PPL 2018

[C4] チュートリアル: ブロックチェーンの計算モデル

Shin Saito, Sachiko Yoshihama
IBM Research - Tokyo

背景

- 近年ブロックチェーン技術に代表される分散台帳技術が注目を浴びている
  - 仮想通貨のインフラとして
  - 業務アプリケーションのプラットフォームとして
    - 金融・保険・貿易・健康管理・トレーサビリティ・食の安全

- 法整備も進んでいる
  - 改正資金決済法 (2016):「仮想通貨」「仮想通貨交換業者」の定義
  - 改正犯罪収益移転防止法 (2017): 「特定事業者」として「仮想通貨交換業者」を追加

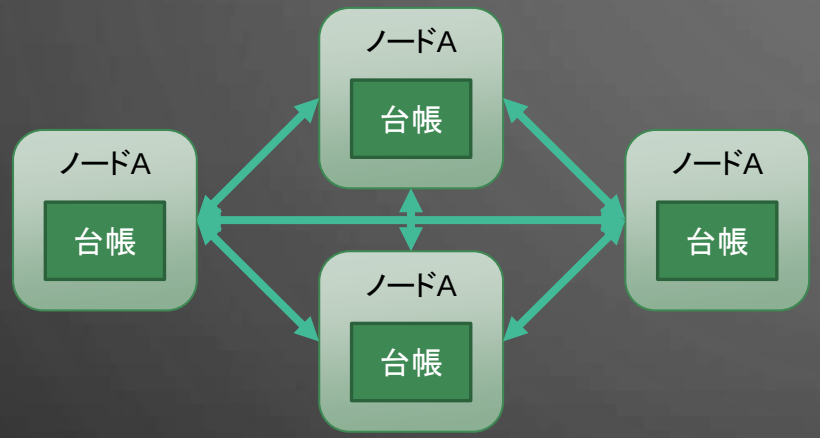
参考: 仮想通貨の定義

- 資金決済に関する法律 (平成二十八年六月三日公布) 改正
- 第二条第5項 この法律において「仮想通貨」とは、次に掲げるものをいう。
  - 一　物品を購入し、若しくは借り受け、又は役務の提供を受ける場合に、これらの代価の弁済のために不特定の者に対して使用することができ、かつ、不特定の者を相手方として購入及び売却を行うことができる財産的価値（電子機器その他の物に電子的方法により記録されているものに限り、本邦通貨及び外国通貨並びに通貨建資産を除く。次号において同じ。）であって、電子情報処理組織を用いて移転することができるもの
  - 二　不特定の者を相手方として前号に掲げるものと相互に交換を行うことができる財産的価値であって、電子情報処理組織を用いて移転することができるもの

※通貨建資産: 電子マネーなどを指すとみられる

分散台帳技術 (Distributed Ledger Technology)

- 複数の対等なノードがそれぞれ台帳を保持・同期



- 台帳 = トランザクション (TX) 列

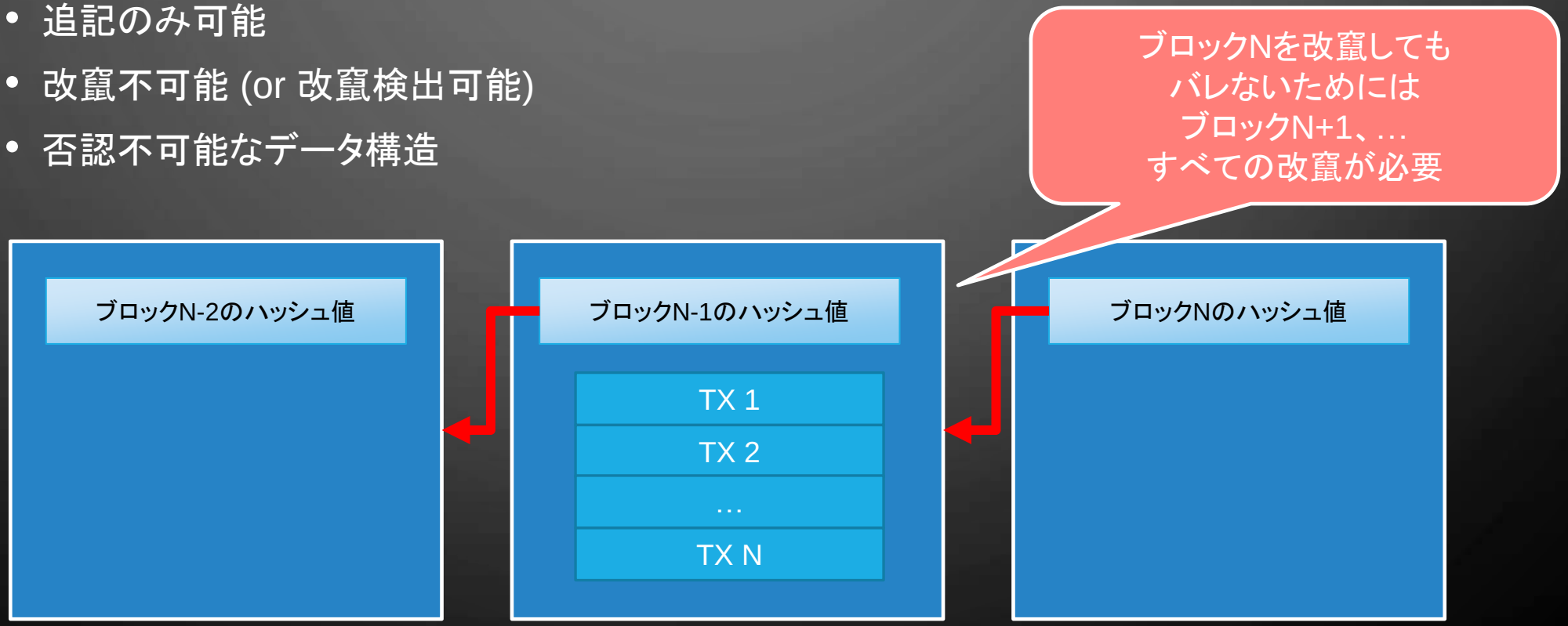
TXs
Alice → Bob: 10BTC
Bob → Charlie: 5BTC
…

- スマートコントラクト
  - TXをacceptするか否か
  - TXに伴うビジネスロジックを実行
- DLTの特徴

得意	不得意
非集中性	秘匿性
耐障害性	スケーラビリティ
耐監査性	
インテグリティ	
否認防止	

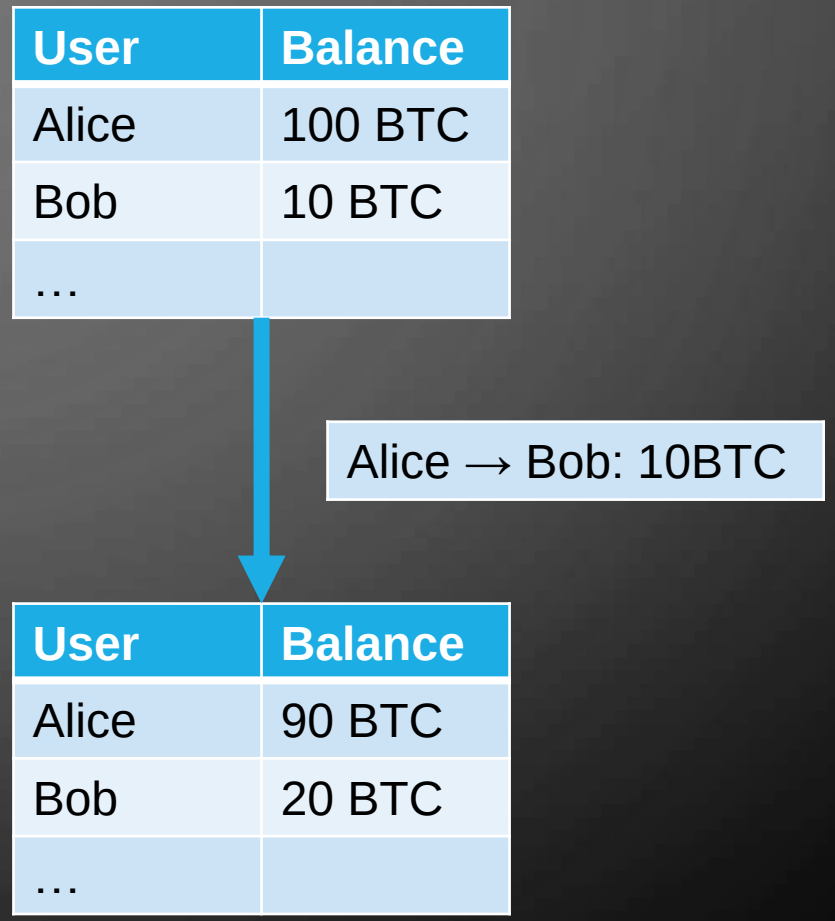
ブロックチェーン (以下、チェーン)

- 台帳のデータ構造の1つ
- 台帳をブロックのチェーンとして構成
  - 追記のみ可能
  - 改竄不可能 (or 改竄検出可能)
  - 否認不可能なデータ構造
- 分散合意 (非集中型コンセンサス)
  - ノードが対等に同期を取る



スマートコントラクト

- スマートコントラクト [Szabo '94]
  - TXをacceptするか否か
    - Aliceの残高は10BTC以上?
  - TXに伴うビジネスロジックを実行
    - Aliceの残高 -= 10BTC
      BoBの残高 += 10BTC
  - チューリング完全/不完全
- 法律の契約 (legal contract) とは直接の関係はない



DLTのバグを突いた攻撃または不具合例

- Bitcoin: Script malleabilityによるDoS攻撃とウォレットクライアントへの攻撃
- Ethereum: The DAO事件 (2016)
  - スマートコントラクトの不具合を突いた65億円仮想通貨流出事件 (後述)
- Ethereum: コンセンサスバグ (2016)
- Parity (Ethereumクライアント) のマルチシグウォッレットバグ (2017)
  - ※マルチシグ＝複数人の署名がないと通貨を使用できない仕組み
  - 34億円の被害
  - ETHが凍結されてしまう
- Parityのバグによりコントラクトを破壊 (2017)
  - 316億円分のETHがロックされる

→ 分散計算モデルとしての検証が不十分では?

