クラウドシステム基礎

第4回：順序づけ

国立情報学研究所
石川　冬樹
f-ishikawa@nii.ac.jp

今回の内容

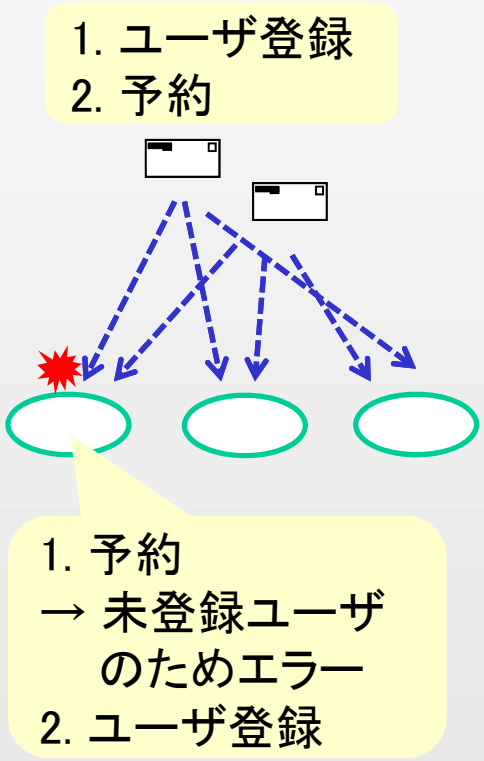
- 順序づけに関する性質とプロトコルを議論する
  - イベントの順序づけのための基本手法である論理クロックを学ぶ
  - 順序つきマルチキャストの性質と実現方法を議論する
  - 複製管理における一貫性の定義を確認する

目次

- 順序と因果関係，一貫性
- 論理クロック
- 順序つきマルチキャスト
- 複製データ管理の一貫性

順序にかかわる不具合

- 複製に対するマルチキャストが様々な順序で到着する可能性があり，状況によってはそれは不具合を引き起こす
  - 右は第1回に挙げた例：
    「意味（因果関係）のある順序で送り出されたが，逆転して到達」

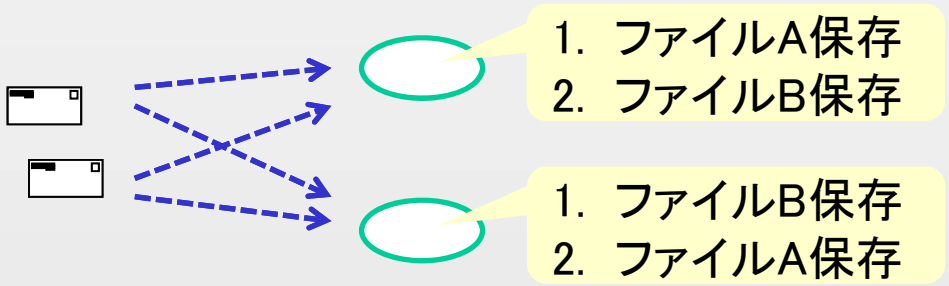


考慮すべき順序は？

どう「順序通り」に届ける？

順序と因果関係

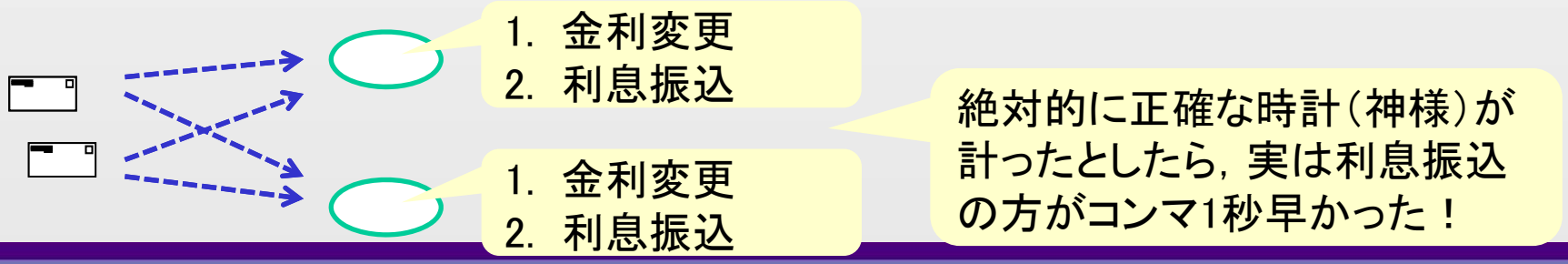
- あらゆる順序を考慮したいわけではない
  - 大半のイベントの間には論理的な因果関係や依存性はない
  - ➡それらの前後関係を常にすべて明確にしたり，複数のプロセス間で一意に合意できるようにしたりする必要は必ずしもない



（ファイルAとBは別ディレクトリで互いに影響がないファイル）

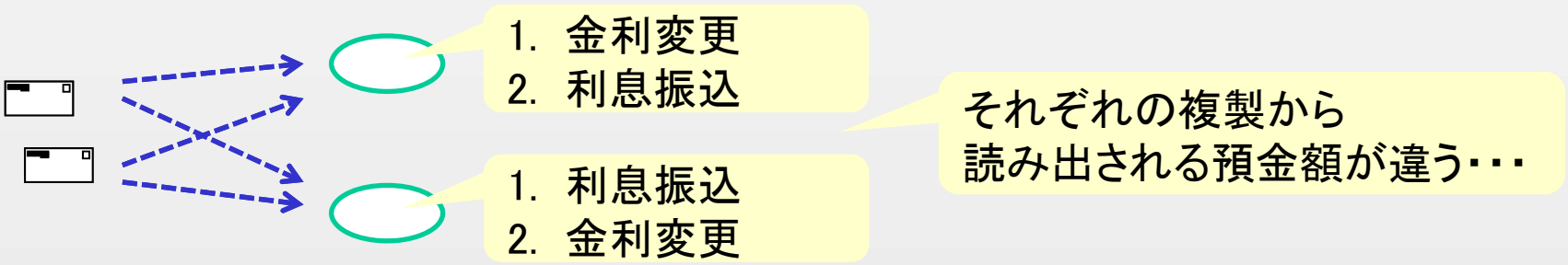
順序と絶対時間

- 絶対的に正しい一つの順序が必要とは限らない
  - 物理的な絶対時間は，ある程度の精度でしか計れず，（神様視点では存在するはずだが）絶対的な前後関係を把握することはできない
  - アプリケーションユーザにとって意味を持つほどの時間差でなければ，こだわる必要がない
  - ただしアプリケーションの意味上適切な順序が存在することはありうる（先の「登録後，予約」）



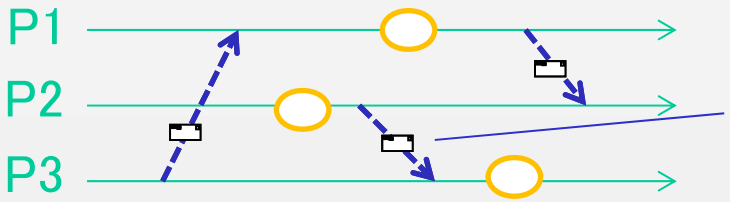
順序と合意

- ただし，複製間である一つの一意な順序を合意すべき状況はある
  - 「合意されるのがこの順序でなければダメ」という1つの正解が決まっている必要は必ずしもないというのが先の話



