FLETA
TECH
PAPER

KOR

FLETA™

TABLE OF CONTENTS

CONCEPT

플레타의 기술은 서비스 가능한 성능의 블록체인 기술을 제공하는 것과 기존 개발 환경과의 통합 지원을 목표로 한다. 서비스 가능한 성능을 제공하기 위하여 높은 처리량과 낮은 블록 시간을 기반으로 하는 초고속 거래 처리를 지원하고 독립 멀티체인 구조를 통해 개별 서비스가 독립적으로 동작할 수 있도록 지원한다.
기존 개발 환경과의 통합을 제공하기 위하여 RDBMS, NoSQL, Event Sourcing 등 기존 개발 형태로 사용 가능한 Smart Contract를 제공한다. 이를 통해 블록체인 기반 서비스 개발과 기존 서비스 개발 사이에서 오는 진입 장벽과 개발 난이도를 낮추고 빠르고 효율적으로 서비스 개발 및 운영이 가능하도록 지원한다.

BLOCKCHAIN STRUCTURE

U.S.P(United States Patent) Application Number : 62717703

플레타는 UTXO와 Account 모델을 모두 지원하는 Hybrid 모델이다. UTXO는 여러 키를 사용하여 사용자를 특정하기 어렵게 만들 수 있다는 장점이 있으며, Account는 상대적으로 적은 데이터를 필요로하고 처리가 빠른 장점이 있다. 누구나 키 생성만으로 UTXO 기반의 거래를 수행할 수 있으며, 소량의 플레타를 이용하면 Account를 개설하여 UTXO에 비하여 낮은 거래 수수료로 거래를 수행할 수 있다. 이를 통해 Account를 개설하지 않아도 누구나 거래를 이용할 수 있도록 함과 동시에 데이터 처리량이 적은 Account에 낮은 수수료를 부과함으로써 처리 효율을 향상시킨다.

UTXO

UTXO 모델에서 UTXO는 하나의 코인 덩어리를 의미하며 개인이 보유한 키를 이용하여 해당 코인 덩어리를 합치거나 나누거나 소유권을 변경하는 형태이다. 플레타에서는 UTXO를 Public Key를 통해 만들어지는 Public Hash를 이용하여 소유권을 나타낸다. Public Hash는 Public Key의 1 Byte의 Checksum과 32 Bytes의 SHA256 Hash로 구성되어 Public Key를 직접적으로 노출하지 않고 사용할 수 있도록 한다. 만약 사용자가 거래마다 새로운 Key를 생성하여 사용한다면, UTXO가 가지는 Privacy 강점을 이용할 수 있다. 또한 Account를 개설하지 않아도 사용 가능하다.

Account

Account 모델에서 Account는 특정한 공개키/비밀키로 접근 가능하고 코인이 Balance로 기재되어 이를 다른 Account로 보내거나 받아 Balance를 증감시키는 형태이다. Account는 개설해야만 사용할 수 있으며, 처리량이 적은 만큼 UTXO보다 거래 수수료를 적게 발생시킨다. 플레타는 여러 종류의 Account를 지원할 수 있도록 개발되었으며 SingleAccount, MultiSigAccount, FormulationAccount, ContractAccount 등 다양한 기본 Account를 제공한다. Account의 종류는 각 DApp이 자체적으로 확장해서 쓸 수 있으며, 자세한 사항은 Architecture의 Account Extension에서 확인 가능하다. 플레타 메인 체인에서 개설된 계좌 주소는 모든 서브 체인에서 동일하게 개설되며, 이를 통하여 모든 코인과 토큰을 하나의 주소를 통해 입출금을 처리할 수 있다.

Account Address

Account Address는 Account 개설 Transaction이 포함된 좌표(블록높이, 블록 내 거래 위치)와 현재 Chain 개설 Transaction이 포함된 좌표, 그리고 Contract에서 사용하는 nonce 값으로 구성된 Bytes를 base58을 이용하여 표현한다. 이를 구조적으로 서술하면 아래와 같다.

Address: Base58({AccountCoordinate(6), ChainCoordinate(6), nonce(8)})

플레타 메인 체인의 경우 ChainCoordinate는 (0, 0)를 가지고, Solidity 등의 Contract에서 코드에 의해 만들어지는 Sub-Contract를 제외하면 nonce는 0을 가지게 된다. 따라서 메인 체인의 일반 주소는 base58을 통해서 9자리의 Address를 가진다. 해당 주소 체계에서는 모든 Account는 중복될 수 없으며, 상대적으로 짧은 주소로 거래를 수행할 수 있다.

Transaction

Transaction은 현재 체인 상태를 변경하는 명령어로 작용하며 이는 Account, AccountData, UTXO를 변경할 수 있음을 의미한다. 플레타는 여러 종류의 Transaction을 지원할 수 있도록 개발되었으며, CreateAccountTransaction, CreateMultiSigAccountTransaction, FormulationTransaction, RevokeFormulationTransaction, TransferTransaction, BurnTransaction, DepositTransaction, WithdrawTransaction 등 다양한 기본 Transaction을 제공한다. 이러한 Transaction을 통해 Account를 생성하고 코인을 송금하며, Contract를 실행하여 AccountData를 변경하고 UTXO를 변경한다. 또한 Account에서 UTXO로 출금하거나 UTXO에서 Account로 입금하는 것도 가능하다. Transaction의 종류는 각 DApp이 자체적으로 확장해서 쓸 수 있으며, 자세한 사항은 Architecture의 Transaction Extension에서 확인 가능하다.

Contract

Contract는 Account와 연결되어 지정된 코드를 동일한 입력을 이용하여 실행하여 AccountData를 동일하게 변경하는 것을 의미한다. 플레타에서는 여러 종류의 Contract를 지원할 수 있도록 개발되었으며, Solidity, RDBMS, NoSQL, Event Sourcing 등 다양한 기본 Contract를 제공한다. 이 중 Solidity의 예를 들어보면, 이는 AccountData를 Solidity와 이를 구동시키는 EVM(Ethereum Virtual Machine)을 이용하여 변경하는 것이며, 동일한 블록까지 전진한 모든 체인은 동일한 AccountData를 가지므로 항상 동일한 Solidity 실행 결과를 나타내게 된다. Contract의 종류는 각 DApp이 자체적으로 확장해서 쓸 수 있으며, 자세한 사항은 Architecture의 Contract Extension에서 확인 가능하다.

