2018 年度　卒業論文

仮想通貨ネットワークシミュレータ

担当教員　鈴木常彦
所属　中京大学
工学部
メディア工学科
鈴木常彦研究室
学籍番号　T415058
氏名　松岡 主馬

教務課提出　2019/01/17
最終修正　2019/02/06

卒 業 論 文 要 旨

題 目	仮想通貨ネットワークシミュレータ				
学籍番号	T415058	氏 名	松 岡 主 馬	指導教員	鈴 木 常 彦

OS レベル仮想化機構実装の一つである FreeBSD Jails を利用し、任意のトポロジーを形成した仮想通貨ネットワークをプログラミングできる Python ライブラリ Fika を作成した。仮想通貨の運用が始まってから約 10 年が経つが、各仮想通貨に対して多くの論文でプロトコル設計や実装上の問題が報告されている。Fika はそのような問題を解決するための研究手段や、仮想通貨を取り巻く環境の学習手段として有効に活用することができるだろう。

目次

1 序論

2008 年、サトシ・ナカモト名義の論文 [1] の発表により P2P ネットワークを基礎とした電子通貨システムが提案された。それまでのデジタル通貨は信用できる第三者機関を通して取引が行われていた。取引の度にコインは第三者機関に戻され、新しいコインが発行される。新しく発行されたコインのみが二重使用されていないものと見なされる。しかし、サトシ・ナカモトの論文で提案されたものは信用できる第三者機関を要さずに同じコインの二重使用防止を解決できる画期的なシステムであった。

2009 年に同上の論文をもとに Bitcoin が誕生した。Bitcoin が運用開始されてから約 10 年経つが、プログラムのバグ以外でシステムの停止やデータのロールバックなどを引き起こしていない。しかし、Bitcoin が極めてセキュアな仮想通貨という意味ではなく、プロトコル設計や実装上の問題により通貨に対する様々な攻撃手法や脆弱性が発見されている。仮想通貨におけるデータの真正性は多数決に依存することが多いため、攻撃手法にはかなりのコスト（コンピュータの計算パワー）を必要とするものが多い。そのため規模の大きい仮想通貨では理論上可能であっても実現の難しい攻撃がある。

また、Bitcoin とは異なる種類の仮装通貨も日々開発されており、2018 年 11 月の時点で運用されている仮想通貨は 2000 種類を超えている。それぞれの通貨は固有の脆弱性に加えて、他の通貨と共通のプロトコルを採用している場合は共通の脆弱性を抱えていることになる。仮想通貨とそれの基盤となる技術開発は黎明期の段階にすぎず、今後も様々な研究機関による発展が期待されている。仮想通貨の研究手段として次の 2 通りの方法が考えられる。

I. 運用されている通貨自体を改善させる
II. 各通貨で問題とされている点を排除した全く新しい通貨を作り上げる

各通貨の問題点は多くの論文の中で明らかになっているため、第三者の立場から研究を進めようと考えると II. を実行するだろう。しかし、Bitcoin などの他の通貨に比べて運用期間が長い通貨では、利用者や動いている金額が巨大に膨れ上がっているため即座に運用停止ができず、フォークをしながら利用者を誘導させていくことしかできない。そうした古い通貨を守るためにも I. という手段は重要となってくる。しかし、第三者がオープンソースをもとにプロトコルの改善を考案したとしても、それを容易に実装実験する環境が存在しない。そこで、擬似的に仮想通貨ネットワークを構築できるライブラリ Fika を作成した。Fika を用いることで任意のトポロジーを形成した P2P ネットワークを構築することができ、コンセンサスアルゴリズムやマイニングの難易度、スクリプト方式等の設定をカスタマイズした擬似的な仮想通貨が動いている環境を作ることができる。設定によって既存通貨の環境を再現したり、オリジナル通貨を設計した環境を構築することも可能である。

本論文は以下のように構成されている。まず第 2 章で本論文を読む上で必要とされる仮想通貨に関する技術や課題とされている内容について簡単にまとめる。第 3 章に先行事例として、今回の研究の参考とした研究を紹介する。第 4 章に Fika ライブラリの構成と設計について解説する。第 5 章に結論を示す。 Fika ライブラリのソースコードは Github にて提供している [15]。

2　仮想通貨の概要と課題

Bitcoin 以前のデジタル通貨は機能していたものの中央管理されたものであったため、政府やクラッカーたちに容易に攻撃されるものだった。Bitcoin は暗号技術や分散システムの研究の集大成であるとともに、4 つの鍵となるイノベーションを独創的かつ効果的に組み合わせることで成り立っている [2]。

- 分散化された P2P ネットワーク（2.1 を参照）
- 公開された取引台帳：ブロックチェーン（2.4 を参照）
- 数学的かつ決定論的な、分散化された通貨発行：分散マイニング（2.3 を参照）
- 分散化された取引検証システム：トランザクションスクリプト（2.6 を参照）

この章では以上の 4 つのキーポイントに加えて、仮想通貨運用に重要な役割を担っている技術、また課題となっている点について記述する。

2.1　P2P ネットワーク

P2P（Peer to Peer） という用語は、ネットワークに参加しているコンピュータ = ノード = ピアがそれぞれ同等の立場を持ち、平等で、特別なノードがなく、全てのノードがネットワークサービスを提供する負荷を分担していることを意味する。P2P ネットワーク内のノードは平等だが、いくつかの役割に分かれている。主にルーティング、ブロックチェーンデータベース、マイニング、ウォレットという機能に分かれている。これら全ての機能を持つものをフルノードという。全てのノードは P2P ネットワークに参加するためにルーティング機能を必ず持っているが、その他の機能についてはユーザーが取捨選択ができる。しかし、機能の選択が可能という点において全てのノードは平等ではなくなる場合がある。フルノードは全ての取引データ（フルブロックチェーン）を持っているため、外部を参照することなく自律的に権威を持ってトランザクションを検証することができる。一方でコインの受送信を行いたいだけならば、ルーティングに加えてウォレット機能を備えておくだけで良い。そのようなノードを軽量ノード（SVP : Simplified Payment Verification ノード）という。しかし、軽量ノードは自身のみでトランザクションの検証を行うことができず、外部ノードに委任するほか手段がない。これはフルノードと同等の立場を持つとは言えないだろう。そういった意味で、ユーザーは自ら平等ではなくなる選択をしている場合がある [2]。とくに PoW（2.5 を参照）を採用している通貨ではマイニングを行うノードとコインを所有しているノード、つまりお金の投資者とブロックチェーンの管理者が明確に区別されていることが問題となっている。