補足：順序と因果関係（複製以外での意義）

- 複製管理に限らず，因果関係による順序は重要
  - 例：スナップショット（各プロセスが「いっせいに」実行状態を記録，様々な利用方法あり）



P3のメッセージ受信は記録されているが
P2からの送信は記録されていない

例：バックアップに用いると，再開したとき再度配信

例：状態監視やテストログに用いると，ロックを2つのプロセスが持っているような状況が出てくるなど，デバッグが混乱

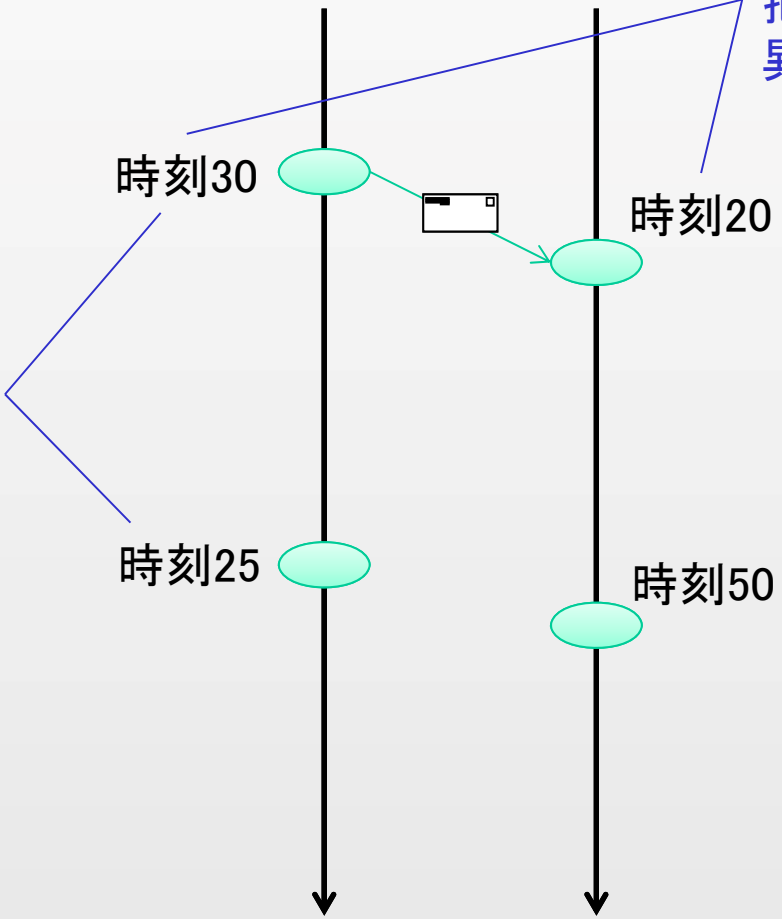
目次

- 順序と因果関係，一貫性
- 論理クロック
- 順序つきマルチキャスト
- 複製データ管理の一貫性

論理クロック

- 異なるノードが持つ物理クロックの示す時間は，必ずしも厳密，正確に合致しないが，
  - 因果関係のあるイベント間の順序関係を正しく把握できるようにしたい
  - システム全体でイベント間の順序関係を合意できるようにしたい
- イベント間の順序関係を表す値をうまく決める（**論理クロック**）
  - 絶対時間に基づく順序関係と一致する必要はないが，因果関係を反映した値になっていて欲しい

