

Proskenion: Blockchain platform that enables to design incentive/consensus algorithm dynamically and freely. Draft v1.0a

Takumi Yamashita

2019 年 3 月 9 日

概要

本プロジェクトはコンテンツの供給者と観客の間に入るブロックチェーン Proskenion を開発した。Proskenion は観客席と舞台を額縁のように区切る構造物を意味する「プロセニウム・アーチ」のギリシャ語の語源から名付けられた。Proskenion は既存のブロックチェーン技術に加えて以下の機能を実現した。「プリミティブな命令セットの組み合わせで高い表現力を持つ」、「独自 DSL(= Proskenion Domain Specific Language) によりインセンティブ/合意形成のカスタマイズが容易」、「ハードフォークせずにインセンティブ/合意形成の仕組みを変えられる」。これにより非中央集権的なコンテンツの供給者を主体としたシステムを実現できるブロックチェーンプラットフォームを実装した。

1 Introduction

昨今では、BitCoin[1] や Ethereum[2] を始めとするブロックチェーン技術が台頭し空前の仮想通貨ブームを引き起こした。ブロックチェーン技術はビットコイン開発の過程で生まれ、ビットコインの取引を記録する分散型台帳を実現するための技術であった。分散型台帳技術はデータベースを一つのサーバで中央集権的に管理するのではなく複数のサーバで分散管理することで単一点障害耐性を持ったデータベースである。ブロックチェーンはそれに加え「ブロック」というデータの単位を生成し、チェーンのように連結していくことによりデータを保管する。このブロックは一つ前のブロックのハッシュ値を持っているためブロックに書き込まれたデータを改ざんするにはその場所から先頭までの全てのブロックを改ざんする必要がある。このため、ブロックチェーンでは改ざんに対する強い耐性を持っているとされている。分散台帳では複数のサーバが同一のデータを保持するために様々な合意形成アルゴリズムが考案されている。しかし、それらには未だ技術的課題がいくつも残されており実用に耐えうる代物ではなかった。代表的な問題としてスケーラビリティ、情報透過性、などがある。また、仕様変更に伴って発生するハードフォークにも問題があり Ethereum の合意形成アルゴリズムの移行は非常に大変な作業だった。

1.1 Precedent

ブロックチェーン技術の応用先として最も注目されているのは金融業界である。なぜなら、既存システムと対比したときにブロックチェーン技術を用いることで減少できるコストが大きく極めて有用性が分かりやすいからである。しかし、その分金融向けブロックチェーンの競合性は高く大手企業や研究開発系ベンチャーが

群雄割拠している。それに対してエンターテインメントの領域におけるブロックチェーンの利用は机上の空論の粋を出ておらず Web3.0 化を行う積極的なモチベーションに不足している。このことからエンターテインメント領域におけるブロックチェーンの利用は実用段階に達していない。そこで本プロダクト "Proskenion" はエンターテインメント領域への活用をメインの用途として構成され、様々なジャンルのコンテンツ配信に対応しさらにコンテンツの配信者を主体とできるようなシステムを目指して開発を行った。つまり、ブロックチェーン上のシステムをコンテンツの供給者が主体で扱えるような世界を創ることを目的とする。Proskenion は一般的なブロックチェーン特有の機能である非中央集権的、非改ざん性、単一障害点耐性に加え、ブロックチェーンの運営主体を自在且つ動的且つセキュアに定義できる仕組みとプリミティブなコマンドの組み合わせで高い表現力のオブジェクトを操作できる仕組みを導入することでこれを実現しようとした。

1.2 Purpose

本プロジェクトでは、コンテンツの供給者（以下クリエイターとする）のために高い表現力と合意形成アルゴリズムを汎用的に設計できるブロックチェーンを開発した。ブロックチェーン技術では、高い信頼性が求められる金融取引や重要データのやりとり等での利用が期待される技術であるが、その本質は非中央集権的な仕組みにある。ブロックチェーンシステムの最大の魅力は中央集権的な支配から解放されたシステムで人々が中〜大規模な経済を回すことができる点にある。ビットコインでは "通貨" を国家による中央集権的な支配から分散管理のもとにおくことに成功したが、この度のプロジェクトでは "デジタルコンテンツの流通" を法人による中央集権的な支配から分散管理におくことを目標にする。しかし、ただ分散管理をするのではコンテンツの流通の元締めが法人からマイナーになったに過ぎない。一般的なブロックチェーンではマイナーにあたる存在は大量の計算資源を持っている者や仮想通貨を大量に保有している者などであり、今回実現したいクリエイター主体であるようなものとは異なる。そこでクリエイター自体をマイナーの立場におけるような合意形成とインセンティブの仕組みを設計できるブロックチェーンプラットフォームを開発した。また、配信するコンテンツやエコシステムの状況に応じて合意形成とインセンティブの仕組みを適宜変更する必要がある。これを実現するために、Proskenion ではクリエイター自信のデータやコンテンツのメタデータを扱えるようなプリミティブな命令群と合意形成とインセンティブの仕組みを動的且つ自由且つセキュア且つ簡単に変更できる仕組みを導入した。当然、ブロックチェーンシステムには、システムの一部が故障しても障害の重大性に応じて機能を低下させながらも処理を続行するフォールトトレラント性を持つこと、外部からの侵入攻撃によってデータの改竄を行うことが困難であることが求められる。これらを解決するために既存のブロックチェーンで採用されている技術も組み込んだ全く新しいブロックチェーンを開発した。次章以降ではクリエイター主体のブロックチェーンシステム運用基盤プラットフォームの内部メカニズムについて解説する。