2.2　トランザクション

仮想通貨における取引記録のことをトランザクションという。トランザクションは公開されている台帳（ブロックチェーン）に記録され、誰でも見ることができる。トランザクションを作る際には、コインの所有者によって署名が行われる。正しく生成され署名されていれば、このトランザクションは有効となりネットワークを伝搬し全てのノードに届く。トランザクションの基本的な構成要素は、未使用トランザクションアウトプット（UTXO : unspent transaction output）である。UTXO は特定の所有者にロックされた分割不可能なコインの塊である [2]。

図 1 のように、トランザクションには input と output の 2 つの領域が存在する。input では、自分の所有するコインのうちどれを取引に用いるのかを示す。つまり、以前に自分宛にコインが送信されたトランザ

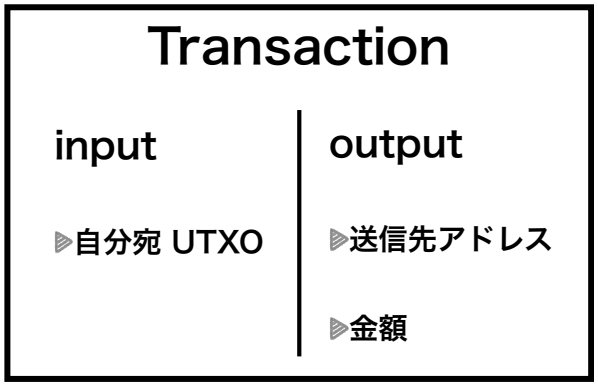

図 1　トランザクションの簡易構造

クション（UTXO）を選択する。output では、送信先アドレスと送信額を指定する。input で指定された UTXO が現在に至るまでに、input として他のトランザクションで使用されているかどうかをブロックチェーン上のトランザクションを走査検証することで二重使用を防ぐことができる。

2.3　マイニング

有効だと検証されたトランザクションデータは、各ユーザーが分散的に保存するためにブロックチェーン（2.4 を参照）というデータ構造に格納する必要がある。決められた時間が経つと、その時間間隔で発生したトランザクションたちを 1 つのブロックに格納しブロックチェーンの最後尾に連結させる。このブロックを生成する一連の作業をマイニングと言い、マイニングを行うユーザー（ノード）をマイナーと呼ぶ。それぞれのノードは 1 つ以上のノードと P2P で繋がり、ネットワークに参加している。トランザクションを受け取ったノードはトランザクションの検証を行い、有効なトランザクションと確認された場合、自分と繋がっている全ノードにブロードキャストしなければならない。この「検証」と「ブロードキャスト」を全ノードが実行することで、ネットワーク内に不正なトランザクションが存在することなく、また全ノードが全トランザクションを検知するはずである。

あるノードがブロックのマイニングに成功したと仮定する。この時にマイナーは自分宛に coinbase トランザクションをブロック内に追加することができる。coinbase トランザクションとは、マイニングに成功した報酬としてマイナーに決められた量のコインを送信することができるトランザクションのことである（ただしマイナーは宛先の変更や報酬の減額を自主的に選択できる）。このトランザクションから生まれるコインは「無」から生じる（送信元がいない）ものであり、ブロックがマイニングされるごとにシステム内に存在するコインの量が増えていく仕組みである。コイン総量は各通貨によって上限が決められているもの、決められていないものがあり、Bitcoin の場合は 2100 万枚の上限が設定されている。マイナーの報酬は 2 種類あり、1 つは上記の coinbase トランザクションによる報酬、2 つ目は各トランザクション作成者が設定したマイニング手数料の合計額である。マイニングに成功した際に得られるこの 2 つの報酬がマイニングするインセンティブになっている。つまりマイナーは、自分がブロックをマイニングする際に、マイニング手数料の高いトランザクションを優先的にブロックに格納していくことが容易に予想できる。Bitcoin のようなコイン総量の上限が設定されている通貨は、いずれ coinbase トランザクションがなくなりマイナーの報酬とインセンティブはマイニング手数料のみになってしまう。

2.4　ブロックチェーン

ブロックチェーンとはトランザクションが格納されたブロックが数珠つなぎに並べられたもので、個々のブロックは 1 つ前のブロックへのリンクを持っている。ブロックチェーンは垂直な積み重なりとして最初のブロック（genesis ブロック）が土台となり、その上にブロックが積み重なねられる形で表現されることが多い。このような表現から、最初のブロックからあるブロックまでの距離を表現するのに「高さ（height）」という用語を用い、最新のブロックを「トップ（top）」または「先端（tip）」という用語で表す（図 2）[2]。

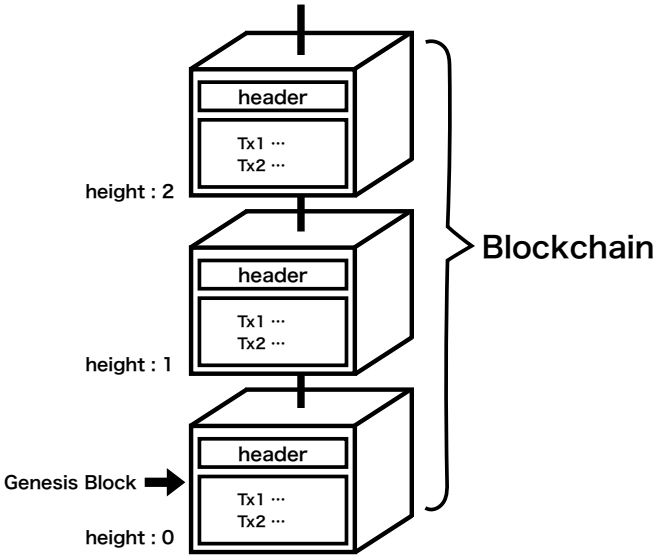

図 2　ブロックチェーン

各ブロックのヘッダーには、自身の 1 つ前のブロックのヘッダーのハッシュ値（previous block hash）を参照しており、この参照しているブロックを親ブロックという。1 つのブロックには 1 つの親ブロックしか存在しないが、1 つのブロックに複数の子ブロックが一時的に存在するケースがある。このような状態は複数のマイナーがほぼ同時にブロックのマイニングに成功した場合に発生し、一時的にブロックチェーンが分岐（フォーク）する。仮想通貨において正しいブロックチェーンとは最も長いチェーンであるというルールに基づくと、この時点ではどちらも正しいブロックチェーンとして評価される。しかし、例え発生したフォークそれぞれにちょうど等しいマシンパワーが割り振られていたとしても、何度もほぼ同時にマイニングが発生してどちらのチェーンも成長し続ける可能性はゼロに等しい。よってフォークを引き起こしたとしても、いずれは 1 つのチェーンが他のチェーンよりも長くなり、正しいチェーンが決定される。

