

# How BBc-1 works

revision 1

for v1.0

21 May 2018

# 本資料について

- BBc-1の動作仕様についてまとめる
  - 対象のBBc-1はgithubに公開されたv1.0 (2018/5/1バージョン) である
    - <https://github.com/beyond-blockchain/bbc1>
  - BBc-1のWhitePaper、YellowPaper (Analysis)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負わない
- 作成日：2018/5/21
- 作成者：takeshi@quvox.net (t-kubo@zettant.com)

# Collaborators' github account

- junkurihara
- imony
- ks91

# Change log

- 2018/5/21: 初版

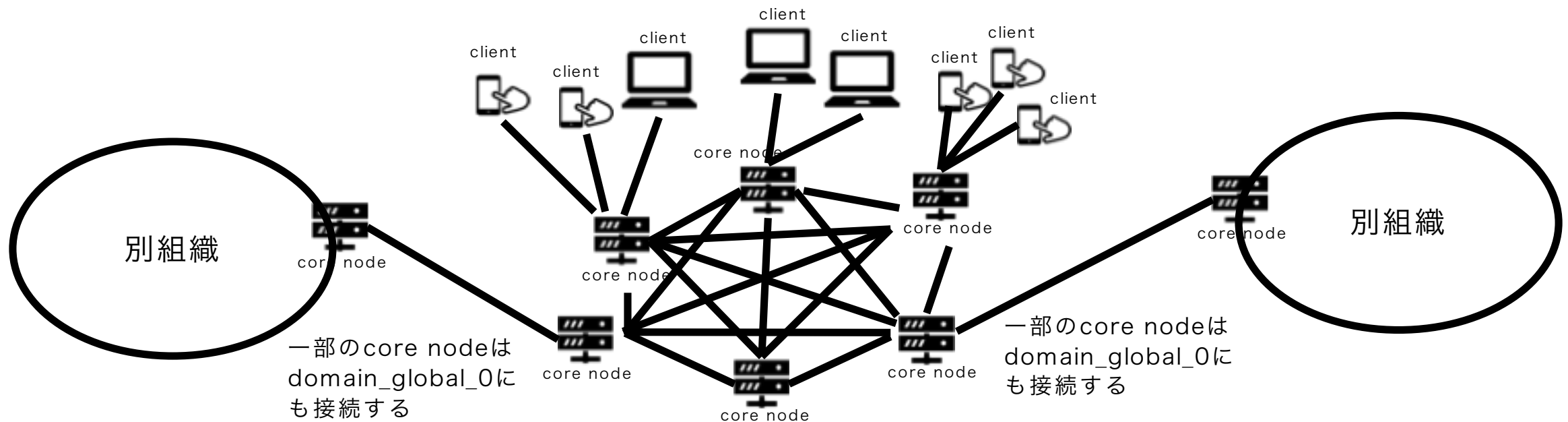
# 目次

タイトル	ページ
通信/セキュリティ仕様	6
ドメインの編成	13
メッセージング (core node-client間)	20
メッセージ仕様 (core node間)	30
データベース	36
ストレージ	44
ダイジェスト計算	46
履歴交差 (Cross_ref) の手順	48
改ざんからの修復	55
プログラム構成	57

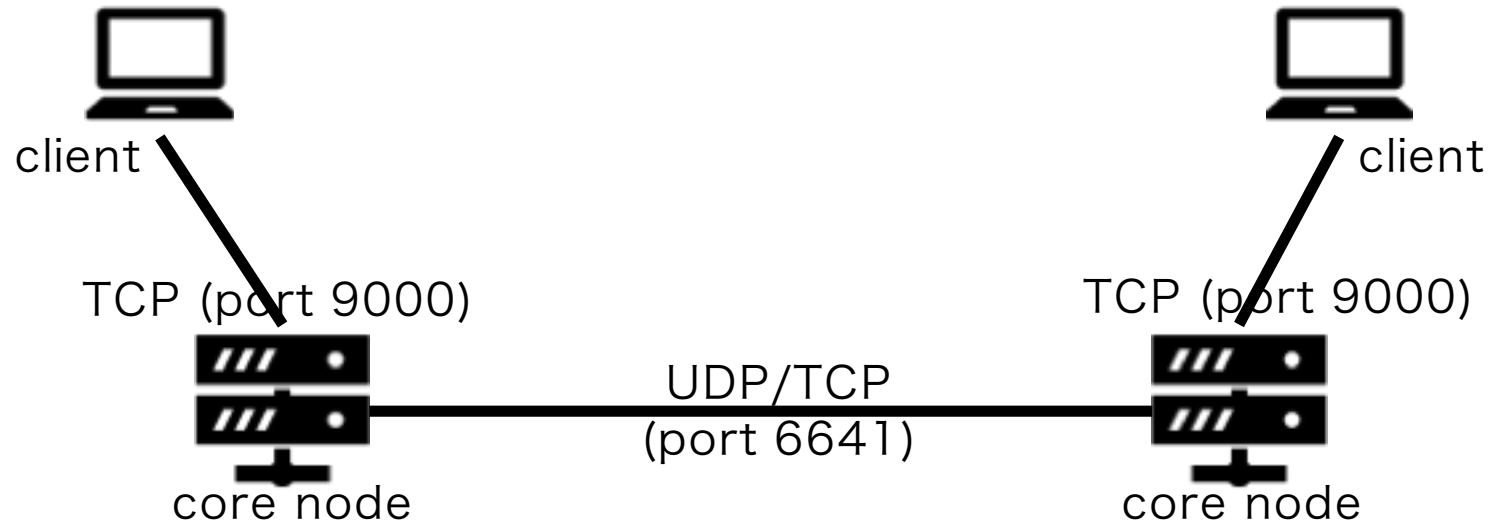
# 通信/セキュリティ仕様

# 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
- client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
- サービス開発・提供者がcore node上にdomainを作りアプリケーション提供する



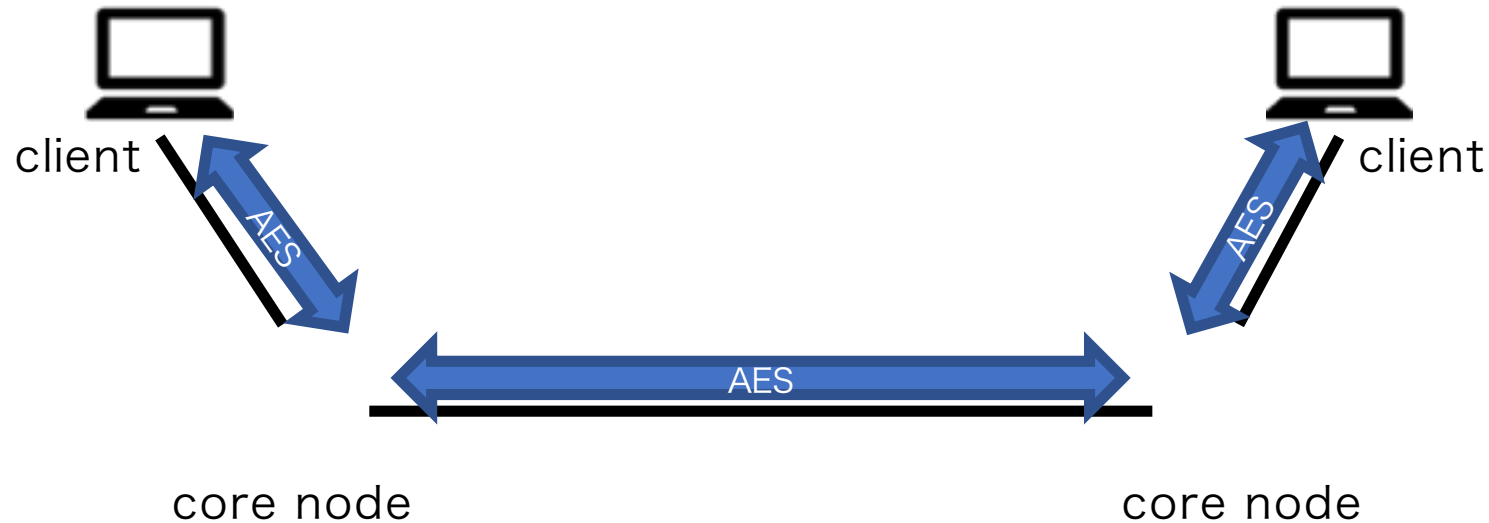
# Layer 4 通信仕様



- core node間の通信は基本はUDPを用いる
  - メッセージサイズが1パケットで収まらない場合はTCPを用いる
- core node-client間の通信はTCPを用いる
- 使用するポート番号は変更可能