2 Software Architecture

ブロックチェーンに必要な構成を図 1 のようにいくつかの独立した機構に分けて考えた。ベースとしてはゲート、合意形成、同期、レポジトリ、コミットと機構を分解した。ソフトウェアアーキテクチャとしては変更に強いクリーン・アーキテクチャを意識しつつ、ディレクトリ構造的には Ethereum のプロジェクトを参考にした。具体的には core ディレクトリ内に Proskenion で用いる全てのモデルと機構のインタフェースを記述する。クリーン・アーキテクチャにおける Entity にあたるところがここである。クリーン・アーキテクチャの最外部層である external interface として convertor ディレクトリ内に、Protocol-Buffers で定義した

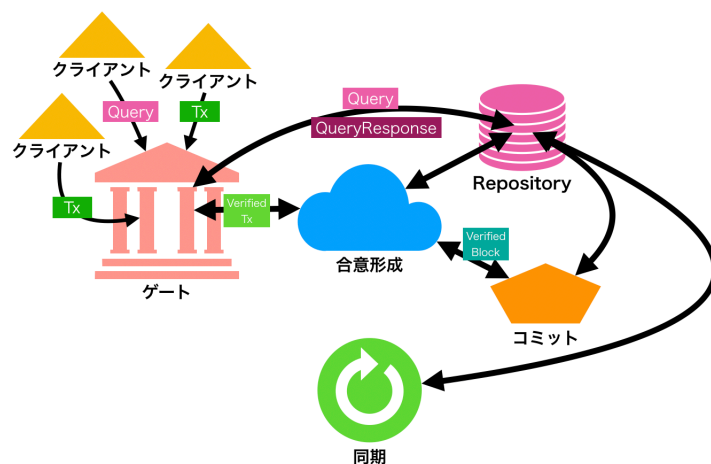


図 1 Blockchain Architecture

モデルと Proskenion 上のインタフェースの変換コードを記述している。入力を受け口として controller ディレクトリ内に通信から受け取ったデータを上述した変換コードを用いて内部で扱うインタフェースに変換してその他の機構 (usecase) に流す役割を記述している。この形式にすることで内部で実際に扱っているシリアル化データ形式やデータベース、暗号ライブラリなどの変更が容易となる。外部ツールの変更に強いだけでなく、各機構の内部実装についても同様に、実装方式が変更されても他の機構への影響がでないようにしている。これは継続的な開発、そして後にチーム開発を行ったり OSS としての開発を進めることを想定してこのように変更に強い設計を採用した。Proskenion では開発言語に Go Language を採用した。その理由として、Go が言語仕様としてエラーチェックを強制させる、書き方がほぼ一意に定まる、型がある、並列処理が得意であるといった、セキュアな分散システムの開発に適している点が挙げられる。また OSS として開発するにあたって、誰が見ても分かりやすい言語であるというのも起用した理由である。書きやすさとセキュリティの面からもバランスが取れているので、ブロックチェーンの開発には非常に適していると考えた。Go のパッケージ管理ツールには Glide を用いた。暗号関数にはハッシュ関数に sha256、署名関数に ed25519 を採用した。これは Ethereum で採用されてる暗号方式であるのと、現存する暗号方式の中でも極めて安全な部類であるのに加え各言語でライブラリが実装されていて扱いやすいという 3 点を理由に適していると考えた。通信プロトコルには GRPC[3]、シリアル化データ形式には Protocol-Buffers[4] を採用した。高速な P2P 通信が可能で且つ型付きのデータ構造を扱え、各言語でライブラリが揃っているという点で、多くのトランザクションを要求されると想定されるブロックチェーンに適していると考えた。また Go との相性も良いからである。

3 Conventions

本章では、Proskenion 上で使われる用語の定義を行う。まず、Proskenion を構成する上の最小単位であるオブジェクトを定義する。

3.1 Primitive

Primitive な型として Protocol-Buffers で定義されている標準型を使用できる。Protocol-Buffers では標準で真偽型, 32bit 符号付き整数型, 64bit 符号付き整数型, 32bit 符号なし整数型, 64bit 符号なし整数型, 文字列型, バイト列型, 配列型, 辞書型がサポートされている。

3.2 Address

Address 型は Proskenion 上に存在する内部データにアクセスするための一意な *id* を表す。*Address* はどの領域 (*Domain*) の誰 (*Account*) が持っているどんなデータ (*Storage*) かを表す *AccountName*, *DomainName*, *StorageName* からなり、*AccountName@DomainName/StorageName* と表記される。例えば、jp にいる bob が持っている passport というデータを示すときは bob@jp/passport と記述する。*Domain* は bob@ac.jp/passport のように . で繋げることで subdomain を定義できる。また *Account* を特定するだけであれば *AccountName* と *DomainName* の情報のみで *AccountName@DomainName* という形で *AccountId* として特定できる。また、便宜上 *AccountName*, *DomainName*, *StorageName* 全てが含まれた *Address* を *WalletId* と呼ぶ。(ex. alice@ac.jp/wallet) *AccountName*, *DomainName* のみが含まれた *Address* を *AccountId* と呼ぶ。(ex. alice@ac.jp) *StorageName* のみまたは *StorageName* と *DomainName* のみが含まれた *Address* を *StorageId* と呼ぶ。(ex. /wallet or ac.jp/wallet)

3.3 Signature

Signature は署名データを表す。*Signature* 型は署名 (*signature*) と公開鍵 (*public_key*) のペアを持つ。署名には秘密鍵によってハッシュ値を暗号化したバイト列を持つ。公開鍵には暗号化に使った秘密鍵と対となる公開鍵のバイト列を持つ。

3.4 Account

Account は Proskenion 上の資源を保持/操作するユーザを表す。*Account* 型は *AccountId*, *AccountName*, *PublicKeys*, *Quorum*, *Balance*, *DelegatePeerId* からなる。*AccountId* は *Account* を一意に特定する *Address* を持つ。*AccountName* は *Account* の表記名を表す文字列を持つ。*Publickeys* は *Account* の権限を行使するために使用する公開鍵のリストをバイト列の配列で持つ。*Quorum* は *Account* の権限を行使するために必要最小の署名の個数を表す 32bit 符号付き整数値を持つ。*Balance* は *Account* が保持している残高の量を表す 64bit 符号付き整数値を持つ。