「previous block hash」フィールドがブロックのヘッダーにあるため、現在のブロックのハッシュ値は「previous block hash」フィールドの影響を受ける。親ブロックの内容を変えるとハッシュ値が変更され、子ブロックのハッシュ値が変わってしまう。この連鎖により、あるブロックの後に多くの世代が続くと、そのブロックを変更するにはその後の世代全てのハッシュ値を再計算しなければならない。この再計算は非常に多くの計算量を必要とするため、古い世代のブロックの変更は極めて困難であり、この変更不可能性こそが安全性を支える重要な特徴となっている [2]（ただしコンセンサスアルゴリズムに PoW を採用している通貨に限る）。

ブロックチェーンというシステム内の全取引記録を含んだデータの集まりを各ノードが分散的に保持するこ

とで、ノード間でのデータの一貫性を確認することができるだけでなく、各ノードがシステム内のデータ管理者としての役割を担っているため信用できる中央管理者をおくことなく運用が成り立つのである。しかし、ブロックチェーンは通貨の運用開始から取引が続く限りコンスタントにデータ量は増大していくものであり、Bitcoin では 2018 年 12 月の時点で 200GB 以上のデータ量となっている。これらの全データ（フルブロックチェーン）をデータベースに保管されているノードのことをフルノードという。マイニングを目的としていないノードや取引をスマートフォンなどで行なっているノードたちは自身に関わる UTXO のみを保持しておけば良い。そのようなノードを軽量ノードという。データの増大に伴い、フルノードとして機能しているノード数の割合は減少しており、通貨の運用開始から時間が経つに連れて分散型の姿が崩れている。

2.5　コンセンサスアルゴリズム

中央管理者を仲介していない仮想通貨ネットワークにおいて、各ノードはどの情報が正しいのかを判断しなければならない。この仕組みをコンセンサスアルゴリズム（合意の仕組み）と呼ぶ。代表的なアルゴリズムを紹介する。

2.5.1　Proof-of-Work（PoW）

PoW とはブロックのヘッダーハッシュ値を計算し、ある目標値よりも低い値になるかどうかを確認することでブロックの有効性を確認するアルゴリズムである。コンセンサスアルゴリズムに PoW を採用している通貨として Bitcoin を例に説明する。

Bitcoin のブロックヘッダーには、親ブロックのヘッダーハッシュ値と全トランザクションのハッシュ値であるマークルルート値、タイムスタンプ、ブロック生成の難易度、nonce（ノンス）が含まれている（図 3）。

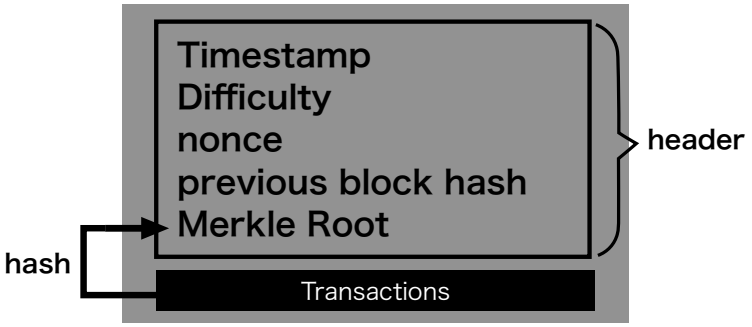

図 3　ヘッダー

$$hash(header + nonce) < target \tag{1}$$

ブロックヘッダーと nonce のハッシュ値が目標値よりも大きい場合、nonce の値を変えてハッシュ値を再計算する。ハッシュ値が目標値（target）を下回る nonce 見つける作業をとくにマイニングと呼ぶ （1）。

目標値（target）はマイニングの難易度を示しており、Bitcoin では 1 つのブロックのマイニングに約 10 分かかるように 2 週間に一度、難易度の調整が自動的に行われている。これにより、マイナーの増減によるコンピュータリソースの変化やマシンパワーの発展に対応することが可能となっている。PoW はその計算量によりブロックチェーンでのデータ改ざんに対して強い耐久力を持っているが、PoW 型通貨にはさまざまな問題

が生じている。例えば、世界中のマイナーがマイニングのためにコンピュータを稼働させていることによって、マイニングにおける大量の電力消費が発生している [3]。また、利用者の増加に伴いマイニング難易度が上昇したことで一般の個人ユーザー単位でのマイニングが困難になった。個人ユーザーたちは高性能の計算機に負けないために自身らのマシンパワーを集めて 1 つの大きなマシンパワーを持つグループ（マイニングプール）を形成して対抗した。PoW 型通貨では、マイニングプールの増加によりマイナーたちの集中化が起き、分散型とは程遠いトポロジーを形成している。

2.5.2 Proof-of-Stake（PoS）

PoS はブロックのマイナーを確率で決定するアルゴリズムである。マイナーに選ばれる確率は、ユーザーの保有するコインの量に比例する。つまり、システム内で最も多くのコインを保有しているユーザーが最もマイナーに選ばれやすい。PoW とは異なりマイニング[*1]に高性能なマシンパワーを必要としないため、電力消費に関する問題が発生しない。しかし、保有するコインの量に比例するというルールにより、PoS ではブロックをマイニングしたいユーザーはコインを使用せず、常に保管することが予想される。その結果、新規ユーザーが後から仮想通貨ネットワークに参加するインセンティブが少ない上、コインの流動性の低下により、通貨としての意味をなしていない状況になる危険性がある。

単に PoS のみに依存している愚直なアルゴリズムには上記の内容も含めて深刻な問題がある。そこで PoS を採用する多くの通貨では Delegated-Proof-of-Stake（DPoS）という PoS を発展させたものを利用している。DPoS ではブロックは前もって決められた、ユーザーの集合（Delegates：代表者たち）によってマイニングされる。まず、ブロックは一人のユーザーがマイニングする。その後ブロックの検証を行い有効であるとみなされたものに対し、その他複数の代表者たちによって署名される必要がある。このユーザーの集合は定期的に更新され、ルールは通貨によって異なる。中には、代表者の資金の一部をタイムロックされた（代表者として選ばれた間は開くことのできない）セキュリティアカウントに入れておく必要のあるバージョンも存在する。もし悪意のある行為を行なった場合、その資金は二度と使用できないものとして押収される。このようなバージョンをとくに Deposit-based-PoS と呼ばれている。Cardano 財団の ADA コインなどが採用している [4], [5]。