Ethereumに多いのはチューリング完全な
スマートコントラクトであったりと、
表現力が強いためか

参考: 仮想通貨流出事件の例

- Mt. Gox (2011): 2609BTC
  - 秘密鍵のないアドレスに誤送信
- BitFloor (2012): 24K BTC (155億円)
  - セキュリティ対策の不備
- Mt. Gox (2014): +750K BTC (470億円) +28億円
- Poloniex (2014): 97BTC (6000万円)
  - 出金コードの脆弱性
- Bitstamp (2015): 19K BTC (12億円)
- Bitfinex (2016): 120K BTC (777億円)
  - マルチシグアーキテクチャの脆弱性?
- NiceHash (2017): 4.7K BTC (60億円)
- Coincheck (2018): 500M XEM (580億円) ← New!
  - セキュリティ対策の不備
- Reference: https://www.coindatabase.net/column/top-5-bitcoin-hacks/

なぜ検証が不十分か

- 分散システムゆえの複雑さ
- スマートコントラクトの表現力ゆえの複雑さ
- どんどん新しい実装が出る

本発表の目的

DLT実装およびその上のスマートコントラクトの解析の参考となる

- 主要なDLT実装の紹介
- モデル化ならびに解析手法の例

を紹介

主要なDLT実装

システムの前提

- 悪意のあるノードの存在 (ビザンチン障害) を仮定する
  - 一部のPermissioned型システムはクラッシュ障害のみを仮定する
- ノードはネットワーク上に分散している
  - 同期には時間がかかる
  - ノード間通信は非同期的である
- ネットワークは信頼できない
  - 送信したメッセージは重複したり、変形したり、順序が変わったり、消えたりする
  - ネットワークが分断することもあり得る

DLTアーキテクチャの大分類

	Public型	Permissioned (Consortium) 型
参加ノード・ユーザ	誰でも	許可制
合意に参加するノード数	不定	たいてい固定
合意アルゴリズム	インセンティブ設計系 (Proof of Workなど)	ビザンチン合意系
ファイナリティ	なし	あり
アプリケーション	仮想通貨など	業務アプリケーション
重視される性質	非集中性・障害耐性	速度・プライバシー

DLTの実装タイプ

- 各カテゴリから主要なものを紹介

スマートコントラクトの表現力	台帳の共有範囲	Public型	Permissioned型
非チューリング完全	台帳全共有	Bitcoin	-
チューリング完全	台帳全共有	Ethereum	Hyperledger Fabric v1
	台帳一部共有	-	Corda

DLTの実装例 (1): Bitcoin

- [Nakamoto 09] をベースにしたブロックチェーン実装
- 各種のPublic型DLT実装のもとになっている

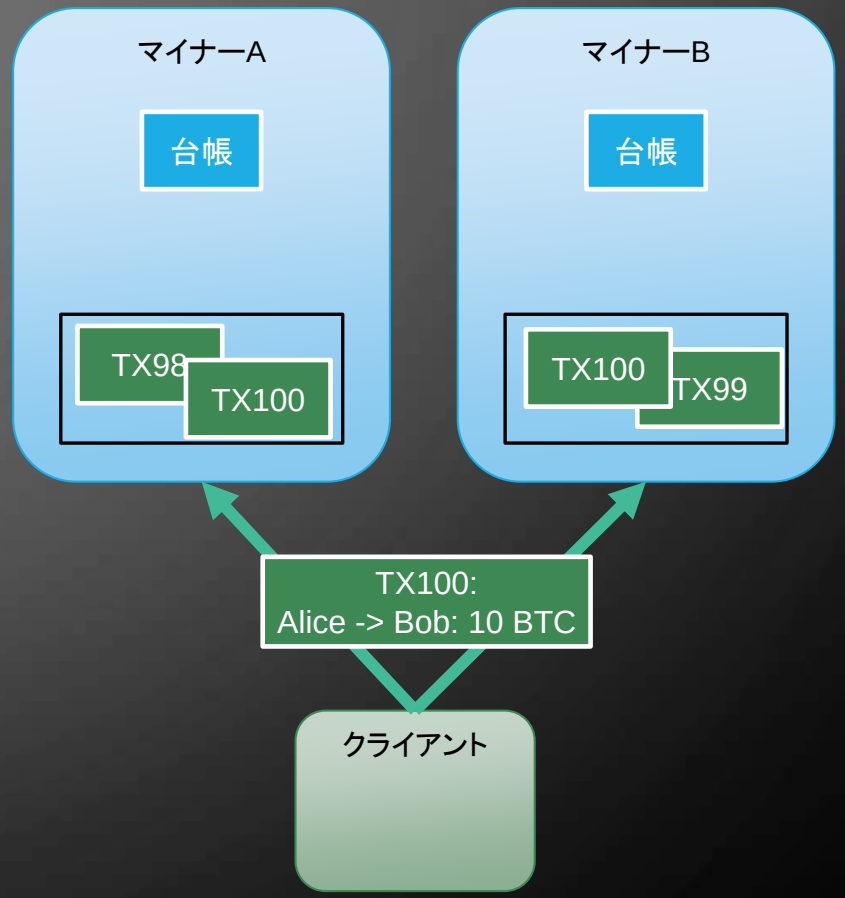
DLTタイプ	Public型
チェーン構造	線形
資産	コイン (BTC/XBT)
TX検証	UTXO + Script
台帳タイプ	UTXOベース
台帳共有	全共有
コンセンサス	Proof of Work
ファイナリティ	なし
スマートコントラクト	Script 非チューリング完全

登場人物

- マイナー (Miner)
  - 報酬を目当てにマイニングを行い、ブロックを作成するノード
- フルノード (Full node)
  - チェーンの正当性検証のみを行う良いノード
    - メリットがないので減少傾向にあるとか
- クライアント (Client)
  - 手数料を支払い、TXを送信してそれがチェーンに加えられるのを待つ

プロトコル (1): TXの送信

- クライアントがTXを送信
  - TX内容はコインの移転のみ
- TXはP2Pネットワークにより各マイナーの mempool (memory pool) に入る

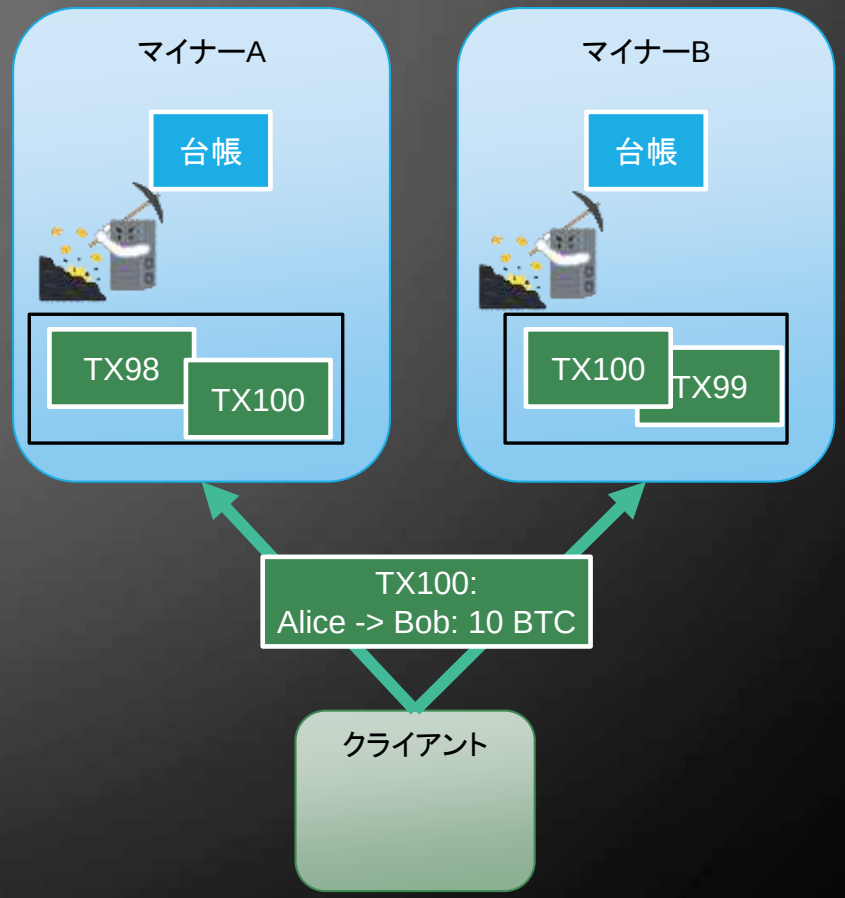


※現在のmempoolサイズは約27M TXs: https://blockchain.info/ja/charts

プロトコル (2): マイニング競争

- 各マイナーはmempoolにあるTXsからブロックを作るべくマイニング競争を行う

- なぜか?
  - マイニングに成功してチェーンにつなげられると、報酬が (BTCで) 入るから (後述)
  - ブロックを捏造して広げるより真面目にマイニングしたほうが割にあう (と考えられる) から

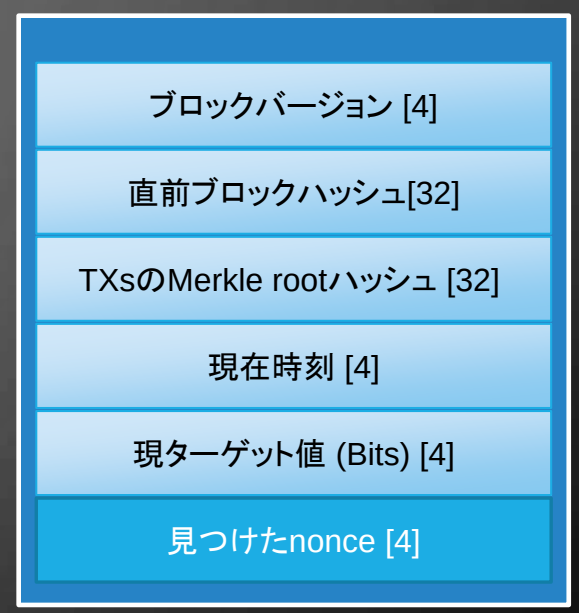


Images from いらすとや

マイニングとは (Proof of Work)

- ブロックヘッダのdhash値が「ターゲット」値以下になるようなnonceを見つける作業
  - double hash: dhash x := sha256 (sha256 x)

- ブロックを作成できたということが「作業の証拠 = Proof of Work」になる
  - Hashcashとして電子メール送信などでも提案されていた



such that
dhash(ブロックヘッダ) <= ターゲット値

マイニングとは (Proof of Work)

- 最近のブロックの例 (#511924)
  - https://blockexplorer.com/block/00000000000000004f7e7e35b090f5fc858f2769867a62376fe3cc01916ee9

Bits値をデコードしたのがターゲット

ターゲットから計算

Bits (hex)	175d97dc
困難度 (difficulty)	3,007,383,866,429.73 (約3.0T)
ターゲット (hex)	00000000000000000dc975d00000000000000000000000000000000000000000
ブロックヘッダ ハッシュ (hex)	00000000000000004f7e7e35b090f5fc858f2769867a62376fe3cc01916ee9
Nonce (hex)	8dfc9f01

たしかに
ハッシュ値 < ターゲット値

※困難度: 条件を満たすnonceを見つけるために必要なハッシュ計算回数の目安

参考: 難易度調整

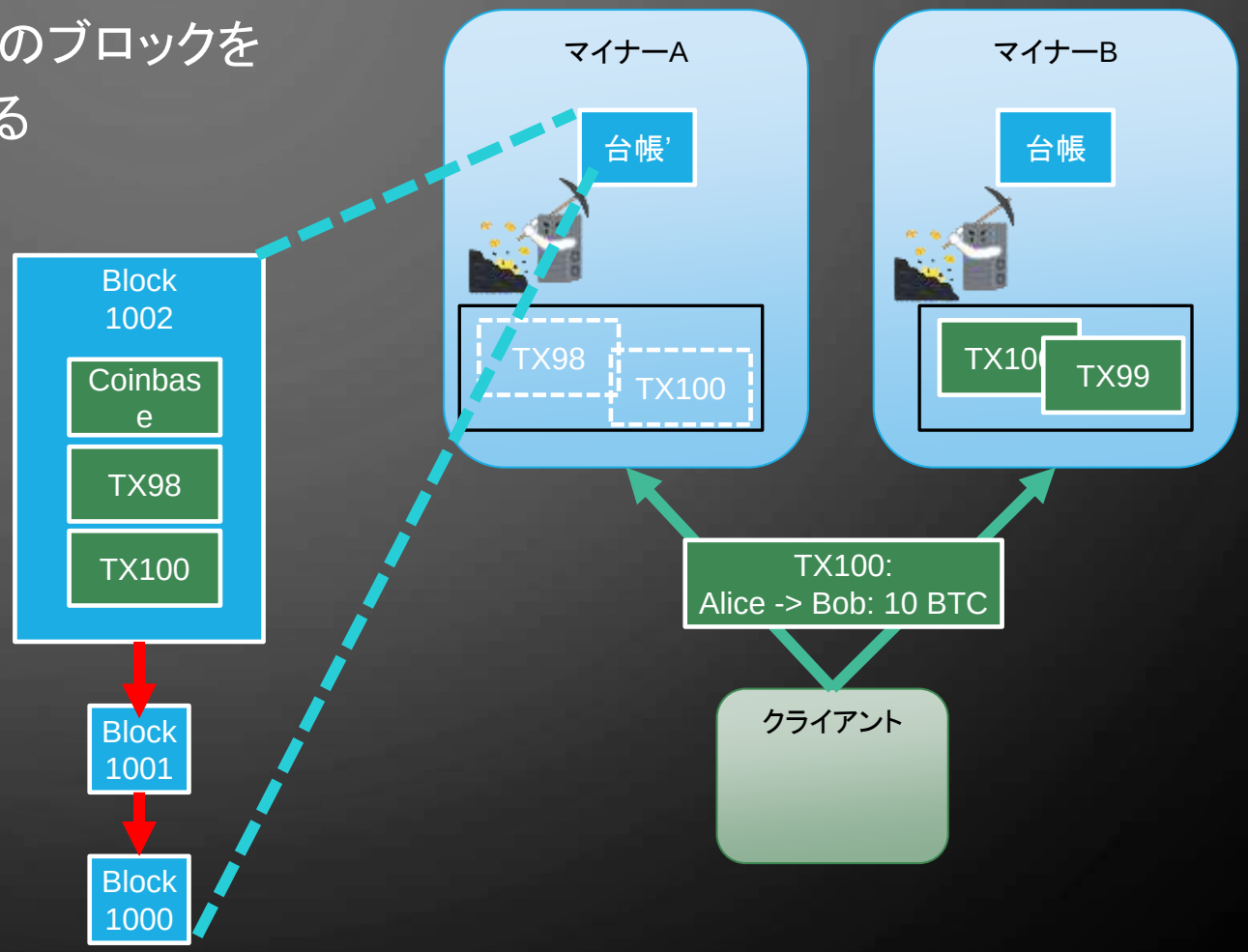
- 困難度は2016ブロック生成される毎に調整される
- 約10分でブロックが生成されるように調整される
- つまり約2週間毎に難易度調整が発生
  - 2016 x 10min = 20160min = 336hrs = 14days