3.5 Peer

Peer は Proskenion が動作しているサーバ (ノード) の情報を表す。*Peer* 型は *PeerId*, *Address*, *PublicKey*, *Active*, *Ban* からなる。*PeerId* はピアを一意に特定する *AccountId* を表す *Address* を持つ。*Address* はピアにアクセスするために必要なグローバル IP アドレスを表す文字列を持つ。(メンバ変数 *Address* は *Address* 型とは関係ない)。*Active* はピアがネットワークに所属しているブロックチェーンと同期しているかどうかを表す真偽値を持つ。*Ban* はピアがネットワークから排斥されているかどうかを表す真

偽値を持つ。

3.6 Storage

Storage は Proskenion 上で定義可能な key-value データ構造である。*Storage* は *Id*, *Object* を持つ。*Id* は *Storage* を一意に特定するような *id* を示す *Address* を持つ。*Object* は *Storage* で保存しているデータ構造を表す文字列型をキー値, *Object* 型をバリュー値とした辞書を持つ。

3.7 Command

Command は Proskenion 上の資源を更新するためのプリミティブな命令を表す。*Command* 型は *AuthorizerId*, *TargetId*, *Onf of command* からなる。*AuthorizerId* は *Command* の執行者を示す *Address* を持つ。*TargetId* は *Command* の操作対象を示す *Address* を持つ。*One of Command* は操作の内容を表す。詳細は [7. API] に記述する。

3.8 Transaction

Transaction は Proskenion の状態を変更する命令群を表す。*Transaction* 型は *TransactionPayload* と *Signature* を持つ。*TransactionPayload* は *Transaction* を生成した時間を示す 64bit 整数値 *CreatedTime* と順次実行される命令群を示す *Command* 型の配列 *Commands* を持つ。*Signature* には *TransactionPayload* の *Hash* を秘密鍵で暗号化したものを *signature*, 暗号化に使用した秘密鍵と対になる公開鍵を *public_key* とした *Signature* を持つ。詳細は [7. API] に記述する。

3.9 Block

Block は Proskenion の状態遷移の単位となる。*Block* 型は *BlockPayload* と *Signature* を持つ。*BlockPayload* はブロックチェーンの高さを表す 64bit 符号付き整数 *Height*, 一個前のブロックのハッシュを表すバイト列 *PreBlockHash*, ブロックを生成した時間を表す 64bit 符号付き整数 *CreatedTime*, *Block* が持っている *Transasction* のリストのハッシュを持つバイト列 *TxListHash*, ブロックが持つ *Transaction* を全て実行した後の Proskenion の状態のルートハッシュを表すバイト列 *WsvHash*, ブロックが持つ *Transaction* を全て実行した後の取引履歴のルートハッシュを表すバイト列 *TxHistoryHash*, ブロックが何番目の候補の *Peer* によって作られたかを示す 32bit 符号付き整数値 *Round* を持つ。*Signature* は *BlockPayload* のハッシュ値を *Peer* の秘密鍵で暗号化したものを *signature*, 暗号化に使用した秘密鍵と対になる公開鍵を *public_key* とする *Signature* を持つ。

3.10 Object

Object は Proskenion 上で作用する全てのデータ型を内包したデータ構造である。具体的には Protocol-Buffers が標準で用意している型に加えて上記の *Address*, *Account*, *Peer*, *Storage*, *Command*, *Transaction*, *Block*, そして辞書型 *ObjectDict*, リスト型 *ObjectList* を持つ。

3.11 ObjectDict

ObjectDict は文字列型のキー値, *Object* 型のバリュー値を持つ辞書型のオブジェクトである。

3.12 ObjectList

ObjectList は *Object* 型の列を持つリスト型のオブジェクトである。

4 System DataStructure Mechanism

4.1 DataStructure

DataStructure は Proskenion 上で実装されている。プリミティブなデータ構造である。

4.1.1 CacheMap

CacheMap は排他制御付きキャッシングデータ構造である。*CacheMap* は *Set*, *Get* メソッドを持つ。*Set* メソッドはハッシュ関数が定義された *Object* をキャッシュに保存する。*Get* メソッドはハッシュ値を指定すると同一のハッシュ値である *Object* を取得することができる。キャッシングアルゴリズムには簡易的な LRU を採用している。この簡易的 LRU を便宜上 N-Stage Cache Algorithm と呼称する。N-Stage Cache Algorithm は N 個の辞書を持つ。 i 番目の辞書を *i-Stage Cache* と呼び、最も最近使われたものが保存される辞書を *recent-Stage*、最も使われたのが古い辞書を *oldest-Stage* と呼ぶ。*Get* または *Set* メソッドでアクセスされた要素を *recent-Stage* のキャッシュに乗せる。*Set* メソッドを実行した際にキャッシュのサイズを上回ったとき、*oldest-Stage* にキャッシュされている要素を無作為に一つ選び削除する。もし *oldest-Stage* のキャッシュが空の場合は *oldest-Stage*, *recent-Stage* をそれぞれインクリメントした後 N で *mod* をとる。これにより擬似的な LRU アルゴリズムを実現した。*Set*, *Get* はそれぞれ $O(1)$ で実行できる。(v1.0a)