2.5.3 PoW / PoS ハイブリッド型

PoW と PoS どちらにも利点と欠点を備えているが、消費電力の観点から今後の仮想通貨には、PoW にとって変わるアルゴリズムが必要だとされている。しかし、完全に PoS（DPoS）型に依存してしまうと様々な攻撃の可能性を孕んでしまう[*2]。そこで両者のメリットを生かして他方のデメリットを解消するハイブリッド型アルゴリズムが誕生した。計算量の必要性を要所に持たせることで、PoS 固有の攻撃から守ることができる [4]。ハイブリッド型通貨の一例として Peercoin[7] がある。

2.6 スクリプト

Bitcoin を始め多くの仮想通貨では、各トランザクションにコインの所有を証明するスクリプト（unlocking script）とコイン所有権の譲渡を意味するスクリプト（locking script）を挿入するフィールドが存在する。

UTXO にある locking script（解錠条件）と通常は署名を含んでいる unlocking script は Script 言語で書

[*1] 実際には PoS では ミンティング（Minting：鋳貨） と呼ばれているが本論文ではマイニングと記す
[*2] 攻撃の説明は論文 [6] を参照されたい

かれている。Script 言語 [8] とは「逆ポーランド記法」の言語である。式を左から順に処理していくことでスクリプトを実行していく。オペレータは 1 つまたは、複数のデータに対してスタックにプッシュかポップ、またはデータに対して何らかの操作をする。条件オペレータは条件を評価し、True か False のブール値をプッシュする。「2 3 OP_ADD 5 OP_EQUAL」というスクリプトを実行した例を図 4 に示す。

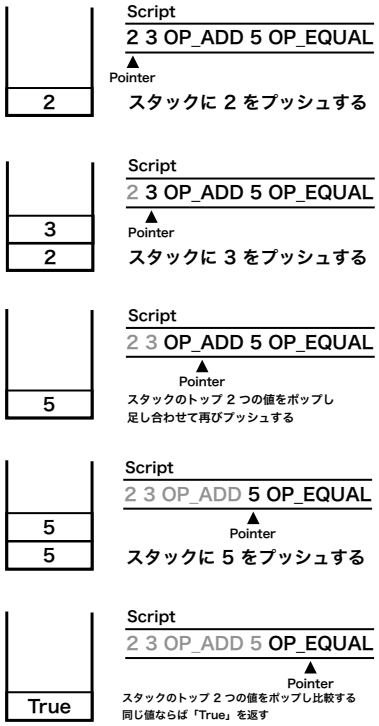

図 4　script の実行例

UTXO にある locking script はトランザクションの output に置かれてたそのコインの解錠条件で、将来その UTXO を使用する際に満たさなければならない条件を指定している。locking script には通常コインの所有者となるユーザーの公開鍵かアドレスが含まれている。unlocking script は locking script に置かれた条件を満たすためのスクリプトであり、トランザクションのアウトプットとして使用できるようにしている。unlocking script には通常コインを使用するユーザーのデジタル署名が含まれている。トランザクションの有効性チェックでは、最初に unlocking script が実行される。もし、エラーなく実行されると続けて locking script が実行される。返り値が「True」であれば、正常に解錠条件を満たしたことになり UTXO を使用できる権限があることが証明される。

Script 言語は多くのオペレータを持っているが、意図的にループや if 文などの分岐がないように仮想通貨の種類によって制限されているオペレータがある。スクリプトは複雑でなく、無限ループを作ることや DOS 攻撃を起こすような論理爆弾を作ることができなくされている。[2]

2.7　ウォレット

仮想通貨におけるウォレットにはユーザーアカウントの秘密鍵と公開鍵のペアが保存されている。ウォレットに格納された鍵は、仮想通貨のプロトコルとは完全に独立しておりブロックチェーンの参照やインターネットへの接続がなくてもウォレット内で鍵の生成と管理が行える。コインを用いて取引を行いたい場合は、予め自分のアドレス宛となっている UTXO をブロックチェーンを参照し、集積してウォレットに保存しておく必要がある。仮想通貨におけるウォレットは物理的な世界のウォレットとは異なり、ブロックチェーンを走査しユーザーが所有している全ての UTXO 掻き集めているに過ぎない。つまり、コインが入っているわけではなくコインと結び付けられているトランザクションが入っている [2]。

所有するコインの管理方法としてはいくつかの方法がある。最も効果的な方法としては、専用のソフトウェアもしくはハードウェアを用いて取引時のみネットワークに参加し、それ以外ではネットワークから切り離しておくコールドウォレットという形態をとることだ。必要以上にネットワークに晒されていると攻撃の対象となる危険性が高くなるため、ウォレットの管理は非常に重要である。通貨によってはフルブロックチェーンを備えたウォレットに加えて、自分のアカウント宛の UTXO のみを管理している軽量ウォレットもリリースされている。ただし、軽量ウォレットはフルブロックチェーンを保持していないためマイニング作業はできない。

3　先行事例

この章では、仮想通貨における運用のテスト環境として既に公開されているものと、本研究で制作したライブラリのうちノード生成とネットワーク構築の参考にした研究について紹介する。

3.1　Testnet

Testnet は Bitcoin においてテスト用に使用されている別のネットワークである。Testnet のコインは実際の Bitcoin とは全く異なり一切の金銭的価値を持たない。これにより、アプリケーションデベロッパーや Bitcoin テスター達が実際のコインを使用することや、Bitcoin のブロックチェーンを破壊してしまう心配がない [9]。Bitcoin と同様なプロトコルで動いており、ネットワークには多くのユーザーが参加しているため Bitcoin と近い環境であると言える。Ethereum でも同様な環境として Ropsten[10] や Kovan[11]、Rinkeby[12] などがある。しかし、ネットワークのトポロジーを意図的に変更することや新たなコンセンサスアルゴリズム、スクリプトを実装実験できるわけではない。一般ユーザーは Bitcoin で実際のコインを扱う前の練習目的での利用がメインになるだろう。