参考: 通貨単位

- ISO 4217: 3文字の通貨コードを規定
  - 始めの2文字: ISO 3166-1に規定の国名コード
  - 3文字目: イニシャル
- 法定通貨の例
  - 日本円: JPY
  - 米ドル: USD
- 例外: 特定の国に関連付けられていない場合
  - 1文字目: X
  - 2-3文字目: 短縮名
- ISO4217準拠の例
  - リップル: XRP
  - ネム: XEM
- 非準拠の例
  - ビットコイン: BTC (XBTの呼称もある)

プロトコル (3): ブロック追加

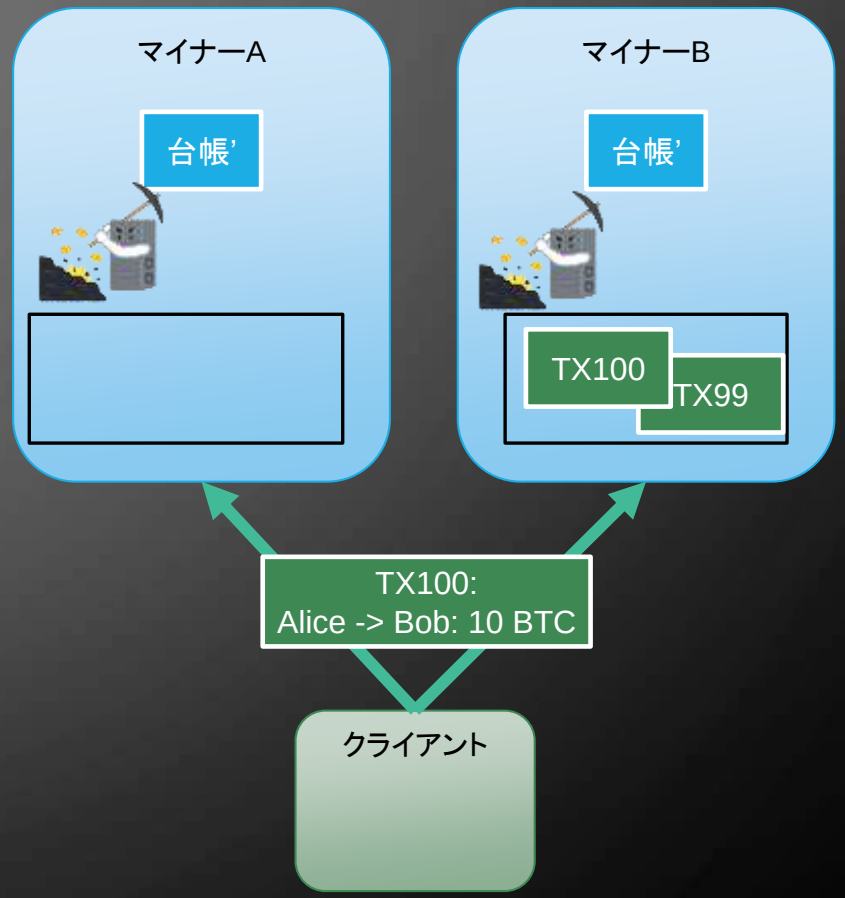
- Nonceが見つかったらマイナーはそのブロックをチェーンに追加、ブロードキャストする



Images from いらすとや

プロトコル (4): ブロック検証

- ブロックを受け取ったマイナーまたはフルノードはブロックならびに含まれるTXsを検証する
  - フォーマット
  - 署名 (Scriptによる。後述)
  - 金額の整合性 (UTXO)
- 検証が済んだら自身のチェーンにブロックを追加する

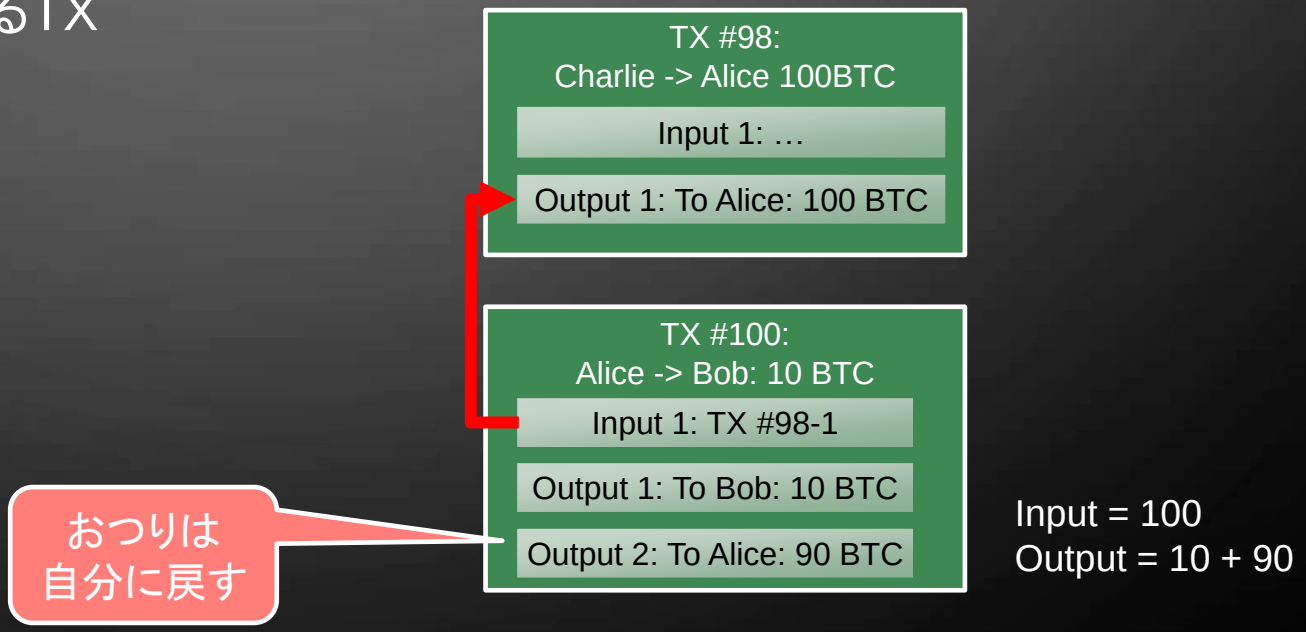


Images from いらすとや

TXの検証: UTXO (Unspent TX Output)

- TXはinput (TXs) とoutputからなる
  - Input = 誰からもらったコインを使うか
  - Output = 誰にコインをあげるか
- TX outputはspentかunspentである
- あるTXのinputで指定されているTX outputはspentであるとよぶ

- 制約
  - InputにはunspentなTX output (UTXO) しか使えない
  - Outputの金額の合計はInputの金額の合計に等しい

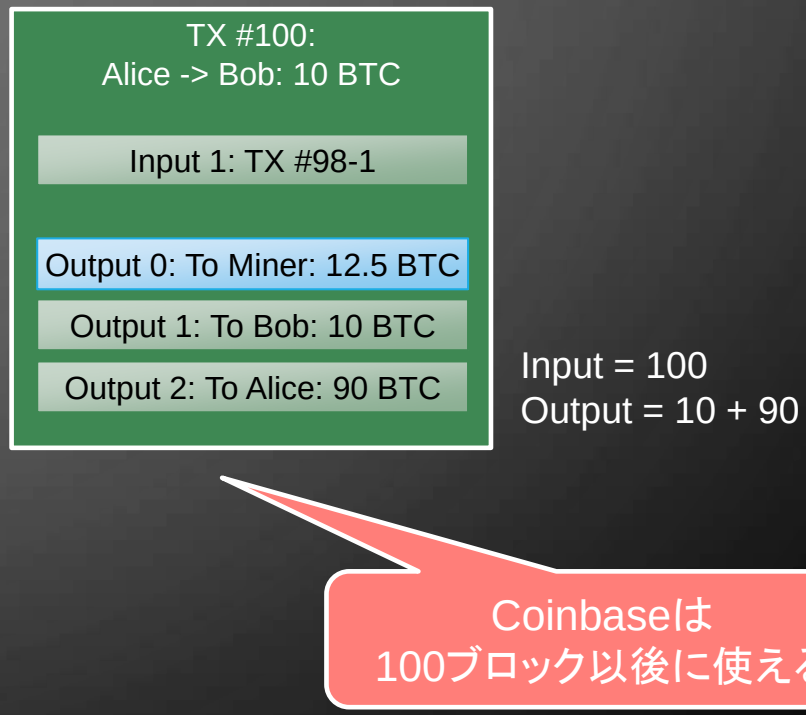


TXの検証: UTXO (Unspent TX Output)

- 正当なTXはコインの所有先を変えるだけ
- ではコインはどこから生まれるのか?