4.1.2 MapQueue

MapQueue は排他制御付きキャッシングキューデータ構造である。*MapQueue* は *Push*, *Pop*, *Erase* メソッドを持つ。*Push* メソッドはハッシュ関数が定義された *Object* を *Queue* に追加する。*Pop* メソッドは *Queue* の先頭 *Object* を取得する。*Erase* はハッシュ値を指定して同一のハッシュ値を持つ *Object* を *Queue* から削除する。*Push*, *Pop*, *Erase* はそれぞれならし計算量 $O(1)$ で実行できる。

4.1.3 MerkleTree

MerkleTree はリストオブジェクトとその部分列のハッシュ値を計算できるデータ構造である。*MerkleTree* は *Push*, *Hash* メソッドを持つ。*Push* メソッドはハッシュ関数が定義された *Object* をリストの先頭に追加する。*Hash* メソッドはリスト全体のハッシュ値を取得する。簡易的な実装として *RootHash* としてハッシュ値の総和を持つ実装をしている。累積的にハッシュ値を計算することで *Push*, *Hash* メソッドをそれぞれ $O(1)$ で実行できる。ここでハッシュ値の計算を $O(1)$ とする。

4.1.4 MerklePatriciaTree

MerklePatriciaTree[5] は部分木のハッシュ値を計算できる基数木データ構造である。*MerklePatriciaTree* は *Search*, *Find*, *Upsert*, *Set*, *Get*, *Hash* メソッドを持つ。*Search* メソッドではキー値と接頭辞が一致している最も浅い内部ノード（部分木）を取得する。*Find* メソッドはキー値で参照した先の葉ノードを取得する。*Upsert* メソッドはキー値とバリュー値が定義されたノードを *MerklePatriciaTree* に追加または更新する。*Set* メソッドはルートハッシュを指定して木のルートを変更する。*Get* メソッドはルートハッシュ指定して内部ノード（部分木）を取得する。*Hash* メソッドでは木のルートハッシュを取得する。*MerklePatriciaTree* ではキー値を基数木の内部ノードとして且つ経路圧縮を行うことで *Search*, *Find*, *Upsert*, をそれぞれ $O(|key|)$, *Set*, *Get*, を $O(1)$, *Hash* を $O(|set\ of\ character\ of\ key|)$ で計算できる。

4.2 Repository

Repository は Proskenion 上でデータを保存する機構ある。データを保存する形式またそのデータ構造について定義する。Proskenion には 3 つの *MerklePatriciaTree* が実装されている。ブロックの列の状態を保存する *BlockChain*, 実行された取引の履歴を保存する *TxHistory*, Proskenion 上のオブジェクトを保存する *WSV(World State View)* とそれぞれ定義する。*WSV* では *Address* 型の *id* によって指定された場所に *Account*, *Peer* または *Storage* を保存している。また一時的に保存されるデータ機構として、*Block* に内包する *Transaction* のリストを保存する *TxList*, クライアントまた他ピアから受け取った *Transaction* を保存する *TxQueue*, 他ピアから受け取った *Block* を保存する *BlockQueue*, 他ピアから受け取った *TxList* を保存する *TxListCache*, *Peer* の通信経路を保存しておく *ClientCache* が実装されている。

4.2.1 TxList

TxList は *Transaction* のリストと全体のハッシュを持つ *MerkleTree* である。*TxList* は *Push*, *List*, *Size*, *Hash* メソッドを持つ。*Push* メソッドは *Transaction* をリストに追加する。*List* メソッドは *Transaction* のリストを取得する。*Size* は *TxList* が持っている *Transaction* の数を取得する。*Hash* メソッドは *Transaction* リストの累積ハッシュを取得する。

4.2.2 BlockChain

BlockChain は *Block* の列を保存する *MerklePatriciaTree* である。*Block* のハッシュ値をキー値, *Block* の実体をバリュー値とおく。*BlockChain* では *Next*, *Get*, *Append* メソッドを持つ。*Next* メソッドは *Block* のハッシュ値を指定してその *Block* の次の *Block* を取得する。*Get* メソッドは *Block* のハッシュ値を指定して *Block* を取得する。*Append* メソッドは *BlockChain* に *Block* を追加する。*MerklePatriciaTree* を利用することでブロックチェーンの分岐に対応することができる。

4.2.3 TxHistory

TxHistory は取引の履歴を保存する *MerklePatriciaTree* である。*TxList* のハッシュ値をキー値, *TxList* の実体をバリュー値におく。また、*Transaction* のハッシュ値をキー値, *TxList* のハッシュ値とその添え字のペアをバリュー値におく。*TxHistory* では *GetTxList*, *GetTx*, *Append* メソッドを持つ。*GetTxList* メソッドは *TxList* のハッシュ値を指定して *TxList* を取得する。*GetTx* は *Transaction* のハッシュ値を

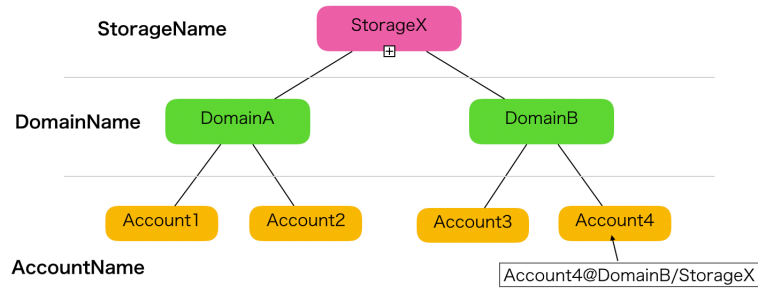


図 2 World State View on MerklePatriciaTree

指定して *Transaction* を取得する。 *Append* は *TxList* とそれが包含する *Transaction* を履歴に追加する。 *MerklePatriciaTree* を利用することでブロックチェーンの分岐に対応することができる。