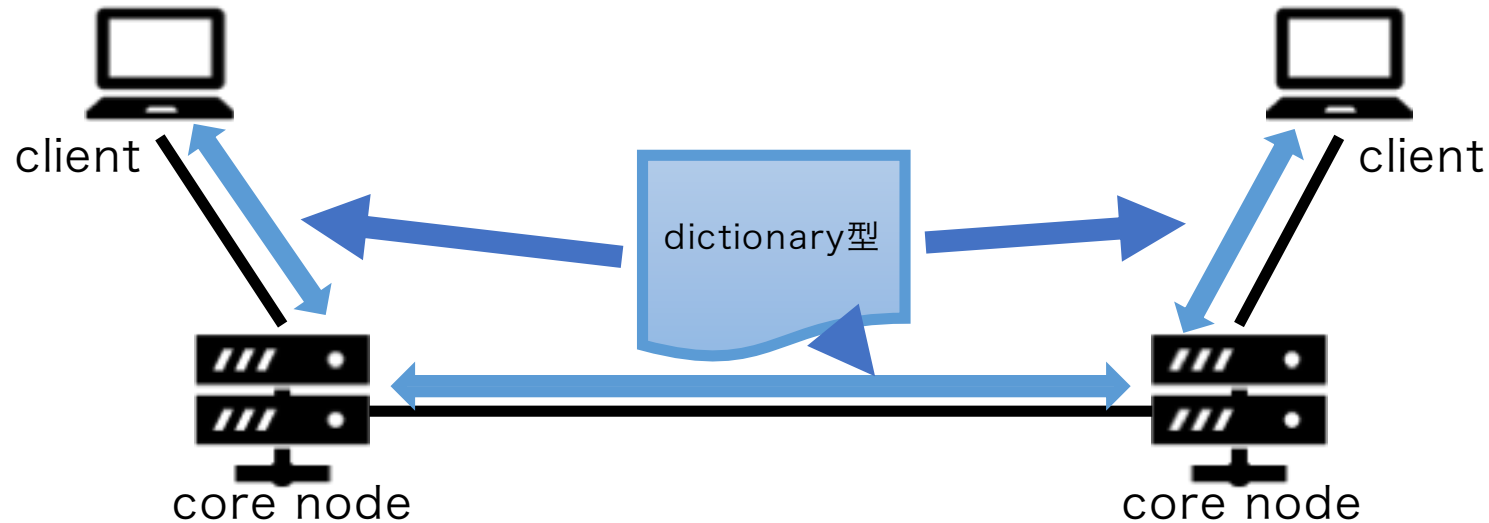


# Application Layer通信仕様 (暗号化)



- core node間およびcore node-client間は1対1で鍵を共有し、メッセージをAES暗号化する
  - 暗号方式：EC\_SECP384R1
  - ECDH (Elliptic Curve Diffie-Hellman)方式による鍵交換
  - 鍵はドメインごとにフルメッシュで交換される（同一のcore node間でもドメインが違えば別途鍵を生成し、共有する）
- 暗号化しなくても良い
  - 起動直後の鍵交換完了前など、平文でのメッセージ交換も実施している

# Application Layer通信仕様 (serialize)



- core node間およびcore node-client間でやり取りされるメッセージは、pythonのディクショナリ形式とする
  - 実際にネットワークでやり取りされるメッセージは、msgpackまたは独自方式 (Type-Length-Value型)でシリアライズする

# Clientとcore node間のセキュリティ

- ドメイン作成や統計情報取得など、システム管理者が利用するコマンドについては、clientはnode\_keyを用いてメッセージに署名(ECDSA)を付加しなければならない

送信したい情報

```
msg = {  
  KeyType.source_user_id: src_id,  
  KeyType.command: AAA,  
  KeyType.xxx: XXX,  
  KeyType.yyy: YYY,  
  KeyType.zzz: ZZZ,  
}
```

実際に送信される情報

```
msg = {  
  KeyType.source_user_id: src_id,  
  KeyType.command: AAA,  
  KeyType.admin_info: *****,  
  KeyType.nodekey_signature:  
    ###,  
}
```

admin情報をシリアライズ

シリアライズされた  
データをnode\_keyで署名

※ admin情報扱いになるかどうかは、KeyType.commandの値による。  
bbc\_core.pyのadmin\_message\_commandsに定義されている。

# core node間のセキュリティ

- core node同士がやり取りするメッセージの内、domain\_pingと鍵交換のメッセージについては、domain\_keyを用いてメッセージに署名(ECDSA)を付加しなければならない

送信したい情報

```
msg = {  
  KeyType.source_user_id: src_id,  
  KeyType.command: AAA,  
  KeyType.xxx: XXX,  
  KeyType.yyy: YYY,  
  KeyType.zzz: ZZZ,  
}
```

情報をシリアル化

実際に送信される情報

```
msg = {  
  KeyType.source_user_id: src_id,  
  KeyType.command: AAA,  
  KeyType.admin_info: *****,  
  KeyType.nodekey_signature:  
    ###,  
}
```

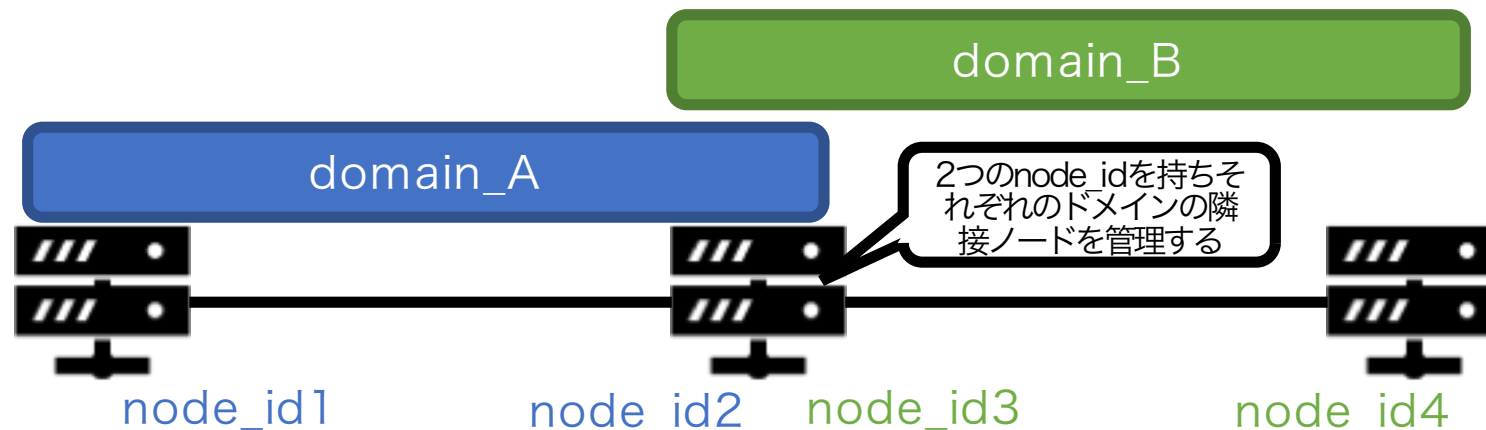
シリアル化された  
データをnode\_keyで署名

※ 処理方法はclient-core間(前ページ)と同様

# ドメインの編成

# ドメイン

- ドメインは、同一ポリシーの運営主体が作成し、管理する
  - トランザクションやアセットはドメインに紐付き、その中でのみ複製が配布され、アクセス可能である
- core nodeは複数のドメインを同時に収容できる
  - 1つのcore nodeは、ドメインごとに異なるnode\_idを持ち、隣接ノードを認識し、フルメッシュトポロジを構成する



# 隣接ノード

- core nodeは、所属する各ドメインについて、ドメイン内の全てのcore nodeを認識し、隣接ノードとしてリスト管理する
  - ドメイン内では全てのcore node同士が互いを認識するため、フルメッシュトポロジーのネットワーク構成になる
    - 将来的には、大規模なシステムになると、KademliaなどのP2Pトポロジーが必要になる
- 相手のcore nodeを「認識する」とは、相手のcore nodeのIPv4アドレス、IPv6アドレス、ポート番号を知ることである
  - IPv4とIPv6はいずれか一方がわかれば良い (IP reachabilityが得られれば良い)
  - 相手の認識方法
    - 何らかのメッセージのやりとりをする → `bbc_ping`を利用する
    - コマンドラインからIPやポート番号を設定する → `set_domain_static_node`メソッド
- 隣接ノードリストの広告
  - core nodeは自分が管理する隣接ノードリストに変更があれば、隣接ノードリスト自体を隣接ノード全てに広告する
    - これにより、即座にフルメッシュトポロジーが形成できる