論理クロック：極端にダメな与え方の例

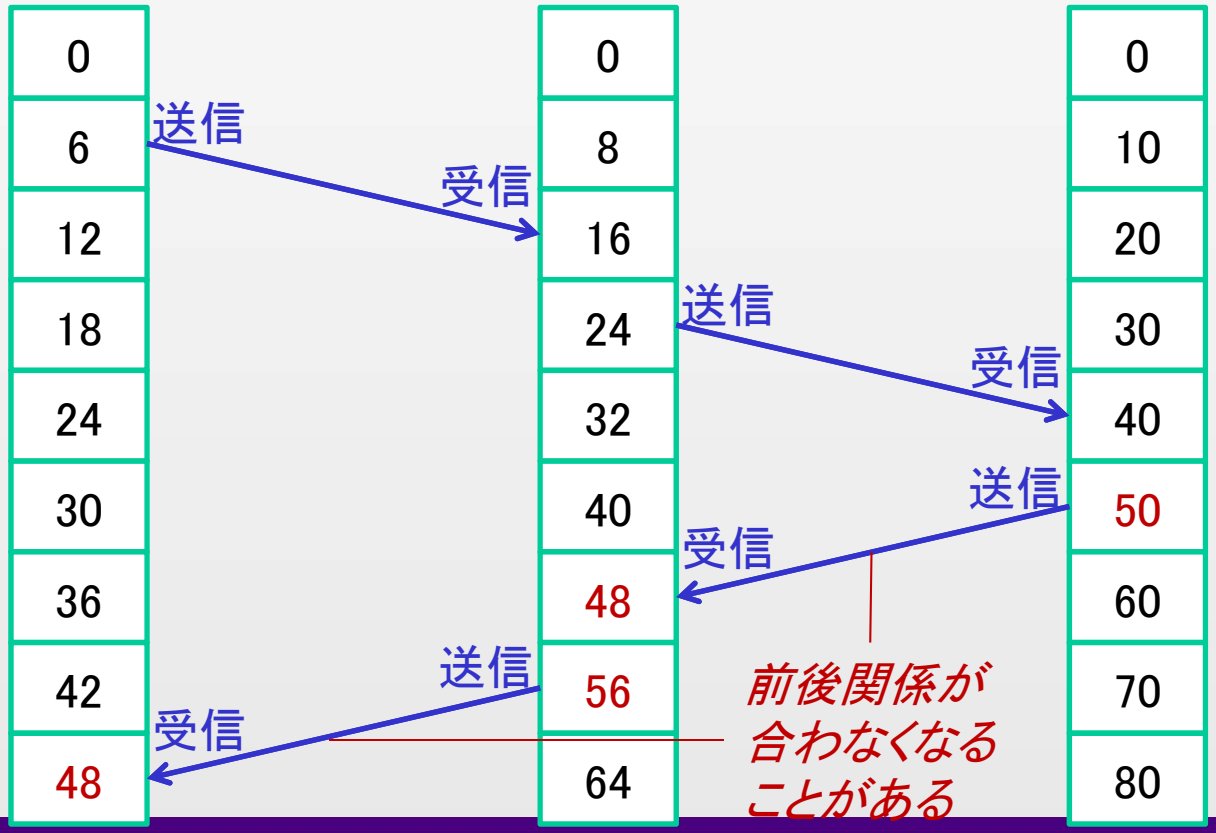


送受信という因果関係から
把握できる前後関係と
異なっている！

同一プロセスの中で
明らかに把握できる
前後関係と
異なっている！

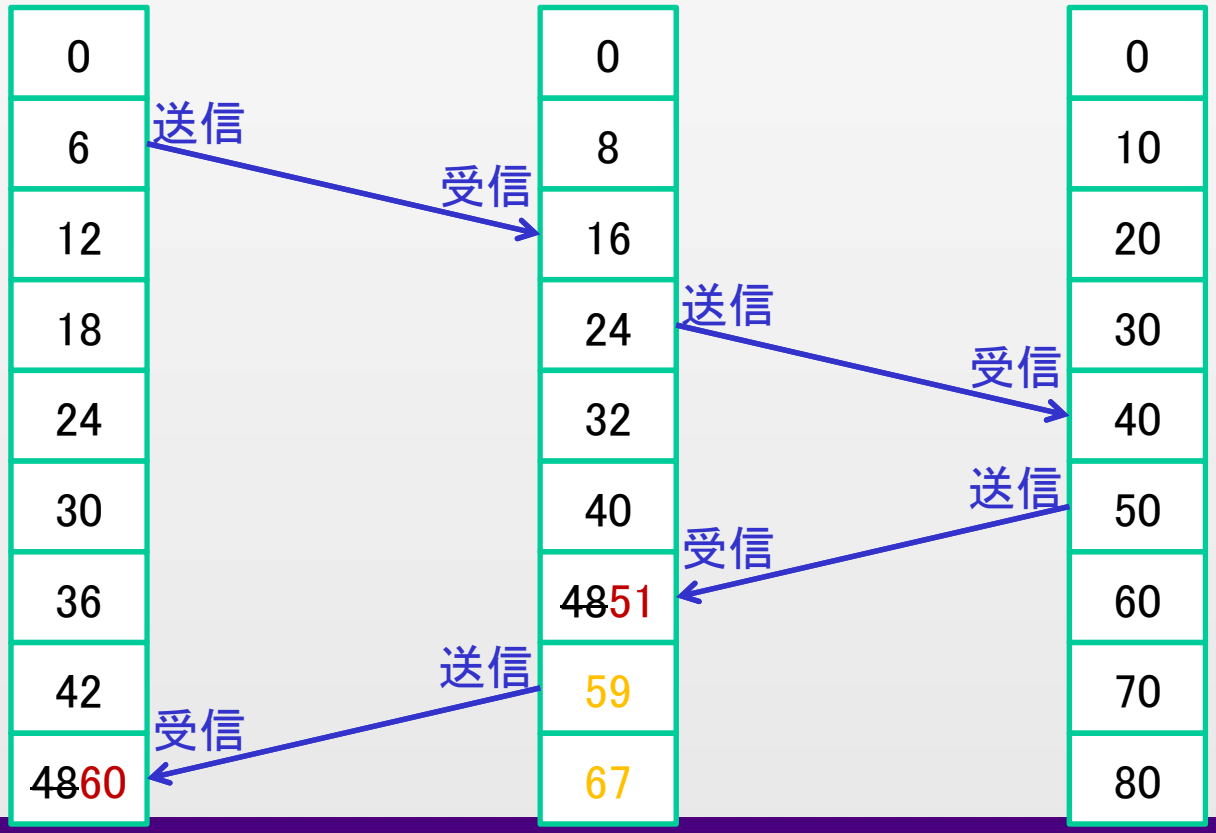
論理クロック：クロック修正

- もし各ノードの物理クロック（互いにずれている）をそのまま使うと・・・



論理クロック：クロック修正

- 論理クロック修正 [Lamport, 1978]
  - 受信時のクロックを必要に応じ調整する



受信時のクロック値が該当する送信発生時のクロック値以下なら，「送信発生のクロック値＋1」に修正

論理クロック：クロック修正

- 保証できる性質
  - 同じプロセスにおいて，イベントaがbより前に発生するならば，Clock(a) < Clock(b)
  - イベントaおよびbがそれぞれ同じメッセージの送信，受信を表すならば， Clock(a) < Clock(b)
- 追加：各プロセスの識別子を小数点以下に付けると，すべてのイベント間に順序が定まる
  - 例えばプロセス1の40を40.1とする
  - このとき，すべての異なるイベントaおよびbに対して，Clock(a) ≠ Clock(b)

論理クロック：クロック修正

- 保証できない性質
  - Clock(a) < Clock(b) であっても，イベントaがbより早く起きたとは限らない
  - 数値の大小が必ずしも因果関係に起因したものになっていないことがある

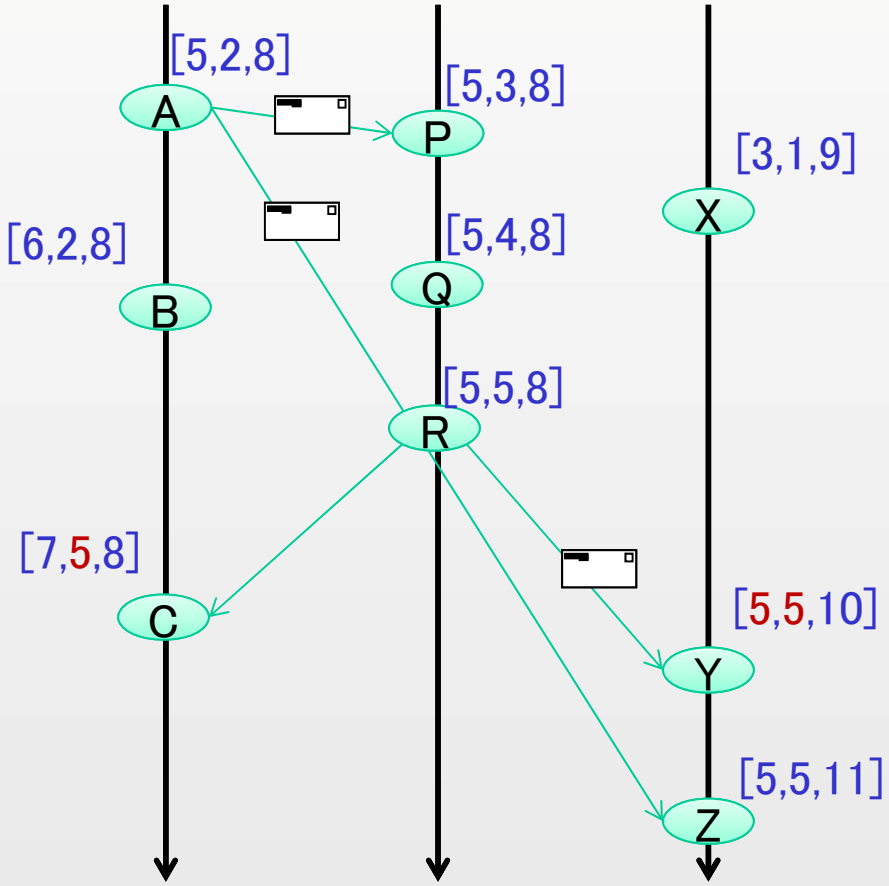
論理クロック：ベクトルタイムスタンプ

- ベクトルタイムスタンプ
  - 「これまでどのプロセスで何個のイベントが発生したのか？」という情報（各プロセスが自分なりに把握しているもの）をタイムスタンプと見なす
  - この情報をメッセージ送信時に添付し，受信者はその情報を更新する

論理クロック：ベクトルタイムスタンプ

- ベクトルタイムスタンプ
  - 各プロセス P_i は，ベクトル V_i を維持し，タイムスタンプとしてメッセージ送信時に添付
    - $V_i[i]$：P_i において今まで発生したイベントの数（イベント発生時にインクリメント）
    - $V_i[j]$：P_j において今まで発生したイベントの数に関するPiの知識
  - 各プロセス P_j は P_i からのメッセージ内のベクトル vs を受信したら，（上記インクリメントに加えて）
    - $V_j[k]$ を $\max\{V_j[k], vs[k]\}$ に更新（自分が知らなかったイベント発生を教えてもらう）

論理クロック：ベクトルタイムスタンプ



v1, v2において,
∀i v1 [i] ≦ v2 [i]
∃i v1 [i] ＜ v2 [i]