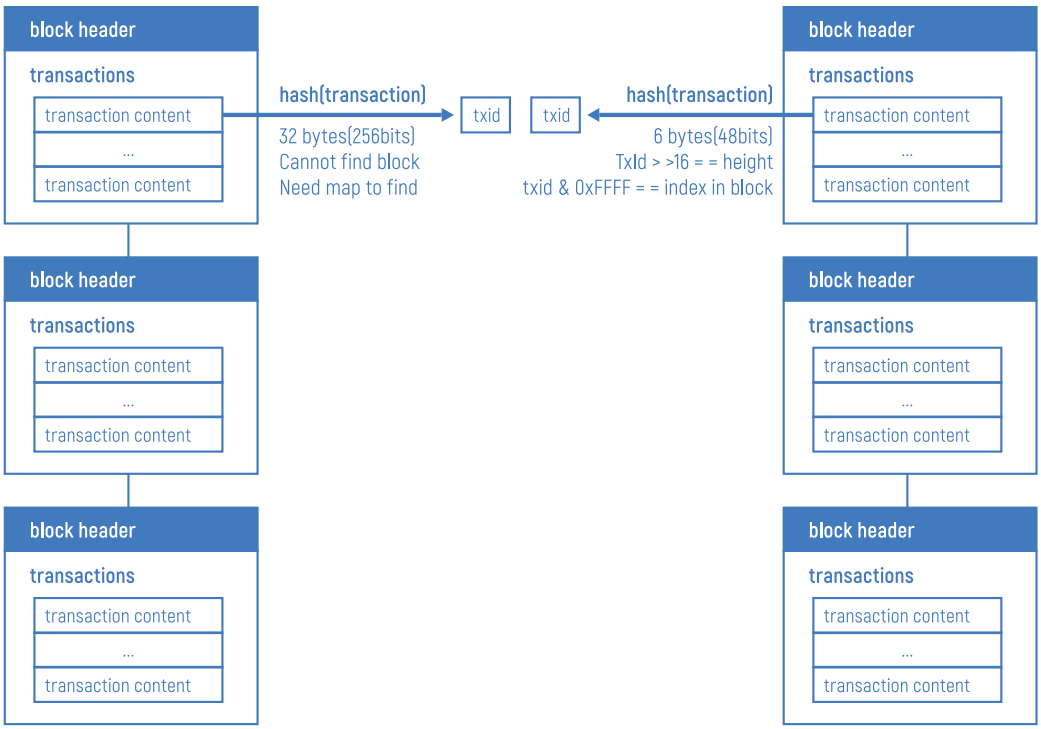
TxId in UTXO

UTXO 모델에서 기존에는 TxId를 거래에 대한 Hash를 사용하고 서명은 이 TxId를 이용해서 수행하였다. 하지만 해당 방식은 TxId로 바로 블록 및 거래 위치를 알 수 없어서 전용 Index가 필요하고 해당 Index의 용량 또한 커지고 동시에 Vin에 들어가는 값 또한 32 바이트로 큰 값이 들어간다.

플레타에서는 거래의 크기를 줄이고 블록의 위치와 거래의 위치를 바로 추적할 수 있게 하기 위하여 TxId의 형태를 6 Bytes의 아래 구조로 구성하였다.

TxId : {Height(4), Index(2)}

상기 구조는 블록의 높이를 표현한 4 Bytes의 Height, 블록 내 트랜잭션 위치를 표현한 2 Bytes의 Transaction Index으로 구성되어 있다. 이를 이용하면 32 bits의 Shift Operation만으로 쉽게 블록 높이를 구할 수 있게 되며, 16 bits의 Shift Operation과 Bits Masking을 통하여 블록 내 Transaction의 위치를 구할 수 있다. 모든 거래의 참조는 완성된 블록체인을 이용해서 이루어지므로 거래의 Hash를 사용하지 않더라도 모두 동일한 값을 가지게 된다.



Block

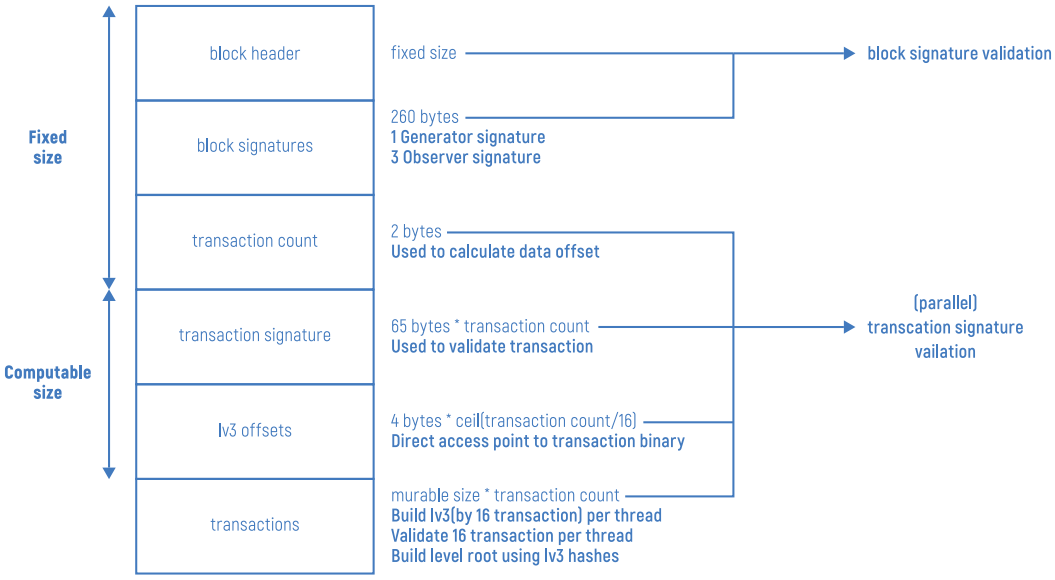
Block은 Header와 트랜잭션 그리고 서명으로 되어있으며, 기존의 Header에는 이전 블록의 거래와 TxId를 이용한 머클 트리 루트 해시가 들어가 있다. 머클 트리는 비효율적인 계산 구조를 가지고 있고 단순한 거래 목록은 검증 및 Light Node 데이터 교환이 어려우므로 머클 트리를 제거하고 레벨 트리를 추가하였다. 머클 트리는 P2P에서 데이터 전송 시 변경 점을 알기 위해서 사용하는데 이를 위해서는 트리 전체 보관이 반드시 필요하다. 하지만 실제로는 블록을 단일 노드에서 통째로 받아오고, 머클 트리 크기가 TxId 전체 목록과 거의 동일한 용량을 가지므로 P2P 데이터 전송을 이용한 부분합 검증에 사용되기 어려움에 따라서 단순히 다른 각도로 SHA256을 추가 수행한 것과 다름이 없다.

플레타에서 블록은 블록 헤더와 트랜잭션 목록으로 구성되어 있으며 Light Node 및 병렬 처리를 지원하기 위해 Level 구조를 사용한다. 이에 기본적은 블록 구조는 아래와 같다.

Block: {BlockHeader, TransactionSignature[], Transaction[], BlockSignature}

BlockHeader: {Version, HashPrevBlock, HashLevelRoot, Timestamp, Timeout, FormulationAddress}

먼저 Level 구조는 거래 Hash를 16개씩 묶고 해당 묶음의 해시를 다시 사용하는 16진수 트리이다. 즉, 한 블록에 최대 기재 가능한 Transaction의 수를 65535개로 잡고, 각 Level 단계마다 16개의 자식을 가지도록 한 트리 구조이며 따라서 레벨은 Lv1, Lv2, Lv3이 존재한다. 블록 헤더에는 16개의 Lv1을 이용한 HashLevelRoot가 기재되고, Lv1은 16개의 Lv2를 이용한 Hash 값이고, Lv3은 16개의 Transaction Hash를 이어 붙여서 이를 Hash한 값을 사용한다. 여기서 Hash를 구성할 때에는 해시 값들 사이에 지정된 패턴의 1 Byte Padding을 삽입하여 연결한 값을 Hash하는 함수인 HashFunciton(Hash1+1Byte+Hash2+1Byte… +1Byte+Hash16)를 이용하여 속도를 향상시키고 변조 가능성을 낮추도록 하였다.