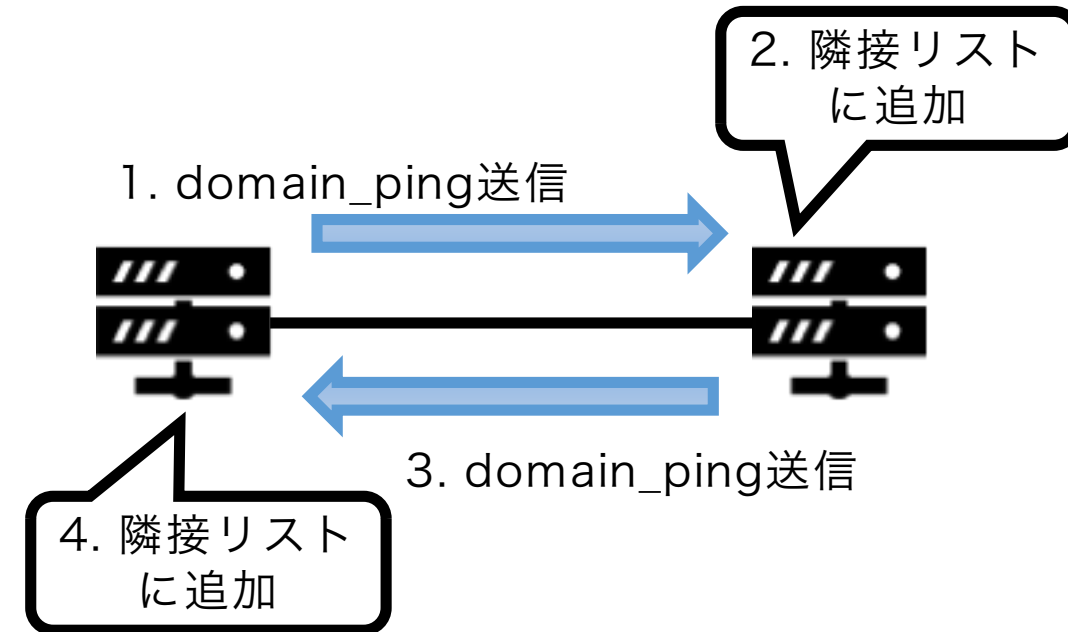
# 隣接ノード情報

- 1件の隣接ノード情報は以下の項目からなる
  - node\_id
  - IPv4アドレス
  - IPv6アドレス
  - port番号
  - メッセージシーケンス番号
    - その相手から重複して受信したメッセージを破棄するため
  - セキュリティステータス
    - core node間のAES暗号化の有無
  - staticフラグ
    - staticエントリとしてconfig.jsonに書き出すかどうか
  - ドメイン0フラグ
    - 履歴交差を扱うためにdomain\_global\_0に接続しているかどうか
  - エントリ更新時刻
    - staticエントリでない場合、一定時間エントリが更新されなければ隣接ノードリストから削除される



# 隣接ノードの設定

- 隣接ノード広告
  - 隣接ノード広告を受け取った場合そこに記載されているすべての情報を自身の隣接ノードリストにコピーする
  - 広告は各core nodeが一定時間ごとにドメイン内にブロードキャストする (設定値は300秒)
- bbc\_pingを用いる場合
  - domain\_ping (request)を受信すると、送信元情報を隣接ノードリストに加える
  - 受信側は自動的にdomain\_ping (reply)を返送する
  - domain\_pingのペイロードには、送信元のIPアドレスとポート番号が記載されており、中間にNATがあるかどうかを確認できる
- set\_domain\_static\_nodeメソッドを用いる場合
  - 隣接リストに直接情報が書き込まれる
    - その情報にはstaticフラグが付与される



config.jsonにその隣接ノード情報が書き出され、起動時に自動的に隣接ノードとして認識されるようになる

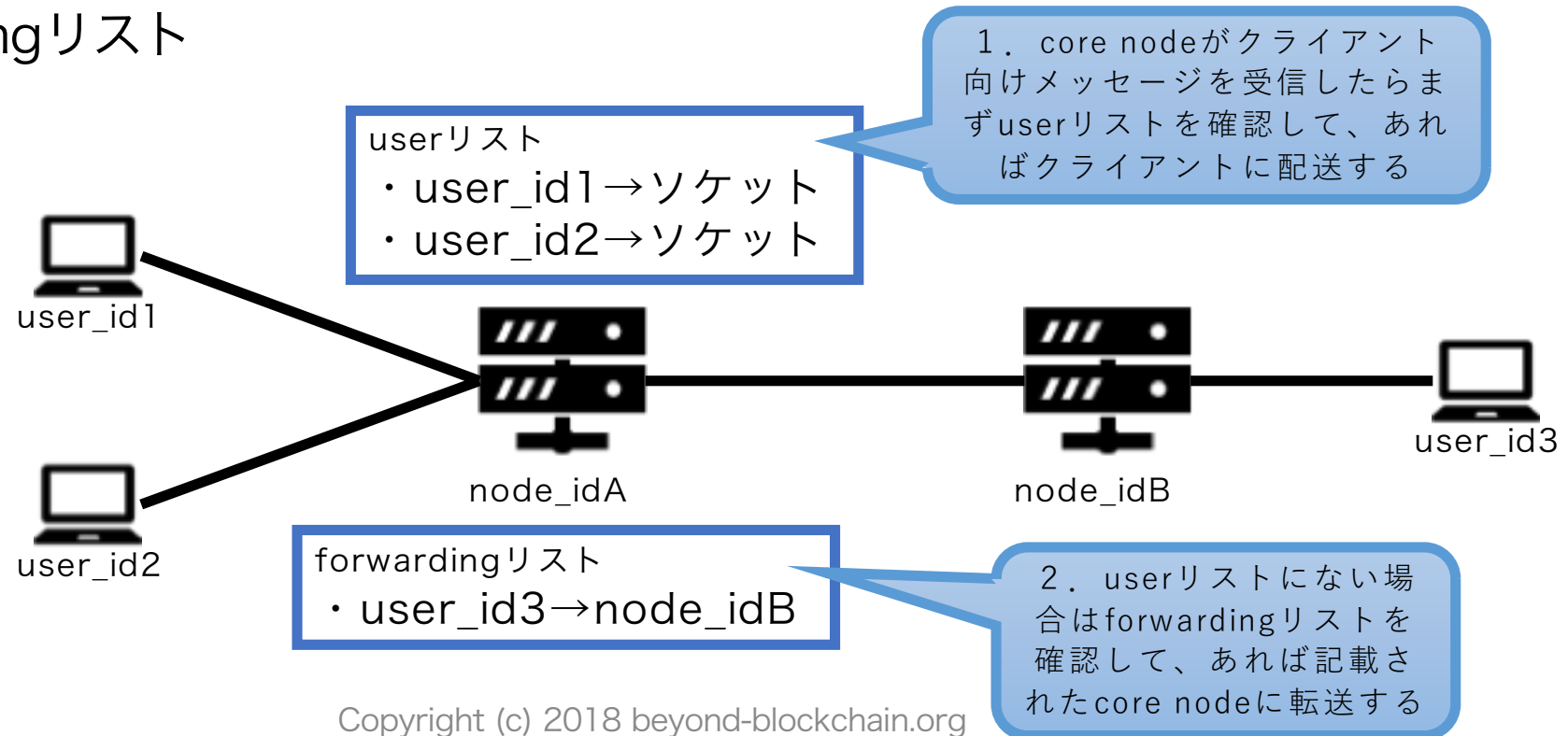
# 隣接ノードリストからの削除

- 離脱メッセージを受け取った時
  - core nodeがドメインから明示的に離脱する場合、ドメイン内に離脱メッセージをブロードキャストする
- 一定時間、隣接ノードリストのエントリが更新されなかった時
  - メッセージ受信や隣接ノードリスト広告の受信により、隣接ノードリストは更新される
  - 一定時間更新がなければ、ノードが離脱してしまったとみなして、リストから削除する
    - 設定値は900秒

# メッセージング (core node-client間)

# クライアントへのメッセージ配送

- core nodeはドメインごとに下記2つのリストを管理し、それを用いてクライアントにメッセージを配送する
  - userリスト
  - forwardingリスト