⇔ v1はv2より前に起こった

例：
A[5,2,8]はY[5,5,10]より前

C[7,5,8]とY[5,5,10]はどちらが前とも断言できない

目次

- 順序と因果関係，一貫性
- 論理クロック
- 順序つきマルチキャスト
- 複製データ管理の一貫性

順序つきマルチキャスト

- 今回の最初に挙げたような不具合を避けるために，特定の順序をつけてマルチキャストを実現する方法を考える
  - 一般的にはミドルウェアやフレームワークなどが実装すべき機能
  - 満たすべき順序の定義はいくつかありうる

FIFOマルチキャスト

- 同じプロセスが送ったメッセージは，そのプロセスが送り出した順序で，各プロセスに受信される
  - TCPやそれに類似した仕組みを用いればよい：

    送信プロセスはシーケンス番号をインクリメントしながら送信し，受信プロセスはそれに基づいてメッセージを並び替えて（届いていない番号が若いメッセージを待って）届ける
  - 送信プロセスがクラッシュしても順番を覚えているようにし，全プロセス宛に送信したことを確認するようにする

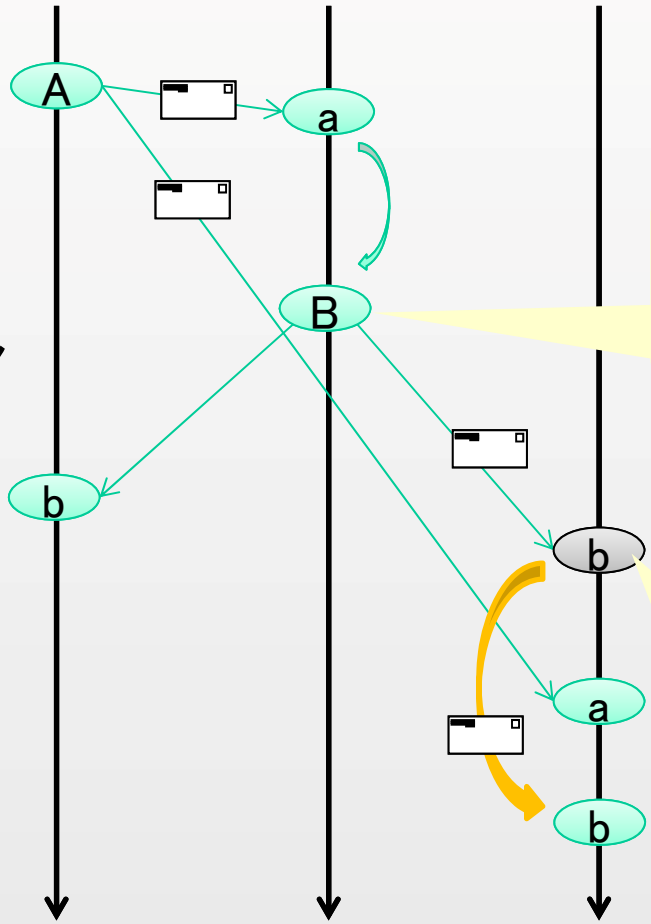
因果順序マルチキャスト

- 因果関係を示唆する前後関係があるメッセージは，それを満たす順序（causal order）で各プロセスに受信される（結果FIFOにもなる）
  - あるメッセージの送信イベントは，そのメッセージの受信イベントよりも前に起きた，と考える
  - 同一プロセス内で，逐次的な実行の中で順に先に実行されたイベントは，後に実行されたイベントよりも前に起きた，と考える
  - イベントAがイベントBより前に起きた，イベントBがイベントCより前に起きた，というとき，イベントAはイベントCより前に起きた，と考える