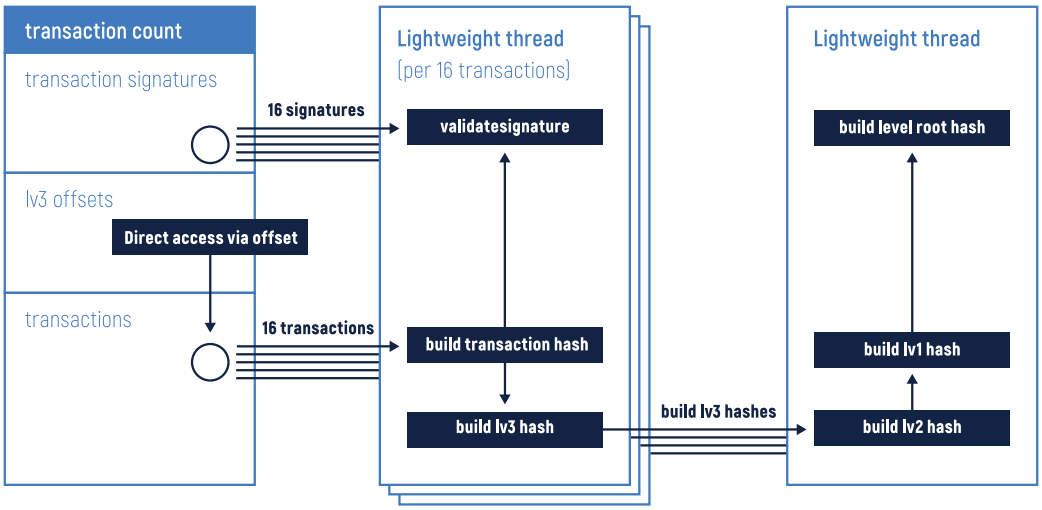
다음 블록의 Serialization은 블록을 빠르게 검증하고 병렬로 검증할 수 있도록 설계되어 아래와 같은 구조를 가지고 있다.

BlockSerialization: {BlockHeader, BlockSignature, TransactionCount, Level3Indexes, TransactionSignatures, Transactions}

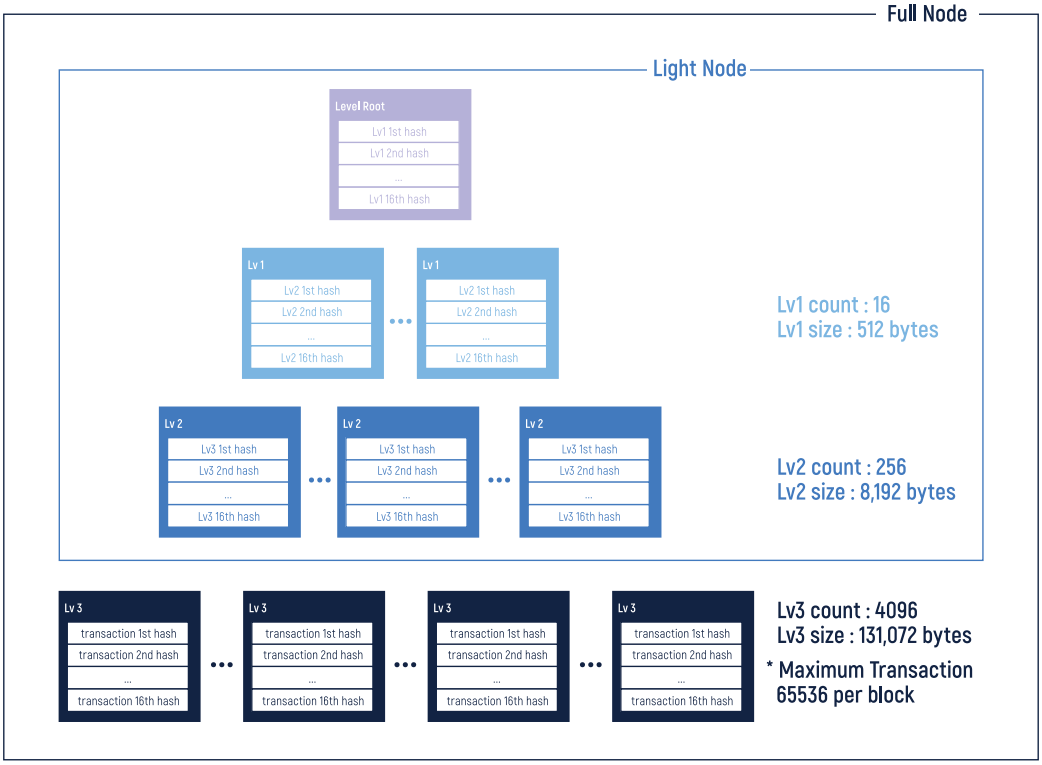
BlockSignature : {CreatorSignature, Signatures[9]}

먼저 BlockHeader와 TransactionCount는 고정 사이즈이며, Level3Indexes과 TransactionSignature는 TransactionCount에 비례하는 고정 사이즈이다. 이는 Transactions의 binary 위치를 바로 확인할 수 있는 것을 의미하며 Leve3Indexes를 이용하면 16개 단위로 뭉쳐진 Transaction의 위치를 바로 알 수 있어 병렬로 빠르게 binary 데이터를 가져가서 바로 검증을 수행할 수 있다. BlockSignature는 BlockHeader에 대한 Hash 값을 이용한 서명으로서, 생성자의 서명과 블록 생성 그룹 및 옵저버 노드의 서명으로 구성되어 있다. 블록 생성은 블록 생성자가, 보상 순서는 생성자 및 블록 생성 그룹이 하고 내용 승인은 옵저버 노드에서 하게 된다. 여기서 옵저버 노드의 서명이 완료되어 완전히 검증된 블록도 TransactionSignatures를 같이 기재해서 전송하고, 개별 노드에서도 거래와 서명을 모두 검증하므로 잘못된 거래는 형성될 수 없다.

