BBc-1 design document

revision 2

for v0.9

28 Feb. 2018

本資料について

- BBc-1の設計思想、実装、利用方法についてまとめる
  - 対象のBBc-1はgithubに公開されたv0.9 (2018/2/28バージョン)である
    - https://github.com/beyond-blockchain/bbc1
  - BBc-1のWhitePaper、YellowPaper (Analysys)はgithubリポジトリのdocs/、および下記URLに公開されている
    - https://beyond-blockchain.org/public/bbc1-design-paper.pdf
    - https://beyond-blockchain.org/public/bbc1-analysis.pdf
  - 本資料の内容に起因するあらゆるトラブルには責任を追わない

- 作成日 : 2018/2/28
- 作成者 : takeshi@quvox.net (t-kubo@zettant.com)

Change log

- 2017/10/31: 初版
- 2018/2/28: 第2版

目次

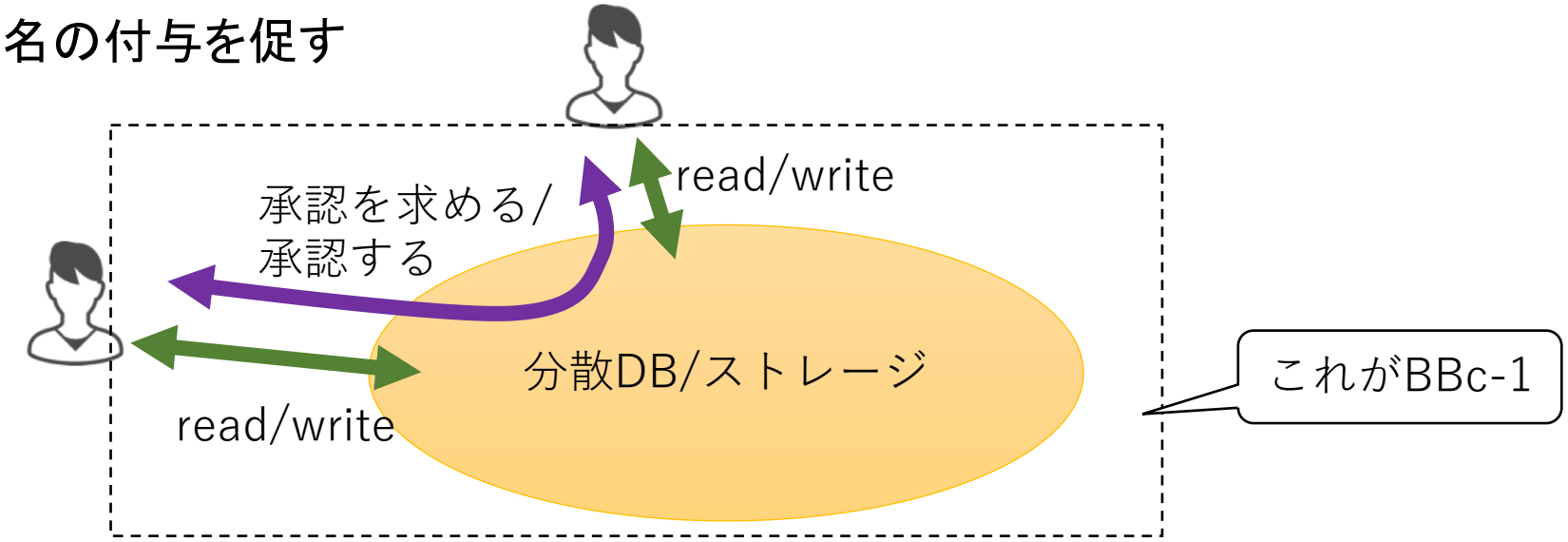
設計思想とアーキテクチャ

BBc-1とは

- BBc-1は既存のブロックチェーン技術およびそれを利用したプラットフォームが抱える下記のような課題を解決することを目的に開発された技術およびプラットフォームの総称である
  - ネイティブ通貨（暗号通貨）の市場価値暴落によるシステム機能不全リスク
  - ブロックのマイニングによるファイナリティの問題
  - 柔軟なシステムコンフィギュレーションの難しさ

BBc-1は何ができるのか？

- 改ざんが困難な分散DB/ストレージ
  - アセット間の関連付けを可能にするデータ構造を用いた情報保存
  - デジタル署名に基づく検証
  - 改ざん検出・回復
- 書き込みの前に「他のentityに承認を得る/承認する」ためのメッセージング機能
  - デジタル署名の付与を促す



BBc-1の設計思想

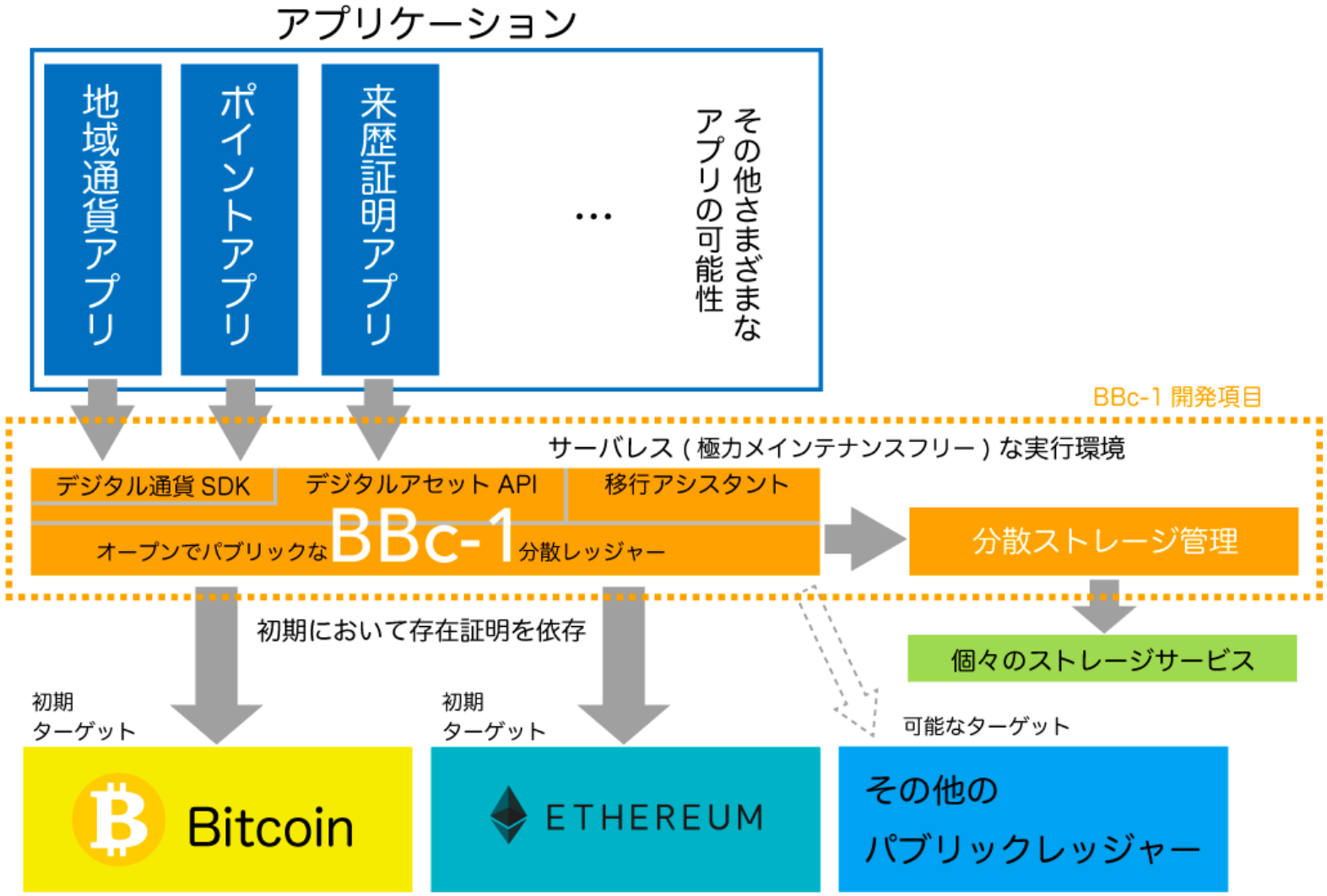
- 本当に大事なことにだけフォーカスする

情報が、たしかにそのentityによって登録され、改ざんされていないということを他entityが確認できるようにする

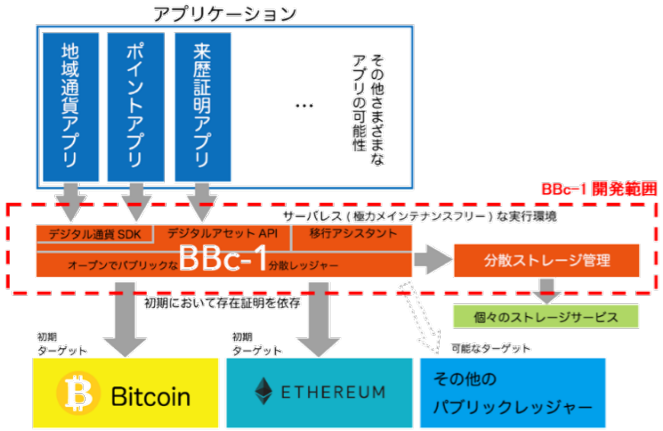
  - これさえできれば、真贋証明、価値移転、スマートコントラクトなど現在ブロックチェーン関連技術で注目されている全てのサービスが実現できる

シンプルで拡張性が高くインターネットにおけるTCP/IPのようなイメージ

BBc-1アーキテクチャ



BBc-1アーキテクチャ



- 既存プラットフォームの利用
  - BBc-1の認知度が低い段階
    - BBc-1単体でも存在証明可能だが、その証拠を外部システム（EthereumやBitcoin）に求め、追認してもらえるようにする
    - 利用する外部システムは自由に選択、切り替え可能
      - v0.7ではEthereumに対応。Bitcoinにも対応予定
  - BBc-1が普及した時
    - 外部システムを切り離し、BBc-1単体で動作させる
      - タイミングはシステム開発者が決めればよい（初期から切り離しても良い）
- 分散ストレージ
  - ファイルの格納場所には、BBc-1のコアシステム内だけでなく、外部ストレージも指定できる
- 分散レッジャー
  - 取引情報を保存するとともに、P2Pネットワーク内で自動的にコピーを配信する

用語の定義

BBc-1における基本概念

- ユーザ（user）
  - BBc-1を利用するエンティティ
  - userは、公開鍵のペアを自分自身で生成する必要がある
- アセット（asset）
  - userが保有するデジタル資産で、BBc-1にその存在が登録されたもの
  - assetにはその所有者のようなメタ情報も含まれる
- トランザクション（transaction）
  - assetの状態変更の手続きを記録した情報（登録、更新、所有権移転など）
- アセット種別（asset_group）
  - 複数種類のassetを取り扱うために、BBc-1ではそれぞれの種別を識別する
    - 例：社内機密ファイル、地域通貨、土地、センサーデータ、etc,,,

BBc-1システムの構成要素

- コアノード（core nodeまたは単にnode）
  - BBc-1のサービスを提供するコンピュータ
  - node同士がP2Pネットワークを構成し、連携してサービスを提供する
- ドメイン（domain）
  - あるasset_groupのasset群およびそれに関連するtransactionの情報が分散配布される範囲
  - domainごとに1つのP2P overlay networkが形成され、その中でのみ情報やメッセージが共有、伝送される

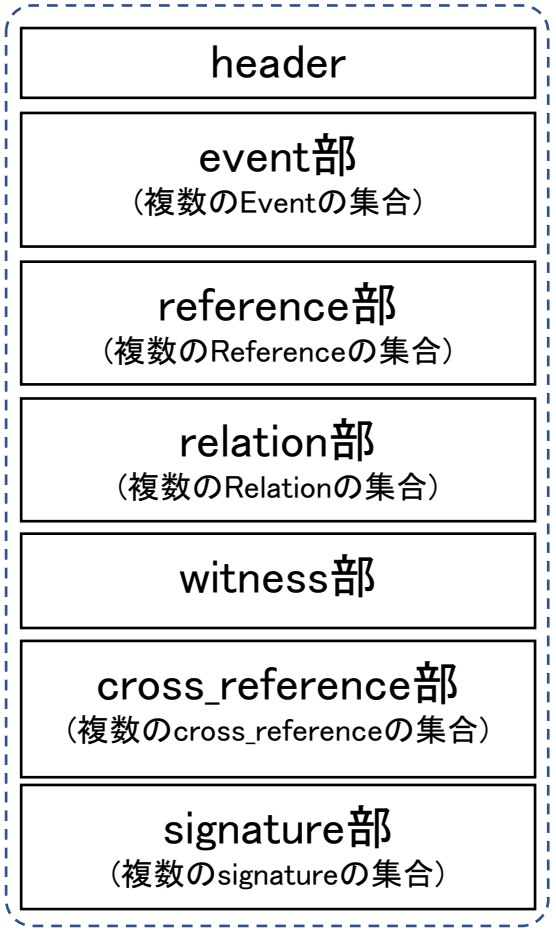
識別子

- 識別子は全て256bitである（SHA256で発生させる）

名前	説明	ユニークネス
user_id	ユーザentityの識別子 （entityごとに鍵ペアを持つ）	asset_groupごとにID空間が定義される。つまり、1つのasset_groupの中では、user_id、asset_id、transaction_idの値はユニークでなければならない。
asset_id	アセット識別子 （アセットのファイルのファイル名にもなる）	
transaction_id	トランザクション識別子	
node_id	ノード識別子 （ドメインごとに作成される）	domainごとにID空間が定義される。つまり1つのdomainの中で、asset_group_idとnode_idの値はユニークでなければならない。
asset_group_id	アセット種別の識別子	グローバルにユニークでなければならない（ただし完全にプライベートだけで使うならその中でユニークであればよい）
domain_id	ドメイン識別子	

トランザクションの構造

改竄耐性をもつトランザクション構造



バージョン及びタイムスタンプ

新しく登録されることになるassetに
関する情報が格納される

eventで新しく登録されるassetに関連する過去のasset
情報に関するtransactionのtransaction_idが格納される

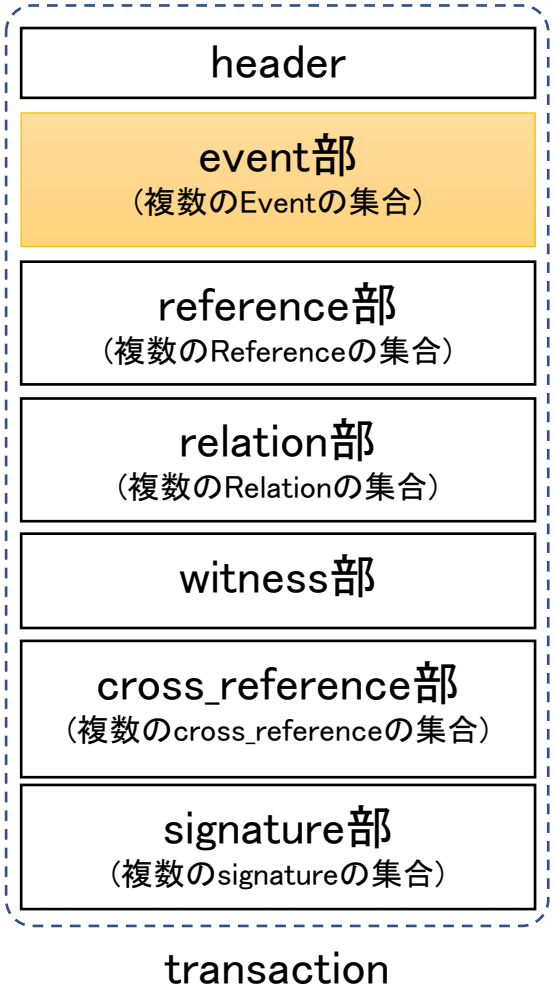
新しく登録されることになるassetとそれに関連する
過去のtransactionに関する情報が格納される

どのユーザ(user_id)が電子署名を施したかを示す

全く無関係なdomainのtransactionの存在
を証明してあげるために存在する

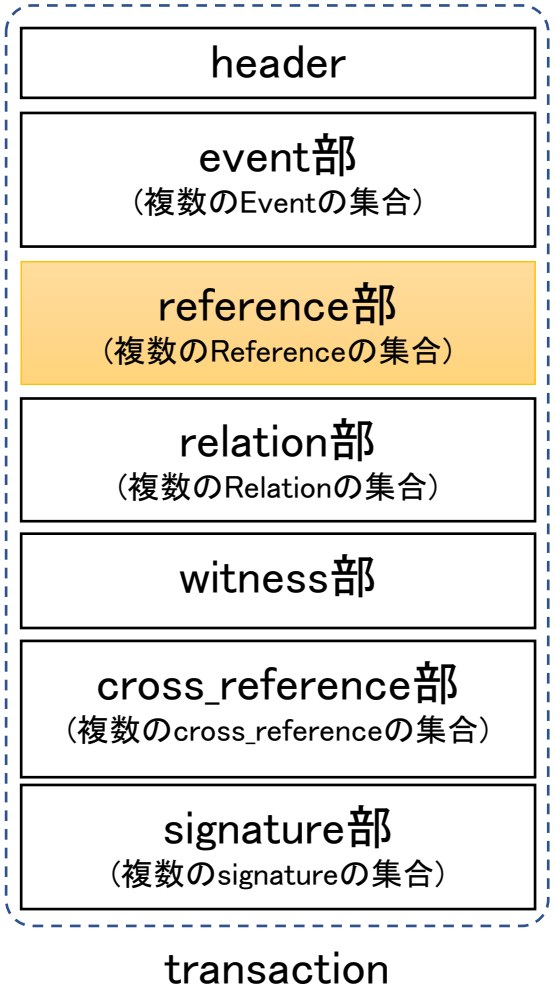
header～cross_referenceまでのデータに対して、
複数のユーザが電子署名を付与する

Event部



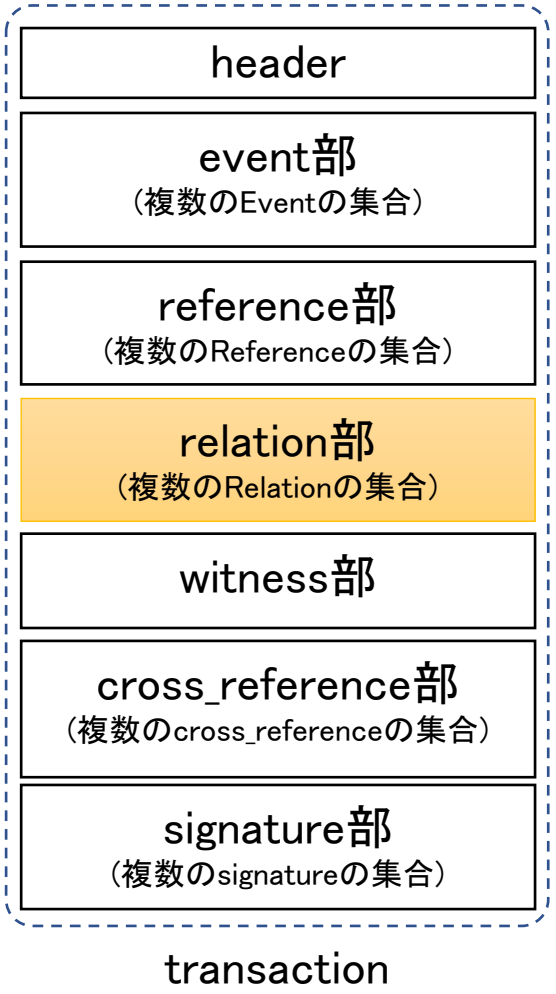
- 含まれる情報
  - このEventに書かれたassetを変更する場合に、誰の承認を得なければならないか？
    - mandatory_approversʼuser_id
    - optional_approversʼuser_id
      - optionalによって、M-out-of-N のマルチシグが可能になる
  - assetに関する情報
    - asset_id
    - asset_group_id
    - 作成者 user_id
    - asset情報のサイズ
    - asset情報
      - 推奨は、256バイト未満（システム設定で変更可能）
      - ここに入らないほど大きい情報は、ファイルとして分散ストレージに保存される(ファイル名=asset_id)

Reference部



- 含まれる情報
  - 参照すべきtransactionとその中のevent
    - asset_group_id
    - transaction_id
    - そのtransactionの何番目のeventか？
- 電子署名による承認
  - 過去のtransactionのevent部にて登録されたassetをこのtransactionで変更を加える（例えばファイル更新や所有者移転）ことになるので、「その行為に対する承認を得る」必要がある
  - 誰に承認を得るべきかは、指定されたevent部のmandatory approversおよびoptional approversに記載されている

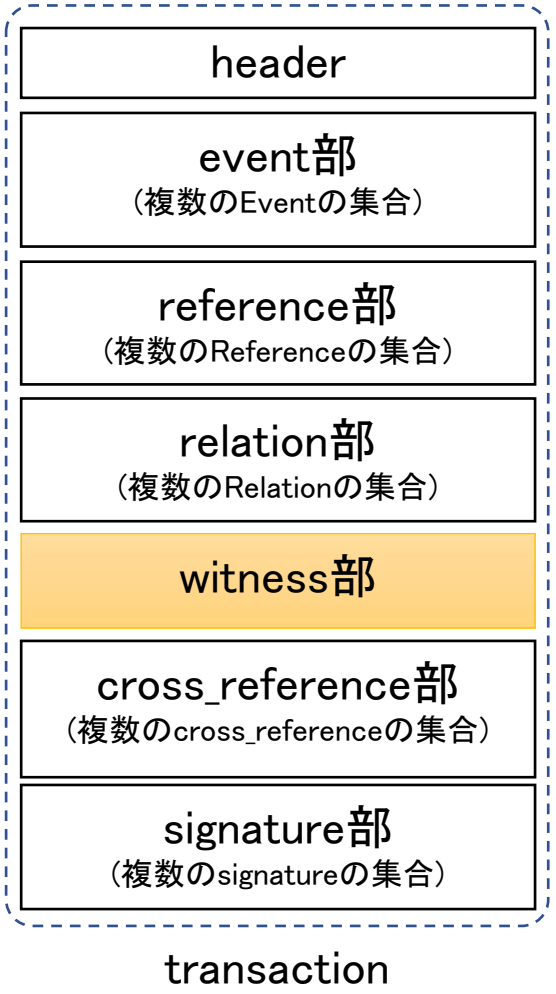
Relation部



- 含まれる情報
  - asset_group_id
  - 過去の関連するtransaction/asset情報
    - transaction_id
    - asset_id
  - assetに関する情報（Event部の中のassetと全く同じ）

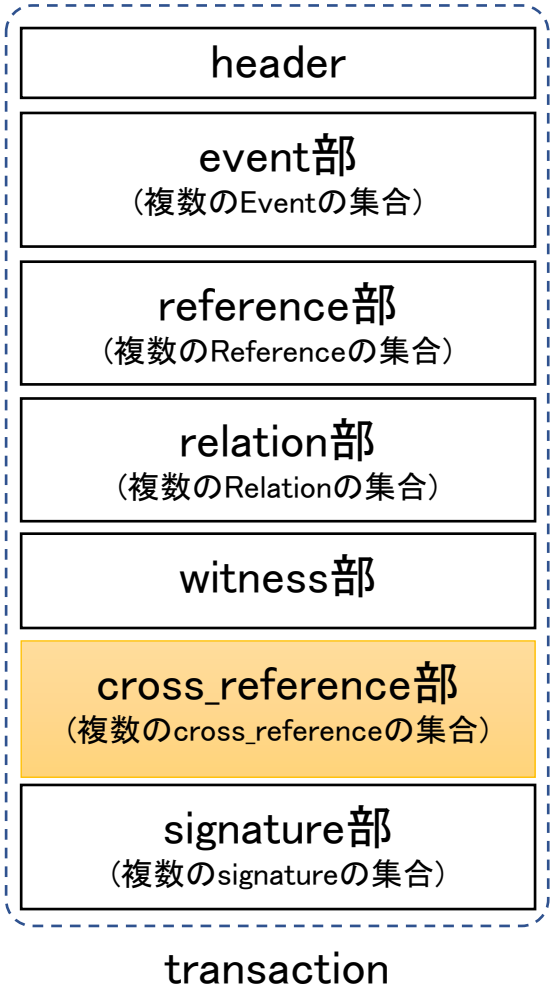
- Event部との違い
  - Event部およびReference部はUTXO (Unspent Transaction Output)構造を意図しているが、Relation部はUTXOとは全く関係なく、単純にassetの情報とそのアセットに関連するassetの情報（transactionとその中のasset）を記述する事ができる
    - approverを明示するというEventが持つ特性を持たない
  - 中身をどのように解釈するかはアプリケーション次第

Witness部



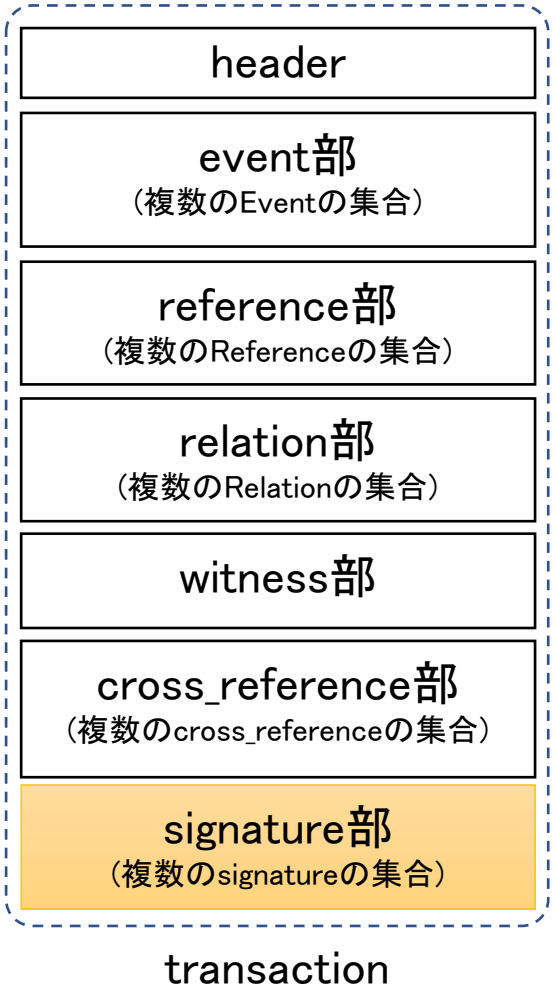
- 含まれる情報
  - user_idのリスト
  - signatureの配列要素のリスト
- 電子署名を付与していることの明示
  - どのユーザ(user_id)がどの署名を行ったかを示す
  - signature部にはpublic keyしか格納されないため、アプリケーションからは使いにくいことが想定されるので導入した

Cross_Reference部



- 含まれる情報
  - domain_id
  - transaction_id
- 効用
  - ここに書かれたtransaction_idが「確かに存在した」ということを証明することになる（signature部の署名で保護されるため）
  - 全く無関係(可能なら無関係なdomain上)のtransaction_idをここに載せるべき
    - トランザクション削除などの改竄によってtransaction自体の存在を否認するためには、無関係なdomainのtransactionまで改ざんしなければならなくなるため、強い改竄耐性をもたせることができる

Signature部



- 含まれる情報
  - 鍵タイプ
  - 鍵長
  - 公開鍵
    - v0.9では、非圧縮のECDSA（SECP256k1）に対応
  - 電子署名

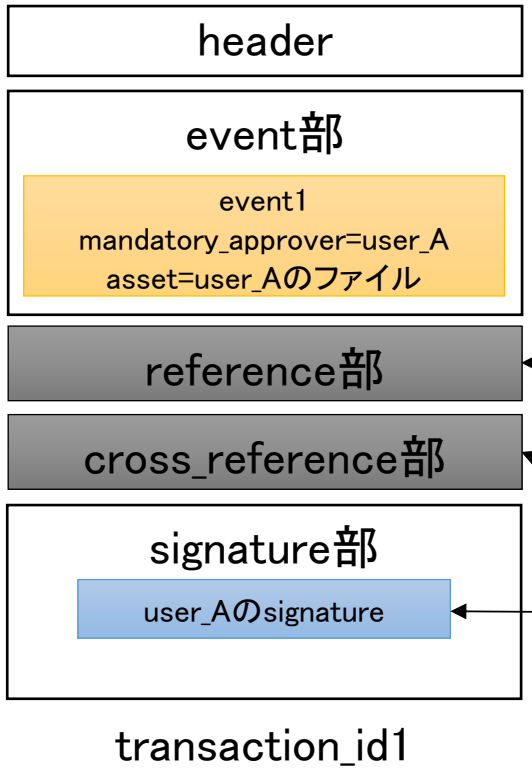
Event/ReferenceとRelation/Witnessの使い分け

- EventとReferenceは「assetを操作する権利が移動したこと」を表すための構造を持っている
  - あるtransactionのEventに含まれているassetは、そのEventでapproverに指定されている人の承認がないとassetを操作できない
  - 操作するときは、次の新しいtransactionの中にReferenceを置き、その中で「approverを指定しているEvent/Transaction」を指定し、署名を付与する
  - つまり、UTXOの構造を意識している
- Relationは、上記に拘らずに、他のasset、transactionと関連付けを行いたいときに利用する
  - どのような関連かはアプリケーションが自由に定義すれば良い
- Witnessは、transactionに何らかの理由で署名を付与する場合に利用する
  - どのような理由で署名を付与するかはアプリケーションが自由に定義すれば良い

トランザクションの事例1(新規ファイル登録)

シナリオ

user_Aがファイルを
新規に登録する



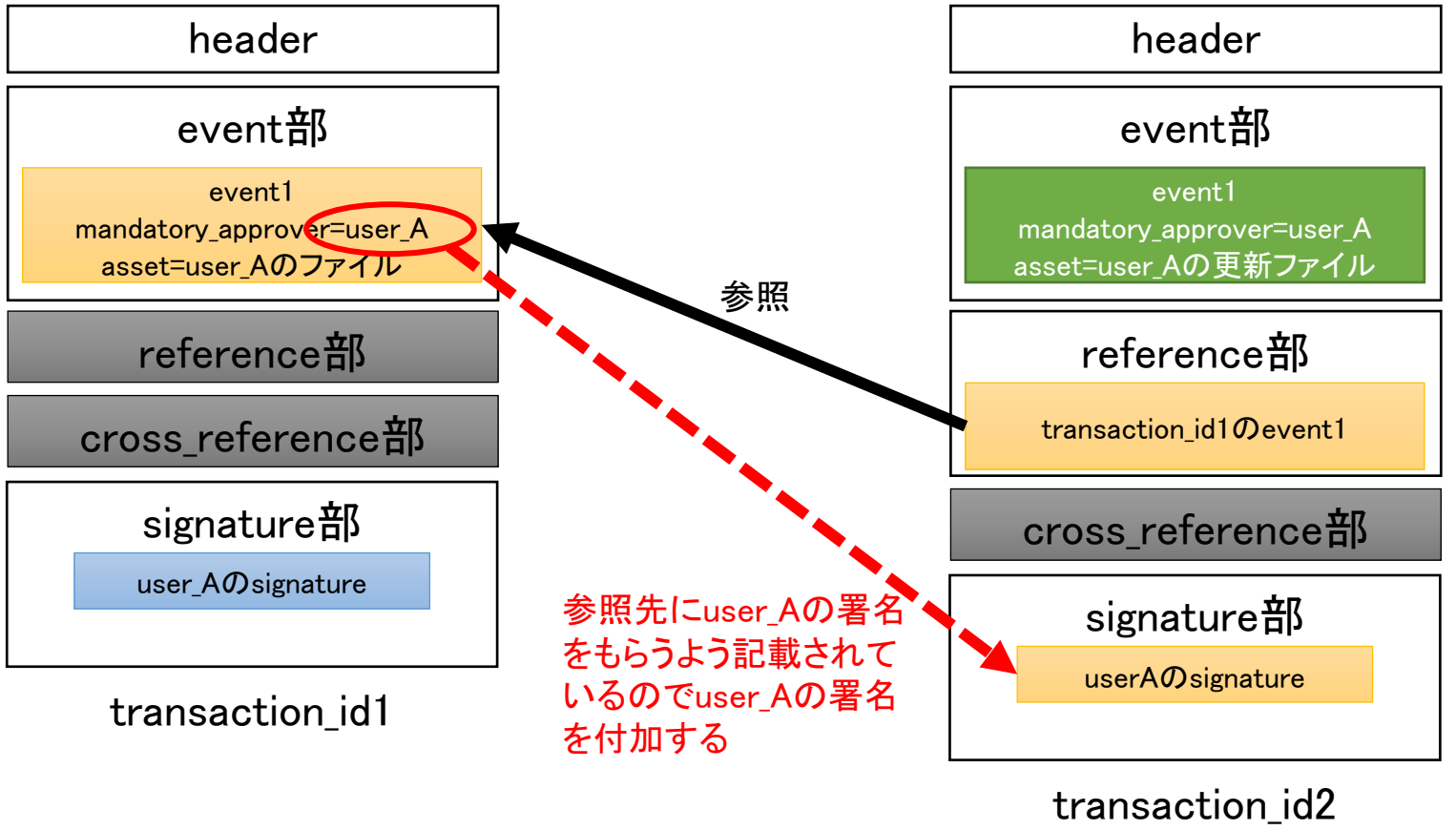
新規登録の場合、過去に参照すべき
transactionは存在しないため不要

簡単のため省略

作成者の署名をつけておく

※ 誰の署名をつけるべきかは各アプリケーションで自由に決めれば良い

トランザクションの事例1(ファイル更新)



シナリオ

user_Aがファイルを更新する

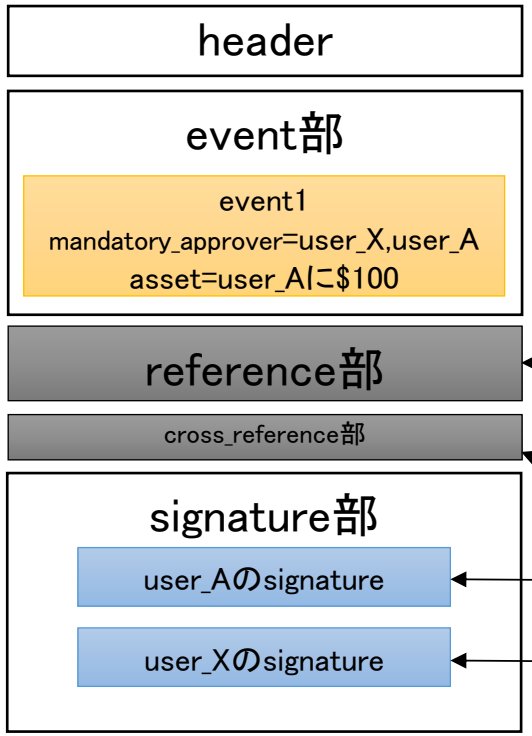
ただし、user_Aは身に覚えのないファイル更新だった場合は、署名を付加してはならない(署名を付与すればそれを認めたことになる)

トランザクションの事例2(仮想通貨付与)

シナリオ

user_Xがuser_Aに通貨を$100発行する

※ user_Xは通貨発行者



新規登録の場合、過去に参照すべきtransactionは存在しないため不要

簡単のため省略

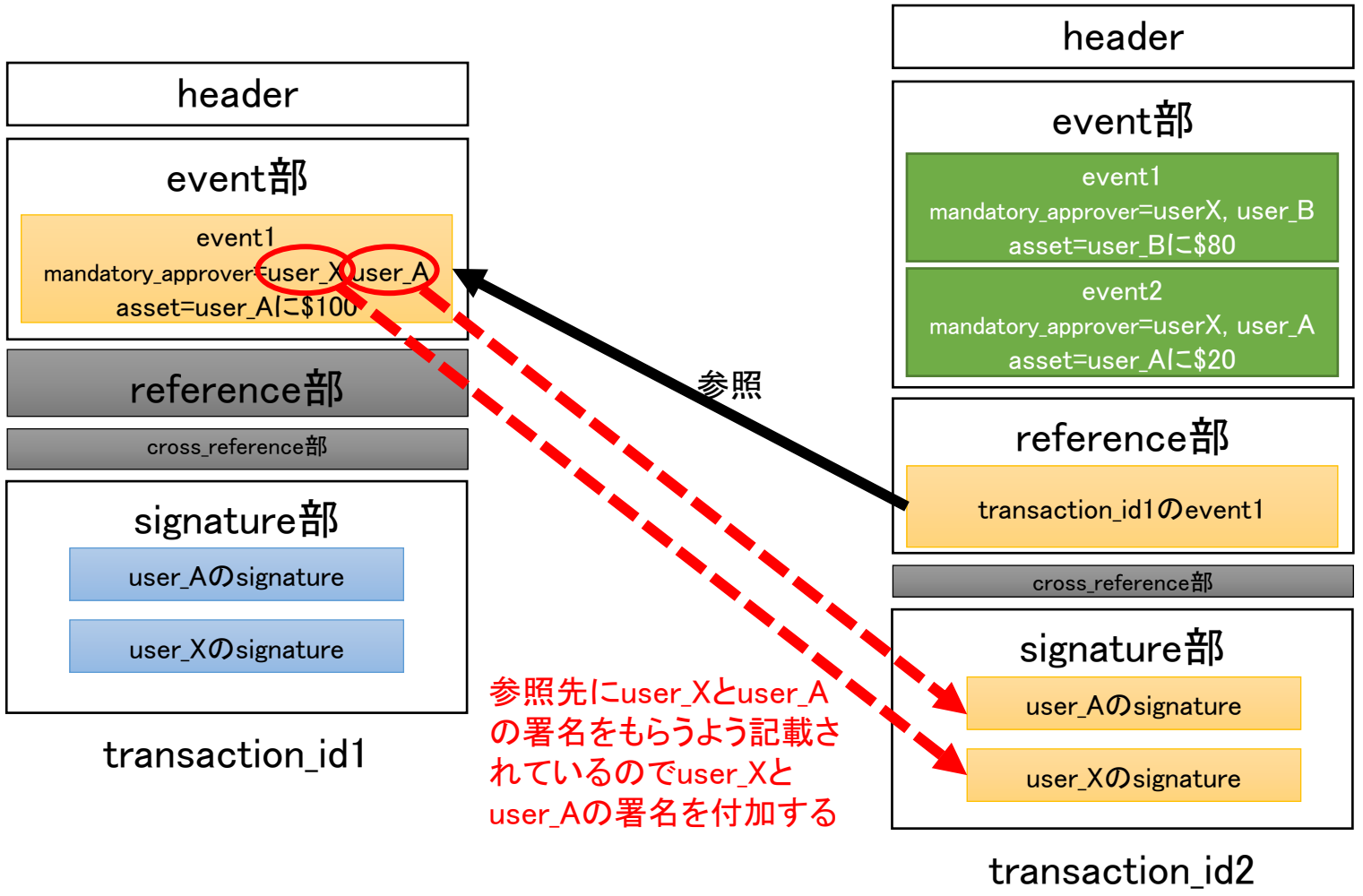
作成者の署名をつけておく

通貨発行者の署名をつける

※ 誰の署名をつけるべきかは各アプリケーションで自由に決めれば良い

transaction_id1

トランザクションの事例2(仮想通貨支払い)

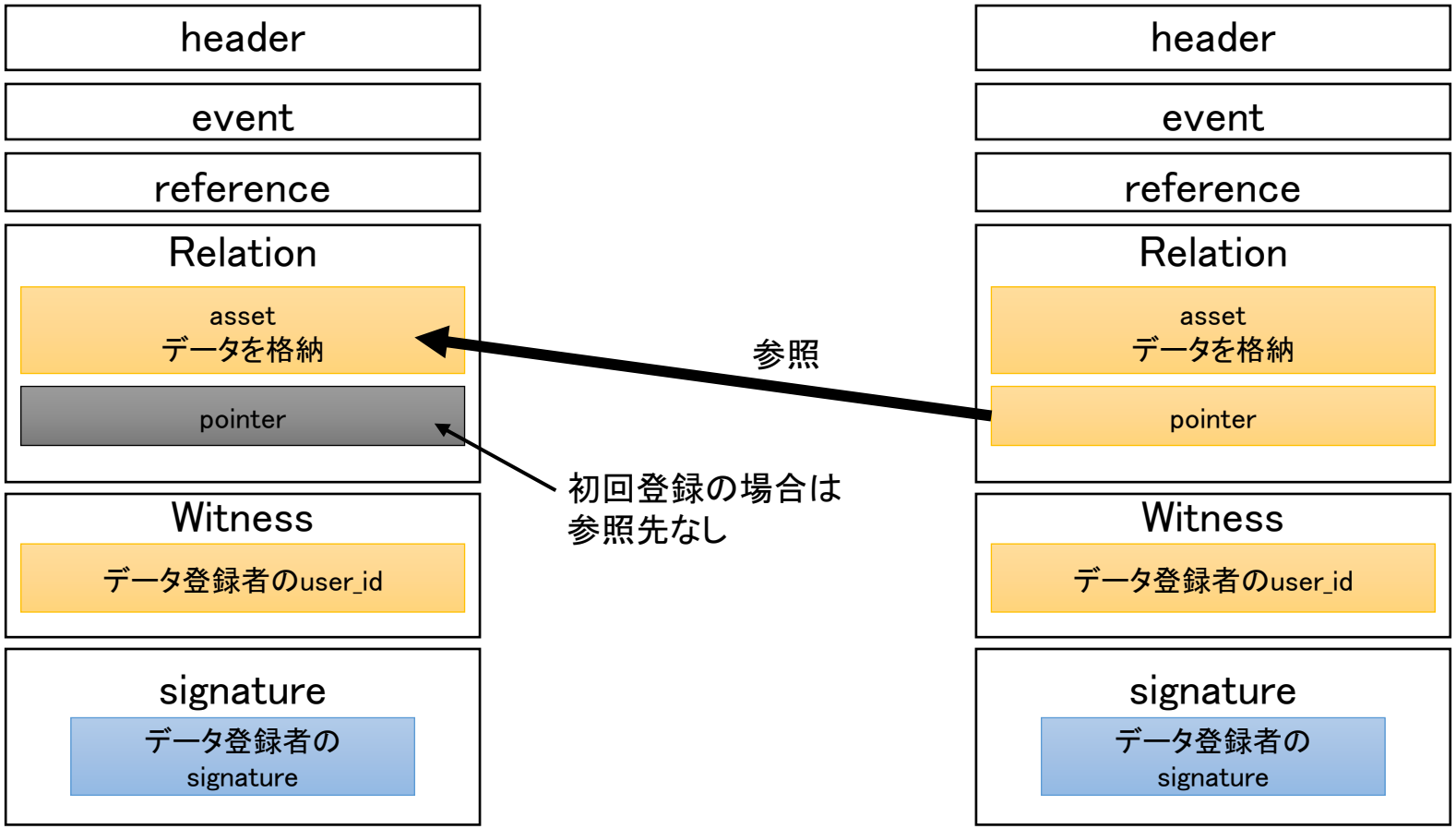


シナリオ

user_Aがuser_Bに通貨を$80支払い、お釣りを$20受け取る

ただし、user_Xは過去のtransaction群をチェックして、仮想通貨の2重消費などの不正が行われていないことを確認しなければならない。確認して問題なければ、user_Xは署名を付加する

トランザクションの事例3(データの格納)



システム設計

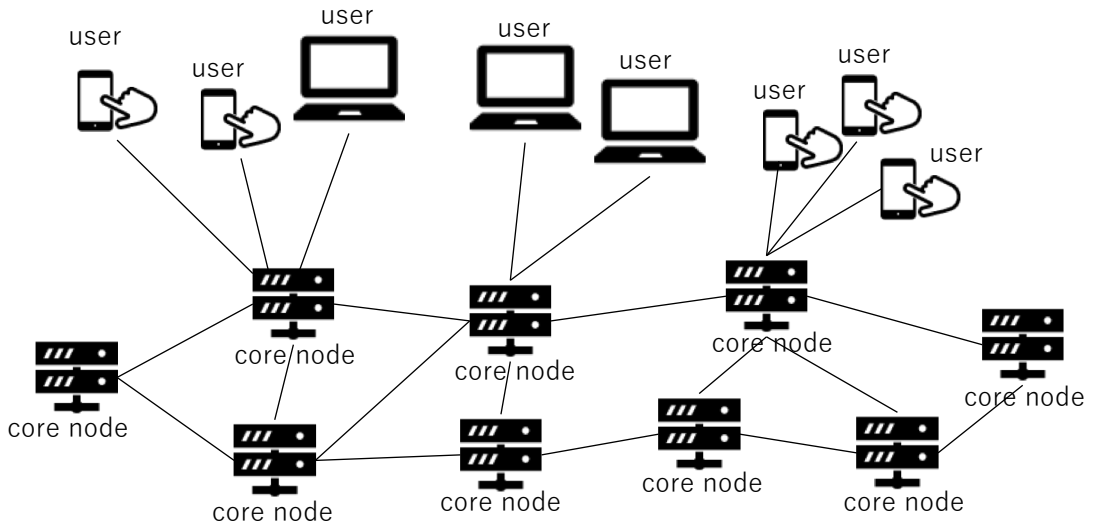
BBc-1システム利用時の流れ

- 登場人物
  - コアノード提供者（≒システム管理者）
  - サービス開発・提供者
  - アプリ利用者

- 流れ
  1. BBc-1の環境を準備する（by コアノード提供者）
  2. コアノード群に対してdomainを定義する（by サービス開発・提供者）
  3. どのようなアセットを取り扱うかを決めて、domain上にasset_groupを登録する（by サービス開発・提供者）
  4. 外部コンピュータ（またはコアノード）にアプリケーションを開発し、コアノードに接続させる（by サービス開発・提供者）
  5. アプリケーションを利用しアセットの登録や移転、検索、取得を行う（アプリ利用者）

典型的なシステム構成

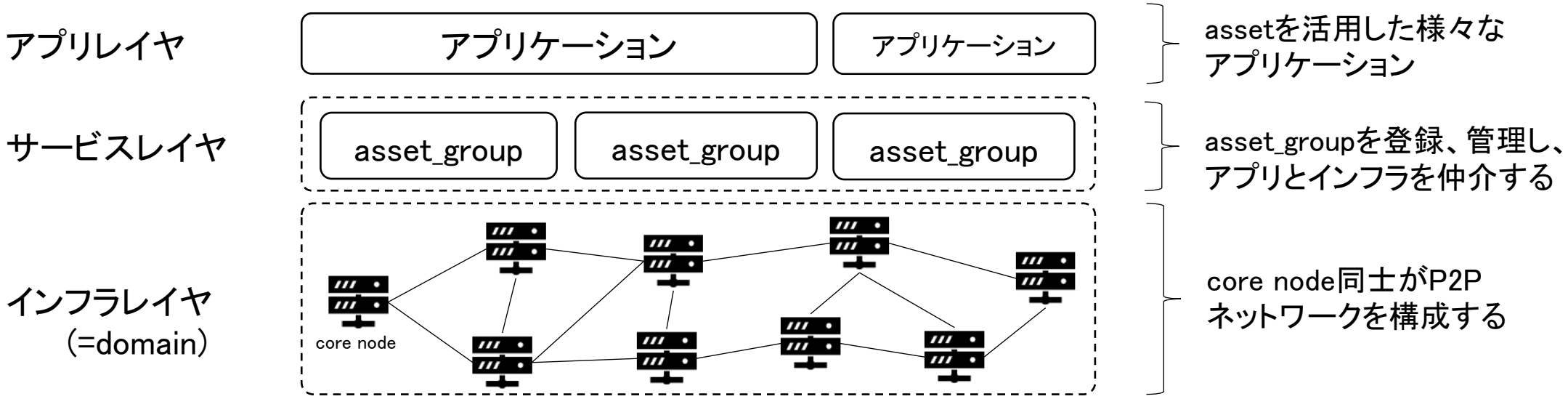
- core nodeは社内、コンソーシアム内、インターネット上に配置される
- userは社内ネットワークやインターネットを通してcore nodeに接続する
  - 指定されたasset_groupを扱うdomain内ならどのnodeに接続しても良い



コアノード提供者が用意した環境を使って、サービス開発・提供者がP2Pネットワークを作る

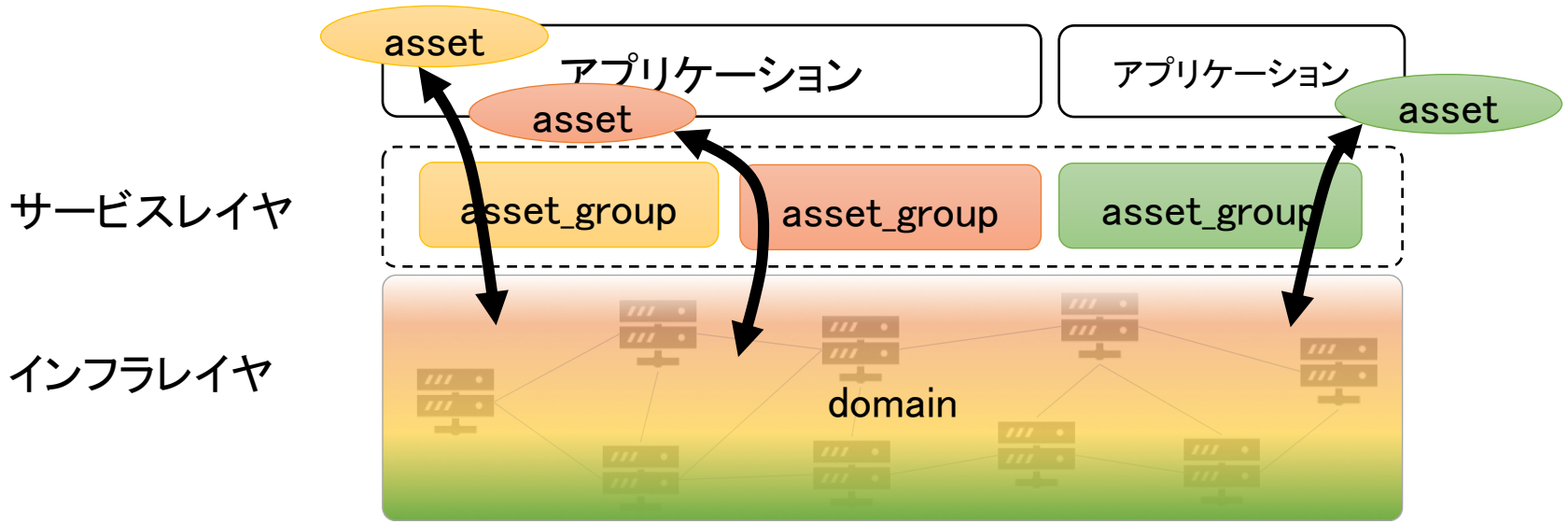
レイヤ構造

- システムは3層のレイヤ構造を持つ



Application/Service layer

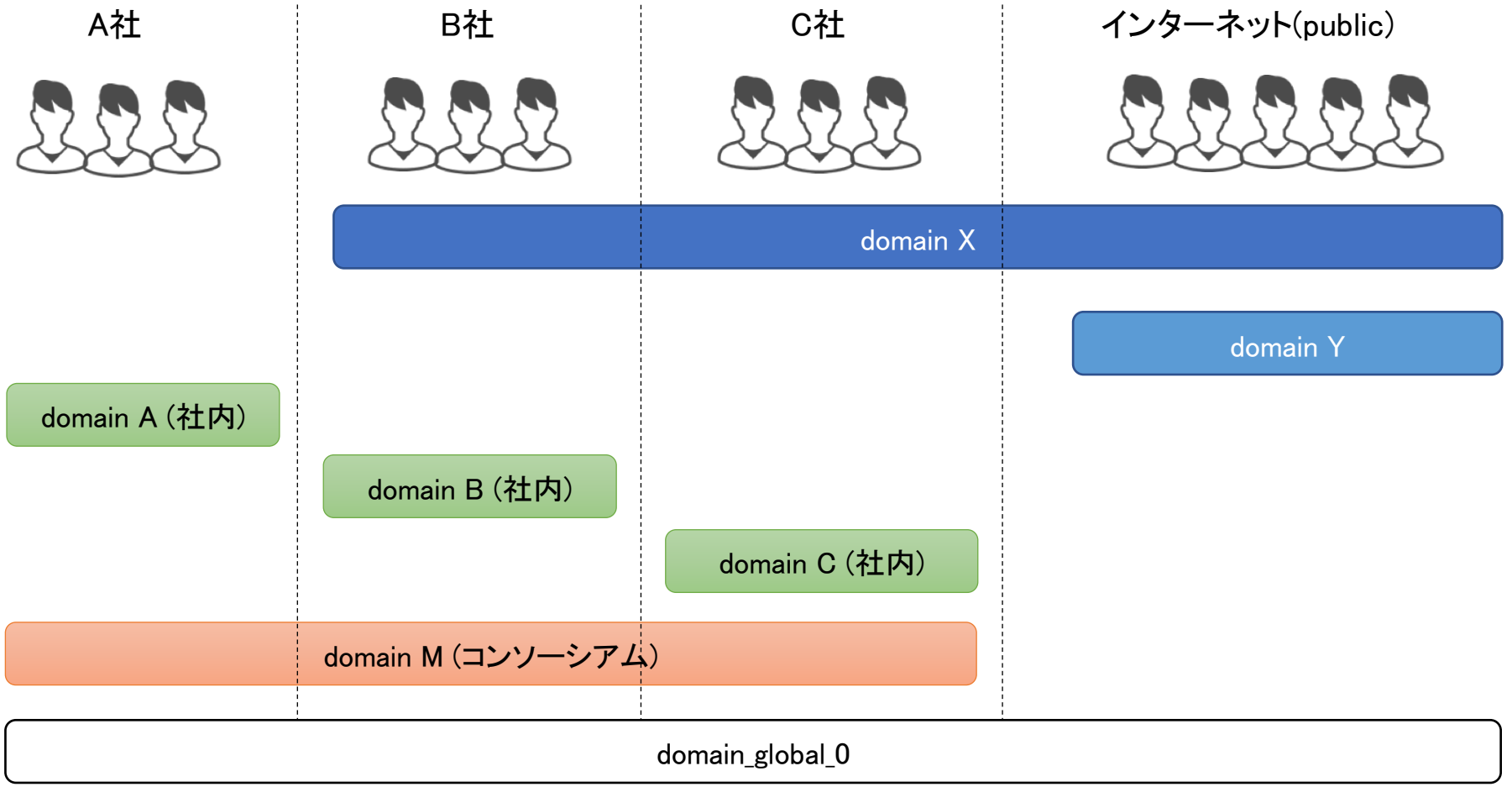
- 1つのdomainの中に、複数のasset_groupを作成できる
  - 各asset_group(assetおよびそれに関連するtransaction)は、domain内の全てまたは一部のcore nodeに自動的に共有される
- アプリケーションは、Service layerを通して様々なassetにアクセスできる



Infrastructure layer

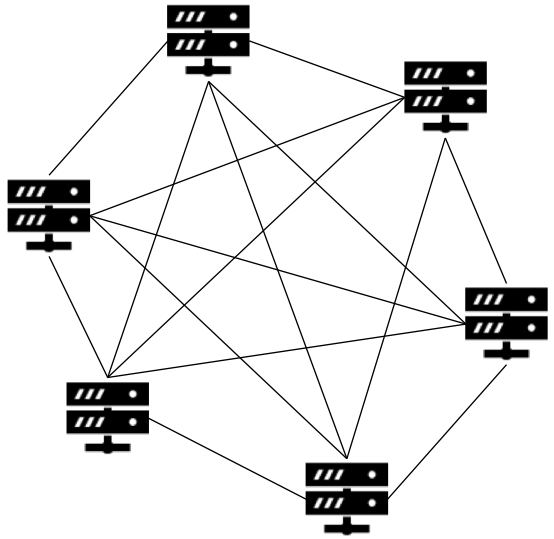
- transaction、assetおよびuser間のメッセージの共有範囲をdomainとび、各core nodeは任意の数のdomainに属することができる
  - 開発者がnodeごとに自由にdomainを設定できる
    - プライベート/コンソーシアム型なら、domainの広がりが狭い
      - 企業内、コンソーシアム内
    - パブリック型なら、domainの広がりが広い
  - グローバルに全てのcore nodeに渡るdomainをデフォルトで1つ定義する
    - domain_global_0 (256bit全て0のIDとする)
    - ただし、必ずしも全てのnodeがそのdomainに所属する必要はない
- domainごとに、どのようなP2Pネットワークを形成するか(トポロジーやそのアルゴリズム)を選択できる

domainのイメージ



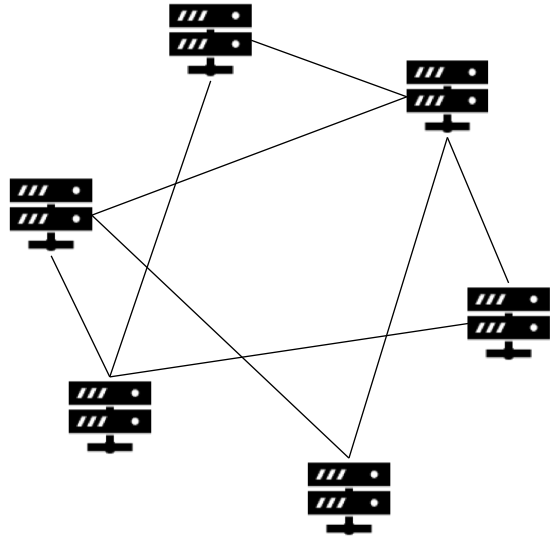
1つのdomainを構成するP2Pネットワーク

- フルメッシュ
  - 全てのcore nodeがお互いを隣接nodeとして認識し、接続し合う



  - メリット：耐障害性が高く、制御が容易
  - デメリット：スケーラビリティが低い

- 効率的トポロジ
  - 適当なnode同士だけが接続し合う
  - 様々な形成アルゴリズムが存在する



  - メリット：スケーラビリティが高い
  - デメリット：耐障害性は多少低く、制御が難しくなる

P2Pネットワークにおける隣接関係

- 隣接ノード
  - 相手のcore nodeのIPアドレスと待受ポート番号を知っていれば、その相手core nodeを隣接ノードと呼ぶ
  - 隣接ノードに対しては、いつでもUDP/TCPにてメッセージを送ることができる
  - P2Pネットワークの全てのcore nodeの隣接関係をトポロジーといい、トポロジーの形成、維持管理のための様々なアルゴリズムが存在する
    - v0.7では、単純なフルメッシュ型トポロジをサポートする
- peer_list
  - 各core nodeは、自分の隣接ノード群をリスト化してpeer_listとして管理する
- static node
  - core node起動直後など、1つも隣接ノードを知らない状態ではP2Pネットワークに参加することが出来ない。強制的にいずれかのcore nodeに接続する（隣接ノードとして設定する）必要がある。このようなノードをstatic nodeと呼ぶ

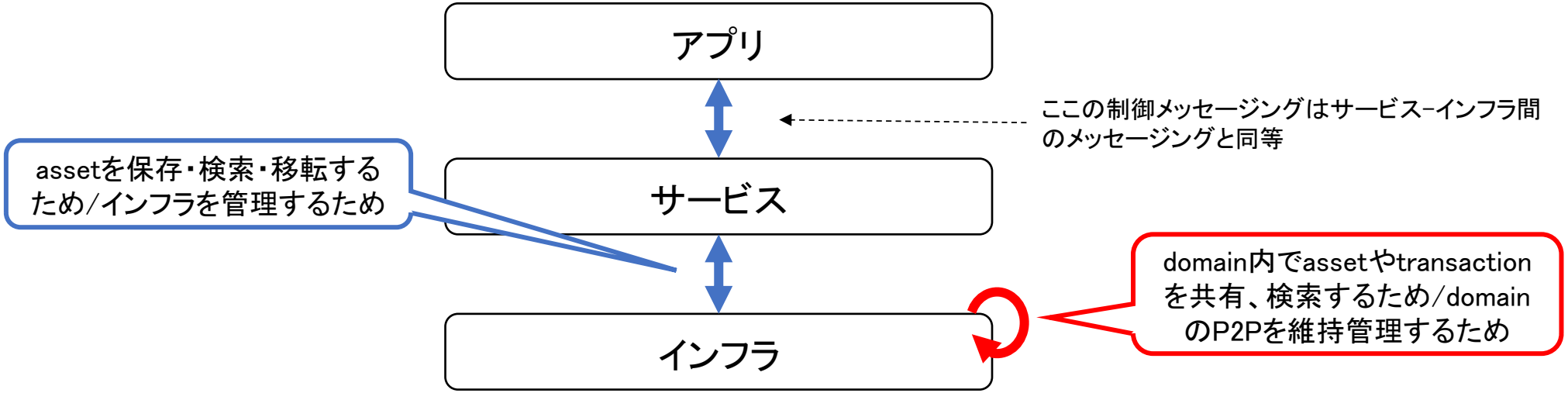
domain_global_0

- 全coreノードが参加することが可能な特別なdomain
  - 完全にプライベート環境でのみ使いたい場合はcore nodeの起動オプションでこのdomainに参加しなくても良い
- 制御用メッセージングにのみ利用する
  - 他のdomainで利用されているasset_groupを発見するため
  - 後述するcross_refの情報をやり取りするため
- このdomain自身にはasset_groupは登録されない

メッセージシーケンス

BBc-1システム内の制御メッセージング

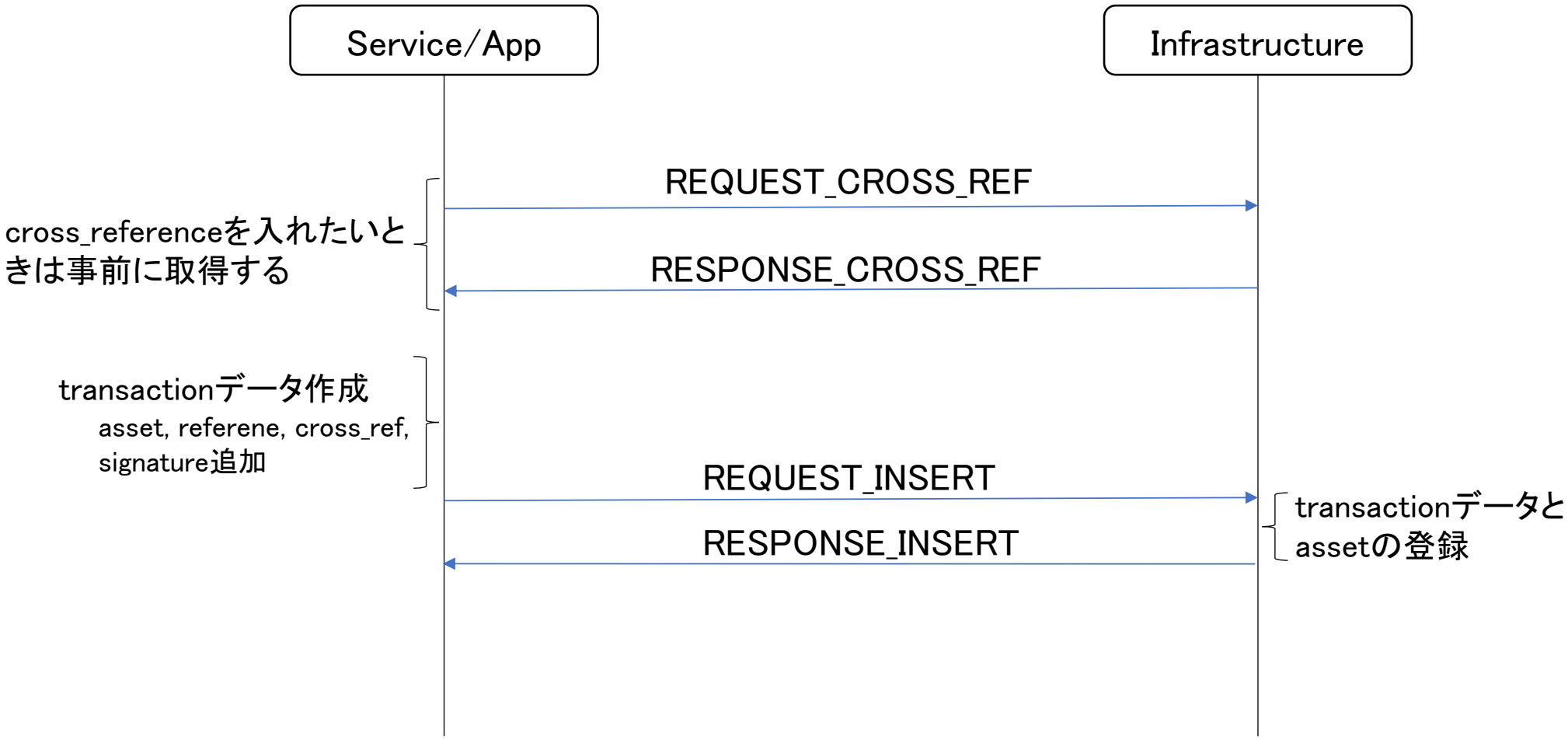
- BBc-1システムの動作を司る主要な制御メッセージング
  - サービスレイヤ-インフラレイヤ間制御メッセージング
    ※ サービスレイヤ-アプリレイヤ間もこのメッセージングと同等
  - インフラレイヤ内制御メッセージング



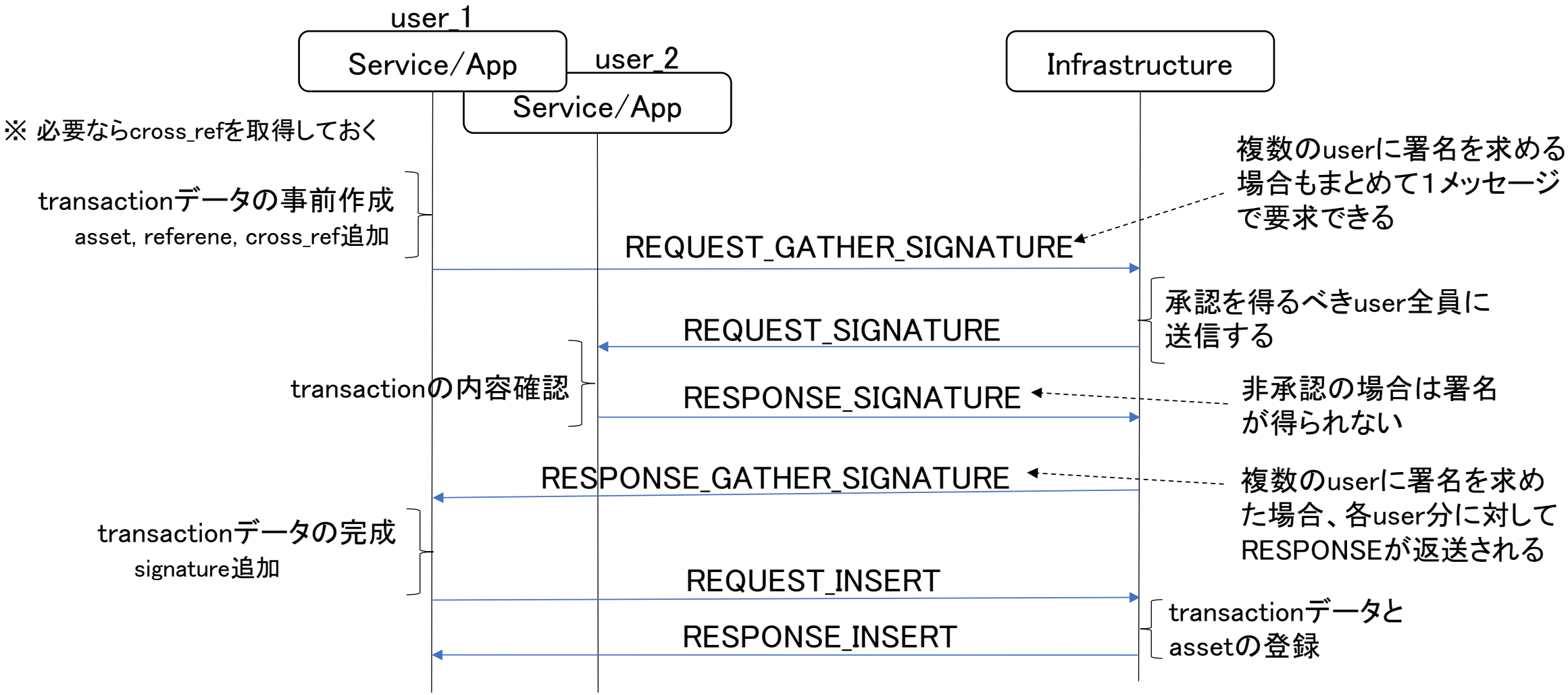
サービスレイヤ–インフラレイヤ間制御メッセージング

- 目的
  - asset活用
    - assetおよびtransactionの登録
    - assetおよびtransactionの検索
  - その他
    - 他userへのメッセージ送信
    - インフラレイヤへの接続・切断
  - システム管理
    - asset_groupの登録
    - domainの定義
    - domainのP2Pネットワークの管理

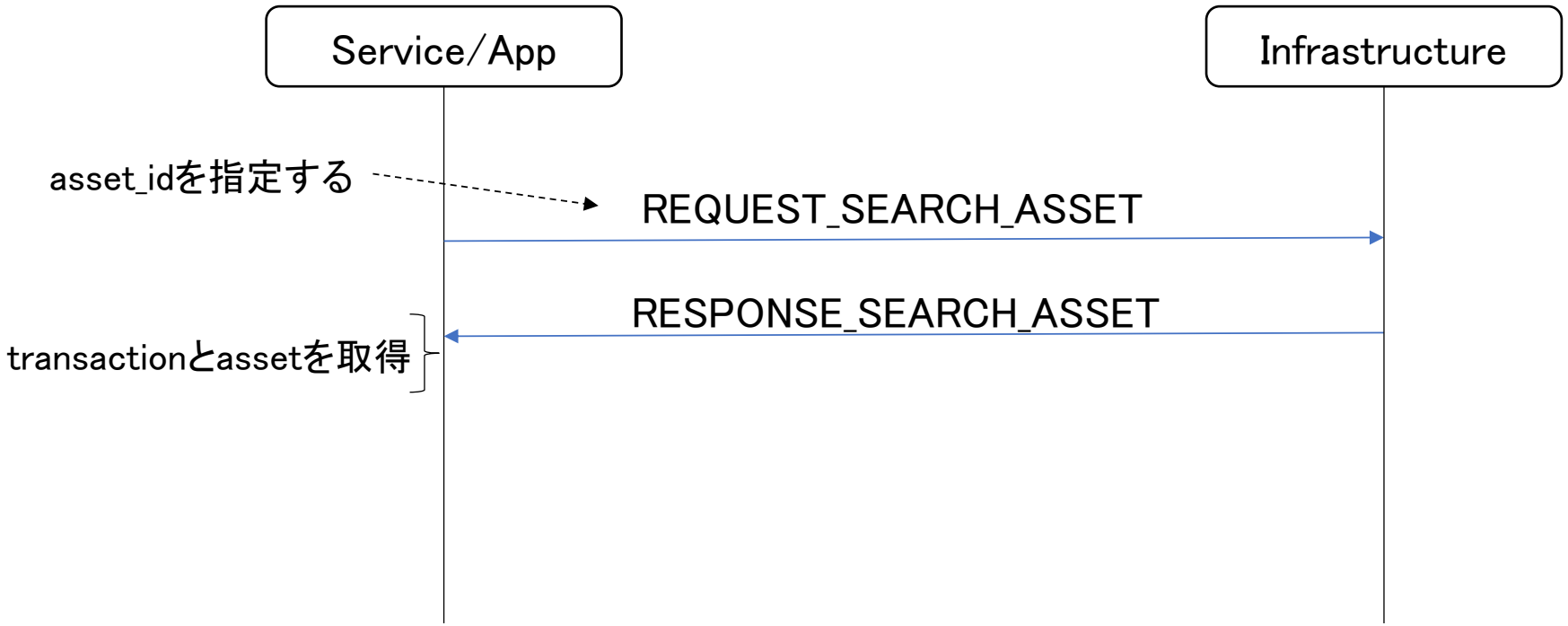
assetとtransactionの登録（署名取得なし）



assetとtransactionの登録（署名取得あり）

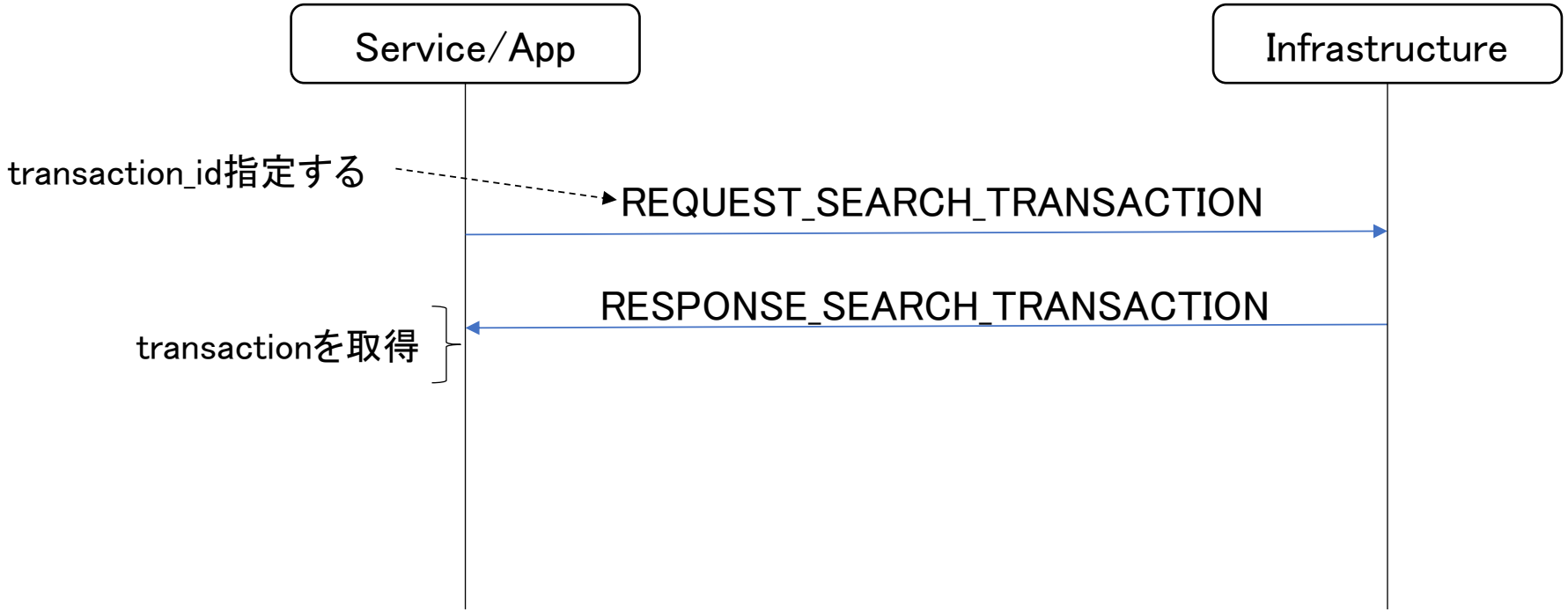


assetの検索



REQUEST_SEARCH_ASSETでは、assetだけでなく、そのassetを登録したときのtransactionも取得できる

transactionの検索



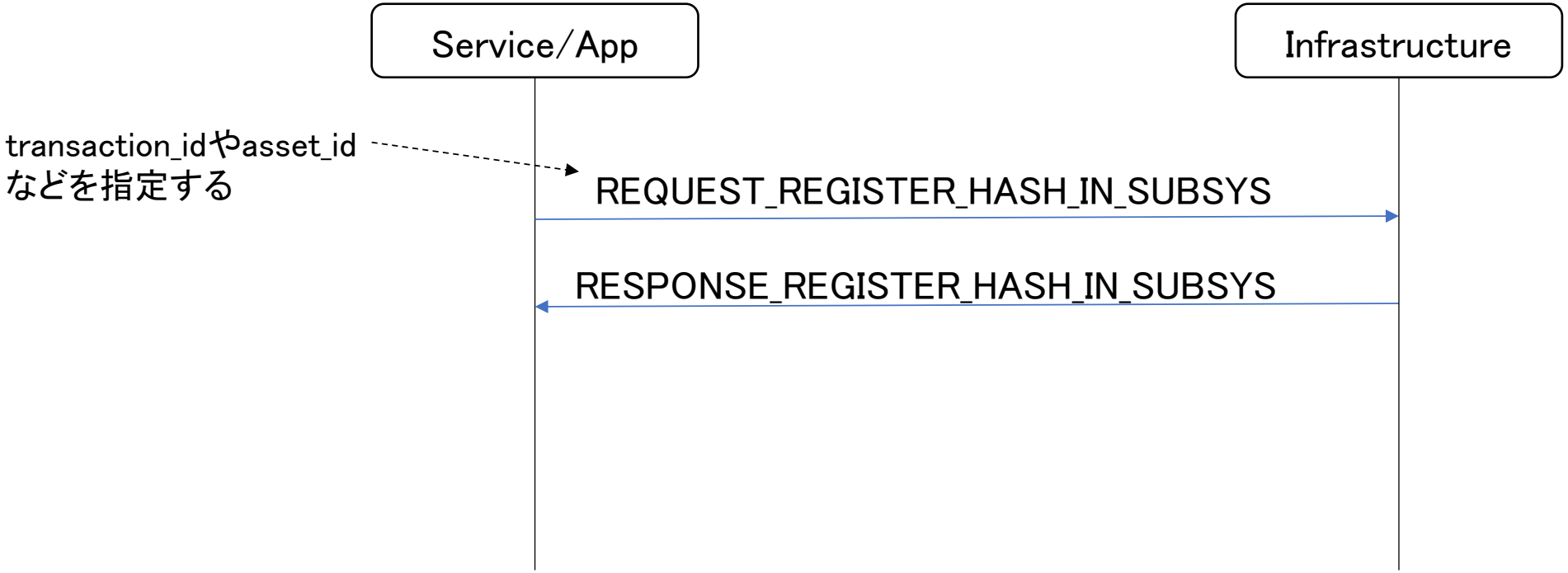
検索の拡張

- v0.7の実装
  - asset_idおよびtransaction_idによる単一の検索のみサポート

- 効率的な検索（将来的にサポート?）
  - 過去の関連transactionを辿るような検索
  - あるtransactionおよびassetについて、それに関する最新のtransactionやassetの検索

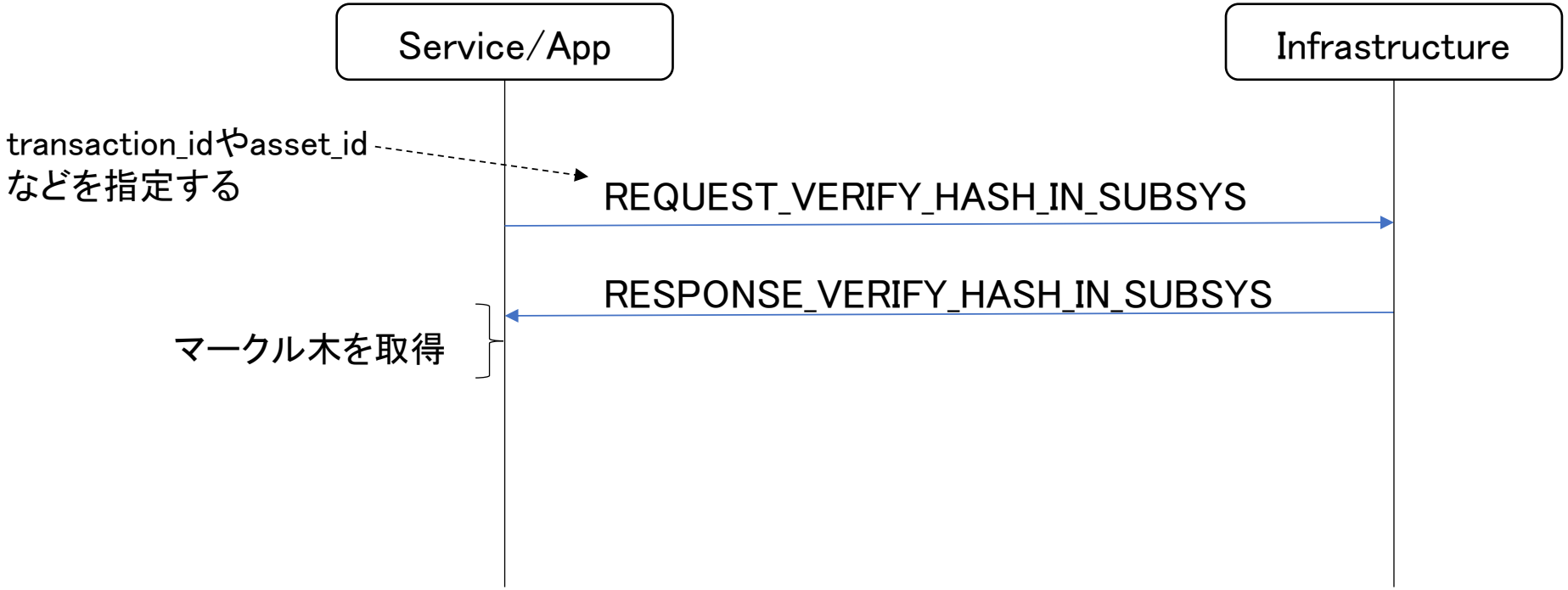
外部ブロックチェーンシステムへの書き込み

- 外部ブロックチェーン（v0.7ではEthereumに対応）には、ハッシュ値を書き込む



外部ブロックチェーンシステムによる検証

- 外部ブロックチェーン（v0.7ではEthereumに対応）に書かれたハッシュ値を確認する



asset活用のための制御メッセージ（まとめ）

メッセージ名	役割
REQUEST_CROSS_REF RESPONSE_CROSS_REF	Cross_referenceを情報を取得する （他者のために存在証明をしてあげる。transactionにCross_refを含めると、自分のtransaction_idが別のdomainに共有され、他者から存在証明してもらえる）
REQUEST_GATHER_SIGNATURE RESPONSE_GATHER_SIGNATURE	他userに承認（署名）を求めるようインフラレイヤに依頼する
REQUEST_SIGNATURE RESPONSE_SIGNATURE	インフラレイヤから、個別にuserに署名を要求する
REQUEST_INSERT RESPONSE_INSERT	transactionを登録する。同時に関連するassetも登録する。
REQUEST_SEARCH_ASSET RESPONSE_SEARCH_ASSET	指定したasset_idのassetおよびそれに関連するtransactionを取得する
REQUEST_SEARCH_TRANSACTION RESPONSE_SEARCH_TRANSACTION	指定したtranaction_idのtransactionを取得する
REQUEST_REGISTER_HASH_IN_SUBSYS RESPONSE_REGISTER_HASH_IN_SUBSYS	外部ブロックチェーンにハッシュ値を登録する
REQUEST_VERIFY_HASH_IN_SUBSYS RESPONSE_VERIFY_HASH_IN_SUBSYS	外部ブロックチェーンにハッシュ値が登録されているかを確認する

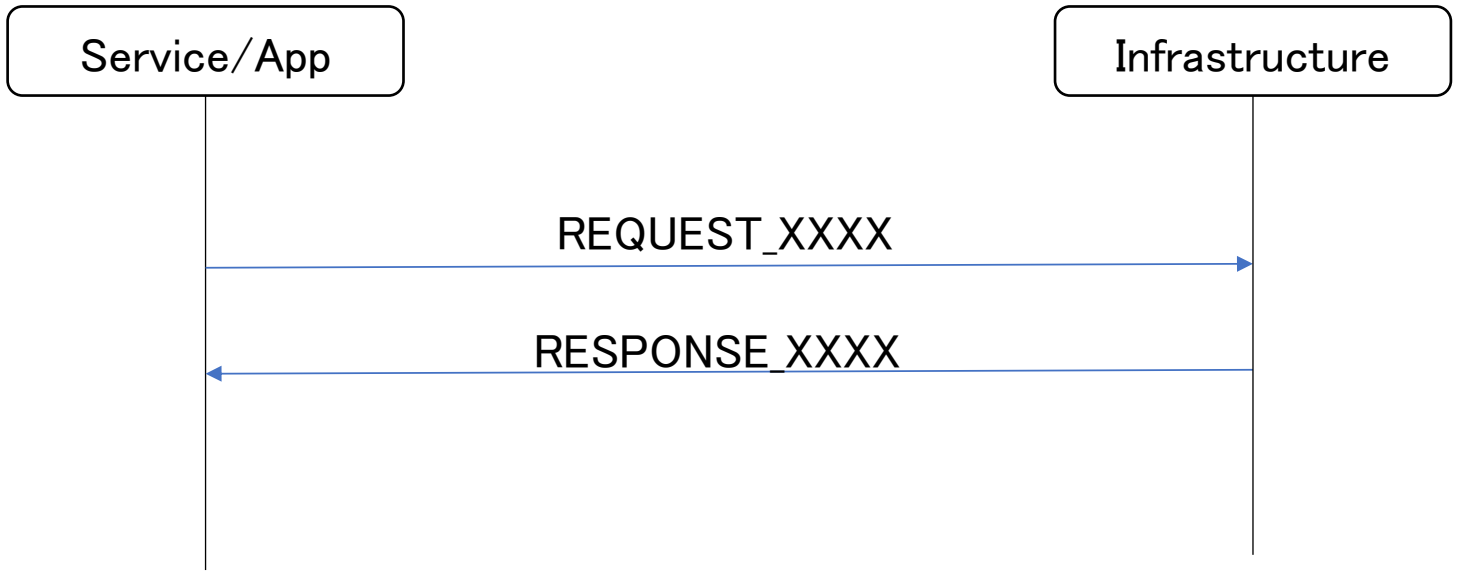
その他の制御メッセージ

- assetやtransactionの制御以外にも、サービス利用開始時の接続切断処理や他userへの任意のメッセージ送信が発生する

メッセージ名	役割
REGISTER UNREGISTER	インフラレイヤに対してuser_idを登録・登録解除を行う。 これにより適切にメッセージを受信できるようになる。
MESSAGE	他userへの任意のメッセージ

システム管理

- システム制御は、必ずサービスレイヤから起動され、単なるREQUEST/RESPONSE型の制御メッセージをやりとりする



RESPONSEが存在しないものもある

システム管理用制御メッセージ

メッセージ名	役割
REQUEST_SETUP_DOMAIN RESPONSE_SETUP_DOMAIN	新しくdomainを設定する（domain_idを登録する） すでに登録済みなら何もしない
REQUEST_GET_DOMAINLIST RESPONSE_GET_DOMAINLIST	そのノードが参加しているdomainのdomain_idのリストを取得する
REQUEST_GET_PEERLIST RESPONSE_GET_PEERLIST	同一domain内で、隣接ノードとして認識しているノードのnode_idやIPアドレス、ポート番号のリスト(Peer_list)を取得する
REQUEST_SET_STATIC_NODE RESPONSE_SET_STATIC_NODE	同一domain内で、初期接続先など強制的に接続する相手を指定する
REQUEST_GET_STATIC_NODES RESPONSE_GET_STATIC_NODES	同一domain内で、初期接続先など強制的に接続する相手（固定接続先）として登録されているノードのリストを取得する
DOMAIN_PING	同一domain内で、指定されたIP/Port番号のノードに通知を送り、相手のnode_idを通知してもらった後、Peer_listに追加する
REQUEST_SETUP_ASSET_GROUP RESPONSE_SETUP_ASSET_GROUP	domain内にasset_groupを登録する
REQUEST_MANIP_LEDGER_SUBSYSTEM RESPONSE_MANIP_LEDGER_SUBSYSTEM	外部ブロックチェーンへの接続機能のon/offを制御する

インフラレイヤ内制御メッセージング

- 目的
  - asset/transaction管理
    - assetおよびtransactionの保存
    - assetおよびtransactionの共有、検索
    - 他userへのメッセージルーティング
  - domain維持、管理
    - domainのトポロジ形成
  - domain_global_0のみの機能
    - Cross referenceの配布
    - asset_groupの広告
- メッセージのやりとり
  - core nodeがdomainごとに制御メッセージのやりとりをする
    - 仮に、全く同じcore node群が2つのdomainを作っていたとしても、それぞれ同様のメッセージのやりとりが各domainについて行われる

asset/transaction管理用制御メッセージ

メッセージ名	役割
REQUEST_STORE RESPONSE_STORE	assetやtransactionなどの登録を要請する
REQUEST_STORE_COPY	assetやtransactionなどの複製を配布する（応答不要）
REQUEST_FIND_USER RESPONSE_FIND_USER	指定したuser_idのuserを検索する
REQUEST_FIND_VALUE RESPONSE_FIND_VALUE	指定したid（asset_idやtransaction_idなど）の情報を検索する
MESSAGE_TO_USER	指定したuser_idのuserへのメッセージ

ドメイン維持・管理用制御メッセージ

メッセージ名	役割
DOMAIN_PING	IPアドレスとポート番号、domain_idだけを指定して、自分の存在を他のcore nodeに通知する。結果として相手のnode_idを取得できるので、後述のREQUEST_PINGが送れるようになる。
NOTIFY_LEAVE	core nodeがdomainから離脱することを同一domainの他のnodeに通知する
NOTIFY_PEERLIST	core nodeが自分自身が持つそのdomainに関するpeer_listを他に通知する
START_TO_REFRESH	トポロジチェック（コネクティビティ確認）プロセスを開始したことをdomain内の他のnodeに通知する。（多量のREQUEST/RESPONSE_PINGが飛ぶため）
REQUEST_PING RESPONSE_PING	domain_idとnode_idを指定して、送信先のcore nodeに自身への接続情報（IPアドレス、待受ポート、node_id）を知らせる。これを受けたnodeは、peer_listに受信した情報を追記する。

ドメインのP2Pネットワークトポロジの維持管理の効率、成否は、NOTIFY_PEERLISTやREQUEST_PINGなどをどのようなタイミングでやり取りするかに依存する。

domain_global_0のみの制御メッセージ

メッセージ名	役割
NOTIFY_CROSS_REF	Cross_reference情報を、他のcore_nodeに通知する。
ADVERTISE_DOMAIN_ID	core nodeが属しているdomain_idを他のcore_nodeに通知する

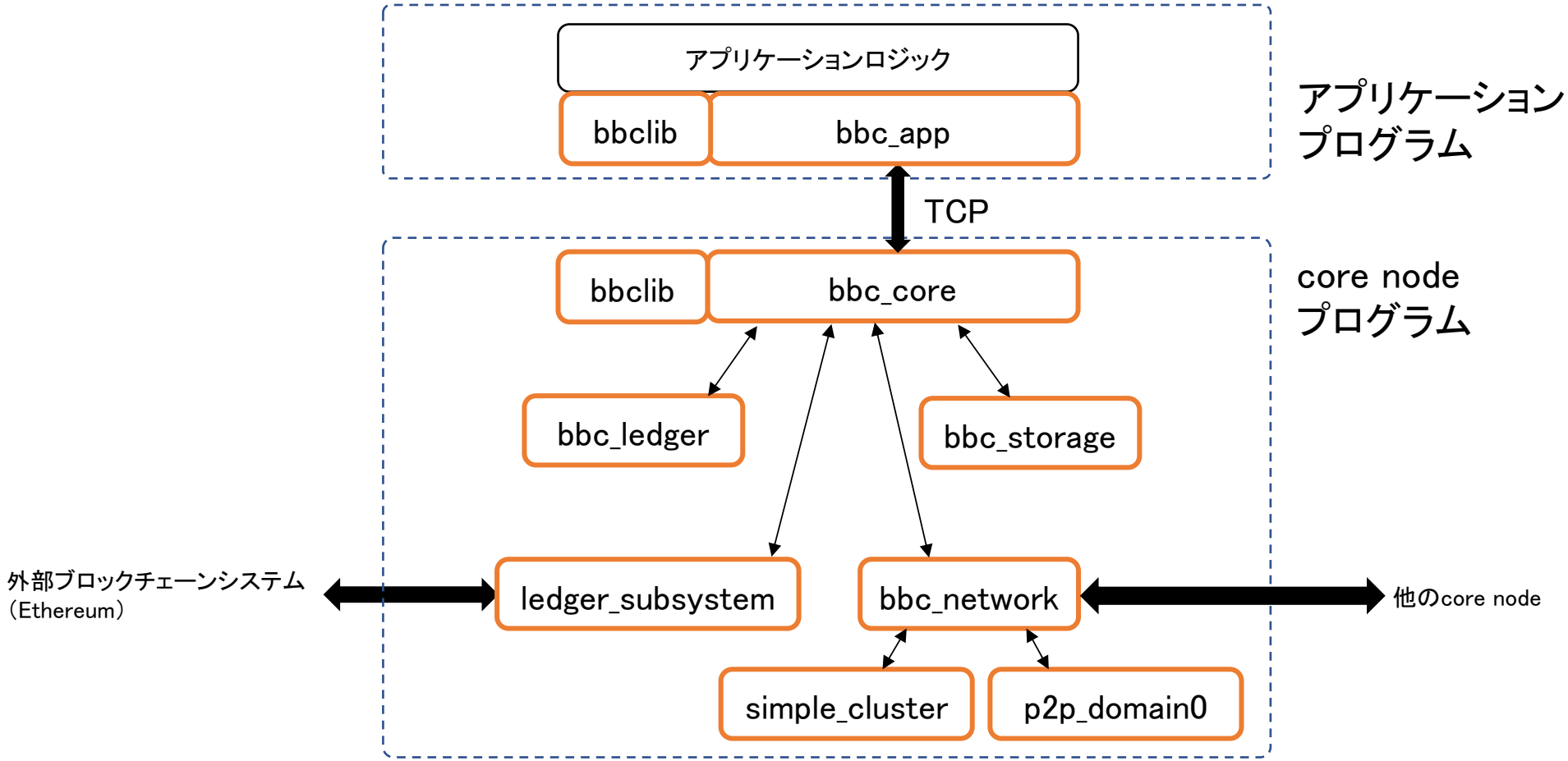
ソフトウェア設計

リファレンス実装

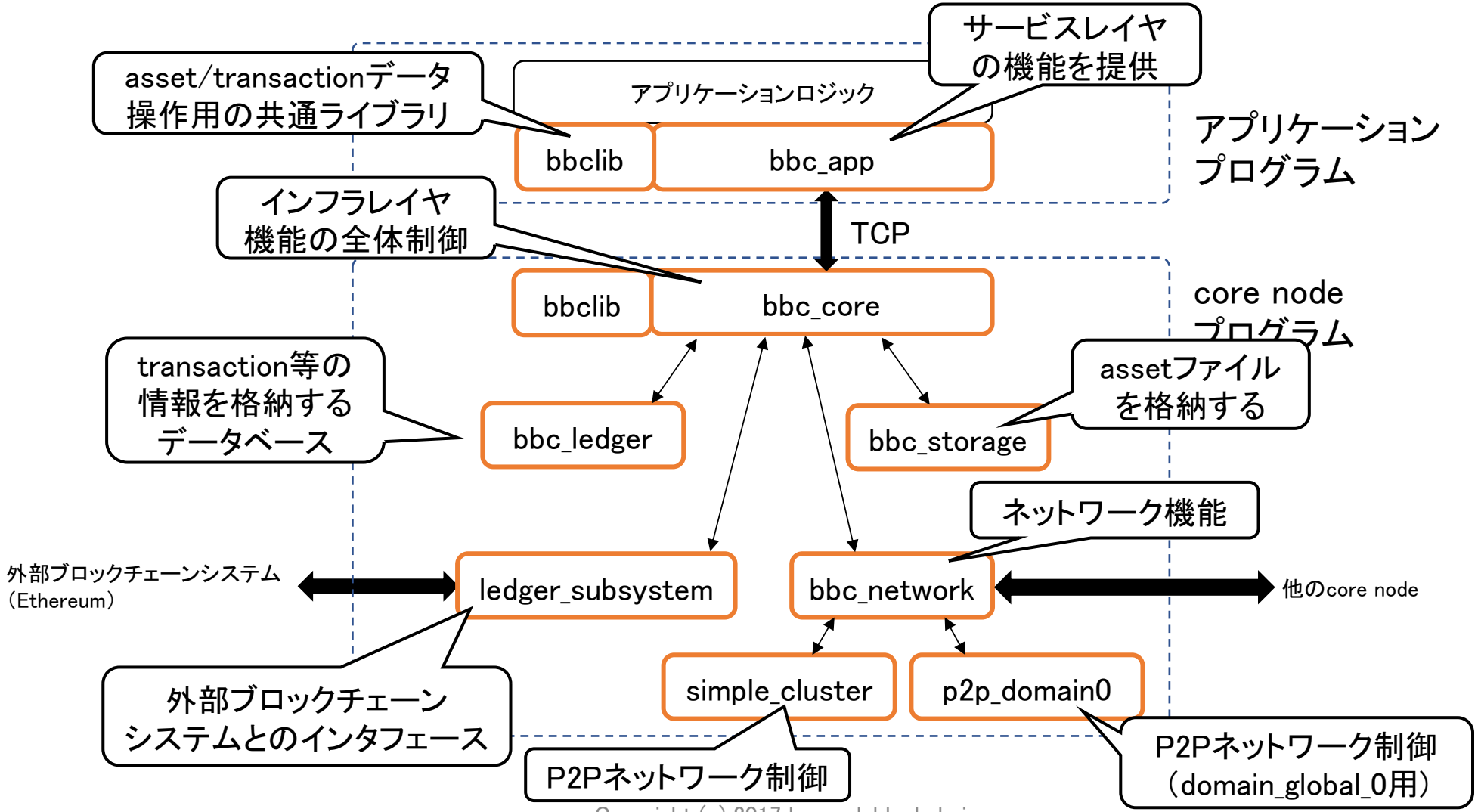
- GitHubに公開済のソースコードはBBc-1のPythonによるリファレンス実装である
  - https://github.com/beyond-blockchain/bbc1
  - 以下に示す要件を満たせば、開発言語や実装方法は自由に選択できる

- BBc-1が満たすべき要件
  - Transactionデータフォーマット（本書p.26～32）
  - サービスレイヤ-インフラレイヤ間制御メッセージングが提供する機能（本書p.54,57）
  - インフラレイヤ内の制御メッセージングが提供する機能（本書p.59,60）
  - domain_global_0の制御メッセージングが提供する機能およびメッセージフォーマット(本書p.61）
    - ただし、v0.7時点では、domain_global_0向けの機能およびフォーマットは完成していない

リファレンス実装の主要機能構成



リファレンス実装の主要機能構成



API仕様

- リファレンス実装のAPI仕様は、sphinxを用いて生成した
  - 参照方法
    - gitリポジトリのdocs/api/_build/html/index.html にブラウザでアクセスする
    - gitリポジトリのdocs/api/ に移動し、下記コマンドでwebサーバを立ち上げ、http://localhost:8000 にアクセスする
      - python -m http.server

アプリケーション/サービスレイヤの実装

- bbc_app.py
  - サービスレイヤ機能を提供するSDK
    - アプリケーションは、BBcAppClientクラスをインスタンス化して機能を利用する
    - 機能の使い方の参考：examples/ディレクトリ配下の2つのサンプルアプリ（file_proofとescrow）
  - bbc_core（インフラレイヤ）との通信を行う
    - デフォルトではbbc_coreがTCP:9000番ポートで待ち受けている
      - メッセージ待受スレッドを起動し、bbc_coreからのメッセージを受信する
    - やり取りするメッセージには、Key-Valueペアのdictionary情報をmessage packでシリアライズしたものを用いている。
      - 参考： BBcAppClient.make_message_structure()メソッド
      - 本実装はリファレンス実装であり、bbc_appとbbc_core間の通信形式は自由に設計して良い

コア制御機能の実装

- bbc_core.py
  - core nodeの起動プログラム
    - python bbc_core.py -h で起動方法のヘルプを表示する
  - core node全体の制御を司る
    - bbc_network、bbc_ledger、bbc_storage、ledger_subsystemなどのインスタンスを作り、動作を開始させる
  - bbc_appとのインタフェースを提供する
    - BBcCoreServiceクラス
      - bbc_appとの通信は前項で示した通り、loopback TCPを用いる
      - メッセージ待受スレッドを起動し、bbc_appからのメッセージを受信する
  - transactionやassetを検索する際に改ざん等を検知すると、そのcore node上の情報を削除して、別のnodeから正しいものを取得し、改ざんから回復させる
    - つまり、全てのnodeの情報を破壊されない限り、復活させることができる

ネットワーク機能の実装

- bbc_network.py
  - 他のcore nodeとの通信インタフェース
    - put/get/route_messageといった基本機能を他のオブジェクトに提供する
    - メッセージ形式はbbc_appとの通信と同様にmessage packを用いる。binary形式のオプションも用意しているがv0.7では未実装
    - メッセージ待受スレッドを起動し、他nodeからのメッセージを受信する
  - domainの作成、削除、domain_ping機能の提供
  - DomainBaseクラス
    - domainごとにP2Pネットワーキングを構成するための機能雛形
      - P2Pネットワーキングのアルゴリズムごとにモジュールファイルを作り、NetworkDomainクラスを継承して機能を実装する。
        - v0.7では、simple_cluster.pyとp2p_domain0.pyが導入されている
      - 作成時にモジュールを指定してインスタンス化する
        - モジュールの指定はファイル名から.pyを除いたものを用いる
        - 参考：BBcNetwork.create_domain() メソッド および simple_cluster.pyのNetworkDomain

P2P機能の実装

- simple_cluster.py
  - フルメッシュ型P2Pネットワーク構成機能
    - network_module名: “simple_cluster”
    - 同一domainの全てのcore nodeを隣接ノードとして認識する
    - あるcore node上に登録されたtransactionおよびassetは、同一domainの全てのnodeに複製が配布される
      - ただし、配布の成否は確認しない

P2P機能の実装（domain0用）

- p2p_domain0.py
  - domain_global_0用のフルメッシュ型P2Pネットワーク構成機能
    - network_module名: “p2p_domain0”
    - 基本機能は、simple_clusterと同等
    - domain_global_0用に、NOTIFY_CROSS_REFとADVERTISE_ASSET_GROUPの機能を実装している

レッジャー機能の実装

- bbc_ledger.py
  - DBへのアクセス機能を提供
    - v0.7では、sqlite3を利用する
    - DBの冗長化（情報の複製）はbbc_networkおよびP2Pネットワーキング機能で実現する
  - DBは2種類利用する
    - transaction_db
      - transactionデータのみを保存する
      - DBファイルは[working_directory]/[domain_id]/[asset_group_id]/bbc_transaction.sqlite3
    - auxiliary_db
      - transactionやassetの検索を効率化するための情報を格納する
      - DBファイルは[working_directory]/[domain_id]/[asset_group_id]/bbc_aux.sqlite3

ストレージ機能の実装

- bbc_storage.py
  - ファイルシステムへのファイル格納・検索機能を提供
    - v0.7では、ローカルのファイルシステムにファイルを書き込む
      - デフォルト設定では、assetファイルは[working_directory]/[domain_id]/[asset_group_id]/storage/ディレクトリの配下にasset_idのファイル名で格納される。
    - configファイルやREQUEST_SETUP_ASSET_GROUPで、storage_path、storage_typeを指定可能
      - storage_path: デフォルト設定のディレクトリ以外にファイルを格納したい場合にパスを指定する。
        - 相対パスならworking_directory以下のディレクトリになる
      - storage_type: StorageTypeクラスの値を指定する。v0.7では、FILESYSTEMとNONEが設定可能
        - FILESYSTEMはローカルファイルシステムへの書き込みを行う
        - NONEが指定された場合は、assetファイルを書き込まない(アプリで管理する)

外部ブロックチェーンとの連携機能の実装

- ledger_subsystem.py
  - 外部ブロックチェーンシステムとのインタフェースを提供
    - v0.7では、Ethereumのtestnetに対応
      - Ethereumにはgethを利用する
  - 外部ブロックチェーンへのハッシュ値の登録
    - メインループでアプリケーションから登録依頼(REQUEST_REGISTER_HASH_IN_SUBSYS)のあったハッシュ値をためておき、それらのハッシュ値からマークル木を作成し、マークルルートを外部ブロックチェーンに書き込む
      - 適当な数だけ集まった時または前回から適当な時間が経過したときに実施
      - asset_groupごとにマークル木を作成する
      - マークル木自体の情報は、ledgerのauxiliary_dbに書き込んでおく
  - 外部ブロックチェーンでのハッシュ値の確認
    - ハッシュ値が登録されているかを問い合わせ(REQUEST_VERIFY_HASH_IN_SUBSYS)、さらにauxiliary_dbの部分木とともに結果を応答する

遅延評価ユーティリティ

- query_management.py
  - タイムアウト付き遅延評価ユーティリティ
  - Tickerクラス
    - クエリーオブジェクトを管理するスケジューラ
    - 期限が来たらクエリーオブジェクトのcallback関数を呼び出す
  - QueryEntryクラス
    - 問い合わせを行った時に遅延評価を実施するための機能を提供するクラス
      - タイマーが満了した時のcallback、問い合わせが成功したときのcallback、問い合わせが失敗したときのcallbackをそれぞれ設定できる
      - その他、任意の情報を付加することができる
    - bbc_network.pyやsimple_cluster.pyの中で、メソッドの引数がquery_entryになっているものは、このQueryEntryオブジェクトを渡している

コンフィグ管理機能の実装

- bbc_config.py
  - BBc-1用のコンフィグを管理するクラス
  - コンフィグファイルへの書き出し、読み込みを行う
    - コンフィグファイルはjson形式で、working_directoryの直下に、config.jsonという名前で保管される
  - domainおよびasset_groupごとの情報を持つ
    - domainやasset_groupの状況が変更されるたびに、configファイルにその時の情報が反映され保存される（途中で再起動してもすぐ復活できるように）

サービス・インフラの共通機能の実装

- bbclib.py
  - transactionおよびassetのデータ操作機能ライブラリ
    - BBcTransaction、BBcEvent、BBcReference、BBcCrossRef、BBcSignatureクラス
      - それぞれtransaction全体、event部、reference部、cross_reference部、signature部に該当し、オブジェクトの作成、シリアライズ、デシリアライズ、verifyなどの機能を併せ持つ
    - KeyPair
      - ECDSAの鍵ペアを作成、管理するユーティリティクラス
        - v0.7では、鍵はX座標、Y座標を両方指定するモード(512bit)に対応する
    - その他、アプリケーション向けのtransaction作成ユーティリティメソッドを含む
      - make_transaction_for_base_asset()
      - add_reference_to_transaction()
      - recover_transaction_object_from_rawdata()
      - recover_signature_object()など
  - ノード情報管理ライブラリ
    - 隣接core nodeの情報(IPアドレス、ポート番号、node_idや更新時刻)を管理する

メッセージフォーマット関連機能の実装

- message_key_types.py
  - rawメッセージのパース機能(Messageクラス)
    - rawメッセージは、ペイロードタイプ(4バイト)、メッセージ長(4バイト)をヘッダとして持つ
    - 参考：Message.parse()
  - KeyTypeクラス
    - リファレンス実装では、node間、およびbbc_app-bbc_core間でやり取り差sれる情報はdictionary型のKey-Valueペアで表現される。そのKeyには、KeyTypeクラスに定義したものが利用される。

以上