因果順序マルチキャスト：動機となる例

書き込み処理を受け付け

掲示板アプリケーションのプロセス複製



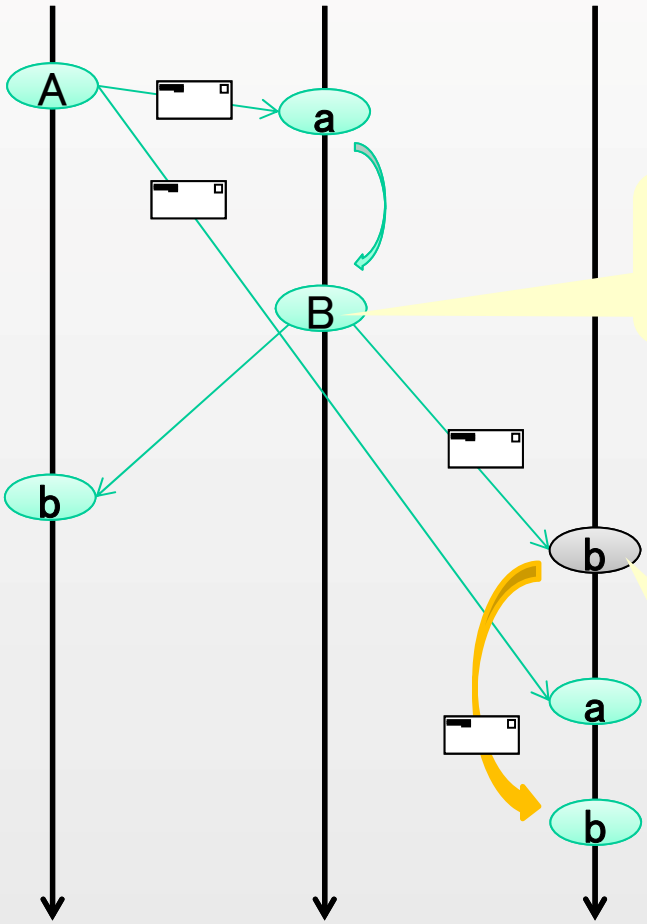
この複製プロセスに接続しているユーザーが，書き込みを見て返信書き込み

Aの結果Bが起こったという因果関係があるため，Bのメッセージはのものよりも後にアプリケーションに渡すようにしたい

（受信側が並び替えをすることによる実現方式のイメージ）

因果順序マルチキャスト：一般論

マルチキャスト



因果関係によりAの結果として
起こった可能性がある
マルチキャスト

Bのメッセージは Aのものより
も後にアプリケーションに渡
すようにする

（受信側が並び替えをすることによる実現方式のイメージ）

因果順序マルチキャスト

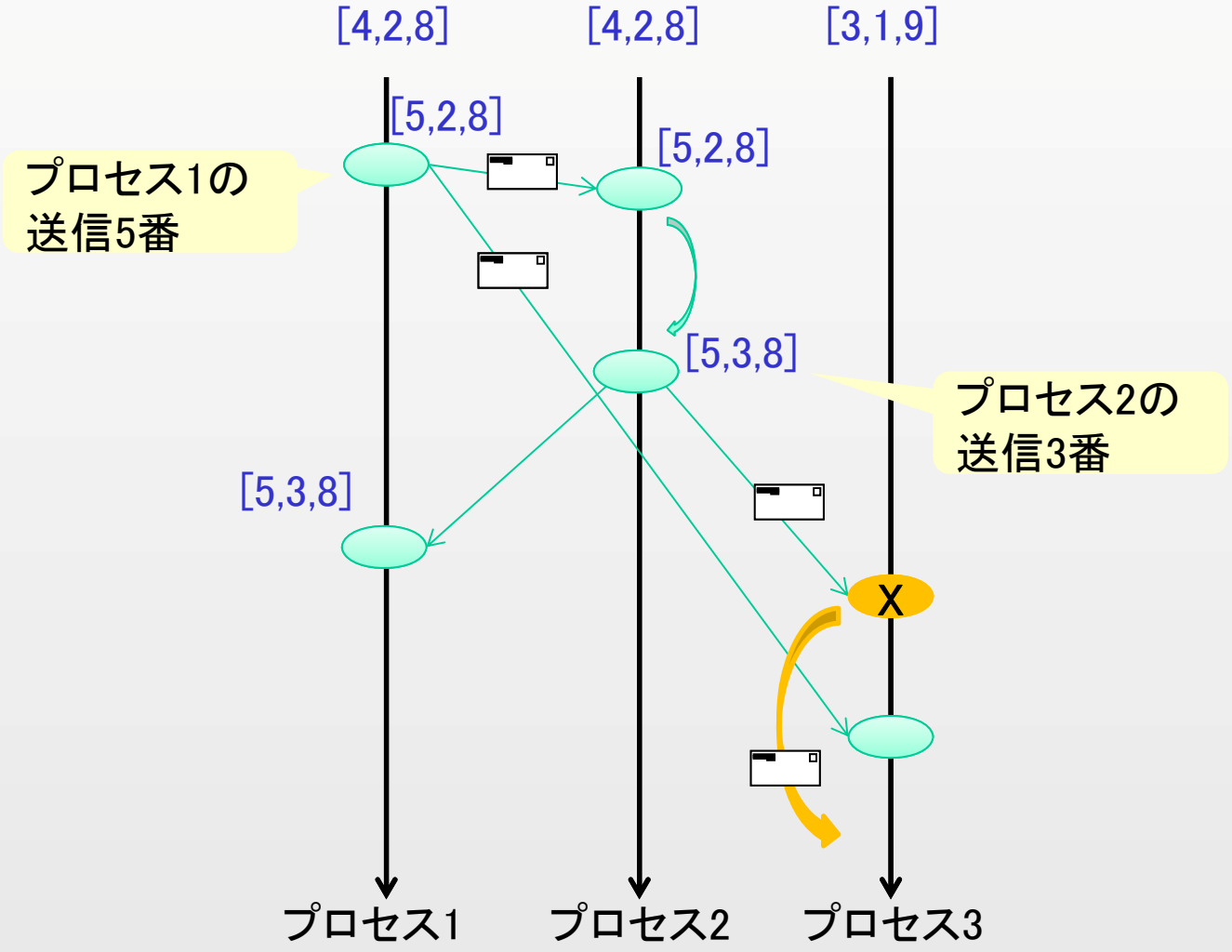
- 論理クロックを用い，「因果的に先に起きたはずのマルチキャスト」を把握すればよい
- ベクトルタイムスタンプの場合，
  - マルチキャスト送信イベントに対してだけ，ベクトル内のカウントインクリメントを行うようにする
  - あるマルチキャストのメッセージは，それよりも先に起きたことが断言できるマルチキャストをすべて受け取るまで，遅らされる

※ 因果関係がある際は，断言できる

演習2: 課題

- 次スライドに，3つのプロセスの実行状況を示す
  - マルチキャストの発生の様子や，それに付けられたタイムスタンプが記されている
  - 上部には，それ以前より各プロセスが保持していたタイムスタンプが記されている
  - タイムスタンプの管理方法は前スライドの通り
- 因果順序マルチキャストに従うとき，プロセス3にXにて届いたメッセージを実際に配信するために最低何個のメッセージを待つ必要があるか？
  - 図に書かれている以前のメッセージもありうる

演習2: シナリオ



演習2：課題

- 余裕があれば，さらに，下記の条件を検討せよ
  （今の例を一般化せよ）
  - プロセス P_j がタイムスタンプ V_j を持つときに，プロセス P_i からタイムスタンプ vs を持つメッセージを受信したとする
  - 因果順序に従うとき，このメッセージを実際に配信してよいのは，V_j と vs の間にどのような条件が成り立つときか？

解説資料は別途配付

因果順序マルチキャスト

- 限界
  - 同じプロセス内の2つの送信イベント間の順序は常に考慮される
    - アプリケーションロジック上では因果関係がなく，たまたまどちらかが先になっただけかも
    - 汎用的な機能としては，安全に考慮
  - 不要な前後関係まで強制するオーバーヘッドが発生し，性能に影響を与える
    - 特にネットワークが不安定でイベントが多く，到達順序が入れ替わりやすい場合
    - 代わりに，アプリケーション側で順序制御する？

全順序マルチキャスト

- 全順序（total-order）：マルチキャストが，各プロセスに同じ順序で配信される
  - 全順序マルチキャスト，全順序FIFOマルチキャスト，全順序因果マルチキャストがありうる
  - 一般的に実現コスト（オーバーヘッド）が高い）

※ 因果順序マルチキャストは，タイムスタンプによるメッセージ増加（ただし圧縮の工夫あり）と，順序調整の待ち時間（ネットワークが安定していればあまり起きない）だけでコストは低い

全順序マルチキャスト

- 全順序マルチキャストの実現方法
  - 実現例：トークンをプロセス間で順に回し，トークンを持つプロセスが順序を決めて周知
  - 実現例：調整者が，各プロセスでの受信時クロックの値を集め，その最大値を皆で用いる
  - 実現例：各プロセスはメッセージ受信時にackをマルチキャストし，クロック修正されたタイムスタンプの小さい順にキューに入れる．キューの先頭にあるものに対し，他の全プロセスからackが来たら実際に受信

アトミックマルチキャスト

- Atomic（原子的）：マルチキャストが対象プロセスのうち生存しているものすべてに確かに配送される，またはいずれにも配送されない
  - 前回のグループ管理に基づくと，メッセージが届かなかった受信者は対象グループから外されるという扱いになるので，基本的に「生存しているものすべてに着く」ことになる
  - ただし，送信者がマルチキャストの途中でクラッシュし，一部のプロセスにだけメッセージが着いた場合，他のプロセスにも届ける必要がある

アトミックマルチキャスト

- 実現方法の例
  - マルチキャストを受け取ったプロセスは，必要ならそれを自分が転送できるように保管しておく
  - 送信者は，TCPなどの到達確認を用いる場合など，送信成功が確認できたら，各ノードにその旨通知し，保管の必要性がないことを伝える（これは次のマルチキャストに織り込んでもよい）
  - 送信者がグループから脱落した場合，各受信プロセスは何かの順に従い，送信者の代わりにマルチキャストと上記の成功通知

耐久性のあるアトミックマルチキャスト

- 「配送される」だけでは，直後にクラッシュして消えてしまうかもしれない
- トランザクション（分散コミット）同様に，耐久性がある（durable）ように，メッセージの効果が生きているプロセスに反映されるようにしたい
- ➡ 2PC同様の，送信（準備），確定という手順を経る（コストが大きい）