Validation



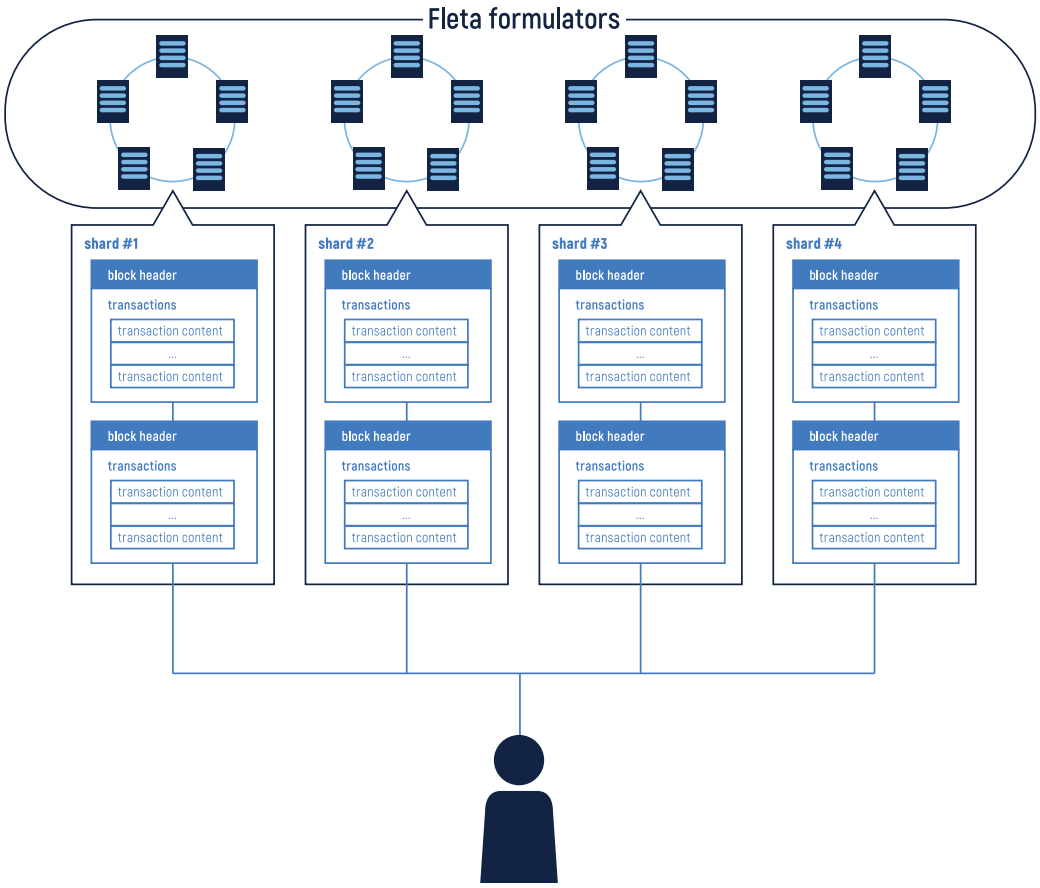
검증은 우선 Transaction의 Hash로 Level을 구성하고 이를 HashLevelRoot와 비교하면 거래 내용에 대한 전체적인 정합성을 알 수 있으며, 서명에 대한 검증은 16개 단위로 TransactionSignatures를 나누고 Level3Indexes를 이용하여 Transaction을 16개 단위로 쓰레드로 나누어 처리하면 Signature와 Transaction, 그리고 Level을 병렬로 비교해서 확인할 수 있다. 모두 읽기 작업에 해당하므로 동시에 진행될 수 있어서 블록을 수신하면 Transaction을 16개 단위로 검증하고 검증하면서 발생한 Hash 값을 이용하여 Lv3를 구성하고 이를 모아서 진행하면서 Level 트리 검증까지 할 수 있다.



Level 구조는 Light Node의 검증에서도 유리한데 우선 Light Node는 각 Level 1은 512 Bytes, Level2는 8192 Bytes로써 Light Node는 블록 헤더를 포함하여 8880 Bytes로 한 블록에 대한 검증 데이터를 보관할 수 있고, 특정 Transaction을 받아오는 것이 필요하면 TxId에서 Height가 나오므로 블록을 바로 알 수 있고, Index에서 위치가 나오므로 Level 트리의 어떤 Node에 Transaction이 담겼는지 바로 알 수 있다. 이를 통하여 Level3의 한 트리에 해당하는 512 Bytes와 평균 3600 Bytes를 가지는 16개의 Transaction을 가져오면 트리 구조를 통해 내용을 검증할 수 있다. 이를 통하여 경량 Node는 적은 데이터 수신으로 높은 수준의 Transaction 검증을 수행할 수 있다.

Sharding (Multinode Parallel Transaction Processing)

샤딩은 크게 데이터를 나누어 보관하는 방식과 거래를 나누어 처리하는 것이 존재한다. 아래 기술되는 샤딩은 여러 노드를 이용하여 트랜잭션을 병렬로 처리하는 기술을 의미한다.



Multiple shard chain with one address

Sharding

플레타에서는 각 샤드를 독립적인 블록체인으로 취급하며 따라서 각 샤드는 완전히 병렬적으로 동작한다. 대신 한 사용자의 공개키 및 비밀키는 샤드와 상관없이 동일한 값을 사용함으로써 물리적으로는 분리된 블록체인의 거래를 한 주소에 대해서 집계해서 보면 하나를 소유한 것처럼 보고 사용할 수 있다. 이러한 방식을 이용하여 이중지불이 원천적으로 불가능한 완전 병렬 샤드를 구성하였다. 사실 상 플레타 코인으로 인정되는 여러 개의 체인에 접근하는 방식이며, 샤드 데이터 보장을 위해 샤드 헤더를 저장하는 헤더 체인이 별개로 구동되어 공유된다.

INDEPENDENT MULTICHAIN STRUCTURE

플레타는 독립 멀티체인 구조를 지원하는 블록체인 서비스 플랫폼이다. 이는 각 DApp이 개별 체인을 구동시키며, ICO 등을 위한 토큰 추가 발행과 Interchain 기능을 사용하는 경우에만 메인 체인과 연동되어 동작한다. 즉, 해당 기능을 제외하고 각 DApp 체인은 완전히 독립적으로 체인을 유지하고 해당 DApp 토큰을 이용해서 독립적으로 Formulator를 구성함으로써 메인 체인이 정지한다 하더라도 동작하도록 구성된다.