4.2.4 WorldStateView

WSV は Prosenion 上で操作する状態を保存する *MerklePatriciaTree* である。WSV では図 2 のように *Address* を *StorageName*, *reverse(DomainName)*, *AccountName* の順で並べたものをキー値、*Account*, *Peer* または *Storage* の実体をバリュー値におく。 *Account* を置く時の *StorageName* は *account*, *Peer* を置く時の *StorageName* は *peer* とデフォルトで定めている。WSV は *Query*, *QueryAll*, *Append* メソッドが実装されている。 *Query* メソッドは検索対象の *Address* をキー値として *Account*, *Peer* または *Storage* を取得する。 *QueryAll* メソッドは検索対象の *Address* をキー値として接頭辞が一致している部分木から *Account*, *Peer* または *Storage* のリストを取得する。 *Append* は追加する対象の *Address* をキー値として *Account*, *Peer* または *Storage* を WSV に追加する。 *MerklePatriciaTree* を利用することでブロックチェーンの分岐に対応することができる。

4.2.5 ProposalTxQueue

ProposalTxQueue はクライアントまたは他ピアから提案された *Transaction* を一時的に保存する *MapQueue* である。 *ProposalTxQueue* は *Push*, *Erase*, *Pop* メソッドが実装されている。 *Push* は *Transaction* を指定して *ProposalTxQueue* に追加する。 *Erase* は指定したハッシュ値と一致する *ProposalTxQueue* 内の *Transaction* を削除する。 *Pop* は *ProposalTxQueue* の先頭 *Transaction* を取得する。

4.2.6 ProposalBlockQueue

ProposalBlockQueue はクライアントまたは他ピアから提案された *Block* を一時的に保存する *MapQueue* である。 *ProposalBlockQueue* は *Push*, *Erase*, *Pop* メソッドが実装されている。 *Push* は *Block* を指定して *ProposalBlockQueue* に追加する。 *Erase* は指定したハッシュ値と一致する *ProposalBlockQueue* 内の *Block* を削除する。 *Pop* は *ProposalBlockQueue* の先頭 *Block* を取得する。

4.2.7 TxListCache

TxListCache は他ピアから提案された *TxList* を一時的に保存する *CacheMap* である。 *TxListCache* は *Set*, *Get* メソッドが実装されている。 *Set* メソッドは指定した *TxList* を *TxListCache* に追加する。 *Get* メソッドは指定したハッシュ値から *TxList* を取得する。

4.2.8 ClientCache

ClientCache は他ピアとの接続情報を一時的に保存する *CacheMap* である。 *ClientCache* は *Set*, *Get* メソッドが実装されている。 *Set* メソッドは *Peer* と接続情報を指定して *ClientCache* に保存する。 *Get* メソッドは指定した *Peer* の接続情報を取得する。

4.2.9 Repository

Repository は Proskension で管理しているデータ全体を総括して管理するインターフェースである。 *Repository* は *Top*, *Me*, *GetDelegatedAccounts*, *Commit*, *GenesisCommit*, *CreateBlock* メソッドが実装されている。 *Top* メソッドは最新且つ最も信頼度の高いブロックチェーンの先頭のブロックを取得する。 *Me* メソッドは自身のピア情報を取得する。 *GetDelegatedAccounts* はブロックの提案者として適当なアカウントのリストを取得する。 *Commit* は *Block* と *TxList* を指定して *BlockChain* に実際にコミットする。 *GenesisCommit* は *TxList* を指定して全ての *Transaction* を強制的に実行して高さ 0 の *Block* をコミットする。 *CreateBlock* は指定した *ProposalTxQueue*, 現在の *Round*, 現在の時刻から提案されたブロックを実行して取得する。先頭の *Block* が持っているハッシュ値から *WSV*, *TxHistory*, *BlockChain* の状態を復元してそれぞれ操作することができる。状態の復元は *MerklePatriciaTree* のルートノードのハッシュ値を参照するだけなので $O(1)$ で実行できる。

5 Proskension Domain Specific Language

Proskension Domain Specific Language(以下、ProSL) は Proskension 上で実行できる DSL である。ProSL は無限ループを避けるためにチューリング完全でない命令セットを用いている。ProSL は Protocol-Buffers で定義されており、内部での DSL の検証、実行もシリアライズされた Protocol-Buffers のデータを用いる。ver 1.0a では YAML[6] によって書かれた ProSL を Protocol-Buffers に変換するコードが実装されている。Proskension 上では Protocol-Buffers を用いて制御するため、Protocol-Buffers への変換コードさえ実装すれば YAML 以外でも ProSL を記述可能になる。

5.1 Design

ProSL は Proskension 上で保存する時の *Storage* の設計は以下のようになっている。

```
{
  "prosl_type": "consensus" or "incentive" or "update",
  "prosl": 0x....(byte data that is marshal of prosl)
}
```

Proskension を動かすには Consensus algorithm, Incentive algorithm, Update algorithm そして Genesis Block の 4 種類の設計をしなければならない。これらのそれぞれ異なるタイミングで動作する。

5.1.1 Consensus Algorithm Design

ブロックの生成者を選出する仕組みを設計する。これは *Block* をコミットまたは作成する直前に実行される。コミットの前に実行される際はブロックの生成者が適切かどうかの検証に用いる。作成の前に実行される際は自分がブロックの作成者として適切かを判定するために用いる。これは *Account* のリストを返す。このリストの先頭から現在の *Round* 番目のアカウントが信頼している *Peer* が正しいブロックの生成者である。

5.1.2 Incentive Algorithm Design

インセンティブ付与の仕組みを設計する。これは *Block* をコミットする過程で実行される。これは *Transaction* を返す。この *Transaction* は ProSL が実行された直後に強制的に実行され変更が *BlockChain* に刻まれる。

5.1.3 Update Algorithm Design

Consensus, Incentive, Update algorithm を変更するための条件を設計する。これは *CheckAndCommitProsl* という特殊なコマンドが実行された時に実行する。これは真偽値を返す。True が返った場合は提案された新たなアルゴリズムが適用される。

5.1.4 Genesis Block Design

高さ 0 の *Block* で実行する命令群を設計する。これは Proskension の初回起動時に一度だけ実行される。これは *Transaction* を返す。この *Transaction* は初回のブロックにて強制的に実行され変更が *BlockChain* に刻まれる。

5.2 Prosl Operators

ProSL の BNF は以下の通りである。この BNF では `<variableName: operatorTypeName>` の形式で記述する。[op] は Operator op のリストを示す。{STRING:op} は *key* が *STRING* である *Operator* op の辞書型を示す。<var:op>? は *Operator* op が *optional* であることを示す。

```
Prosl := <ops:[ProslOperator]>
```

```
ProslOperator := <set:SetOperator> |  
                 <if:IfOperator> |  
                 <elif:ElifOperator> |  
                 <else:ElseOperator> |  
                 <err:ErrOperator> |  
                 <require:RequireOperator> |  
                 <return:ReturnOperator>
```

```
SetOperator := <var:STRING> <value:ValueOperator>
```

```
IfOperator := <cond:ConditionalFormula> <do:Prosl>
```

```
ElifOperator := <cond:ConditionalFormula> <do:Prosl>
```

```
ElseOperator := <do:Prosl>
```

ErrCatchOperator := <code:ERRCODE> <do:prosl>

RequireOperator := <cond:ConditionalFormula>

ReturnOperator := <return:ReturnOperator>

ValueOperator := <query:QueryOperator> |
 <tx:TxOperator> |
 <cmd:CommandOperator> |
 <storage:StorageOperator> |
 <plus:PlusOperator> |
 <minus:TxOperator> |
 <mult:MultipleOperator> |
 <div:DivisionOperator> |
 <mod:ModOperator> |
 <and:AndOperator> |
 <or:OrOperator> |
 <xor:XorOperator> |
 <concat:ConcatOperator> |
 <valued:ValuedOperator> |
 <indexed:IndexedOperator> |
 <var:VariableOperator> |
 <list:ListOperator> |
 <map:MapOperator> |
 <cast:CastOperator> |
 <list_cmp:ListComprehensionOperator> |
 <sort:SortOperator> |
 <slice:SliceOperator> |
 <is_defined:IsDefinedOperator> |
 <verify:VerifyOperator> |
 <pagerank:PageRankOperator> |
 <len:LenOperator> |
 <object:OBJECT>

<QueryOperator> ::= <authorizer:ValueOperator> <select:STRING> <type:OBJECTCODE>
 <from:ValueOperator> <where:ValueOperator> <orderBy:ORDERBY> <limit:INT32>

<CommandOperator> ::= <create_account:MapOperator> |
 <create_storage:MapOperator> | <add_balance:MapOperator>
 <transfer_balance:MapOperator> | <add_publickeys:MapOperator>
 <remove_remove:MapOperator> | <set_quorum:MapOperator>
 <define_storage:MapOperator> | <create_storage:MapOperator>
 <update_object:MapOperator> | <add_object:MapOperator>
 <transfer_object:MapOperator> | <add_peer:MapOperator>

```

        <activate_peer:MapOperator> | <suspend_peer:MapOperator>
        <ban_peer:MapOperator> | <consign:MapOperator>
        <check_and_commit_prosl:MapOperator> |
        <update_storage:MapOperator>
<StorageOperator> ::= <object:MapOperator>
<PlusOperator> ::= <ops:ValueOperator+>
<MinusOperator> ::= <ops:ValueOperator+>
<MultipleOperator> ::= <ops:ValueOperator+>
<DivisionOperator> ::= <ops:ValueOperator+>
<ModOperator> ::= <ops:ValueOperator+>
<AndOperator> ::= <ops:ValueOperator+>
<OrOperator> ::= <ops:ValueOperator+>
<XorOperator> ::= <ops:ValueOperator+>
<ConcatOperator> ::= <ops:ValueOperator+>
<ValuedOperator> ::= <obj:ValueOperator> <type:OBJECTCODE> <key:STRING>
<IndexedOperator> ::= <obj:ValueOperator> <type:OBJECTCODE> <index:ValueOperator>
<ListOperator> ::= <objects:[ValueOperator]>
<MapOperator> ::= <objects:{STRING:ValueOperator}>
<ListComprehensionOperator> ::= <list:ValueOperator> <var:STRING> <if:ConditionalFormula>? <elem:Val
<SortOperator> ::= <list:ValueOperator> <order:ORDERBY> <type:OBJECTCODE> <limit:ValueOperator>?
<SliceOperator> ::= <list:ValueOperator>? <left:ValueOperator>? <right:ValueOperator>?
<IsDefine::=> ::= <var:ValueOpeartor>
<VerifyOperator> ::= <sig:ValueOperator> <hasher:ValueOperator>
<PageRankOperator> ::= <Storages:ValueOperator> <toKey:ValueOperator> <outName:ValueOperator>
<LenOperator> ::= <list:ValueOperator>

<ConditionalFormula> ::= <or:OrOperator> |
                        <and:AndOperator> |
                        <not:NotOperator> |
                        <eq:EqOperator> |
                        <ne:NeOperator> |
                        <gt:GtOperator> |
                        <ge:GeOperator> |
                        <lt:LtOperator> |
                        <le:LeOperator> |
                        <verify:VerifyOperator>
<NotOperator> ::= <op:ValueOperator>
<EqOperator> ::= <ops:[ValueOperator]>
<NeOperator> ::= <ops:[ValueOperator]>
<GtOperator> ::= <ops:[ValueOperator]>

```

```

<GeOperator> ::= <ops:[ValueOperator]>
<LtOperator> ::= <ops:[ValueOperator]>
<LeOperator> ::= <ops:[ValueOperator]>

<OrderByOperator> ::= <key:ValueOperator> <order:ORDERCODE>

<STRING> ::= String
<OBJECT> ::= Object
<INT32> ::= int32
<OBJECTCODE> ::= any | bool | int32 | int64 | uint32 | uint64 | string |
                bytes | address | signature | account | peer | list | dict |
                storage | command | transaction | block
<ORDERCODE> ::= DESC | ASC

```

6 System Core Mechanism

Proskenion はパブリックブロックチェーンプラットフォームである。Ethereum でも用いられている MerklePatriciaTree 上にトランザクションデータと状態を保存する分散型台帳基盤である。トランザクションは Proskenion の状態を変化させるコマンド列を持つ。クライアントはトランザクションに署名をつけて Proskenion ピアへと送信する。Proskenion は受け取ったトランザクションを他のピアへ伝搬して各自トランザクションを保存する。また、合意形成過程でブロックの生成者がトランザクションの列としてブロックを生成し他のピアへ伝搬する。ブロックを受け取ったピアはブロックを Commit して良いかの検証を行い妥当なブロックチェーンの先頭に繋げる。本章ではこれら一連の動作を行う Proskenion で実装されている主な機構であるゲート、合意形成、同期、について説明する。

6.1 Gate

Gate はクライアント又は他のピアから送られてくるメッセージを受け取って処理する機構である。Gate には APIGate, ConsensusGate, SyncGate がある。APIGate では *Transaction* を受け取る WriteRPC, Query を受け取る ReadRPC が実装されている。APIGate で受け取った *Transaction* は静的検証後 *ProposalTxQueue* に挿入された後、他のピアに伝搬を行う。APIGate で受け取った *Query* は検証後、*Query* のルールに従ってデータを取得しピアの署名をつけて *Query* の送り主に返す。*SyncGate* では同期願いを受けたさいに自分の持っている主体の *BlockChain* の情報を全て送り主に送信する。同期処理は相互 Streaming 通信で行われる。

6.2 Consensus

Consensus は合意形成を行う機構である。*Consensus* では 3 つの処理が並列して無限ループしている。1 つ目は *Block* を生成して他の *Peer* に伝搬する処理である。このループでは *Block* がコミットされない限り一定時間おきに *Round* をインクリメントされて現在の *Round* における *Block* 生成者が自分であった

場合に *Block* を生成、伝搬する。2 つ目は伝搬された *Block* をコミットする処理である。このループでは *ProposalBlockQueue* に *Block* が入ったイベントを受け取り *Block* と *TxList* を復元し検証の結果妥当であればその *Block* をコミットする。3 つ目は自ピアが *Active* でない場合にデータの同期を行う処理である。このループでは自ピアが全体と同期がとれているのかをチェックし取れていない場合は *Synchronizer* に同期リクエストを投げる。

6.3 Synchronizer

Synchronizer はデータ同期を行う機構である。*Synchronizer* では *Peer* を指定してその *Peer* からメインの *BlockChain* のデータを取得して同期を行う。この同期処理は *StreamingRPC* を用いて行う。同期願いを出された側は *Block* と *TxList* を同期依頼主に送り続ける。終了条件は元のチェーンを持っている側が全てのブロックを送信しきるか、同期依頼を出した側が合意形成の過程で伝搬された *Block* で先頭ブロックが更新されるか、エラーが発生するかのいずれかである。

7 API

各コマンドの作用などは API Documents (<https://proskenion.github.io/docs>) に記す。

7.1 Write

Transaction の設計は以下のようになっている。

```
message Transaction {
  message Payload {
    int64 createTime = 1;
    repeated Command commands = 2;
  }
  Payload payload = 1;
  repeated Signature signatures = 2;
}

message Command {
  string authorizerId = 1;
  string targetId = 2;
  oneof command {
    CreateAccount createAccount = 3;
    AddBalance addBalance = 4;
    TransferBalance transferBalance = 5;
    AddPublicKeys addPublicKeys = 6;
    RemovePublicKeys removePublicKeys = 7;
    SetQuorum setQuorum = 8;
    DefineStorage defineStorage = 9;
```

```

    CreateStorage createStorage = 10;
    UpdateObject updateObject = 11;
    AddObject addObject = 12;
    TransferObject transferObject = 13;
    AddPeer addPeer = 14;
    ActivatePeer activatePeer = 15;
    SuspendPeer suspendPeer = 16;
    BanPeer banPeer = 17;
    Consign consign = 18;
    CheckAndCommitProsl checkAndCommitProsl = 19;
    ForceUpdateStorage forceUpdateStorage = 30;
}
}

```

Transaction の静的検証では署名と *Payload* の中身が一致しているかをチェックする。*Transaction* の動的検証ではコマンド列の中にある *AuthorizerId* を操作するために必要な署名が揃っているかをチェックする。*Transaction* のコマンド列は ACID 性を備えている。

7.2 Read

Query の設計は以下のようになっている。

```

message Query {
    message Payload {
        string authorizerId = 1;
        string select = 2;
        ObjectCode requestCode = 3;
        string fromId = 4;
        string where = 5;
        OrderBy orderBy = 6;
        int32 limit = 7;
        int64 createTime = 8;
    }
    Payload payload = 1;
    Signature signature = 2;
}

message QueryResponse {
    Object object = 1;
    Signature signature = 2;
}