補足：マルチキャスト送信途中での障害

- 送信者がマルチキャストの途中でクラッシュする状況：メッセージを受け取り済みで，送信者の脱落を知った受信者が再送するようにしてもよい

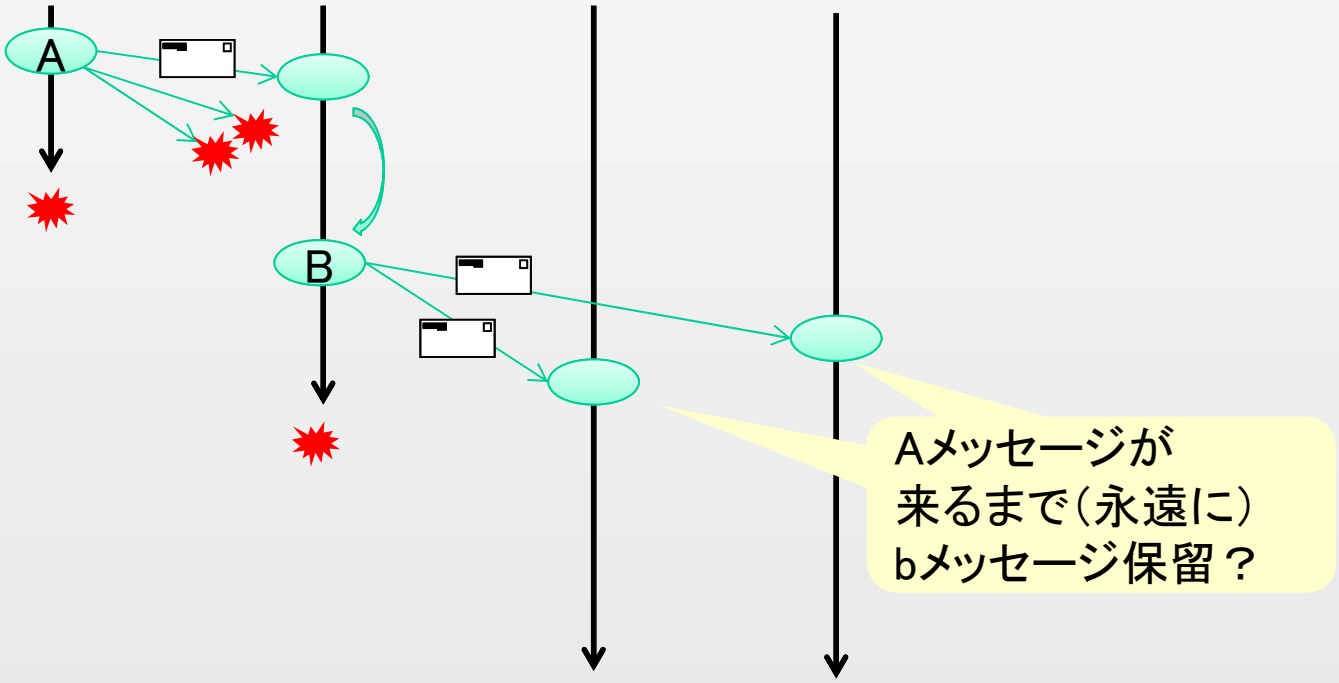
  ※ 前回のグループ管理のような仕組みを前提

  - ただし，先に送信者の脱落が共有されてしまうかもしれない

  ➡ 脱落通知を止めて該当プロセスからのメッセージの配信を確定させるといった対応を検討する必要がある

補足：マルチキャスト送信途中での障害

- 因果順序マルチキャストの場合，脱落した送信者によるメッセージの扱いを徹底しないと・・・



おまけ

- 例えば,
  - http://docs.jboss.org/jbossas/docs/Administration_And_Configuration_Guide/5/html/ch10s01.html

目次

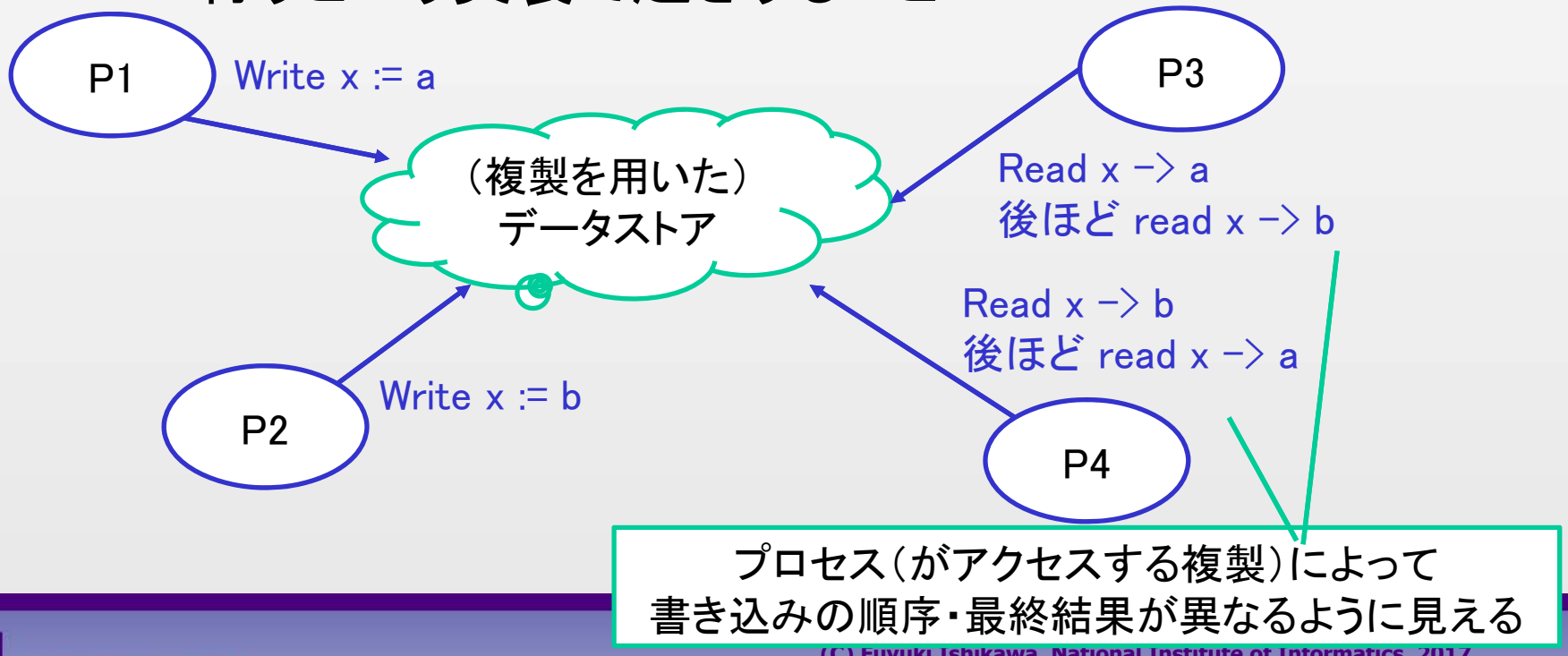
- 順序と因果関係，一貫性
- 論理クロック
- 順序つきマルチキャスト
- **複製データ管理の一貫性**

データの複製における一貫性

- 今度はアプリケーション側の視点から，データの複製に対する操作を考えてみる
  - ミドルウェアやフレームワークなどは，データの読み取りや書き込みのAPIをアプリケーションに提供
    （裏では，今回学んだマルチキャストなどを用い，複製間での書き込み内容共有などを行う）
  - アプリケーションにとって，読み書きしたときにどう見えるのか？
    ※ APIにより保証される「一貫性」は？

データの複製における一貫性

- 順序制御など全く行わない場合
  - 書き込みは順序制御なしのマルチキャストで全複製に反映し，読み込みは複製いずれかに対して行うという実装で起きうること



一貫性モデル

- 一貫性モデル
  - データを管理するデータストア(ミドルウェアなど)と，アプリケーションプロセスとの間の約束事
    - プロセスが守るべきルール
    - データストアが保証する性質
  - 例：厳密な一貫性(strict consistency)

    「データに対する読み取りは，必ずそのデータの絶対時間で最も新しい書き込みの結果を返す」
    - 分散システムでは実装できない
      (例：絶対時間1ナノ秒の差で2つの更新が別の複製に行われた場合，新しい値を必ず残せる？)