Token Chain

Token Chain

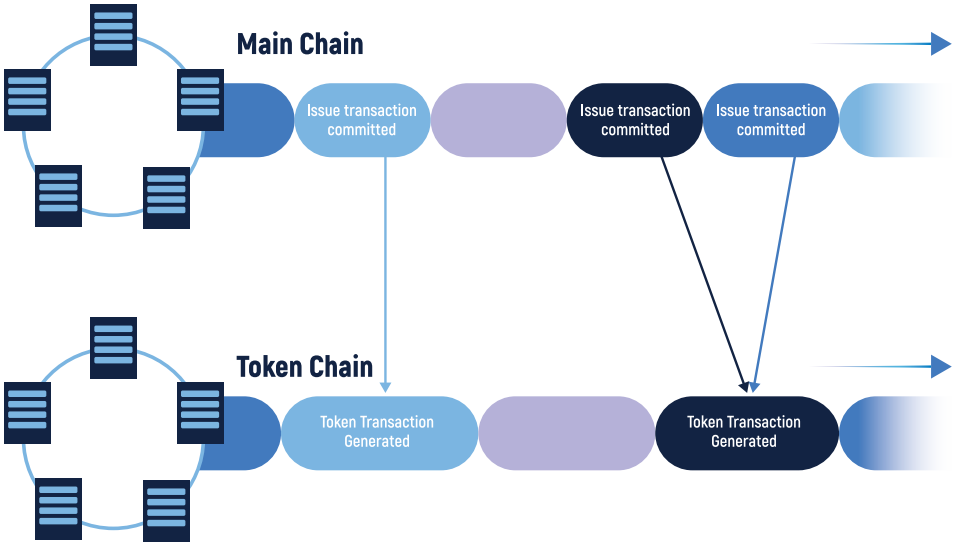
플레타의 DApp은 개별 토큰을 발행해야 하며, 발행 시 독립적인 Token Chain을 구성할 수 있게 된다. 토큰의 전체 수량, 초기 계좌 구성, Observer Node 공개키, 시드 노드 IP, 락업, 토큰 판매 등 Genesis 정보를 설정하여 TokenCreationTransaction을 수행하게 되면 Token Account가 개설되며, 해당 정보를 이용하여 Token Chain 노드의 설정 값을 지정하고 네트워크를 구성할 수 있다. 이러한 초기 구성이 완료되면 해당 체인은 메인 체인과 분리되어 블록이 진행되고 동작한다. DApp의 Smart Contract는 모두 해당 Token Chain에서 동작하게 되며, 이를 통해 서로 다른 DApp이 같은 영역을 사용하면서 발생하는 수수료 문제나 부하 누적 문제 등을 해결한다. Token Chain은 토큰 발행과 Interchain 기능을 위하여 메인 체인의 블록을 가져오게 되는데, 이 때 해당 Token Chain에서 처리해야하는 TokenIssueTransaction 등을 처리하여 블록 헤더에 처리된 위치까지 기록하고 토큰을 실제로 발행하는 작업을 수행한다.

Token Issue

Token Issue

TokenIssueTransaction은 생성된 토큰의 초기 값에 토큰 판매 정보가 있는 경우 사용 가능하다. 사용자가 Token Account에 TokenIssueTransaction을 통해 토큰을 입금하게 되면, 해당 판매 정보를 통해 이를 검증하고 승인한다. Token Chain은 메인 체인의 거래 중 해당 Token Chain에서 처리해야하는 정보를 처리하므로, 이 때 실제 Token을 발행하게 된다.

정리하면, Token Issue를 통해 플레타를 토큰으로 교환하는 메커니즘을 제공하며, 이 때 토큰으로 교환된 플레타는 토큰 관리자 주소로 들어가고, 토큰 생성에 대한 TokenIssueTransaction이 발행된다. Token Chain은 발행된 Issue Transaction을 확인하여 해당 금액을 입금한 주소에 지정된 토큰을 생성하여 제공하게 된다.



Interchain Communication

플레타의 DApp들은 기본적으로 모두 독립된 블록 체인으로 구동된다. 따라서 해당 DApp 체인 간의 통신을 지원하려면 인터체인 기술이 필요하다. 해당 기술은 DApp 체인이 해당 체인의 블록 헤더를 주기적으로 메인 체인에 보고하여 기재함으로써 이루어지며, 이를 통해 한 DApp에서 다른 DApp으로 토큰을 이전할 수 있다. 이전된 토큰은 완전히 해당 체인에서 삭제되고 이를 받는 체인은 메인 체인에서 해당 토큰의 최신 블록 헤더 정보를 가져오고, 해당 체인에 Light Node로 접근하여 블록을 받아 토큰 이전 작업을 완료하여 토큰을 생성한다. 이는 하나의 체인이 여러 종류의 토큰을 보유할 수 있음을 의미하며, Smart Contract 또한 여러 종류의 토큰으로 실행할 수 있음을 의미한다. Interchain 기능은 DApp이 다른 토큰 전송받는 것을 허가한 경우에 한해서 허가된 토큰만 전송이 가능하며, 이 때 거래 수수료와 지불 토큰 등을 모두 설정하게 된다. 이러한 허가는 Token Account 개설자에 의해 이루어지며, TokenAllowanceTransaction을 해당 TokenAccount에 발행함으로써 이루어진다.

CONSENSUS : Proof-of-Formulation

U.S.P(United States Patent) Application Number : 62717695

컨센서스는 블록을 생성에 대한 합의를 의미하며 체인 진행에 있어서 다음 블록을 누가 생성하는지 또는 생성된 블록 중 어떤 블록을 선택하는지 합의하는 것이다. 기존에는 난이도 기반으로 합의하여 네트워크 전체에 블록을 전파시켜 임의의 사용자가 채굴이 가능하도록 지원하는 방식을 사용하였다. 이는 최신 블록이 네트워크 전체에 전파가 이루어져야 채굴자가 다음 블록을 만들 수 있으므로 높은 블록 타임 또는 높은 Confirmation 회수를 요구한다. 이를 해결하기 위하여 소수 채굴자를 선발하는 방식은 낮은 블록 타임을 달성하였지만, 소수만 채굴에 참여할 수 있다.

플레타에서는 PoF(Proof-Of-Formulator)를 개발하여 Formulator 보상 순서를 이용하여 채굴을 할 대상을 정하여 전파 범위를 좁혀서 빠르게 블록을 생성하고 전파할 수 있도록 하였으며 옵저버 노드를 두어 즉시 승인을 처리하고 블록의 Fork를 방지하는 방법을 사용한다. 누구나 Formulator를 만들 수 있으므로 자유로운 참여가 가능하며, Formulator의 채굴 순서가 정해져 있으므로 최신 블록의 전파 범위가 매우 적어 낮은 블록 타임을 달성할 수 있다.

Rank Table

RankTable은 모든 FormulationAccount에 대해서 점수를 산출하고, 해당 점수를 통해서 순위를 메기는 기능을 수행한다. 모든 노드는 RankTable을 보유하고 있으며, 점수 공식은 거래와 체인 높이에 의해 결정되는 값을 사용하므로 동일한 목록을 가지게 된다. 새로운 블록을 생성할 권리는 순위가 가장 높은 Formulator에게 주어지며, 블록을 생성하고 반영하면 점수가 바뀌므로 순서가 변경되어 차례가 변경되게 된다.

RankTable에서의 점수는 Phase와 Hash로 구성되어 있다. Phase는 시간에 관련된 값으로 RankTable이 몇 번 회전했는지를 나타낸다. 새로운 Formulator는 항상 LargestPhase+1의 값으로 RankTable에 참여하게 되고 블록을 생산하고 나면 해당 Formulator의 Phase를 증가시켜 합리적인 순서를 제공한다. 상세한 공식은 아래와 같다.

Score : uint64(Phase) << 32 + uint64(binary.LittenEndian.Uint32(hash[:4]))

해당 공식을 통하여 모든 Phase에 모든 Formulator가 1번의 채굴 기회를 갖도록 보장하며, 서로 다른 Phase에서는 서로 다른 Formulator 순서를 가지게 함으로써 Formulator에 의한 공격이나 담합을 방지한다.