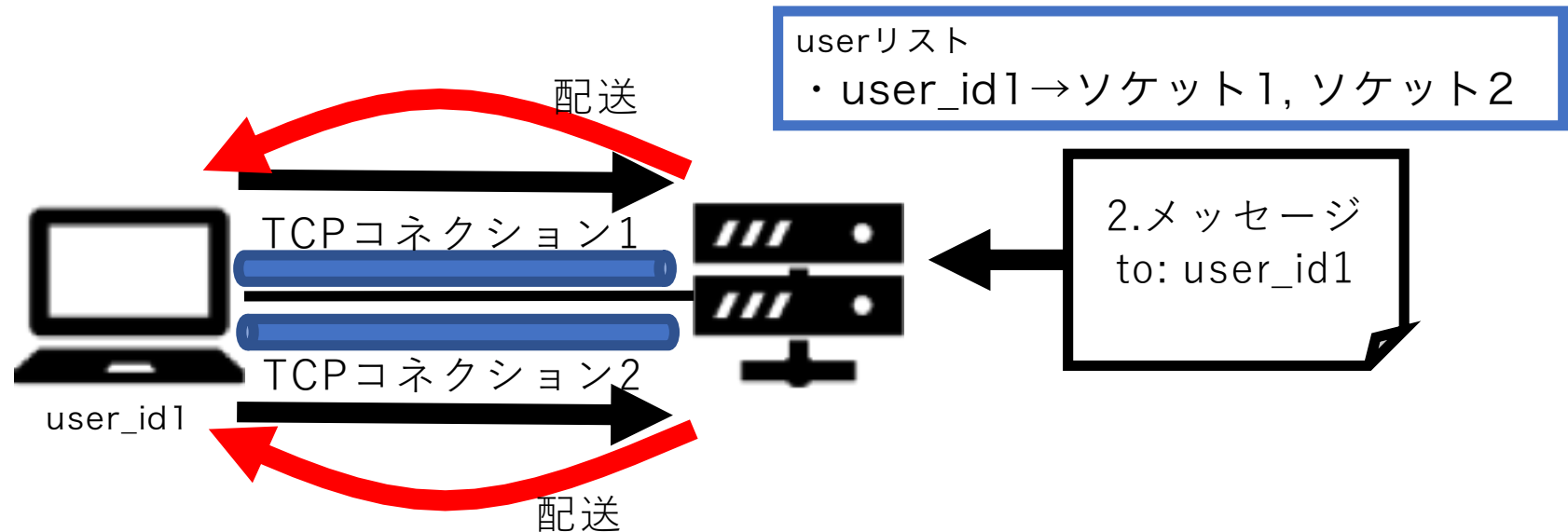
# unicast、multicast、anycast

- clientはcore nodeに対してuser\_idを登録することで、そのuser\_idを宛先とするメッセージを受信できるようになる
- 送信方法
  - unicast (宛先となるuser\_idに対応するクライアントが1つに送る場合)
  - multicast (宛先となるuser\_idに対応するクライアントが複数の場合)
    - 複数のcore nodeにまたがって、複数のclient群が同じメッセージを取得できる
    - multicastであることを示すフラグを付けて登録するだけでよい
      - forwardingリストの同一user\_idに複数のnode\_idがリスト化されて登録される
  - anycast (multicast受信者から1つのクライアントをランダムに選んで送る場合)
    - 送信するメッセージに、anycastであることを示すフラグを付ければよい
    - ロードバランスなどに利用できる

unicast、multicast、anycastかどうかはuser\_idを見ただけでは区別できない

# マルチコネクション

- 1つのcore nodeに対して同じuser\_idの登録が複数のコネクションでやってきた場合、そのuser\_id宛のメッセージは両方のコネクションに向けて送信される
  - core nodeをまたがることは出来ない。その場合はmulticastとして登録する必要がある



# メッセージフォーマット

- core nodeとclient間でやり取りされるメッセージも、core node間と同様にKey-Valueペアが複数格納される
- 主要Key
  - KeyType.domain\_id: どのドメインのメッセージかを表す
  - KeyType.source\_user\_id: 送信元クライアントのuser\_id
  - KeyType.destination\_user\_id: 宛先クライアントのuser\_id
  - KeyType.command: メッセージの種類（これを見て適切な処理を行う）
  - KeyType.query\_id: Request/Response型のメッセージに付加する問い合わせID

# メッセージ種別（管理用コマンド）

タイプ値 (KeyType.command)	説明
REQUEST_SETUP_DOMAIN RESPONSE_SETUP_DOMAIN	core nodeに新しいドメインを作成する
REQUEST_SET_STATIC_NODE RESPONSE_SET_STATIC_NODE	指定ドメインにstatic隣接ノードを設定する
REQUEST_GET_CONFIG RESPONSE_GET_CONFIG	コンフィグファイルの内容を取得する
REQUEST_MANIP_LEDGER_SUBSYS RESPONSE_MANIP_LEDGER_SUBSYS	指定のドメインでledger_subsystemを有効化/無効化する
REQUEST_GET_DOMAINLIST RESPONSE_GET_DOMAINLIST	core nodeが収容しているドメイン一覧を取得する
DOMAIN_PING	指定のドメインにpingを送信する
REQUEST_GET_STATS RESPONSE_GET_STATS	core nodeの統計情報を取得する



# メッセージ種別（管理用コマンド）

タイプ値 (KeyType.command)	説明
REQUEST_GET_NEIGHBORLIST RESPONSE_GET_NEIGHBORLIST	指定ドメインの隣接ノードリストを取得する
REQUEST_GET_USERS RESPONSE_GET_USERS	core nodeの指定ドメインに接続しているユーザのリストを取得する
REQUEST_GET_FORWARDING_LIST RESPONSE_GET_FORWARDING_LIST	指定ドメインで他のcore nodeに接続しているユーザのリストを取得する
REQUEST_GET_NODEID RESPONSE_GET_NODEID	core nodeの指定ドメインにおけるnode_id
REQUEST_GET_NOTIFICATION_LIST RESPONSE_GET_NOTIFICATION_LIST	トランザクション登録完了通知の設定リストを取得する
REQUEST_CLOSE_DOMAIN RESPONSE_CLOSE_DOMAIN	ドメインを削除して、ドメインから離脱する
REQUEST_ECDH_KEY_EXCHANGE RESPONSE_ECDH_KEY_EXCHANGE	ECDHで通信の暗号鍵を交換する

# メッセージ種別（特殊コマンド）

タイプ値 (KeyType.command)	説明
REGISTER	クライアントのuser_idをcore nodeに登録する
UNREGISTER	クライアントのuser_idをcore nodeから削除する
MESSAGE	クライアント間のメッセージであることを表す

# メッセージ種別（登録コマンド）

タイプ値 (KeyType.command)	説明
REQUEST_GATHER_SIGNATURE RESPONSE_GATHER_SIGNATURE	core nodeにSIGN_REQUEST送信を依頼する
REQUEST_SIGNATURE RESPONSE_SIGNATURE	core nodeから対象のclientに対して署名要求 (SIGN_REQUESTの一貫)
REQUEST_INSERT RESPONSE_INSERT	トランザクションを登録する
REQUEST_INSERT_NOTIFICATION CANCEL_INSERT_NOTIFICATION	トランザクション登録完了通知をセットする
NOTIFY_INSERTED	トランザクション登録完了通知
NOTIFY_CROSS_REF	cross_refの配布 (core node間)
REQUEST_REPAIR	改ざんが検出されたトランザクションの修復を要求する

# メッセージ種別（検索コマンド）

タイプ値 (KeyType.command)	説明
REQUEST_SEARCH_TRANSACTION RESPONSE_SEARCH_TRANSACTION	transaction_idを指定してトランザクションデータおよびアセットファイルを取得する
REQUEST_SEARCH_WITH_CONDITIONS RESPONSE_SEARCH_WITH_CONDITIONS	asset_group_id, asset_id, user_idを指定してトランザクションデータおよびアセットファイルを取得する
REQUEST_TRAVERSE_TRANSACTIONS RESPONSE_TRAVERSE_TRANSACTIONS	指定したtransaction_idから過去または未来のトランザクションの履歴を取得する
REQUEST_CROSS_REF_VERIFY RESPONSE_CROSS_REF_VERIFY	他ドメインに指定したtransaction_idの存在を確認する
REQUEST_CROSS_REF_LIST RESPONSE_CROSS_REF_LIST	他のドメインにcross_refとして登録されているはずのtransaction_idのリストを取得する

# メッセージ種別（サブシステム）

タイプ値 (KeyType.command)	説明
REQUEST_REGISTER_HASH_IN_SUBSYS RESPONSE_REGISTER_HASH_IN_SUBSYS	Ledger_subsystemにダイジェストを登録する
REQUEST_VERIFY_HASH_IN_SUBSYS RESPONSE_VERIFY_HASH_IN_SUBSYS	Ledger_subsystemで指定したダイジェストの存在を確認する

※ ethereumまたはbitcoinへの書き込み・検証を行う

# メッセージ仕様 (core node間)

# メッセージフォーマット

- core node間でやり取りされるメッセージは、本書P.10で示した通り、dictionary型のデータ構造であり、Key-Valueペアが複数格納される
- 主要Key
  - KeyType.domain\_id: どのドメインのメッセージかを表す
  - KeyType.source\_node\_id: 送信元core nodeのnode\_id
  - KeyType.destination\_node\_id: 宛先core nodeのnode\_id
  - KeyType.infra\_command: core\_nodeが受信したメッセージをどの機能に渡すかを判断するために用いる
  - KeyType.command: メッセージの種類（これを見て適切な処理を行う）

# メッセージ種別 (bbc\_network)

タイプ値 (KeyType.command)	説明
BBcNetwork.NOTIFY_LEAVE	ドメインを離脱する
BBcNetwork.REQUEST_KEY_EXCHANGE	ECDH鍵交換 (1つ目のメッセージ)
BBcNetwork.RESPONSE_KEY_EXCHANGE	ECDH鍵交換 (2つ目のメッセージ)
BBcNetwork.CONFIRM_KEY_EXCHANGE	ECDH鍵交換 (3つ目のメッセージ)



# メッセージ種別 (data\_handler)

タイプ値 (KeyType.command)	説明
DataHandler.REQUEST_REPLICATION_INSERT DataHandler.RESPONSE_REPLICATION_INSERT	トランザクション複製の登録要求
DataHandler.REQUEST_SEARCH DataHandler.RESPONSE_SEARCH	トランザクション検索
DataHandler.NOTIFY_INSERTED	トランザクション登録通知
DataHandler.REPAIR_TRANSACTION_DATA	トランザクション修復用のデータ
DataHandler.REPLICATION_CROSS_REF	Cross_ref情報の通知

# メッセージ種別 (domain0\_manager)

タイプ値 (KeyType.command)	説明
Domain0Manager.ADV_DOMAIN_LIST	収容しているドメインのリスト広告
Domain0Manager.DISTRIBUTE_CROSS_REF	他ドメインへのCross_ref情報の配布
Domain0Manager.NOTIFY_INSERTED	トランザクション登録通知
Domain0Manager.NOTIFY_CROSS_REF_REGISTERED	Cross_ref登録完了通知
Domain0Manager.REQUEST_VERIFY	Cross_refによる存在証明要求 (同ドメイン内でのメッセージングに利用)
Domain0Manager.REQUEST_VERIFY_FROM_OUTER_DOMAIN Domain0Manager.RESPONSE_VERIFY_FROM_OUTER_DOMAIN	外部ドメインへのCross_refによる存在証明の 要求(domain0でのメッセージング用)

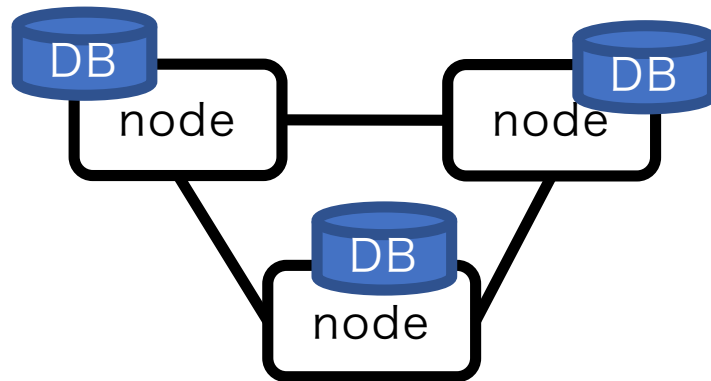
# メッセージ種別 (user\_message\_routing)

タイプ値 (KeyType.command)	説明
UserMessageRouting.RESOLVE_USER_LOCATION UserMessageRouting.RESPONSE_USER_LOCATION	クライアントuser_idの探索する（どのcore nodeに接続しているか）
UserMessageRouting.JOIN_MULTICAST_RECEIVER	user_idをマルチキャストアドレスとして登録する
UserMessageRouting.LEAVE_MULTICAST_RECEIVER	user_idをマルチキャストツリーから削除する
UserMessageRouting.CROSS_REF_ASSIGNMENT	ドメイン内のcore nodeにcross_refを配布する

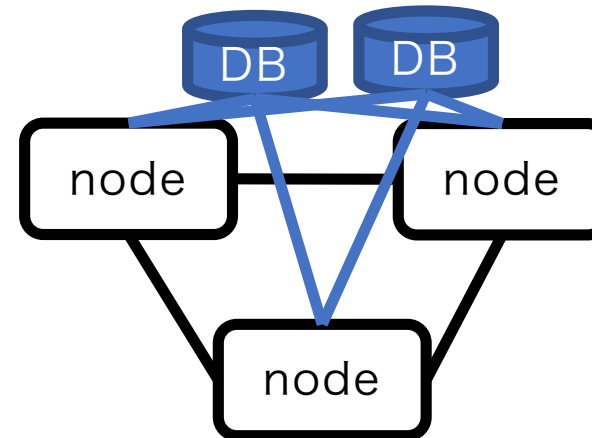
# データベース

# トランザクションの保管

- トランザクション情報およびそれにまつわる付加情報はデータベースに格納される
  - v1.0時点ではSQLite3かMySQLを選択できる
- DBの配置パターンには下記のように2通りある



各core nodeがDBを持つパターン  
(設定は"replication\_strategy: all")

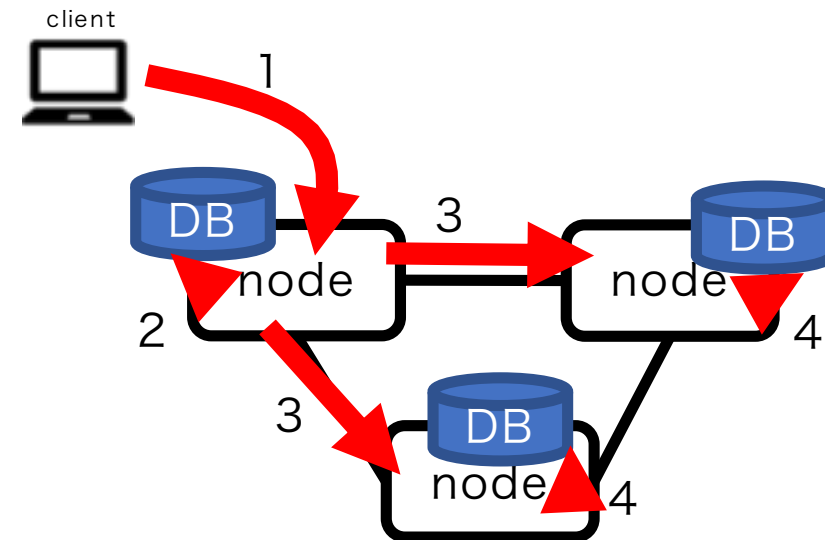


DBを外部ノードに配置するパターン  
(設定は"replication\_strategy: external")

# トランザクションのinsert

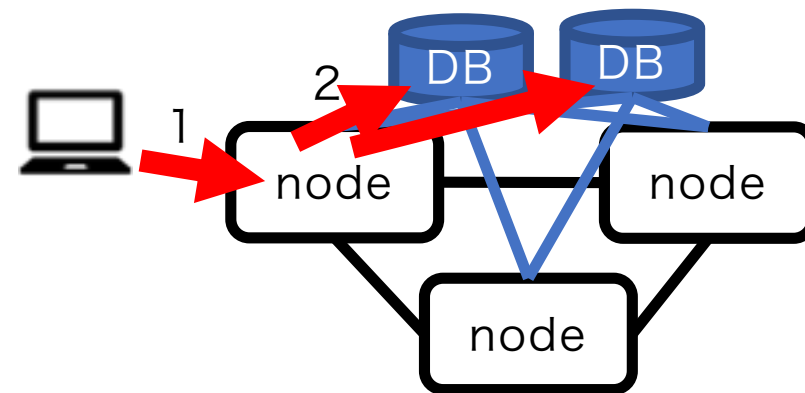
- 全てのcore nodeがDBをもつ場合

1. core nodeはクライアントからinsertコマンドを受け付ける
2. core nodeはトランザクションの署名を検証し、問題なければ自分自身のDBにトランザクションを登録する
3. ドメイン内の全てのcore node (つまり隣接ノード) にトランザクションの複製を通知する
4. 複製を受信したcore nodeも、トランザクションの署名を検証し、問題なければ自分自身のDBにトランザクションを登録する



- 外部のDBを利用する場合

1. core nodeはクライアントからinsertコマンドを受け付ける
2. core nodeはトランザクションの署名を検証し、問題なければ接続している全てのDBにトランザクションを登録する



# トランザクションのsearch

- 全てのcore nodeがDBをもつ場合/外部のDBを利用する場合
  1. core nodeはクライアントからsearchコマンドを受け付ける
  2. core nodeは接続しているDBからトランザクションを検索する
    - 外部のDBを利用している場合は、いずれか1つのDBを検索する
  3. トランザクションに紐づくアセットファイルがあればそれもストレージから取得する
  4. 取得したトランザクションの署名の検証とアセットファイルのダイジェストを検証する
    - 不正があった場合は、応答メッセージのKeyType.compromised\_transactionsやKeyType.compromised\_asset\_filesの項目にデータを格納する
  5. 検索結果をクライアントに返答する

# トランザクションの修復

- トランザクション検索結果メッセージに、 `compromised_transactions` や `compromised_asset_files` が含まれていた場合、コア内のDBが改ざんされている
  - BBc-1 v1.0では改ざんデータの自動修復は行わない
  - これは「改ざんがあった」という事実自体にも情報があると考えられるため、それをユーザに通知することを目的としている
- 改ざんされたトランザクションを修復するには、複製の中から正しいものを見つけてそれで上書きする
  - トランザクション修復は、クライアントから対象となる `transaction_id` を指定して `REQUEST_REPAIR` コマンドを `core node` に通知すれば、修復手順が作動する



# データベーススキーマ (基本)

- 各データベースは以下のテーブルを持つ

- transaction\_table

transaction_id	transaction_data
----------------	------------------

カッコ内は型。カッコのないものはBLOB型

- asset\_info\_table (transaction\_idを様々な条件の組み合わせで検索できる)

id (int)	transaction_id	asset_roup_id	asset_id	user_id
----------	----------------	---------------	----------	---------

- topoloby\_information\_table (トランザクションの履歴を辿るときに利用する)

id (int)	base	point_to
----------	------	----------

- cross\_ref\_table (他ドメインにcross\_refとして登録されているものの情報を保持)

id (int)	transaction_id	outer_domain_id	txid_having_crossref
----------	----------------	-----------------	----------------------