一貫性モデル

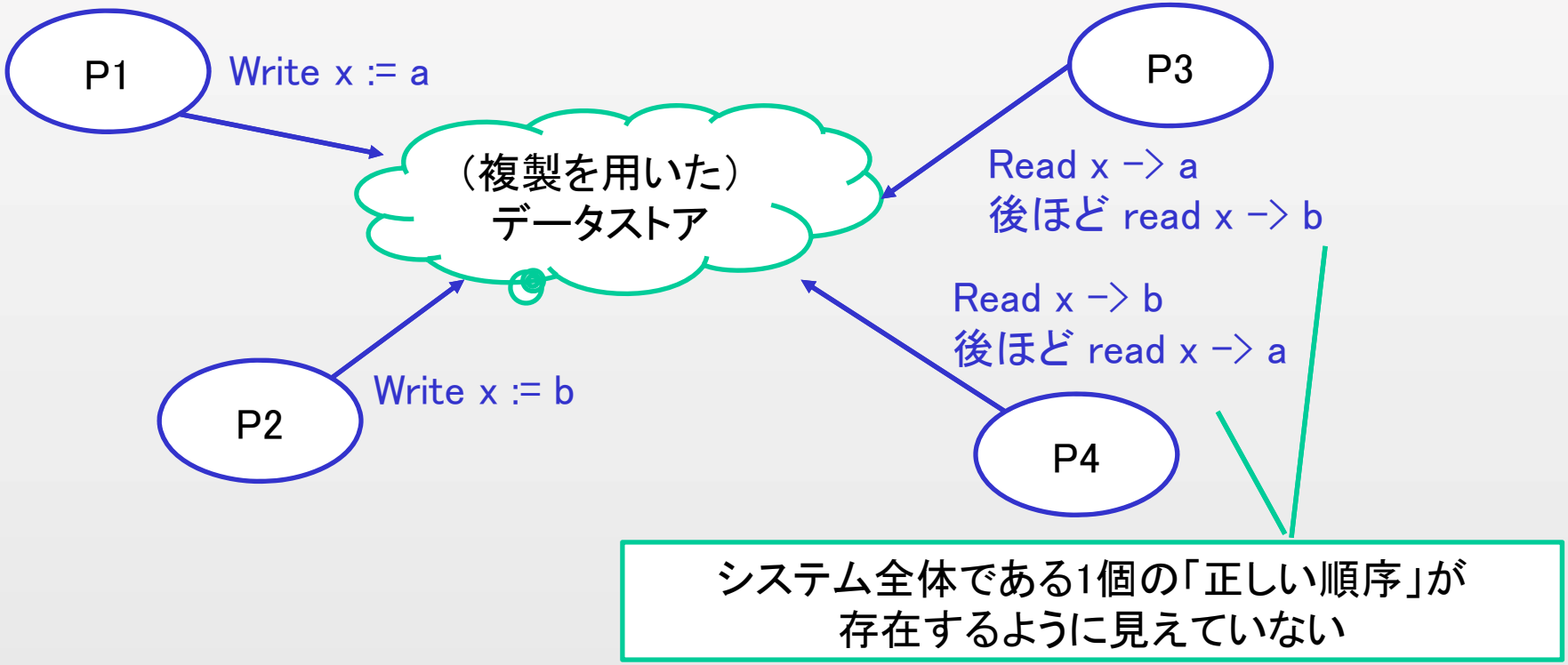
- 一貫性モデルの実現
  - より強い性質を保証するような一貫性は，実現が高コストになる
  - 様々な種類の一貫性モデルが定義されている
  - マルチキャストと同様の分類も出てくる
    （内部の仕組みからの視点か，外部の仕様からの視点かの違い）

一貫性：順序一貫性

- 順序一貫性（sequential consistency）
  - 「どのような操作の結果も，すべてのプロセスによる読み書き操作があたかもある直線的な順序で行われたとしたときの結果と同一となる」
  - 「各プロセスの操作は，そのプログラムによって指定された順序で現れる」