Connectivity

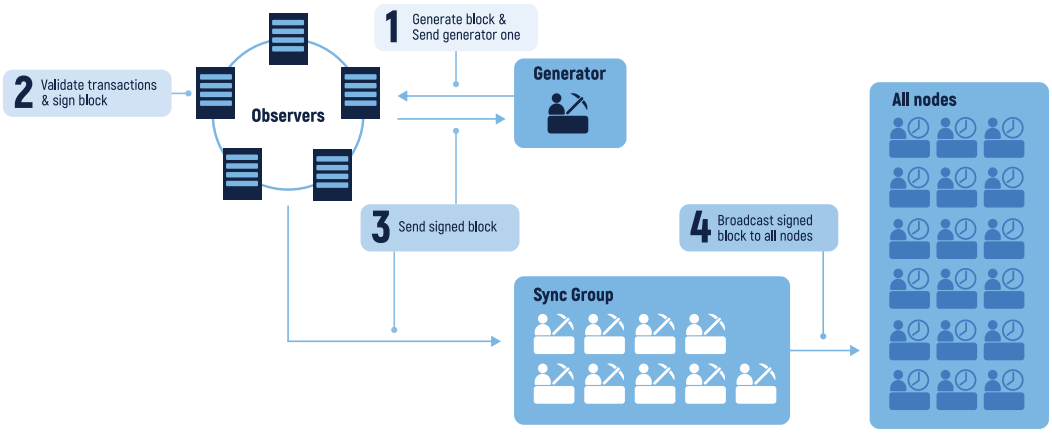
DDoS 공격에 대비하여 Formulator에 대한 IP를 감추면서 유기적으로 순서를 할당받고 처리하기 위해 모든 Formulator는 옵저버 노드에게 접속한다. 따라서 모든 DDoS 방어와 보안 비용은 옵저버 노드가 담당하게 되며, 옵저버 노드는 상대적으로 적은 숫자 로 구성되어 있으므로 보다 적은 유지비용으로 효과적으로 방어할 수 있다. 따라서 옵저버 노드는 실시간으로 Formulator의 활동 정보와 여부를 알 수 있게 되며, 노드 현황과 구조 정보를 Formulator와 사용자에게 공개함으로써 투명하게 운영할 수 있다. 연결이 되지 않은 노드 차례가 오는 경우 TimeoutCount를 통해 해당 노드들을 제외하고 채굴을 진행할 수 있게 되며, 자기 차례가 넘어간 Formulator는 이를 인지할 수 있게 되어 사용자가 쉽게 모니터링 할 수 있다.

Block Generation

블록의 생성은 Formulator 간의 합의된 블록 생성 순서에 따라 진행되게 되며, 이 때 생성한 블록의 보상을 블록을 생성한 Formulator가 받으므로 보상과 직결되는 부분이다. 앞서 기술된 Connectivity를 이용하여 Formulator가 연결되고 RankTable을 이용하여 블록 생성 순위를 동기화하게 되는데, 이를 통하여 블록 생성 순서를 동일하게 합의하게 된다. 또한 블록 생성은 1순위만 가능하므로 서명이 들어가므로 포크 생성이 가능한 노드는 최상위 노드만 가능하므로 이를 방지하면 포크가 발생할 수 없다.

채굴자 그룹은 1순위는 채굴자, 2순위부터 10순위까지는 동기화 그룹으로 구성된다. Connectivity에서의 설명처럼 DDoS 방어를 위하여 모든 연결을 옵저버 노드가 중계하므로 그룹은 옵저버 노드 내부에 설정된다. 채굴자는 블록을 생성하고 생성한 블록을 옵저버 노드에 전송하고, 옵저버 노드는 블록의 거래 및 서명을 모두 검토하고 옵저버 노드 간에 서명을 교환한다. 총 5개의 옵저버 노드 중에서 3개의 서명이 모이면 블록이 완성되며, 옵저버 노드는 완성된 블록을 동기화 그룹에 빠르게 전파하여 다음에 있을 채굴에 대비하게 한다. 동기화 그룹은 완결된 블록을 빠르게 검증 및 연결하고 이를 대기 그룹에 전파한다.

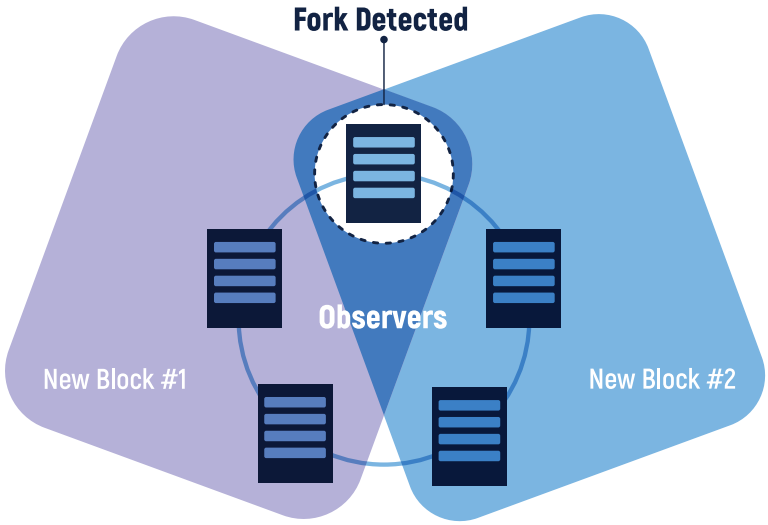
대기 그룹은 받은 블록을 네트워크에 배포하게 된다. 이를 통하여 블록 생성자는 빠르게 블록을 생성하고 옵저버 노드의 3/5에 의해 서명되므로 최소 1개의 옵저버 노드에 의해 포크가 발견되므로 포크를 할 수 없다. 그리고 모든 노드가 RankTable을 통해서 순서 검증 진행하므로 옵저버 노드가 편향적으로 서명하는 것을 방지한다. 또한 생성한 블록을 동기화 그룹이 네트워크에 전파함으로써 전송 트래픽을 분할하고 더 넓고 빠르게 네트워크에 블록이 퍼질 수 있도록 지원한다.



보다 빠른 채굴을 위하여 1순위는 옵저버 노드 외에도 2순위에게 블록을 보내서 2순위를 미리 준비하게 할 수 있다. 물론 1순위가 제공한 블록이 문제가 있거나 서명에 실패하면 준비한 블록을 버리고 새로 준비해서 전달한다. 해당 방법은 문제가 없는 상황에서 빠르게 서명을 수행할 수 있게 보조하는 역할을 한다. 그리고 만약 1순위가 1초 이상 정상적인 블록을 생성하지 못하는 경우에는 2순위는 새 블록 생성을 해두고 3초 이상 정상적인 블록을 생성하지 못하면 이를 바로 전파하는 과정을 거쳐서 블록 생성을 이어간다. 옵저버 노드는 1순위가 3초 이상 만들지 못함을 확인하고 서명 프로세스를 진행한다.

Fork Prevention

최상위 Formulator가 블록을 생성하고 옵저버 노드의 서명을 받으면, 옵저버 노드는 해당 블록을 서명하고 저장하고 다른 옵저버 노드로부터 서명을 받아 전진시키므로 포크 블록이 발생하더라도 옵저버 노드를 통과할 수 없기 때문에 포크가 발생할 수 없다. 해당 개념은 Formulator 순서가 올바르게 구성되어 있는 상황에서 1순위가 블록을 생성하고 서명할 권한을 가지게 되는데, 이 때 두 개 이상의 블록을 만들어서 블록체인을 포크하는 것을 옵저버 노드 3/5 서명을 받아서 불가능하게 만드는 것이다. 따라서 Formulator 순위가 동기화되면 블록 생성자와 옵저버 노드 서명을 검증하는 것만으로도 포크가 되지 않은 블록을 받을 수 있다. 이는 만들어진 블록이 확정성을 가지며 이에 따라 옵저버 노드를 거쳐서 승인된 모든 블록의 거래는 즉시 승인이 된다.