# データベーススキーマ (subsystem)

- つづき

- merkle\_branch\_table

digest	leaf_left	leaf_right
--------	-----------	------------

- merkle\_leaf\_table

digest	leaf_left	leaf_right	prev
--------	-----------	------------	------

- merkle\_root\_table

root	spec
------	------

カッコ内は型。カッコ  
のないものはBLOB型

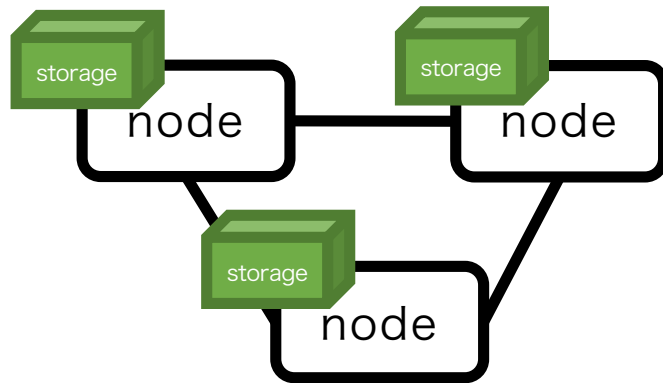
# トランザクションをinsertするときのレコード作成手順

1. core nodeがシリアルライズされたトランザクションを受信する
  - そのtransaction\_idをtx\_Aとする
2. トランザクションをデシリアルライズして、トランザクションの署名を検証する
3. トランザクションの構造を解析し、含まれているBBcEventまたはBBcRelationオブジェクトを見つける
  - 各オブジェクトについて、含まれているアセットのasset\_group\_idとasset\_id、およびそのアセットの所有者user\_idを、レコードにして、asset\_info\_tableにinsertする
  - 別ファイルとして取得したアセットがあれば、それはストレージに格納する
4. トランザクションの構造を解析し、含まれているBBcReferenceまたはBBcPointerオブジェクトを見つける
  - 各オブジェクトについて、含まれている参照先transaction\_idとtx\_Aを使って2つのレコードを作り、topoloby\_information\_tableにinsertする
  - 2つのレコードは、transaction\_idとtx\_Aをそれぞれbaseとpoint\_toおよびその逆の組み合わせにしたものである
5. 最後にtransaction\_tableに、transaction\_id (=tx\_A)とシリアルライズされたトランザクションの組をinsertする
6. ここまでの手順で何か失敗があれば、すべてロールバックする

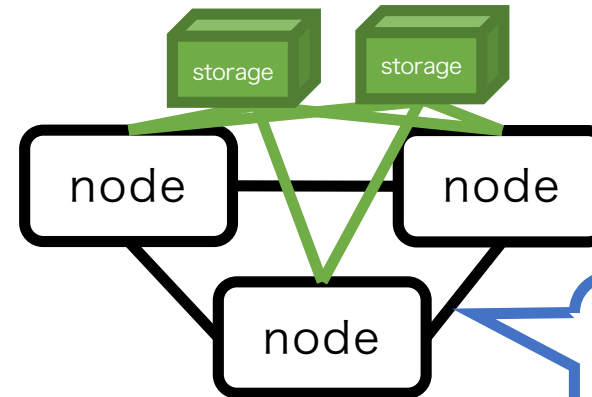
# ストレージ

# アセットファイルの保管

- アセットファイルは、ストレージに保存される
  - アセットファイルはdomain\_id/asset\_group\_id/ディレクトリの下にasset\_idというファイル名で保存される
- 配置パターンには下記のように2通りある



各core nodeがストレージを持つパターン  
(設定は"type: internal")



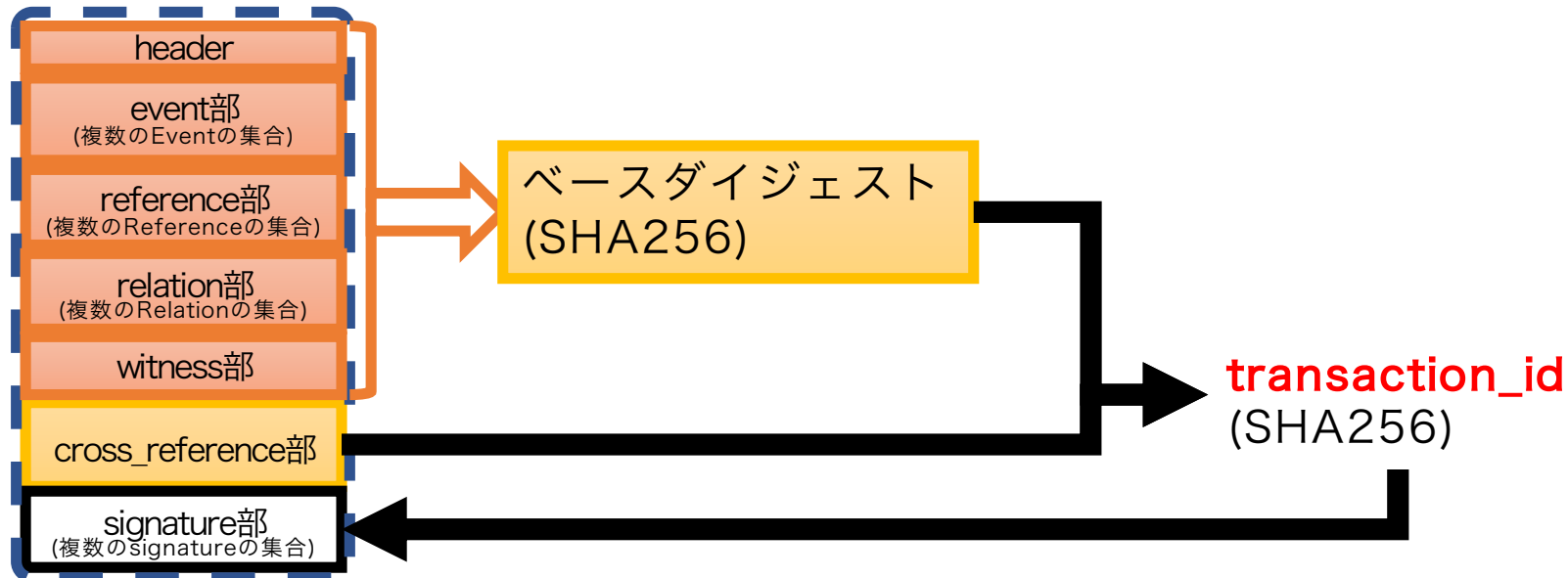
ストレージを外部ノードに配置するパターン  
(設定は"type: external")

このパターンではbbc\_coreはファイルアクセスを行わず、クライアントが管理する必要がある

# ダイジェスト計算

# トランザクションとダイジェスト

- トランザクションデータは、2段階のダイジェスト計算によって、transaction\_idの算出する
  - ダイジェスト関数はSHA256を用いる
- トランザクションへの署名は、実際にはtransaction\_idへの署名である

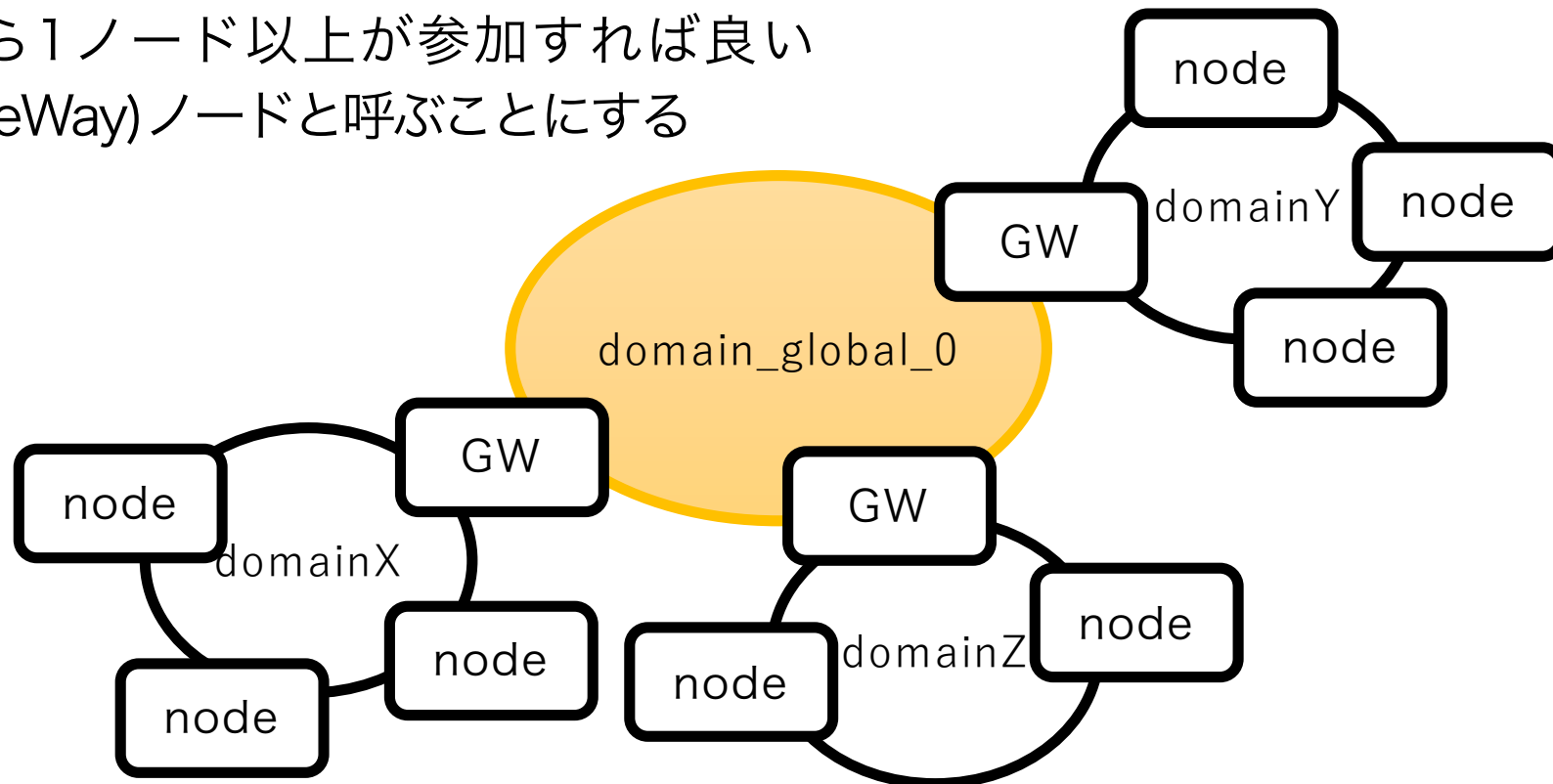


# 履歴交差 (Cross\_ref) の手順



# domain\_global\_0

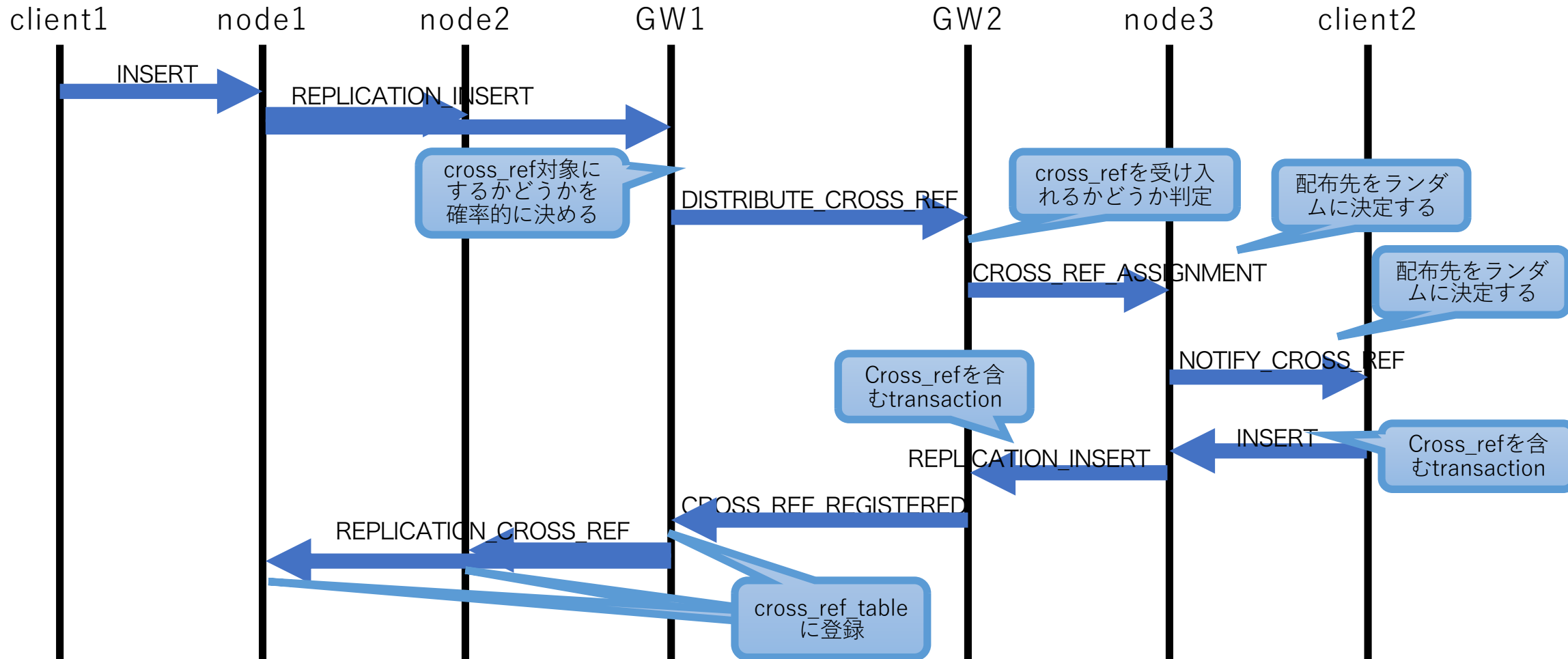
- 履歴交差情報(Cross\_ref)をドメイン間で交換するためのグローバルな共有ドメイン
  - domain\_idは256bit全て0で表される
- ドメインから1ノード以上が参加すれば良い
  - GW (GateWay)ノードと呼ぶことにする



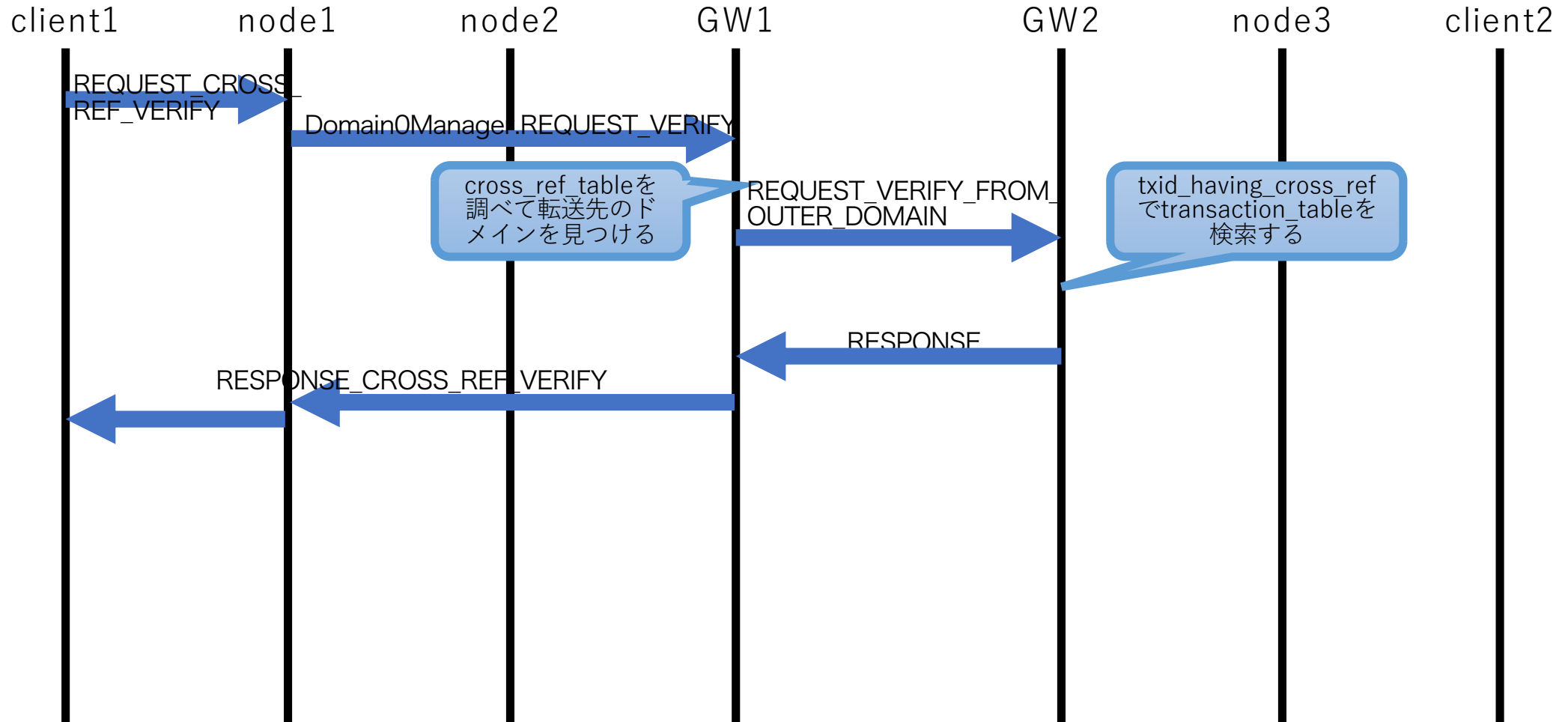
# Cross\_refの利用方法

- 組み込み
  - あるドメイン(domainX)で発生したトランザクション(Tx\_A)に関するcross\_refを他のドメイン(domainY)で発生するトランザクション(Tx\_B)に組み込んでもらう
    - domainX側では、Tx\_AがdomainYに登録されたことを1レコードとして、DBのcross\_ref\_table に書き込む
- 検証（存在確認）
  - domainXのクライアントがTx\_Aのトランザクションが確かに過去に正しく登録されたものであることを確認する
    - domainYに[domainA, Tx\_A]をcross\_refにもつトランザクションを問い合わせる
    - domainYのGWは、検証用のベースダイジェストと署名をdomainXのGWを経由してクライアントに送信する（ベースダイジェストについてはP.48参照）
    - ベースダイジェストとCross\_ref情報を用いて署名が正しく検証できれば、過去にTx\_Aのトランザクションが確かに存在していたことが証明できる