- 答: 新規ブロックが生成されるとき、そのブロックに採掘報酬 (coinbase) が含まれている
  - そのコインはブロックを生成したマイナーに帰属する
  - これにより系全体のコインが増加

正しくはこうなっている



参考: 採掘報酬 (Coinbase)

- ブロックを作成したマイナーには報酬が支払われる。このトランザクションをcoinbaseとよぶ
  - 1ブロック50 BTCから開始
  - 210,000ブロックごとに半減 (現在12.5 BTC)
  - 0になったあとはクライアントの払う手数料 (TX fee) のみがマイニングの動機になる

- BTCの発行総額
  - 6,929,999番目のブロックが最後
  - 約2100万BTCが発行される
  - 約132年かかる見込み

$$
\begin{aligned}
&210000 \times 50 \times \frac{1-(1/2)^{6930000/210000}}{1-1/2} \\
=&210000 \times 50 \times \frac{1-(1/2)^{33}}{1-1/2} \\
=&21000000 \times \left(1-(1/2)^{33}\right)
\end{aligned}
$$

UTXOベースとアカウントベース

- Bitcoinでは「現在の残高」を保持していない。この方式をUTXOベースとよぶ
  - UTXOからわかるから
  - ブロックを作成する際に「最新の状態」を計算する必要はない

- 一方、一部のPublic型や多くのPermissioned型の実装では、系の最新の状態 (= world state) をデータベースに保存している。これをアカウントベースとよぶ
  - ブロックを作成する際に「最新の状態」を計算する必要がある

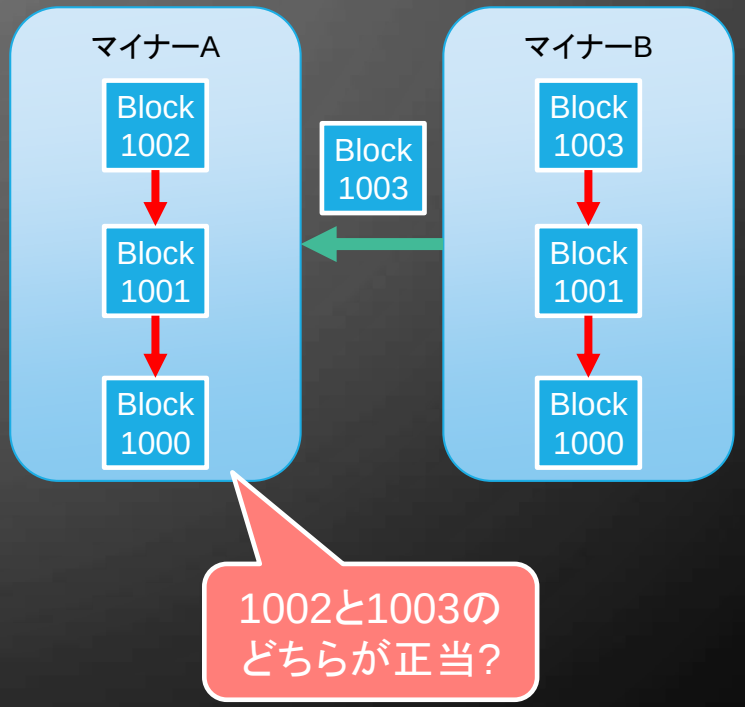
TXの検証: Script

- TX outputを費消 (spend) するには、秘密鍵による署名が必要である
- 必要な署名がそろっていることをScriptの実行で検証する

- ScriptはTX outputおよび対応するTX inputに格納されている
- これらを連結して実行した結果がtrueになる場合のみTXは正当であるとされる
- Script
  - Forth言語に似たスタックベース言語
  - 非チューリング完全
- これにより単一の署名もしくは複数の署名 (マルチシグ) を要求できる

チェーンのフォークとコンセンサス (PoW)

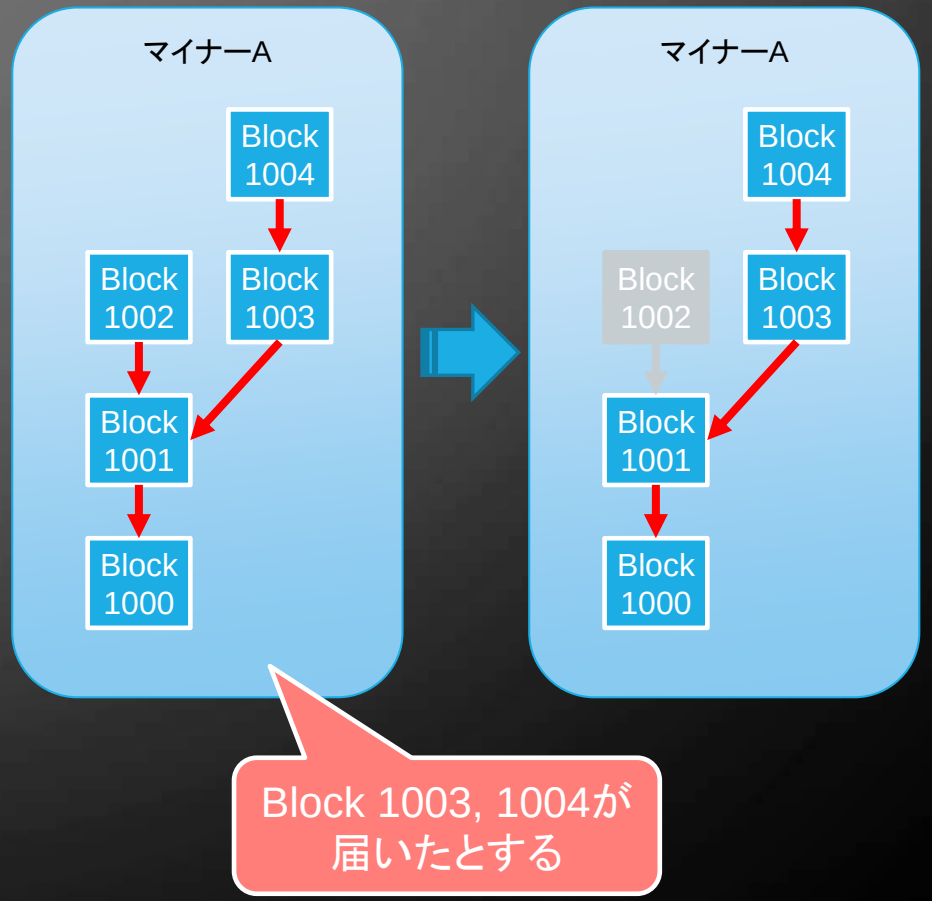
- マイニングとブロックの生成は各マイナーが独立に行うため、チェーンが分岐することがある。これをフォーク (fork) とよぶ
- Forkの解消をBitcoinではコンセンサスとよぶことが多い
- コンセンサスにはProof of Workが用いられる



チェーンのフォークとコンセンサス (PoW)

- Proof of Workでの正しい振る舞い:
- 困難度の和が最大のbranchをmainとみなし、そのbranchにブロックをつなげていく
- Main branchからはずれたブロック(とTXs) は、無かったことになる

要は長い方



- マイニング競争とはmain branchを取る競争
- ファイナリティがないコンセンサス
  - いまmain branchにいるブロックでも、いつかmainから外れる可能性
  - 51%問題: 51%のハッシュパワーで乗っ取り可能

参考: プロトコル変更に伴うフォーク

- ソフトフォーク: 後方互換性のあるプロトコル変更 → 収束
  - P2SH (Pay to Script Hash) スクリプトの導入
  - Segwit (Segregated Witness)

- ハードフォーク: 互換性のないプロトコル変更 → 分裂
  - The DAO事件 (後述) の結果: Ethereum v.s. Ethereum Classic
  - ブロックサイズ拡大: Bitcoin Core v.s. Bitcoin Unlimited

Bitcoinまとめ

- 利点
  - 巧みなインセンティブ設計により、Bitcoinに価値があると思うマイナーはまじめに動作するほうが合理的
    - 単に系を破壊したい場合を除く
- 欠点
  - 電気代の無駄
  - 低速
  - 結託により51%より小さいハッシュパワーでも乗っ取りできる可能性 (Selfish mining)

DLTの実装例 (2): Ethereum

- Bitcoinをもとにチューリング完全なスマートコントラクトの実行系を組み込んだ
- スマートコントラクトの実行に仮想通貨を使う、という建付け
  - The DAO事件を引き起こした
- 任意の資産を扱える

DLTタイプ	Public型
チェーン構造	線形
資産	コイン (ETH) + 任意
TX検証	スマートコントラクト
台帳タイプ	アカウントベース
台帳共有	全共有
コンセンサス	Proof of Work
ファイナリティ	なし
スマートコントラクト	EVM チューリング完全

Bitcoinからの変更点

- アカウント (アドレス) に2種類ある
  - EOA (Externally Owned Account): いわゆるユーザアカウント。
    通貨 Ether (ETH) を保持する
  - Contract: Etherを送信することによってスマートコントラクトを実行できる
- アカウントベース
  - 現在の各アカウントが所持するETHおよび各contractの内部状態がブロックに格納されている
- PoWの変種 (Ethash) を採用
  - ASIC耐性: 専用ハードで高速にハッシュ計算することを防止
  - Proof of Stake (PoS) への移行が計画されている、が依然としてすすまない

スマートコントラクト

- Contract
  - EVMで動作: スタックベースのバイトコード言語
  - Solidityなどの高級言語からコンパイルして実行
  - 実行にはgasが必要。ステップごとに減っていく。0になると実行失敗
  - Storageとよぶ内部状態 (KVS) を持つ
  - 他アカウントへのETH送信ができる
  - 他コントラクトの呼び出しができる
- デプロイ: コントラクトの送信もTX
- 実行
  - コントラクトへEther (と引数) を送信
  - ユーザの指定したレートでEtherがgasに変換される
  - Gasを消費しながら実行。Gasが0になったら失敗
  - 実行が終了したら残ったgasはEtherに変換され送信者に戻される
  - 消費されたgasはマイナーへの報酬になる

ブロックの追加・検証

- アカウントベースなので、マイナーのブロック追加時には「最新の状態」の計算が必要
- TXがコントラクトの呼び出しの場合、その計算時にEVM上のバイトコード実行が行われる
- その結果に基づきコントラクトの内部状態および各カウントの保持するETHの更新が行われ、最新ブロックの内容となる

Ethereumまとめ

- 利点
  - Bitcoinにチューリング完全なスマートコントラクトの処理系を組み込み、多彩な処理が可能になった
- 欠点
  - スマートコントラクトの表現力、それらの間の呼び出しが可能であること、が相まって操作的意味論が複雑になり、致命的なバグを含みかねない (実際に含んだ)

The DAO事件 (The DAO Attack) (2016)

- Ethereum上で出資者が非集中的な意思決定を行う “DAO” の1つ
  - 例えば資金の投資先を出資率で多数決して決める
  - DAO: Decentralized Autonomous Organization
- 1万人以上から7M ETH ($150M) 以上を集める

- 攻撃
  - 2016/6/17
  - Recursive Call Bugを悪用してDAO Splitをループ
  - 6.5M ETHが流出
- ハードフォークにより解決
  - 要はなかったことにした
  - Ethereum (ETH) とEthereum Classic (ETC) への分裂を招いた

参考: The DAOの仕組み

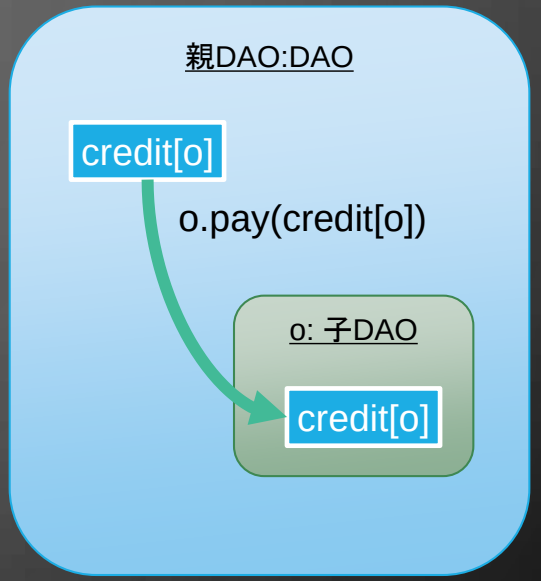
- スマートロックのベンチャーSlok.itが開発
- 入金
  - 出資者はThe DAOにETHを出資
  - The DAOが出資者にDAOトークンを配布
- 投資
  - 投資先の決定はDAOトークンによる多数決
- 出金
  - DAOトークンの引き上げ (Split): 子DAOに資金 (ETH) を移す、という提案をsubmit
  - 子DAOからは28日後から資金を
  - 引き上げられる
- Reference: https://book.mynavi.jp/manatee/detail/id=72171

Recursive Call Bug: Parent DAO

```
Object DAO {
  Map<Object, int> credit
  int balance
  Invariant (sum o: credit[o]) = balance

  Method withdrawAll(Object o) {
    2: if (credit[o] > 0) {
      // 2.5: credit[o] = 0
      3: this.balance -= credit[o]
      4: o.pay(credit[o])
      5: credit[o] = 0
    }
  }
}
```

親DAO this から
子DAO o へ資金を移す
移す際に子DAOの処理が走る



From [Grossman+, POPL ’18]

Recursive Call Bug: Good Child DAO

```
Object DAO {
  Map<Object, int> credit
  int balance
  Invariant (sum o: credit[o]) = balance

  Method withdrawAll(Object o) {
    2: if (credit[o] > 0) {
      // 2.5: credit[o] = 0
      3: this.balance -= credit[o]
      4: o.pay(credit[o])
      5: credit[o] = 0
    }
  }
}
```

```
Object GoodClient {
  Object Dao
  int balance

  Method init(Object dao) {
    1: this.Dao = dao
  }

  Method pay(int profit) {
    2: this.balance += profit
  }
  …
}
```

お金を受け取ったので
内部変数balanceを更新

Recursive Call Bug: Bad Child DAO

```
Object DAO {
  Map<Object, int> credit
  int balance
  Invariant (sum o: credit[o]) = balance

  Method withdrawAll(Object o) {
    2: if (credit[o] > 0) {
      // 2.5: credit[o] = 0
      3: this.balance -= credit[o]
      4: o.pay(credit[o])
      5: credit[o] = 0
    }
  }
}
```

(2) 呼び返された場合、
credit[o] の値は
まだ変わっていない！
→ o.pay() をまた呼べる

```
Object Attacker {
  Object Dao, bool stop, int balance

  Method init(Object dao) {
    1: Dao = dao
    2: stop = false
  }

  Method pay(int profit) {
    3: this.balance += profit
    4: if (!stop) {
      5: stop = true
      6: Dao.withdrawAll(this)
      7: Dao.withdrawAll(this)
    8: }
    9: stop = false
  }
  …
}
```

(1) 呼び返す！

Recursive Call Bug

- ここまで大規模な被害になった原因
  - Recursive call bugの利用
  - ソース中で transfer() を呼ぶべきところを Transfer() を呼んでいるというtypo

DLTの実装例: Hyperledger Fabric v1

- ファイナリティを持つPBFTベースのコンセンサスを採用
- 処理並列化のため各機能コンポーネントを分解
  - MVCC (Multi-version Concurrency Control) による楽観的実行と検証
- 系の最新の状態 (KVS) を保存している

DLTタイプ	Permissioned型
チェーン構造	線形
資産	任意
TX検証	スマートコントラクト MVCC
台帳タイプ	アカウントベース
台帳共有	全共有
コンセンサス	PBFT変種
ファイナリティ	あり
スマートコントラクト	Go言語 チューリング完全

参考: Hyperledger Fabric v0.x

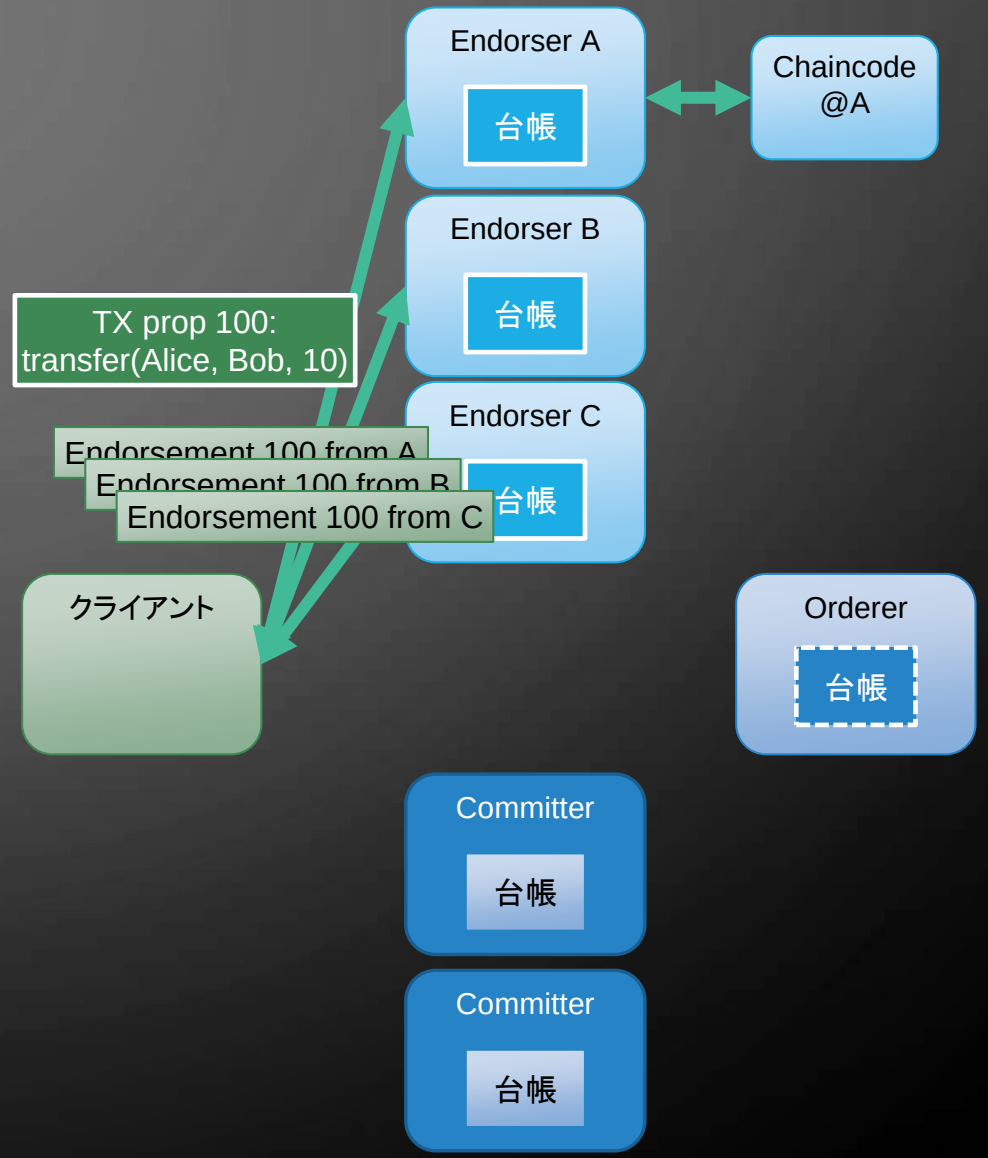
- HyperledgerプロジェクトのPermissioned型実装の1つ
- ほぼPBFT [Castro and Liskov 99] に基づいた実装
- スケーラビリティ的には不利であった
  - 全ノード間でbloadcastを繰り返すための通信オーバーヘッド
  - TXの実行がlinearに行われることによる非並列化オーバーヘッド

登場人物 (いずれもDockerコンテナ)

- Endorser
  - TXのsimulationを行うノード。Committerも兼ねている
- Commiter
  - Ordererからブロックを受け取って (それを検証し)、チェーンに追加するノード
- Orderer
  - TXの順序付けを行い、Committerにブロックを配布するノード
- Client
  - TXのsimulationをendorserに依頼して結果を取りまとめ、ordererに送るプログラム

プロトコル (1) TXの送信とsimulation

- Clientは必要なEndorsersにTX proposalを送信し、simulationを依頼する
- Endorserは現在の台帳の状態をもとにスマートコントラクトの実行 (simulation) を行う
  - スマートコントラクト (チェーンコードとよぶ) もコンテナ上で動作
- 実行後、KVSのRead/Write set (RWSet) などを計算しclientに返す
  - この時点ではKVSの更新は行わない
  - 結果をendorsementとよぶ



RWSetの計算

- KVSの各エントリは (key, value, version)
  - 自然数とみなせる
- Read setの各エントリは (key, version)
- Write setの各エントリは (key, new_value)

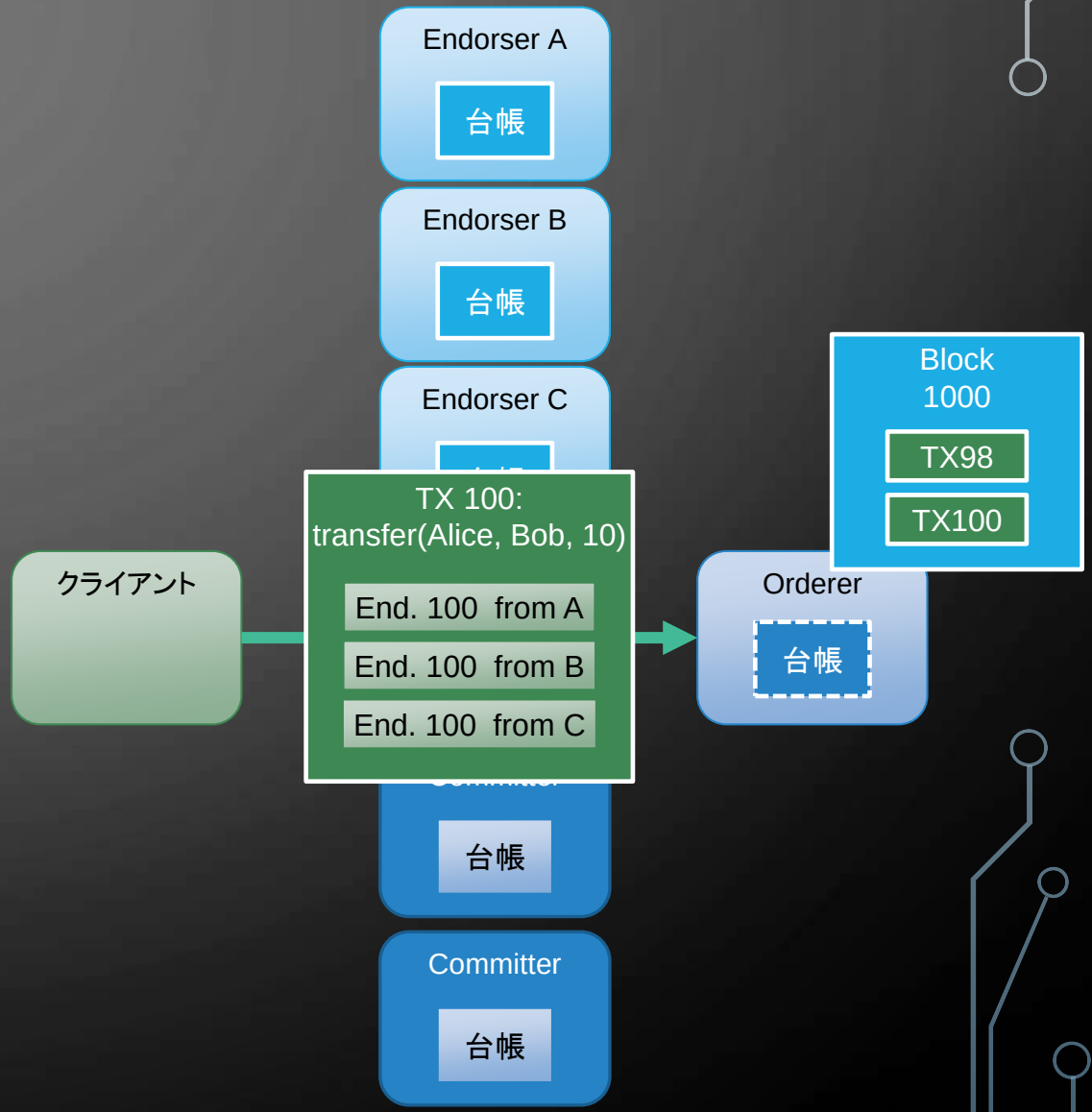
Key	Value	Version
Alice	100	5
Bob	10	3
…		

Simulates transfer(Alice, Bob, 10)

Read set: [(Alice, 5), (Bob, 3)]
Write set: [(Alice, 90), (Bob, 20)]

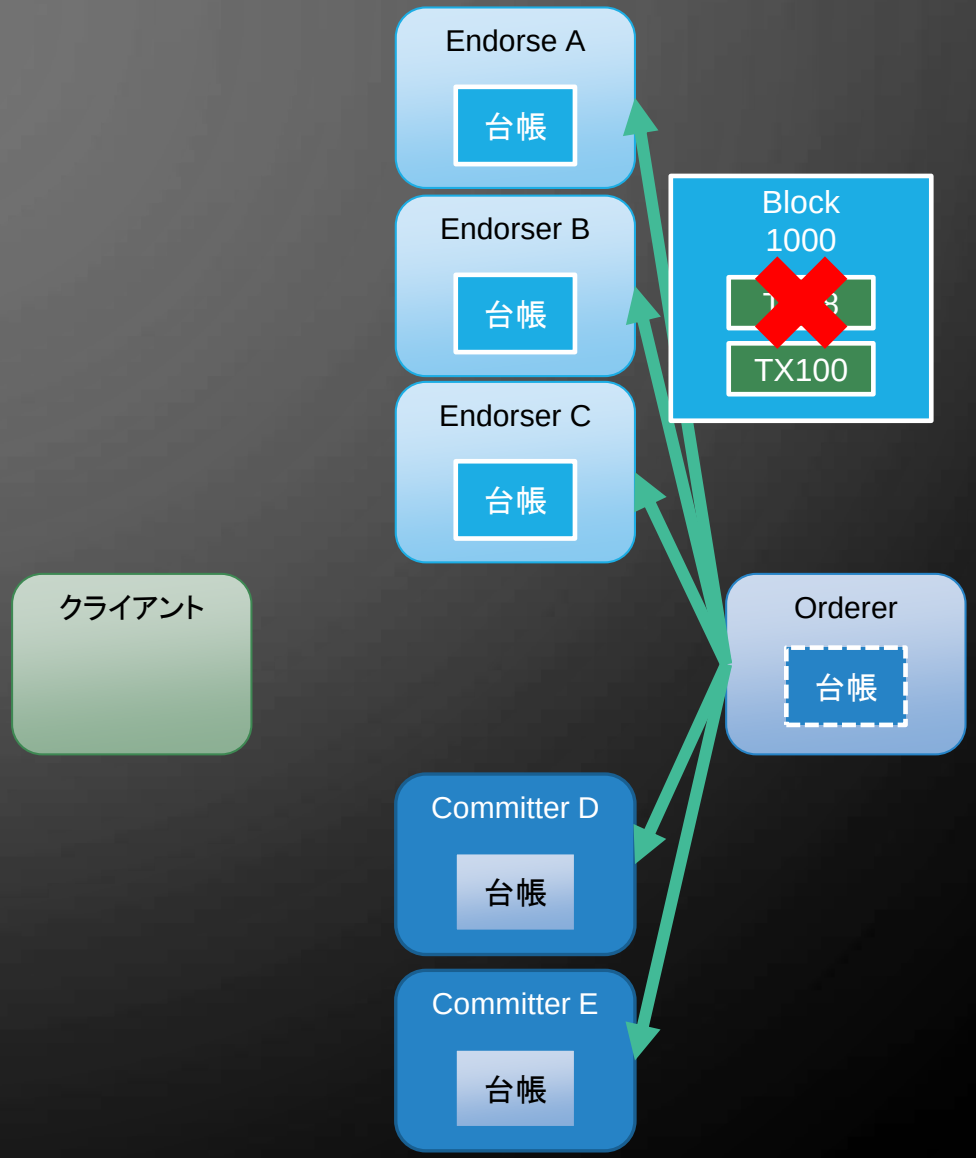
プロトコル (2) Ordererでの順序付け

- Clientは必要なだけの (endorsement policy を満たせるだけの) endorsementが集まったらTXにまとめ、Ordererに送信する
- Ordererは届いたTXを一列に順序付けし、いくつかまとめてブロックにする
  - Ordererはブロック (TX) の内容の検証を行わない



プロトコル (3) Blockの配布とチェーンへの追加

- OrdererはブロックをCommitter (Endorser含む) に配布する
- Committerは届いたブロック (とTXs) の検証を行い、検証結果と共にチェーンに追加する
  - TXに含まれるEndorsementが互いに一致しないならrejectフラグを立てる
  - EndorsementがpolicyをTたさないならrejectフラグを立てる
  - MVCC: Endorsementに含まれるread setのバージョンが現在のKVSのバージョンと一致するならacceptフラグ、そうでないならrejectフラグ



Endorsement Policy

- TXが正当であると規定するためのendorsementに関する制約を endorsement policyとよぶ
  - Policyはスマートコントラクトのdeploy時に指定する
- Policy例
  - 3個以上のendorsementが必要でそのうち2個以上が一致する必要がある
  - 基本的には *m*-of-*n* をネストしたもの

- Policyの設定はビザンチン耐性と密接な関連があるので慎重な設定が必要
  - ノードが全4台、policyを2-of-3とすると最大1台までの障害ノードを許容する ビザンチン耐性を実現できるようにクライアントを設計できる

参考: MSP (Membership Service Provider)

- Hyperledger Fabricは複数の利害関係者による業務使用を想定している
- ノードやユーザがあるFabricネットワークに属しているかどうかをMSP単位で判定する
- 例
  - MSP IBMはノードA, B, CおよびユーザMをメンバとする
  - MSP BOJはノードD, EおよびユーザN, Oをメンバとする
- Endorsement policyも正しくはMSPを意識した形で書く
  - 例: 3-of-3(IBM, BOJ, JPX): それぞれのOrgからのendorsementが必要で、かつ、それらがすべて一致する必要がある

Hyperledger Fabric v1まとめ

- 利点
  - Permissioned型ならではの、ファイナリティのあるコンセンサス
  - TX simulationの並列化による高パフォーマンス
  - Docker採用による環境非依存性
- 欠点
  - 設定の複雑化
  - High frequencyな環境ではTXのrejectが多発する
  - 匿名性の欠如 (v1.1で対応予定)
  - クライアントのやるべきことが多い

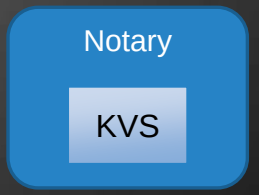
DLTの実装例: R3 Corda

- 業務アプリケーションで必要なプライバシーの保護に特化
- 金融アプリを想定
- 関係者にしかTXを送らない
- NotaryがUTXO検証

DLTタイプ	Permissioned型
チェーン構造	資産インスタンスごと 線形
資産	任意
TX検証	Notary + スマートコントラクト
台帳タイプ	UTXOベース
台帳共有	一部共有
コンセンサス	Notary
ファイナリティ	あり
スマートコントラクト	コントラクトコード (on JVM) チューリング完全

登場人物

- ノード
  - ネットワークを構成するマシン。ユーザとほぼ1:1
- Notary (公証人)
  - TXの二重使用を検証するためのサービス。ノード上で動作

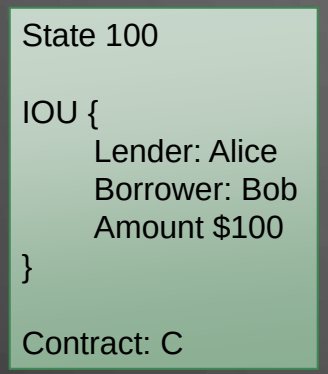


参考: 他のDLTとの相違点

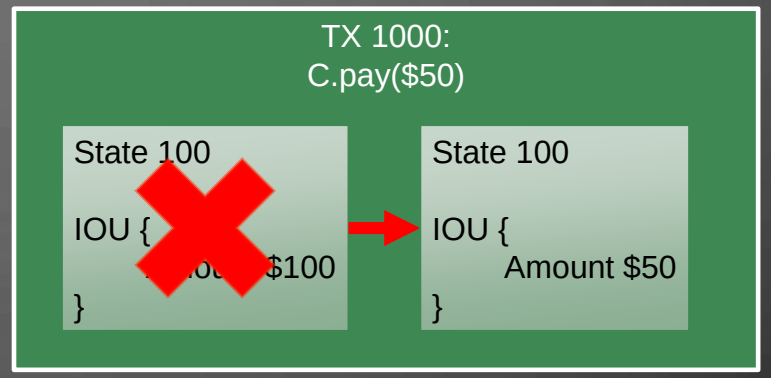
- (Bitcoinと同じ) UTXOをもとにしたTX構成
  - そもそもUTXOとは何であるか? → コインの所有先を表す
- (Bitcoinと異なる) ただしコイン以外の任意のデータ型を構成できる
  - Cordaではstateとよぶ
  - (Cordaは金融アプリを想定しているので) なんらかの債権を表すことが多い
    - IOU = I Owe You
- (Bitcoinと異なる) TXは関係者としか共有しない
- (Hyperledgerと同じ) スマートコントラクトの実行結果が決定的でないといけない

Cordaのデータ構造

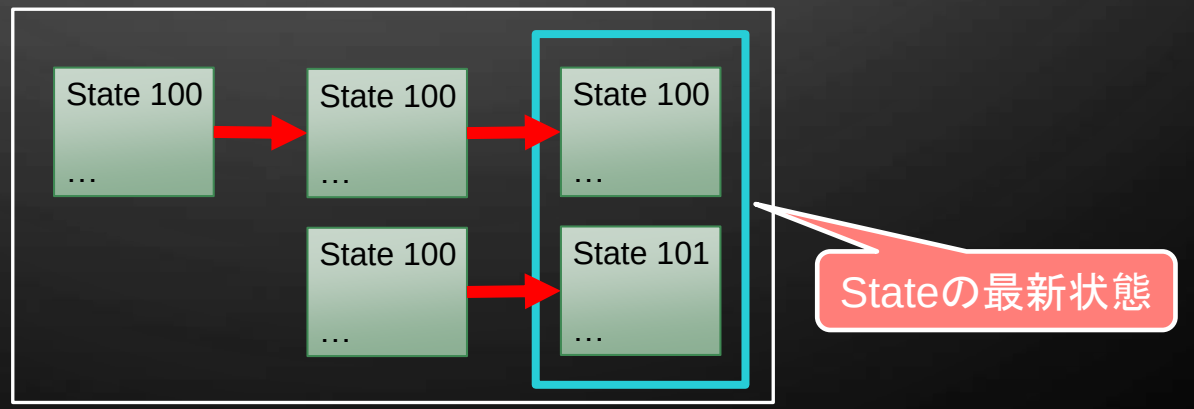
State
インスタンスに相当
対応付けられたコントラクトC



TX
古いstatesをinputにし、新しいstatesをoutputに指定したもの
※古いstateはunspentでないといけない



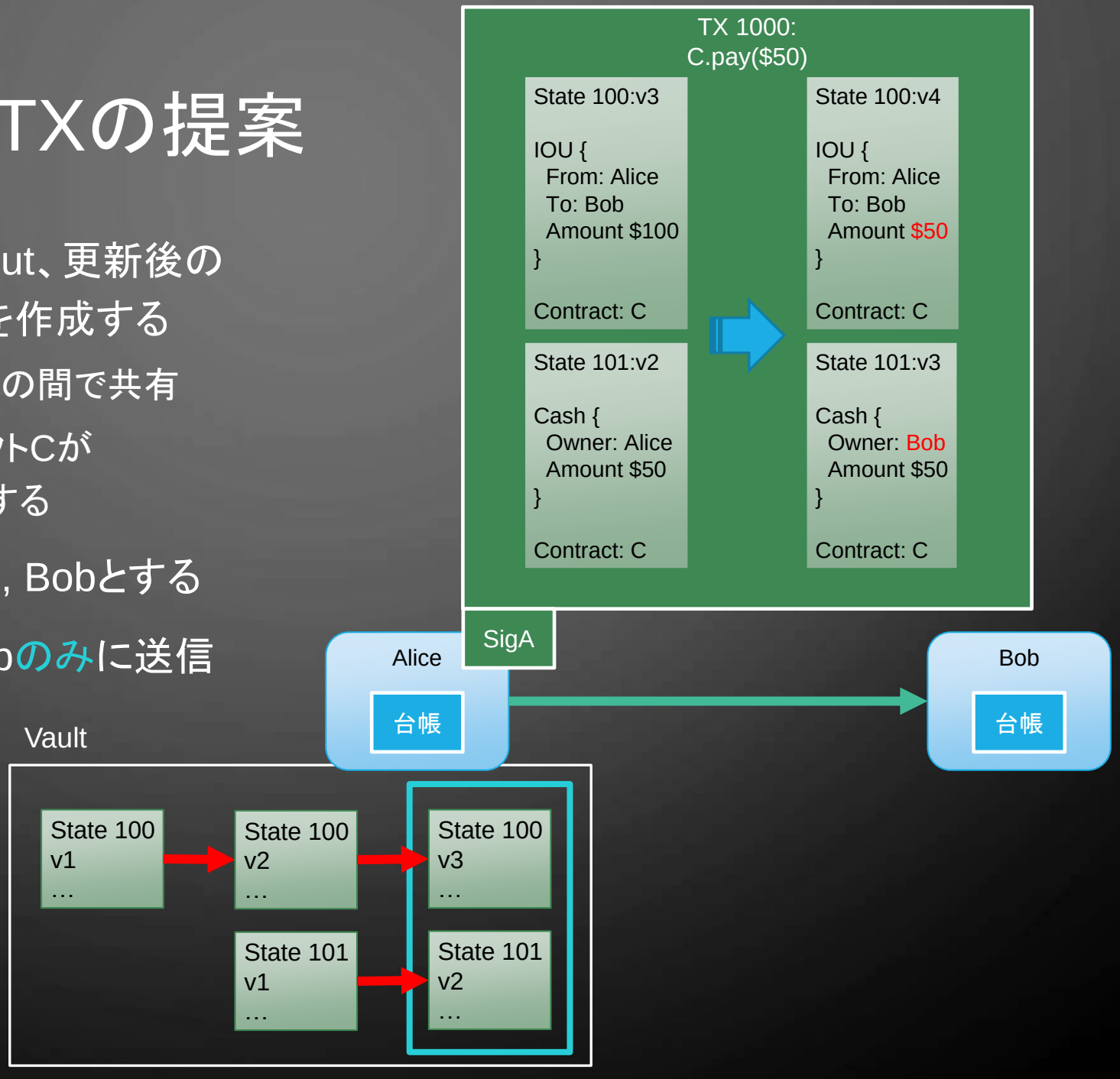
Vault
いわゆる台帳。Spent、unspentふくめ全てのstateを保持



あるノードのVault

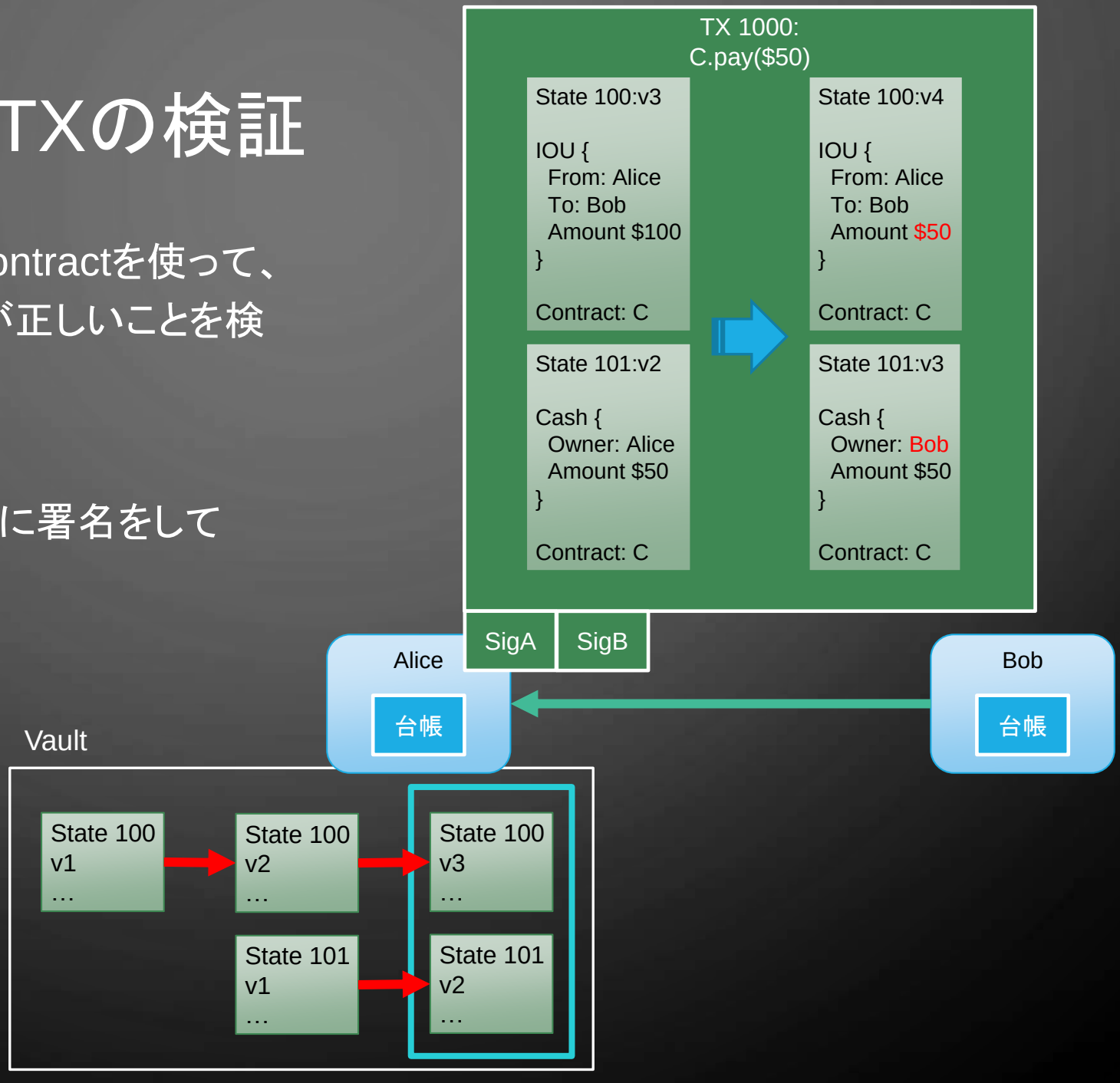
プロトコル (1): TXの提案

- Aliceが現状のstateをinput、更新後のstateをoutputに含むTXを作成する
  - このstateはAliceとBobの間で共有
  - このstateにはコントラクトCが関連付けられているとする
- TXのParticipantsをAlice, Bobとする
- AliceはTXに署名してBobのみに送信



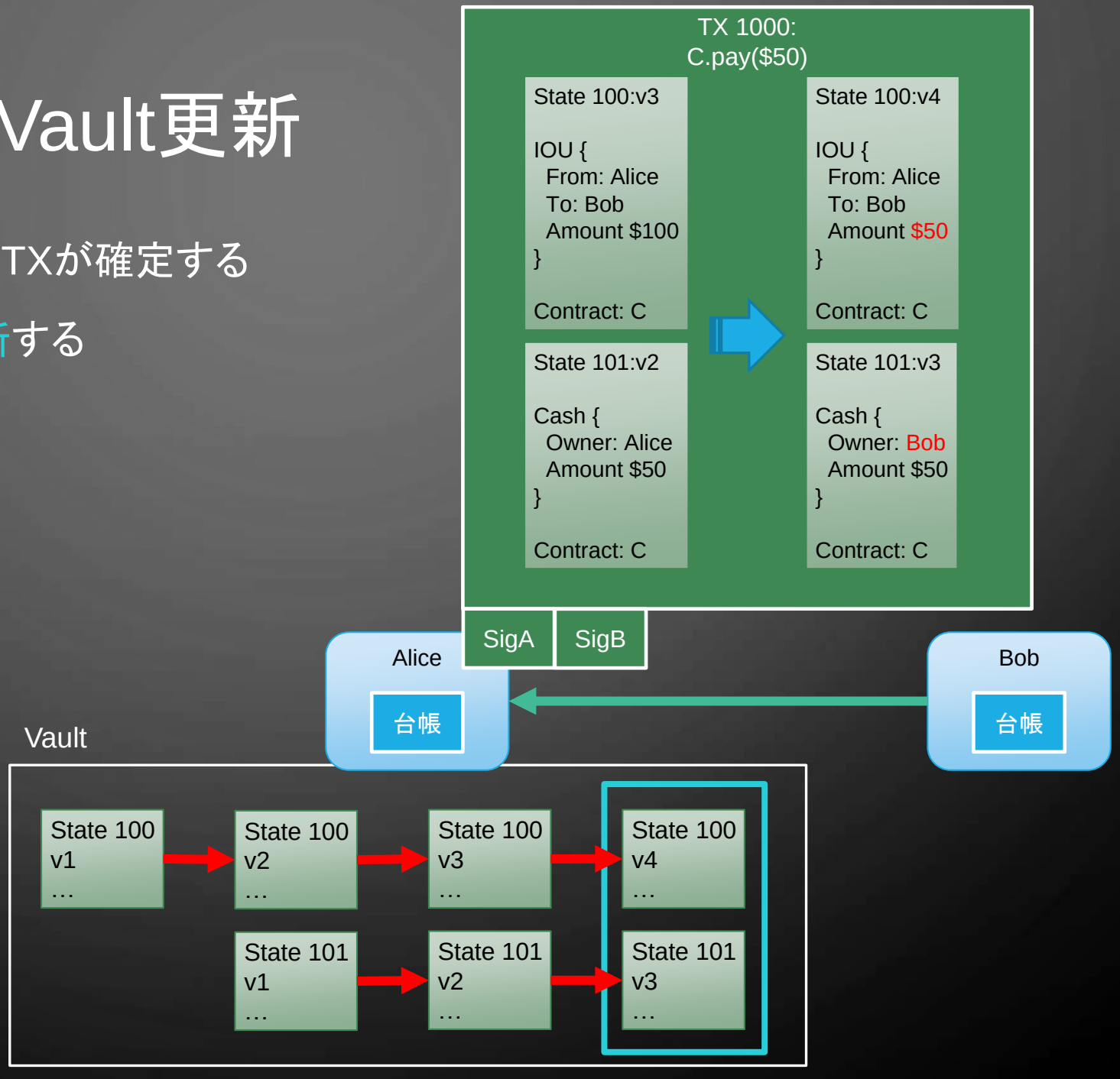
プロトコル (2): TXの検証

- Bobは関連付けられたContractを使って、提案された更新後stateが正しいことを検証する
  - UTXO制約も確認する
- 検証を通ったらBobはTXに署名をしてAliceに送り返す



プロトコル (3): Vault更新

- 両者の署名が得られたらTXが確定する
- AliceとBobはVaultを更新する



Cordaまとめ

- 利点
  - 業務アプリケーションに特化したプライバシーの実現
    - 他にも (省略したが) オラクルサービスなど
  - 通信量削減とプライバシー保護のためのP2Pプロトコル (ブロードキャストしない)

- 欠点
  - 資産が多人数の間を渡るにしたがい、プライバシーがなくなっていく
  - Stateの遷移タイミングの厳密な順序関係はわからない
  - P2Pプロトコルの欠点をカバーする特権ノードが必要

DLT実装の検証

DLTに求められる性質 [Yoshihama and Saito 17]

- Safety/Consistency
  - 確定したブロックやTXが求められる制約を満たす (例: 二重使用が起こらない)
  - Authenticity/Non-repudiation
  - (スマートコントラクト) 意図しない実行パスを辿らない
- Liveness (ファイナリティ): コンセンサス手続きが完了する
  - Public型の場合は漸近的ファイナリティ
  - 決定性が重要になることが多い
- Tamper Resistance: チェーンの改竄耐性
- Privacy/Anonymity:
  - 主体者の秘匿、TX内容の秘匿、TXの存在の秘匿
- Resiliency
  - 障害からの復旧がいかになされるかは実運用で非常に重要

(古典的な) 否定的結果

- FLP定理 [Fischer, Lynch and Paterson, JACM 85]
  - 各ノードが決定的に動作し、通信がasynchronousならば、1台でもfaultyなノードがいると合意は実現できない
  - タイムアウトや確率的アルゴリズムで回避
- (Brewerの) CAP定理 [Gilbert and Lynch, SIGACT News '02]
  - データ複製において、Consistency、Availability、Partition-toleranceの全てを保証することは不可能である
  - DLT実装では上記条件のいずれかを妥協して実現

検証例: 確率的解析

- Selfish miningの解析 [Eyal and Sirer, FC ‘14]
  - Selfish mining: マイナーが結託し、マイニングしたブロックを公開しないまま次のマイニングを続ける
  - 結託側/その他のハッシュパワー比を α:1-α としてマルコフ過程を構築

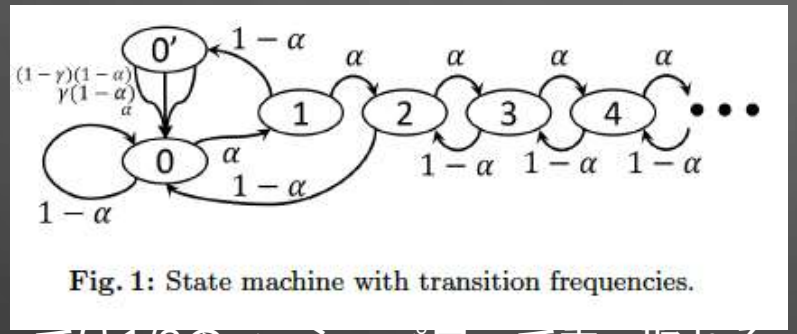
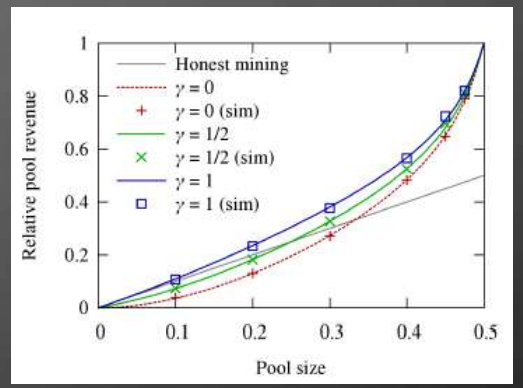


Fig. 1: State machine with transition frequencies.

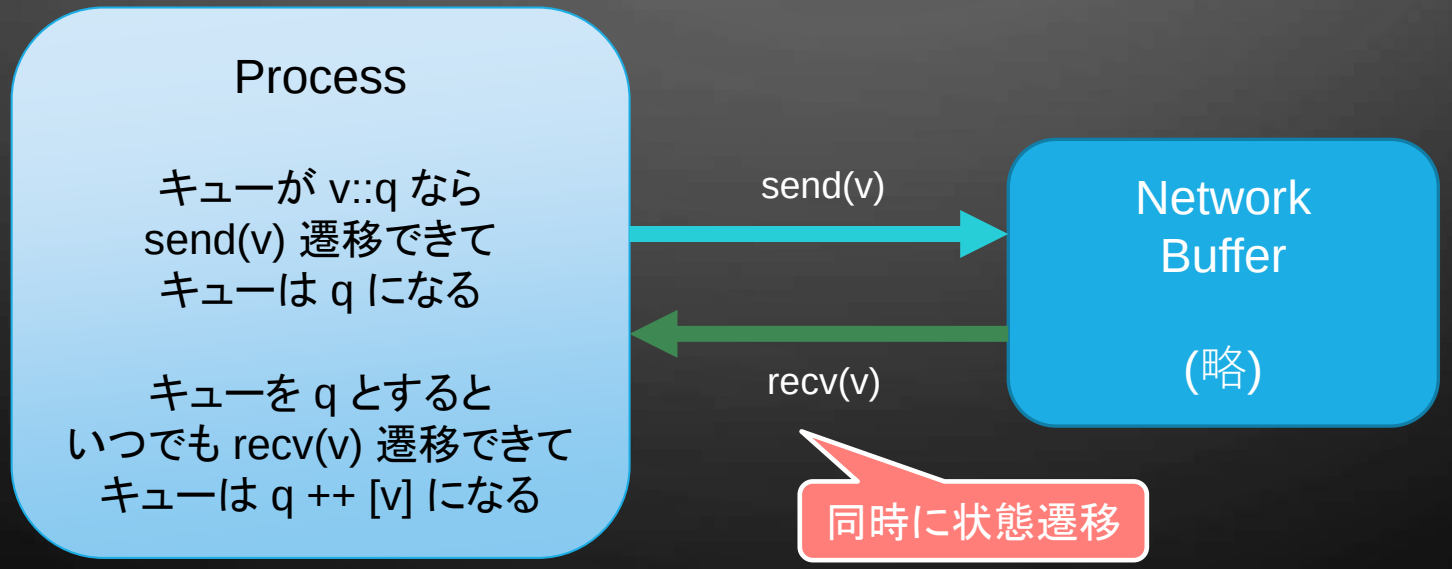


  - 条件によっては1/3のハッシュパワーで乗っ取れる
- ほかには [Garay and Kiayias, Eurocrypt ‘15] など

参考: 松尾ほか「ブロックチェーンの未解決問題」

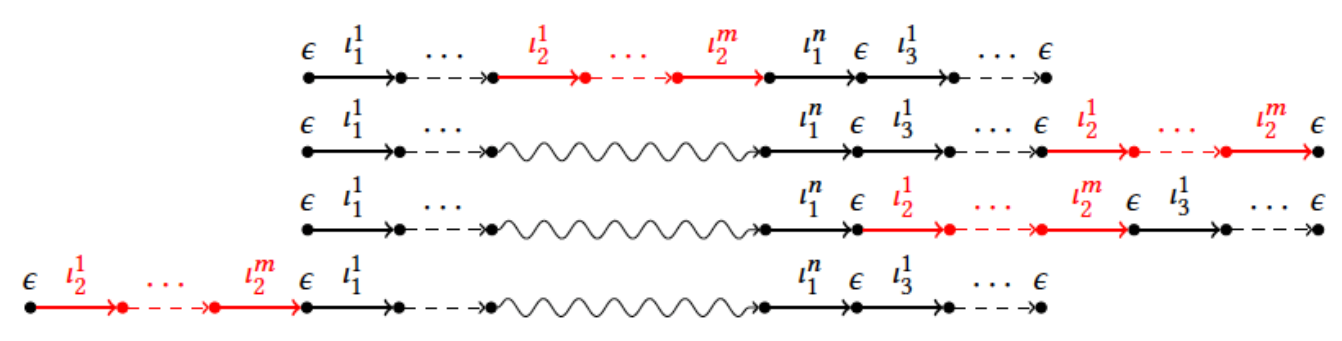
検証例: プログラム/システム検証

- I/Oオートマトン
  - PBFT (Practical Byzantine Fault Torelance) の定理証明 [Castro+, OSDI ’99]
    - 実装をI/Oオートマトンで記述
    - 仕様 (単一ノードからなるステートマシン) をI/Oオートマトンで書いてsimulationを証明
  - もちろんCSP [Hoare 78] でもCCS [Milner 80] でもπ計算 [Milner 92] でもよい



検証例: プログラム/システム検証

- Callbackを判定するようなプログラム解析 [Grossman+, POPL '18]
  - コントラクトは、任意のtraceからcallbackを含まない等価な (= 状態を保存する) traceが構成できるとき、Effectively Callback Free (ECF) であるという定義を与えた
    - つまり、callbackで呼ばれたときの状態がきれいならcallbackありでもECFといえる
  - コントラクトがECFであるか否かの決定手続を与えた



1行目のtraceと等価な
callback free traceの例

- スマートコントラクトの決定性解析 [Iwama+, PPL '18] (このあとすぐ)

検証のためのツール

- モデル検査
  - SPIN、NuSMV、FDR4
  - Prism
  - UPPAAL
  - 自動だが複雑なモデルを表現しにくい
- 証明ベース
  - Event-B、仕様記述言語
  - Coq、Agda、Isabelle/HOL、Lean、その他
  - 現状では人手の介入が必要だが複雑なモデル (例: 無限集合など) が表現可能

まとめ

- ブロックチェーンに代表される分散台帳技術において、
  実装の形式的な検証は非常に有効である一方、複雑な分散システムを
  記述・検証できるための高度な手法やフレームワークが要求される
- 本会議参加者のみなさまの知見により、
  この分野の研究がさらに進むことを願います

参考文献 (技術資料)

- Bitcoin
  - Bitcoin Wiki. https://en.bitcoin.it/wiki/Main_Page
  - Bitcoin Project. https://bitcoin.org
  - Bitcoin日本語情報サイト (2017). ビットコイン解説 > ビットコインとは何か?: https://jpbitcoin.com/about/whatisbitcoin1
  - 漆嶌賢二 (2014). Bitcoinを技術的に理解する: https://www.slideshare.net/kenjiurushima/20140602-bitcoin1-201406031222
- Ethereum
  - Ethereum Project. https://www.ethereum.org/
- Hyperledger Fabric v1
  - http://hyperledger-fabric.readthedocs.io/en/release-1.0/
- R3 Corda
  - Cordaで認識相違のない取引を. https://www.corda.net/ja/

参考文献 (論文)

- S. Grossman et al. (2017). Online detection of effectively callback free objects with applications to smart contracts. POPL 2018, Article 48. DOI: https://doi.org/10.1145/3158136
- 岩間, 立石, 齋藤, 天野, 吉濱 (2018). ブロックチェーンにおけるチェーンコードの決定性の解析. PPL ’18.
- 吉濱佐知子, 齋藤新 (2017). 分散台帳技術におけるインテグリティとプライバシー保護. CSS ’17.
- C. Cachin and M. Vukolić (2017). Blockchain Consensus Protocols in the Wild. http://arxiv.org/abs/1707.01873.
- J. Garay, A. Kiayias, N. Leonardos (2015). The Bitcoin Backbone Protocol: Analysis and Applications. EUROCRYPT ’15. DOI: https://doi.org/10.1007/978-3-662-46803-6_10. Latest version (2017) appears at https://eprint.iacr.org/2014/765.pdf
- I. Eyal, E. Sirer (2014). Majority is Not Enough: Bitcoin Mining is Vulnerable. FC ’14. DOI: https://doi.org/10.1007/978-3-662-45472-5_28

参考文献 (論文) (2)

- S. Nakamoto (2008). Bitcoin: A Peer-to-Peer Electronic Cash System. http://bitcoin.org/bitcoin.pdf.
- S. Gilbert and N. Lynch (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News 33 (2). DOI: https://doi.org/10.1145/564585.564601
- M. Castro and B. Liskov (1999). Practical Byzantine fault tolerance. OSDI ’99.
- M. Fischer, N. Lynch and M. Paterson (1985). Impossibility of distributed consensus with one faulty process. J. ACM 32 (2). DOI: http://dx.doi.org/10.1145/3149.214121