3/5 sign prevent chain from forking

옵저버 노드에 의하여 공격자는 이중지불을 유도하는 포크 블록을 생성할 수 없게 된다. 그리고 블록 생성의 주체는 Formulator이므로 블록체인 유지는 Formulator를 생성한 개인이 담당하고 보상도 옵저버 노드는 보상을 받지 않으므로 해당 개인이 받아가게 된다.

ARCHITECTURE : Microkernal System

플레타의 시스템 아키텍쳐는 Microkernal 구조이다. 각 체인이 서로 다른 Account와 Transaction 그리고 Contract를 지원할 수 있도록 Kernel은 Consensus, Store, Generator 등 블록체인을 구성하고 진행하는 기능을 담당한다. 만약 DApp 새로운 Account나 Transaction 그리고 Contract를 추가하고 싶다면, accounter.RegisterAccount와 같은 함수에 정해진 인터페이스에 맞는 생성, 검증, 실행 방식을 정의하면 해당 타입이 추가되게 된다. 소스 코드가 아닌 플러그인 형태로도 추가할 수 있도록 격리되어 있으며, 등록된 타입에 대하여 각 체인마다 필요한 타입만 선택하여 넣을 수 있도록 개발되었다.

Microkernal식 접근의 근간은 플레타가 블록체인을 바라보는 관점을 보여준다. 블록체인은 서로의 데이터를 P2P 등으로 교환하는 과정, 모두가 같은 History를 보관하고 이를 Hash Chaining을 통해서 검증하는 과정, Digital Signature를 통해 데이터에 대한 생성/변경/삭제에 대한 권한을 관리하는 과정, 그리고 Consensus를 통해 History를 전진시켜서 데이터 변경을 수행하는 과정으로 구성되어 있다. Microkernal은 실제로 변경하는 수식에 해당하는 Transaction과 이를 보관하고 관리하는 Account, 그리고 Code 등으로 작업을 수행하는 Contract를 추상화하고 이 외의 나머지 과정을 모두 처리하는 구조이다. Transaction은 Context를 통해 Account, AccountData, UTXO를 변경하게 되며, 이에 맞추어서 Account와 Contract가 실행되는 구조이다.

Context based Approach

Transaction을 처리하는 Transactor는 Vaidate와 Execute를 Transaction이 정의할 수 있도록 한다. 그리고 해당 Transaction은 Context를 통해 Account, AccountData, UTXO를 Snapshot, Revert, Commit 기능을 활용하여 조회하고 변경하고 복구하는 과정을 거치고 해당 Context를 Store에 반영하여 현재 체인 상태를 갱신하는 구조로 되어 있다. 이러한 접근에서 Context는 메모리에서만 동작하고 Store에는 FileSync Mode로 기록함으로써 처리는 빠르게 하고 데이터가 중도 처리에서 멈추거나 Corruption 되는 것을 방지한다.

Context based Approach는 하나의 Transaction 타입이 적층되어 처리되고 필요에 따라 이를 해석하는 구조가 아니라 필요에 따라 여러 타입의 Transaction을 처리하고 이를 통해 Store를 업데이트 할 수 있음을 의미한다. 는 플레타의 DApp이 블록체인이라는 환경 아래서 기반 처리는 Microkernel에 맡기고, Context를 제어하는 추가적인 기능을 것만으로도 쉽게 새로운 기능을 개발할 수 있음을 의미한다. 이를 통하여 플레타와 플레타 DApp이 블록체인 기반으로 서비스를 쉽게 개발하고 확장할 수 있도록 한다.

Account Extension

Account는 데이터를 보관하고 관리하는 주체로써 Accounter를 통해 관리되며, 새로운 유형을 Global로 추가하는 Register 과정과 이를 실제 사용하기 위해 해당 체인 용 Accounter Instance를 만들고 Global로 등록한 타입을 해당 Instance에 세부 값, 수수료 값 등과 함께 기록하는 과정으로 나누어 진다. Formulator를 위한 FormulationAccount도 이러한 방식으로 추가되어 있으며, 기본 거래를 수행하는 SingleAccount와 MultiSigAccount 등도 이러한 방식으로 추가되어 있다.

만약에 Coin에 대한 Amount를 관리하는 Account가 아니라 고유한 데이터를 보유하고 거래하는 Account를 만들고 싶다면, Account 타입을 추가하여 Data Map을 구성하고 DataCreationTransaction 타입과 DataTransferTransaction 타입을 추가하면 이를 달성할 수 있다. Account를 SNS와 같은 구성으로 처리하고 싶다면, 새로운 Account 타입을 추가하고 FriendRequestTransaction, FriendResponseTransaction, SubmitPostTransaction와 같은 Transaction 타입을 추가하여 개발할 수 있다.

Transaction Extension

Transaction은 데이터를 변경하는 명령어로써 Transactor를 통해 관리되며, 새로운 유형을 Global로 추가하는 Register 과정과 이를 실제 사용하기 위해 해당 체인 용 Transactor Instance를 만들고 Global로 등록한 타입을 해당 Instance에 세부 값, 수수료 값 등과 함께 기록하는 과정으로 나누어 진다. FormulationAccount를 생성하고 삭제하는 FormulationTransaction과 RevokeFormulationTransaction도 이러한 방식으로 추가되어 있으며, 기본 거래를 수행하는 TransferTransaction이나 소각하는 BurnTransaction 등도 이러한 방식으로 추가되어 있다. 만약 Transaction이 Key/Value Store인 AccountData를 넘어서는 처리를 해야 한다면, 새로운 Account 타입을 추가하여 처리할 수 있다.

Contract Extension

Contract는 Virtual Machine과 같은 형태로 사용자 정의 Code를 처리하는 처리기로써 Contractor를 통해 관리되며, 새로운 유형을 Global로 추가하는 Register 과정과 이를 실제 사용하기 위해 해당 체인 용 Contractor Instance를 만들고 Global로 등록한 타입을 해당 Instance에 세부 값, 수수료 값 등과 함께 기록하는 과정으로 나누어 진다. Solidity를 실행하는 SolidityContract도 이러한 방식으로 추가되어 있으며, RelationDatabaseContract도 이러한 방식으로 추가되어 있다. 새로운 언어를 지원하거나 외부 소프트웨어와

Context based Approach
Account Extension
Transaction Extension
Contract Extension

연계된 처리가 필요한 경우 이를 이용하면 된다. 외부 소프트웨어는 Snapshot, Revert, Commit과 같은 상태 관리 기능을 지원해야 하며, 해당 Contract에 의한 접근에 의해서만 변경되어야 한다.

SUPPORTED CONTRACT

플레타는 기본적으로 Solidity, Relational Database, Event Sourcing 타입의 Contract를 제공한다. 이 후 기술 개발 및 연구 진행에 따라 새로운 유형의 Contract가 제공될 수 있다.

Solidity

Solidity는 EVM(Ethereum Virtual Machine)을 통해 제공된다. EVM에서 Trie 기반의 Store로 구성된 부분을 Context 기반의 접근으로 변경하여 구성되었으며, 하드포크로 인해 갈라지는 코드를 모두 제거하고 최신 기준으로 제공하도록 변경되었다. 일부 특수 명령어가 삭제되거나 추가될 수는 있으나 대부분 Contract는 코드 변경 없이 적용 가능하다.

Relational Database

Relational Database는 Database Transaction 기능을 활용하여 제공된다. 우선 Begin, Rollback, Commit으로 대표되는 Transaction 기능을 통하여 실행 상태를 관리하고 체인에 반영되는 경우에만 Commit을 실행하며, Nested Transaction을 이용하여 개별 상태를 나누어 처리하는 방법을 사용한다. 이를 통하여 해당 Database를 변경하는 코드와 함께 Contract를 구성하면 블록체인에 의한 변경이 이루어지게 되며, 조회는 Database에 직접 연결하여 기존 시스템 개발과 같은 형태로 사용할 수 있다.

Event Sourcing

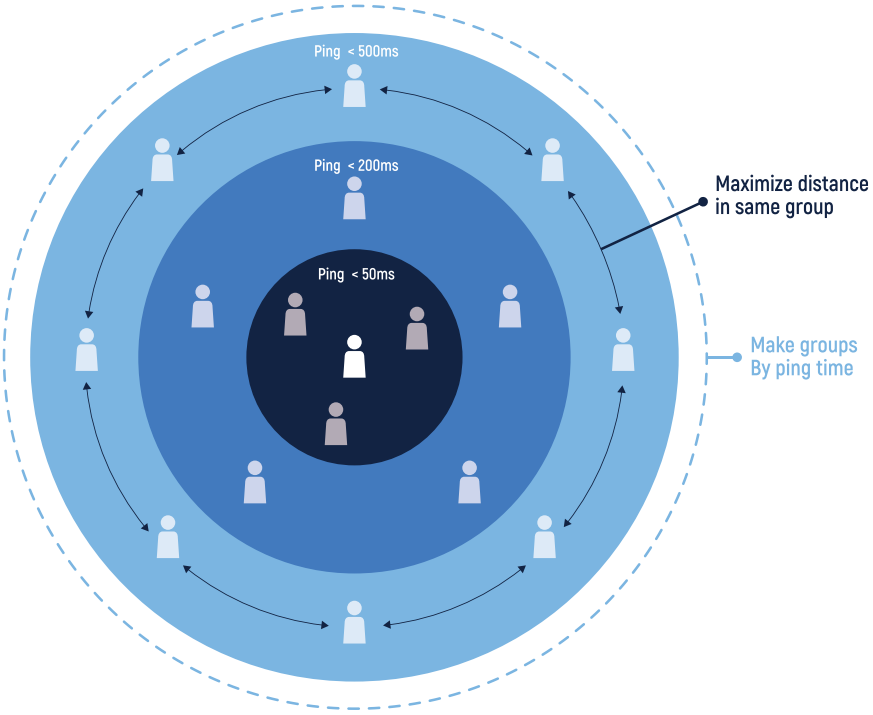
Event Sourcing은 Contract는 검토할 Event를 정하고 해당 Event가 Contract의 검증을 통과하게 되면 Commit되어 블록체인에 기록되는 방식이다. 블록체인에 기재된 이벤트는 각 노드로 퍼지게 되고, DApp을 제공하기 위한 Front End 서버에서는 해당 Event Stream을 Subscribe하여서 서비스 제공에 알맞은 형태의 스토리지를 구성하여 데이터를 제공할 수 있다. 그리고 특정한 동작에 의해서 Event가 발생하면 이는 Contract에 넘어가게 되고 Contract가 이를 검토하고 Commit하게 되는데, 이는 분산 환경에서 단일 지점 쓰기 검증 및 수행을 하는 것과 동일하므로 충돌 문제없이 Eventually Consistency를 유지할 수 있게 해준다. 그리고 새로운 Event를 처리할 때에는 새 Contract를 배포하기만 하면 되며, 기존 Event 처리기를 업데이트할 때에는 Migration을 거쳐서 업데이트 할 수 있게 된다. Contract는 공통된 Interface와 과정을 통해서 진행 및 검증만 가능하면 언어와 상관없이 유지가 가능하므로 현재 다양한 언어의 지원을 고려하고 있다.

NETWORK

Peer Algorithm : Geologically Balanced Peer Group

네트워크는 P2P 네트워크 형식으로 Peer 목록을 노드가 공유하는 방식으로 유지된다. 만약 Peer 목록이 없으면 Seed 노드에 연결하여 Peer 목록을 갱신하게 된다. 일반 노드 Peer와 Formulator Peer를 별개로 유지하고 해당 목록은 근접 노드 Peer와 원거리 노드 Peer를 골고루 유지하게 하여 망이 한쪽으로 쏠리는 문제를 최소화하고 전파속도를 극대화한다. 이를 위하여 새로운 Peer를 지속적으로 탐색하게 하고 허용범위 내에서 망 쏠림이 적어지는 Peer를 선택하여 Peer 연결을 갱신하도록 한다.

망 쏠림에 대한 부분의 해결은 자신이 ping을 통한 거리 유추를 하고 새로운 Peer를 받을 때 보낸 사람 기준의 ping을 받음으로써 전체적인 ping 거리가 골고루 퍼지도록 하여 수행한다. Ping 거리를 골고루 퍼지게 하는 방법은 Ping 값에 따라 그룹을 형성하고 해당 그룹에서 서로 간의 거리 차가 최대가 되도록 하는 방법을 사용한다. 이를 통해 거리가 가깝고 먼 노드를 골고루 배정하여 거리 쏠림을 감소시키고 비슷한 거리를 가진 Peer 중 서로의 거리가 먼 Peer를 선택하여 각도 쏠림을 감소시켜 망 쏠림을 해결할 수 있다.



Geological balanced peer group

Peer 간 연결 후 통신 방법은 TLS가 적용된 TCP/IP 통신을 사용하며, 패킷 프로토콜은 아래와 같다.

Packet Protocol : {MagicWord(8), Compression(8), Size(32), Payload, Integrity(*)}

이 방식을 이용하여 TLS를 통한 보안을 유지하면서 동시에 최소화된 오버헤드를 가진 통신을 수행할 수 있게 된다. Payload의 경우에는 길이 등을 이용하여 gzip을 통한 압축 설정을 할 수 있으며 압축된 경우에는 Compression의 값을 이용하여 확인이 가능하다. Size는 Payload의 크기를 의미하며 압축이 된 경우에는 압축된 Size를 의미한다. Integrity는 CRC나 Parity 부호, Hash 등 Payload의 내용을 검증할 수 있는 내용이 올 수 있으며 Optional한 값으로써 물리적으로 신뢰성이 낮은 Network에서 데이터 교환시에 사용된다.

THANK YOU

FLETA.io

Update 2018.10.17_v2.0