Vite: 高パフォーマンスの非対称性分散型アプリケーションプラットフォーム

概要

Vite は、一般向けの分散型アプリケーションプラットフォームです。高スループットや待ち時間の短さ、スケーラビリティに対する業務用アプリケーションの水準を満たし、同時にセキュリティも考慮しています。Vite は DAG レジャーストラクチャ（台帳構造）を用いており、レジャー内のトランザクションはアカウントによりグループ化されます。Vite 内のスナップショットチェーン構造により、DAG レジャーにおけるセキュリティを補うことができます。HDPoS コンセンサスアルゴリズムにより、トランザクションの書き込みと確認が非同期的になりますので、高パフォーマンスとスケーラビリティが実現されます。Vite VM は EVM や Solidity からの拡張スマートコントラクト言語と互換性があり、これによりさらにパワフルなディスクリプションが可能となります。加えて、Vite のデザインにおける重要な改善点として、非同期イベントによるアーキテクチャの採用があります。これにより、スマートコントラクト間でメッセージを通じて情報をやり取りすることで、システムのスループットやスケーラビリティを大幅に改善することになりました。ビルトインのネイティブトークンに加え、Vite では、ユーザーが自身のデジタルアセットを発行したり、クロスチェーンの価値転移やループリングプロトコル[1]に基づいた交換を行ったりすることができます。Vite により、割り当てに基づいたリソースの配分を行うことが可能となり、ライトユーザーはトランザクション費用を支払う必要がなくなります。また、コントラクトのスケジューリングやネームサービス、コントラクトの更新、ブロックプルーニングなどの機能もサポートしています。

1　導入

1.1　定義

Vite は、ユニバーサルな分散型アプリケーションプラットフォームで、スマートコントラクト一式をサポートしています。このスマートコントラクトはそれぞれ、独立した状態と異なるオペレーションロジックを有する状態機械で、メッセージのやり取りによってコミュニケーションを取ることができます。一般に、システムはトランザクション状態機械です。システムの状態は s ∈ S（ワールドステートとも）で、これは独立したアカウントの状態から成り立っています。アカウントの状態に変更を及ぼすイベントはトランザクションと呼ばれます。より正確な定義は以下を参照してください。:

定義 1.1（トランザクション状態機械）：トランザクション状態機械は、4 タプル（T, S, g, δ）で、T はトランザクション一式、S は状態一式、g ∈ S は初期状態（ジェネシスブロック）、δ ：S × T → S は状態トランザクション関数です。

このトランザクション状態機械のセマンティクスは、離散変移システムで、これは以下のように定義されます:

定義 1.2（トランザクション状態機械のセマンティクス）：トランザクション状態機械のセマンティクス（T, S, s0, δ）は離散遷移システムです。(S, s0,→).→∈ S × S は変移の関係性です。

同時に、分散型アプリケーションプラットフォームは、結果総合性を有する分散型システムです。コンセンサスアルゴリズムを介して、ノード間の最終的な状態が取得されます。現実的なシナリオでは、スマートコントラクトの状態には分散型アプリケーションの複雑なデーター式が大量に保存されており、ノード間でのやり取りができません。したがって、最終的な総合性を獲得するため、ノードはトランザクションの一式を遷移しなければなりません。こうしたトランザクションのグループは、特定のデータ構造（レジャーと呼ばれるもの）にまとめられます。.

定義 1.3（レジャー）：レジャーは、繰り返し生成される抽象データ型を持つトランザクション一式から成ります。これは以下のように定義されます。:

$$\begin{cases} l = \Gamma(T_t) \\ l = l_1 + l_2 \end{cases}$$

この中で、Tt ∈ 2 T はトランザクション一式を示し、, 「 ∈ 2 T → L はトランザクション一式により帳簿を構築する関数を表しています。L はレジャー一式を表し、+ ：L×L → L は 2 つのサブレジャーを 1 つに統合するオペレーションを表しています。

このようなシステムにおいて、レジャーは通常、状態ではなくトランザクションのグループを表すのに用いられることに注意してください。Bitcoin[2]や Ethereum[3]においては、そのレジャーはブロックチェーン構造になっており、トランザクションは全体的に順序立てられます。レジャーのトランザクションを修正するには、帳簿の中にサブレジャーを再構築する必要があります。このため、トランザクションを改竄するコストが高まります。

同一グループのトランザクションに応じて、異なる有効な帳簿が作成されますが、これはそれぞれ異なるトランザクションの順序を表しており、システムを別々の状態にしてしまう場合があります。これが発生した場合、それは通常フォークと呼ばれます。

定義 1.4（フォーク）：$T_t,\ T_{t'} \in 2T, T_t \subseteq T_{t'}$. *Mëu l = 「1(Tt), l′ = 「2(Tt′)*で、かつ l ≤ l ′ を満たさないとき、 l と l ′ をフォークレジャーとすることができます。≤はプレフィックスの関係性を表しています。

トランザクション状態機械のセマンティクスにより、初期状態から、レジャーがフォークしていない場合、それぞれのノードは最終的に同じ状態に入ったことを容易に確認することができます。一方、フォークの発生したレジャーが受信された場合に、間違いなく異なる状態に入ってしまったかどうかの判断は、レジャー内のトランザクションに内在するレジャーがトランザクション間の順序をどのようにまとめたかによります。実際には、交換法則を満たすトランザクションが要因である場合が多いのですが、アカウントデザインの問題により、これが頻繁にフォークを起こす

場合があります。システムが最初の状態から始まり、2 つのフォークが発生したレジャーを受信して同一の状態に落ち着いた場合には、この 2 つのレジャーについて疑似フォークが起こったと言います。
定義 1.5（疑似フォーク）：初期状態 $s0 \in S$, $ledger\ l_1, l_2 \in L$, $s_0\ (l_1) \rightarrow s_1$, $s_0\ (l_2) \rightarrow if\ l1 \neq l2, and\ s_1 = s_2$の場合、この 2 つのレジャー（l1, l2）は疑似フォークが発生したレジャーであるとします。
精巧にデザインされたレジャーにおいては、疑似フォークの発生可能性は最小限でなければなりません。
フォークが発生した場合、それぞれのノードは複数のフォークが発生したレジャーから 1 つを選ばなければなりません。状態の整合性を確保するため、ノードは選択を完了する上で同一のアルゴリズムを用いなければなりません。このアルゴリズムは、コンセンサスアルゴリズムと呼ばれます。
定義 1.6（コンセンサスアルゴリズム）：コンセンサスアルゴリズムとは、レジャー一式を受信し、以下を満たすレジャーのみを返す関数です。: $\Phi : 2L \rightarrow L$

コンセンサスアルゴリズムは、システムデザインにおいて重要な部分です。優れたコンセンサスアルゴリズムとは、それぞれのフォークのコンセンサスのブレを最小化するため、高速で収斂を処理でき、悪意のある攻撃に対して守ることのできる優れた能力を有しているものでなければなりません。

1.2 現在の進行

Ethereum[4]はそのようなシステムを実現する上で先見を担っています。Ethereum のデザインにおいて、ワールドステートの定義は S = ΣA であり、これはアカウントから a ∈ A およびこのアカウントの状態 σa ∈ Σ のマッピングです。したがって、Ethereum の状態機械の状態のいずれかがグローバルである場合、それはノードがどのアカウントの状態でもいつでも得られるということを示しています。

Ethereum の状態遷移機能 δ は、プログラムコード一式によって定義されます。コードのそれぞれのグループはスマートコントラクトと呼ばれます。Ethereum はチューリング完全仮想マシン（EVM。その命令セットが EVM コード）を定義します。ユーザーは JavaScript に類似したプログラミング言語である Solidity を用いてスマートコントラクトを作成し、それを EVM コードにコンパイルして Ethereum 上で展開できます[5]。スマートコントラクトが正常に作成されたら、コントラクトアカウント a が状態変位関数 δa を受信したと定義するのと同等となります。EVM はそのようなプラットフォームで広く用いられていますが、問題もあります。例えば、ライブラリ関数がサポートされていなかったり、セキュリティ上に問題があったりという具合です。

Ethereum のレジャー構造はブロックチェーン[2]です。ブロックチェーンはブロックからできており、それぞれのブロックには一連のトランザクションが入っていて、最近のブロックは直前のブロックのハッシュを参照し、チェーン構造を作ります。
$\Gamma(\{t1, t2, ... | t1, t2, ... \in T\}) = (..., (t1, t2, ...))$ (1)
この構造の一番の利点は、トランザクションの改竄を効果的に防ぐことができるという点です。しかし、トランザクションの全ての順序を完全に保持することになるため、2 つのトランザクションの順序の交換により新たなレジャーが生まれることとなって、これにフォークが起こる可能性が非常に高くなります。事実、この定義において、トランザクション状態機械の状態空間はツリーと見なされます。初期状態はルートノードで、それぞれのトランザクションの順序が異なるパスを表し、リーフノードが最終的な状態となります。現実には、大量のリーフノードの状態が同一となり、これが大量の疑似フォークに繋がります。

コンセンサスアルゴリズム Φ は PoW と呼ばれ、これは当初、Bitcoin のプロトコルで提唱されていました[2]。PoW のアルゴリズムは、容易に証明可能ながら答えを導くのが難しい数学的問題によるものです。例えば、ハッシュ関数 h : N → N に基づいて、h(T + x) ≥ d を満たす特定の値 d（difficulty: 採掘難易度）を満たすように答え x を求めるとき、T はブロック内に含まれるトレードリストを二進数で表したものとなります。ブロックチェーン内のそれぞれのブロックには、このような問題のソリューションが含まれています。全てのブロックの採掘難易度を合計すると、それがブロックチェーンレジャーの合計の採掘難易度となります。:
$D(l) = D(\sum_i li) = \sum_i D(li)$ (2)
したがって、フォークから適切なアカウントを選ぶとき、最も難易度の高いフォークを選ぶことになります:

$$\Phi(l_1, l_2, ..., l_n) = l_m \text{ where } m = \arg\max_{i \in 1..n}(D(l_i)) \quad (3)$$

PoW のコンセンサスアルゴリズムのセキュリティはより強固で、Bitcoin や Ethereum との相性も優れています。しかしこのアルゴリズムには 2 つ大きな問題があり、1 つは数学的問題を解くためには膨大な計算リソースが必要であって結果としてエネルギーが浪費されてしまうこと、もう 1 つはアルゴリズムの収斂速度が遅いためにシステム全体のスループットが阻害されてしまうことです。現在、Ethereum の TPS はおよそ 15 で、分散型アプリケーションのニーズには到底及びません。

1.3 改善の方向性

Ethereum の誕生後、Ethereum のコミュニティおよびその他の類似プロジェクトは、システムを様々な面から改善しようと試み始めました。システムの抽象モデルから、以下の方向性の改善が見込まれます:

- システム状態 S の改善
- 状態遷移関数 δ の改善
- レジャー「 の構造の改善
- コンセンサスアルゴリズム Φ の改善

1.3.1 システムの状態の改善

システムの状態の改善の根幹にあるのは、世界のグローバルステートのローカライズです。それぞれのノードはあらゆるトランザクションや状態遷移と繋がらなくなり、ただ全体の状態機械のサブセットを維持するようになります。こうして、セット S とセット T のポテンシャルが大幅に減少し、これによってシステムのスケーラビリティが改善されます。このようなシステムとして、Cosmos[6]や Aelf[7]、PChain などがあります。

重要な点として、スキームに依存したこのサイドチェーンは、スケーラビリティの代償としてシステム状態の健全性を犠牲にしてしまいます。これにより、その上で稼働している dApp それぞれの分散化が脆弱になります。スマートコントラクトの取引履歴がネットワーク全体の全てのノードに保存されなくなり、一部のノードにのみ保存されるようになります。加えて、そのようなシステムでは、コントラクト間のインタラクションがボトルネックになります。例えば Cosmos では、異なるゾーンにおけるインタラクションには、完了のために共通のチェーン Hub が必要になります[6]。

1.3.2 状態遷移関数の改善

EVM の改善に基づいて、プロジェクトの中には、より潤沢なスマートコントラクトプログラミング言語を提供しているものもあります。例えば、スマートコントラクト言語の Rholang は、π の計算に基づいた RChain で定義されています。NEO のスマートコントラクトは NeoContract（ネオコントラクト）と呼ばれ、これは Java や C#といったようなメジャーなプログラミング言語で開発が可能です。EOS は C/C++でプログラムされています。