# Cross\_ref配布・組み込みシーケンス



# Cross\_ref検証シーケンス



# 配布/組み込み手順の中でのGWの処理

- 自ドメインないで発生したトランザクションをCross\_ref対象にするかどうかの判定
  - v1.0では、10%程度の確率で対象にする（暫定値）
  - 対象にするものだけ、他のドメインにむけてDISTRIBUTE\_CROSS\_REFメッセージでCross\_refを配布する
    - 配布先はdomain\_global\_0のノードからランダムに3ノード選ぶ
- DISTRIBUTE\_CROSS\_REFを受け入れるかどうかの判定（暫定）
  - 一定期間内に1つのドメインから受け取れるDISTRIBUTE\_CROSS\_REFの数には上限を設ける
    - 上限値に達した場合、それ以降はそのドメインからDISTRIBUTE\_CROSS\_REFは破棄する
    - 一定時間ごとに受け入れ可能数が回復する
  - CROSS\_REF\_REGISTEREDを受け取ると、その上限値を増加させる

# 検証手順の中でのGWの処理

- 自ドメインからDomain0Manager.REQUEST\_VERIFYメッセージを受け取ると、ドメインリストを調べて、他ドメインの宛先core nodeを取得する
  - そのcore nodeにREQUEST\_VERIFY\_FROM\_OUTER\_DOMAINメッセージを送る
- REQUEST\_VERIFY\_FROM\_OUTER\_DOMAINを受け取ると、cross\_ref\_tableから要求されているtransaction\_idをCross\_refに含むトランザクションのtransaction\_idを取得する
  - さらにそこからトランザクションを取得し、ベースダイジェスト(P.48)を計算して、結果を返答する

このやり方だと  
cross\_refの配布先が同  
じようなドメインばか  
りになる可能性がある

# 改ざんからの修復

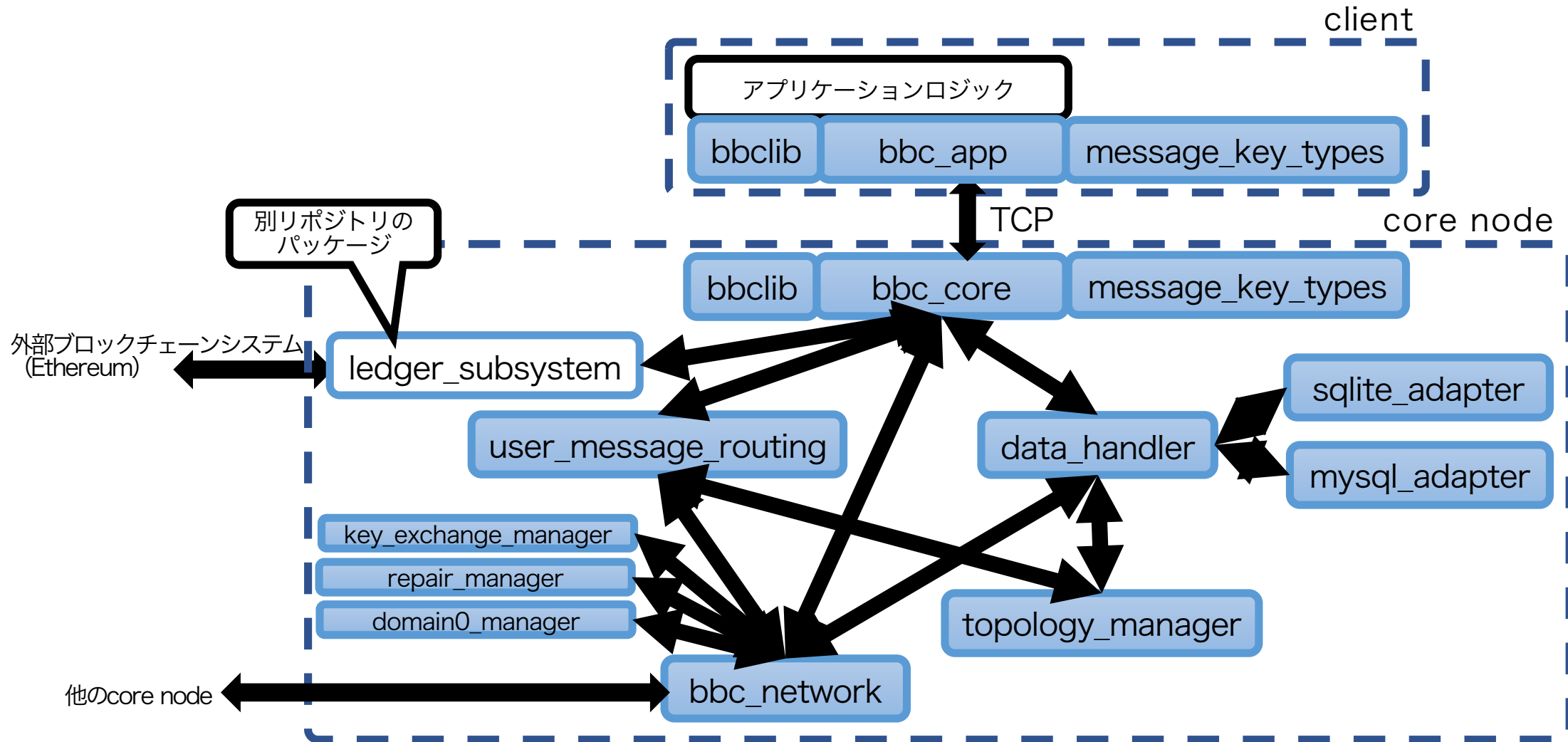
# 改ざんのパターンと修復方向

- トランザクションデータが改ざんされたときの修復方法
  - 改ざんを検出したcore nodeが複数のDBに接続している場合、接続している全てのDBをチェックし、正しいものがあればそれで改ざんされたレコードを上書きする
  - 全てのDBで当該トランザクションデータが改ざんされていた場合、他のノードに正しいものを送ってもらうように依頼する
    - この手順は、コンフィグのdb→replication\_strategy: allの場合のみ有効
- アセットファイルが改ざんされたときの修復方法
  - 他のcore nodeから正しいアセットファイルを送ってもらうように依頼する
    - この手順は、コンフィグのstorage→type: internalの場合のみ有効



# プログラム構成

# core nodeのプログラム構成



# ソースコードとその役割(全体制御)

- `bbc_app.py`
  - クライアント機能 (アプリケーションはこの機能を利用、または継承する)
- `bbclib.py`
  - トランザクションオブジェクトの生成、操作
- `bbc_core.py`
  - BBc-1システムの起動スクリプト
  - システム初期化、コンフィグ読み込み・書き出し
  - クライアントとの接続、コマンド受け付け
- `bbc_network.py`
  - ドメインオブジェクトの管理
    - ドメイン作成、ドメイン削除など
    - ドメイン制御機能オブジェクトの初期化・保持
  - 隣接ノードとの通信機能
- `message_key_types.py`
  - メッセージパース、シリアライズ、暗号化/復号処理

# ソースコードとその役割(ドメイン制御)

- data\_handler.py
  - トランザクション、アセットの登録・検索
  - DB、ストレージの管理
- topology\_manager.py
  - 隣接ノードの発見、トポロジ管理
    - v1.0時点では、フルメッシュトポロジにのみ対応するので単純な実装になっている
- user\_message\_routing.py
  - クライアントへのメッセージング機能
- repair\_manager.py
  - トランザクションが改ざんされたときの修復機能
- domain0\_manager.py
  - 履歴交差 (cross\_ref) 関連機能
- ledger\_subsystem.py (別パッケージとしてインストールが必要)
  - EthereumやBitcoinへのアンカリング機能

# ソースコードとその役割(その他)

- `sqlite_adapter.py`
  - SQLite3のDBへのアクセス機能
- `mysql_adapter.py`
  - MySQLのDBへのアクセス機能
- `key_exchange_manager.py`
  - ECDH (Elliptic Curve Diffie-Hellman)による鍵交換機能

以上