3.2　VITOCHA

VITOCHA は FreeBSD の OS パーティショニング機能である Jail とネットワーク仮想化機能 VIMAGE を操作し、自由に仮想ネットワークをプログラミングできる Ruby ライブラリである。jail は 1 つの FreeBSD を仮想的に複数の FreeBSD にパーティショニング（隔離）する技術である。ファイルシステム、プロセスなどを他の jail から隔離する。仮想マシン（VM）とは異なりカーネルや主要なシステムファイルは共用するため少ないリソースで多数の jail を動作させることができる。VIMAGE は jail が提供する隔離機構に加えて、ネットワークスタックやルーティングテーブルなどの隔離も行う機構である。FreeBSD の ifconfig が提供する仮想インターフェースや仮想ブリッジを VIMAGE で各 jail 内で隔離することにより、1 台のマシンの中で複数の jail を繋ぐ仮想ネットワークを構築することができる。VITOCHA ライブラリについてのより詳しい情報については以下の Web サイトまたは論文を参照されたい [13], [14]。

4　Fika の構成と設計

Fika は FreeBSD の jail を用いて仮想通貨ネットワークを構築する Python ライブラリである。ここでの「仮想通貨ネットワーク」とは仮想通貨の運用に参加しているノード（サーバとクライアント）を P2P 形態に構築したものを指す。jail の生成やセットアップは VITOCHA を参考にしている。各 jail は仮想通貨ネットワークにおけるサーバもしくはクライアントとして機能する。シミュレータを実行しネットワークを構築した場合、接続されている機器同士は epair と呼ばれる仮想 LAN ケーブルで結ばれており 両端にそれぞれ IP アドレスが割り振られている。各機器のインターフェースに設けられた IP アドレスとポート番号を用いてソケット通信を行うことでトランザクションやブロックをメッセージとして送受信している（図 5）。

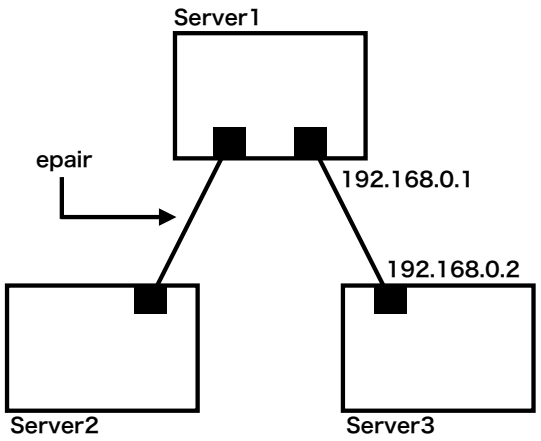

図 5　jail 接続

Fika ライブラリは以下のように大きく 2 つのプログラムに分かれている。この章ではまずライブラリの設計について触れ、サーバとクライアントの役割と全体像を解説する。また、本ライブラリの研究への適用例をいくつか示す。最後に Fika の使用例を記述する。

I. jail を用いてネットワークを構築するプログラム（4.1 を参照）
II. jail を仮想通貨ネットワーク内のノードとして機能させるプログラム（4.2, 4.3 を参照）

4.1　Fika プログラムの設計

Fika の構成は以下のようになっている。

```
/jails/bin/
├── fika.py ..... メインライブラリ
├── shcommand.py ..... jail のコマンド群を定義するクラス
├── equipment.py ..... サーバ・クライアント機器クラス
├── mkserver ..... サーバ用 jail 生成スクリプト
├── mkclient ..... クライアント用 jail 生成スクリプト
└── unjail.py ..... jail 破棄用スクリプト
```

```
└── demo.py ..... 仮想通貨ネットワーク構築サンプル
```

ネットワーク構築用プログラム（上記では demo.py にあたる）を実行することで、事前に作成しておいた jail 同士を epair で接続させセットアップすることができる。各ソースコードは Github にて掲載している [15]。

4.2 サーバ設計

サーバプログラムの構成は以下のようになっている。

```
/usr/local/server/
├── server_core.py ..... メインライブラリ
├── connection_manager.py ..... ノード間の通信を管理
├── message_manager.py ..... メッセージのビルド・解析
├── keymanager.py ..... 鍵の管理
├── rsa_utils.py ..... 各種検証
├── socket_make.py ..... ソケットの管理
├── neighbour_server_<servername>.txt ..... 接続するサーバ情報
├── neighbour_client_<servername>.txt ..... 接続するクライアント情報
├── log.txt ..... 出力テキスト
├── transaction
│   ├── transactions.py ..... トランザクションの定義
│   └── utxo_manager.py ..... UTXO の管理
└── blockchain
    ├── block.py ..... ブロックの定義
    ├── block_builder.py ..... ブロック生成
    ├── blockchain_manager.py ..... ブロックチェーンの管理
    └── transaction_pool.py ..... トランザクションプールの管理
```

サーバはバックグラウンドで実行させておくため、サブプロセス管理をするモジュール subprocess.Popen を活用している。サーバとして機能するノードの出力は全てテキストファイル（log.txt）に出力される。各ソースコードは Github にて掲載している [15]。

サーバ設計の略図を図 6 に示す。仮想通貨ネットワークにおけるサーバの役割は以下の点を想定している。

- 伝播されてきたトランザクション、ブロックの有効性検証
- 有効なトランザクション、ブロック伝搬
- ブロックマイニング
- ブロックチェーン管理

あるサーバが ネットワーク内から新しいトランザクションを知らせるメッセージを受信したとする（図 7 を参照）。(1)： Connection Manager（CM） は受信したメッセージを解析するために Message Manager（MM） にメッセージを渡す。MM はそのメッセージの目的とコンテンツを解析し CM 回答する。MM はメッセージがフォーマットに当てはまらない場合 CM に False を返しこの時点で処理を終了させる。(2)：

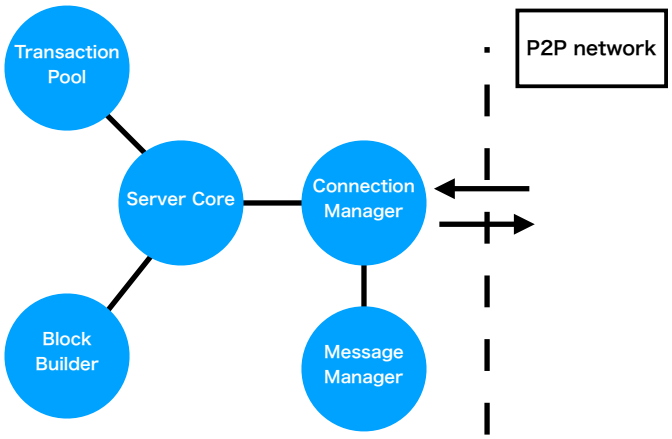

図 6　サーバ構造

False ではない返り値を得た CM は、そのまま Server Core（SC）にメッセージの目的とコンテンツを送る。(3)：SC はメッセージの目的に応じた処理を実行する。トランザクションが届いた場合はトランザクションの有効性を検証し、有効である場合は Transaction Pool（TP）に保存させる。トランザクションが有効ではない場合や、既に TP の中に同じものが存在していた場合は SC に False を返し処理を終了させる。(4)：正常にトランザクションが TP に保存された場合、送られて来た相手を除く自分と接続している全てのサーバに同じトランザクションをメッセージとして送信するため MM でメッセージを作成する。(5)：MM から受け取ったメッセージを他のサーバ群に送信する。