1.3.3 レジャー構造の改善

レジャー構造の改善の方針は、同一クラスの構築です。複数のトランザクションの全体順序を持つ線形レジャーは、半順序関係を記録するだけの非線形レジャーに改善されます。この非線形レジャー構造は DAG（有向非巡回グラフ）です。現在、Byteball[8]や IOTA[9]、Nano[10]などのプロジェクトが DAG のアカウント構造に基づいたマネーの暗号化関数を実現しています。プロジェクトの中には、DAG を用いてスマートコントラクトを実行しようとしているものもありますが、現在の所、この方向性の改善はこれといった結果に結びついていません。

1.3.4 コンセンサスアルゴリズムの改善

コンセンサスアルゴリズムの改善は、主にシステムのスループットを改善のことであり、主な方向性としては疑似フォークの生成を抑圧することになります。疑似フォークにどのようなファクターが含まれるの

かについては、以下を参照してください。

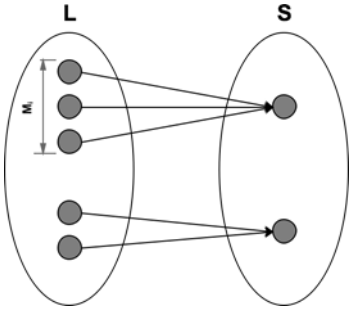

図 1：疑似フォーク

図の通り、L は、トランザクション一式に対する、可能性があるあらゆるフォークの発生したアカウントです。また、S は異なる順序によって得られる状態の集積です。定義 1.4 により、マッピング f：L → S は全射となります。また、定義 1.5 により、このマッピングは全射ではありません。ここから、疑似フォークの可能性を計算します。:

C ユーザーがレジャーを生成する権限を持っているとき、 M = |L|,N = |S|,Mi = |Li |で、この場では Li = {I|f(I) = si , si ∈ S}です。疑似フォークの可能性は以下のようになります。:

$$Pff = \sum_{i=1}^{n} \left(\frac{Mi}{M}\right)^{c} - \frac{1}{M^{c-1}} \qquad (4)$$

この式から、疑似フォークの可能性を低くするには、2 つの方法があると分かります。:

- レジャー一式の L の同値関係を定め、同値類をそれらに分割し、フォークが発生したレジャーの構築を少なくする。
- レジャーを生成する権限を持つユーザーを限定し、これにより C を減らす。

最初の方法は、Vite デザインにおいて重要な方向性です。これについては後ほど詳しく触れます。2 つ目の方法は、多くのアルゴリズムにより採用されているものです。PoW アルゴリズムにおいて、あらゆるユーザーはブロックを生成する権限を有しています。PoS アルゴリズムは、ブロックを生成する権限を、システム権限を持つ人に限定します。DpoS アルゴリズム[11]は、ブロックを生成する権利を有するユーザーを限定し、エージェントノードのグループ内においてより強く制限します。

現在、改善されたコンセンサスアルゴリズムにより、影響力のあるプロジェクトが表れてきました。例えば、Cardano は Ouroboros と呼ばれる PoS アルゴリズムを用いており、literature（リテラチャー）[12]はアルゴリズムの関連する記号の厳密な証明を行います。BFT-DPOS アルゴリズムは EOS[13]に用いられており、DPoS アルゴリズムのバリアントとして、ブロックの生成を早めることによってシステムのスループットを改善しています。Qtum[14]のコンセンサスアルゴリズムも PoS アルゴリズムです。Casper アルゴリズムは RChain[15]に採用されており、これも PoS アルゴリズムの一種です。

コンセンサスアルゴリズムの改善については、独自の取り組みを掲げるその他多くのプロジェクトが存在します。NEO[16]は BFT アルゴリズム（dBFT）を、Cosmos[6]は Tendermint[17]と呼ばれるアルゴリズムを用いています。

2 レジャー

2.1 概要

レジャーのロールは、トランザクションの順序を決定することであり、トランザクションの順序は以下の 2 点に影響を及ぼします。:

- 状態の整合性：システムの状態が CRDT（コンフリクトしない複製可能なデータ）[18]であるため、全てのトランザクションが交換可能というわけではなく、異なるトランザクションの実行シークエンスによってシステムが異なる状態に入ってしまう場合があります。
- ハッシュの効果性：レジャーでは、トランザクションはブロックにパッケージングされます。ブロ

ックには、互いを参照するハッシュが含まれています。トランザクションの順序は、レジャーで引用されるハッシュの接続性に影響を与えます。この影響力が大きければ大きいほど、トランザクションの改竄コストも大きくなります。これは、トランザクションに対するあらゆる変更がハッシュによって再構築されなければいけないためであり、これが直接的・間接的にトランザクションのブロックを参照するためです。

レジャーのデザインには主に 2 つの目的があります:

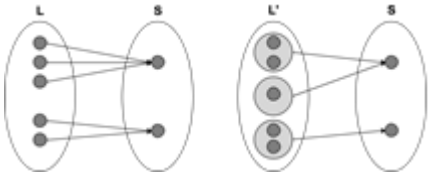

図 2：レジャーの統合

疑似フォーク率を低減：前項で触れたとおり、疑似フォークの発生率を低減する方法のひとつとして、同値類を作成し、システムを同一の状態に導くアカウントグループを単一のアカウントにまとめてしまうという方法があります。上に示されている通り、疑似フォーク率の計算式に基づいて、左のレジャーの疑似フォーク率は$Pff = \left(\frac{3}{5}\right)^c + \left(\frac{2}{5}\right)^c - \frac{1}{5^{c-1}}$ となり、レジャースペースの統合の後、右のグラフの疑似フォーク率は$Pff' = \left(\frac{2}{3}\right)^c + \left(\frac{1}{3}\right)^c - \frac{1}{3^{c-1}}$ となっています。C > 1 であれば、Pf f ′ < Pf f であることが知られています。つまり、トランザクション間の半順序関係を最小化することで、より多くのトランザクションを連続して交換することができるのです。

改竄防止：トランザクション t がレジャーl 内で変更されるとき、帳簿 l = l1 + l2 の 2 つのサブレジャー内で、サブレジャーl1 は影響を受けず、サブレジャーl2 内のハッシュ参照は新たに有効なレジャーl ′ = l1 + l2 ′ を作成するために再構築されなければいけません。影響を受けたサブレジャー l2 = ⌈(T2)については、T2 = {x|x ∈ T, x > t}です。したがって、トランザクションの改竄コストを増加させるため、トランザクション間の半順序関係を可能な限り多く維持し、改竄する|T2|の範囲を拡張することが必要です。

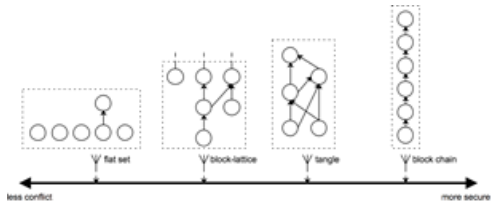

図 3：レジャー構造の比較 i

上記 2 つの目的は互いに相反するものであり、アカウント構造をデザインする上では、それなりの対価が必要となります。アカウントの維持がトランザクション間の半順序であるため、これは当然、半順序集合（ポセット）[19]ということになり、ハッセ図で示す[20]とき、トポロジー上の DAG になります。

上の図では、いくつかの共通したレジャー構造を比較しています。左側のレジャーは半順序が少なく維持されています。ハッセ図はフラットで、疑似フォーク率も低くなっています。右側のレジャーはより多くの半順序関係が維持されており、ハッセ図はよりスレンダーで、より改竄に対して防御力があることを示しています。

図において、最も左にあるのは、中央集権システムにおける、あらゆる改竄防止機構を持たない、共通のセットに基づいた構造です。最も右にあるのが、最高の改竄防止機構を持つ一般的なブロックチェーンレジャーです。両者の間には 2 つの DAG レジャーがあり、左には Nano が用いている block-lattice（ブロックラティス）アカウント[10]、右側には IOTA が用いている tangle book（タングルブック）[9]があります。特長の面からすると、ブロックラティスには半順序関係がより少なく、高パフォーマンスの分散型アプリケーションプラットフォームのアカウント構造により向いています。ただし、改竄に対しては弱いため、セキュリティリスクに暴露する可能性があり、現在のところ、このレジャー構造を採用しているプロ

ジェクトは Nano だけとなっています。
高パフォーマンスを追求するため、Vite は DAG レジャー構造を採用しています。同時に、追加のチェーン構造 Snapshot Chain を導入し、コンセンサスアルゴリズムを改善することによって、ブロックラティスのセキュリティにおける不安点を完全に穴埋めすることに成功しました。この改善点については後ほど詳細をご説明します。

2.2 先行制約（Pre Constraint）

最初に、このレジャー構造を状態機械モデルとして用いる上での前提条件について見てみましょう。この構造は、全体の状態マシンの組み合わせを個別の状態マシンのセットとして組み合わせることにより成り立っており、それぞれのアカウントは独立した状態機械に対応していて、それぞれのトランザクションは特定の 1 つのアカウントの状態以外には影響を与えません。レジャー内で、全てのトランザクションはアカウントにグループ化され、同じアカウント内でトランザクションのチェーンに統合されます。したがって、Vite における状態 S とトランザクション T には以下の 4 つの制限が生まれます:

定義 2.1（単一自由度の制限）：システム状態 s ∈ S は、それぞれのアカウントの状態 si により生成されたベクターs = (s1, s2, ..., sn)です。∀ti ∈ T であるため、トランザクション ti を実行後、システム状態は(s1 ′ , ..., si ′ , ..., sn ′) = σ(ti ,(s1, ..., si , ..., sn))といったように遷移し、sj ′ = sj , j /= i を満たす必要があります。この制限は、トランザクションに対する単一自由度の制限と呼ばれます。
直感的には、単一自由度のトランザクションは、システム内のその他のアカウントの状態に影響を与えず、1 つのアカウントの状態のみを変更します。状態空間のベクターが位置している多次元空間では、トランザクションが実行され、システムの状態は座標軸に平行の方向に沿って移動します。この定義は、Bitcoin や Ethereum などのモデルにおけるトランザクション定義よりも厳密であることに注意してください。Bitcoin のトランザクションは、送信者と受信者の 2 つのアカウントの状態に変更を加えます。Ethereum は message call（メッセージコール）を介して、2 つのアカウントより多くの状態に変更を加える場合があります。
この制約下なら、トランザクション間の関係性は単純化が可能です。2 つのトランザクションは、いずれも直角か平行のいずれかです。これにより、アカウントに従ってトランザクションをグループ化する状況が整います。以下、例で確認してみましょう:

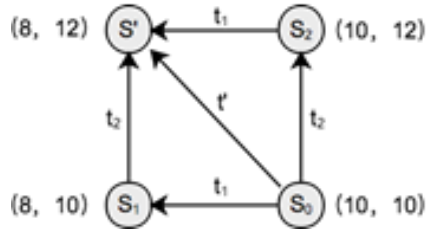

図 4：単一自由度の取引と中間状態

上図では、アリスとボブが 10USD をそれぞれ有していると仮定されています。システムの初期状態は s0 = (10, 10)です。アリスが 2USD をボブに送金したいとき、Bitcoin および Ethereum のモデルにおいて、トランザクション t ′ は、システムを直接最終状態（0 t ′ → s ′ ）に入れることができます。
Vite の定義では、トランザクション t ′ はアリスとボブの 2 つのアカウントの状態を変更しましたが、単一自由度の原則に従っていませんでした。したがって、トランザクションは 2 つのトランザクションに分裂することになります:

1) アリスにより送金された 2USD を示すトランザクション t1
2) ボブが受け取った 2USD を示すトランザクション t2

このようにして、初期状態から最終状態 s ′ になる中で、異なる 2 つのパス（ s0 → t1 s1 → t2 s ′ と s0 → t2 s2 → t1 s ′ ）が発生しうるのです。この 2 つのパスはそれぞれ中間状態 s1 および s2 を通過

しており、この 2 つの中間状態は 2 つのアカウント次元において最終状態 s ′ のマッピングとなります。つまり、アカウントのうちの 1 つの状態が重要である場合、そのアカウントに関連する全てのトランザクションを実行するだけで良く、その他のアカウントのトランザクションを行う必要はないのです。
次に、Ethereum におけるトランザクションを、Vite 求められる単一自由度のトランザクションに分割する方法について定義します:

定義 2.2（トランザクションの分解）：1 より大きな自由度を持つトランザクションを単一トランザクションのセットに分解することで、これをトランザクションの分解（Transaction Decomposition）と言います。遷移トランザクションは送信トランザクションと受信トランザクションに分けることができます。コントラクト呼び出しトランザクションはコントラクト要求トランザクションとコントラクト反応トランザクションに分けることができます。それぞれのコントラクト内のメッセージコールはコントラクト要求トランザクションとコントラクト反応トランザクションに分けることができます。
このようにして、レジャー内に異なる 2 つのトランザクションが生まれます。これは取引ペア（Trading pairs）と呼ばれます。

定義 2.3（取引ペア）：送信トランザクションまたはコントラクト要求トランザクションは、要求トランザクションと総称されます。受信トランザクションとコントラクト反応トランザクションは、反応トランザクションと総称されます。要求トランザクションと対応する反応トランザクションは、トランザクションペアと呼ばれます。トランザクション t の要求を開始するためのアカウントは、A(t)として記録されます。対応する反応トランザクションは et として記録されます。このトランザクションに対応するアカウントは A(et)として記録されます。
上記の定義に基づいて、Vite の 2 つのトランザクション間についてありうる関連性を定義することができます。:

定義 2.4（トランザクションの関係）：t1 および t2 という 2 つのトランザクションについて、以下のような関係性が想定されます。:
直角：A(t1) /= A(t2)なら、その 2 つのトランザクションは直角であり、t1 ⊥ t2 として記録されます。
平行：A(t1) = A(t2)なら、その 2 つのトランザクションは平行であり、 t1 // t2 として記録されます。
因果関係：t2 = te1 なら、その 2 つのトランザクションは因果関係にあり、 t1 ▷ t2 または t2 ◁ t1 として記録されます。

2.3 レジャーの定義

レジャーを定義することは、ポセットを定義することです。まず、Vite 内のトランザクション間の半順序関係を定義します。:
定義 2.5（トランザクションの半順序) 私たちは二次元的関係<を用いて 2 つのトランザクションの半順序関係を表します。
反応トランザクションは、対応する要求トランザクションに続かなければなりません:t1 ≺ t2 t1 ▷ t2;
アカウントにおける全てのトランザクションは厳密かつ全体的に順序立てられます。∀t1 // t2 ならば、:t1 ≺ t2 または t2 ≺ t1 でなければなりません。
トランザクション一式に形成される半順序関係により、T は特性を満たします:

- I 非反射: ∀t ∈ T ,there is no t ≺ t;
- 推移的：∀t1, t2, t3 ∈ T で、if t1 ≺ t2, t2 ≺ t3 ならば、t1 ≺ t3
- 非対称的：∀t1, t2 ∈ T で、if t1 ≺ t2 ならば、t2 ≺ t1 は存在しない

このようにして、Vite アカウントを厳密な半順序セットにおいて定義できます:
定義 2.6（Vite レジャー）：Vite レジャーは、特定のトランザクション T のセットから成る厳密なポセット、および部分的なポセットです。

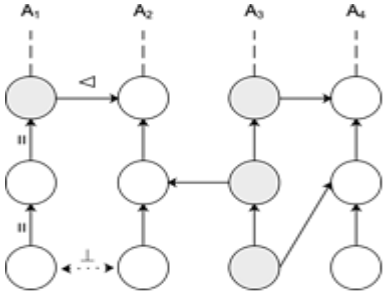

図5：Vite におけるレジャーとトランザクション間の関係性

厳密なポセットは DAG 構造に対応できます。上図では、円はトランザクションを、矢印はトランザクション間の依存を表しています。Ab は a が b に依存していることを表します。

上図で定義される Vite レジャーは構造的にブロックラティスに類似しています。トランザクションは要求トランザクションと反応トランザクションに分けられ、それぞれが個別のブロックに対応しており、それぞれのアカウント Ai はチェーン、トランザクションペア、対応する要求トランザクションのハッシュを参照する反応トランザクションに対応します。

3 スナップショットチェーン（Snapshot chain)

3.1 トランザクションの確認

アカウントがフォークしたとき、コンセンサスの結果が 2 つのフォークしたレジャー間でぶれることがあります。例えば、ブロックチェーン構造に基づいたとき、あるノードがより長いフォークしたチェーンを受信した場合、新たなフォークがコンセンサスの結果として選択されることになり、元々のフォークは却下され、元々のフォークにあったトランザクションはロールバックされることとなります。そのようなシステムでは、トランザクションのロールバックは重大なイベントであり、二重支払いに繋がる場合もあります。ある事業が支払いを受けた痕、商品やサービスを提供し、その支払いが取り消されたなら、売り手は損失を被ることとなります。したがって、ユーザーが支払いトランザクションを受信するときには、ロールバックの可能性が可能な限り低くなるように、システムがトランザクションを"確認"するまで待つことが必要になります。

定義 3.1（トランザクションの確認）：ロールバックされるトランザクションが発生する確率が特定の閾値 ϵ よりも低いとき、トランザクションは確認されたとします。$Pr(t) < \epsilon \Leftrightarrow t$ は確認された状態です。

トランザクションの確認が非常に複雑なコンセプトであるのは、トランザクションが認識されるかどうかは、暗黙の確信レベル（$1 - \epsilon$）に事実上依存しているからです。ダイアモンドを販売する売り手とコーヒーを販売する売り手は、二重支払いによる被害を受けた場合の損失に違いがあります。したがって、前者はトランザクションにおける ϵ の値を小さく設定することになります。これは Bitcoin における確認の番号の要素でもあります。Bitcoin においては、確認番号はブロックチェーン内のあるトランザクションの深さを示しています。確認番号が大きければ大きいほど、トランザクションがロールバックされる可能性が低いことになります[2]。したがって、売り手は、確認番号の待ち番号を設定することで、確認の確信レベルを間接的に設定することになります。

トランザクションのロールバックの可能性は、そのアカウント構造におけるハッシュ参照関係の時間によって減少します。既に言及したように、レジャーのデザインが改竄に対して強い性質を有しているなら、それだけトランザクションのロールバックにおいて、ブロックにおけるやり取りの全てのシークエンスブロックを再構築する必要があります。新たなトランザクションが継続的にレジャーに追加されるとき、トランザクション内にはそれだけ多く連続したノードが発生し、改竄される可能性が低くなるのです。

ブラックラティス構造においては、トランザクションはアカウントによりグループ化されるため、トランザクションは自身のアカウントのアカウントチェーンの末端以外に添付されることがなく、最も多くのその他のアカウントにより生成されたトランザクションがトランザクションの継続ノードに自動的になるということもありません。したがって、コンセンサスアルゴリズムを慎重にデザインし、二重支払いの一見して分からない危険性を避けることが必要となります。

Nano は投票に基づいたコンセンサスアルゴリズム[10]を採用しており、トランザクションは、ユーザーグループにより選択された代表ノードのセットによる署名を得ることになります。それぞれの代表ノードには重さ（ウェイト）があり、あるトランザクションへの署名が充分な重さを持つとき、そのトランザクションは確認されたものと見なされます。このアルゴリズムには、以下のような問題があります。:

まず、確認についてより大きな確信度レベルが必要なとき、投票のウェイトの閾値もそれに従って上昇します。充分な代表ノードがオンラインになっていないときには、区分スピードが確保されない場合があり、やり取りを確認するために必要なだけのチケット数をユーザーが集めることができないという可能性が発生します。;

また、トランザクションがロールバックされる可能性が時間によって低下しません。これは、過去の投票結果を覆す

ためのコストは常に同じであるためです。

最後に、過去の投票結果はレジャーに残らず、ノードのローカルストレージにしか保存されないという点です。ノードが他のノードからアカウントを得るとき、過去のトランザクションがロールバックされる可能性を定量化するのに信用できる方法がありません。

まとめると、投票のメカニズムは部分的な中央集権的ソリューションであると言えます。投票結果をレジャーの状態のスナップショットとして見ることが可能です。このスナップショットは、ネットワークにおけるそれぞれのノードのローカルストレージ内に分散されます。ブロックチェーンで同様の改竄防止能力を実現するため、こうしたスナップショットを、Vite デザインのカーネルのひとつであるチェーン構造（スナップショットチェーン[21]）に取り入れることも可能です。

3.2 スナップショットチェーンの定義

スナップショットチェーンは、Vite における最も重要なストレージ構造です。その主な機能は Vite レジャーのコンセンサスを維持することにあります。まず、スナップショットチェーンの定義をご説明します:

定義 3.2（スナップショットブロックおよびスナップショットチェーン）：アカウントのバランス、コントラクト状態の Merkle ルート、それぞれのアカウントチェーンの最後のブロックのハッシュを含む、ある Vite レジャーの状態スナップショットを保存するスナップショットブロックです。スナップショットチェーンとは、スナップショットブロックから成るチェーン構造で、次のスナップショットブロックは以前のスナップショットブロックのハッシュを参照します。

ユーザーアカウントの状態には、そのアカウントチェーンの最後のブロックのハッシュとバランスが含まれています。上記 2 つの領域に加え、コントラクトアカウントの状態には、その Merkle ルートハッシュが含まれます。アカウントの状態の構造は、次のようになります:

```
struct AccountState {
// account balance
map<uint32, uint256> balances;
//Merklerootofthecontractstate optional uint256storageRoot;
// hash of the last transaction
//oftheaccountchain uint256  lastTransaction;
}
```

スナップショットブロックの構造は以下のように定義されます。:

```
struct SnapshotBlock {
//hashofthepreviousblock uint256 prevHash;
// snapshot information
map<address, AccountState>snapshot;
// signature uint256signature;
}
```

複数のトークンを同時にサポートするため、Vite のアカウント状態のバランス状態を記録する構造は uint256 ではなく、トークン ID からバランスへのマッピングとなっています。

スナップショットチェーンの最初のスナップショットブロックは、″ジェネシススナップショット（Genesis snapshot）″と呼ばれます。これは、そのアカウント内のジェネシスブロックのスナップショットを保存します。

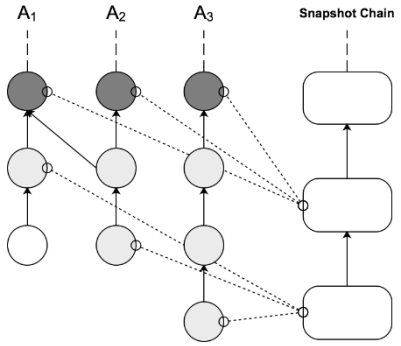

図6：スナップショットチェーン

スナップショットチェーンにおけるそれぞれのスナップショットブロックは、Vite レジャーの唯一のフォークに対応しているため、スナップショットブロックがスナップショットブロック内でフォークしないとき、スナップショットブロックによって Vite レジャーの結果にコンセンサスを得ることが可能となります。

3.3 スナップショットチェーンとトランザクションの確認

スナップショットチェーンを導入することで、ブロックラティス構造のセキュリティにあった欠点を解消できます。攻撃者が二重支払いのトランザクションを生成しようとしても、Vite レジャーにおけるハッシュ参照を再構築する必要に加え、そのトランザクションの最初のスナップショットブロックの後の全てのブロックについてスナップショットチェーンを再構築しなければならず、また、より長いスナップショットチェーンを生成しなければなりません。こうして、攻撃のコストが非常に高くなるのです。

Vite では、トランザクションの確認メカニズムは Bitcoin と類似しており、以下のように定義できます。:

定義 3.3（Vite のトランザクション確認）：Vite では、トランザクションがスナップショットチェーンによって瞬間記録された場合、トランザクションが確認されます。最初のスナップショットにおけるスナップショットブロックの深度が、そのトランザクションの確認番号となります。

この定義により、確認されたトランザクションの数は、スナップショットチェーンが成長するにつれて 1 ずつ上昇し、二重支払いの攻撃の可能性はスナップショットチェーンの増加によって減少していきます。このようにして、ユーザーは、異なる確認番号を特定のシナリオに合わせて待つことで、必要なだけの確認番号をカスタマイズすることが可能です。

スナップショットチェーンそれ自体はコンセンサスアルゴリズムによるものです。スナップショットチェーンがフォークした場合、もっと長いフォークが有効なフォークとして選択されます。スナップショットチェーンが新たなフォークにスイッチした場合、元々のスナップショットの情報はロールバックされますので、そのレジャーにおける元々のコンセンサスは無かったことになり、新たなコンセンサスに置換されます。したがって、スナップショットチェーンは全体のシステムセキュリティの柱として、重大に扱わなければいけないものと言えます。

3.4 圧縮ストレージ

全てのアカウント状態がスナップショットチェーン内の全てのスナップショットブロックに保存されなければいけないため、ストレージ空間が非常に大きくなります。このため、スナップショットチェーンへの圧縮は不可欠です。

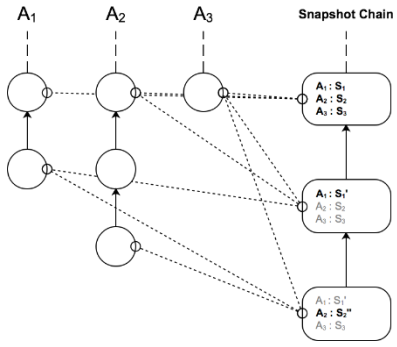

図7：圧縮前のスナップショット

スナップショットチェーンのストレージ空間の圧縮における基本的なアプローチは、増分ストレージを用いることです。スナップショットブロックは、以前のスナップショットブロックと比較して変更が加えられたデータのみを保存します。2 つのスナップショット間で 1 つのアカウントについてトランザクションが無かった場合、後者のスナップショットブロックはそのアカウントのデータを保存しません。

スナップショット情報を回復するため、最初から最後まで、スナップショットブロックを否認し、現在のデータによ

って全てのスナップショットブロックのデータをカバーすることもできます。

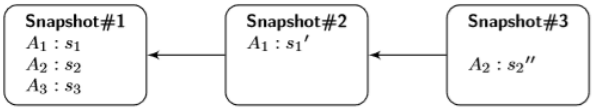

図8：圧縮後のスナップショット

スナップショットでは、1 つのアカウントのそれぞれのスナップショット の最終状態だけが保存され、中間状態は考慮されません。つまり、2 つのスナップショット間で、アカウントによって生成されたトランザクションがどれだけあっても、スナップショットのデータのコピーは 1 つしか保存されません。したがって、スナップショットのブロックは最大で S * A バイトとなります。この中で、S=sizeof(si)はそれぞれのアカウント状態を占めるバイト数を、A はシステムアカウントの合計数を表しています。アクティブアカウントと全体アカウントの平均比を a とすると、圧縮比は 1-a となります。

4 合意(コンセンサス)

4.1 設計の目標

合意のプロトコルを設計するに当たって、次の事項を必要とする:

- 性能。Vite の主な目的は速い事である。システムの高い処理能力と少ない遅延の処理を確実なものとするために、速い収束速度のコンセンサスアルゴリズムを採用する必要がある。
- 拡張可能性。Vite は公共のプラットフォームであり全ての非集中化応用に対して開かれていなければならず、そのため拡張性の考慮もまた重要である。
- 安全性。Vite の設計原則は究極の安全性を追求することではないが、しかし、十分な安全性の基礎を確実なものとし、全種類の攻撃に対して効果的に防衛する必要がある。

いくらかの既存のコンセンサスアルゴリズムと比較して、PoW(プルーフ・オブ・ワーク)の確保がよりよいものであり、悪意のあるノードの計算能力が５０%以下であればコンセンサスに達することができる。しかし、交差する PoW の速度が遅く要求性能に合致できないと、PoS(プルーフ・オブ・ステーク）とその変形アルゴリズムはその数学的な問題を解くステップを除去し、交差する速度を改善して単一攻撃のコストとし、エネルギー消費を削減する。しかし、PoS の拡張性は未だ乏しく、「出資者に無し」問題 [22] は解決が困難である。BFT アルゴリズムは安全性と性能においてより良い性能を持つが、その拡張性には問題ががあり、通常私的なチェーンまたは共同体のチェーンに適している。DPoS [11] シリーズのアルゴリズムは虚偽の分岐となる可能性をブロック生成の許可を制限することで効果的に減らす。性能と拡張性も良い。結論として、DPoS が安全性に少し犠牲を強いるが、悪意あるノードの数は 1/3 以上にはならない[23]。

一般的に DPoS アルゴリズムは性能と拡張性であきらかに優位にある。したがって、DPoS を Vite のコンセンサスプロトコルの基礎とし、それをベースにして正しく拡張する。階層的な委任のコンセンサスプロトコルと非同期モデルによって、プラットフォームの総合的な性能は更に改善できる。

4.2 階層的コンセンサス

Vite のコンセンサスプロトコルは HDPoS（階層的委任でのプルーフ・オブ・ステーク）である。基本的なアイデアはコンセンサス関数 Φ（機能分割）を分解することである:

$$\Phi(l_1, l_2, \ldots, l_n) = \Psi(\Lambda_1(l_1, l_2, \ldots, l_n), \Lambda_2(l_1, l_2, \ldots, l_n), \ldots \Lambda_m(l_1, l_2, \ldots, l_n)) \quad (5)$$

Λi: 2^L L, はローカルコンセンサス関数と呼ばれ、それが戻す結果はローカルコンセンサスと呼ばれる：Ψ : 2L → L は全体的コンセンサス関数として知られ、ローカルコンセンサス中の候補グループからのユニークな結果を最

終のコンセンサス結果として選択する。

最終的なコンセンサス結果として、ローカルコンセンサスの候補者グループからの結果。

この分離の後で、全体システムのコンセンサスは２つの独立な処理となる:

ローカルコンセンサスは要求取引に対応するブロックを生成し、ユーザーのアカウントまたはコントラクトアカウントでの取引に応答し、台帳に書き込まれる。

グローバルコンセンサスは台帳中のデータをスナップショットし、スナップショットブロックを生成する。台帳が分岐していればその一方を選ぶ。

4.3 ブロック生成とコンセンサスグループの権利

では、誰が元帳にトランザクションブロックを生成し、スナップショットチェーンにスナップショットブロックを生成する権利があるのでしょうか？
コンセンサスに達するためにどのコンセンサスアルゴリズムが採用されているのでしょうか？
Vite の元帳構造は、各アカウントに応じてそれぞれがマルチプルアカウントチェーンに編成されています
勘定科目の定義に従って、元帳のブロックの生成権と、スナップショットブロックの生成権の両方を単一のユーザーグループが有するように定義することができます。
これにより、多数のアカウントチェーンまたはスナップショットチェーンをコンセンサスグループにまとめることができます。また、私たちは統一された方法でブロックを生成し、コンセンサスに達することができます。

定義 4.1（コンセンサスグループ）　コンセンサスグループは４つ組 (L,U,Φ,P) で表わされ、アカウントまたはスナップショットのチェーンの一部分のコンセンサス構造を記述する；L ∈ A|{As} は１つあるいはいくつかのアカウントのチェーン、または台帳のコンセンサスグループのスナップショットのチェーン；U は L で特定されたチェーン上のブロックを生成する権利を持つユーザーを表し；Φ はコンセンサスグループのコンセンサスアルゴリズムを特定し、P はアコンセンサスルゴリズムのパラメーターを特定する。

この定義の下で、ユーザーは柔軟にコンセンサスグループを設定し、ニーズに応じてさまざまなコンセンサスパラメーターを選択できます。 次に、さまざまなコンセンサスグループについて詳しく説明します。

4.3.1 スナップショットのコンセンサスグループ

スナップショットチェーンのコンセンサスグループはスナップショットコンセンサスグループと呼ばれ、Vite でもっとも重要なコンセンサスグループである。スナップショットコンセンサスグループのアコンセンサスルゴリズム Φ は DPoS アルゴリズムを採用し階層モデル中の Ψ に対応する。代理人の数とブロック生成の間隔はパラメーター P で特定される。

例えば、スナップショットコンセンサスグループを 25 の代理ノードで、スナップショットブロックを１秒の間隔と指定することができる。こうして確実に取引が充分速いと確認できる。１０回の取引きの確認は最大１０秒待てばよい。

4.3.2 個人グコンセンサスループ

個人コンセンサスグループは台帳の取引きブロックの生成に適用されるだけであり、個人コンセンサスグループのアカウントチェーンに属する。アカウントの個人キーの所有者だけが生成することができる。既定では、全てのユーザーのアカウントは個人コンセンサスグループに属する。

個人コンセンサスグループの最大の利点は分岐の可能性が減少することである。一人のユーザーのみがブロックを生成する権利を持っているので、分岐する可能性は個人的に二重消費を仕掛けるかまたはプログラムのエラーに限られる。

個人コンセンサスグループの不利な点は、ユーザーのノードが取引を梱包することができる前にオンラインでなければならないことである。これは同意アカウントにとって非常に適しているとは言えない。オーナーのノードが失敗すると、他のどのノードもコントラクトを生成する取引の応答を置き直すことができず、これは dApp のサービスの可能性を減らすことに等しい。

4.3.3 コンセンサスグループの委任

コンセンサスグループの委任に当たっては、ユーザーアカウントに代わって一組の指定された代理ノードが DPoS アルゴリズムによって取引を梱包するのに使われる。ユーザーアカウントとコントラクトアカウントのどちらも同意グループに追加できる。ユーザーは一組の別の代理ノードを設定し新しいコンセンサスグループを確立できる。Vite にはまた既定の公共コンセンサスグループとして知られるコンセンサスグループがあって、まだ個別に委任するコンセンサスグループを確立していない他の全てのアカウントの取引きを梱包するのを助ける。

コンセンサスグループの委任はほとんどのコントラクトアカウントに適している、それはコントラクトアカウントでのほとんどの取引きはコントラクト応答取引であり、そこではユーザーアカウントにおける受取取引より高い利用可能性と少ない遅延が必要とされるからである。

4.4 コンセンサス の優先度

Vite プロトコルにおける、グローバルコンセンサスの優先度はローカルコンセンサスの優先度より高い。ローカルコンセンサスが分岐した場合、グローバルコンセンサスを選択する結果が優勢となる。いい変えれば、一旦グローバルコンセンサスが選択されると、最終結果としてのローカルコンセンサスの分岐はは、以降のアカウントに確定したアカウントのチェーンのさらに長い分岐が起こったとしても、グローバルコンセンサスの結果を引き戻すことにはならない。この問題は、クロスチェーンプロトコルを実装するときにもっと注意する必要がある。目標チェーンは巻き戻る可能性があるため、チェーンをマッピングするリレーコントラクトに対応するアカウントチェーンもそれに応じて巻き戻す必要がある。現時点で、リレーチェーンのローカルコンセンサスがグローバルコンセンサスによって採用されている場合、巻き戻しを完了することは不可能であり、それによってリレーコントラクトとターゲットチェーン間のデータが矛盾する可能性がある。
スナップショットを作成するスナップショットコンセンサスグループを指定し、遅延ブロック後にローカルコンセンサスのみが完成するように指定する。これにより、リレーコントラクトの不整合の可能性が大幅に減少するが、完全に回避することはできない。リレーコントラクトのコード論理では、ターゲットチェーンの巻き戻しを個別に処理することもまた必要である。

4.5 非同期モデル

システムのスループットをさらに向上させるためには、コンセンサスメカニズムに関するより完全な非同期モデルをサポートする必要がある。
取引のライフサイクルには、取引の開始、取引の書き込み、および取引の確認が含まれる。システムのパフォーマンスを向上させるために、これら 3 つのステップを非同期モードに設計する必要がある。これは、異なる時間には、ユーザによって開始された取引の量が異なり、システムによって処理される取引の書き込み速度および取引確認の速度が相対的に固定されるためである。非同期モードは山と谷を平らにするのを助け、システムの全体的なスループットを改善する。
Bitcoin と Ethereum の非同期モデルは単純である。すべてのユーザーによって開始された取引は未確認プールに入る。マイナーがそれをブロックにまとめると、その取引は同時に書かれ確認される。ブロックチェーンが拡大し続けると、取引は最終的に事前設定された確認信頼レベルに達する。
この非同期モデルには２つの問題がある:

- 取引は未確認の状態でレジャーに自動記録されている。認識されていない取引は不安定であり、関与するコンセンサスはない、それは取引の繰り返し送信を防ぐことはできない。
- 取引の書き込みと確認のための非同期メカニズムはない。取引は確認時にのみ書き込まれ、書き込み速度は確認速度によって制限される。

Vite プロトコルは、より改善された非同期モデルを確立する。まず、取引が転送請求であれコントラクト請求であれ、取引は「要求 - 応答」モデルに基づいて取引ペアに分割され、取引がレジャーに書き込まれたときに、取引が成功裏に発行される。さらに、取引の書面による確認も非同期である。取引は最初に Vite の DAG アカウントに書き込むことができ、確認プロセスによってブロックされることはない。取引の確認はスナップショットチェーンを通じて行われ、スナップショットアクションも非同期である。

これは典型的な生産者 - 消費者モデルである。取引のライフサイクルでは、上流で生産率がどのように変化しても、ダウンストリームはプラットフォームリソースを完全に利用してシステムのスループットを向上させるために、一定のレートで取引を処理できる。

5 仮想マシン

5.1 EVM 互換性

現在、Ethereum 分野には多くの開発者がいて、多くのスマートコントラクトが堅実性と EVM に基づいて適用されている。したがって、Vite 仮想マシンで EVM 互換性を提供することにした。ほとんどの EVM 命令セットの元のセマンティクスは Vite でも保たれている。 Vite のアカウント構造と取引定義は Ethereum とは異なるため、一部の EVM 命令のセマンティクス、たとえばブロック情報を取得するための一連の命令を再定義する必要がある。詳細なセマンテックスの違いは、付録 A で参照できる。

それらのうち、最大の違いはメッセージ呼び出しのセマンティクスであり。次に詳細に説明する。

5.2 イベント駆動

Ethereum のプロトコルでは、取引またはメッセージが複数のアカウントの状態に影響を与える可能性がある。たとえば、コントラクト呼び出し取引では、メッセージ呼び出しを通じて複数のコントラクトアカウントのステータスが同時に変更されることがある。これらの変更は同時に発生するか、またはまったく発生しない。したがって、Ethereum での取引は実際には ACID の特性（原始性、一貫性、分離、堅牢制）[24]を満たす一種の厳密な取引であり、これも Ethereum での拡張性の欠如の重要な理由である。

拡張性とパフォーマンスの考慮に基づいて、Vite は BASE（基本的に利用可能、ソフト状態、最終的な一貫性）[25]セマンティックスを満たす最終的な一貫性スキームを採用した。具体的には、Vite をイベント駆動型アーキテクチャ（EDA）として設計している[26]。各スマートコントラクトは独立したサービスと見なされ、メッセージはコントラクト間で通信できるが、状態は共有されない。

したがって、Vite の EVM では、コントラクト間の同期ファンクション呼び出しのセマンティックスは取り消し、そしてコントラクト間のメッセージ通信だけを可能とした。影響を受けた EVM の命令は主に Vite EVM の CALL と STATICCALL で、これら２つの命令は直ちに実行できず、呼び出しに結果を戻すことも出来ない。それらはレジャーへ書き込むための要求取引を生成するだけである。したがって Vite では、ファンクション・ コールのセマンティックスはこの命令に含まれていないが、アカウントにメッセージを送ることになる。

5.3 スマートコントラクトの言語

Ethereum は、スマートコントラクトを開発するための Turing 完全プログラミング言語 Solidity を提供する。非同期セマンティクスをサポートするために、Solidity を拡張し、メッセージ通信のためのシンタックスのセットを定義した。
拡張 Solidity は Solidity ++ と呼ばれる。

Solidity の構文の大部分は Solidity ++ によってサポートされているが、コントラクト外のファンクション呼び出しは含まない。開発者は、キーワード message を通じてメッセージを定義し、キーワード on を通じてメッセージプロセッサ（MessageHandler）を定義して、クロスコントラクトコミュニケーション機能を実装することができる。

例えば、コントラクト A がコントラクト B で add () メソッドをそのリターン値に基づいて状態を更新するために必要とする。Solidity ではそれはファンクションコールで実装できる。そのコードは次のようになる:

```
pragma solidity^0.4.0; contract B {
function add(uint a, uint b) returns
(uint ret) {
return a + b;
}
}

contract A {
```

```
uint total;

functioninvoker(addressaddr,uinta, uint b) {
// message call to A.add() uintsum=B(addr).add(a,b);
//usethereturnvalue if (sum > 10){
total += sum;
}
}
}
```

Solidity ++では、ファンクション・コール codeuint sum = B（addr）.add（a、b）;はもう有効ではない。その代わりに、コントラクト A とコントラクト B は互いにメッセージを送信することによって非同期的に通信する。そのコードは次のようになる:

```
pragmasolidity++^0.1.0; contract B {
message Add(uint a, uintb);
message Sum(uint sum);

Add.on {
// read message
uinta=msg.data.a; uintb=msg.data.b;
address sender =msg.sender;
// do things
uint sum = a +b;
//sendmessagetoreturnresult send(sender, Sum(sum));
}

}

contract A {
uint total;

functioninvoker(addressaddr,uinta, uint b) {
// message call to B send(addr, Add(a,b))
// you can do anything after sending
// a message other than using the
// return value
}
Sum.on {
//getreturndatafrommessage uint sum =msg.data.sum;
//usethereturndata if (sum > 10){
total += sum;
}
}
}
```

１行目で、code pragma solidity++ 0^.1.0; はそのソースコードが Solidity++ で書かれていることを示しているが、直接 Solidity のコンパイラーでコンパイルされるものではない、それはコンパイルされた EVM コードが期待したセマンティックスと整合しないことを避けるためである。Vite は Solidity++ をコンパイルするための特別なコンパイラを提供する予定である。このコンパイラーは部分的に前方コンパティブルである。Vite セマンティックスと矛盾する Solidity コードが無ければ直接コンパイルでき、そうでなければエラーが報告される。例えば、ローカルな他のアカウントに転送するファンクション・コールのシンタックスは互換性が保たれている；コントラクトに

跨るファンクション・コールの戻り値を得ること、また金銭単位の何れもコンパイルされない。
コントラクト A において、起動ファンクションが呼ばれたときに、Add メッセージがコントラクト B に送られるが、それは非同期で結果は即時には戻されない。したがって、A におけるそのキーワードを用いて戻った結果を受け取り状態を更新するようメッセージプロセッサーを定義する必要がある。
契コントラクト B では、メッセージ Add が監視される。処理後、Sum メッセージが結果を戻すためにメッセージ ADD の送り主に送られる。
Solidity ++のメッセージは CALL 命令にコンパイルされ、要求取引がレジャーに追加される。 Vite では、レジャーはコントラクト間の非同期通信のためのメッセージミドルウェアとして機能する。それによって信頼性の高いメッセージの保存を保証し、重複を防ぐ。同じコントラクトによって 1 つのコントラクトに送信された複数のメッセージは FIFO（先入れ先出し）を保証できるが、異なるコントラクトによって同じコントラクトに送信されたメッセージは FIFO を保証しない。
Solidity（Event）のイベントと Solidity ++のメッセージは同じ概念ではないことに注意すべきである。イベントは EVM ログを介して間接的に前面に送信される。

5.4 標準ライブラリ

Ethereum でスマートコントラクトを開発する開発者は、Solidity に標準ライブラリがないことにしばしば悩まされる。たとえば、ループプロトコル内でのループ検証をチェーンの外側で実行しなければならない。また重要な理由の 1 つは、浮動小数点演算関数が Solidity で提供されていないことであり、特に浮動小数点数の平方根[1][1]がサポートされていない。
EVM では、ライブラリファンクションの機能を実現するために、DELEGATECALL コマンドで事前に配備されたコントラクトを呼び出すことができる。 また Ethereum はいくつかの事前にコンパイルされたコントラクトを提供しているが、これは主にいくつかのハッシュ操作である。しかし、これらの機能は複雑なアプリケーションのニーズを満たすには単純すぎる。

したがって、Solid ++ では一連の標準ライブラリ、文字列処理、浮動小数点演算、基本的な数学演算、コンテナ、ソートなどを提供する。

パフォーマンスの観点から、これらの標準ライブラリはローカルエクステンション（Native Extension）方式で実装され、ほとんどの操作は Vite ローカルコードに組み込まれており、関数は EVM コードの DELEGATECALL 命令を介してのみ呼び出される。

標準ライブラリは必要に応じて拡張することができるが、システム全体のステートマシンモデルは決定論的であるため、乱数のような機能を提供することはできない。Ethereum と同様に、スナップショットチェーンのハッシュを通じて擬似乱数をシミュレートできる。

5.5 ガス

Ethereum のガスには 2 つの主要な機能がある。1 つ目は、EVM コードの実行によって消費されるコンピューティングリソースとストレージリソースを定量化することである。2 つ目は、EVM コードを確実に停止させることである。計算可能性理論によると、チューリング機械の停止問題は計算不可能な問題である[27]。それは、EVM コードを分析することによって、限定された実行の後にスマートコントラクトを停止することができるかどうかを決定することが不可能であることを意味する。
したがって、Vite での EVM のガス計算は保たれている。ただし、Vite にはガス価格の概念はない。ユーザーは、料金を支払って交換用のガスを購入することはないが、計算リソースを得るための割当てベースによる。割り当ての計算は「経済モデル」の章で詳しく論じる。

6 経済モデル

6.1 ネイティブ・トークン

プラットフォームのコンピューティングリソースとストレージリソースを定量化し、ノードの実行を促進するために、Vite はネイティブのトークン ViteToken を構築した。トークンの基本ユニットは Vite、最小ユニットは attov で 1 Vite = 1018 attov である。

スナップショットチェーンは、Vite プラットフォームのセキュリティとパフォーマンスにとって重要である。ノードにトランザクション検証への参加を促すために、Vite プロトコルはスナップショットブロックの生成に対してプレス報酬を設定する。

逆に、ユーザーが新しいトークンを発行したり、コントラクトを展開したり、VNS ドメイン名を登録したり、リソースクォータを取得したりする場合は、ViteToken を使用するか、担保とする必要がある。

これら 2 つの要素が組み合わさって、システムリソースの割り当てを最適化するのに役立つ。

6.2 リソース割り当て

Vite は一般的な dApp プラットフォームであるため、それらに展開されているスマートコントラクトの機能はさまざまで、スマートコントラクトごとにスループットと遅延の要件が異なる。同じスマートコントラクトであっても、異なる段階でのパフォーマンス要件は異なる。

Ethereum の設計では、他のトランザクションと競合してアカウントを作成するために、起動時に各トランザクションにガス価格を割り当てる必要がある。これは、原則として需給バランスを効果的に制御できる典型的な入札モデルである。しかし、ユーザーは現在の需給状況を定量化することが難しく、他の競合他社の価格を予測することができないため、市場での失敗が発生しやすくなる。さらに、各入札のために競合するリソースは一つの取引きに向けられ、アカウントの大きさに従ったリソースの合理的割り当てについてのコンセンサスもない。

6.2.1 分配計算

Vite では、割当てベースのリソース分配プロトコルを採用した。これにより、ユーザーは次の 3 つの方法でより高いリソース割り当てを取得できる:

- 取引が初期化されたときに PoW（プルーフ・オブ・ワーク）が計算される ;
- アカウントに一定量の Vite を抵当に入れる;
- 一度に少量の Vite を破壊する。特定の割り当て量は、次の式で計算できる:

$$Q = Q_m \cdot \left(\frac{2}{1 + exp\left(-\rho \times \xi^{\top}\right)} - 1 \right) \qquad (6)$$

ここで、Qm は定数であり、単一アカウント割り当て量の上限を表す。これは、システムの総スループットとアカウントの総数に関係する。ξ=（ξ、d、ξ、s、ξ、f）は、次の式を表すベクトルである。リソースを入手するためのユーザーのコスト：ξ dは、ユーザーが取引を生成するときに計算する PoW の難易度であり、ξ sは、口座内の抵当の残高であり、そして ξ fは、ユーザーが割当ての増加に望んでいる一回限りのコストである。 ξ f は取扱手数料とは異なり、これらの Vite はマイナーに支払われるのではなく直接抹殺されることになる。

この式において、ベクトル ρ＝（ρ d、ρ s、ρ f）は、割当てを得るための３つの方法の重みを表し、すなわち、１Vite の破壊によって得られる割当ては、抵当にした ρ s/ ρ f Vite に等しい。

この式から、ユーザーが Vite を担保にすることも一度限りの費用を支払うこともしなければ、PoW を計算する必要があることが分る。そうでなければ取引を初期化するための割当ては無く、その割当ては効果的に混乱が襲うのを防止し、システムのリソースを不正使用されることを防止できるものである。同時に、この式はロジスティック関数である。ユーザーが比較的低い割当てを使用することは比較的簡単で、低頻度ユーザーのしきい値を減らすことができる；高頻度ユーザーはより高い割当てを取得するために多くのリソースを投資する必要がある。彼らが払う追加のコストはすべてのユーザーの利益を増加させることになる。

6.2.2 リソースの定量化

スナップショットチェーンはグローバルクロックと同等であるため、これを使用してアカウントのリソース使用量を

正確に定量化できる。各トランザクションでは、スナップショットブロックのハッシュが引用され、スナップショットブロックの高さがトランザクションのタイムスタンプとして使用される。したがって、2 つのトランザクションタイムスタンプの違いに基づいて、2 つのトランザクション間の間隔が十分に長いかどうかを判断できる。

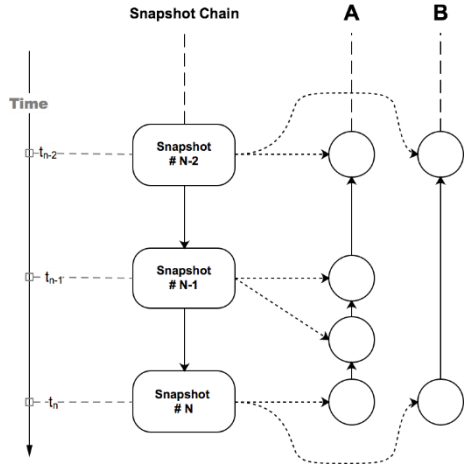

図９：グローバルクロックとしてのスナップショットチェーン

上記のように、アカウント A は２つの時間間隔で 4 つのトランザクションを生成したが、アカウント B は２つのトランザクションのみを生成した。したがって、この期間の A の平均 TPS は B の２倍である。これが単に送金取引であれば、定量化したアカウントの平均の TPS は十分である。スマートコントラクトでは、各取引所ごとにリソースの消費量が異なるため、一定期間の平均リソース消費量を計算するには、トランザクションごとにガスを累積する必要がある。あるアカウントチェーンで「n」の長さを持つ最近の k 取引の平均リソース消費は次にようになる:

$$Cost_k(T_n) = \frac{k \cdot \sum_{i=n-k+1}^{n} gas_i}{timestamp_n - timestamp_{n-k+1} + 1} \qquad (7)$$

この中で、トランザクション Tn について、タイムスタンプ n はその取引のタイムスタンプ、つまり参照するスナップショットブロックの高さである。ガス n は取引のために消費されるガスである。

トランザクションを検証するとき、ノードは割当てが次の条件を満たすかどうかを判断する。Cost（T）≤Q。それが満たされない場合、トランザクションは拒否される。この場合、ユーザは取引を再パッケージ化し、1 回限りの手数料を払うか、または取引内のより高いスナップショットを割り当てるために一定期間待つことで割当てを増加する必要がある。

6.2.3 割当てのリース

ユーザーが十分な数の Vite 資産を持っていても、それほど多くのリソース割り当て量を使用する必要がない場合、彼は自分の割り当て量を他のユーザーに貸すことを選択できる。
Vite システムは、アカウントリソース割当てを使用する権利を移転するための特別なタイプの取引きをサポートする。このトランザクションでは、抵当に入れることができる Vite 数、譲受人の住所、およびリース期間を指定することができる。取引が確認されると、トークンの金額に対応するリソース割り当て量が譲受人のアカウントに含まれる。リース期間を超えると、その割り当て量が計算され移転元口座に含まれる。リース時間の単位は秒であるが、システムはスナップショットブロックの高さの差に変換するため、多少の偏差がある可能性がある。
リース収入はユーザーが取得できる。Vite システムは割当て転送トランザクションのみを提供し、リースの価格設定と支払いはサードパーティのスマートコントラクトを通じて実現できる。

6.3 資産

ネイティブトークンの ViteToken に加えて、Vite はユーザーが自分のトークンを発行することもサポートしている。トークンの発行は、MintTransaction という特別な取引を通じて実行できる。MintTransaction のターゲットアドレスは 0 である。トランザクションのフィールドデータでは、トークンのパラメータは次のように指定される:

```
Mint: {
name: "MyToken",
totalSupply: 99999999900000000000000000,
decimals: 18,
owner:"0xa3c1f4...fa", symbol: "MYT"
}
```

ネットワークでリクエストが承認されると、MintTransaction に含まれている Vite がミントトランザクション料金として創始者アカウントから差し引かれる。システムは新しいトークンの情報を記録し、それに token_id を割り当てる。新しく生成されたトークンは、所有者アドレスに追加される。つまり、所有者アカウントはトークンの創始アカウントである。

6.4 チェーン交差プロトコル

デジタル資産のクロスチェーンバリュー転送をサポートし、「バリューアイランド」を排除するために、Vite は Vite Crosschain Transfer Protocol（VCTP）を設計した。

ターゲットチェーン上でクロスチェーン伝送が必要なすべてのアセットについて、それに対応するトークンが Vite 内のターゲットトークン循環内に必要であり、これは ToT（Token of Token）と呼ばれる。たとえば、Ethereum アカウントの ether を Vit e に転送する場合は、Vite で ETH の識別子を持つ ToT を発行できる。最初の To T の量は、ether の総量と同じになる。

各ターゲットチェーンごとに、Vite 取引とターゲットチェーン取引の間のマッピング関係を維持するための Vite 上のゲートウェイコントラクトがある。コントラクトが割り当てられた合意グループ内で、ブロックを生成することを担当するノードは VCTPRelay と呼ばれる。 VCTPRelay は同時に Vite ノードとターゲットチェーンのフルノードになり、両側でトランザクションを待機する。ターゲットチェーンでは、Vite ゲートウエイコントラクトも導入する必要がある。

リレーが機能し始める前に、対応する Vite の ToT をゲートウェイコントラクトに転送する必要がある。その後、ToT の供給はゲートウェイコントラクトによってのみ制御でき、 ToT とターゲット資産との間の 1：1 の交換比率を保証するために、誰も追加することはできない。同時に、対象チェーン上の資産は Vite ゲートウェイコントラクトによって管理され、ToT が全額引き受け準備金を確保できるように、どのユーザーもそれを使用できない。

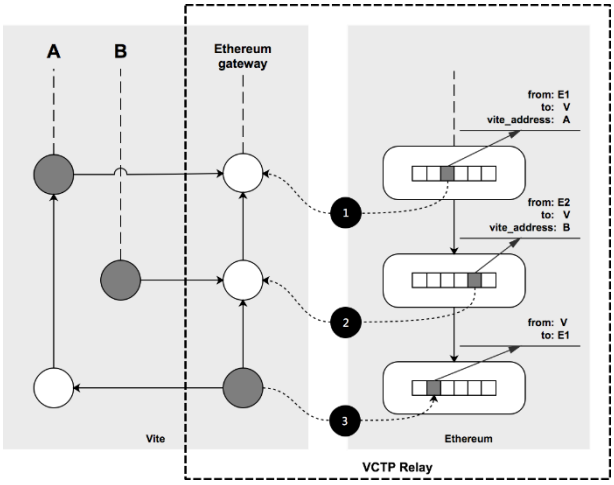

図 10:クロスチェーンプロトコル

上の図は Vite と Ethereum の間のクロスチェーン価値移転の例を示す。Ethereum ユーザー E1 がそのトークンを Ethereum から Vite に移転したい場合、そのユーザーの Vite でのアドレス A をパラメーターに置いておき、 取引を Vite のゲートウエイコントラクトのアドレス V に送ることができる。移転の残高はゲートウエイ

コントラクトアカウントで固定され、そして ToT 準備高の部分となる。取引の聴取の後、VCTPrelay ノードが対応する Vite の取引きを送り出すアカウントを生成し、ToT の同じ額を Vite のユーザーアカウント A に送る。図に置いて、１と２はそれぞれ Vite のアカウント A と B への E1 と E2 の移転を示している。もしそのユーザーが移転の際に Vite のアドレスを指定しないと、コントラクトは移転を棄却することに注意するべきである。

逆の流れを 3 に示す。ユーザー A が Vite アカウントから イーサリアムアカウントへの転送を開始すると、ある取引が Vite ゲートウェイコントラクトに送信され、一定量の ToT に転送され、その取引内の Ethereum の受信アドレス E1 が指定される。VCTPrelay ノードは、Ethereum ゲートウエイコントラクト上に応答ブロックを生成し、Ethereum の取引きを Ethereum の Vite ゲートウェイコントラクトにパッケージ化する。Ethereum では、Vite ゲートウェイコントラクトは、この取引が信頼できる VCTPrelay によって開始されたかどうかを検証し、そして、同じ額の ethel が Vite ゲートウェイコントラクトからターゲットアカウント E1 に転送される。

全てのクロスチェーンリレーノードはターゲットネットワークを監視し、各クロスチェーン取引が正しいかどうかを確認し、コンセンサスグループ内でコンセンサスに達することができる。ただし、スナップショットコンセンサスグループはターゲットチェーンのトランザクションを監視せず、２つのチェーン間のマッピングが正しいかも監視しない。ターゲットネットワークが巻き戻しまたはハード分岐された場合、Vite システム内のマッピングされた取引は同時に巻き戻しできない；同様に、Vite 内のクロスチェーン取引が巻き戻しされている場合は、対応するトランザクションターゲットネットワークは同時に巻き戻しできない。したがって、クロスチェーン取引を実行する場合は、コントラクトロジックの取引巻き戻しで対応する必要がある。同時に、4.4 に記述したように、クロスチェーンリレーコントラクトグループのための遅延パラメータを設定する必要がある。

6.5 ループするプロトコル

ループするプロトコル[1]は、非中央化資産取引ネットワークを構築するためのオープンプロトコルである。他の DEX ソリューションと比較して、ループするプロトコルは複数パーティのループマッチングに基づいている。これは先手を取る取引を防止するための二重承認技術を提供し、完全にオープンである。

ループするプロトコルを Vite に組み込む。これは、Vite 内のデジタル資産の流通を促進するのに役立ち、その結果、価値システム全体を流通させることができる。この価値システムでは、ユーザーは自分のデジタル資産を発行し、VCTP を介して資産をチェーン外で転送し、ループするプロトコルを使用して資産交換を実現できる。プロセス全体は Vite システム内で完了でき、完全に非中央化されている。

Vite では、ループするプロトコルのスマートコントラクト（LPSC）は Vite システムの一部である。資産の転送の承認と複数パーティアトミック(不可分処理）保護が Vite ですべてサポートされている。ループするリレーもオープンであり、自身のエコシステムと統合する。
ユーザースキャンは資産交換取引のための支払いに Vite を使用するので、Vite プラットフォームでループマッチングを実行するルーピングのマイナーが獲得したトークンはまだ Vite である。

7 他のデザイン

7.1 スケジュール

Ethereum では、スマート コントラクトがトランザクションで駆動し、コントラクトの実行は、ユーザーがトランザクションを開始したときのみ行われます。用途によっては、コントラクトの実行をトリガするために時計を使用してしたタイミングスケジューリング機能が必要です。
Ethereum では、この機能はサードパーティーコントラクトで実行され、1 パフォーマンスやセキュリティーは保証されていません。Vite では、内蔵されているコントラクトへのタイミングスケジュール機能を追加しています。ユーザーは、スケジュール論理を時間ごとのスケジュールコントラクトに登録できます。一般の統合グループは、スナップショットチェーンを時計として使用し、ユーザーが定義したスケジュール論理に従って、ターゲットコントラクトにトランザクション要求を送信できます。
Solidity++には、タイマーに特別のメッセージがあります。ユーザーは、コントラクトコードのタイマーをオンにすることで独自のスケジュール論理を設定できます。

7.2 ネームサービス

Ethereum では、コントラクトによってアドレスが生成され、展開した際にコントラクトを特定できます。アドレスを使用した場合、コントラクトを特定するのに問題が 2 つあります:

- アドレスとは、20 バイト以内の意味を持たない識別子です。ユーザーには使いにくく、不都合にできています。
- コントラクトとアドレスは一対です。コントラクトの書き換えはサポートされていません。

この 2 つの問題を解決するために、Ethereum 開発者は第三者契約” ENS 2 ”を用意しました。しかしながら、実際にはネームサービスの使用はとても頻繁に起こり、第三者契約の使用によって、世界で通用するユニークなネームを使用できるわけではありません。そこで、Vite ではネームサービス VNS (Vite Name Service) を構築します。ユーザーは覚えやすいネームの設定を登録し、VNS を使用して実際のアドレスに変更できます。

ネームは、vite.myname.mycontract のようなドメイン名で整理されます。システムによって、トップレベルのドメイン名は特有の目的別として保持されます。例えば、vite.xx は Vite のアドレスで、eth.xx は Ethereum のアドレスです。第 2 レベルのドメイン名は、すべてのユーザーが利用できます。ユーザーが第 2 レベルのドメイン名を保持すれば、サブドメインは任意に拡張できます。ドメイン名の所有者は、ドメイン名によって指令されたアドレスをいつでも変更でき、この機能がコントラクト更新に使用できます。

ドメイン名の長さには制限がありません。VNS では、ドメイン名のハッシュ値が保管されます。ターゲットのアドレスは、クロス チェーンリアクションへ使用できる 256 バイト以内の Vite アドレスでない場合があります。

VNS は、Ethereum のスマートコントラクトパッケージの仕様 EIP1903 とは異なることにご注意ください。VNS はネーム解決サービスで、ネームが実行時に作成されるので、解決ルールを動的に変更できます; EIP190 はパッケージ管理仕様です。名前のスペースは変化されず、コンパイル時に作成されます。

7.3 コントラクト更新

Ethereum のスマートコントラクトは不変です。一度展開すると変更はできません。コントラクトの中にバグが存在しても変更できません。このことは、開発者にはとても都合が悪く、dApp の反復作業を難しくしています。Vite は、戦略を考えスマートコントラクトの更新をサポートする必要に迫られています。

Vite では、コントラクトを更新する過程に次が含まれます:

A. 元のコントラクト状態を継承するために、新しいバージョンのコントラクトを展開。
B. VNS の新しいアドレスのコントラクトのネームを指定。
C. SELFDESTRUCT インストラクションを使用して、古いコントラクトを削除 C. Xóa hợp đồng cũ thông qua lệnh SELFDE- STRUCT

これらの 3 つの手順を、同時に完了させる必要があります。Vite プロトコルによって運営の不可分性が確実になります。開発者は、使っていないコントラクトデータを新しいバージョンのコントラクトで正確に処理できるようにする必要があります。

新しいコントラクトは古いコントラクトのアドレスを継承できません。ご注意ください。新しいアドレスに一部でも使用された場合は、トランザクションが古いコントラクトへ送信されてしまいます。異なるバージョンでのコントラクトは、動的に変更できてもできなくても、コントラクトの内容に依り、基本的に全く違うコントラクトになります。Vite システムでは、スマートコントラクトは実際に 2 つのタイプに別れています。はじめのタイプは dApp のバックグランドで、ビジネス論理が説明されています; 2 番目のタイプは実世界を地図にするコントラクトです。

はじめのタイプは、アプリのバックグランド サービスと同等で、常にアップグレードして繰り返し適用する必要があります。2 番目のタイプは契約と同等で、一度実行されれば変更できず、変更は契約違反になります。そのような契約は、変更を許可されないので、例えば、Solidity++ のスタティック キーワードを使って修飾してください :

```
pragma solidity++ ^0.1.0;
```

```
static contract Pledge {
// the contract that will never change
}
```

7.4 ブロック プルーニング

レジャーでは、どんなトランザクションも不変です。ユーザーはトランザクション履歴を変更、削除することなくレジャーに新しいトランザクションを追加できます。つまり、システムの運営によって、レジャーが拡大します。ネットワークに参加する新しいノードが最新の状態を保存する場合は、ジェネシスブロックから開始し、履歴のトランザクションすべてをやり直します。システムをしばらく実行後、会計帳簿に占領されたスペースと、トランザクションのやり直しに使用される時間は認められなくなります。Vite のトランザクション高速処理システムで、成長率は Bitcoin や Ethereum より高くなりますので、レジャーでブロックをクリップできる技術を使用する必要があります。

ブロックのクリップは、レジャーで使用できないトランザクション履歴の削除に関わり、トランザクションステートマシンの運営には影響を与えません。どちらのトランザクションが正常に削除されているかは、トランザクションが次のうちのどのシナリオを使用するかにより異なります：

- 回復。トランザクションの主な役割は、状態を回復することです。Vite では、スナップショット チェーンはアカウントの状態についてのスナップショット情報を保存するので、ノードがスナップショット ブロックからの状態を回復できます。 スナップショット ブロックにある過去のトランザクションすべてが状態回復に対応します。
- トランザクションの検証。新しいトランザクションの検証には、アカウント チェーンでの交換トランザクション履歴を検証する必要があり、レスポンス判定ならそれに相応するリクエスト トランザクションを検証する必要がるので、使用された会計レジャーで、少なくても 1 つのトランザクション履歴が各アカウント チェーンに残っている必要があります。さらに、すべての公開リクエスト トランザクションは、ハッシュ値が後のレスポンス トランザクションによって参照される場合があるので、使用できません。
- 割り当て計算。あるトランザクションが割り当てに当てはまるかどうかは、最後の 10 トランザクションでの移動平均の判断によって計算されます。つまり、各アカウントチェーンに最低でも過去 9 つのトランザクションが保存されている必要があります。
- 履歴についての質問。ノードでトランザクション履歴について質問がある場合、質問に関連しているトランザクションは使用できません。

他の使用シナリオでは、各ノードは上記のクリップ戦略からいくつかの組み合わせを選ぶことができます。スナップショット チェーンを完璧に維持する必要がある一方で、クリップはレジャーのトランザクションに関連していることに注意してください。さらに、スナップショット チェーン記録された内容は、コントラクト状態のハッシュ値になります。アカウントがクリップされた場合、スナップショットに対応する状態を完璧に維持する必要があります。

Vite データの完全性を保証するために、ネットワークで「フルノード」を維持し、すべてのトランザクション データを保存する必要があります。 スナップショット一致のグループ ノードはフルノードで、さらに交換のような主要ユーザーもフルノードになります。

8 統制

分散型アプリプラットフォームで健全なエコシステムを維持するためには、効率の良い統制システムが必要不可欠です。統制システムを構築する場合、効率と公平性を考える必要があります。

Vite の統制システムは、オンチェーンとオフチェーンの 2 つがあります。オンチェーンとはプロトコルを基にした投票方法で、オフチェーンとはプロトコル自体の繰り返しです。

投票方法は、世界投票と地域投票の 2 つがあります。世界投票は、ユーザーによる選挙の投票に基づいており、投票数に合わせて権利を計算でき、スナップショット一致のグループプロキシーノードの選挙に主に使用されます。地域投票は、コントラクトを目的にしています。コントラクトが展開された場合、選挙の基本としてトークンが指定されます。コントラクトのある場所の一致グループのノード エージェントを選ぶのに使用されます。

トランザクションの検証の他、スナップショット一致のグループ には、Vite システム不適合を更新するかどうか決定する権利があります。選出一致グループのプロキシノードには、コントラクトの規模拡大によって起こる潜在

的な危険性を避けるために、コントラクトを更新すべきかどうか決定する権利があります。ノードエージェントは、ユーザーの代わりに効率よく決断する能力を更新する目的と、選挙への参加不十分による決断の失敗を避ける目的で使用されます。またこれらのプロキシノード自体は一致プロトコルで制限されています。 ほとんどの ノードエージェントが通過したら、更新が実行されます。これらのエージェントがユーザーの期待する決断能力に達しない場合、投票によってプロキシの能力をキャンセルできます。

オフチェーン統制は、コミュニティにより実現しています。Vite コミュニティ参加者は Vite プロトコル自体やシステムに関連する提案プランを提出できます。この提唱プランは VEP (Vite 強化提案)と呼ばれています。

VEP はコミュニティの中で広く討論され、ソリューションを実施するかは、Vite 環境部によって決められます。VEP 実行のためにプロトコルを更新するかどうかが、ノードエージェントによって最終的に決定されます。もちろん、差が大きい場合はそのチェーンでの選挙を開始して、様々なユーザー オプションを集められ、プロキシノードが選挙の結果から更新をするかどうかを決定します。

Vite 参加者の一部には、オプションへ投票するトークンを十分もたない方もいますが、それでも自由に VEP を提出して、意見を述べることができます。投票の権利があるユーザーは、Vite での権利を健全に保つ必要があるので、環境部の意見を真剣に聞き入れてください。

9 未来のタスク

スナップショットチェーンのトランザクション検証はシステムの主要業績のネックです。Vite は同時性デザインと DAG アカウント構成を採用しているので、トランザクション検証は同時に行われる可能性があります。しかしながら、異なったアカウントのトランザクションでの差異で、同時進行の度合いが大部分で制限されています。トランザクション検証での同時進行過程の向上や、分散検証戦略がこれから適応していくうえで重要な方向になります。

現在の HDPoS コンセンサスアルゴリズムの中にも、欠陥が存在します。これも適応化の方向であり、コンセンサスアルゴリズムを向上させ、言い換えれば、選出コンセンサスグループのもっと多くのコンセンサスアルゴリズムと適合するようにできます。

さらに実際のマシンの適応化も、システムの遅れを減らしてシステムの処理能力を向上させるのに重要です。EVM のシンプルなデザインと命令セットの簡略化により、もっとパワフルな実際のマシンを将来デザインし、記述能力が優れ、しかもセキュリティ脆弱性が少ないスマート コントラクト プログラミング言語を定義する必要性があるのかもしれません。

最後に、Vite 主要同意に加え、環境開発をサポートする付属設備の建設も、大きなトピックです。dApp 開発者への SDK 支援に加え、dApp フォアグランド生態系構築に向けて多くの努力が必要です。例えば、Vite のモバイルウォレットアプリで、HTML5 を基に dApplet エンジンを構築でき、開発者は低価格で dApp を開発、公開できます。

10 まとめ

他の同様のプロジェクトと比較して、Vite の特徴には以下が含まれます :

- 高い処理能力。Vite は、DAG レジャー構築を使用し、直行トランザクションをレジャーと同時に記述できます。さらに、複数のコンセンサスグループが HDPoS コンセンサスアルゴリズムで存在してもお互い依存することなく、同時に 作業可能です。一番大切なのは、 Vite 内のコントラクト コミュニケーションが 非同期モデルを基盤としていることです。これらすべてがシステムの処理能力をあげるのに役立っています。
- 低遅延。Vite は、HDPoS コンセンサスアルゴリズムを使用して、共同作業でプロキシノードで反復生産ブロックを完成させます。PoW を計算する必要なく、ブロックの休憩時間を 1 秒に縮小でき、トランザクション完了の遅れをなくすのに役立っています。
- スケーラビリティ。スケーラビリティ要件を満たすために、Vite はトランザクション一回につき一回の選択を与えています。アカウントの状態に従ってトランザクションをグループ化し、違うアカウントのブロック商品によって、別のノードで完了できるようにし、メッセージに基づいて BASE セマンティクへクロスコンタクトで呼び出された ACID のセマンティクスを削除できるようにします。こうすれば、ノードがすべての世界状況を保存する必要がなくなり、シャーディング モードでデータを全分散型ネットワークに保存できます。

- 使いやすさ。Vite の使いやすさへの改善点には、Solidity++での標準ライブラリサポートが含まれており、メッセージ 構文の処理、コントラクトのスケジュールのタイミング、 VNS 名前サービス、コントラクト更新などに特化しています。
- 値計算。Vite はデジタル資産発行、クロスチェーン・バリュートランスファー、ループリング プロトコールを基盤にしたトークン交換などをサポートし、完璧な換算システムを形成しています。ユーザーの観点から見ると、Vite はフル機能を備えた分散型両替所です。
- 経済。Vite はリソース割り当てモデルを採用しているので、あまりトレードを行わないユーザーは、高い手数料やガス代を払う必要がありません。ユーザーは様々な方法を使用して、計算を変更することができます。余分な割り当ては割り当て貸出同意書により他のユーザーへトランスファーすることもでき、システムのリソース使用への効率を上げることができます。

11 ありがとうございます。

当社はこの記事の作成を案内、支援していただいたコンサルタントの皆様に、心より感謝いたします。特に、このプロジェクトに参加してくださったループリング チームとコミュニティに心より感謝を申し上げます。

参考文献

[1] **Daniel Wang, Jay Zhou, Alex Wang, and Matthew Finestone**. Loopring: A decentralized token exchange protocol. *URL https://github.com/Loopring/whitepaper/blob/master/en_whitepaper.pdf*

[2] **Satoshi Nakamoto**. Bitcoin: A peer-to-peer electronic cash system. 2008.

[3] **Gavin Wood**. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151, 2014.

[4] Ethereum: a next generation smart contract and decentralized application platform (2013). *URL http://ethereum.org/ethereum.html*, 2017.

[5] **Chris Dannen**. *Introducing Ethereum and Solidity*. Springer, 2017.

[6] Cosmos a network of distributed ledgers. *URL https://cosmos.network/whitepaper*.

[7] **Anonymous**.aelf-a multi-chain parallel computing blockchain framework. *URL https://grid.hoopox.com/aelf_whitepaper_en.pdf*, 2018.

[8] **Anton Churyumov**. Byteball: A decentralized system for storage and transfer of value. *URL https://byteball.org/Byteball.pdf*.

[9] **Serguei Popov**. The tangle. URL https://iota.org/IOTA_Whitepaper.pdf.

[10] **Colin LeMahieu**. Raiblocks: A feeless distributed cryptocurrency network. URL https://raiblocks.net/media/RaiBlocks_Whitepaper

English.pdf.

[11] **Anonymous**. Delegated proof-of-stake consensus, a robust and flexible consensus protocol. URL https://bitshares.org/technology/delegated-proof-of-stake-consensus/.

[12] **Bernardo David, Peter Gazi, Aggelos Kiayias, and Alexander Russell**. Ouroboros praos: An adaptively-secure, semi- synchronous proof-of-stake blockchain. URL https://eprint.iacr.org/2017/573.pdf, 2017.

[13] **Anonymous**. Eos.io technical white paper v2. URL *https://github.com/EOSIO/Documentation/blob/master/TechnicalWhiteP*

[14] **Dai Patrick, Neil Mahi, Jordan Earls, and Alex Norta**. Smart-contract value-transfer protocols on a distributed mobile application platform. URL https://qtum.org/uploads/files/cf6d69348ca50dd985b60425ccf282f3.pdf, 2017.

[15] **Ed Eykholt, Lucius Meredith, and Joseph Denman.** Rchain platform architecture. URL http://rchain-architecture.readthedocs.io/en/latest/.

[16] **Anonymous**. Neo white paper a distributed network for the smart economy. URL

[17] **Anonymous.** Byzantine consensus algorithm. URL https://github.com/tendermint/tendermint/wiki/Byzantine- Consensus-Algorithm.

[18] **Shapiro Marc, Nuno Preguiça, Carlos Baquero, and Marek Zawirski**. Conflict-free replicated data types. URL https://hal.inria.fr/inria-00609399v1, 2011.

[19] **Deshpande and Jayant V.** On continuity of a partial order. Proc. Amer. Math. Soc. 19 (1968), 383-386, 1968.

http://docs.neo.org/en- us/index.html.

[20] **Weisstein and Eric W. Hasse diagram**. URL http://mathworld.wolfram.com/HasseDiagram.html.

[21] **Chunming Liu**. Snapshot chain: An improvement on block-lattice. URL https://medium.com/@chunming.vite/snapshot-chain-an-improvement-on-block-lattice-561aaabd1a2b.

[22] **Anonymous**. Problems. URL https://github.com/ethereum/wiki/wiki/Problems.

[23] **Dantheman**. Dpos consensus algorithm - the missing white paper. URL https://steemit.com/dpos/@dantheman/dpos-consensus-algorithm-this-missing-white-paper.

[24] **Theo Haerder and Andreas Reuter.** Principles of transaction-oriented database recovery. ACM **Comput. Surv**., 15(4):287–317, December 1983.

[25] **Dan Pritchett**. Base: An acid alternative. Queue, 6(3):48–55, May 2008.

[26] **Jeff Hanson**. Event-driven services in soa. URL https://www.javaworld.com/article/2072262/soa/event-driven-services-in-soa.html.

[27] **Michael Sipser**. Introduction to the Theory of Computation. PWS Publishing, second edition, 2006.

付録

付録 A　EVM 命令セット

A.0.1 A.0.1　　OS:ストップと代数操作命令セット

番号	用語	ポップ	プッシュ	EVM での意味	Vite での意味
0x00	STOP	0	0	実行を停止	同じ意味
0x01	ADD	2	1	２つのオペランドを加算する	同じ意味
0x02	MUL	2	1	２つのオペランドを掛け算する	同じ意味
0x03	SUB	2	1	２つのオペランドを減算する	同じ意味
0x04	DIV	2	1	２つのオペランドを割り算する.そして 0 を返す	同じ意味
0x05	SDIV	2	1	シンボルで割る	同じ意味
0x06	MOD	2	1	剰余演算	同じ意味
0x07	SMOD	2	1	シンボルでの剰余演算	同じ意味
0x08	ADDMOD	3	1	最初の２つのオペランドを加算し、３番目で剰余を求める	同じ意味
0x09	MULMOD	3	1	最初の２つのオペランドを掛け算し３番目で剰余を求める	同じ意味
0x0a	EXP	2	1	２つのオペランドのオペランドで累乗演算をする	同じ意味
0x0b	SIGNEXTEND	2	1	シンボルの拡張	同じ意味

A.0.2 10s: 比較とビット操作の命令セット

番号	命令表記	ポップ	プッシュ	EVM での意味	Vite での意味
0x10	LT	2	1	より小さい	同じ意味
0x11	GT	2	1	より大きい	同じ意味
0x12	SLT	2	1	シンボルでより小さい	同じ意味
0x13	SGT	2	1	シンボルでより大きい	同じ意味
0x14	EQ	2	1	に等しい	同じ意味
0x15	ISZERO	1	1	0 であるか？	同じ意味
0x16	AND	2	1	ビット毎アンド	同じ意味
0x17	OR	2	1	ビット毎オア	同じ意味
0x18	XOR	2	1	ビット毎イクスクルーシブオア	同じ意味
0x19	NOT	1	1	ビット毎反転	同じ意味
0x1a	BYTE	2	1	バイトの１つを取る	同じ意味

A.0.3 20s: SHA(セキュア・ハッシュ・アルゴリズム) 命令セット

番号	用語	ポップ	ッシュ	EVM での意味	Vite での意味
0x20	SHA3	2	1	Keccak-256 ハッシュを計算	同じ意味

A.0.4 30s: : 環境情報命令セット

番号	用語	ポップ	ッシュ	EVM での意味	Vite での意味
0x30	ADDRESS	0	1	現アカウントのアドレスを得る	同じ意味
0x31	BALANCE	1	1	アカウントの残高を得る	同じ意味
0x32	ORIGIN	0	1	元の取引の送り主のアドレスを得る	違う意味 ずっと 0 を戻す Vite は内部取引とユーザー取引の間の原因関係を保持しない
0x33	CALLER	0	1	直接の発信者のアドレスを得る	同じ意味
0x34	CALLVALUE	0	1	呼ばれた取引の移転金額を得る	同じ意味
0x35	CALLDATALOAD	1	1	この呼のパラメーターを得る	同じ意味
0x36	CALLDATASIZE	0	1	この呼のパラメーター・データの大きさを得る	同じ意味
0x37	CALLDATACOPY	3	0	呼ばれたパラメーター・データをメモリーにコピーする	同じ意味
0x38	CODESIZE	0	1	現環境での実行コードの大きさを得る	同じ意味
0x39	CODECOPY	3	0	現環境での実行コードをメモリーにコピーする	同じ意味
0x3a	GASPRICE	0	1	現環境でのガス価格を得る	
0x3b	EXTCODESIZE	1	1	アカウントのコードの大きさを得る	同じ意味
0x3c	EXTCODECOPY	4	0	アカウントのコードをメモリーにコピーする	同じ意味
0x3d	RETURNDATASIZE	0	1	前の呼から戻ったデータの大きさを得る	同じ意味
0x3e	RETURNDATACOPY	3	0	前の呼から戻ったデータをメモリーにコピーする	同じ意味

A.1.5 40s: ブロック情報命令セット

番号	用語	ポップ	ッシュ	EVM での意味	Vite での意味
0x40	BLOCKHASH	1	1	ブロックのハッシュを得る	異なる意味 対応するスナップショットのハッシュを戻す
0x41	COINBASE	0	1	現ブロックの受け取りマイナーのアドレスを得る	異なる意味 ずっと 0 を戻す
0x42	TIMESTAMP	0	1	現ブロックのタイムスタンプを戻す	異なる意味 アカウントチェーンの対応する取引ブロックの数を戻す
0x43	NUMBER	0	1	現ブロックの数を戻す	異なる意味 ずっと 0 を戻す

0x44	DIFFICULTY	0	1	ブロックの困難度を戻す	異なる意味 ずっと 0 を戻す
0x45	GASLIMIT	0	1	ブロックのガス制限を戻す	異なる意味 ずっと 0 を戻す

A.1.6 50s: スタックメモリーストレージの制御の流れ命令セット

番号	用語	ポップ	ッシュ	EVM での意味	Vite での意味
0x50	POP	1	0	スタックのトップから データを１つ取り上げる	同じ意味
0x51	MLOAD	1	1	メモリーから１語ロードする.	同じ意味
0x52	MSTORE	2	0	１語メモリーに保存する	同じ意味
0x53	MSTORE8	2	0	１バイトメモリーに保存する	同じ意味
0x54	SLOAD	1	1	ストレージから１語ロードする	同じ意味
0x55	SSTORE	2	0	１語ストレージに保存する.	同じ意味
0x56	JUMP	1	0	ジャンプ命令.	同じ意味
0x57	JUMPI	2	0	条件付きジャンプ命令.	同じ意味
0x58	PC	0	1	プログラムカウンターの値を得る.	同じ意味
0x59	MSIZE	0	1	メモリーの大きさを得る.	同じ意味
0x5a	GAS	0	1	利用できるガスを得る.	異なる意味
0x5b	JUMPDEST	0	0	ジャンプ先をマークする.	同じ意味

A.0.7 60s と 70S: スタック操作命令セット

番号	用語	ポップ	ッシュ	EVM での意味	Vite での意味
0x60	PUSH1	0	1	スタックのトップに １バイトのオブジェクトをプッシュする	同じ意味
0x61	PUSH2	0	1	スタックのトップに 2 バイトのオブジェクトをプッシュする	同じ意味
0x7f	PUSH32	0	1	スタックのトップに 32 バイトのオブジェクトをプッシュする	同じ意味

番号	用語	ポップ	ッシュ	EVM での意味	Vite での意味
0x90	SWAP1	2	2	Hoán đổi đối tượng thứ nhất và thứ 2 trong ngăn xếp.	同じ意味
0x91	SWAP2	3	3	Hoán đổi đối tượng thứ nhất và thứ 3 trong ngăn xếp.	同じ意味
0x9f	SWAP16	17	17	Hoán đổi đối tượng thứ nhất và thứ 17 trong ngăn xếp.	同じ意味

A.1.8 80s: 複製操作命令セット

番号	用語	ポップ	ッシュ	EVM での意味	Vite での意味
0x80	DUP1	1	2	第 1 のオブジェクトを複写しスタックのトップにプッシュする	同じ意味
0x81	DUP2	2	3	第 2 のオブジェクトを複写しスタックのトップにプッシュする	同じ意味
0x8f	DUP16	16	17	第 16 のオブジェクトを複写しスタックのトップにプッシュする	同じ意味

A.1.9 90s: スワップ操作命令セット

STT	Từ	POP	PUSH
0xa0	LOG0	2	0
0xa1	LOG1	3	0
0xa4	LOG4	6	0

番号	用語	ポップ	ッシュ	EVM での意味	Vite での意味
0x90	SWAP1	2	2	Hoán đổi đối tượng thứ nhất và thứ 2 trong ngăn xếp.	同じ意味
0x91	SWAP2	3	3	Hoán đổi đối tượng thứ nhất và thứ 3 trong ngăn xếp.	同じ意味
0x9f	SWAP16	17	17	Hoán đổi đối tượng thứ nhất và thứ 17 trong ngăn xếp.	同じ意味

A.1.10 a0s: ログ操作命令セット

STT	Từ	POP	PUSH	番号	用語	ポップ	ッシュ	EVM での意味	Vite での意味
0xa0	LOG0	2	0						
0xa1	LOG1	3	0						
0xa4	LOG4	6	0						
0x0a					LOD1	2	0	ログ記録を拡大する ゼロスキーム	同じ意味
0xa1					LOG2	3	0	ログ記録を拡大する １スキーム	同じ意味
0xa4					LOG4	6	0	ログ記録を拡大する ４スキーム	同じ意味

A.0.11 f0s: Hướng dẫn vận hành hệ thống

番号	用語	ポップ	ッシュ	EVM での意味	Vite での意味
0xf0	CREATE	3	1	新しい契約を作成する	同じ意味
0xf1	CALL	7	1	他の契約を呼ぶ	異なる意味. アカウントにメッセージを送ることを示す 戻り値はずっと 0
0xf2	CALLCODE	7	1	他の契約のコードを呼ぶ アカウントの状態を変える	同じ意味
0xf3	RETURN	2	0	実行を停止し値を戻す	同じ意味
0xf4	DELEGATECALL	6	1	他の契約のコードを呼び、契約を変更し、現アカウントの状態を変更し元取引の情報を保つ	同じ意味
0xfa	STATICCALL	6	1	他の契約を呼ぶ 状態の変更を許さない	異なる意味 契約にメッセージを送ることを表す。 目標契約の状態を変更しない ずっとを戻す。結果が必要、 他のメッセージを目標契約を通じて送り戻る
0xfd	REVERT	2	0	実行を停止し状態を復元し、値を戻す	同じ意味
0xfe	INVALID	ø	ø	無効な命令	同じ意味
0xff	SELFDESTRUCT	1	0	事項を停止し、全ての残高の戻しを消去するのを待つように契約を設定する	同じ意味