一貫性: 順序一貫性

- 順序一貫性(sequential consistency)
  - 下記のような状況は許さない

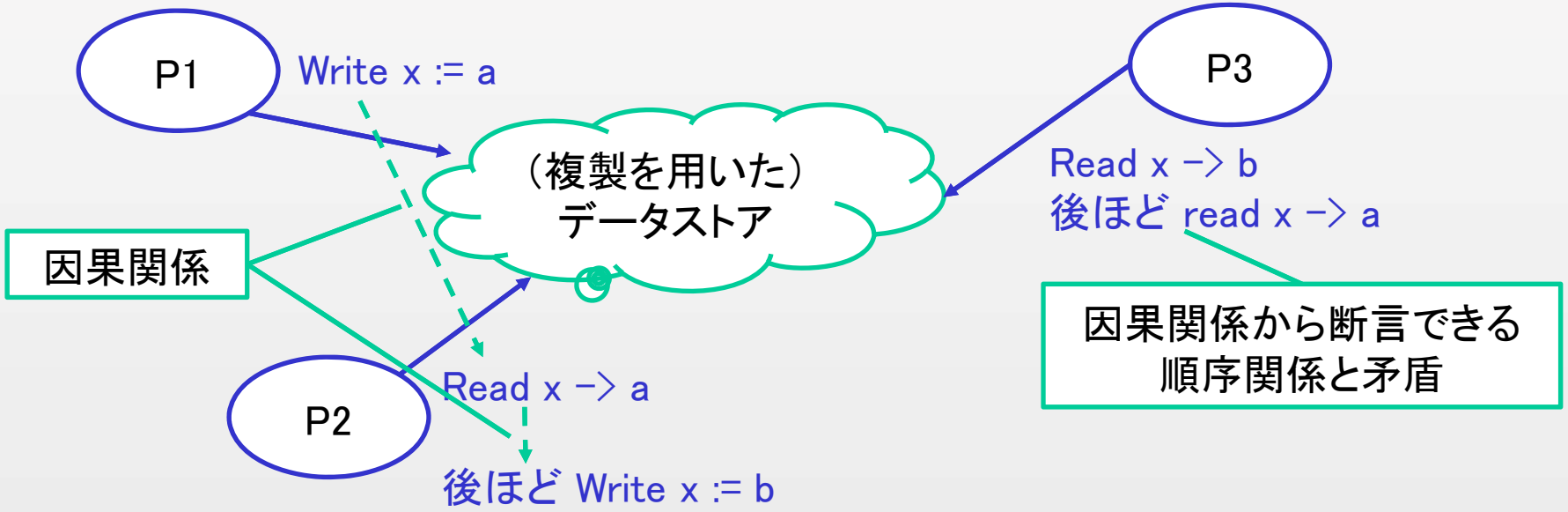


一貫性: 因果一貫性

- 因果一貫性(causal consistency)
  - 「因果的に関連している可能性のある書き込みは,すべてのプロセスによって同じ順序で観測される」

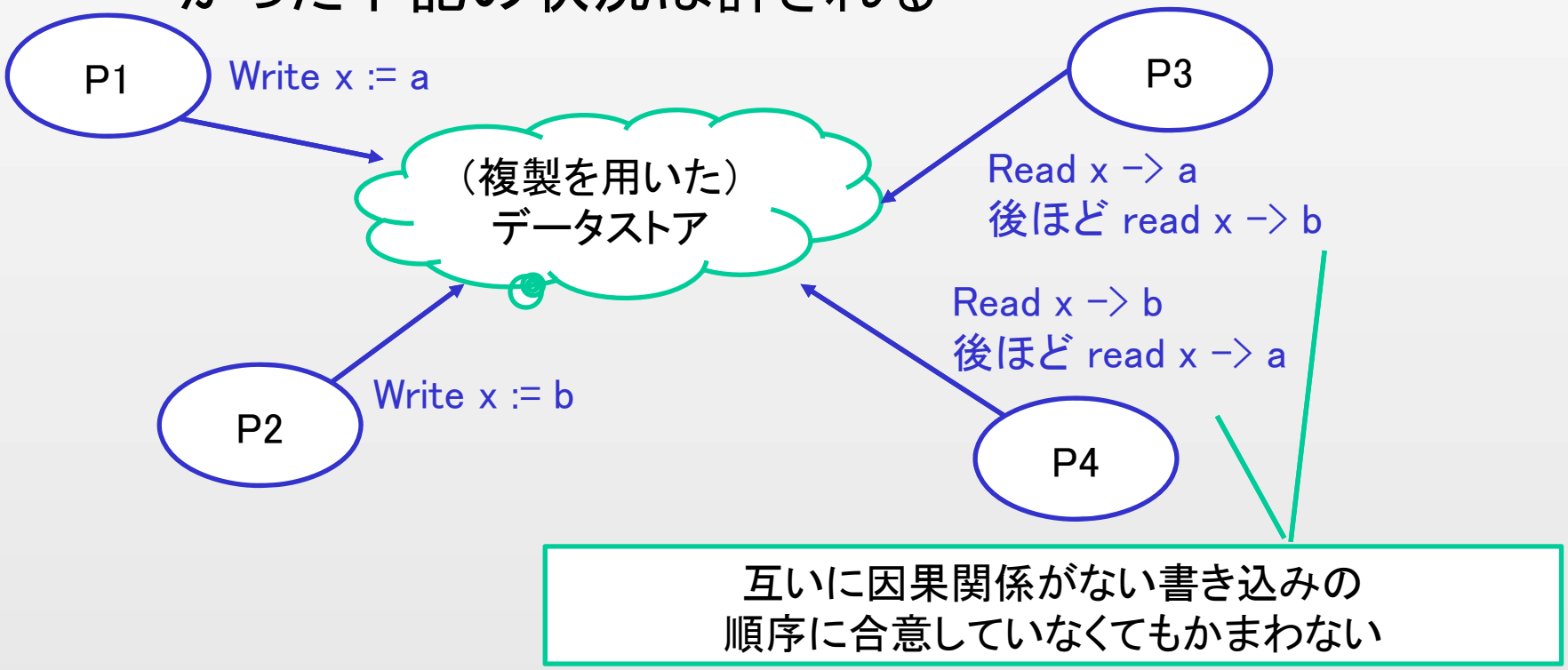
一貫性: 因果一貫性

- 因果一貫性(causal consistency)
  - 下図のような状況は許さない



一貫性: 因果一貫性

- 因果一貫性(causal consistency)
  - 順序一貫性を保証するデータストアでは許されなかった下記の状況は許される

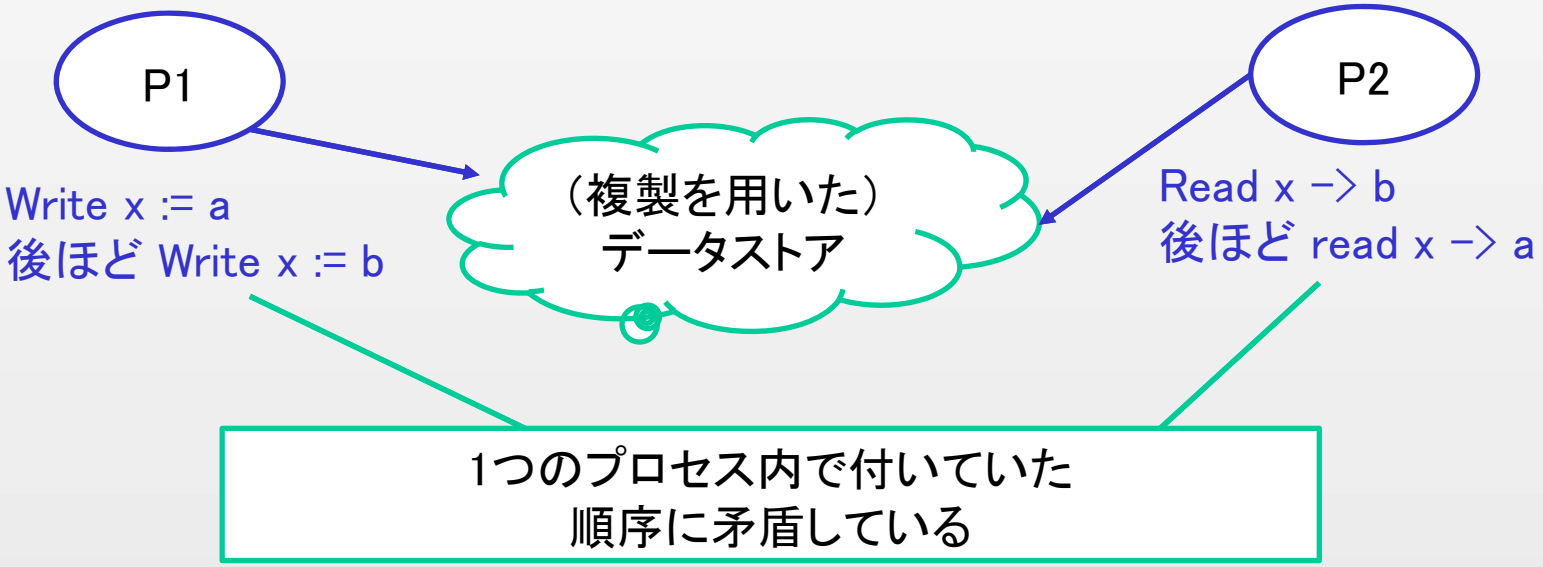


一貫性モデル：FIFO一貫性

- FIFO一貫性（FIFO consistency）
  - 「単一のプロセスによってなされた書き込みは，すべてのプロセスによって同じ順序で観察される」

一貫性モデル: FIFO一貫性

- FIFO一貫性(FIFO consistency)
  - 下図のような状況は許さない



一貫性モデル：弱一貫性

- 弱一貫性（weak consistency）
  - 「同期変数へのアクセスは順序一貫である」
    - タイミング合わせを行う手段を用意する
  - 「すべての先行の書き込みがすべての場所で完了するまで，同期変数への操作は許容されない」
    - 読み込み前に同期操作を行えば，最新の値を読み取ることができる
  - 「すべての先行の同期変数へのオペレーションが完了するまで，データへの操作は許容されない」
    - 書き込み後に同期操作を行えば，進行中・部分的な書き込み，書き込み結果の伝搬を完了できる

一貫性モデル：イベンチュアル一貫性

- イベンチュアル一貫性（eventual consistency）
  - 「結果一貫性（結果整合性）」などの翻訳も
  - 「更新はすべてのコピーに伝搬する」（更新がなければすべての複製は同一に収束していく）
  - 以下の性質を持つデータストアに適する（DNSやWebキャッシュが代表例）
    - 書き込み同士の衝突はない
    - 読み込みが必ずしも最新でなくてもかまわない

近年クラウドで注目を集めている（次回の内容）

「一時的に一貫性が崩れてもよいのでスケール！」

今回のまとめ

- イベント発生の順序，特に因果関係によるものの制御や合意は，アプリケーションが整合性ある形で実行されるために非常に重要である
  - その制御や合意のための実現手法は多く提案されており，利用されている
  - 実現する性質とオーバーