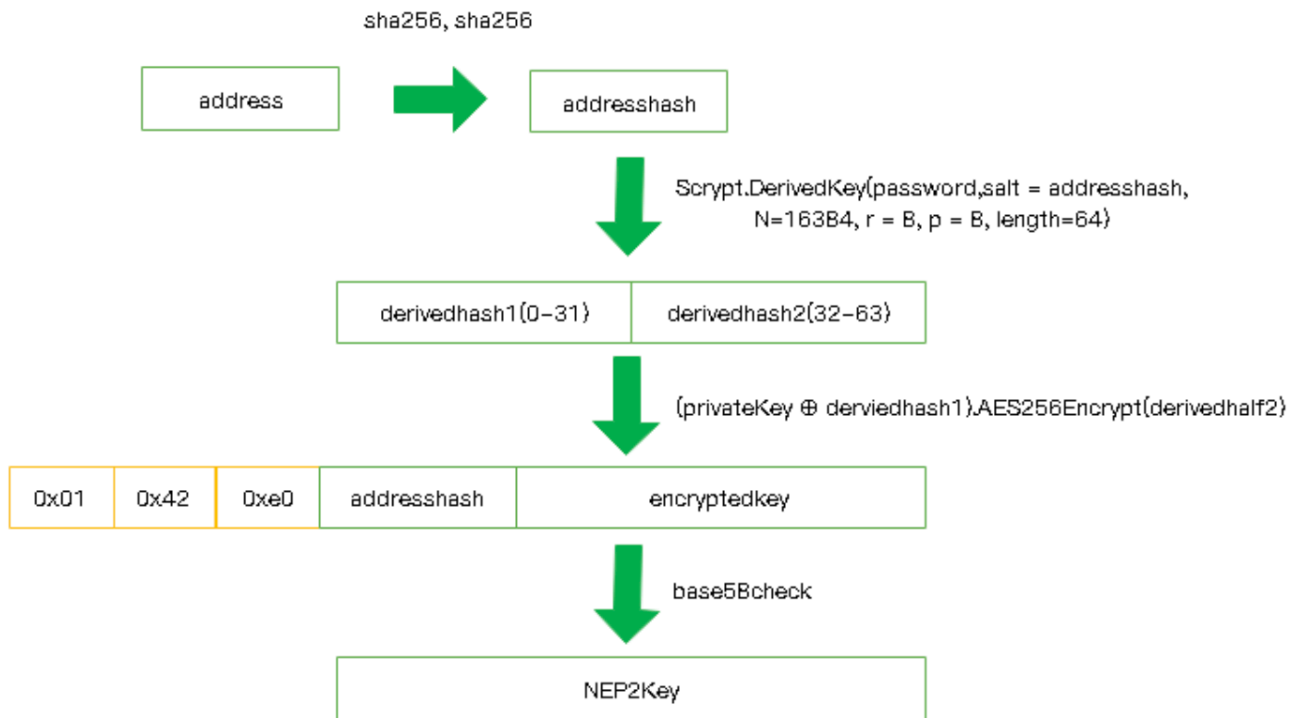
이 예시에서 비밀번호는 1입니다.

NEP6 지갑 JSON 필드

필드 (Field)	설명 (Description)
name	사용자가 지갑 파일에 부착하는 라벨
version	현재 값은 3.0
script (n/r/p)	(n/r/p)는 지갑의 개인 키 암호화·복호화에 사용되는 script 알고리즘의 매개변수
accounts	지갑의 각 계정 세부정보를 설명하는 Account 객체 배열
account.address	계정 주소
account.label	계정 라벨, 기본값은 null
account.isDefault	해당 계정이 지갑의 기본 계정인지 여부
account.lock	계정이 잠겨 있는지 여부
account.key	개인 키의 nep2key 내보내기
account.contract	스크립트의 컨트랙트
account.contract.script	주소 스크립트
account.contract.parameters	주소 스크립트 컨트랙트의 매개변수 목록
account.contract.parameter.name	주소 스크립트 컨트랙트의 매개변수 이름
account.contract.parameter.type	주소 스크립트 컨트랙트의 매개변수 타입
account.contract.deployed	배포 여부
account.extra	계정의 추가 속성, 기본값은 null
extra	지갑의 추가 속성, 기본값은 null

NEP6 지갑은 script 알고리즘을 지갑 암호화 및 복호화의 핵심 방식으로 사용합니다.

암호화 단계 (Encryption steps)



1. 주소는 공개 키로부터 도출되며, 주소 해시는 SHA256(SHA256(Address))의 첫 4바이트입니다.
2. scrypt 알고리즘을 사용하여 derivedkey를 계산하고, 64바이트 데이터를 derivedhalf1과 derivedhalf2로 나눕니다.
scrypt는 다음 매개변수를 사용합니다:
 - ciphertext: 입력된 비밀번호 (UTF-8 형식)
 - salt: 주소 해시
 - n: 16384
 - r: 8
 - p: 8
 - length: 64
3. 개인 키와 derivedhalf1을 XOR 연산한 뒤, derivedhalf2를 사용하여 AES256으로 암호화하여 encryptedkey를 얻습니다.
4. 다음 형식에 따라 데이터를 연결하고, Base58Check 인코딩을 통해 NEP2Key를 생성합니다:
 $0x01 + 0x42 + 0xe0 + \text{address hash} + \text{encryptedkey}$

복호화 단계 (Decryption steps)

1. Base58Check 디코딩을 사용하여 NEP2Key를 디코딩합니다.
 2. 디코딩된 데이터의 길이가 39바이트인지, 그리고 처음 세 바이트(data[0-2])가 0x01, 0x42, 0xe0인지 확인합니다.
 3. data[3-6]을 addresshash로 사용합니다.
 4. 비밀번호와 addresshash를 scrypt 알고리즘에 입력하고, 결과 길이를 64로 지정하여 derivedkey를 얻습니다.
 5. derivedkey[0-31]을 Derivedhalf1, derivedkey[32-63]을 Derivedhalf2로 사용합니다.
 6. data[7-38]을 Encryptedkey(32바이트)로 설정하고, derivedhalf2를 초기화 벡터 사용하여 AES256 복호화를 수행합니다.
 7. 복호화된 데이터와 derivedhalf1을 XOR 연산하여 개인 키를 얻습니다.
 8. 개인 키로부터 ECC 알고리즘을 통해 공개 키를 생성하고, 해당 공개 키로 주소를 도출합니다.
- SHA256(SHA256(Address))의 첫 4바이트가 addresshash와 동일하다면, 올바른 개인 키를 획득한 것입니다.

NEP2 및 NEP6 제안서에 대한 자세한 내용은 Neo 공식 문서를 참고하세요.

- NEP2 제안서: <https://github.com/neo-project/proposals/blob/master/nep-2.mediawiki>

- NEP6 제안서: <https://github.com/neo-project/proposals/blob/master/nep-6.mediawiki>

서명(Signature)

Neo는 지갑 컴포넌트를 통해 트랜잭션에 서명하기 위해 ECDSA 알고리즘을 사용합니다.
ECC 곡선으로는 nistP256 또는 Secp256r1을 사용하며, 해시 알고리즘으로는 SHA256을 사용합니다.

C# 코드

```
public static byte[] Sign(byte[] message, byte[] prikey, byte[] pubkey)
{
    using (var ecdsa = ECDsa.Create(new ECDsaParameters
    {
        Curve = ECCurve.NamedCurves.nistP256,
        D = prikey,
        Q = new ECPoint
        {
            X = pubkey[..32],
            Y = pubkey[32..]
        }
    }))
    {
        return ecdsa.SignData(message, HashAlgorithmName.SHA256);
    }
}
```

예시

형식 (Format)	값 (Value)
data	hello world
PrivateKey	f72b8fab85fdcc1bdd20b107e5da1ab4713487bc88fc53b5b134f5eddea1a19
PublicKey	031f64da8a38e6c1e5423a72ddd6d4fc4a777abe537e5cb5aa0425685cda8e063b
signature	b1855cec16b6ebb372895d44c7be3832b81334394d80bec7c4f00a9c1d9c3237541834638d11ad9c62792ed548c9602c1d8cdoca92fdd5e68ceea40e7bcfbbeb2

지갑 기능(Wallet Function)

함수명 (Function Name)	설명 (Description)
Import wallet file	지정된 지갑 파일에서 계정 정보를 가져옵니다.
Export wallet file	계정 정보(개인 키, 비밀번호, 주소 등)를 db3 지갑 파일 또는 nep6 json 파일 등 지정된 지갑 파일에 저장합니다.
Unlock wallet	사용자의 비밀번호를 검증하여 보안 유출을 방지합니다.
Create private key	안전한 난수 생성기를 사용하여 개인 키를 생성합니다.
Import private key	wif 형식 또는 디지털 인증서를 사용하여 새 개인 키를 지갑에 추가합니다.
Export private key	계정의 개인 키를 내보냅니다.
Generate public key	ECC 알고리즘을 통해 개인 키에서 공개 키를 생성합니다.
Generate address	개인 키를 기반으로 주소를 생성합니다.
Import address	새 주소를 지갑에 추가합니다.
Export address	계정의 주소를 내보냅니다.
Import offline data	chain.acc 파일의 블록 데이터를 불러와 동기화 시간을 단축합니다.
Export offline data	블록 데이터를 chain.acc 파일로 내보냅니다.
Synchronize block data	블록 데이터를 동기화합니다.
Transfer	다른 주소로 자산을 전송합니다.
Sign	트랜잭션 등의 데이터를 서명합니다.
Claim Gas	계정이 보유한 NEO로부터 새로 생성된 GAS를 청구합니다.
Get balance	현재 지갑의 잔액을 표시합니다.
Get transaction	현재 지갑의 트랜잭션 내역을 표시합니다.
Construct multi-signature contract	다중 서명 컨트랙트를 생성합니다.
Extend	확장 기능을 수행합니다.
Deploy smart contract	스마트 컨트랙트를 배포합니다.
Test smart contract	스마트 컨트랙트를 테스트합니다.

지갑 소프트웨어(Wallet Software)

풀노드 지갑(Full-node Wallet)

풀노드 지갑은 블록체인 데이터를 완전히 백업하는 형태의 지갑으로, 모든 온체인 데이터를 저장하고 P2P 네트워크에 참여합니다. 따라서 상당한 저장 공간이 필요합니다.

Neo-CLI와 Neo-GUI는 모두 풀노드 지갑입니다.
자세한 내용은 [Neo node](#)를 참고하세요.

SPV 지갑(SPV Wallet)

SPV(Simplified Payment Verification, 단순 결제 검증) 지갑은 풀노드 지갑과 다릅니다.

모든 블록 데이터를 저장하지 않고 **블록 헤더 데이터만 저장**하며,

Bloom 필터와 **Merkle 트리 알고리즘**을 사용해 데이터를 검증합니다.

이 방식은 저장 공간을 효율적으로 절약할 수 있어 **모바일 앱**이나 **라이트 클라이언트**에서 주로 사용됩니다.

SPV 지갑을 개발하려면 NEO 네트워크 프로토콜 인터페이스를 참조하세요.

사용 방법(Usage):

1. SPV 지갑이 Bloom 필터를 풀노드에 전송하면, 풀노드는 해당 필터를 로드합니다.
2. SPV 지갑이 Bloom 필터의 매개변수를 풀노드에 전송하면, 풀노드는 해당 매개변수를 로드합니다. (선택 사항)
3. SPV 지갑이 풀노드로부터 트랜잭션을 조회하면, 풀노드는 Bloom 필터와 Merkle 트리 경로를 이용해 필터링된 트랜잭션 데이터를 반환합니다.
4. SPV 지갑은 Merkle 트리 경로를 사용하여 트랜잭션 데이터를 검증합니다.
5. SPV 지갑은 `clear the bloom filter` 명령을 풀노드에 전송하고, 풀노드는 이를 초기화합니다.

블록(Block)

블록체인은 데이터 구조(data structure)입니다.

블록은 **블록 헤더(block header)**와 **블록 본문(block body)**으로 구성됩니다.

각 블록은 이전 블록의 암호학적 해시값(PrevHash)을 포함하고 있으며,

이를 통해 **연속적인 체인 구조(chain structure)**가 형성됩니다.

블록 헤더(Block Header)

블록의 데이터 구조는 다음과 같습니다:

크기 (Size)	필드(Field)	이름(Name)	타입 (Type)	설명(Description)
4	Version	블록 버전	uint	블록 버전, 현재는 0
32	PrevHash	이전 블록 해시	UInt256	이전 블록의 해시값
32	MerkleRoot	머클 트리 루트	UInt256	블록 내 트랜잭션의 머클 트리 루트 해시
8	Timestamp	블록 타임스탬프	ulong	블록이 생성된 시간
8	Nonce	난수	ulong	블록의 난수(Random number)
4	Index	블록 인덱스	uint	블록 높이(Height), 제네시스 블록의 인덱스는 0
1	PrimaryIndex	스피커 인덱스	byte	현재 라운드에서 제안자(Validator)의 인덱스
20	NextConsensus	다음 라운드 합의 검증자 주소	UInt160	다음 라운드 검증자의 서명 중 3분의 2 이상으로부터 생성된 스크립트 해시
?	Witness	검증자 스크립트	Witness	실행 가능한 검증 스크립트

블록 헤더는 블록의 기본 정보를 포함하며, 블록 검증의 근거를 제공합니다.

블록은 hash와 index를 통해 고유하게 식별할 수 있습니다.

해시 값은 블록 헤더의 처음 일곱 개 속성을 연결한 후, SHA256 연산을 두 번 수행하여 얻습니다.

일반적으로 Neo 블록체인은 단일 체인 구조를 유지하며, 각 블록은 전체 합의 노드의 **3분의 2 이상이 확인(confirmed)** 한 후 체인에 추가됩니다.

따라서 블록 높이(Block Height)는 고유하며, 항상 **이전 블록 높이 + 1**의 관계를 가집니다.

제네시스 블록(Genesis Block)의 높이는 0입니다.

Timestamp는 블록의 타임스탬프를 나타내며, 이전 블록보다 커야 합니다.

두 블록 간의 간격은 약 15초이며, 이 값은 구성 파일 config.json의 MillisecondsPerBlock 변수로 설정됩니다.

NextConsensus는 **다중 서명 컨트랙트(Multi-signature Contract)**의 해시를 의미하며,

3분의 2 이상의 합의 노드 서명을 파라미터로 필요로 합니다.

아래 예시 스크립트는 이를 보여줍니다.

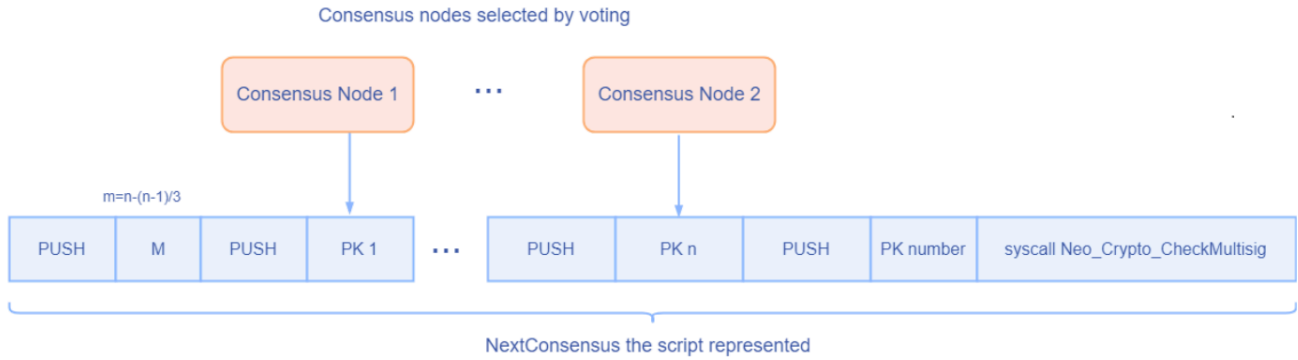
각 블록의 NextConsensus 필드는 다음 합의 라운드에 참여할 노드들을 고정(lock)합니다.

이전 합의 라운드에서 스피커(Speaker)는 당시의 투표 결과를 기반으로 **다음 라운드의 합의 노드 세트**를 계산하고, 그 다중 서명 컨트랙트를 생성하여 해당 해시 값을 블록의 NextConsensus 필드에 할당합니다.

Witness는 블록의 검증 스크립트로,

InvocationScript와 VerificationScript로 구성됩니다.

InvocationScript는 VerificationScript 실행 시 필요한 파라미터를 제공합니다.



블록(Block)

블록의 데이터 구조는 다음과 같습니다:

크기(Size)	필드(Field)	이름(Name)	타입(Type)	설명(Description)
?	Header	블록 헤더	Header	블록의 헤더
?*?	Transactions	트랜잭션 목록	Transaction[]	블록의 페이로드(Payload)

블록 본문은 트랜잭션 리스트로 구성되어 있으며, 트랜잭션의 개수(리스트 길이)로 시작하고 그 뒤에 트랜잭션들이 이어집니다. 하나의 합의 라운드(Consensus Round) 동안, 스피커(Speaker)는 자신의 메모리 풀에서 검증된 트랜잭션들을 선택하고, 이들의 트랜잭션 해시를 합의 메시지(PrePrepareRequest)에 포함시켜 블록체인 네트워크에 브로드캐스트합니다. 합의 과정에 대한 자세한 내용은 [Consensus Mechanism](#)을 참고하세요.

현재 블록당 최대 512개의 트랜잭션을 포함할 수 있습니다.

참고(NOTE)

블록이 영구적으로 저장될 때, 블록의 트랜잭션 해시 목록이 함께 저장되며 트랜잭션 데이터는 별도로 보관되어 조회 효율성을 높입니다.

트랜잭션(Transaction)

트랜잭션은 Neo 네트워크 전체의 기본 동작 모델입니다. 지갑(Wallet), 스마트 컨트랙트(Smart Contract), 계정(Account) 등이 Neo 네트워크와 상호작용할 때 트랜잭션을 통해 이루어집니다.

Neo의 P2P 네트워크에서는 정보가 전송될 때 InvPayload 형태로 패키징됩니다 (Inv는 Inventory의 약자). 각 페이로드(Payload)는 고유한 데이터 구조를 가지며, InventoryType.Tx는 트랜잭션 데이터가 InvPayload에 포함되어 전송됨을 의미합니다.

구조(Structure)

Neo 블록체인에서 트랜잭션의 데이터 구조는 다음과 같습니다:

크기(Size)	필드(Field)	설명(Description)
version	byte	트랜잭션 버전, 현재는 0
nonce	uint	난수(Random number)
sysfee	long	네트워크 자원 사용을 위한 시스템 수수료(System fee)
netfee	long	트랜잭션을 블록에 패키징한 검증자에게 지급되는 네트워크 수수료(Network fee)
validUntilBlock	uint	트랜잭션의 유효 기간(Validity period)
signers	Signer[]	송신자 및 서명 범위 정보
attributes	TransactionAttribute[]	트랜잭션 속성(Attribute)
script	byte[]	NeoVM에서 실행되는 스크립트
witnesses	Witness[]	트랜잭션 검증에 사용되는 스크립트 목록

version

version 필드는 트랜잭션 구조가 향후 업데이트되더라도 하위 호환성을 유지할 수 있도록 합니다.
현재 버전은 0입니다.

서명자(Signers)

첫 번째 필드는 트랜잭션 송신자 계정의 **스크립트 해시(Script Hash)** 입니다.
Neo N3에서는 UTXO(Unspent Transaction Output) 모델이 더 이상 사용되지 않으며, 기존의 기본 자산인 NEO와 GAS는 **NEP-17 자산 표준**으로 전환되었습니다.
따라서 트랜잭션 구조에는 더 이상 입력(input)과 출력(output) 필드가 기록되지 않습니다.
그 대신, sender 필드를 통해 트랜잭션의 송신자를 추적합니다.

이후의 필드들은 **서명의 유효 범위(Signature Scope)** 를 정의하는 데 사용됩니다.
트랜잭션 검증 시 checkwitness가 호출되면,
트랜잭션 송신자를 제외한 공동 서명자(cosigner)는 자신의 서명 범위를 명확히 지정해야 합니다.
자세한 내용은 [Signature Scope](#)를 참고하세요.

필드(Field)	설명(Description)	타입(Type)
Account	계정의 스크립트 해시	UInt160
Scopes	서명의 유효 범위	WitnessScope
AllowedContracts	서명이 허용된 컨트랙트 스크립트의 배열	UInt160[]
AllowedGroups	서명이 허용된 컨트랙트 그룹의 공개 키 배열	ECPint[]

시스템 수수료(System Fee)

시스템 수수료는 트랜잭션의 **스크립트(script)**에 따라 결정됩니다.
즉, 스크립트의 크기, 명령어의 개수 및 NeoVM 명령어의 종류에 의존합니다.
Neo N3에서는 과거의 “10 GAS 무료 시스템 수수료” 정책이 폐지되었습니다.
시스템 수수료의 계산 공식은 다음과 같습니다:

$$\text{SystemFee} = \sum_{i \in \text{OpcodeSet}} \text{OpcodePrice}_i \cdot n_i$$

여기서,

- $\text{\$OpcodeSet\$}$ 은 사용된 **명령어 집합(opcode set)**,
- $\text{\$OpcodePrice}_i\text{\$}$ 는 각 명령어 $\text{\$i\$}$ 의 실행 비용,
- $\text{\$n}_i\text{\$}$ 는 해당 명령어 $\text{\$i\$}$ 가 컨트랙트 스크립트 내에서 실행된 횟수입니다.

각 명령어별 수수료에 대한 자세한 정보는

Fees for Instructions(<https://github.com/neo-project/neo/blob/master/docs/fees/instructions.md>)를 참고하세요.

네트워크 수수료(Network Fee)

네트워크 수수료는 사용자가 Neo 블록체인에 트랜잭션을 제출할 때 발생하며,
이는 블록을 생성하는 **합의 노드(Consensus Nodes)**에 대한 보상으로 지급됩니다.
모든 트랜잭션에는 기본 수수료(Base Fee)가 존재하며,
사용자가 지불한 수수료가 기본 수수료보다 크거나 같을 때만 트랜잭션이 실행됩니다.
그보다 적으면 트랜잭션은 **무효(Invalid)**로 처리됩니다.

네트워크 수수료의 계산 공식은 다음과 같습니다:

$$\text{NetworkFee} = \text{VerificationCost} + \text{tx.Length} \times \text{FeePerByte}$$

여기서, VerificationCost 는 NeoVM이 트랜잭션 서명을 검증하기 위해 실행하는 명령어의 비용, tx.Length 는 트랜잭션 데이터의 바이트 길이, FeePerByte 는 바이트당 트랜잭션 수수료이며, 현재 값은 0.00001 GAS 입니다.

속성(Attributes)

특정 유형의 트랜잭션에는 추가적인 속성(attributes)을 부여할 수 있습니다.
각 속성에 대해 **사용 목적(usage type)**, **내부 데이터 크기**, **외부 데이터 크기**를 정의해야 합니다.

하나의 트랜잭션에는 최대 **16개의 속성**을 추가할 수 있습니다.

스크립트(Script)

NeoVM 상에서 실행되는 스크립트로,
트랜잭션의 **효과(effects)** 를 결정하는 핵심 코드입니다.

검증자(Witnesses)

Witnesses는 트랜잭션의 **유효성(Validity)** 과 **무결성(Integrity)** 을 검증합니다.
이는 다음 두 가지 속성으로 구성됩니다.

필드(Field)	설명(Description)
InvocationScript	스크립트를 실행하고 검증 스크립트(Verification Script)에 파라미터를 전달합니다.
VerificationScript	검증 스크립트(Verification Script)

각 트랜잭션에는 여러 개의 witness를 추가할 수 있으며,
다중 서명(Multi-Signature) witness도 사용할 수 있습니다.

호출 스크립트(Invocation Script)

서명을 추가하기 위해 호출 스크립트를 구성합니다:

참고 (NOTE)

`WriteVarInt(value)` 는 값(value)의 크기에 따라 **가변 길이(Variable-length)** 형식으로 데이터를 저장하며,
값의 범위에 따라 저장 크기가 결정됩니다.

값의 범위 (Value Range)	저장 형식 (Storage Type)
value < 0xFD	byte(value)
value <= 0xFFFF	0xFD + ushort(value)
value <= 0xFFFFFFFF	0xFE + uint(value)
value > 0xFFFFFFFF	0xFF + value

트랜잭션 서명(Transaction Signature)

트랜잭션 서명은 트랜잭션 자체의 데이터를 **ECDSA** 방식으로 서명하는 것을 의미합니다.
이때 서명 데이터(witnesses 부분)는 제외됩니다.
서명 완료 후, 해당 서명 데이터가 트랜잭션 본문 내 witnesses 필드에 추가됩니다.

아래는 JSON 형식의 트랜잭션 예시로,
script와 witnesses 필드는 기존의 Hexstring 인코딩 대신 Base64 인코딩을 사용합니다.

```
{
  "hash": "0xd2b24b57ea05821766877241a51e17eae06ed66a6c72adb5727f8ba701d995be",
  "size": 265,
  "version": 0,
  "nonce": 739807055,
  "sender": "NMDf1XCbioM7ZrPZAdQKQt8nnx3fWr1wdr",
  "sys_fee": "9007810",
  "net_fee": "1264390",
  "valid_until_block": 2102402,
```

```

"signers": [{
  "account": "0xdf93ea5a0283c01e8cdfae891ff700faad70500e",
  "scopes": "FeeOnly"
},
{
  "account": "0xdf93ea5a0283c01e8cdfae891ff700faad70500e",
  "scopes": "CalledByEntry"
}],
"attributes": [],
"script":
"EQwUDlBwrfoA9x+Jrt+MHsCDAlrqk98MFA5QcK36APcfia7fjB7AgwJa6pPfE8AMCHRYW5zZmVyDBSJdyDYzXb08Aq/o
3w03YicII/em0FifVt50A==",
"witnesses": [{
  "invocation":
"DEDy/g4Lt+FTMBHHF84TSVXG9aSN0D0jj0aPaJq8u0c6eMzqr8rARqpB4gWGXNfzLyh9qKvE++6f6XoZeaEoUPeH",
  "verification": "DCECCJr46zTvjdE0jA5v5jrry4Wi8Wm7Agjf6zGH/7/1EVELQQqQatQ="
}]
}

```

서명 범위(Signature Scope)

Neo Legacy에서는 트랜잭션 서명이 전역(Global)으로 유효했습니다.

하지만 Neo N3에서는 사용자가 서명의 **적용 범위(Scope)**를 더 세밀하게 제어할 수 있도록 WitnessScope가 추가되었고, 이에 따라 트랜잭션 구조의 signers 필드가 변경되었습니다.

이제 서명은 지정된 컨트랙트에 대해서만 유효하며, 무단 컨트랙트가 사용자의 서명을 도용하는 것을 방지할 수 있습니다.

서명 범위 유형(Scopes)

트랜잭션을 구성할 때 signers 필드 내 scopes 값을 지정해야 하며, 이는 서명의 유효 범위를 정의합니다. 다음과 같은 유형이 있습니다:

코드 (Field)	이름(Name)	설명(Description)
0x00	None	서명이 트랜잭션에만 사용되며, 컨트랙트 내에서는 비활성화됩니다.
0x01	CalledByEntry	Entry에서 호출된 컨트랙트 스크립트에 대해서만 서명이 유효합니다.
0x10	CustomContracts	지정된 컨트랙트 스크립트에 대해서만 서명이 유효합니다. CalledByEntry와 함께 사용할 수 있습니다.
0x20	CustomGroups	그룹 내 컨트랙트에 대해서만 서명이 유효합니다. CalledByEntry와 함께 사용할 수 있습니다.
0x80	Global	서명이 전역적으로 유효합니다. 단, 매우 위험하며 신뢰도가 높은 컨트랙트에만 사용해야 합니다. (이 서명은 주소 내 모든 자산을 이동시킬 수 있습니다.)
0x40	WitnessRules	규칙 및 범위를 지정해야 합니다. 자세한 내용은 WitnessRule 를 참고하세요.

이해를 돕기 위해 다음과 같은 컨트랙트 호출 체인을 가정해 보겠습니다:

[Entry] → [Contract A] → [Contract B] → [Contract C] → ... → [Target]

Target 컨트랙트가 CheckWitness를 호출하여 서명을 검증할 때, scopes 값에 따라 검증 결과가 달라집니다.

- **None** – Target 컨트랙트의 위치와 상관없이 검증에 실패합니다.
- **Global** – Target 컨트랙트의 위치와 관계없이 항상 검증에 성공합니다.
- **CalledByEntry** – Target이 Entry 또는 **Contract A**일 때만 검증이 통과됩니다.
- **CustomContracts** – Target이 사용자가 정의한 CustomContracts 목록에 포함된 컨트랙트일 때만 검증이 통과됩니다.
- **CustomGroups** – Target 컨트랙트가 CustomGroups 내의 공개 키(ECPoint) 중 하나로 인증된 경우에만 검증이 통과됩니다.

WitnessRule

Action(Allow | Deny) 과 Condition으로 구성됩니다.

실행 로직은 다음과 같습니다:

- 조건(Condition)을 평가하여 충족되면 Action을 반환합니다.
- Allow는 검증 성공을 의미하며,
- Deny는 검증 실패를 의미합니다.

WitnessCondition (증인 조건)

- Boolean: true | false
"expression" = <bool>

```
//Equals to WitnessScope.Global
{
  "account": "NdUL5oDPD159KeFpD5A9zw5xNF1xLX6nLT",
  "scopes": "WitnessRules",
  "rules": [{
    "action": "Allow",
    "condition": {
      "type": "Boolean",
      "expression": true
    }
  }]
}
```

- Not: 논리 부정(Logical NOT). 다른 조건의 결과를 반전시킵니다.
"expression" = <Condition>

```
// The signature is allowed only when the contract is not
0xef4073a0f2b305a38ec4050e4d3d28bc40ea63f5
{
  "account": "NdUL5oDPD159KeFpD5A9zw5xNF1xLX6nLT",
  "scopes": "WitnessRules",
  "rules": [{
    "action": "Allow",
    "condition": {
      "type": "Not",
      "expression": {
        "type": "ScriptHash",
        "hash": "0xef4073a0f2b305a38ec4050e4d3d28bc40ea63f5"
      }
    }
  }]
}
```

```
}  
}
```

- And: 논리적 AND (논리곱).
모든 조건이 충족될 때만 참(true)을 반환합니다.
"expressions" = <Condition[]>

```
// The signature is allowed only when the contract is  
0xef4073a0f2b305a38ec4050e4d3d28bc40ea63f5 and is invoked at entry  
{  
  "account": "NdUL5oDPD159KeFpD5A9zw5xNF1xLX6nLT",  
  "scopes": "WitnessRules",  
  "rules": [{  
    "action": "Allow",  
    "condition": {  
      "type": "And",  
      "expressions": [{  
        "type": "ScriptHash",  
        "hash": "0xef4073a0f2b305a38ec4050e4d3d28bc40ea63f5"  
      }, {  
        "type": "CalledByEntry"  
      }  
    ]  
  }  
}]  
}
```

- Or: 논리적 OR (논리합).
하나 이상의 조건이 충족되면 참(true)을 반환합니다.
"expressions" = <Condition[]>

```
// The signature is allowed only when the contract is  
0xef4073a0f2b305a38ec4050e4d3d28bc40ea63f5 or is invoked at entry  
{  
  "account": "NdUL5oDPD159KeFpD5A9zw5xNF1xLX6nLT",  
  "scopes": "WitnessRules",  
  "rules": [{  
    "action": "Allow",  
    "condition": {  
      "type": "Or",  
      "expressions": [{  
        "type": "ScriptHash",  
        "hash": "0xef4073a0f2b305a38ec4050e4d3d28bc40ea63f5"  
      }, {  
        "type": "CalledByEntry"  
      }  
    ]  
  }  
}]  
}
```

- ScriptHash: 현재 컨트랙트의 해시가 지정된 값과 일치하는지 검증합니다.
이는 CustomContracts와 동일한 역할을 합니다.

"hash" = <UInt160>

```
// Only the contract 0xef4073a0f2b305a38ec4050e4d3d28bc40ea63f5 is allowed to use the signature
{
  "account": "NdUL5oDPD159KeFpD5A9zw5xNF1xLX6nLT",
  "scopes": "WitnessRules",
  "rules": [{
    "action": "Allow",
    "condition": {
      "type": "ScriptHash",
      "hash": "0xef4073a0f2b305a38ec4050e4d3d28bc40ea63f5"
    }
  }]
}
```

- **Group:** 현재 컨트랙트의 공개 키(public key)가 지정된 값과 일치하는지 검증합니다.
이는 CustomGroups와 동일한 역할을 합니다.
"group" = <ECPoint>

```
// Only the contract authorized by
021821807f923a3da004fb73871509d7635bcc05f41edef2a3ca5c941d8bbc1231 is allowed to use the
signature
{
  "account": "NdUL5oDPD159KeFpD5A9zw5xNF1xLX6nLT",
  "scopes": "WitnessRules",
  "rules": [{
    "action": "Allow",
    "condition": {
      "type": "Group",
      "group": "021821807f923a3da004fb73871509d7635bcc05f41edef2a3ca5c941d8bbc1231"
    }
  }]
}
```

- **CalledByEntry:** 현재 컨트랙트가 Entry 컨트랙트에 의해 호출되었는지를 검증합니다.
이는 CallByEntry와 동일한 의미를 가집니다.

```
// Equals to WitnessScope.CallByEntry
{
  "account": "NdUL5oDPD159KeFpD5A9zw5xNF1xLX6nLT",
  "scopes": "WitnessRules",
  "rules": [{
    "action": "Allow",
    "condition": {
      "type": "CalledByEntry"
    }
  }]
}
```

- **CalledByContract:** 이전 단계의 컨트랙트 해시가 지정된 값과 일치하는지를 검증합니다.
"hash" = <UInt160>

```
// The signature is allowed only when the previous level contract is
0xef4073a0f2b305a38ec4050e4d3d28bc40ea63f5
{
  "account": "NdUL5oDPD159KeFpD5A9zw5xNF1xLX6nLT",
  "scopes": "WitnessRules",
  "rules": [{
    "action": "Allow",
    "condition": {
      "type": "CalledByContract",
      "hash": "0xef4073a0f2b305a38ec4050e4d3d28bc40ea63f5"
    }
  }
]
}
```

- CalledByGroup: 이전 단계의 컨트랙트 공개 키가 지정된 값과 일치하는지를 검증합니다.
"group" = <UInt160>

```
// The signature is allowed only when the previous level contract is authorized by the public
key 021821807f923a3da004fb73871509d7635bcc05f41edef2a3ca5c941d8bbc1231
{
  "account": "NdUL5oDPD159KeFpD5A9zw5xNF1xLX6nLT",
  "scopes": "WitnessRules",
  "rules": [{
    "action": "Allow",
    "condition": {
      "type": "CalledByGroup",
      "group": "021821807f923a3da004fb73871509d7635bcc05f41edef2a3ca5c941d8bbc1231"
    }
  }
]
}
```

예시(Example)

현재 이 필드는 SDK를 사용하여 트랜잭션을 구성할 때만 정의할 수 있습니다.
아래 JSON 형식의 코드 예시는 이해를 돕기 위한 샘플입니다.

```
{
  "account": "NdUL5oDPD159KeFpD5A9zw5xNF1xLX6nLT",
  "scopes": "WitnessRules",
  "rules": [{
    "action": "Allow",
    "condition": {
      "type": "Not",
      "expression": {
        "type": "And",
        "expressions": [{
          "type": "ScriptHash",
          "hash": "0xef4073a0f2b305a38ec4050e4d3d28bc40ea63f5"
        }, {
          "type": "CalledByEntry"
        }
      ]
    }
  }
]}
```

```
}  
}  
}
```

합의 메커니즘(Consensus Mechanism)

개요(Overview)

블록체인은 **탈중앙화된 분산 원장 시스템(Decentralized Distributed Ledger System)**입니다.

이는 디지털 자산의 등록 및 발행, 재산권 증서, 신용 점수 등의 관리에 활용될 수 있으며,

P2P(Peer-to-Peer) 방식으로 송금, 결제 및 거래를 가능하게 합니다.

블록체인 기술은 원래 **사토시 나카모토(Satoshi Nakamoto)**가 암호학 메일링 리스트에서 비트코인(Bitcoin)을 제안하면서 등장했습니다.

이후 전자화폐 시스템(e-cash systems), 주식 거래 시스템, 스마트 컨트랙트(Smart Contract) 시스템 등 다양한 응용 사례가 등장했습니다.

블록체인 시스템은 기존의 중앙집중식 원장 시스템에 비해

공개성(Openness), **불변성(Immutability)**, **이중 지불 방지(Anti-Multiple-Spend)** 등의 장점을 가지고 있으며, 신뢰할 수 있는 제3자(Trusted Third Party)에 의존하지 않습니다.

그러나 다른 모든 분산 시스템과 마찬가지로, 블록체인 시스템은

네트워크 지연(Network Latency), **전송 오류(Transmission Error)**, **소프트웨어 버그**,

보안 취약점(Security Loopholes), **악의적 행위자(Black-hat Hacker)** 등의 위협에 노출되어 있습니다.

또한 시스템의 탈중앙화된 특성상, 어떤 참여자도 완전히 신뢰할 수 없으며

악의적인 노드(Malicious Nodes)가 등장할 수 있고, 이해관계 충돌로 인해 데이터 불일치가 발생할 수 있습니다.

이러한 잠재적 오류를 방지하기 위해 블록체인 시스템에는

모든 노드가 동일한 원장 상태를 합의할 수 있도록 하는 **효율적인 합의 메커니즘(Consensus Mechanism)**이 필요합니다.

전통적인 **결함 허용(Fault Tolerance)** 메커니즘은 이러한 문제를 완전히 해결하기 어렵기 때문에,

보다 **보편적이고 강력한 합의 알고리즘**이 요구됩니다.

비트코인(Bitcoin)이 사용하는 **작업 증명(Proof of Work, PoW)** 메커니즘은

이 문제를 비교적 효과적으로 해결했지만,

막대한 전력 소모와 에너지 소비라는 단점을 가지고 있습니다.

또한, PoW의 존재로 인해 새로운 블록체인은 계산 공격(Computational Attack)을 피하기 위해

비트코인과 다른 해시 알고리즘을 사용해야 합니다.

예를 들어, 라이트코인(Litecoin)은 비트코인의 SHA256 대신 **SCRYPT**를 채택했습니다.

비잔틴 결함 허용(Byzantine Fault Tolerance, BFT) 메커니즘은

분산 시스템에서 널리 적용되는 범용 솔루션입니다.

NEO는 **PBFT (Practical Byzantine Fault Tolerance)** 알고리즘을 기반으로 한

위임 비잔틴 결함 허용(dBFT, delegated Byzantine Fault Tolerance) 알고리즘을 제안했습니다.

이 알고리즘은 실시간 블록체인 투표에 따라 검증자(Validator) 집합을 동적으로 결정함으로써

효율성을 높이고, 블록 생성 시간 및 트랜잭션 확인 시간을 단축합니다.

또한, 2019년 3월에 공개된 **dBFT 2.0**은

3단계 합의 구조(3-Stage Consensus)와 복구 메커니즘(Recovery Mechanism)을 도입하여

안정성과 견고성을 한층 강화했습니다.

시스템 모델(System Model)

블록체인은 **P2P 네트워크**를 통해 참여자들이 서로 연결된 **분산 원장 시스템**입니다.

이 시스템 내 모든 메시지는 **브로드캐스트(Broadcast)** 방식으로 전송됩니다.

이 시스템에는 두 가지 역할이 존재합니다:

- **일반 노드(Ordinary Nodes)**: 트랜잭션 송수신 및 원장 데이터 수신

- **장부 노드(Bookkeeping Nodes):** 전체 네트워크의 거래를 기록하고 원장을 유지 관리

이론적으로, 이 시스템의 메시지는 손실, 손상, 지연, 중복될 수 있으며 송신 순서와 수신 순서가 일치하지 않을 수 있습니다. 또한 노드의 동작은 예측할 수 없으며, 언제든지 네트워크에 참여하거나 이탈할 수 있습니다. 일부 노드는 데이터를 조작하거나 거짓 정보를 유포하거나, 단순히 작동을 멈출 수도 있습니다. 이러한 문제는 인위적이거나 자연적으로 발생할 수 있습니다.

데이터 전송의 **무결성(Integrity)**과 **진위성(Authenticity)**은 암호학(Cryptography)에 의해 보장되며, 송신자는 반드시 발신 메시지의 해시 값에 서명(Signature)을 첨부해야 합니다.

알고리즘(The Algorithm)

이 알고리즘은 **보안성과 실용성(Security & Usability)**을 모두 보장합니다. 합의 과정에서 오류가 발생한 노드 수가 $\lfloor (N - 1) / 3 \rfloor$ 이하일 경우, 시스템의 기능성과 안정성이 유지됩니다.

여기서,

- $N = |R|$ 은 합의에 참여한 총 노드 수를 의미하며,
- R 은 합의 노드 집합(Consensus Node Set)을 나타냅니다.
- $F = \lfloor (N - 1) / 3 \rfloor$ 는 시스템이 허용할 수 있는 최대 오류 노드 수를 의미합니다.

실제 원장은 **장부 노드(Bookkeeping Nodes)**에 의해 유지되며, **일반 노드(Ordinary Nodes)**는 합의 과정에는 참여하지 않습니다. 아래는 합의 절차 전체의 개념을 설명합니다.

모든 합의의 노드는 현재 합의 상태를 기록하기 위해 **상태 테이블(State Table)**을 유지해야 합니다. 합의의 시작부터 종료까지 사용되는 데이터 집합을 **View**라고 하며, 현재 View 내에서 합의에 도달하지 못하면 **View Change**가 발생합니다. 각 View는 v 라는 번호로 구분되며, 0부터 시작하여 합의가 이루어질 때까지 증가할 수 있습니다.

모든 합의의 노드에는 0부터 시작하는 고유한 번호가 부여됩니다. 마지막 노드의 번호는 $N - 1$ 입니다. 각 합의 라운드에서 하나의 노드가 **Speaker(스피커)** 역할을 맡고, 나머지 노드는 **Delegate(대표자)** 역할을 수행합니다. 스피커의 인덱스 p 는 다음 공식으로 결정됩니다:

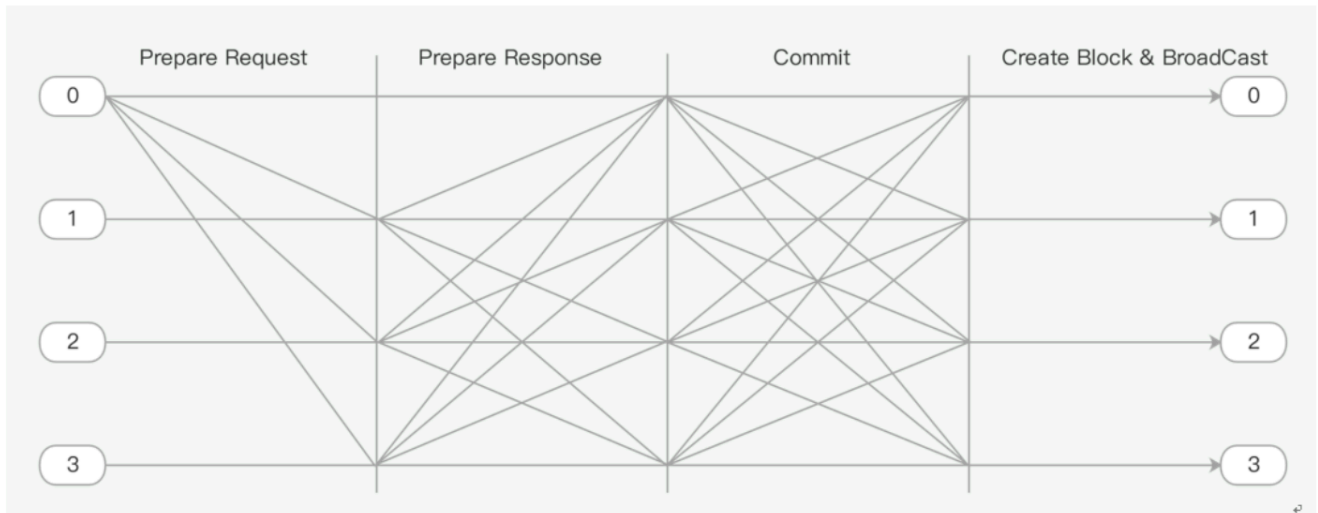
$$p = (h - v) \mod N$$

여기서,

- h 는 현재 블록 높이(Block Height),
 - v 는 현재 View 번호,
 - N 은 합의 노드 수입니다.
- p 의 값 범위는 $0 \leq p < N$ 입니다.

각 합의 라운드에서 새로운 블록이 생성되며, 최소한 $N - F$ 개의 장부 노드 서명이 필요합니다. 블록이 생성된 후 새로운 합의 라운드가 시작되며, 이때 v 는 다시 0으로 초기화됩니다.

일반 절차(General Procedures)



한 라운드의 합의 절차는 위 그림과 같이 총 4단계로 구성됩니다.

1. **스피커(Speaker)**가 Prepare Request 메시지를 브로드캐스트하여 합의를 시작합니다.
2. **대표자(Delegates)**는 Prepare Request 메시지를 수신한 후 Prepare Response를 브로드캐스트합니다.
3. **검증자(Validators)**는 충분한 Prepare Response 메시지를 수신하면 Commit을 브로드캐스트합니다.
4. **검증자(Validators)**는 충분한 Commit 메시지를 수신한 후 새 블록을 생성하고 네트워크에 브로드캐스트합니다.

뷰 변경(View Change)

다음과 같은 상황이 발생할 경우,
Change View Request가 브로드캐스트되어 스피커 교체를 시도합니다:

- 트랜잭션 검증에 실패한 경우
- Prepare Request 또는 Prepare Response 대기 중 타임아웃이 발생한 경우

복구 메커니즘(Recovery Mechanism)

Change View Request를 생성할 때

활성 합의의 노드가 충분하지 않은 경우(즉, Commit을 전송한 노드와 오류 노드의 합의 $\$F\$$ 보다 큰 경우),
합의 노드들은 **Recovery Request** 메시지를 브로드캐스트하여
로컬 합의 컨텍스트(Local Consensus Context)를 갱신합니다.

Recovery Request를 수신하면, 특정 조건이 충족되는 경우

해당 합의 노드는 **Recovery Message**를 생성하고 이를 네트워크에 브로드캐스트합니다.

dBFT2.0의 결함 허용(Fault Tolerance of dBFT2.0)

$\$dBFT2.0\$$ 합의 시스템에서 검증자(Validator)의 총 수를 $\$N\$$ 이라고 할 때,
시스템은 최대 $\$F\$$ 개의 비정상 노드(오류 또는 악의적 노드)를 허용할 수 있습니다.

각 합의 단계(예: Commit, View 변경, 블록 생성 등)에서 합의에 도달하기 위해서는
최소 $\$M\$$ 개의 정상 노드가 필요합니다.

즉, 정상 검증자의 수가 $\$M\$$ 이상인 한, 합의 프로세스는 정상적으로 진행될 수 있습니다.

예를 들어, 검증자 수 $\$N = 4\$$ 인 경우:

$$M = N - \lfloor (N - 1) / 3 \rfloor = 4 - \lfloor 3 / 3 \rfloor = 3$$

따라서 최소 **3개의 정상 검증자**만으로도 합의 시스템은 정상적으로 유지될 수 있습니다.

dBFT2.0의 단일 블록 최종성(Single Block Finality of dBFT2.0)

Neo의 dBFT 1.0 알고리즘은 드물게 발생하는 **네트워크 지연(Network Latency)** 상황에서 단일 블록 포크(Single Block Fork)가 생길 가능성이 있었습니다.

그러나 **dBFT 2.0**은 이 문제를 해결하였으며, 이후로는 포크(Fork)가 발생할 가능성이 없습니다.

그 메커니즘은 다음과 같습니다:

- 새로운 블록을 생성하기 위해서는, 해당 블록 제안(Proposed Block)에 대해 **서로 다른 \$M\$개의 검증자로부터 Commit 메시지를** 수집해야 합니다.
- 검증자는 Commit 메시지를 브로드캐스트한 이후에는 **현재 View를 변경하지 않습니다**.

따라서 블록 생성이 성공했다는 것은 다음을 의미합니다:

1. 최소 \$M\$개의 검증자가 블록 제안에 서명하고, Commit 메시지를 브로드캐스트한 상태입니다. 또한 이 검증자들은 해당 합의 라운드에서 View를 변경하지 않습니다.
2. 남은 검증자 수는 새로운 다른 블록을 생성하기에는 부족합니다.

즉, 특정 블록 높이에서 **새로운 블록의 최종성(Finality)** 이 보장됩니다. 한 번 생성된 블록은 포크되지 않으며, 되돌릴 수 없습니다.

합의 알고리즘

dBFT 2.0 알고리즘(dBFT 2.0 Algorithm)

용어 정의(Terms)

용어(Term)	정의(Definition)
Consensus Node	새로운 블록을 제안하고 해당 블록에 대한 투표를 수행할 수 있는 노드
Normal Node	트랜잭션을 전송하거나 생성할 수 있으며 원장을 저장하지만, 블록 제안이나 투표에는 참여하지 않는 노드
Speaker	네트워크에 블록 제안을 생성하고 브로드캐스트하는 책임을 가진 검증자(Validator)
Delegate	블록 제안에 대해 투표하는 검증자
Candidate	검증자 선출을 위해 후보로 등록된 계정
Validator	후보(Candidate) 중 투표를 통해 선출되어 합의에 참여하는 계정
View	합의 라운드에서 사용되는 데이터 집합. 각 라운드는 View 번호 0에서 시작하며, 합의 실패 시 점진적으로 증가하고 블록 제안이 승인되면 다시 0으로 초기화됨.

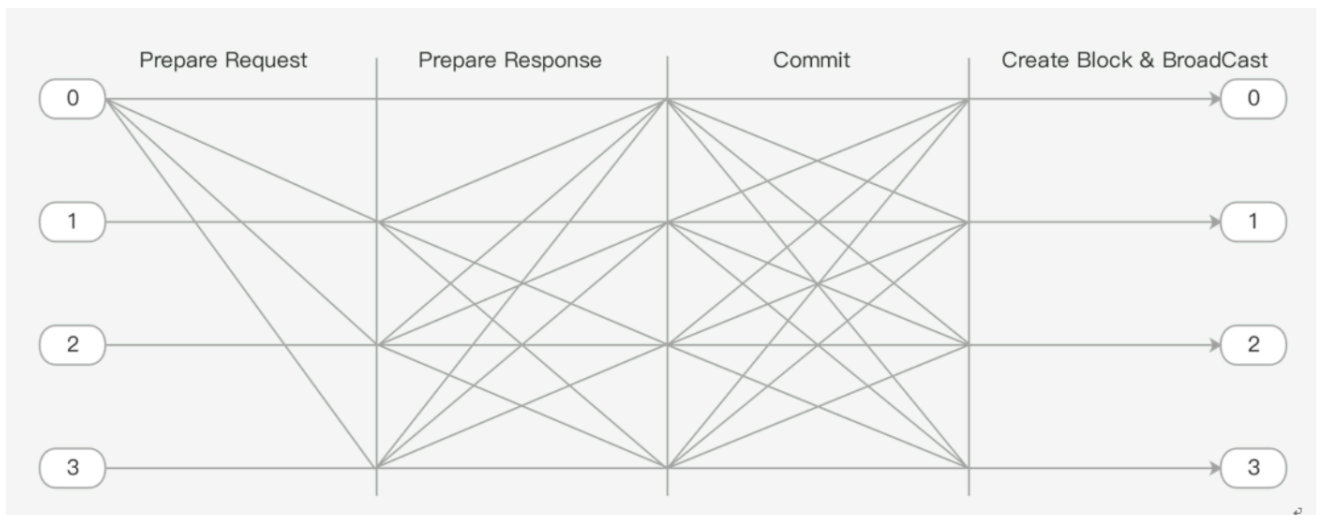
합의 메시지(Consensus Message)

dBFT 2.0에서는 다음 **6가지 합의 메시지 유형**이 정의되어 있습니다:

메시지(Message)	정의(Definition)
Prepare Request	새로운 합의 라운드를 시작하기 위한 메시지
Prepare Response	블록 생성을 위한 모든 트랜잭션이 수집되었음을 다른 검증자에게 알리는 메시지
Commit	충분한 Prepare Response 메시지가 수집되었음을 알리는 메시지
Change View Request	View 변경 시도를 위한 메시지
Recovery Request	합의 데이터 동기화를 요청하는 메시지
Recovery Message	Recovery Request에 대한 응답 메시지

합의 흐름(Consensus Flow)

3단계 합의 흐름(3-Stage Consensus Flow)



하나의 합의 라운드는 위 그림과 같이 4단계로 구성됩니다.

1. 스피커(Speaker)가 Prepare Request 메시지를 브로드캐스트하여 합의를 시작합니다.
2. 대표자(Delegates)가 Prepare Request 메시지를 수신한 후 Prepare Response 메시지를 브로드캐스트합니다.
3. 검증자(Validators)가 충분한 Prepare Response 메시지를 수신하면 Commit 메시지를 브로드캐스트합니다.
4. 검증자(Validators)가 충분한 Commit 메시지를 수신하면 새로운 블록을 생성하여 네트워크에 브로드캐스트합니다.

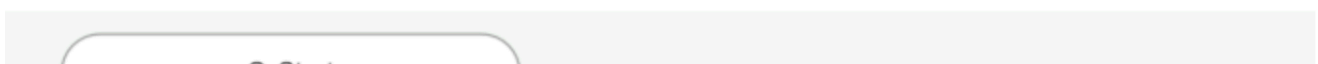
다음 두 변수를 도입합니다:

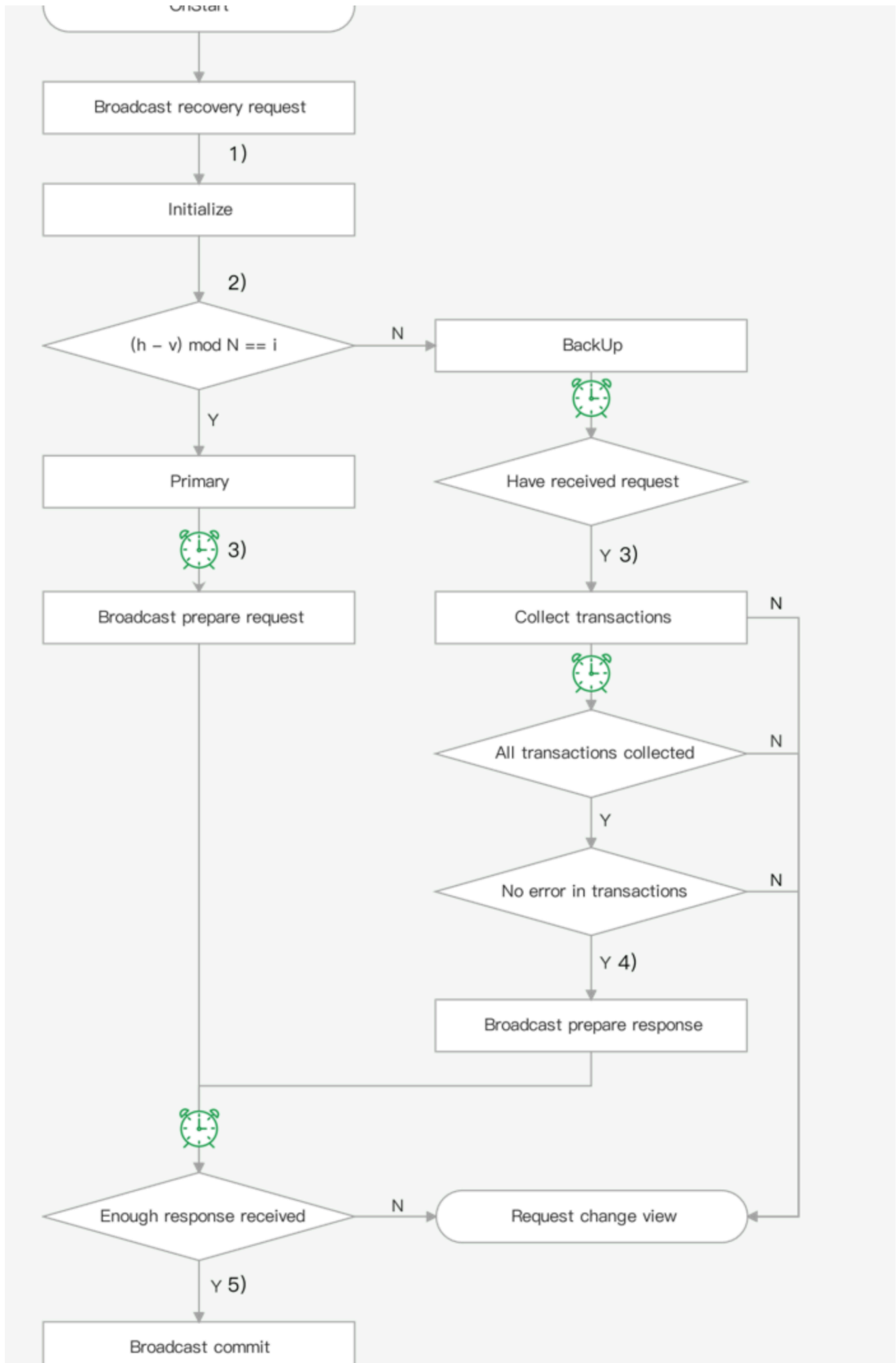
$$F = \left\lfloor \frac{N-1}{3} \right\rfloor$$

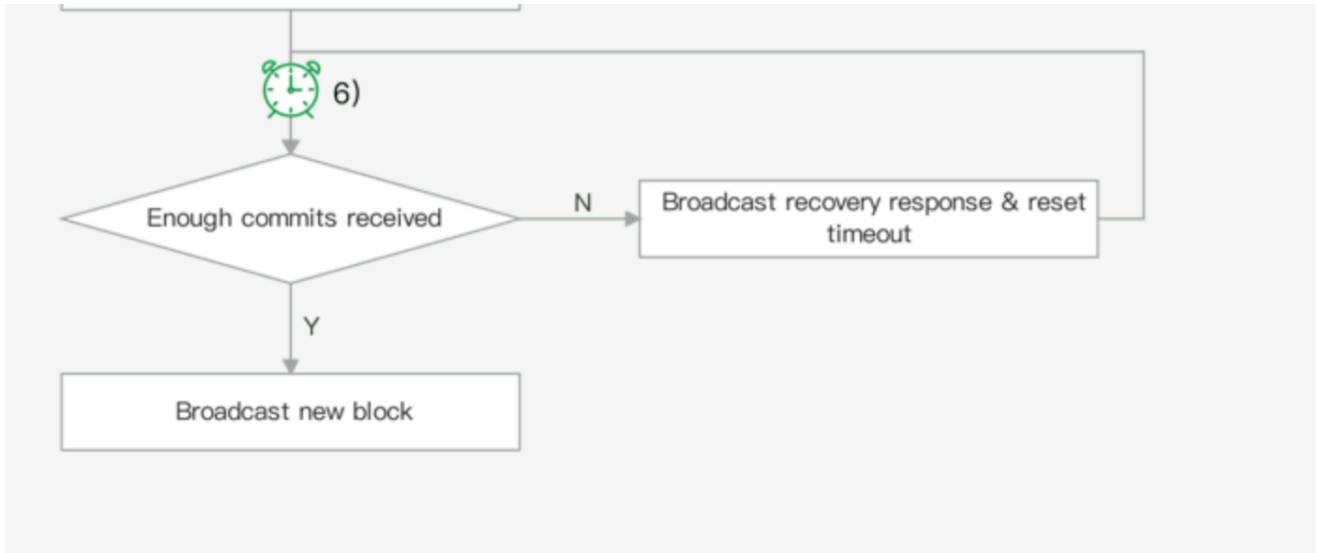
$$M = N - F = N - \left\lfloor \frac{N-1}{3} \right\rfloor$$

여기서 \$N\$은 검증자(Validator)의 총 수를 의미합니다.

아래는 일반적인 알고리즘의 흐름도입니다.







1) 로컬 합의 정보 초기화(Initialize Local Consensus Information)

1. 합의 컨텍스트(Consensus Context) 를 초기화합니다.
2. 스피커(Speaker)를 설정합니다.
 - 스피커의 인덱스는 $(h - v) \bmod N$ 으로 계산됩니다.
 - 여기서 h 는 현재 블록 높이(Block Height), v 는 현재 View, N 은 검증자 수입니다.
3. 타임아웃 시간(Timeout Period) 을 설정합니다.
 - 스피커가 다른 노드로부터 RecoveryMessage를 처리 중인 경우:
타임아웃은 $2^{v+1} \times T_{block}$ 로 설정됩니다.
여기서 T_{block} 은 블록 생성 시간(현재 약 15초)입니다.
 - 스피커가 이전 합의에서 블록 생성에 참여하지 않았을 경우:
타임아웃은 T_{block} 로 설정됩니다.
 - 스피커가 이전 합의에 참여한 경우:
 D = 현재 시각 - 마지막 블록 생성 시각으로 계산하고,
 $D < T_{block}$ 이면 타임아웃을 $T_{block} - D$ 로, 그렇지 않으면 0으로 설정합니다.
4. 합의가 새로 시작된 경우, 현재 합의 컨텍스트를 획득하기 위해
Recovery Request 메시지를 브로드캐스트합니다.

2) 검증자(Validators)는 네트워크를 모니터링하고, 타임아웃 전까지 트랜잭션을 수집합니다.

3) 합의 시작(Start Consensus)

스피커(Speaker)의 경우:

1. 타임아웃 이후, 합의 정책(Consensus Policy)에 따라 메모리 풀(Memory Pool)에서 트랜잭션을 선택하고, 해당 트랜잭션 해시를 포함한 Prepare Request 메시지를 생성 및 브로드캐스트하여 새로운 합의 라운드를 시작합니다.
2. 선택된 트랜잭션 중 500개 단위로 패키징하여 브로드캐스트합니다.
3. 타임아웃을 다음과 같이 설정합니다:

$$T_{timeout} = (2^{v+1-k(v)}) \times T_{block}$$

여기서,

$$k(v) = \begin{cases} 1, & v = 0 \\ 0, & v \neq 0 \end{cases}$$

대표자(Delegates)의 경우:

- 타임아웃이 발생하기 전에 스피커로부터 Prepare Request 메시지를 수신한 경우:
 1. 메시지의 유효성과 로컬 합의 컨텍스트(Local Consensus Context)와의 일치 여부를 검증합니다.
 2. 로컬 타임아웃을 $\frac{2}{M} \times T_{\text{block}}$ 만큼 연장합니다.
 3. 로컬 합의 컨텍스트를 갱신합니다.
 4. 메시지에 포함된 각 트랜잭션 해시(Hash)에 대해, 메모리 풀(Memory Pool) 또는 미검증 트랜잭션 풀(Unverified Transaction Pool)에서 해당 트랜잭션을 획득하여 합의 컨텍스트에 추가합니다.
 5. 4단계에서 찾을 수 없는 트랜잭션이 있을 경우, 다른 노드들로부터 해당 트랜잭션을 요청(Request)합니다.
- 위 절차를 완료하기 전에 타임아웃이 발생하거나 유효성 검증에 실패할 경우, **뷰 변경(View Change)** 을 시도합니다.

4) Prepare Response 브로드캐스트(Broadcast Prepare Response)

대표자가 타임아웃 전에 Prepare Request에 필요한 모든 트랜잭션을 수집한 경우:

1. 수신된 각 트랜잭션에 대해, 트랜잭션 검증에 실패하거나 합의 정책(Consensus Policy)에 위배될 경우 **뷰 변경(View Change)** 을 시도하고, 그렇지 않으면 트랜잭션을 합의 컨텍스트에 추가합니다.
 2. Prepare Response 메시지를 브로드캐스트합니다.
 3. 로컬 타임아웃을 $\frac{2}{M} \times T_{\text{block}}$ 만큼 연장합니다.
- 타임아웃 전에 모든 트랜잭션을 확보하지 못할 경우, **뷰 변경(View Change)** 을 시도합니다.

5) Prepare Response 수집 및 Commit 브로드캐스트

(Collect Prepare Response and Broadcast Commit)

- 스피커 및 Prepare Request를 수신한 대표자들은, 타임아웃 전에 **M 개의 서로 다른 대표자들로부터** Prepare Response 메시지를 수신한 경우:
 - 수신된 각 Prepare Response 메시지에 대해:
 1. 메시지의 유효성과 로컬 합의 컨텍스트와의 일치 여부를 검증합니다.
 2. 로컬 타임아웃을 $\frac{2}{M} \times T_{\text{block}}$ 만큼 연장합니다.
 - 이후 Commit 메시지를 브로드캐스트합니다.
- 충분한 Prepare Response 메시지를 수신하지 못한 경우, **뷰 변경(View Change)** 을 시도합니다.

6) Commit 메시지 수집 및 새로운 블록 생성

(Collect Commit Message and Create New Block)

- Prepare Request에 포함된 모든 트랜잭션을 확보한 검증자(Validator)는 서로 다른 **M 개의 검증자로부터** Commit 메시지를 수신한 경우 다음을 수행합니다:
 - 수신된 각 Commit 메시지에 대해:
 1. 메시지의 유효성과 로컬 합의 컨텍스트와의 일치 여부를 검증합니다.
 2. 로컬 타임아웃을 $\frac{4}{M} \times T_{\text{block}}$ 만큼 연장합니다.
 - 이후 새 블록(New Block)을 생성하고 네트워크에 브로드캐스트합니다.

- 충분한 Commit 메시지를 수신하지 못한 경우,
Recovery Message를 브로드캐스트하고 타임아웃을 $2 \times T_{\text{block}}$ 으로 설정합니다.

7) 1단계로 돌아가 새로운 합의 라운드 시작

(Go back to Step 1 to Start a New Round of Consensus)

모든 절차가 완료되면, 합의 프로세스는 다시 **1단계(Initialize Local Consensus Information)** 로 돌아가 새로운 합의 라운드를 시작합니다.

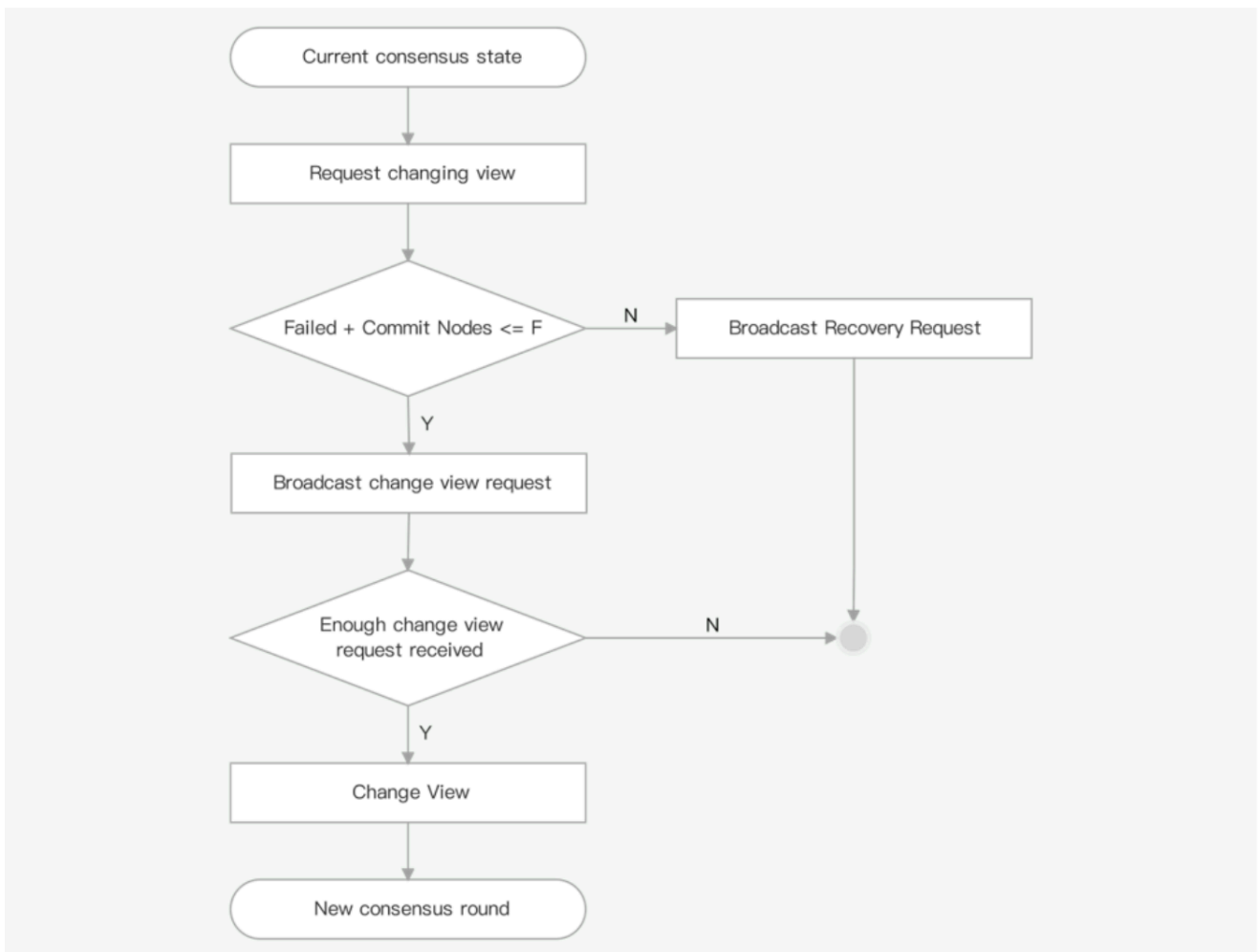
뷰 변경 요청(Change View Request)

트리거 조건(Triggering Conditions)

다음과 같은 상황에서 대표자(Delegate)는 **스피커 교체(View Change)** 를 시도하기 위해 Change View Request 메시지를 브로드캐스트합니다:

- 트랜잭션 검증(Transaction Verification)에 실패한 경우
- Prepare Request 또는 Prepare Response 메시지를 대기하는 동안 **타임아웃(Timeout)** 이 발생한 경우

흐름(Flow)



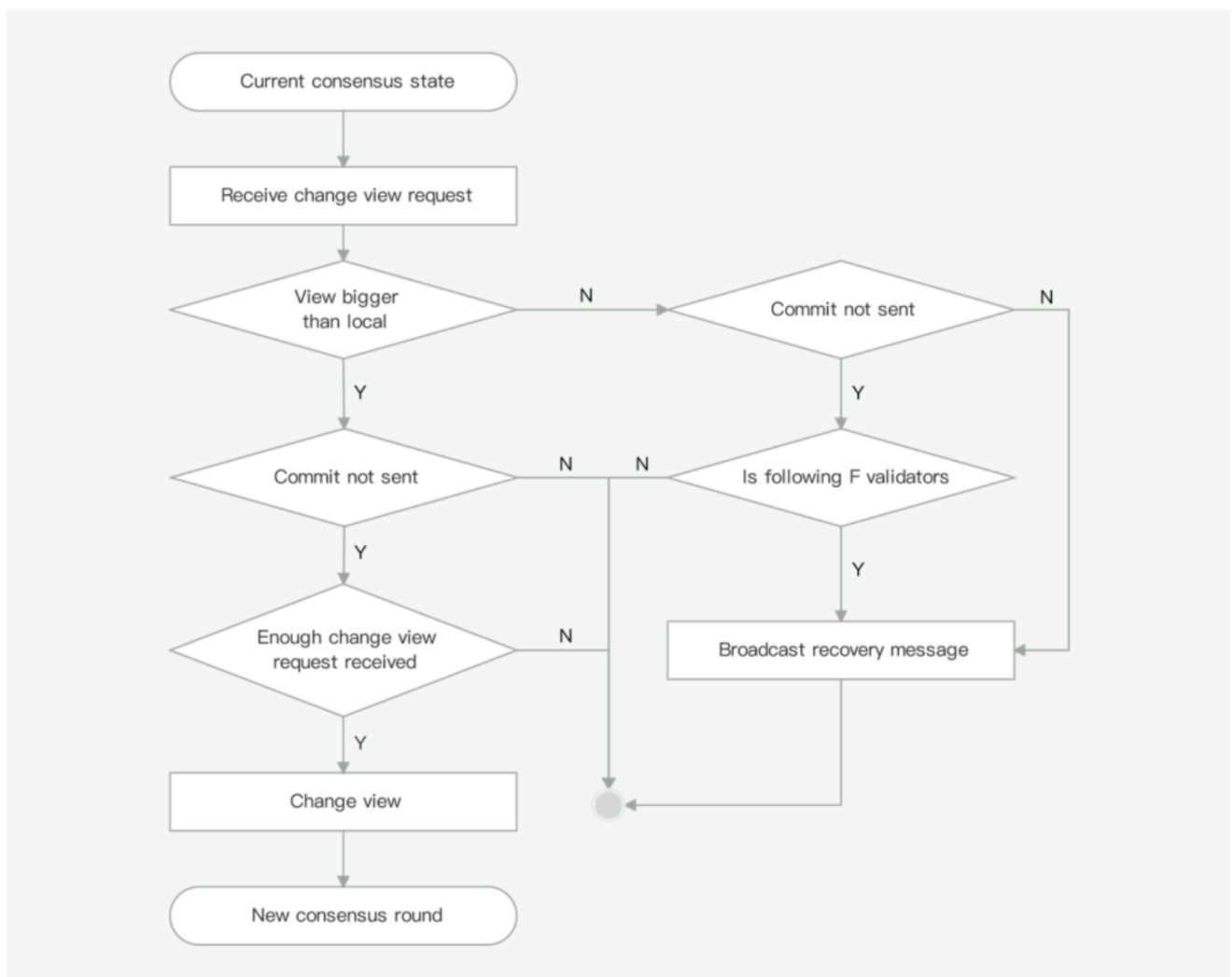
- 타임아웃을 $2^{v+2} \times T_{\text{block}}$ 으로 설정합니다.
- Commit 메시지를 이미 전송한 노드의 수와 결함 노드(Fault Node)의 수(즉, 블록 시간 동안 메시지를 송신하지 않은 검증자)가 합쳐져

- \$F\$보다 큰 경우, Recovery Request 메시지를 브로드캐스트합니다.
3. 그렇지 않은 경우, Change View Request 메시지를 브로드캐스트합니다.
 이후 수신된 Change View Request의 수를 확인합니다.
 만약 \$M\$개 이상의 검증자가 뷰 변경(View Change)에 합의했다면,
 로컬 뷰(Local View)를 변경하고, 합의 컨텍스트를 초기화한 뒤
 새 뷰(View)에 따라 다음 라운드의 스피커(Speaker)를 결정합니다.

처리 로직(Process Logic)

검증자(Validator)가 Change View Request 메시지를 수신했을 때 다음과 같이 처리합니다:

- 수신된 메시지의 View 번호가 로컬 View 번호보다 크지 않다면,
해당 메시지는 Recovery Request로 처리합니다.
- 메시지의 유효성을 검증합니다.
- 수신된 Change View Request 메시지의 수를 확인합니다.
만약 \$M\$개 이상의 검증자가 뷰 변경(View Change)에 합의했다면,
로컬 뷰를 변경하고, 합의 컨텍스트를 초기화하며,
새로운 뷰(View)에 따라 다음 라운드의 스피커를 결정합니다.



복구 메시지(Recovery Message)

내용(Content)

- M 개 이하의 대표자(Delegates)로부터 수신된 Change View Request 메시지
- Prepare Request 및 Prepare Response 메시지
- Commit 메시지

트리거 조건(Triggering Conditions)

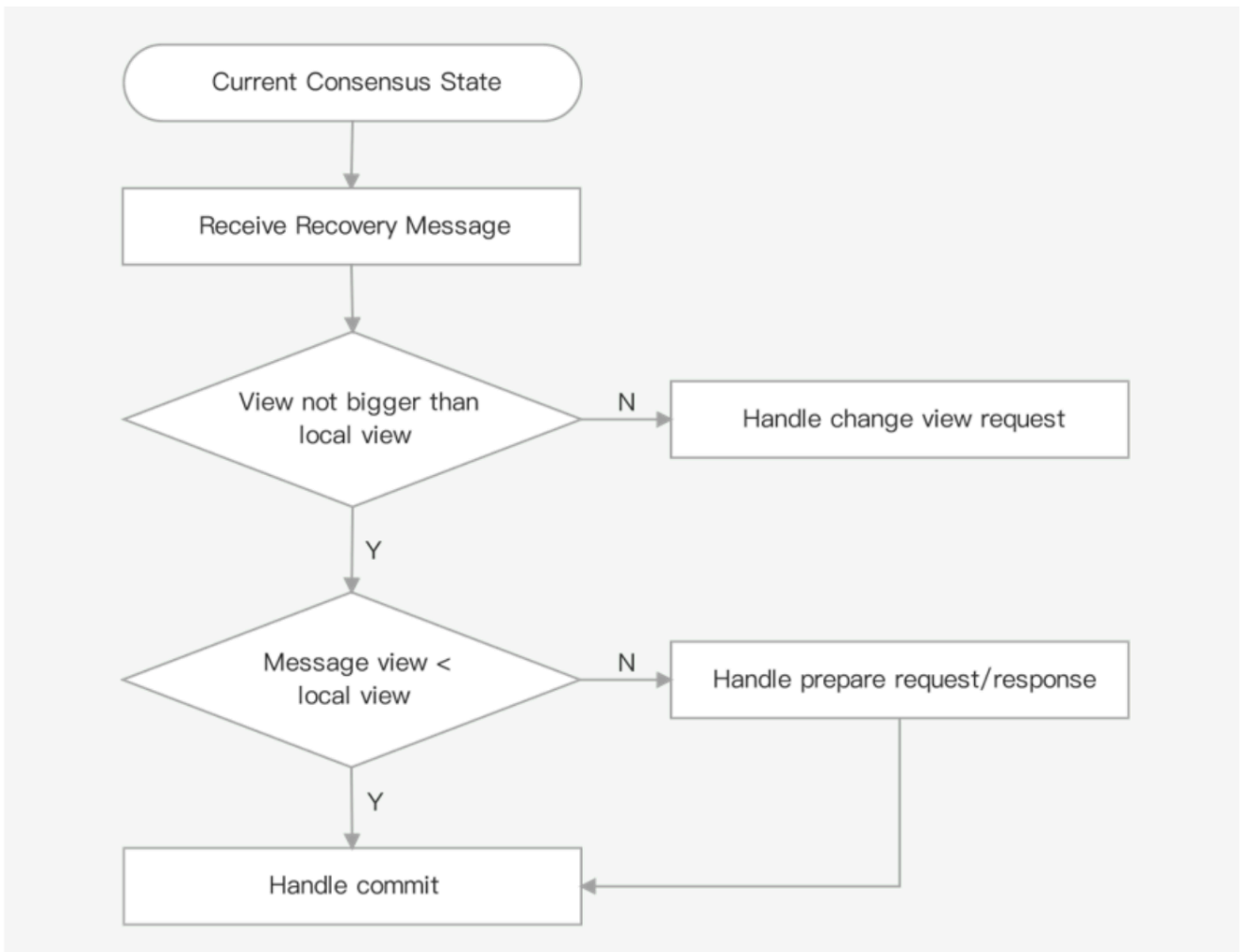
다음과 같은 조건에서 Recovery Message가 브로드캐스트됩니다:

- Recovery Request 메시지를 수신했을 때,
해당 노드가 이미 Commit 메시지를 브로드캐스트했거나,
그 노드의 인덱스가 다음 구간에 속하는 경우:
$$[0, j + F - N] \cup [j + 1, j + F]$$

여기서 j 는 Recovery Request 발신자의 인덱스입니다.
- Change View Request 메시지를 수신했을 때,
메시지의 View 번호가 로컬 View 번호보다 크지 않은 경우,
해당 메시지는 Recovery Request로 처리됩니다.
- Commit 메시지를 대기하는 동안 타임아웃이 발생한 경우,
Recovery Message를 브로드캐스트하여 Commit 메시지를 다시 전송합니다.
(이는 주로 네트워크 지연 또는 불안정으로 인한 일반적인 상황입니다.)

처리 흐름(Process Flow)

1. 메시지와 로컬 합의 컨텍스트(Local Consensus Context)의 유효성을 검증합니다.
 - 만약 메시지의 View가 로컬 View보다 크고,
이 노드가 이미 Commit 메시지를 전송한 상태라면,
해당 메시지는 **무시(Ignore)** 합니다.
2. 그렇지 않고 메시지의 View가 로컬 View보다 클 경우,
메시지 내부의 Change View Request를 처리합니다.
3. 메시지의 View가 로컬 View와 동일할 경우:
 - 내부의 Prepare Request 메시지를 처리합니다.
 1. 이 노드가 아직 Prepare Request를 전송하거나 수신하지 않았다면, 이를 처리합니다.
 2. 반대로, 이 노드가 스피커(Speaker)인 경우 Prepare Request 메시지를 브로드캐스트합니다.
 - 내부의 Prepare Response 메시지를 처리합니다.
4. 메시지의 View가 로컬 View보다 작거나 같을 경우,
내부의 Commit 메시지를 처리합니다.



Change View Request, Recovery Request, Recovery Message의 메커니즘은 네트워크 지연(Network Timeout), 비정상 노드(Malicious or Fault Nodes) 등으로 인한 타임아웃 상황에서도 합의 프로세스의 **안정성(Safety)**을 유지합니다.

합의 정책(Consensus Policy)

합의 정책은 다음과 같은 시나리오에서 사용됩니다:

- 다른 노드로부터 트랜잭션을 수신했을 때, 노드는 합의 정책을 기반으로 검증을 수행하여, 정책에 위배되는 트랜잭션을 필터링합니다.
- 트랜잭션을 수신할 때, 합의 모듈은 해당 트랜잭션이 합의 정책을 충족하는지 검증해야 하며, 그렇지 않은 경우 **뷰 변경(View Change)**을 시도합니다.
- 검증자(Validator)는 합의 정책을 활성화한 상태에서 자신의 컨텍스트 내 트랜잭션을 필터링해야 하며, 검증된 트랜잭션만 메모리 풀(Memory Pool)에 추가할 수 있습니다.
- 스피커(Speaker)는 새로운 Prepare Request를 생성할 때, 합의 정책에 따라 메모리 풀에서 트랜잭션을 선택해야 합니다.

합의 프로토콜(Consensus Protocol)

합의 메시지 형식(Consensus Message Format)

ExtensiblePayload

크기 (Size)	필드(Field)	타입 (Type)	설명(Description)
?	Category	String	메시지 카테고리, 현재는 dBFT
4	ValidBlockStart	uint	메시지가 유효한 블록 시작 높이
4	ValidBlockEnd	uint	메시지가 유효한 블록 종료 높이
20	Sender	UInt160	현재 합의 노드의 주소 해시
?	Data	byte[]	ChangeView, PrepareRequest, PrepareResponse, Commit, RecoveryMessage, RecoveryRequest 등의 데이터 포함
?	Witness	Witness	실행 스크립트(Invocation Script)와 검증 스크립트(Verification Script)를 포함

ConsensusMessage

ConsensusMessage는 모든 합의 메시지 타입의 **기본 추상 타입(Abstract Type)**입니다.
다른 합의 메시지 타입들은 이 타입을 상속(Inherit)합니다.

크기 (Size)	필드(Field)	타입(Type)	설명(Description)
1	Type	ConsensusMessageType	메시지 유형: ChangeView, PrepareRequest, PrepareResponse, Commit, RecoveryMessage, RecoveryRequest
4	BlockIndex	uint	메시지가 생성된 블록 높이
1	ValidatorIndex	byte	검증자 배열 내에서 송신자의 인덱스
1	ViewNumber	byte	메시지가 생성된 View 번호

ChangeView

크기(Size)	필드(Field)	타입(Type)	설명(Description)
8	Timestamp	ulong	ChangeView 메시지가 생성된 시점의 타임스탬프
1	Reason	ChangeViewReason	뷰 변경(View Change)이 발생한 이유

Commit

크기(Size)	필드(Field)	타입(Type)	설명(Description)
?	Signature	byte[]	메시지 서명(Signature)

PrepareRequest

크기(Size)	필드(Field)	타입(Type)	설명(Description)
4	Version	uint	기본값은 0
32	PrevHash	UInt256	이전 블록의 해시값
8	Timestamp	ulong	PrepareRequest 메시지가 생성된 시점의 타임스탬프
?	TransactionHashes	UInt256[]	블록 내 트랜잭션의 해시 목록

PrepareResponse

크기(Size)	필드(Field)	타입(Type)	설명(Description)
32	PreparationHash	UInt256	해당 PrepareRequest의 해시(Hash)

RecoveryMessage

크기(Size)	필드(Field)	타입(Type)	설명(Description)
?	ChangeViewMessages	Dictionary<int, ChangeViewPayloadCompact>	수집된 ChangeView 메시지
?	PrepareRequestMessage	PrepareRequest	현재의 PrepareRequest 메시지
32	PreparationHash	UInt256	Prepare Request의 해시
?	PreparationMessages	Dictionary<int, PreparationPayloadCompact>	수집된 Prepare Response 메시지
?	CommitMessages	Dictionary<int, CommitPayloadCompact>	수집된 Commit 메시지

RecoveryRequest

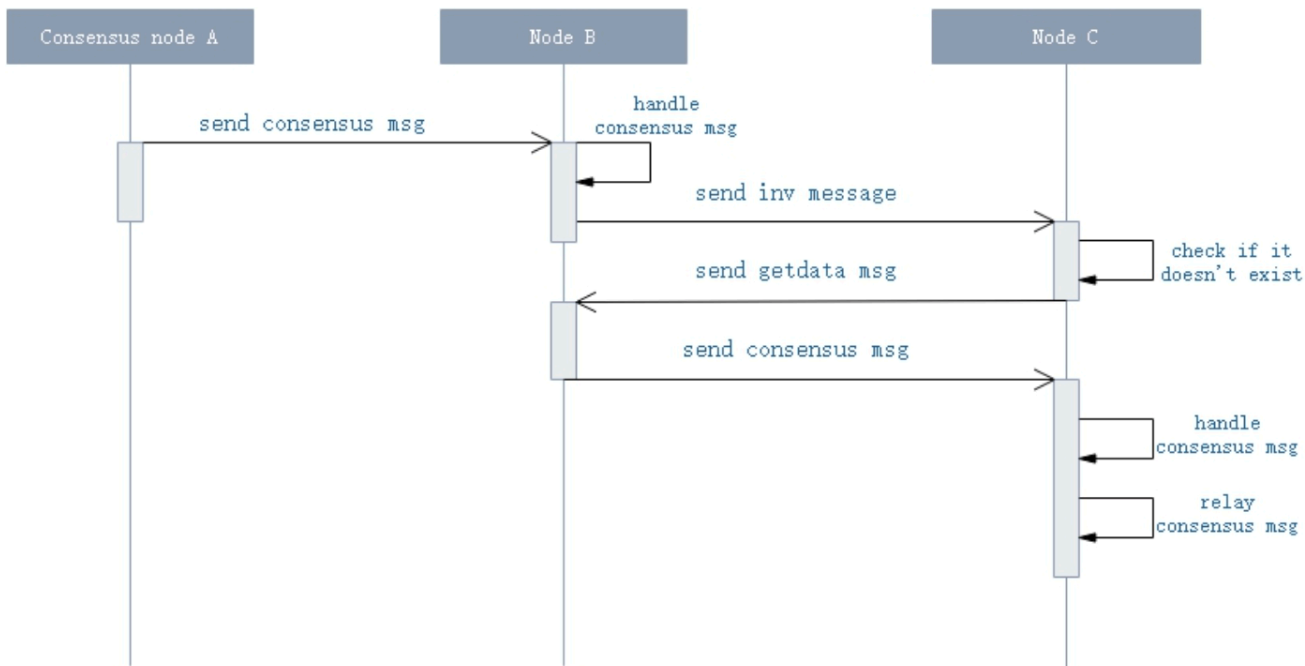
크기(Size)	필드(Field)	타입(Type)	설명(Description)
8	Timestamp	ulong	메시지가 생성된 시점의 타임스탬프

전송 프로토콜(Transport Protocol)

합의 메시지가 P2P 네트워크로 전송될 때,
이는 일반 메시지와 동일하게 **브로드캐스트(Broadcast)** 되어 전달됩니다.
그 이유는 합의의 노드 간에 서로의 **IP 주소를 직접적으로 알지 못하기** 때문입니다.
즉, 합의의 노드들은 직접 연결되어 있지 않으며,

일반 노드(Ordinary Node)도 합의 메시지를 수신할 수 있습니다.

합의 메시지의 브로드캐스트 흐름은 다음과 같습니다:



1. 합의 노드 A가 연결된 노드들(예: 노드 B)에게 consensus 메시지를 직접 브로드캐스트합니다.
2. 노드 B는 consensus 메시지를 수신한 후, 우선 해당 메시지를 처리하고, 이후 다른 노드로 전달(forward)하기 전에 inv 메시지를 전송합니다.
inv 메시지에는 consensus 메시지의 페이로드(payload)에 대한 해시 데이터가 포함됩니다 (예: 노드 C로 전송).
3. 노드 C가 해당 해시값에 해당하는 데이터를 이미 알고 있다면, inv 메시지를 무시합니다.
그렇지 않으면 4단계로 이동합니다.
4. 노드 C는 inv 메시지 내 해시 데이터를 포함한 getdata 메시지를 노드 B에게 전송합니다.
5. 노드 B는 getdata 메시지를 수신한 후, 해당 데이터를 포함한 consensus 메시지를 노드 C로 전송합니다.
6. 노드 C는 consensus 메시지를 수신한 후, 합의 모듈(Consensus Module)을 통해 메시지를 처리하고, 다시 해당 메시지를 브로드캐스트하며 2단계로 돌아갑니다.

inv와 getdata 메시지는 모두 InvPayload를 메시지 운반체(Message Carrier)로 사용합니다.
그 정의는 다음과 같습니다:

InvPayload

크기(Size)	필드(Field)	타입(Type)	설명(Description)
1	Type	InventoryType	메시지 유형
?	Hashes	UInt256[]	브로드캐스트되거나 요청된 해시(Hash) 목록

InventoryType에는 다음 세 가지 유형이 존재합니다:

- 0x2b : **Transaction** — Hashes가 트랜잭션에 해당합니다.
- 0x2c : **Block** — Hashes가 블록에 해당합니다.
- 0x2e : **Consensus** — Hashes가 ConsensusPayload 메시지에 해당합니다.

합의 메시지 처리(Consensus Message Process)

검증(Verification)

다음 조건 중 하나라도 해당하면, 합의 메시지는 무시(Ignore) 됩니다:

1. ValidBlockStart가 ValidBlockEnd보다 작은 경우
2. 현재 블록 높이가 [ValidBlockStart, ValidBlockEnd] 범위를 벗어난 경우
3. 메시지 발신자가 합의 화이트리스트(Consensus Whitelist)에 포함되어 있지 않은 경우
4. 검증 스크립트(Verification Script)가 실패했거나, Category가 "dBFT"가 아닌 경우
5. 해당 노드가 이미 새 블록을 전송한 경우
6. 합의 메시지 데이터 형식이 올바르지 않은 경우
7. message.BlockIndex가 현재 블록 높이보다 낮은 경우
8. ConsensusPayload.ValidatorIndex가 현재 합의 노드 배열의 인덱스 범위를 벗어났거나, payload.Sender가 올바른 해시와 불일치할 경우

처리 절차(Process)

1. 스피커(Speaker)가 전송한 PrepareRequest 메시지 수신 시 (블록 제안 데이터 포함):

1. PrepareRequest를 이미 수신했거나, 노드가 뷰 변경(View Change) 중이라면 무시합니다.
2. message.ValidatorIndex가 현재 라운드 스피커의 인덱스와 다르거나, PrepareRequest.ViewNumber가 현재 View 번호와 다르면 무시합니다.
3. message.Version 또는 message.PrevHash가 로컬 컨텍스트와 다를 경우 무시합니다.
4. 트랜잭션 수가 MaxTransactionsPerBlock을 초과하면 무시합니다.
5. message.Timestamp가 이전 블록의 타임스탬프보다 크지 않거나, 현재 시각보다 8블록 이상 미래일 경우 무시합니다.
6. 제안된 트랜잭션 중 일부가 이미 블록체인에 포함된 경우 무시합니다.
7. 합의 컨텍스트를 갱신하고, 기존에 수신된 잘못된 서명을 초기화합니다.
(PrepareResponse가 먼저 도착했을 수 있기 때문입니다.)
8. 스피커의 서명을 현재 합의 컨텍스트에 저장합니다.
9. 요청에 트랜잭션이 없는 경우, 로컬에 저장된 PrepareResponse를 바로 확인하여 충분한 응답이 있다면 Commit 메시지를 브로드캐스트합니다.
10. 제안된 블록의 트랜잭션을 메모리 풀(Memory Pool)에서 수집하고 검증합니다.
 - 검증에 실패하거나 정책에 위배되면 무시합니다.
 - 그렇지 않으면 트랜잭션을 합의 컨텍스트에 추가합니다.
11. 미확인 트랜잭션 풀(Unconfirmed Transaction Pool)에서 필요한 트랜잭션을 검증하고, 합의 컨텍스트에 추가합니다.
12. 누락된 트랜잭션 해시 목록을 포함한 getdata 메시지를 브로드캐스트합니다.

2. 합의 노드가 서명과 함께 전송한 PrepareResponse 수신 시:

1. 메시지의 View가 현재 View와 다르면 무시합니다.
2. 송신자의 서명이 이미 저장되어 있거나, 노드가 뷰 변경 중이라면 무시합니다.
3. 아직 PrepareResponse를 수신하지 않았다면 임시로 저장합니다.
(이후 PrepareRequest를 수신하면 정식 처리합니다.)
4. 서명의 유효성을 검증하고, 통과하면 저장합니다.
실패하면 무시합니다.
5. 노드가 이미 Commit을 전송한 상태라면 무시합니다.
6. 노드가 이미 PrepareRequest를 전송 또는 수신한 경우,
서명 수를 확인합니다.

\$N - f\$ 개 이상의 유효한 서명이 모이면 Commit을 브로드캐스트하고,
\$N - f\$ 개의 Commit 메시지를 수신한 후 블록을 생성합니다.

3. 합의 노드가 전송한 ChangeView 수신 시:

1. 메시지의 View 번호가 현재 View보다 작거나 같으면,
RecoveryMessage를 전송합니다.
2. 노드가 이미 Commit을 전송한 상태라면 무시합니다.
3. 동일한 새로운 View 번호를 가진 ChangeView 메시지가
\$N - f\$ 개 이상 수신된 경우, View 변경(View Change)을 수행하고,
새로운 View 번호로 합의 프로세스를 재설정합니다.

4. \$N - f\$ 개의 PrepareResponse 이후, 합의 노드의 Commit 수신 시:

1. 동일한 노드로부터 이미 수신된 Commit 메시지라면 무시합니다.
2. 서명이 유효한 경우, 메시지를 합의 컨텍스트에 저장합니다.
\$N - f\$ 개의 Commit이 수신되면 블록을 생성하고 브로드캐스트합니다.

5. 합의 시작 시 또는 커밋 및 실패 노드 수 합의 \$f\$를 초과한 경우,

합의 노드가 RecoveryRequest를 수신할 때:

1. 이미 동일한 요청을 수신했다면 무시합니다.
2. 노드가 이미 Commit을 전송했거나,
송신자 인덱스보다 \$f\$ 이하 차이로 뒤따르는 노드라면 응답합니다.
3. 응답해야 하는 경우, RecoveryMessage를 전송합니다.

6. 합의 노드가 RecoveryRequest 또는 Commit 타임아웃 이후

RecoveryMessage를 수신했을 때:

1. 메시지의 View 번호가 로컬 View보다 크다면 내부의 ChangeView를 처리합니다.
2. 메시지의 View 번호가 로컬 View와 동일하고,
노드가 View 변경 중이 아니며 Commit을 전송하지 않았다면
내부의 PrepareRequest 및 PrepareResponse를 처리합니다.
3. 메시지의 View 번호가 로컬 View 이하일 경우,
내부의 Commit 메시지를 처리합니다.

7. 타이머 이벤트(OnTimer) 수신 시:

1. 타이머의 높이 또는 View 번호가 로컬 컨텍스트와 다르다면 무시합니다.
2. 스피커 타임아웃 시:
 - 첫 번째 타임아웃에서는 PrepareRequest를 브로드캐스트합니다.
 - 이후 타임아웃에서는,
이미 Commit을 전송한 경우 RecoveryMessage를,
그렇지 않으면 ChangeView를 브로드캐스트합니다.
3. 대표자 타임아웃 시:
 - 이미 Commit을 전송한 경우 RecoveryMessage를,
 - 그렇지 않으면 ChangeView를 브로드캐스트합니다.

8. PersistCompleted 이벤트 수신 시:

- 합의 프로세스를 초기화(Reset)합니다.

9. 합의용 새로운 트랜잭션(New Transaction) 수신 시:

1. 노드가 이미 PrepareRequest 또는 PrepareResponse를 전송했거나, 뷰 변경 중이거나, 이번 라운드에서 이미 새 블록을 전송했다면 무시합니다.
2. 동일한 트랜잭션을 이미 수신한 경우 무시합니다.
3. 제안 블록(Proposed Block)에 포함되지 않은 트랜잭션이라면 무시합니다.
4. 트랜잭션 검증에 실패한 경우 ChangeView를 브로드캐스트합니다.
5. 트랜잭션을 제안 블록에 저장합니다.
6. 트랜잭션이 Oracle 트랜잭션인 경우 해당 로직을 처리합니다.
7. 노드가 대표자인 경우:
새 블록이 MaxBlockSize 또는 MaxBlockSystemFee 기준에 부합하지 않으면 ChangeView 메시지를 브로드캐스트합니다.
또한 로컬의 PrepareResponse 수집 상태를 확인하여, 충분한 응답이 존재할 경우 Commit 메시지를 브로드캐스트합니다.

합의 노드 선출(Consensus Nodes Election)

Neo는 개방적(Open) 이고 투명한(Transparent) 블록체인 네트워크입니다.
누구나 트랜잭션을 발생시켜 검증자 후보(Validator Candidate) 로 등록할 수 있으며,
또는 투표를 통해 특정 후보를 합의 노드(Consensus Node) 로 선출할 수 있습니다.
위원회 구성원(Committee Members)과 검증자(Validators)는 이러한 투표 결과에 따라 선출됩니다.

참고 (NOTE)

위원회 구성원은 투표를 통해 Neo 네트워크의 일부 설정을 변경할 수 있는 권한을 가집니다.
현재 변경 가능한 항목에는 다음이 포함됩니다:

- 네트워크 전송 시 바이트당 수수료 (Fee per Byte)
- 실행 수수료 계수 (Execution Fee Factor)
- 스토리지 가격 (Storage Price)
- 계정 차단/차단 해제 (Blocking / Unblocking Account) 등

후보자에게는 별도의 의무(Duty)가 부여되지 않습니다.
다만, 위원회 구성원(Committee Members) 과 검증자(Validators) 는
가장 많은 표를 얻은 일정 수의 후보들 중에서 선출됩니다.

이들의 관계는 다음 그림과 같이 설명됩니다.
위원회 구성원과 검증자 사이에는 명시적인 연결 관계가 없지만,
기본 위원회 구성원 수(21명)가 검증자 수(예: 7명)보다 많기 때문에
일반적으로 검증자는 위원회의 부분집합(Subset) 으로 볼 수 있습니다.



Neo 네트워크의 모든 주소(Address)는 단 하나의 주소에만 투표할 수 있습니다.
(해당 주소가 후보자일 수도 있고 아닐 수도 있습니다.)
후보자의 득표 수는 그에게 투표한 유권자들이 보유한 NEO의 총합으로 계산됩니다.

투표는 **동적(Dynamic)** 이고 **지속적(Continuous)** 인 과정입니다.
즉, 투표자의 NEO 잔고가 변동되면,
이전에 투표한 주소의 득표 수도 자동으로 변경됩니다.
이에 따라 합의 노드와 위원회 구성원 명단은 매 21블록마다 갱신됩니다.

참고 (NOTE)

제네시스 블록(Genesis Block)은 네트워크의 첫 번째 블록으로,
NextConsensus 값이 예비 합의 노드(Standby Consensus Nodes)의
다중 서명 컨트랙트(Multi-Signature Contract) 스크립트 해시로 설정됩니다.

대표자에서 스피커로(From Delegate to Speaker)

스피커(Speaker)는 다음 제안 블록(Proposal Block)을 생성하는 합의 노드(Consensus Node)입니다.
합의 노드 목록은 위에서 설명한 투표 메커니즘을 통해 얻어지며,
스피커는 다음의 공식에 의해 결정됩니다:

$$p = (h - v) \mod N$$

여기서,

- h : 제안 블록의 높이 (Block Height)
- v : 뷰 번호(View Number, 0부터 시작)
- N : 합의 노드의 총 수

합의 과정(Consensus Phase) 중, 스피커는 NextConsensus를 포함한 PrepareRequest 메시지를 전송합니다.
이 NextConsensus는 다음 블록에서 사용할 합의 노드 세트를 결정합니다.
스피커는 제안 블록에 포함된 트랜잭션과 블록체인 상의 기존 투표 데이터를 결합하여
다음 라운드의 합의 노드 목록을 계산하고,
2/3 다중 서명 컨트랙트의 스크립트 해시를 NextConsensus 필드에 할당합니다.

NeoVM

소개(Introduction)

NeoVM(Neo Virtual Machine)은 NEO 스마트 컨트랙트를 실행하기 위한 **경량 가상 머신(Lightweight Virtual Machine)**입니다. Neo의 핵심 구성 요소로서, NeoVM은 **튜링 완전성(Turing Completeness)**과 **높은 일관성(High Consistency)**을 가지며, 임의의 실행 로직을 구현할 수 있고, 분산 네트워크 상의 모든 노드에서 동일한 실행 결과를 보장합니다. 이를 통해 탈중앙화 애플리케이션(Decentralized Applications, DApps)에 강력한 기반을 제공합니다.

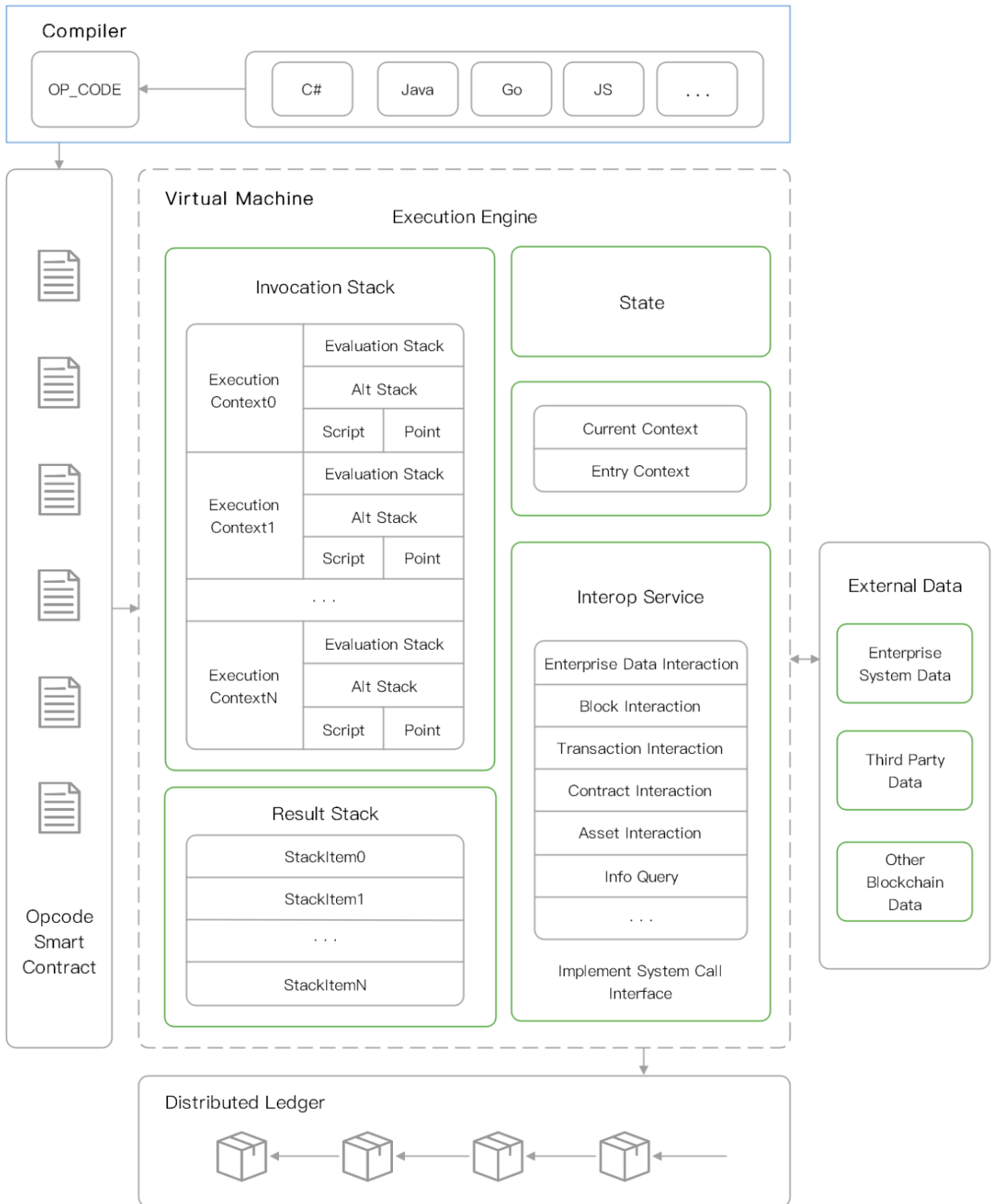
NeoCompiler를 활용하면 Java, C#, Python 등 다양한 고급 언어로 작성된 소스 코드를 통합된 NeoVM 명령어 집합(Instruction Set)으로 컴파일할 수 있습니다. 이로써 **크로스 플랫폼(Cross-Platform)** 실행이 가능해지고, 새로운 언어를 배우지 않아도 스마트 컨트랙트 개발에 쉽게 참여할 수 있습니다.

또한, NeoVM은 상위 코드(Upper Layer Code)와 **높이 분리(Decoupled)**되어 있으며, Interop Services와 같은 기술을 통해 쉽게 **사용자 정의(Customization)**가 가능합니다. NeoVM은 단순히 인스턴스 하나를 생성하는 것만으로도 사용 가능하며, 블록체인뿐만 아니라 **비블록체인 환경(Non-Blockchain Scenarios)**에서도 활용될 수 있습니다.

인프라 구조 및 실행 과정(Infrastructure and Execution Process)

인프라 구조(Infrastructure)

NeoVM의 아키텍처는 주로 **실행 엔진(Execution Engine)**, **스택(Stack)**, **상호운용 서비스 계층(Interoperation Service Layer)**으로 구성됩니다.



실행 엔진(ExecutionEngine)

ExecutionEngine은 NeoVM의 핵심(Core)으로, 스크립트를 로드하고 명령어를 실행하는 역할을 담당합니다.

이 엔진은 다음과 같은 작업을 수행합니다:

- 흐름 제어(Flow Control)
- 스택 연산(Stack Operation)
- 비트 연산(Bit Operation)

- 산술 연산(Arithmetic Operation)
- 논리 연산(Logical Operation)
- 암호화(Cryptography)

또한, 시스템 호출(System Call)을 통해

상호운용 서비스 계층(Interop Service Layer) 과 상호작용하며 외부 데이터를 참조할 수 있습니다.

스택(Stack)

NeoVM은 **스택 기반(Stack-Based)** 가상 머신입니다.

NeoVM에는 세 가지 주요 스택이 존재합니다:

InvocationStack, EvaluationStack, ResultStack.

- **InvocationStack**
현재 NeoVM의 모든 실행 컨텍스트(Execution Context)를 저장하며, 각 컨텍스트는 스택 내에서 서로 독립적으로 유지됩니다.
현재 컨텍스트(Current Context)는 스택의 최상단 요소(ExecutionContext0)를 가리키며, 진입 컨텍스트(Entry Context)는 스택의 마지막 요소(ExecutionContextN)를 가리킵니다.
- **EvaluationStack**
실행 중인 명령어가 사용하는 데이터를 저장합니다.
각 실행 컨텍스트는 자체적인 EvaluationStack을 가집니다.
- **ResultStack**
모든 스크립트가 실행된 후 결과(Result)를 저장합니다.

상호운용 서비스 계층(Interoperation Service Layer)

Interoperation Service Layer는

VM과 외부 데이터 사이를 연결하는 **브리지(Bridge)** 역할을 수행합니다.

상호운용 인터페이스(Interop Interfaces)를 호출함으로써, NeoVM은

- 블록 정보(Block Information)
- 트랜잭션 정보(Transaction Data)
- 컨트랙트 정보(Contract Data)
- 자산 정보(Asset Data) 등
스마트 컨트랙트 실행에 필요한 외부 데이터를 접근할 수 있습니다.

또한 각 스마트 컨트랙트는 **개인 저장 공간(Private Storage Area)** 을 선택적으로 활성화할 수 있습니다.

이 저장소는 키-값(Key-Value) 형식으로 데이터를 저장하며,

NeoVM은 실행 중 해당 저장소의 데이터를 동적으로 수정할 수 있습니다.

추가적으로, 상호운용 서비스 계층은

- 암호화 알고리즘(Cryptographic Algorithms)
- 영지식 증명(Zero-Knowledge Proof)
- 네트워크 리소스 접근(Network Access)
등을 제공하여, 개발자가 고급 애플리케이션을 구축할 수 있도록 지원합니다.

이 계층은 또한 **사용자 정의 확장(Custom Extension)** 과 **수정(Modification)** 을 지원하여,

개발자의 맞춤형 요구사항을 충족할 수 있습니다.

실행 과정(Execution Process)

Neo는 **다중 언어(Multi-language)** 스마트 컨트랙트 개발을 지원합니다.

NeoCompiler는 다양한 언어로 작성된 스마트 컨트랙트를 **통합된 NVM 바이트코드 파일(bytecode file)** 로 컴파일하며,

이후 **NeoVM**이 해당 바이트코드를 디코딩하고 실행합니다.

이로써 **다중 언어 컴파일러와 가상 머신(Virtual Machine)** 을 통한 **크로스 플랫폼 호환성(Cross-Platform Compatibility)** 이 달성됨

니다.

전체 실행 과정은 다음과 같습니다:

1. **스마트 컨트랙트 소스 코드**를 해당 언어의 컴파일러를 통해 **통합 바이트코드 파일(bytecode file)**로 컴파일합니다.
2. NeoVM의 **실행 엔진(Execution Engine)**이 바이트코드 파일을 로드하고, 파일 내 관련 매개변수(Parameter)와 함께 이를 **실행 컨텍스트(Execution Context)**로 구성한 후 **Invocation Stack**에 푸시(Push)합니다.
3. 실행 엔진은 현재 컨텍스트로부터 명령어(Instruction)를 하나씩 가져와 실행하며, 실행 과정에서 생성된 데이터는 **Evaluation Stack** 및 **Alt Stack**에 저장됩니다.
4. 외부 데이터 접근(External Data Access), 데이터 암호화(Encryption), 영지식 증명(Zero-Knowledge Proof) 등과 같은 연산이 필요할 경우, **상호운용 인터페이스(Interoperation Interfaces)**가 호출됩니다.
5. 모든 스크립트 실행이 완료되면, 결과(Result)는 **Result Stack**에 저장됩니다.

거래소(EXCHANGE)

개요(OVERVIEW)

소개(Introduction)

이 문서는 **거래소 개발자(Exchange Developers)**가 거래소 서버에 Neo 노드를 구축하고, Neo 자산의 입출금 및 트랜잭션 처리를 위한 필수 프로그래밍 작업을 수행하도록 안내합니다.

이 문서에는 다음 주제가 포함되어 있습니다:

- 서버에 Neo 노드 배포하기(#deploying-neo-nodes-on-server)
- Neo-CLI를 사용하여 지갑 및 입금 주소 생성하기(#using-neo-cli-to-create-wallets-and-deposit-addresses)
- 자산 트랜잭션 처리(#dealing-with-asset-transactions)
- 사용자에게 GAS 분배하기(#distributing-gas-to-users)

참고 (NOTE)

Neo-CLI의 버전에 따라 **거래소 인터페이스 방식이 다릅니다**.
클라이언트 버전을 업그레이드하기 전에 반드시
버전별 동작 차이를 확인해야 합니다.

서버에 Neo 노드 배포하기(Deploying Neo Nodes on Server)

Neo 클라이언트 설치(Install Neo Client)

1. 서버에 .NET Core Runtime(<https://dotnet.microsoft.com/en-us/download/dotnet/5.0>) 버전 **5.0 이상**을 설치합니다.
2. GitHub의 Neo-CLI 저장소(<https://github.com/neo-project/neo-node>)에서 **Neo-CLI 프로그램**을 다운로드하여 Neo 노드를 활성화합니다.

플러그인 설치(Install Plugins)

Neo는 보안(Security), 안정성(Stability), 확장성(Flexibility)을 강화하기 위해 일부 기능을 개별 플러그인(Plugin) 형태로 제공합니다.

자세한 내용은 Installing plugins(<https://github.com/neo-project/neo-modules>)를 참고하세요.

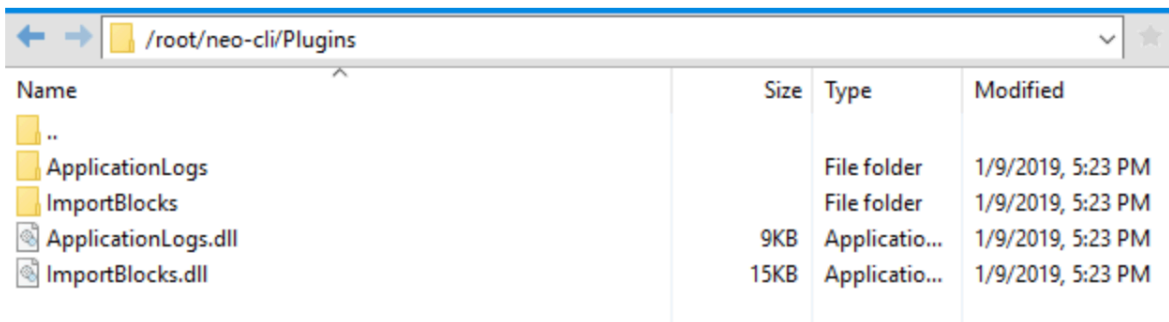
플러그인은 여기(<https://github.com/neo-project/neo-modules/releases>)에서 다운로드할 수 있습니다.

거래소는 다음 플러그인을 반드시 설치해야 **트랜잭션 로그 API** 및

오프라인 패키지 자동 동기화 기능을 완전히 사용할 수 있습니다:

- **ApplicationLogs** (필수)
- **LevelDBStore** (기본 포함)
- **RpcServer** (필수)
- **RpcNep17Tracker** (권장)

다운로드한 플러그인 압축 파일을 Neo-CLI 루트 디렉터리에 압축 해제하면, 모든 플러그인 파일이 자동으로 Plugins 폴더에 생성됩니다.



Name	Size	Type	Modified
..			
ApplicationLogs		File folder	1/9/2019, 5:23 PM
ImportBlocks		File folder	1/9/2019, 5:23 PM
ApplicationLogs.dll	9KB	Applicatio...	1/9/2019, 5:23 PM
ImportBlocks.dll	15KB	Applicatio...	1/9/2019, 5:23 PM

또는 CLI 명령을 사용하여 플러그인을 설치할 수도 있습니다.

```
install ApplicationLogs
install LevelDBStore
install RpcServer
install RpcNep17Tracker
```

Show all

Copy

참고 (NOTE)

- *RpcServer* 플러그인은 *RpcNep17Tracker* 플러그인을 사용하기 **전에 반드시 설치**해야 합니다.
- *ApplicationLogs* 및 *RpcNep17Tracker* 플러그인은 **Neo 클라이언트 동기화 전에 설치**해야 합니다.
그렇지 않으면, 동기화 이전 블록의 로그 데이터가 손실될 수 있습니다.

설정 파일 수정(Modifying Configuration Files)

Neo-CLI를 실행하기 전에,

Neo-CLI 설정 파일(Configuration Files)에서 필요한 구성을 완료해야 합니다.

자세한 내용은 [Configuring and Starting Neo-CLI](#)를 참고하세요.

위험

지갑의 자동 열기 기능을 활성화할 때는 주의하십시오. 이 기능을 사용하기 전에 방화벽이 열려 있고 안전한지 반드시 확인해야 합니다. 파일에 지정된 비밀번호는 평문(암호화되지 않은 상태)으로 포함되어 있기 때문입니다.

Neo-CLI 사용하기(Using Neo-CLI)

Neo-CLI 클라이언트는 P2P 네트워크 노드(Node)로 동작하며, 동시에 크로스 플랫폼 지갑(Cross-Platform Wallet)로서 다양한 자산 트랜잭션을 처리합니다.

Neo-CLI 보안 정책(Neo-CLI Security Policies)

DANGER

The exchange must use a white list or firewall to block external server requests; otherwise there will be a significant security risk.

Neo-CLI는 지갑을 원격으로 켜거나 끄는(Remote On/Off) 기능을 제공하지 않으며, 지갑을 열 때 프로세스 검증 절차도 수행하지 않습니다. 따라서 거래소는 자체적인 보안 정책(Security Policies)을 수립해야 합니다.

지갑은 항상 열려 있는 상태여야 하며, 사용자의 출금 요청(Withdrawal Requests)에 즉시 응답할 수 있어야 합니다. 보안 강화를 위해, 지갑은 방화벽이 적절히 설정된 독립 서버(Independent Server)에서 실행되어야 합니다.

다음 표는 메인넷(Mainnet) 및 테스트넷(Testnet) 환경에서 사용하는 기본 포트를 나타냅니다:

구분	Mainnet	Testnet
JSON-RPC via HTTPS	10331	20331
JSON-RPC via HTTP	10332	20332
P2P	10333	20333
WebSocket	10334	20334

Neo-CLI 소개(About Neo-CLI)

Neo-CLI는 개발자를 위한 명령줄(Command-Line) 기반의 클라이언트 지갑(Wallet)입니다. 개발자는 두 가지 방식으로 Neo-CLI와 상호작용할 수 있습니다:

1. CLI(Command-Line Interface) 명령어를 사용하는 방법

- 예: 지갑 생성(Create Wallet), 주소 생성(Generate Address) 등
2. **RPC(Remote Procedure Call)** 을 사용하는 방법
- 예: 지정된 주소로 송금(Transfer),
특정 블록 높이(Block Height)의 블록 정보 조회,
특정 트랜잭션 정보 조회 등

Neo-CLI는 다음 기능을 제공합니다:

- **지갑 기능 (Wallet Management)**

명령줄을 통해 자산을 관리할 수 있습니다.

Neo-CLI 디렉터리에서 아래 명령어를 입력하여 지갑을 실행할 수 있습니다:

```
dotnet neo-cli.dll
```

Show all

Copy

모든 사용 가능한 명령어를 확인하려면 다음 명령을 입력합니다:

```
help
```

Show all

Copy

자세한 명령어 목록은 [CLI Command Reference](#)를 참고하세요.

- **RPC API 기능 (RPC Interface)**

RpcServer 플러그인과 함께 사용하면,
노드로부터 블록체인 데이터를 가져올 수 있는 **RPC API 세트**를 제공합니다.
인터페이스는 [JSON-RPC](#)를 통해 제공되며,
하위 통신은 **HTTP/HTTPS 프로토콜**을 사용합니다.

RPC 서비스를 제공하는 노드를 시작하기 전에,
반드시 RpcServer 플러그인이 Neo-CLI에 설치되어 있어야 합니다.

API 목록 및 사용법은 [API Reference](#)를 참고하세요.

- **NEP-17 자산 트랜잭션 정보 제공**

NEP-17 표준 자산의 트랜잭션 내역을 조회할 수 있는 기능을 제공합니다.

지갑 생성(Creating a Wallet)

거래소는 사용자들의 **입금 주소(Deposit Address)** 를 관리하기 위해
온라인 지갑(Online Wallet) 을 생성해야 합니다.

지갑은 계정의 **공개 키(Public Key)** 와 **개인 키(Private Key)**,
그리고 **스마트 컨트랙트 정보(Contracts)** 를 저장하는 파일로,
사용자가 자산을 보유하고 있음을 증명하는 **가장 중요한 증거(Proof of Ownership)** 입니다.

사용자는 **지갑 파일(Wallet File)** 과 **지갑 비밀번호>Password)** 를
절대 분실하거나 외부에 노출해서는 안 됩니다.

거래소는 각 주소마다 별도의 지갑을 생성할 필요가 없습니다.
일반적으로, 하나의 온라인 지갑이 모든 사용자 입금 주소를 관리합니다.

보안 강화를 위해,
인터넷에 연결되지 않은 **콜드 월렛(Cold Wallet, 오프라인 지갑)** 을 사용하는 것도 좋은 선택입니다.
콜드 월렛은 해킹 위험을 최소화하며, 보다 높은 수준의 자산 보호를 제공합니다.

참고 (NOTE)

Neo-CLI는 두 가지 형식의 지갑을 지원합니다: *sqlite 지갑(.db3)* 과 새로운 NEP6 표준 지갑(.json) 입니다.
거래소의 경우 *sqlite 지갑* 사용을 권장합니다.

지갑을 생성하려면 다음 단계를 수행하십시오:

1. `create wallet <path>`를 입력합니다.
 - `path`는 지갑의 경로와 파일 이름입니다.
파일 확장자는 사용하는 지갑 유형에 따라 `.db3` 또는 `.json`을 선택할 수 있습니다.
 - 예시:

```
create wallet /home/mywallet.db3
```

파일 확장자를 지정하지 않으면 기본적으로 NEP6 형식(`.json`)이 사용됩니다.

2. 지갑의 비밀번호를 설정합니다.

입금 주소 생성(Generating Deposit Addresses)

하나의 지갑은 여러 개의 주소를 저장할 수 있습니다.
거래소는 각 사용자마다 별도의 입금 주소를 생성해야 합니다.

입금 주소를 생성하는 방법에는 두 가지가 있습니다:

- 사용자가 처음으로 NEO 또는 GAS를 입금할 때, 프로그램이 동적으로 NEO 주소를 생성합니다.
이 방법의 장점은 일정한 주기로 주소를 생성할 필요가 없다는 점이며,
단점은 백업이 불편하다는 것입니다.

주소를 동적으로 생성하는 프로그램을 개발하려면 `RpcServer API`의 `getnewaddress` 메서드를 사용하십시오.
생성된 주소가 반환됩니다.
- 거래소가 미리 여러 개의 NEO 주소를 일괄적으로 생성합니다.
사용자가 처음 NEO 또는 GAS를 입금할 때, 거래소는 그 중 하나의 NEO 주소를 사용자에게 할당합니다.
이 방법의 장점은 백업이 용이하다는 점이며,
단점은 수동으로 NEO 주소를 생성해야 한다는 것입니다.

주소를 일괄 생성하려면 NEO-CLI 명령어 `create address [n]`을 실행합니다.
생성된 주소는 자동으로 `address.txt` 파일에 내보내집니다.
[n]은 선택 항목이며 기본값은 1입니다.
예를 들어, 한 번에 100개의 주소를 생성하려면 다음과 같이 입력합니다:
`create address 100`

참고 (NOTE)

어떤 방법을 사용하든, 거래소는 생성된 주소를 데이터베이스에 가져와
사용자에게 할당해야 합니다.
일반적으로 두 번째 방법을 사용하는 것이 권장됩니다.

이렇게 하면 외부 제어를 줄이고 지갑을 보다 안정적으로 운영할 수 있습니다.

사용자에게 GAS 분배(Distributing GAS to Users)

거래소는 사용자에게 GAS를 분배할지 여부를 결정할 수 있습니다.

GAS는 Neo 블록체인에서 트랜잭션 기록 및 추가 서비스를 위한 수수료로 사용됩니다.

GAS란 무엇인가?(What is GAS?)

GAS는 Neo 블록체인의 **사용 권리(Right to Use)** 를 나타냅니다.

NEO를 보유하면 시스템 내에서 알고리즘에 따라 GAS가 생성됩니다.

Neo N3에서는 주소에서 NEO 전송 트랜잭션이 발생할 때마다,
마지막 NEO 전송 트랜잭션 이후 생성된 GAS가 자동으로 청구됩니다.

청구된 GAS 양 계산(Calculating the Claimed GAS Amount)

청구된 GAS는 다음과 같이 계산됩니다:

$$\text{Claimed GAS} = f(\text{neoamount}, \Delta t_{\text{const}})$$

$$\Delta t_{\text{const}} = t_{\text{end}} - t_{\text{start}}$$

- t_{end} = NEO가 주소로 입금되거나 출금되는 현재 시점
- t_{start} = NEO가 마지막으로 입금되거나 출금된 시점

Δt_{const} 는 고정된 값이므로, 청구된 GAS의 양 또한 고정됩니다.

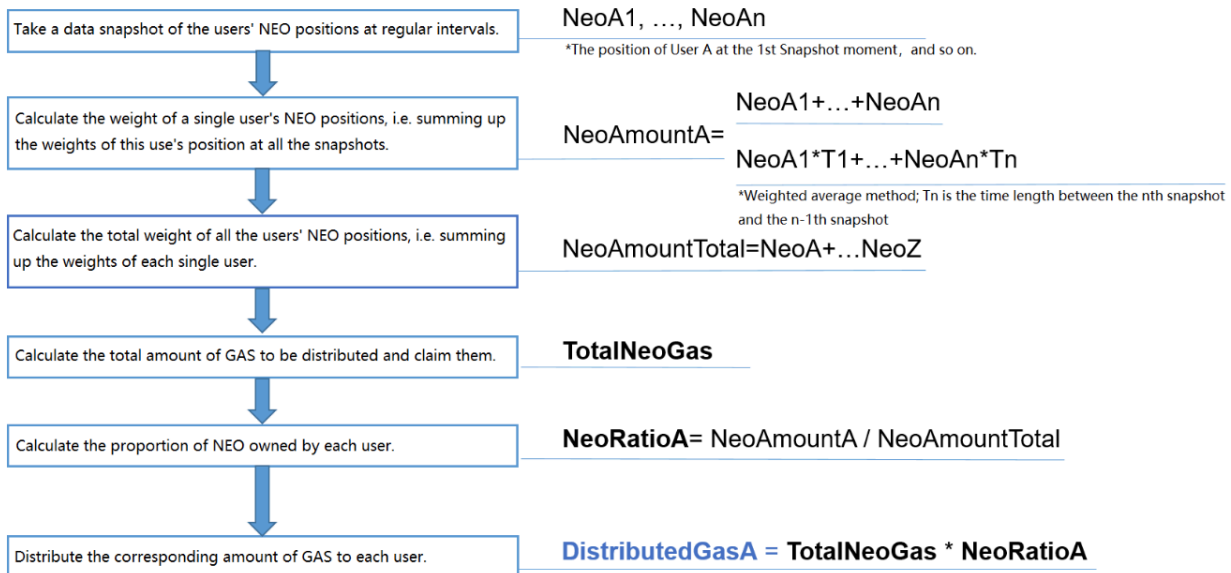
이 양은 사용자가 보유한 NEO의 양과

그 NEO가 주소에 입금되어 있던 시간 간격에 따라 달라집니다.

거래소의 GAS 분배 절차(Distributing GAS to Users)

거래소의 모든 주소가 하나의 지갑에 저장되어 있다고 가정하면,

다음 그림은 거래소가 사용자 A에게 GAS를 분배하는 절차와 계산 공식을 보여줍니다.



스냅샷 간격이 짧을수록 계산이 더 정밀해집니다.

스냅샷 간격이 일정하지 않은 경우, **가중 평균(Weighted Average)** 계산 방법을 사용하십시오.

참고 (NOTE)

Neo N3에서는 거래소 사용자가 직접 투표에 참여할 수 없으므로, 고정 수익은 청구 가능한 GAS 총량의 **10%**로 설정되어 있습니다. 자세한 내용은 [GAS 분배 규칙\(GAS distribution rule\)](#)을 참고하십시오.

RPC 메서드(RPC Methods)

다음 RPC 메서드를 사용하여 거래소가 사용자의 GAS 정보를 조회할 수 있습니다.

아래 표에서 원하는 메서드 이름을 클릭하면 자세한 내용을 확인할 수 있습니다.

메서드(Method)	설명(Description)
getunclaimedgas	현재 지갑에 청구되지 않은 GAS의 총량을 반환합니다.

GAS 청구(Claiming GAS)

GAS는 사용자가 **NEO를 입금하거나 출금할 때 자동으로 청구(Claim)** 됩니다.

예를 들어, 주소 A에 NEO를 보유하고 있으면서 아직 GAS를 청구하지 않은 상태에서, 자신의 주소(A)로 NEO를 다시 전송하면, GAS가 자동으로 청구됩니다.

아래 표는 GAS를 청구하는 절차와 이에 대응하는 명령어를 보여줍니다.

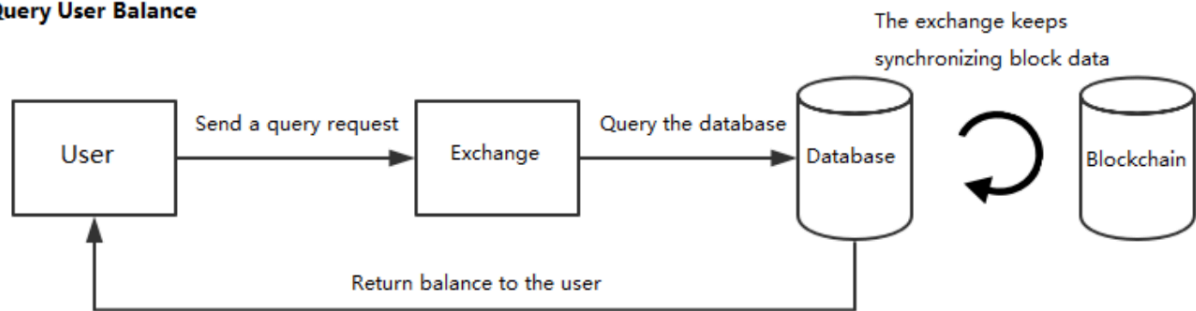
#	단계(Steps)	명령어(Command)
1	Neo-CLI 실행	dotnet neo-cli.dll
2	클라이언트 버전 확인	version
3	클라이언트의 동기화 높이 확인 (Height: height/header height, Nodes: 연결된 노드 수)	show state
4	지갑 열기	open wallet /home/NeoNode/test.json
5	지갑 내 주소 및 자산 확인	list asset
6	지갑 내 GAS 잔액 확인	show gas
7	NEO를 자신의 주소(예: NTdzVdQ8SmFobD1XWCA1fR9tQr7gFv1SXf)로 전송하여 자동으로 GAS를 청구	send neo NTdzVdQ8SmFobD1XWCA1fR9tQr7gFv1SXf 1
8	다시 자산 잔액 확인	list asset

자산 트랜잭션 처리(Dealing with Asset Transactions)

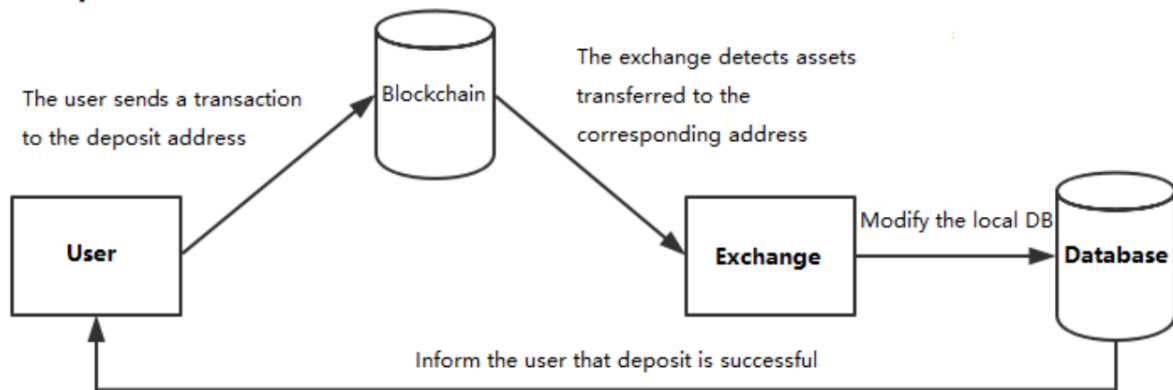
개요(Overview)

Neo N3에는 단 하나의 디지털 자산 유형인 **NEP-17 자산(NEP-17 Assets)** 만 존재합니다. 이 자산들은 **BALANCE** 구조로 관리되며, 거래소는 주로 **사용자 잔액 조회**, **입금(Deposit)**, **출금(Withdrawal)** 등의 작업을 처리합니다. 아래 흐름도는 이러한 작업의 전체 프로세스를 보여줍니다:

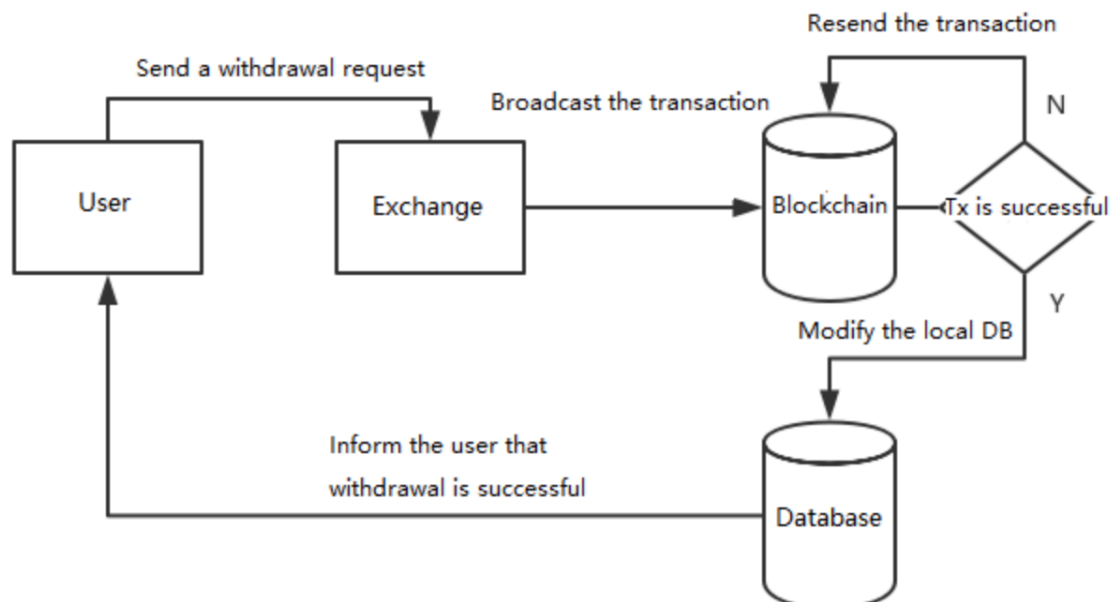
Query User Balance



User deposit



User Withdrawal



네트워크 수수료(Network Fee)

네트워크 수수료(Network Fee) 는

블록을 생성하는 **합의 노드(Consensus Nodes)** 에 대한 보상으로 부과됩니다.

사용자가 Neo 블록체인에 트랜잭션을 제출할 때마다 네트워크 수수료가 발생합니다.
각 트랜잭션에는 기본 수수료(Base Fee)가 존재하며,
그 계산식은 아래와 같습니다.

$$\text{NetworkFee} = \text{VerificationCost} + \text{tx.size} * \text{FeePerByte}$$

[Show all](#)[Copy](#)

사용자가 지불한 수수료가 기본 수수료보다 크거나 같을 경우에만
해당 트랜잭션이 실행됩니다.
그보다 낮으면 트랜잭션은 **무효(Invalid)** 로 처리됩니다.

- **VerificationCost:** NeoVM이 트랜잭션 서명을 검증하기 위해 실행하는 명령어의 수수료입니다.
- **tx.size:** 트랜잭션 데이터의 바이트 길이입니다.
- **FeePerByte:** 바이트당 트랜잭션 수수료로, 현재 PolicyContract에서 **0.00001 GAS**로 정의되어 있습니다.

시스템 수수료(System Fee)

시스템 수수료(System Fee)는 NeoVM이 실행하는 **명령어(Instructions)** 에 대해 부과됩니다.
각 명령어별 수수료는 [System Fee](#)를 참조하십시오.

지불해야 하는 총 시스템 수수료는
스마트 컨트랙트가 실행하는 명령어의 **개수(Number)** 와 **유형(Type)** 에 따라 달라집니다.

아래 그림은 시스템 수수료의 계산 공식을 보여줍니다:

$$\text{SystemFee} = \text{InvocationCost} = \text{The sum of all executed opcode fee}$$

[Show all](#)[Copy](#)

명령어 수수료(Instructions Fee)

Neo N3에서는 NeoVM의 명령어 수수료가
Neo Legacy 대비 1/1000 수준으로 감소하였습니다.
이로써 개발 비용이 크게 절감됩니다.

Neo Legacy와의 비교는 다음 표와 같습니다:

Case	Cost of Neo2.x	Cost of NEO3
Deploy contract	400 (basic fee) +100 (storage fee) + 500 (dynamic invoke fee)	contract.size * 0.001
Invoke contract	InvocationCost	InvocationCost
Systemfee	Different transaction types for different system fees, like IssueTransaction about 500 GAS.	InvocationCost
Networkfee	0.001, if tx size <= 1024 bytes / > (tx.size - 1024) * 0.00001 + 0.001, if tx size > 1024	VerificationCost + tx.size * 0.00001
Opcode fee		Reduced to about 1/1000

트랜잭션 조회 처리(Dealing with Query Transactions)

거래소가 사용자의 입금 주소 잔액을 직접 조회하는 방법은, 사용자가 자신의 잔액을 조회하는 요청을 처리하는 방식과 다릅니다.

사용자 입금 주소 잔액 조회(Querying the User Deposit Address Balance)

거래소는 다음 절차를 수행해야 합니다:

1. JSON 파일을 구성하여 아래의 RPC 메서드 중 하나를 호출합니다:
 - `getnep17balances` (플러그인 `TokensTracker` 필요)
 - `invokefunction` (플러그인 `RpcServer` 필요)
2. Neo RPC 서버에 `getnep17balances` 요청을 전송하여 자산의 **해시(Asset Hash)**와 **보유량(Amount)**을 가져옵니다.
3. Neo RPC 서버에 `invokefunction` 요청을 두 번 전송하여 각각 해당 자산의 **심볼(Symbol)**과 **소수점 자릿수(Decimals)**를 조회합니다.
4. 반환된 값을 기반으로 사용자의 잔액(Balance)을 계산합니다.

특정 사용자의 자산 잔액을 조회하려면, `invokefunction`을 사용하여 해당 자산의 `balanceOf` 메서드를 호출하십시오.

`getnep17balances` 호출을 통한 조회(Invoking `getnep17balances` to Query)

JSON 형식에서 일반적인 `getnep17balances` 요청 본문은 다음과 같습니다:

```
{
  "jsonrpc": "2.0",
  "method": "getnep17balances",
  "params": ["NVfJmhP28Q9qva9Tdtpt3af4H1a3cp7Lih"],
  "id": 1
}
```

요청을 전송하면 다음과 같은 응답이 반환됩니다:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "balance": [
      {
        "asset_hash": "0xef4073a0f2b305a38ec4050e4d3d28bc40ea63f5",
        "amount": "2",
        "last_updated_block": 52675
      },
      {
        "asset_hash": "0xd2a4cff31913016155e38e474a2c06d08be276cf",
        "amount": "700000000",
        "last_updated_block": 52675
      }
    ],
    "address": "NVfJmhP28Q9qva9Tdtpt3af4H1a3cp7Lih"
  }
}
```

위 요청 예시에서 볼 수 있듯이, 두 종류의 자산 해시가 존재합니다:

"0xef4073a0f2b305a38ec4050e4d3d28bc40ea63f5" 와
"0xd2a4cff31913016155e38e474a2c06d08be276cf" 입니다.

이제 각 자산의 **symbol(심볼)** 과 **decimals(소수점 자리수)** 를 조회하기 위해
invokefunction 메서드를 호출해야 합니다.
해당 절차는 아래에서 자세히 설명됩니다.

예를 들어 위의 응답에서,

- 자산 A의 심볼(Symbol)은 **NEO**, 소수점 자릿수(Decimals)는 **0**, 사용자 A의 잔액은 **2 NEO**입니다.
- 자산 B의 심볼(Symbol)은 **GAS**, 소수점 자릿수(Decimals)는 **8**, 잔액은 **700000000/10⁸ GAS (7 GAS)** 입니다.

invokefunction 호출을 통한 조회(Invoking invokefunction to Query)

JSON 형식에서 일반적인 invokefunction 요청 본문은 다음과 같습니다:

```
{
  "jsonrpc": "2.0",
  "method": "invokefunction",
  "params": [
    "script hash",
```

```

    "method name",
    [
      {
        "optional arguments"
      }
    ],
    "id": 1
  }

```

사용자의 잔액을 조회할 때는 다음 문자열들을 교체해야 합니다:

- script hash
조회하려는 NEP-17 자산의 스크립트 해시입니다. 예를 들어:
- NEO는 `oxef4073aof2b305a38ec4050e4d3d28bc40ea63f5`
- GAS는 `oxd2a4cff31913016155e38e474a2c06do8be276cf`
- method name*
호출하려는 메서드의 이름입니다. 사용자의 잔액을 조회하려면 다음 세 가지 메서드를 호출해야 합니다:

balanceOf

- 문법:
`public static BigInteger balanceOf(byte[] account)`
- 설명:
"balanceOf"는 "account"의 토큰 잔액을 반환합니다.
- 문법:
`public override byte Decimals { [Safe] get => 8; }`
- 설명:
"decimals"는 해당 토큰이 사용하는 소수점 자릿수를 반환합니다.
- 문법:
`public override string Symbol { [Safe] get => "EXAMPLE"; }`
- 설명:
"symbol"은 토큰의 심볼을 반환합니다.

decimals

symbol

- optional arguments
선택 사항입니다. 호출하려는 메서드에 인자가 필요한 경우, 해당 파라미터들을 배열 형태로 구성하여 전달할 수 있습니다.
예를 들어, NEP-17의 "balanceOf"는 "account"의 토큰 잔액을 반환하므로:
`public static BigInteger balanceOf(byte[] account)`
따라서 "balanceOf" 메서드에 account를 인자로 전달해야 합니다.

예시

balanceOf 호출

계정 주소가 NYxb4fSZVKAz8YsgaPK2WKtT3KcAE9b3Vag라고 가정하면, 이를 Hash160 타입으로 변환하고 이 파라미터를 JSON 객체로 구성해야 합니다:

```

{
  "type": "Hash160",
  "value": "0x762f8a2bf0e8673c64cc608ba69b9c2a946a188f"
}

```

Show all

Copy

그런 다음 다음과 같이 JSON 메시지를 구성할 수 있습니다:

요청 본문:

```
{
  "jsonrpc": "2.0",
  "method": "invokefunction",
  "params": [
    "0xd2a4cffe31913016155e38e474a2c06d08be276cf",
    "balanceOf",
    [
      {
        "type": "Hash160",
        "value": "0x762f8a2bf0e8673c64cc608ba69b9c2a946a188f"
      }
    ]
  ],
  "id": 3
}
```

요청을 전송한 후, 다음과 같은 응답을 받게 됩니다:

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "result": {
    "script":
      "DBSPGqUKpybpotgzGQ8Z+jwK4ovdhHAHwwJYmFsYW5jZU9mDBTPduKL0AYsSke041VhARMZ88+k0kFifVtS",
    "state": "HALT",
    "gasconsumed": "2028330",
    "exception": null,
    "stack": [
      {
        "type": "Integer",
        "value": "1938845793634190"
      }
    ]
  }
}
```

잔액을 얻기 위해서는 반환된 값을 decimals로 나누면 되며, 데이터 변환은 필요하지 않습니다.

decimals 호출

요청 본문:

```
{
  "jsonrpc": "2.0",
  "method": "invokefunction",
  "params": [
    "0xd2a4cffe31913016155e38e474a2c06d08be276cf",
    "decimals",
    []
  ],
  "id": 2
}
```

요청을 전송한 후, 다음과 같은 응답을 받게 됩니다:

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "result": {
    "script": "wh8MCGRLY2ltYWxzDBTPduKL0AYsSke041VhARMZ88+k0kFifVtS",
    "state": "HALT",
    "gasconsumed": "984060",
    "exception": null,
    "stack": [
      {
        "type": "Integer",
        "value": "8"
      }
    ]
  }
}
```

정수 8이 반환됩니다.

symbol 호출

요청 본문:

```
{
  "jsonrpc": "2.0",
  "method": "invokefunction",
  "params": [
    "0xd2a4cff31913016155e38e474a2c06d08be276cf",
    "symbol",
    []
  ],
  "id": 3
}
```

요청을 전송한 후, 다음과 같은 응답을 받게 됩니다:

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "result": {
    "script": "wh8MBnN5bwJvbAwUz3bii9AGLEpHjuNVYQETGfPPpNJBYn1bUg==",
    "state": "HALT",
    "gasconsumed": "984060",
    "exception": null,
    "stack": [
      {
        "type": "ByteString",
        "value": "R0FT"
      }
    ]
  }
}
```

"R0FT"가 반환되며, 이는 "GAS"로 디코딩할 수 있습니다.

사용자 잔액 계산

모든 반환 값을 바탕으로, 사용자 잔액은 다음과 같이 계산할 수 있습니다:

잔액 = 반환값/\$10^{\text{decimals}}

사용자 질의 처리

거래소 내의 실제 사용자 잔액은 거래소 데이터베이스에 기록됩니다. 거래소는 각 블록의 각 트랜잭션을 모니터링하고, 모든 입금 및 출금 트랜잭션을 데이터베이스에 기록하며, 이에 따라 사용자 잔액을 데이터베이스에서 수정하는 프로그램을 작성해야 합니다.

사용자 입금 처리

사용자 입금 정보를 얻기 위해 거래소는 다음과 같은 작업을 수행해야 합니다:

1. `getblock` API를 사용하여 각 블록의 세부 정보를 가져옵니다. 여기에는 해당 블록에 포함된 모든 트랜잭션의 세부 정보가 포함됩니다.
2. `getapplicationlog` API를 호출하여 각 "InvocationTransaction" 트랜잭션의 세부 정보를 얻고, 트랜잭션 내용을 분석하여 사용자 입금을 처리합니다.

getapplicationlog 호출

이 API는 트랜잭션 정보를 얻는 데 사용됩니다.

ApplicationLogs 플러그인을 올바르게 설치하고 `neo-cli` 노드를 시작한 후, 루트 경로 하단에 "ApplicationLogs"라는 폴더가 생성되는 것을 확인할 수 있습니다. 이 디렉토리에는 전체 컨트랙트 로그가 기록되며, 각 NEP-17 트랜잭션은 `leveldb` 파일로 저장됩니다.

다음은 API 호출 결과의 예시입니다.

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "txid": "0xd9aaa1243cae91e063a140239807a9de45f82850130ec36403f44770955dd2d7",
    "trigger": "Application",
    "vmstate": "HALT",
    "gasconsumed": "11819770",
    "stack": [],
    "notifications": [
      {
        "contract": "0xd2c270ebfc2a1cdd3e470014a4dff7c091f699ec",
        "eventname": "Transfer",
        "state": {
          "type": "Array",
          "value": [
            {
              "type": "ByteString",
              "value": "uXtKzX+CD2HS1NT5rqXrUEmN31U="
            },
            {
              "type": "ByteString",
              "value": "7ztGBn8vR7L38EQqojcghdCHC08="
            },
            {
              "type": "Integer",
              "value": "800000000000"
            }
          ]
        }
      }
    ]
  }
}
```

```
}  
  ]  
}  
}
```

참고

이 예시는 토큰 전송에 성공한 로그를 보여줍니다. 그러나 전송 실패나 NeoVM 예외 발생 시, 결과는 다음과 같을 수 있습니다.

- **전송 실패:** Transfer 알림 이벤트가 반환되지 않으며, 실행은 스택 값이 `False` 인 `HALT` 상태로 종료됩니다.
- **NeoVM 예외:** Transfer 알림 이벤트가 반환될 수도 있고 아닐 수도 있으며, 실행은 `FAULT` 상태로 종료됩니다.

파일 내 트랜잭션과 관련된 파라미터는 다음과 같습니다:

- **contract:** 스마트 컨트랙트의 스크립트 해시입니다. 거래소의 경우 NEP17 자산 유형의 스크립트 해시이며, 자산의 고유 식별자 역할을 합니다. 예를 들어 "0xd2c270ebfc2a1cdd3e470014a4dff7c091f699ec"는 NEP17 자산의 스크립트 해시입니다.
- **eventname:** 스마트 컨트랙트 이벤트 식별자입니다. 거래소는 사용자의 전송 트랜잭션을 확인하기 위해 "transfer" 트랜잭션만 수신하면 됩니다. Notifications 배열에는 여러 개의 eventname이 있을 수 있으며, Transfer 키워드를 가진 항목만이 NEP17 Transfer 데이터입니다.
- **state:** 배열에 포함된 객체들은 다음과 같습니다:
 - **from account:** 배열의 첫 번째 객체는 자산이 전송된 계정 주소입니다. 타입은 "bytearray"이며 값 "uXtkXz+CD2HS1NT5rqXrUEmN31U="는 base64로 디코딩하면 "NcphjtgTye3c3ZL5J5nDZhsf3JJMGAjd7o"가 됩니다.
 - **to account:** 배열의 두 번째 객체는 자산이 전송된 계정 주소입니다. 타입은 "bytearray"이며 값 "7ZtGBn8v7RL38EQqojcgHdCHC08="는 base64로 디코딩하면 "Nh1uh11SHF4n9FE6G5LuFHHYc7Lgws9U1z"가 됩니다. 해당 주소가 거래소의 계정 주소인 경우, 이는 입금 트랜잭션입니다.
 - **amount:** 배열의 마지막 객체는 전송된 금액입니다. 값은 8000000000000이며, 소수점 자릿수가 8이므로 실제 금액은 8000.00000000입니다.

참고

Neo에서는 0x로 시작하는 16진수 문자열은 빅엔디언(big-endian) 순서로 처리되고, 그렇지 않은 경우에는 리틀엔디언(little-endian) 순서로 처리됩니다.

Show all Copy

```
{
  "type": "ByteString",
  "value": "uXtKzX+CD2HS1NT5rqXrUEmN31U="
}
```

Show all Copy

```
{
  "type": "ByteString",
  "value": "7ztGBn8vR7L38EQqojcghdCHC08="
}
```

Show all Copy

```
{
  "type": "Integer",
  "value": "8000000000000"
}
```

참고

파일 내 전송 데이터의 형식 변환에 대해서는 [Neo3 데이터 변환](#)을 참고하십시오.

사용자 출금 처리

거래소는 사용자에게 자산을 전송하기 위해 다음 중 하나의 방법을 선택할 수 있습니다:

- Neo-CLI 명령어: send
- RPC 메서드: sendfrom
- RPC 메서드: sendtoaddress
- RPC 메서드: sendmany

Neo-CLI 명령어: send

문법

```
send <id|alias> <address> <amount>|all [from=null] [signerAccounts=null]
```

파라미터

- id|alias : 자산 ID 또는 자산 약어 (예: neo, gas)
- address : 자산을 전송할 대상 주소
- amount|all : 전송할 금액
- from : 자산을 전송할 발신 주소
- signerAccounts : 서명자의 주소

이 명령은 지갑 비밀번호를 검증합니다.

예시

주소 NYxb4fSZVKAz8YsgaPK2WkT3KcAE9b3Vag로 100 Neo를 전송:

```
neo> send a1760976db5fcdfab2a9930e8f6ce875b2d18225 NYxb4fSZVKAz8YsgaPK2WkT3KcAE9b3Vag 100
password: *****
TXID: 0x8f831d8de723093316c05749a053a226514bc06338b2bceb50db690610e0b92f
```

자산 ID가 확실하지 않은 경우 list asset을 입력하여 지갑 내 모든 자산을 확인할 수 있습니다.

위 예시에서는 자산 ID 대신 자산 약어를 사용할 수도 있으며, 다음과 같이 입력합니다:

```
neo> send gas NYxb4fSZVKAz8YsgaPK2WkT3KcAE9b3Vag 100
password: *****
TXID: 0xae0675797c2d738dcadb21cec3f1809ff453ac291046a05ac679cbd95b79c856
```

RPC 메서드: openwallet

참고

지갑 관련 RPC 메서드를 호출하기 전에 반드시 openwallet 메서드를 먼저 호출해야 합니다.

키 "params"는 두 개의 파라미터로 구성된 배열을 포함합니다.

"params": [path, password]

예를 들어, a.json이라는 지갑을 비밀번호 111111로 열기 위해서는 아래와 같이 JSON 파일을 구성하여 RPC 서버로 전송할 수 있습니다.

요청 본문:

```
{
  "jsonrpc": "2.0",
  "method": "openwallet",
  "params": ["a.json", "111111"],
  "id": 1
}
```

Show all

Copy

요청을 전송한 후, 다음과 같은 응답을 받게 됩니다:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": true
}
```

Show all

Copy

RPC 메서드: sendfrom

키 "params"는 네 개의 파라미터로 구성된 배열을 포함합니다.

"params": [script hash, address from, address to, amount]

예를 들어, NcphtjgTye3c3ZL5J5nDZhsf3JJMGAjd7o 주소에서

Nhiuh11SHF4n9FE6G5LuFHHYc7Lgws9U1z 주소로 10 NEO를 전송하려면, 아래와 같이 JSON 파일을 구성하여 RPC 서버로 전송합니다.

요청 본문:

```
{
  "jsonrpc": "2.0",
  "method": "sendfrom",
  "params":
    ["0xf61eebf573ea36593fd43aa150c055ad7906ab83", "NcphtjgTye3c3ZL5J5nDZhsf3UJMGAjd7o", "Nhiuh11SHF4n9FE6G5LuFHHYc7Lgws9U1z", 10],
  "id": 1
}
```

요청을 전송한 후, 다음과 같은 응답을 받게 됩니다:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "hash": "0x2dad82755c3b3e3233c10a49402bea9b8bb3f43b079102bbc3c5a50c3b522137",
    "size": 264,
    "version": 0,
    "nonce": 1073258915,
    "sender": "NcphtjgTye3c3ZL5J5nDZhsf3UJMGAjd7o",
    "sysfee": "9007990",
    "netfee": "1264390",
    "validuntilblock": 2107189,
    "attributes": [
      {
        "type": "Cosigner",
        "account": "0x55df8d4950eba5aef9d4d4d2610f827fcd4a7bb9",
        "scopes": "CalledByEntry"
      }
    ],
    "script":
      "GgwU7ztGBn8vR7L38EQqojcghdCHC08MFL17Ss1/gg9h0tTU+a6l61BJjd9VE8AMCHRYYW5zMmVyDBQlBZ7LSHjTqHX5H
      FH03tMw1Fdf3kFifVt50A==",
    "witnesses": [
      {
        "invocation":
          "DEBL7Fxz2ZyIgtz+kESSs8YjbJd5dcc13gpx0wrLjU+WiIa0fuFQSGHXM75S1Z21wDMvEirUHpU1rIYylfnQH6U1",
        "verification": "DCECTLb+CYh0tAkrQbRliAmdLaB5NLR0FqIWxgiCPlnz/B4LQZVEDXg="
      }
    ]
  }
}
```

RPC 메서드: sendtoaddress

키 "params"는 세 개의 파라미터로 구성된 배열을 포함합니다.

"params": [script hash, address, amount]

예를 들어, Nhiuh11SHF4n9FE6G5LuFHHYc7Lgws9U1z 주소로 1000 GAS를 전송하려면, 아래와 같이 JSON 파일을 구성하여 RPC 서버로 전송합니다.

요청 본문:

```
{
  "jsonrpc": "2.0",
  "method": "sendtoaddress",
  "params": ["0x70e2301955bf1e74cbb31d18c2f96972abadb328", "Nhiuh11SHF4n9FE6G5LuFHHYc7Lgws9U1z",
    1000],
  "id": 1
}
```

요청을 전송한 후, 다음과 같은 응답을 받게 됩니다:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "hash": "0xda4de7d6fc3bcd0eba51a3dcba01eaba7d59467acf91525c5f3f0b56df06aec8",
    "size": 272,
    "version": 0,
    "nonce": 1325103139,
    "sender": "NcphtjgTye3c3ZL5J5nDZhsf3UJMGajd7o",
    "sysfee": "9007990",
    "netfee": "1272390",
    "validuntilblock": 2107253,
    "attributes": [
      {
        "type": "Cosigner",
        "account": "0x55df8d4950eba5aef9d4d4d2610f827fcd4a7bb9",
        "scopes": "CalledByEntry"
      }
    ],
    "script":
    "AwDodkgXAAAADBTv00YGfy9HsvfwRCqiNyCF0IcI7wwUuXtKzX+CD2HS1NT5rqXrUEmN31UTwAwIdHJhbnNmZXIMFLyvQ
    daEx9StbuDZnalwe50fDI5mQWJ9W1I4",
    "witnesses": [
      {
        "invocation":
        "DEBd+BDi7LWMQ5zzWxmzvH9zs09fRZpdqn9SqnyEfSzazVnFsUldJG7ik79epcqpF+IWGQJM1lS1oDeIEh/Yq04",
        "verification": "DCECTLb+CYh0tAkrQbRliAmdLaB5NLR0FqIWxgiCPlnz/B4LQZVEDXg="
      }
    ]
  }
}
```

RPC 메서드: sendmany

키 "params"는 최소 두 개의 파라미터로 구성된 배열을 포함합니다:

"params": [address from(optional), []]

예를 들어, NcphtjgTye3c3ZL5J5nDZhsf3JJMGAjd7o 주소에서
Nhiuh11SHF4n9FE6G5LuFHHYc7Lgws9U1z 주소로
100 NEO와 1000 GAS를 전송하려면, 아래와 같이 JSON 파일을 구성하여 RPC 서버로 전송합니다.

요청 본문:

```
{
  "jsonrpc": "2.0",
  "method": "sendmany",
  "params": [
    "NcphtjgTye3c3ZL5J5nDZhsf3JJMGAjd7o",
    [
      {
        "asset": "0xf61eebf573ea36593fd43aa150c055ad7906ab83",
        "value": 100,
        "address": "Nhiuh11SHF4n9FE6G5LuFHHYc7Lgws9U1z"
      },
      {
        "asset": "0x70e2301955bf1e74cbb31d18c2f96972abadb328",
        "value": 1000,
        "address": "Nhiuh11SHF4n9FE6G5LuFHHYc7Lgws9U1z"
      }
    ]
  ],
  "id": 1
}
```

요청을 전송한 후, 다음과 같은 응답을 받게 됩니다:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "hash": "0xea4564840441713481363ffc0b3e2df95e5319af4d5da4189603c2333d6702f5",
    "size": 358,
    "version": 0,
    "nonce": 93745276,
    "sender": "NcphtjgTye3c3ZL5J5nDZhsf3JJMGAjd7o",
    "sysfee": "18015980",
    "netfee": "1358390",
    "validuntilblock": 2107284,
    "attributes": [
      {
        "type": "Cosigner",
        "account": "0x55df8d4950eba5aef9d4d4d2610f827fcd4a7bb9",
        "scopes": "CalledByEntry"
      }
    ],
    "script":
    "AGQMf087RgZ/L0ey9/BEKqI3IIXQhwjvDBS5e0rNf4IPYdLU1PmupetQSY3fVRPADAh0cmFuc2ZlcgwUJQWey0h406h1+
    RxRzt7TMNRXX95BYn1bUjgDA0h2SBcAAAAMF087RgZ/L0ey9/BEKqI3IIXQhwjvDBS5e0rNf4IPYdLU1PmupetQSY3fVRP
    ADAh0cmFuc2ZlcgwUvK9B1oTH1K1u4NmdqXB7nR8MjmZBYn1bUjg=",
    "witnesses": [
      {
        "invocation":
        "DEA1J31Wq9CS6s7Zyzv71jS/LXbJroKgZMhTk176KaCNDBIas5kqBgsv0hHVxetxdwnapXU7Cui/9PlHr3fZNPf3",
        "verification": "DCECTLb+CYh0tAkrQbRliAmdLaB5NLR0FqIWxgiCPlnz/B4LQZVEDXg="
      }
    ]
  }
}
```