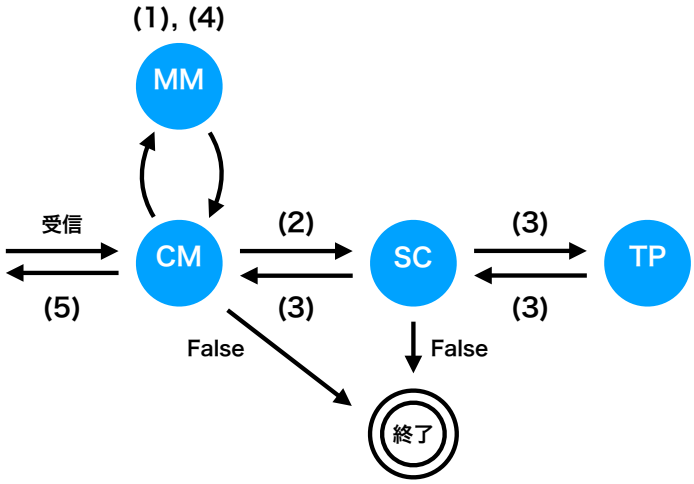

図 7　メッセージ送受信の流れ

送られてくるメッセージ内容によって SC が行う処理が異なるため、図 7 で言うと（3）の進む向きが変わってくる。

4.3 クライアント設計

クライアントプログラムの構成は以下のようになっている。

```
/usr/local/client/
├── client_core.py ..... メインライブラリ
├── connection_manager_4client.py ..... ノード間の通信を管理
├── message_manager.py ..... メッセージのビルド・解析
├── keymanager.py ..... 鍵の管理
├── socket_make.py ..... ソケットの管理
├── neighbour_server_<clientname>.txt ..... 接続するサーバ情報
├── privatekey_<clientname>.pem ..... ノードの秘密鍵
├── transaction
│   ├── transactions.py ..... トランザクションの定義
│   └── utxo_manager.py ..... UTXO の管理
└── blockchain
    ├── block.py ..... ブロックの定義
    ├── block_builder.py ..... ブロック生成
    └── blockchain_manager_4client.py ..... ブロックチェーンの管理
```

基本的にはサーバの設計と同じである。クライアントはバックグラウンドで実行せず、画面に出力される案内を元に実行したい内容に沿うコマンドを入力していけば良い。ネットワーク全体で正しく処理が反映されているかどうかは各サーバの log.txt を確認すれば良い。例えば、トランザクションを生成し接続しているサーバにメッセージを送信した場合、一度クライアントから exit しサーバにある log.txt を確認する。そこにメッセージを受け取り、検証し、他のサーバに送信したという記述を見つけることができれば正しく処理されている証拠である。再びクライアントに戻りクライアントプログラムを起動させたとき、鍵を新しく生成してしまうと今までの UTXO が自分の鍵と対応されなくなってしまうので、プログラムの初期化フェーズでディレクトリ内にノード名の.pem ファイルの有無を確認し、存在する場合はテキストファイルから鍵を復元するようになっている。各ソースコードは Github にて掲載している [15]。

クライアントは図 8 のような設計になっている。クライアントはサーバとは異なり、接続相手はサーバのみである。仮想通貨ネットワークにおけるクライアントの役割は以下の点である。

- トランザクション作成
- 鍵の生成・管理
- UTXO の管理
- ブロックチェーンの更新

今回のクライアントは仮想通貨における一般的なウォレットアプリケーションとほぼ同じ役割を担っている。クライアントは常に最新のブロックチェーンを保持しているわけではない。自分の保持しているチェーンを更新したければ、接続しているサーバに問い合わせる必要がある。ブロックチェーンをもとに自分宛の UTXO を抽出し、次の取引に使えるように UTXO リストとして管理している。トランザクション作成時において、使用する UTXO は自動で選択される。必要事項を記入された後に、Key Manager（KM）によって

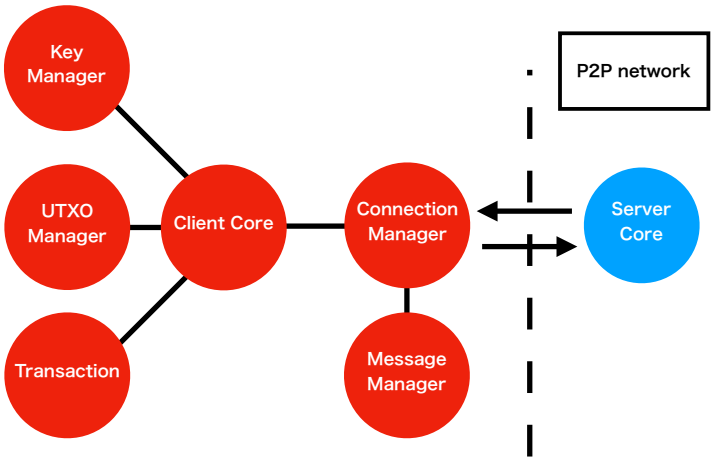

図 8　クライアント構造

署名が行われる。トランザクションが作成されたら MM に渡しトランザクションを伝播するためのメッセージを作成する。出来上がったメッセージを受け取った CM は接続している全てのノードにメッセージを送信する。

4.4　研究への適用

Fika ライブラリはブロックチェーンを用いた仮想通貨システムの実験を容易に行うことが可能である。例えば、デフォルトでは PoW を採用しているが、新たなコンセンサスアルゴリズムを実装したければ blockchain_manager.py の内容を変更すれば良い。マイニング方法やマイナーの選出方法も変える必要があるため別ファイルが必要かもしれないが、基本的動作（メッセージ管理、コネクション管理等）はそのまま再利用可能なため、一からシステムを設計する必要がない。また鍵生成のアルゴリズムを変更したければ、keymanager.py や rsa_utils.py の内容を変更すれば良い。トランザクションの内容を変更したければ transactions.py を変更すれば良い。ユーザーは自分の実験したい内容に該当する部分を書き換えた上でノードを再起動することで、自分の想定する動作を実現できたかどうかを確認することができる。

ただし Fika ライブラリでは、簡単な仮想通貨システムを実現しているものであり、実際に仮想通貨として運用する上でのセキュリティ面については触れていないので注意されたい。

4.5　Fika の利用例

最後に Fika ライブラリの利用例を示す。まずは各 jail のセットアップとネットワーク構築をする demo.py を実行する（事前に使用する名前の jail を作成しておく必要がある）。

Listing 1　セットアップ

```
root@fika:/jails/bin/fika # python3.6 demo.py
I'll do your job
Setup server server1 done!
Setup server server2 done!
Setup server server3 done!
```

```
6 Setup client client1 done!
7 Setup client client2 done!
8 epair0a is connected to server1
9 epair0b is connected to server2
10 epair0a of server1 has 192.168.0.1/255.255.255.0
11 epair0b of server2 has 192.168.0.2/255.255.255.0
12 epair0a up
13 epair0b up
14 server1 connect with {"server2": ["192.168.0.2", 50200]}
15 server2 connect with {"server1": ["192.168.0.1", 50100]}
16 epair1a is connected to server1
17 epair1b is connected to server3
18 epair1a of server1 has 192.168.1.1/255.255.255.0
19 epair1b of server3 has 192.168.1.2/255.255.255.0
20 epair1a up
21 epair1b up
22 server1 connect with {"server2": ["192.168.0.2", 50200], "server3": ["192.168.1.2",
       50301]}
23 server3 connect with {"server1": ["192.168.1.1", 50101]}
24 ...
25 server1 start!!
26 server2 start!!
27 server3 start!!
28 client1 start!!
29 client2 start!!
```

各ノード同士の epair 接続が正常に行われて、セットアップが完了すると上記の 25 行目以降のような出力が最後に確認できる。demo.py ではノードが図 9 のようなネットワーク構造を構築する。サーバは他のサーバ、もしくはクライアントと接続されている。クライアントは他のクライアントと接続していることはなく、1 つ以上のサーバ群と接続している。

次にサーバ 1 のサーバプログラムを実行する。

Listing 2　サーバ起動

```
1 root@fika:/jails/bin/fika # jexec server1
2 root@server1:/ # cd usr/local/server
3 root@server1:/usr/local/server # python3.6 server_core.py
4 Initializing server1...
5 Client node: {"client1": ["192.168.2.2", 60102]}
6 Server node: {"server2": ["192.168.0.2", 50200], "server3": ["192.168.1.2", 50301]}
7 Mydata: {"epair0a": ["192.168.0.1", "50100"], "epair1a": ["192.168.1.1", "50101"],
      "epair2a": ["192.168.2.1", "50102"]}
8 Initializing MessageManager...
9 ip : 192.168.0.1, port : 50100
10 Waiting for the connection...
11 ip : 192.168.1.1, port : 50101
12 Waiting for the connection...
```

```
13 Initializing BlockBuilder...
14 ip : 192.168.2.1, port : 50102
15 Waiting for the connection...
16 2019/01/08 22:24:45 -> 2019/01/08 22:24:48
17 Initializing BlockchainManager...
18 Initializing TransactionPool...
19 Initializing KeyManager...
20 Initializing UTXOManager...
```

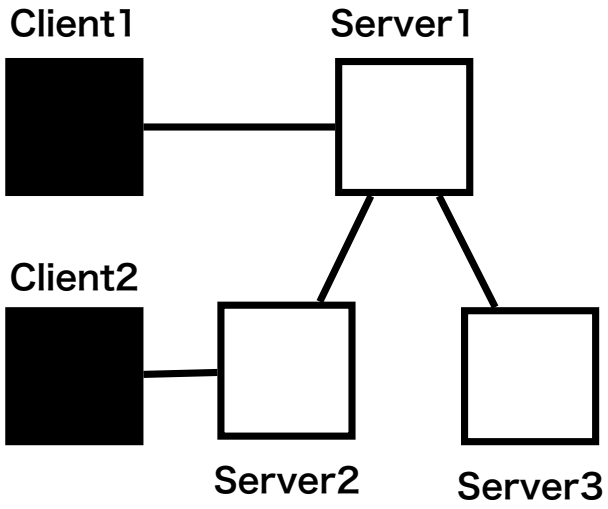

図 9　demo.py

サーバは起動時に自分と接続しているサーバとクライアントの IP アドレスとポート番号情報を持っている（5, 6 行目）。また、自分のネットワークインターフェースの IP アドレスとポート番号情報も取得している（7 行目）。セットアップを行った（demo.py を実行した）ことで各サーバはこれらをテキストデータで保持している。

残りのサーバ 2 とサーバ 3 を起動させた後、クライアント 1 を起動する。

Listing 3　クライアント起動

```
1 root@fika:/jails/bin/fika # jexec client1
2 root@client1:/ # cd usr/local/client
3 root@client1:/usr/local/client # python3.6 client_core.py
4 Initializing client1...
5 Server node: {"server1": ["192.168.2.1", 50102]}
6 Mydata: {"epair2b": ["192.168.2.2", "60102"]}
7 Initializing MessageManager...
8 ip : 192.168.2.2, port : 60102
9 Waiting for the connection...
10 Initializing BlockBuilder...
11 2019/01/08 23:15:17 -> 2019/01/08 23:15:23
```

```
12 Initializing BlockchainManager...
13 Initializing KeyManager...
14 Initializing UTXOManager...
15 extract_utxos called!
16 _set_my_utxo_txs was called
17
18 put_utxo_tx was called
19 _compute_my_balance was called
20 My Balance is... 30
21 put_utxo_tx was called
22 _compute_my_balance was called
23 My Balance is... 60
24 put_utxo_tx was called
25 _compute_my_balance was called
26 My Balance is... 90
27 _compute_my_balance was called
28 My Balance is... 90
29
30 Please enter the "command" following down
31
32 "exit": stop this client node
33 "tx": make transaction
34 "upbc": update my blockchain
35 "chain": show blockchain this node has
36 My Balance is: 90
37 >>>:
```

サーバと同様に、クライアントも接続相手と自分のネットワークインターフェース情報を持っていることがわかる（5, 6 行目）。クライアントは起動時にコインを持っていないと取引ができないので、デフォルトでは 90 コインが与えられている（28 行目）。全ての初期化が終わるとメニューが表示される（30 − 36 行目）。”tx”コマンドを入力し、クライアント 2 にコインを送るトランザクションを作成する。

Listing 4　トランザクション作成

```
1 Please enter the "command" following down
2
3 "exit": stop this client node
4 "tx": make transaction
5 "upbc": update my blockchain
6 "chain": show blockchain this node has
7 My Balance is: 90
8 >>>: tx
9 Please enter the recipient
10 If you finish this mode, enter command "exit"
11 dict_keys(["client1", "client2"])
12 >>>client2
13 How many coins do you want to send?
```

```
>>>40
Please configure the Tx's fee
>>>1
Build a message
{
    "protocol": "fika_coin",
    "version": "0.1.0",
    "sender": "client1",
    "msg_type": "NEW_TRANSACTION",
    "payload": "{\"inputs\": ...トランザクション内容は割愛する

}
_compute_my_balance was called
My Balance is... 49

Please enter the "command" following down
...
```

各クライアントはネットワークに参加している全クライアントのアドレスを知っている（11 行目）。送信先を選択した後、送信するコイン額とトランザクションを送信する手数料を設定する。その後トランザクションが作成され、それをメッセージにしてサーバに送る処理を行う。27 行目を見てわかるように自分のコインが 90 から送信に使用したコインと手数料が引かれていることが確認できる。

最後にサーバ 1 の log.txt を確認する。

Listing 5　サーバ 1 の log.txt

```
root@server1:/ # less usr/local/server/log.txt
Initializing server1...
Client node: {"client1": ["192.168.2.2", 60102]}
...
Connected by...("192.168.2.2", 35063)
Received a Message
{
    "protocol": "fika_coin",
    "version": "0.1.0",
    "sender": "client1",
    "msg_type": "NEW_TRANSACTION",
    "payload": "{\"inputs\":...トランザクション内容は割愛する

}
```

5 行目を見ると、クライアント 1 からソケットを通じてメッセージが送られてきていることがわかる。

上記の demo.py の実行において Fika はメモリを約 63MB 消費していた。サーバはバックグラウンドで常に稼働しているが、クライアントは 1 台ずつしか稼働していないため同時に 4 台のマシンが稼働しているとすると、1 台あたり約 16MB のメモリが使われていた。よって Fika ライブラリはメモリ 8 GB 程度のミドルクラスマシン上においても多くのサーバを動かせるはずだ。ただし、ブロックチェーンは作業時間とともにデー

タ量が増加していくため、それに伴って各サーバの消費メモリも増加していくことに注意されたい。また、任意のトポロジーを形成するためのセットアッププログラムの作成については demo.py を参考にすることで簡単に実行できるだろう。仮想通貨のシステム（コンセンサスアルゴリズムやマイニング方法等）を変更する場合はサーバ、クライアントそれぞれのプログラムの該当箇所を変更する必要があるため、内容によって必要となる工数が変わる。Fika ライブラリを使用せずに同じ実験環境を実ネットワーク上に再現した場合、サーバ、クライアントとしての機能だけでなくルーティング機能やセキュリティ機能等の管理作業も加わるため相当な工数を要することが予想できる。

5　結論

仮想通貨について Web や書籍等で様々な情報を得ることが可能であるが、それついての主な仕組みは理解できてもそれらを組み合わせて実験を行う環境がないため知識としての理解しか得られなかった。Fika は簡単な仮想通貨システムを実現しているのものであり、どの通貨のライブラリにも依存していないシステムである。そのため、既存の通貨を模して実験やシステムの改造を行いたければ、その通貨のソースコードを照らし合わせて必要な処理を Fika の Python プログラムに書き加えれば良い。

本研究で作成した Fika プログラムにはスクリプト概念を取り入れることができていない。そのためスクリプトを必要とした実験ができない状態である。また、プログラム中に発生した出力は PC メモリに保存しているため、多くのノードを動かしたい仮想通貨での実験環境では効率的ではない。とくにデータ量が常に増加していくブロックチェーンについては、用意されたデータベースに格納していくことが好ましい。また、クライアントがトランザクションを生成するには実際にユーザーが送り先と金額を指定する必要がある。そのため、多くのトランザクションが飛び交う環境をユーザーが手作業で再現し実験を行うことは負担となり困難である。そのためにはクライアントが自動でトランザクションを生成するプログラムを組み込んでおき、サーバだけでなくクライアントもバックグラウンドで実行させておく。そうすればユーザーはそれらから出力されたログを観察すれば良いだろう。これらの点について改善が必要であると考える。

参考文献

[1] Satoshi Nakamoto, *Bitcoin: A Peer-to-Peer Electronic Cash System*, 2008, https://www.bitcoin.org/bitcoin.pdf

[2] Andress M. Antonopoulos, ビットコインとブロックチェーン　暗号通貨を支える技術, 翻訳：今井崇也, 鳩貝淳一郎,

[3] *Bitcoin mining now consuming more electricity than 159 countries including Ireland & most countries in Africa*, https://powercompare.co.uk/bitcoin

[4] BitFury Group, *Proof of Stake versus Proof of Work*, 2015, https://bitfury.com/content/downloads/pos-vs- pow-1.0.2.pdf

[5] Aggelos Kiayias, Alexander Russell, Bernardo David, Roman Oliynykov, *Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol*, 2017, https://iohk.io/research/papers/#ouroboros-a-provably-secure-proof-of-stake-blockchain-protocol

[6] Iddo Bentov, Ariel Gabizon, Alex Mizrahi, *Cryptocurrencies without Proof of Work*, 2014, https://fc16.ifca.ai/bitcoin/papers/BGM16.pdf

[7] http://wiki.peercointalk.org/index.php

[8] https://en.bitcoin.it/wiki/Script

[9] https://en.bitcoin.it/wiki/Testnet

[10] https://ropsten.etherscan.io

[11] https://kovan.etherscan.io

[12] https://rinkeby.etherscan.io

[13] 鈴木常彦, 仮想ネットワーク構築ライブラリ *VITOCHA* とネットワーク技術者教育, 2018, http://sim.internot.jp/vitocha/ce145/paper.html

[14] http://sim.internot.jp/vitocha

[15] https://github.com/tenkanooodorobou-ishikawagoemon/Fika

謝辞

本研究を進めるにあたり、指導教官の鈴木常彦教授からは多大な助言を賜りました。厚く感謝を申し上げます。