```

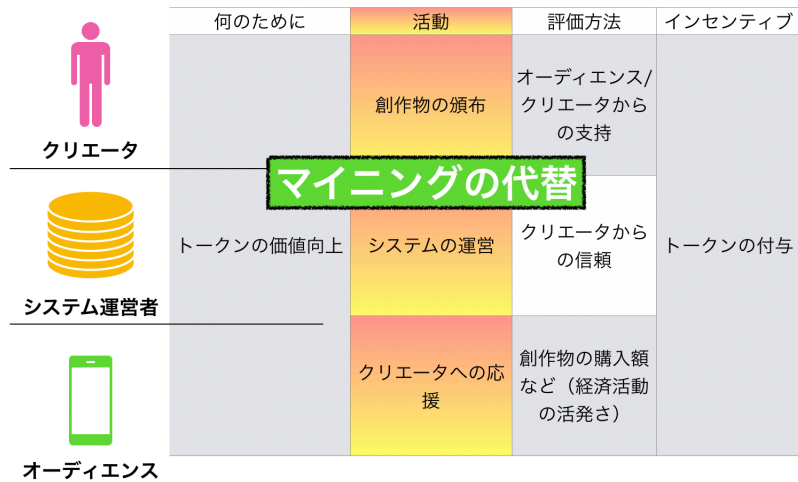


図3 Proof of Creator Mining

AuthorizerId は誰の権限で *Query* を発行するかを *AccountId* で指定、*FromId* は検索対象となる *Address* を指定、*Select* は取得する *Object* の型を指定する。*FromId* で指定した検索対象が範囲検索 (*WalletId* ではない) の場合に追加で検索クエリを設定できる。*Where* は取得した *Object* のリストにフィルターをかける条件式を指定、*OrderBy* は取得したリストをソートするルールを指定、*Limit* は取得したリストの何番目までを返すかを指定する。*Signature* には *Payload* を *Authorizer* が署名したものを指定する。検索結果は *QueryResponse* として返ってくる。*Object* は *Query* を実行した結果返ってきたオブジェクト、*Signature* は *Query* を実行した *Peer* が *Object* を署名したものが入っている。

8 Discussion

Proskenion のユースケースについて考える。

8.1 Proof of Creator

クリエイターを主体とする新たな合意形成アルゴリズム "Proof of Creator" を提案する。仮にクリエイター同士の関係をフォロー/フォロワーを示す有効グラフで表現する。この有向グラフから頂点のページランクを計算して出したクリエイターのランクをクリエイターの信頼度とする。信頼度の上位4人によって指定されたシステム運営者達がブロックの生成を順繰りに行う。これにより、より良いクリエイターがサーバを選出するクリエイターが主体であるような運営システムを構築することができる。図3では大まかな報酬設計を示す。

8.2 Selection of Ms.Contest

システムのテスト運用という意味でミスコンのような短期的な人気投票システムを行うのも面白い。ミスコンのエントリー者をクリエイターと定義して彼らのインセンティブ設計が直接評価方法となり実証実験的なコンテストが開催可能であると思われる。図4では大まかな報酬設計を示す。




	何のために	活動	評価方法	インセンティブ
 同人作家		同人誌の頒布	オーディエンス/ 作家からの支持	トークンの付与/ 頒布利益
 システム運営者	トークンの価値向上	システムの運営	作家からの信頼	トークンの付与/ 頒布利益の手数料
 オーディエンス		作家への応援	創作物の購入額など (経済活動の活 発さ)	トークンの付与

図 4 Ms.Contest Mining

	何のために	活動	評価方法	インセンティブ
 エントリー者	グランプリ (トークン数=票数)	セルフプロデュース/写 真・ブログの更新	オーディエンス システム運営者から の支持	トークンの付与
 システム運営者	支持者のNFT取得 またそれに準じて 得られるリターン	システムの運営	エントリー者からの 信頼	支持者のNFTの付与
 オーディエンス		エントリー者への 応援	投票/投げ銭 “いつ”したか	支持者のNFTの付与

図 5 Online comic market platform Mining

8.3 Online Comic Market Platform

オンライン上でコミックマーケットのような仕組みを模擬する。Proskenion を用いることで自治コミュニティが運営する同人即売サービスが実現できる。クリエイターを同人作家と定義して作品の販売をマイニングと位置づける。評価は他のクリエイターからの信頼度と販売利益と定義する。図 5では大まかな報酬設計を示す。

9 Conclusion

Proskenion はプリミティブな命令セットの組み合わせで高い表現力を持つパブリックブロックチェーンを実現した。また、合意形成とインセンティブの設計をハードフォーク無しに変更できる仕組みを導入/実現した。さらに、応用先としてあらゆるデジタルクリエイターを対象に新たな収益源となりうる分散運営システムの例を提案した。

10 Feature works

Feature works としては実証実験での利用として Proskenion の特性を活かして実証実験で利用する形を模索している。具体的にはインセンティブと合意形成アルゴリズムを簡単に設計できるため、”どういった報酬設計” が ”どのような結果をもたらすか” の実験を中~小規模で行うのに向いている。また、Proskenion が実運用されるための世界の基盤を創るための新たなブロックチェーン開発も進めたい。最近注目されているブロックチェーン開発ツールキットとして”Substrate[7]” というものがある。これは複数のブロックチェーンを繋げる Project の一環として作られた Polkadot[8] で運用されるブロックチェーン開発ツールである。これを用いてスケーラビリティと相互交換性を担保する新たなブロックチェーンの開発を行う。

11 Acknowledgement

参考文献

- [1] BitCoin [<https://bitcoin.org/bitcoin.pdf>]
- [2] Ethereum [<https://ethereum.github.io/yellowpaper/paper.pdf>]
- [3] GRPC [<https://grpc.io/>]
- [4] Protocol-Buffers [<https://developers.google.com/protocol-buffers/>]
- [5] MerklePatriciaTree [<https://github.com/ethereum/wiki/wiki/Patricia-Tree>]
- [6] YAML [<https://yaml.org/spec/history/2001-05-26.html>]
- [7] Substrate [<https://www.parity.io/what-is-substrate/>]
- [8] Polkadot [<https://polkadot.network/>]