```
}  
}  
}
```

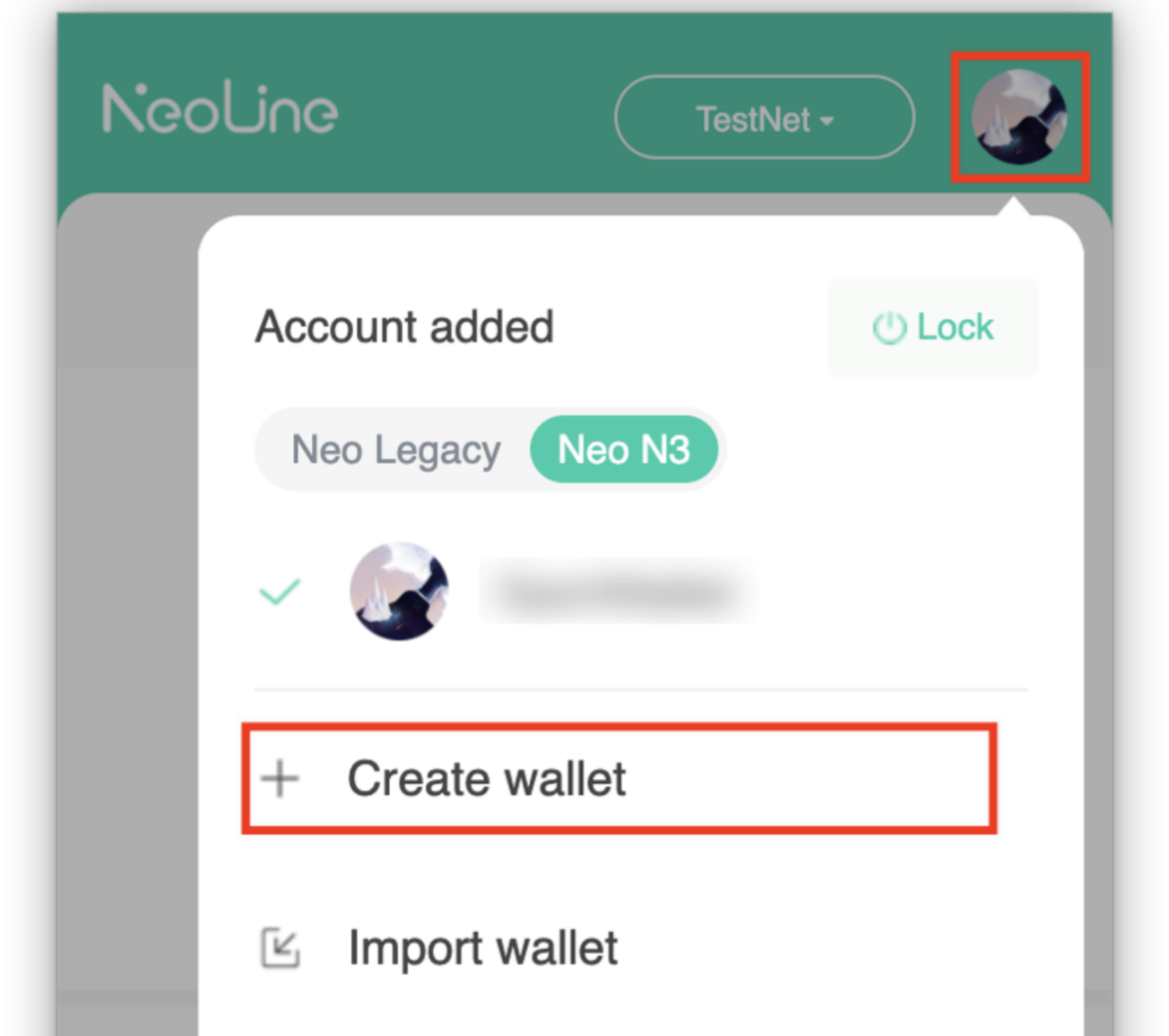
크로스체인 마이그레이션

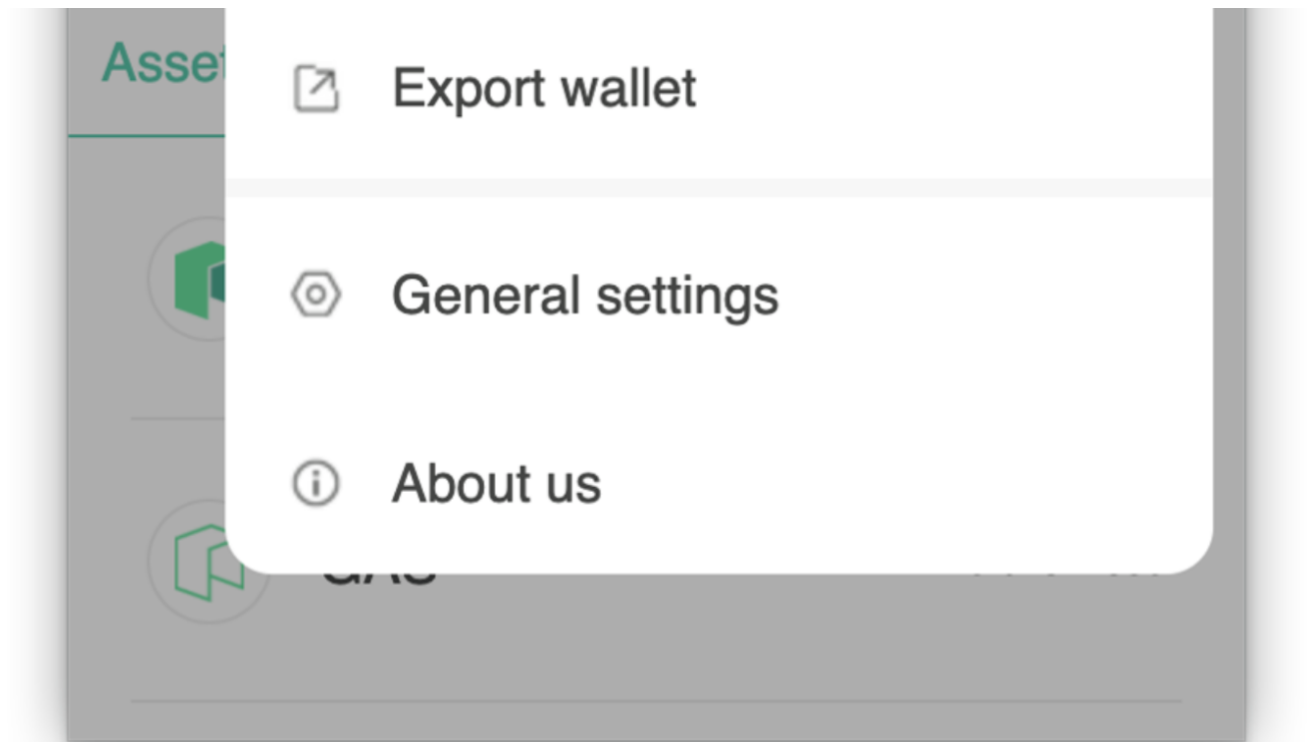
마이그레이션 페이지에서의 Neo N3 마이그레이션 가이드라인

이 튜토리얼은 Neo 호환 지갑을 사용하여 Neo 레거시에서 Neo N3로 Neo 및 GAS 토큰을 마이그레이션하는 과정을 마이그레이션 웹사이트에서 단계별로 안내합니다.

NeoLine 지갑을 이용한 마이그레이션

1. 오른쪽 상단의 아이콘을 클릭한 후 지갑 생성(Create wallet)을 클릭합니다.





2. Neo N3를 선택하여 Neo N3 지갑을 생성합니다.



SamWallet

NSKKL5EAftBwjnPMTZAvqWrMY6Wj...



Select a Chain

Neo Legacy



Neo N3



NEO

426



GAS

115.07821835

3. 지갑 이름과 비밀번호를 입력한 뒤 생성(Create)을 클릭합니다. 새로운 Neo N3 지갑이 성공적으로 생성됩니다.

참고

안내 메시지를 따라 개인 키를 반드시 안전하게 백업하십시오.

NeoLine

TestNet ▾

Create Import

Welcome to NeoLine. Create

Set wallet name *
TestWallet

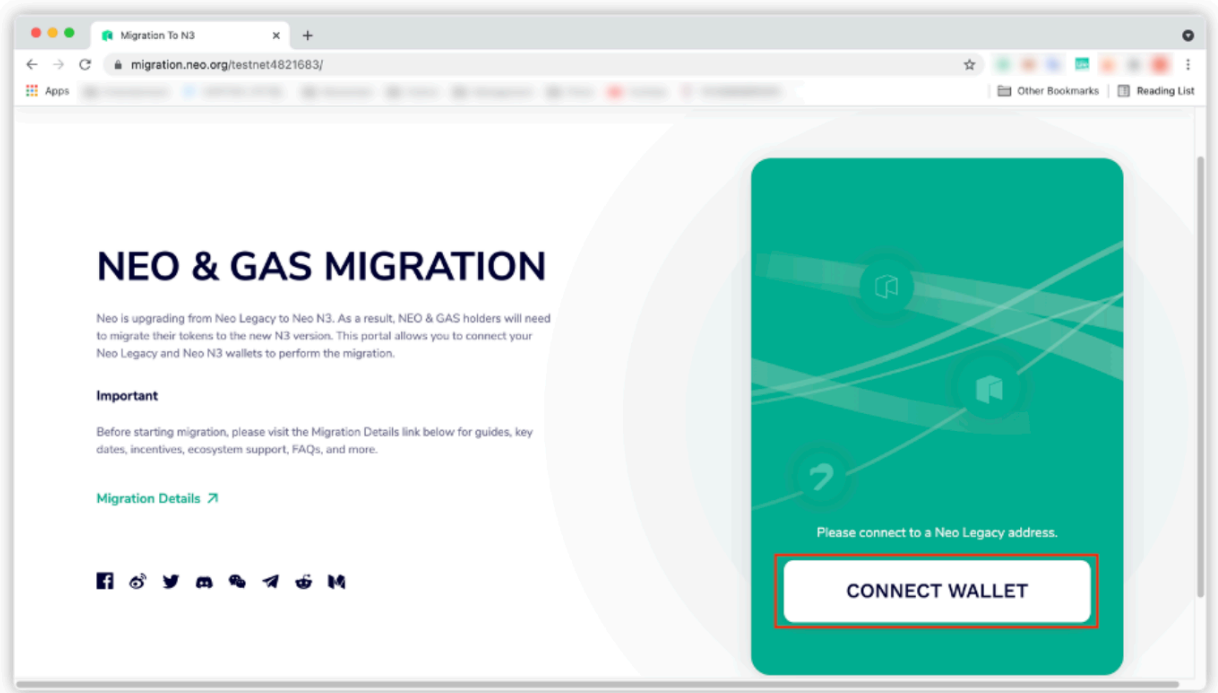
Enter password *
.....

Enter password again *
.....

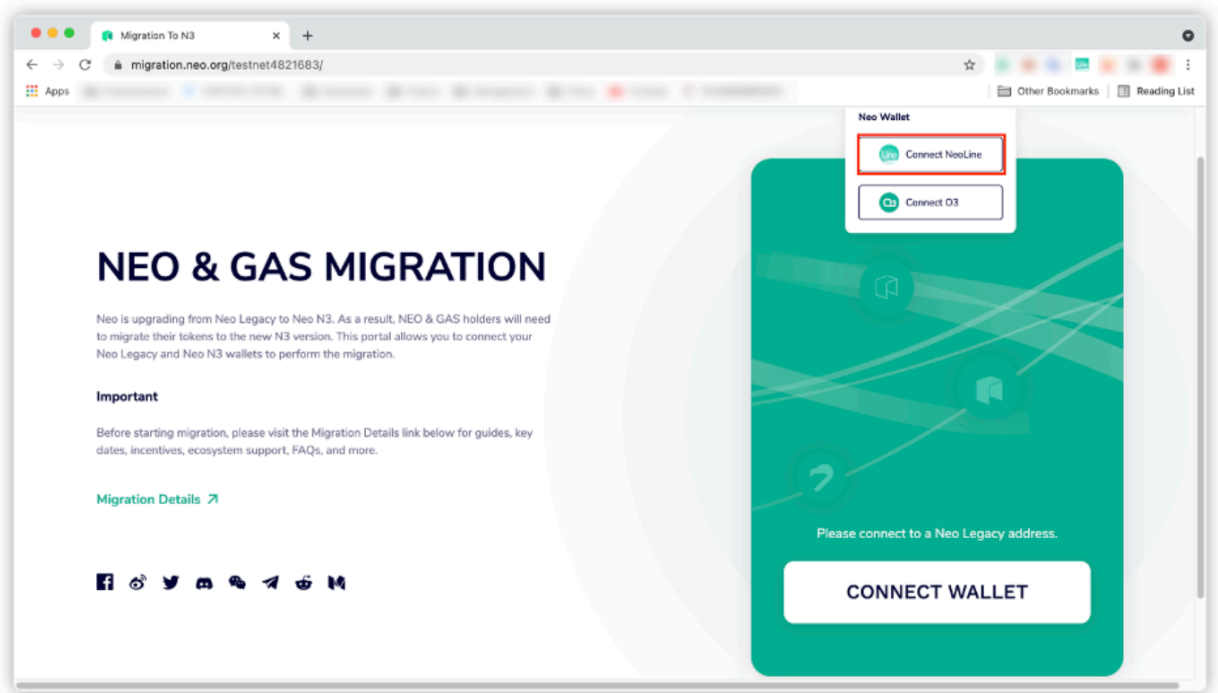
Create

Cancel

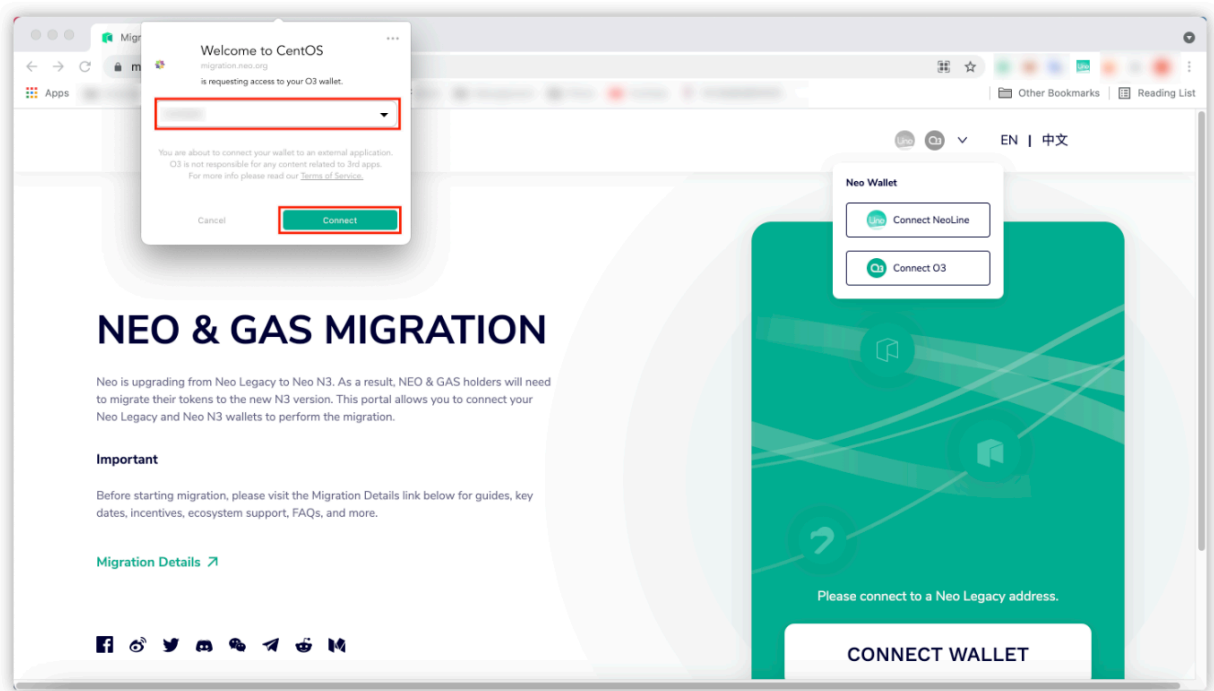
4. [Migration To N3](#) 웹사이트로 이동합니다. CONNECT WALLET 버튼을 클릭합니다.



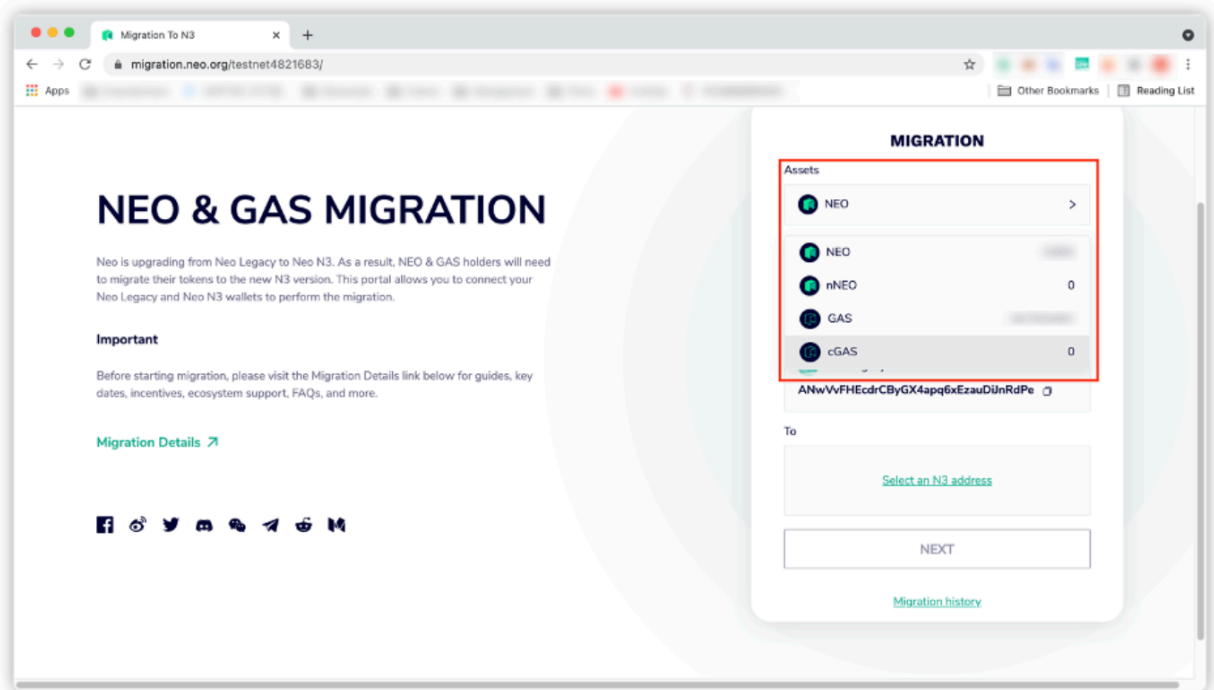
5. Connect NeoLine을 클릭하여 NeoLine 지갑을 연결합니다.



6. Connect를 클릭합니다.



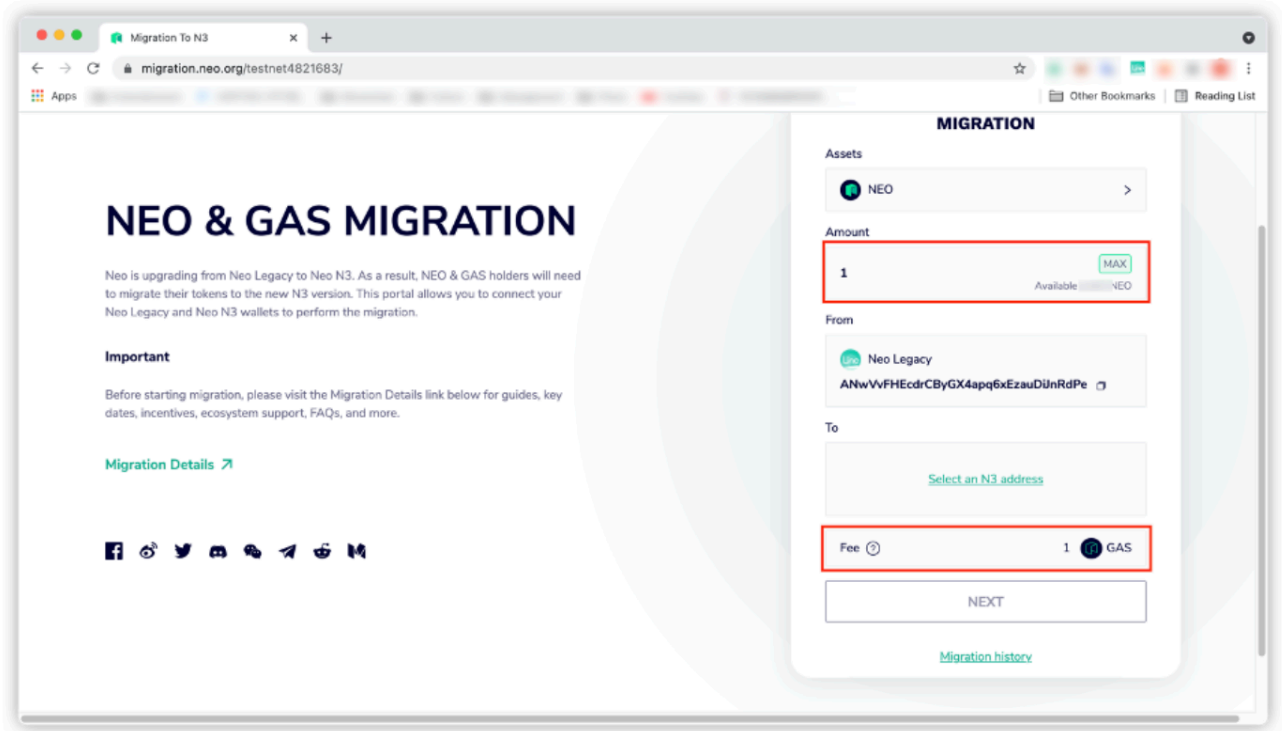
7. 마이그레이션할 자산을 선택합니다.



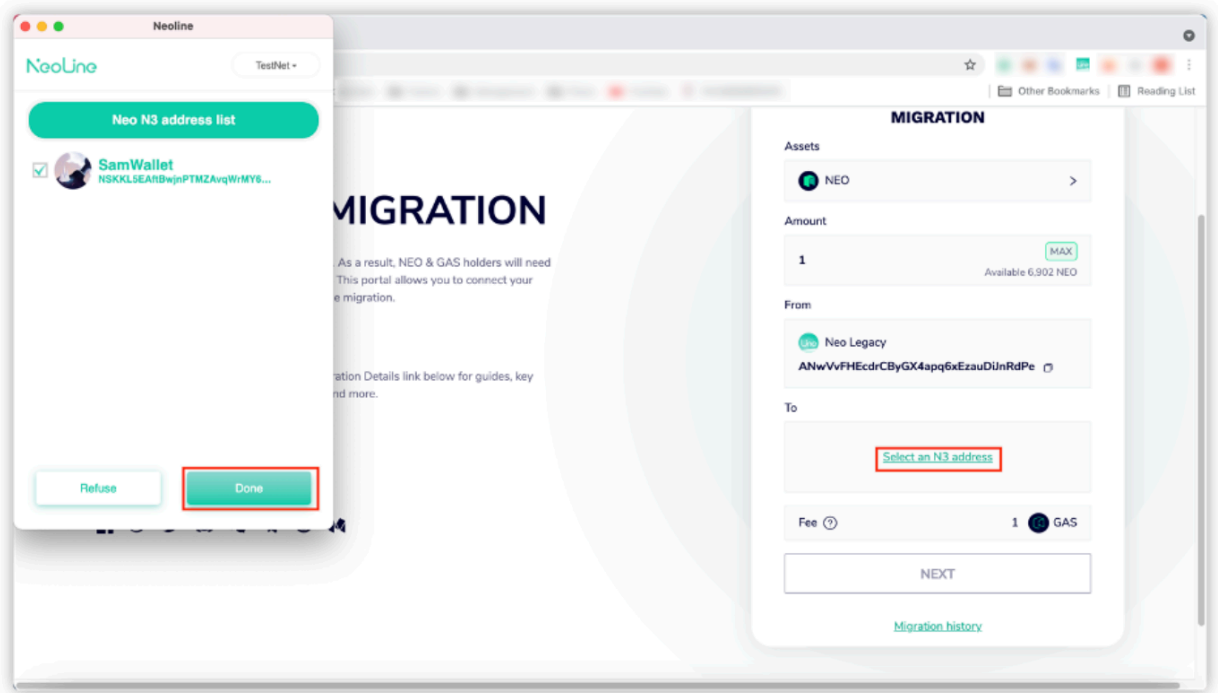
8. 마이그레이션할 금액을 입력합니다.

참고

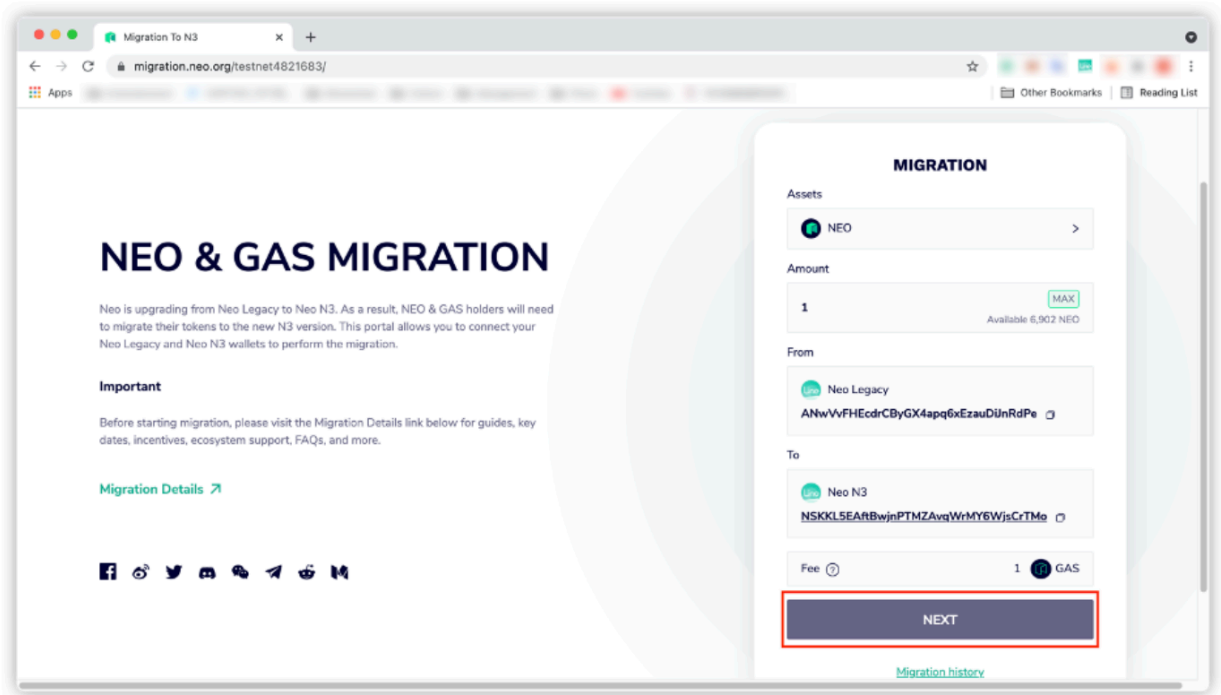
10 NEO 또는 20 GAS 미만을 마이그레이션할 경우 1 GAS의 수수료가 필요합니다.
 이는 Neo N3에서 모든 트랜잭션에 수수료가 요구되기 때문입니다.
 사용자가 소량의 토큰을 수수료 없이 마이그레이션할 수 있도록 허용할 경우,
 Neo 재단의 GAS를 비효율적으로 소모시키는 트랜잭션 스팸 공격에 악용될 수 있습니다.



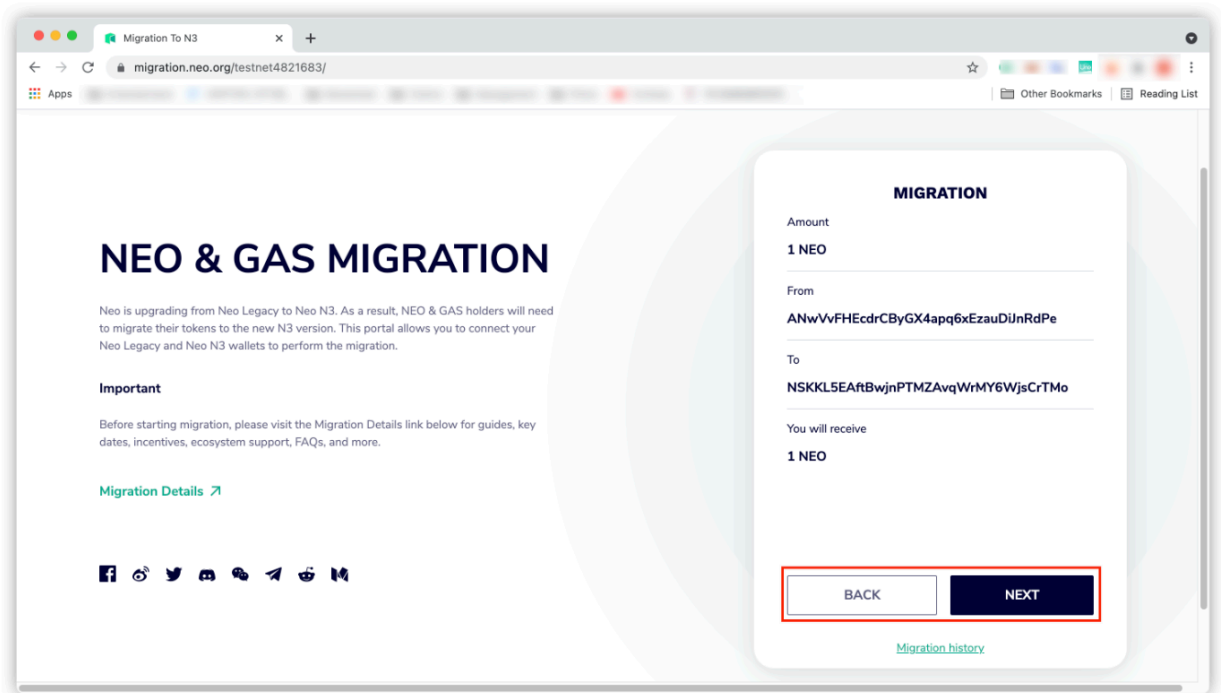
9. Select an N3 address를 클릭하여 마이그레이션 대상인 N3 지갑 주소를 선택한 후 Done을 클릭합니다.



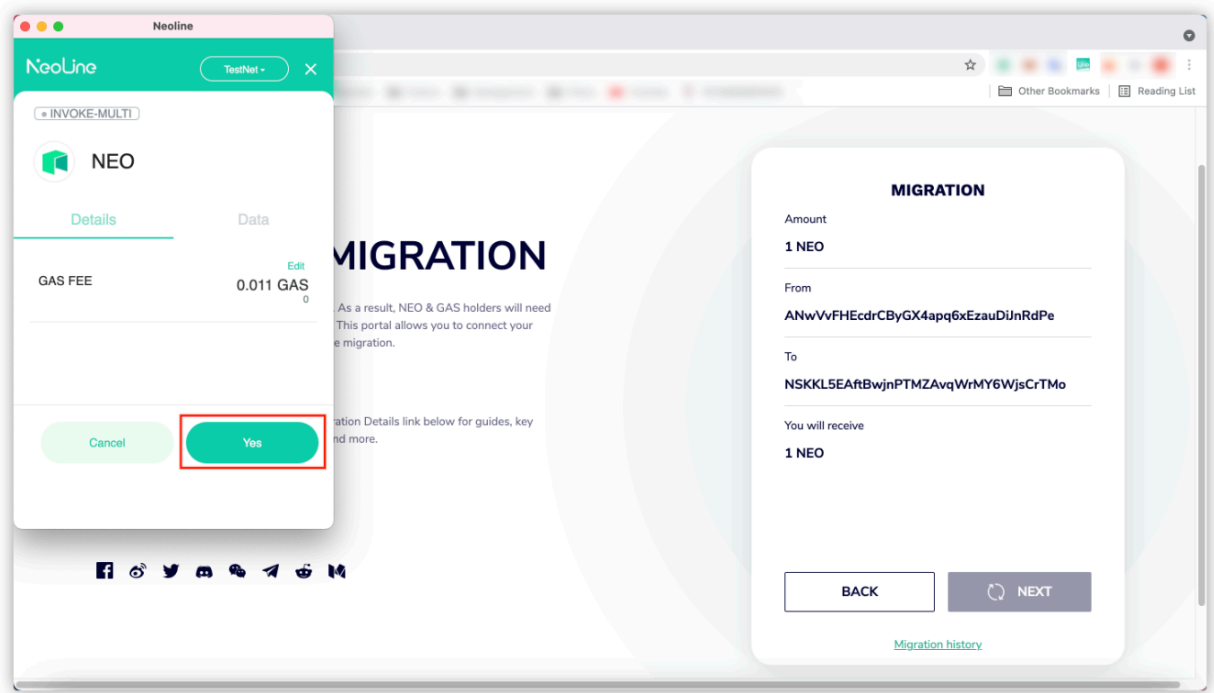
10. NEXT를 클릭하여 마이그레이션을 시작합니다.



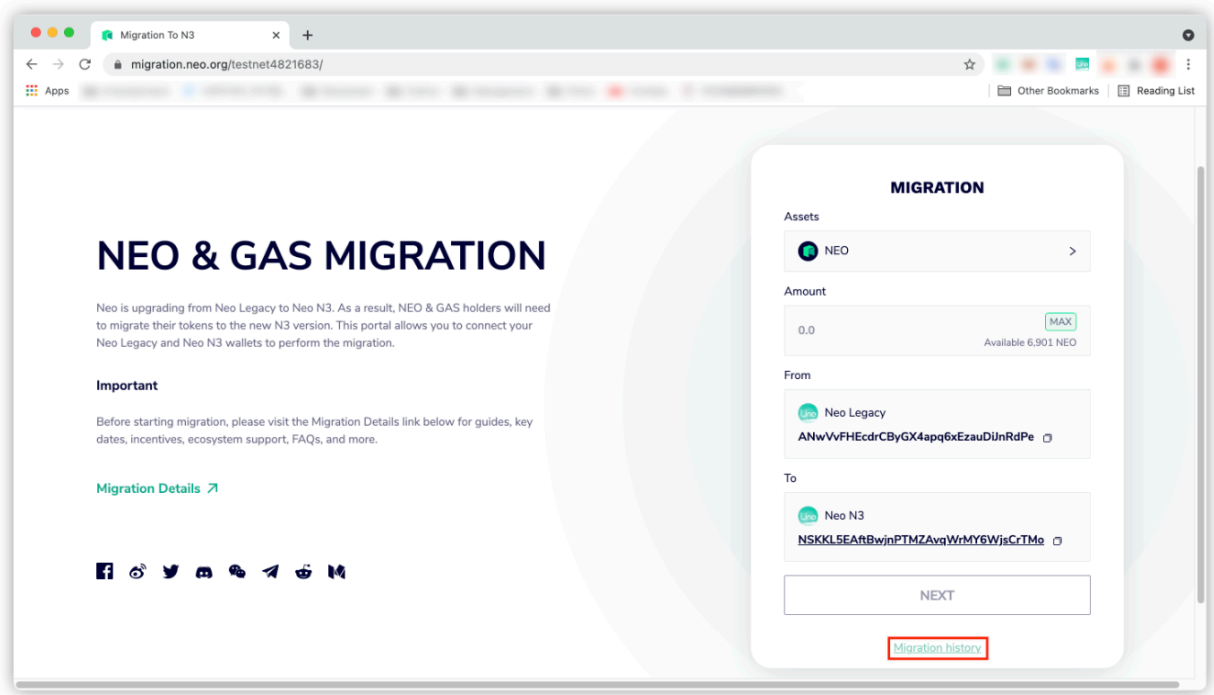
11. 마이그레이션 정보를 확인합니다. 정보가 정확하다면 NEXT를 클릭합니다.
정보를 수정하려면 BACK 버튼을 클릭하여 이전 단계로 돌아갑니다.



12. Yes를 클릭하여 마이그레이션을 승인합니다.



13. 마이그레이션 프로세스가 완료될 때까지 기다립니다.
Migration history를 클릭하여 마이그레이션 상태를 확인할 수 있습니다.

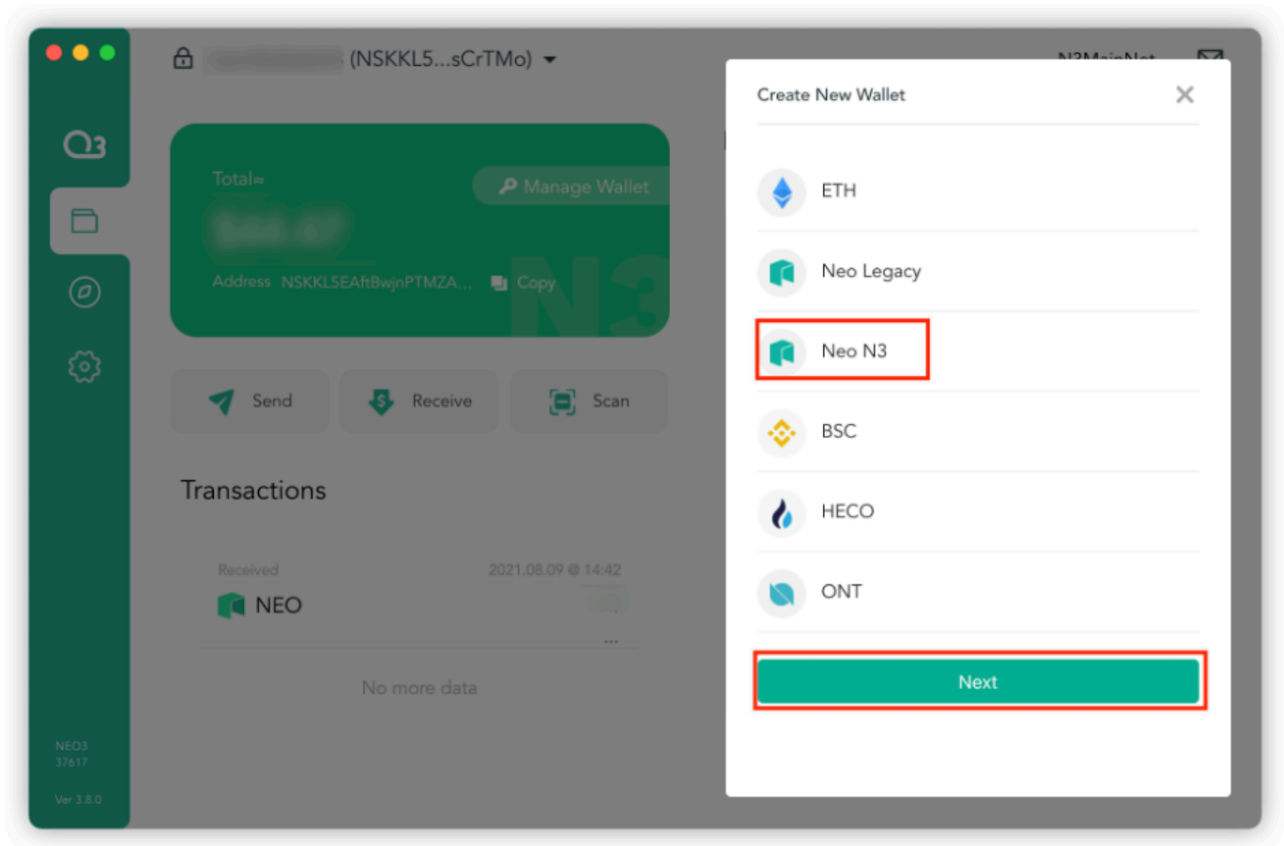


HISTORY						×
From	To	Assets	Amount	Time	Status	
hash:77d44f...7f5222	hash:N/A	NEO	1	2021.08.18 14:35	Pending	
hash:b09801...78594b	hash:46daeb...ac3b67	NEO	2	2021.08.18 11:57	Done	
hash:969492...f9a2cc	hash:de7253...73664e	NEO	10	2021.08.18 11:19	Done	

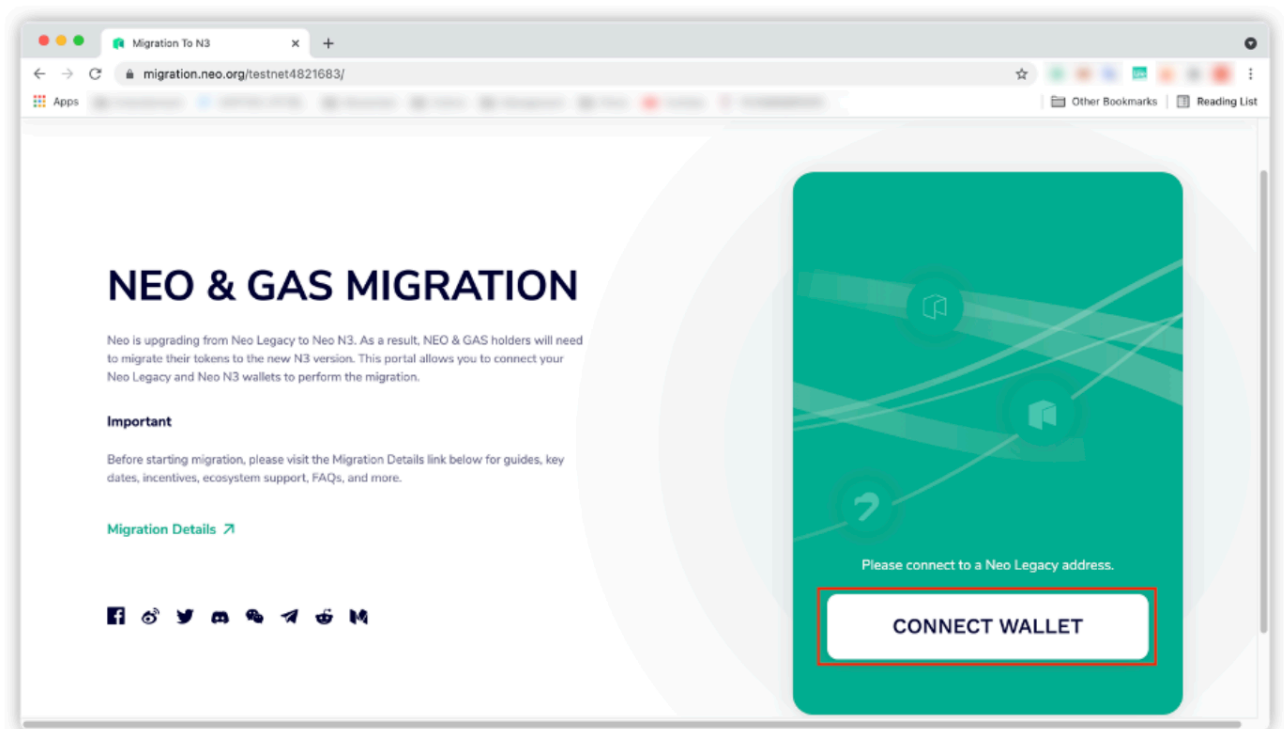
← 1 of 1 →

O3 Wallet을 이용한 마이그레이션

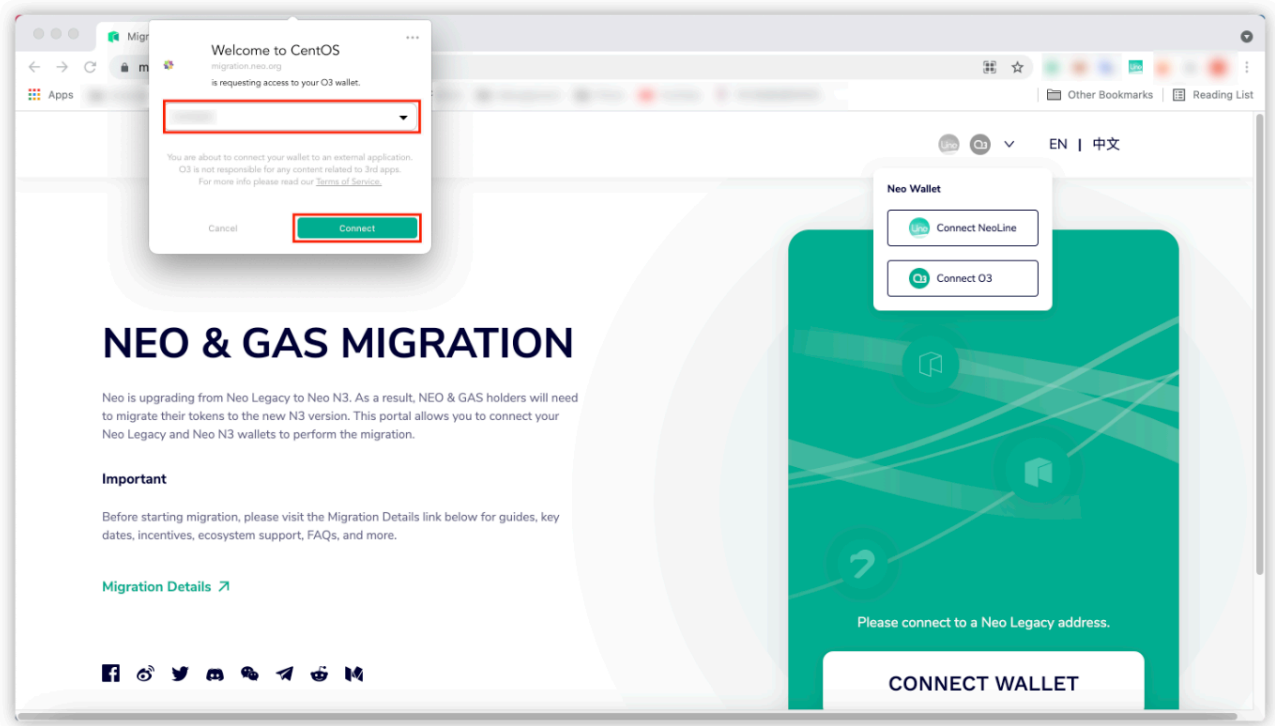
1. O3 지갑 앱을 실행합니다. 지갑 계정을 클릭한 후 Create(생성)을 클릭합니다.
2. NEO N3를 선택하고 Next를 클릭하여 Neo N3 지갑을 생성합니다.



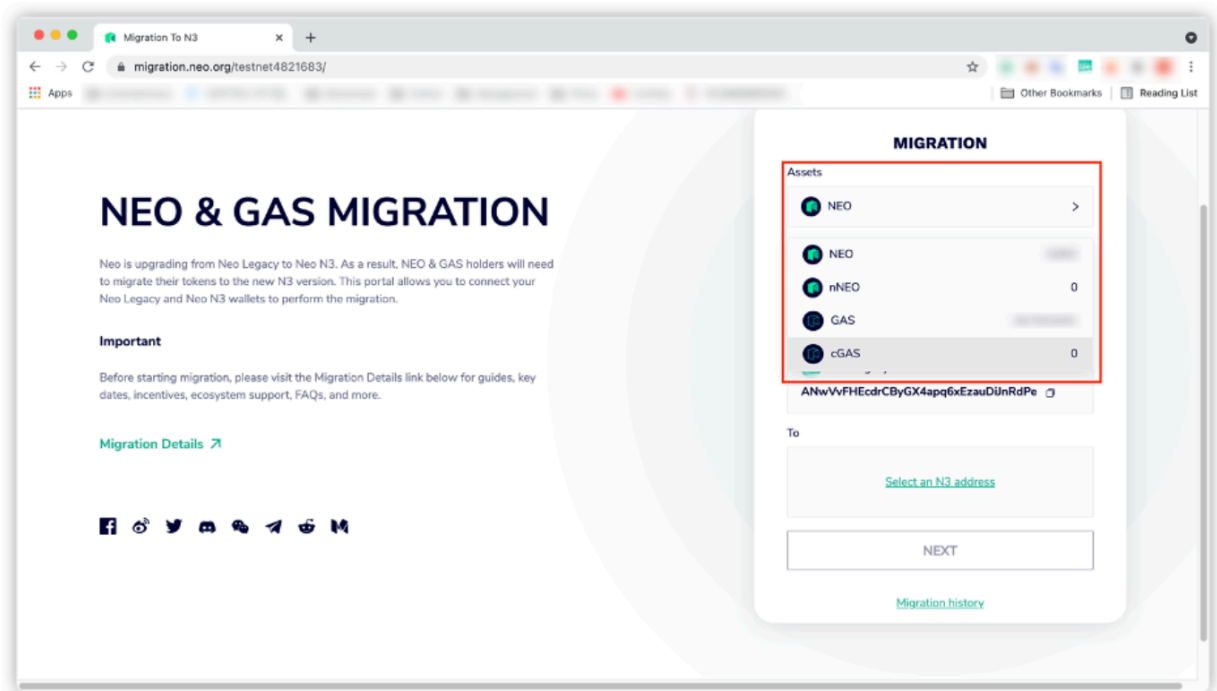
3. 지갑 이름과 비밀번호를 입력한 후 Create를 클릭합니다. 새로운 Neo N3 지갑이 성공적으로 생성됩니다.
4. [Migrate To N3](https://migration.neo.org/testnet4821683/) 웹사이트로 이동합니다. CONNECT WALLET을 클릭하여 지갑을 연결합니다.



5. Connect O3를 클릭하여 O3 지갑을 연결합니다.
6. 계정을 선택하고 Connect를 클릭합니다.



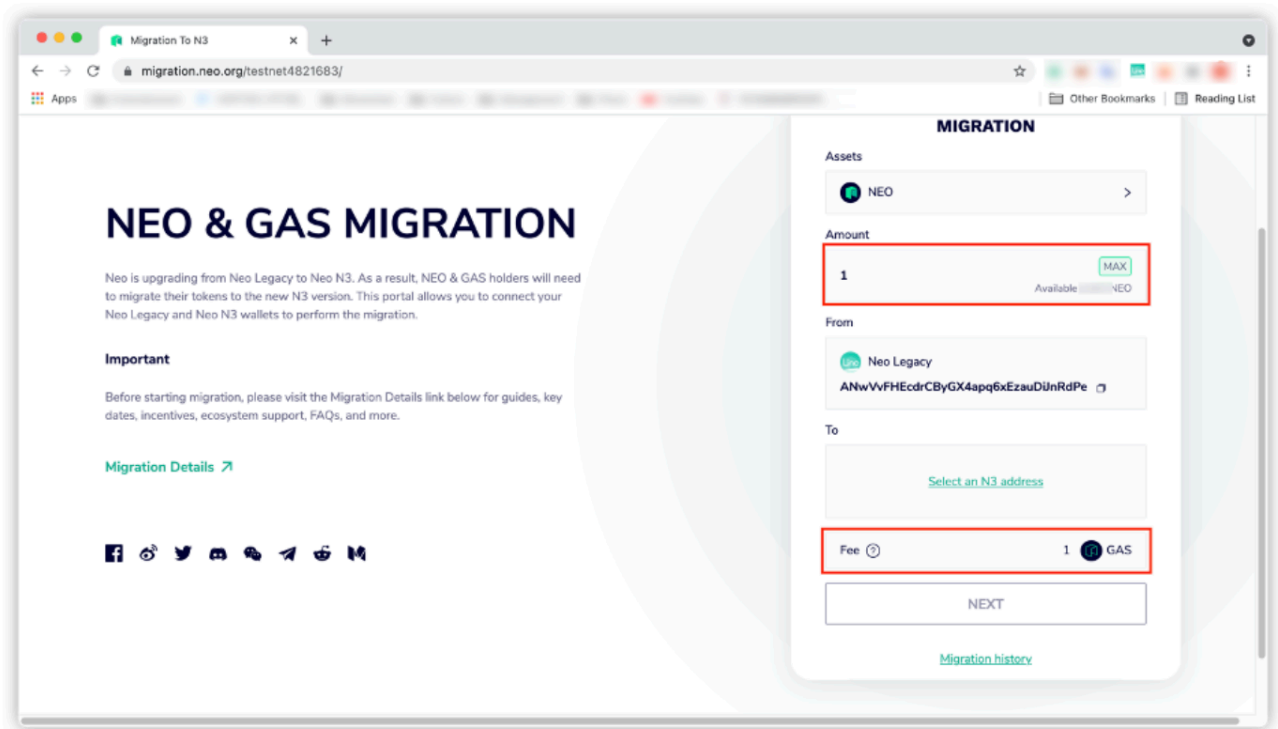
7. 마이그레이션할 자산을 선택합니다.



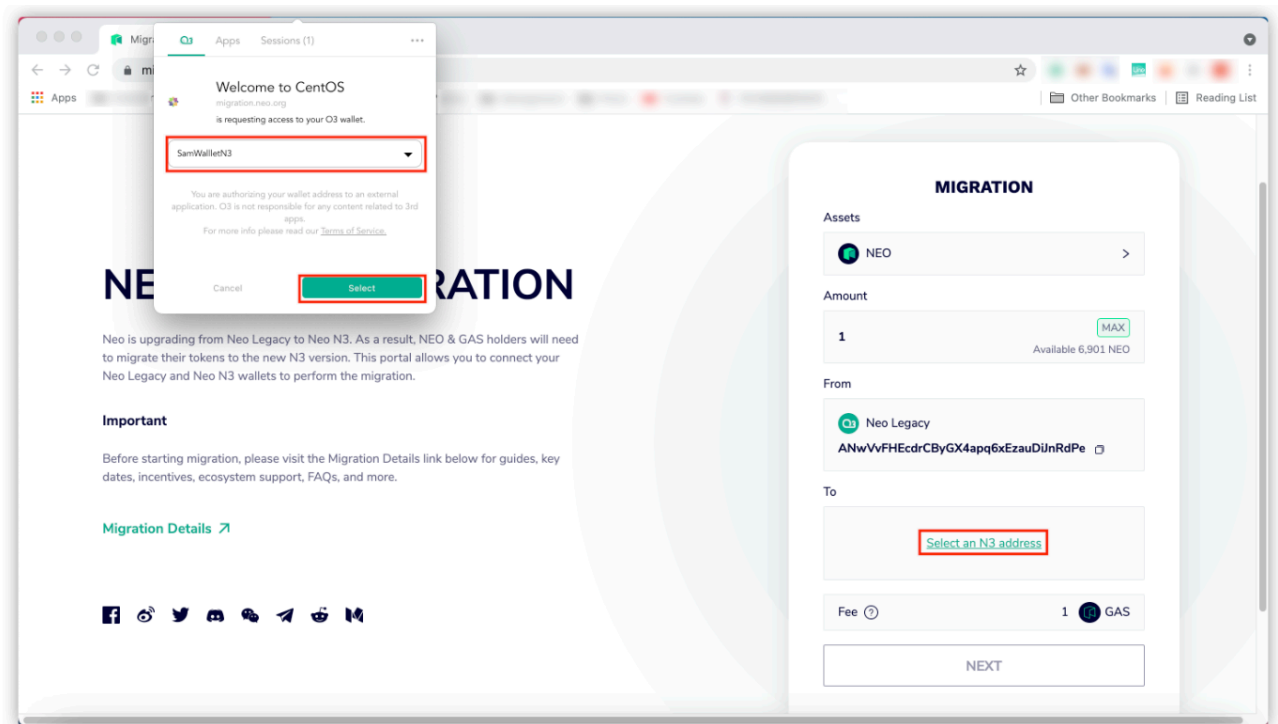
8. 마이그레이션할 금액을 입력합니다.

참고

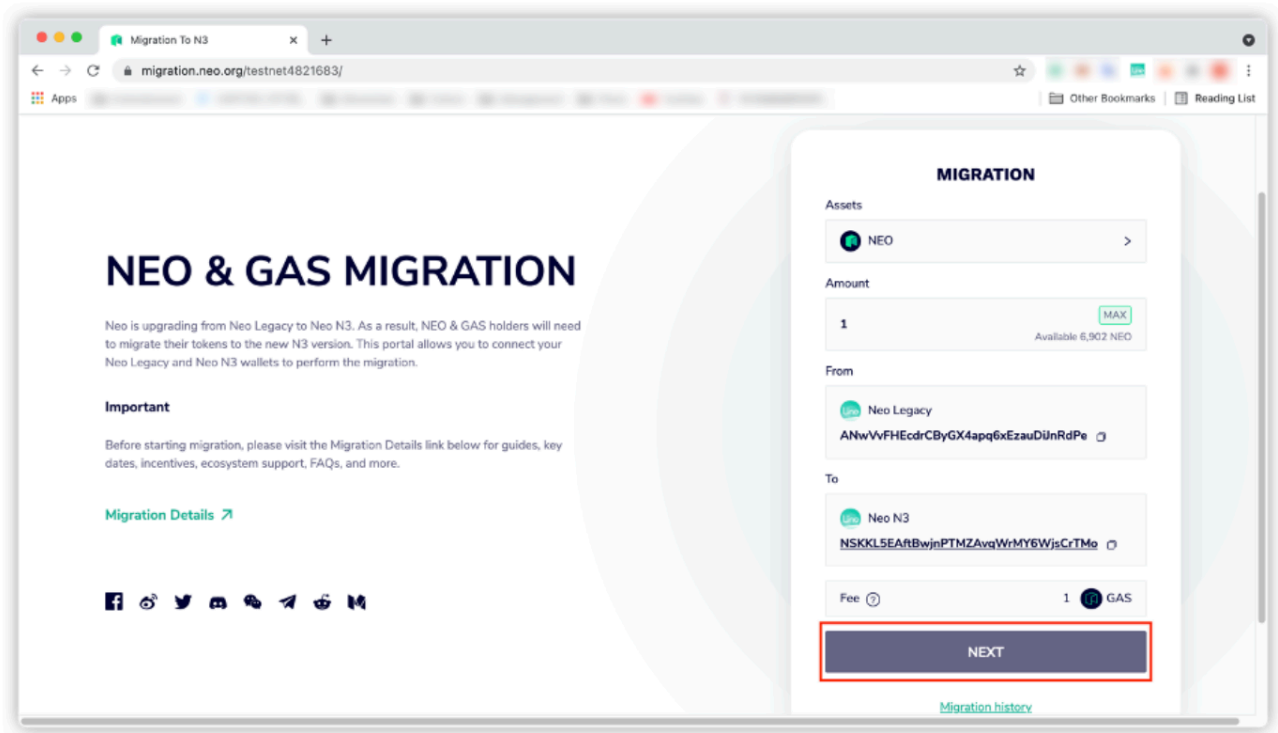
10 NEO 또는 20 GAS 미만을 마이그레이션할 경우 1 GAS의 수수료가 필요합니다.
 이는 Neo N3에서 모든 트랜잭션에 수수료가 요구되기 때문입니다.
 사용자가 소량의 토큰을 수수료 없이 마이그레이션할 수 있도록 허용하면,
 Neo 재단의 GAS를 비효율적으로 소모시키는 트랜잭션 스팸 공격에 악용될 수 있습니다.



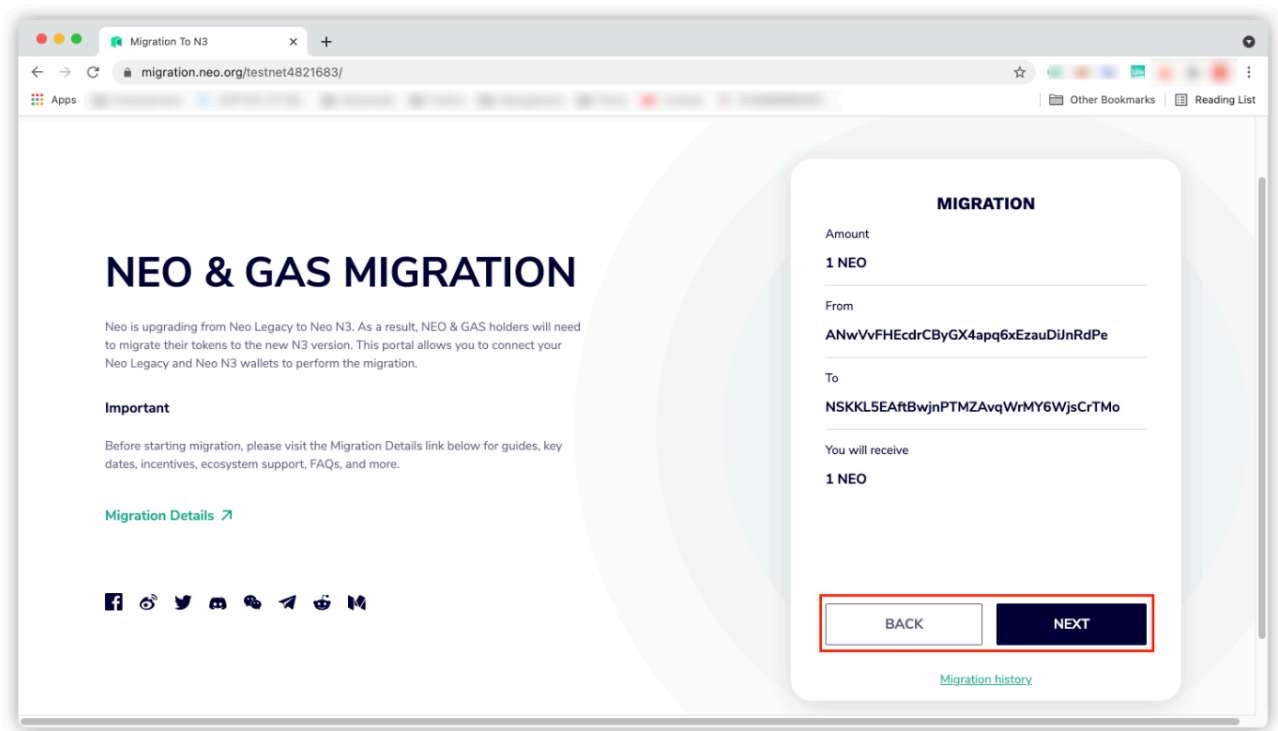
9. Select an N3 address를 클릭하여 마이그레이션 대상 주소를 선택한 후 Select를 클릭합니다.



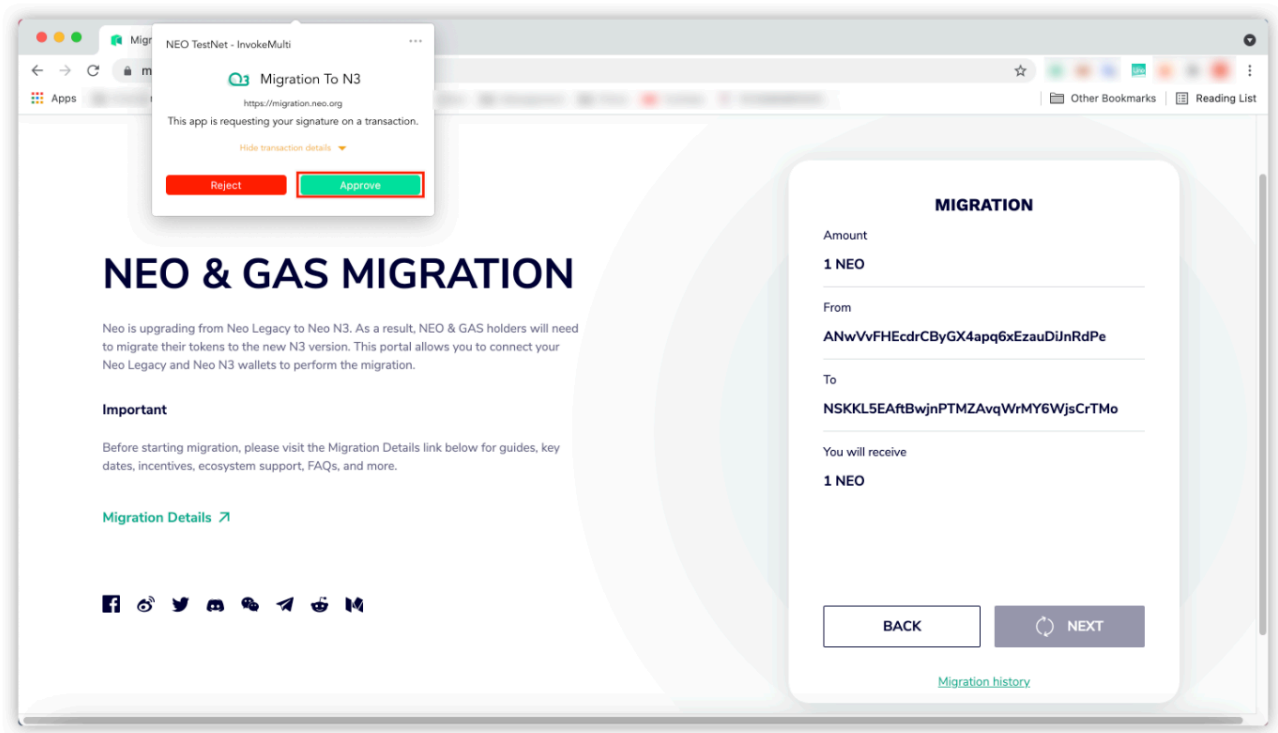
10. NEXT를 클릭하여 마이그레이션을 시작합니다.



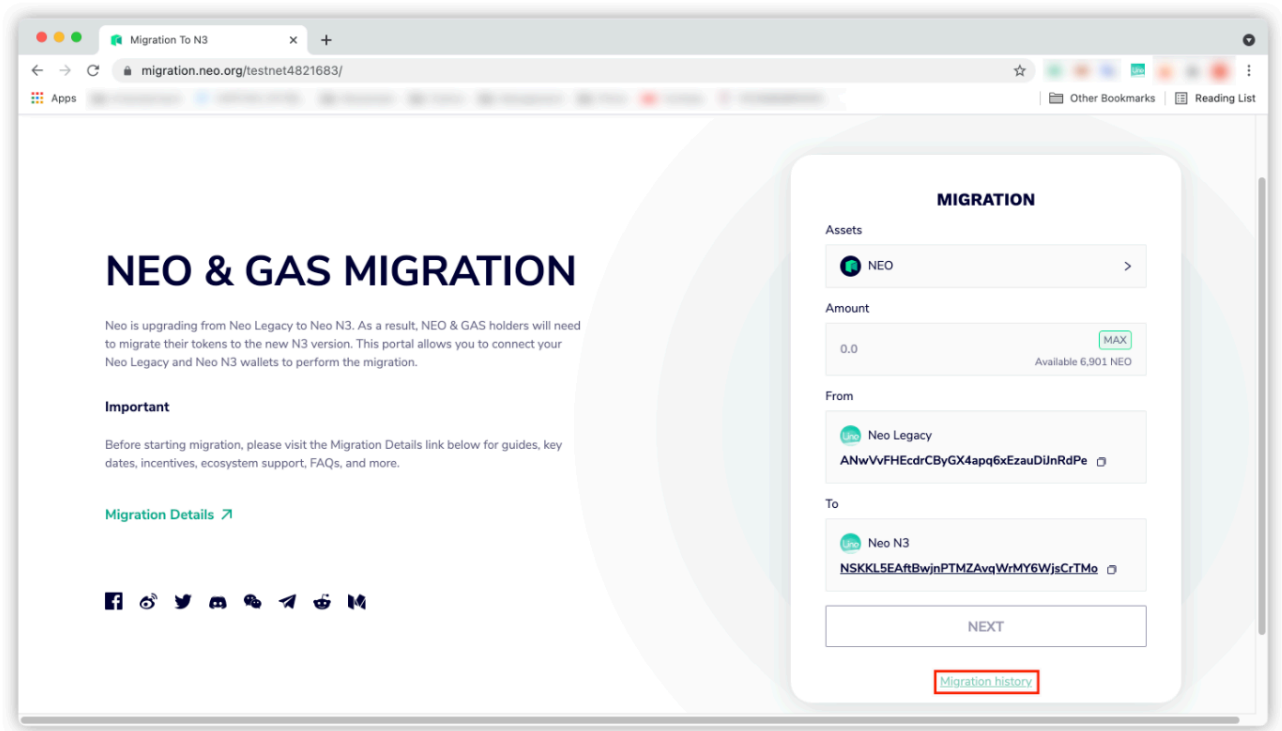
11. 마이그레이션 정보를 확인합니다. 정보가 정확하면 NEXT를 클릭하고, 수정이 필요하면 BACK을 클릭하여 정보를 수정합니다.



12. Approve를 클릭하여 마이그레이션을 승인합니다.



13. 마이그레이션 프로세스가 완료될 때까지 기다립니다.
Migration history를 클릭하여 마이그레이션 상태를 확인할 수 있습니다.



HISTORY

×

From	To	Assets	Amount	Time	Status
hash:77d44f...7f5222	hash:N/A	NEO	1	2021.08.18 14:35	Pending
hash:b09801...78594b	hash:46daeb...ac3b67	NEO	2	2021.08.18 11:57	Done
hash:969492...f9a2cc	hash:de7253...73664e	NEO	10	2021.08.18 11:19	Done

← 1 of 1 →

Neo Legacy에서 Neo N3로의 마이그레이션

이 튜토리얼은 거래소 개발자를 대상으로, Neo Legacy 체인의 글로벌 자산 NEO/GAS/nNEO/cGAS를 Neo N3 체인으로 전송 트랜잭션을 통해 마이그레이션하는 방법을 안내합니다.

마이그레이션 절차 흐름

- Neo Legacy 측에서 사용자는 다음과 같은 지정 주소로 전송 트랜잭션(InvocationTransaction 또는 ContractTransaction)을 보냅니다:
ANeo2toNeo3MigrationAddressxwPB2Hz
(테스트넷 주소: AJ36ZCpMhiHYdMAUaP7i1i9pJz4jMdiQV)
이때 트랜잭션의 attribute에 N3 표준 주소를 Remark14 키를 사용하여 포함시킵니다.

다양한 언어로 작성된 코드 데모가 참고용으로 제공됩니다: Demos

⚠ 위험

트랜잭션을 전송하기 전에 아래 모든 조건을 반드시 충족해야 합니다.
조건을 만족하지 않으면 트랜잭션이 처리되지 않으며, 전송된 자산은 손실될 수 있습니다!

- 트랜잭션에 포함된 N3 주소는 유효해야 하며, attribute 내에 Remark14는 단 하나만 존재해야 합니다.

- 마이그레이션 금액이 10 NEO 이상 또는 20 GAS 이상인 경우에는 수수료가 없습니다.
해당 기준 미만의 금액을 마이그레이션할 경우, **1 GAS를 네트워크 수수료**로 반드시 첨부해야 합니다.
- 하나의 트랜잭션에는 GAS(네트워크 수수료 목적)를 제외하고, 단일 종류의 자산만 마이그레이션할 수 있습니다.

참고

nNEO의 소수점 이하 단위는 마이그레이션이 지원되지 않습니다.

N3 블록체인 상에서 nNEO를 분배할 때, 마이그레이션 금액은 내림(round down) 처리됩니다.

2. 우리는 지정 주소로 전송된 모든 Neo Legacy 트랜잭션을 스캔하고, 유효한 경우에 한하여 정해진 간격(최대 영업일 기준 1일 이내)으로 해당 N3 주소에 상응하는 NEO/GAS 토큰을 전송합니다.

함께 보기

- 자산 컨트랙트 해시 목록(<https://github.com/neo-ngd/sdkDemo/blob/master/contracthash.md>)
- [마이그레이션 페이지 가이드라인](#)

수수료(Fees)

Neo 생태계의 참여자는 Neo 네트워크를 사용할 때 GAS로 네트워크 수수료 및 시스템 수수료를 지불해야 합니다.

시스템 수수료는 NeoVM에서 트랜잭션 실행 시 소비되는 리소스 비용입니다.

스마트 컨트랙트의 실행(자체 스크립트의 실행 및 다른 컨트랙트 호출 포함)에는 시스템 수수료가 필요합니다.

네트워크 수수료는 트랜잭션을 블록에 포함시키기 위한 비용입니다.

블록체인에 스마트 컨트랙트를 배포하거나 호출하고, Verify 메서드를 실행하는 트랜잭션은 네트워크 수수료를 요구합니다.

따라서, 스마트 컨트랙트를 호출하는 트랜잭션은 시스템 수수료와 네트워크 수수료를 모두 필요로 합니다.

sysfee는 각 트랜잭션마다 소각되며, netfee는 현재 블록을 생성한 합의 노드의 주소로 보상됩니다.

Neo3의 수수료는 동적으로 변경될 수 있습니다.

이 문서에서는 기본적인 수수료만을 소개합니다.

시스템 수수료 (System fees)

시스템 수수료에는 다음 항목이 포함됩니다:

- 명령어 수수료 (OpCode)
- 시스템 호출 수수료 (SysCall)
- 네이티브 컨트랙트에 대한 CPU 처리 수수료
- 저장소 수수료 (Storage fee)

명령어 수수료 (Instruction fee)

수수료 (Datashi)	OpCode
32768	CALLT
8192	VALUES, APPEND, SETITEM, REVERSEITEMS, CONVERT
4096	PUSHDATA4
2048	MEMCPY, CAT, SUBSTR, LEFT, RIGHT, MODPOW, PACKMAP, PACKSTRUCT, PACK, UNPACK
512	PUSHDATA2, CALL, CALLL, CALLA, THROW, NEWARRAY, NEWARRAYT, NEWSTRUCT
256	NEWBUFFER
64	INITSLT, POW, SQRT, HASKEY, PICKITEM
32	EQUAL, NOTEQUAL, MODMUL
16	XDROP, CLEAR, ROLL, REVERSE, INITSLT, NEWARRAYo, NEWSTRUCTo, KEYS, REMOVE, CLEARITEMS, POPITEM
8	PUSHDATA1, AND, OR, XOR, ADD, SUB, MUL, DIV, MOD, SHL, SHR, BOOLAND, BOOLOR, NUMEQUAL, NUMNOTEQUAL, LT, LE, GT, GE, MIN, MAX, WITHIN, NEWMAP
4	PUSHINT128, PUSHINT256, PUSHA, TRY, TRYL, ENDTRY, ENDTRYL, ENDFINALLY, INVERT, SIGN, ABS, NEGATE, INC, DEC, NOT, NZ, SIZE
2	JMP, JMPL, JMPIF, JMPIFL, JMPIFNOT, JMPIFNOTL, JMPEQ, JMPEQL, JMPNE, JMPNEL, JMPGT, JMPGTL, JMPGE, JMPGEL, JMPLT, JMPLTL, JMPLE, JMPLE_L, DEPTH, DROP, NIP, DUP, OVER, PICK, TUCK, SWAP, ROT, REVERSE3, REVERSE4, LDSFLD0, LDSFLD1, LDSFLD2, LDSFLD3, LDSFLD4, LDSFLD5, LDSFLD6, LDSFLD, STSFLD0, STSFLD1, STSFLD2, STSFLD3, STSFLD4, STSFLD5, STSFLD6, STSFLD, LDLOC0, LDLOC1, LDLOC2, LDLOC3, LDLOC4, LDLOC5, LDLOC6, LDLOC, STLOC0, STLOC1, STLOC2, STLOC3, STLOC4, STLOC5, STLOC6, STLOC, LDARG0, LDARG1, LDARG2, LDARG3, LDARG4, LDARG5, LDARG6, LDARG, STARG0, STARG1, STARG2, STARG3, STARG4, STARG5, STARG6, STARG, ISNULL, ISTYPE
1	PUSHINT8, PUSHINT16, PUSHINT32, PUSHINT64, PUSHT, PUSHF, PUSHNULL, PUSHM1, PUSH0, PUSH1, PUSH2, PUSH3, PUSH4, PUSH5, PUSH6, PUSH7, PUSH8, PUSH9, PUSH10, PUSH11, PUSH12, PUSH13, PUSH14, PUSH15, PUSH16, NOP, ASSERT, ASSERTMSG
0	ABORT, ABORTMSG, RET, SYSCALL

참조:
ApplicationEngine.OpCodePrices.cs(<https://github.com/neo-project/neo/blob/master/src/Neo/SmartContract/ApplicationEngine.OpCodePrices.cs>)

시스템 호출 수수료 (System call fee)

인터롭 서비스 (Interop Service)	수수료 (GAS)
System.Contract.Call	32768
System.Contract.CallNative	네이티브 컨트랙트 수수료 참조
System.Contract.IsStandard	1024
System.Contract.GetCallFlags	1024
System.Contract.CreateStandardAccount	256
System.Contract.CreateMultisigAccount	256
Neo.Crypto.CheckSig	32768
Neo.Crypto.CheckMultisig	서명 수에 따라 동적으로 계산됨
System.Iterator.Create	16
System.Iterator.Next	32768
System.Iterator.Value	16
System.Runtime.Platform	8
System.Runtime.GetTrigger	8
System.Runtime.GetTime	8
System.Runtime.GetScriptContainer	8
System.Runtime.GetExecutingScriptHash	16
System.Runtime.GetCallingScriptHash	16
System.Runtime.GetEntryScriptHash	16
System.Runtime.CheckWitness	1024
System.Runtime.GetInvocationCounter	16
System.Runtime.Log	32768
System.Runtime.Notify	32768
System.Runtime.GetNotifications	256
System.Runtime.GasLeft	16

참조:

- ApplicationEngine.Contract.cs(<https://github.com/neo-project/neo/blob/master/src/Neo/SmartContract/ApplicationEngine.Contract.cs>)
- ApplicationEngine.Crypto.cs(<https://github.com/neo-project/neo/blob/master/src/Neo/SmartContract/ApplicationEngine.Crypto.cs>)
- ApplicationEngine.Iterator.cs(<https://github.com/neo-project/neo/blob/master/src/Neo/SmartContract/ApplicationEngine.Iterator.cs>)

- ApplicationEngine.Runtime.cs(<https://github.com/neo-project/neo/blob/master/src/Neo/SmartContract/ApplicationEngine.Runtime.cs>)
- ApplicationEngine.Storage.cs(<https://github.com/neo-project/neo/blob/master/src/Neo/SmartContract/ApplicationEngine.Storage.cs>)

네이티브 컨트랙트 실행 수수료 (Native contract execution fee)

네이티브 컨트랙트	메서드	수수료 (GAS)
ContractManagement	Deploy	저장소 수수료 참조. 최소 10 GAS
ContractManagement	Update	저장소 수수료 참조
LedgerContract	GetTransactionFromBlock	65536
NeoToken	UnclaimedGas	131072
NeoToken	RegisterCandidate	동적 계산. 기본값 1000 GAS
NeoToken	UnregisterCandidate	65536
NeoToken	Vote	65536
NeoToken	GetCandidates	4194304
NeoToken	GetCommittee	4194304
NeoToken	GetNextBlockValidators	4194304
NeoToken, GasToken	Transfer	131072
OracleContract	Request	동적 계산. 호출 시 사용자가 수수료 지정
StdLib	Deserialize	8192
StdLib	JsonDeserialize	8192
StdLib	기타	2048

기타 명시되지 않은 네이티브 컨트랙트 메서드 수수료는 32768 Datoshi입니다.

참조:
neo/SmartContract/Native(<https://github.com/neo-project/neo/tree/master/src/Neo/SmartContract/Native>)

저장소 수수료 (Storage fee)

저장소 수수료는 기록된 바이트 수에 따라 부과됩니다. 기본 단가는 **0.001 GAS / Byte**입니다.
위원회 구성원이 동적으로 조정할 수 있으며, 상한은 **1 GAS / Byte**입니다.

시나리오	부과 방식	예시	수수료 (0.001 단가 기준)
최초 기록	(key + value 길이) × 단가	key + hello world (총 14 바이트)	0.014 GAS
이후 기록 (신규 ≤ 기존)	첫 1바이트 정상 요금 + 나머지 75% 할인	값 변경: hello neo3 (10바이트)	$(1 + (10-1)/4) \times 0.001 =$ 0.003 GAS
이후 기록 (신규 > 기존)	이전 요금 + 추가 바이트 요금	값 변경: hello neo3.0 (12바이트)	$0.003 + (12-10) \times 0.001 =$ 0.005 GAS
이후 기록 (신규 >> 기존)	위와 동일	값 변경: hello neo3.0 preview5 (21바이트)	$0.005 + (21-12) \times 0.001 =$ 0.014 GAS
값 삭제	없음	값 삭제	0 GAS

참조:
 ApplicationEngine.Storage.cs(<https://github.com/neo-project/neo/blob/master/src/Neo/SmartContract/ApplicationEngine.Storage.cs>)

네트워크 수수료 (Network fee)

네트워크 수수료는 다음을 포함합니다:

- 네트워크 바이트 수수료 (Network byte fee)
- 스크립트 검증 수수료 (Script verification fee)

네트워크 바이트 수수료

기본 단가는 **0.000001 GAS / Byte**이며, 위원회에서 동적으로 조정 가능합니다.

참조:

- PolicyContract.cs(<https://github.com/neo-project/neo/blob/master/src/Neo/SmartContract/Native/PolicyContract.cs>)
- Transaction.cs#L302(<https://github.com/neo-project/neo/blob/master/src/Neo/Network/P2P/Payloads/Transaction.cs#L302>)

스크립트 검증 수수료

스크립트 검증 수수료는 **최대 0.5 GAS**로 제한되며, 계산식은 다음과 같습니다:

$$\text{스크립트 검증 수수료} = \text{스크립트 실행 수수료} \times \text{계수(multiple)}$$

- 스크립트 실행 수수료는 다음을 포함합니다:
OpCode 수수료 + SysCall 수수료 + 네이티브 컨트랙트 CPU 수수료 + 저장소 수수료
- 기본 multiple 값은 **30**이며, 위원회에서 **최대 1000**까지 조정 가능
- InvokeContractVerify API를 사용해 예상 수수료를 계산할 수 있습니다.

예시:

표준 주소의 검증 수수료:

$$(\text{PUSHDATA1} + \text{PUSHDATA1} + \text{SYSCALL} + \text{CheckSig}) \times 30 = \mathbf{0.0098352 \text{ GAS}}$$

참조:

- PolicyContract.cs(<https://github.com/neo-project/neo/blob/master/src/Neo/SmartContract/Native/PolicyContract.cs>)
- Transaction.cs#L302(<https://github.com/neo-project/neo/blob/master/src/Neo/Network/P2P/Payloads/Transaction.cs#L302>)

용어집 (Glossary)

블록 (Block)

블록체인에서 블록은 여러 개의 트랜잭션 기록을 포함하는 단위입니다. 각 블록은 이전 블록의 타임스탬프와 암호학적 해시를 블록 헤더에 포함하여 이전 블록과 연결되며, 이렇게 연결된 블록들의 연속이 블록체인(Blockchain)을 형성합니다.

지갑 (Wallet)

지갑은 Neo 네트워크에서 발생하는 데이터베이스 트랜잭션에 서명하고 인증하는 데 사용되는 암호학적 공개/비공개 키 쌍입니다.

계정 / 주소 (Account / Address)

은행 계좌와 유사하게, 블록체인 계정은 트랜잭션에서 자산을 전송하거나 수신하는 데 사용됩니다. 주소는 스크립트 해시로부터 특정 연산을 통해 생성되며, 다시 스크립트 해시로 변환할 수 있습니다. Neo Legacy에서는 주소가 **A**로 시작하며, Neo N3에서는 **N**으로 시작합니다.

개인 키 / 공개 키 (Private Key / Public Key)

개인 키는 사용자 본인이 보관하며, 타인에게 공개하지 않는 256비트의 난수입니다. 계정의 사용 권한 및 해당 계정의 자산 소유권을 증명하는 수단입니다. 각 개인 키는 그에 대응하는 공개 키를 가집니다.

트랜잭션 (Transaction)

Neo의 트랜잭션은 네트워크에 지시를 전달하는 서명된 데이터 패키지입니다. 예: 사용자가 특정 주소로 자산을 전송하고자 하는 요청. 블록체인 원장 내의 각 Neo 블록은 하나 이상의 트랜잭션을 포함하며, 각 블록은 트랜잭션 배치로 간주됩니다.

스크립트 (Script)

스크립트는 OpCode와 오퍼랜드(Operands)로 구성된 코드 조각입니다. 각 계정은 스크립트를 가진 컨트랙트이며, 표준 계정 스크립트는 공개 키와 서명을 검증하는 OpCode로 구성됩니다. 형식 예시는 다음과 같습니다:

```
PUSHDAT1 03ac765294075da6f7927c96bfe3d3f64ae3680c5eb50f82f55170a9f1bea59dad
SYSCALL Neo.Crypto.VerifyWithECDSaSecp256r1
```

스크립트 해시 (Script Hash)

스크립트 해시는 스마트 컨트랙트 스크립트에서 **RIPEMD-160** 알고리즘을 통해 생성됩니다. 각 스크립트는 고유한 해시 값을 가지며, 스크립트 내용이 변경되면 스크립트 해시도 변경됩니다.

- 스크립트 해시 길이는 **20 바이트 (160비트)**입니다.
- 스크립트 해시는 단방향 해시로 생성되며, 스크립트로 역산할 수 없습니다.
- 스크립트 해시와 주소는 서로 변환 가능합니다.

OpCode (연산 코드)

OpCode는 어셈블리 언어의 명령어와 유사한 형태입니다.
전체 OpCode 목록은 [OpCode 소스](#)를 참고하세요.

암호화된 개인 키 (Cryptographic Private Key)

암호화된 개인 키란 지갑 비밀번호를 이용해 개인 키를 암호화한 형태를 의미합니다.
이는 평문 상태의 개인 키보다 안전합니다.
이 방식은 개인 키를 종이에 적어두고, 비밀번호만 기억하는 백업 전략을 가능하게 합니다.

스마트 컨트랙트 (Smart Contract)

블록체인 관점에서 스마트 컨트랙트는 디지털 형식으로 명시된 일련의 약속이며,
각 당사자는 해당 프로토콜에 따라 약속을 수행합니다.

NEP-17

NEP-17 제안서는 Neo 블록체인 상의 토큰 표준을 정의하며,
토큰화된 스마트 컨트랙트와 상호작용하는 일반화된 메커니즘을 시스템에 제공합니다.

SysCall (시스템 호출)

시스템 호출은 특별한 연산 코드이며, 이를 통해 상호운용 서비스 계층 인터페이스를 호출할 수 있습니다.
NeoVM은 SysCall을 통해 스마트 컨트랙트 실행 시 필요한 블록, 트랜잭션, 컨트랙트, 자산 정보에 접근할 수 있습니다.
자세한 정보는 Neo 스마트 컨트랙트 모듈의 `ApplicationEngine`으로 시작하는 소스들을 참고하십시오:
예: `ApplicationEngine.Contract.cs`, `ApplicationEngine.Blockchain.cs`
링크: Neo SmartContract GitHub(<https://github.com/neo-project/neo/tree/master/src/Neo/SmartContract>)

다이나믹 콜 (Dynamic Call)

컨트랙트 내부에서 다른 컨트랙트를 호출하는 특수한 시스템 호출입니다. 다음과 같이 작성됩니다:
`Contract.Call(scriptHash, method, params)`

자세한 내용은 [스마트 컨트랙트 호출 문서](#)를 참고하세요.

저장소 (Storage)

Neo 블록체인에 배포된 각 스마트 컨트랙트는 애플리케이션 데이터를 저장하기 위한 전용 저장소 영역을 가집니다.
컨트랙트를 생성하거나 트랜잭션을 통해 컨트랙트를 사용할 때, 해당 컨트랙트 코드는 이 저장소를 읽고 쓸 수 있어야 합니다.
각 컨트랙트는 자신의 저장소를 선언할 수 있습니다.
자세한 정보는 [Storage 문서](#)를 참고하세요.

NEF (Neo Executable Format)

NEF는 Neo 실행 포맷(Neo Executable Format)의 약자입니다.
스마트 컨트랙트 컴파일러는 소스 코드를 컴파일하여 NEF 파일과 매니페스트 파일(Manifest)을 생성합니다.

필드 (Field)	길이 (Length)	설명 (Comment)
Magic	4 바이트	매직 헤더
Compiler	32 바이트	사용된 컴파일러
Version	16 바이트	컴파일러 버전 (주버전, 부버전, 빌드, 수정)
ScriptHash	20 바이트	스크립트의 ScriptHash
Checksum	4 바이트	헤더의 Sha256 (CRC)
Script	가변 바이트	페이로드에 대한 가변 바이트 데이터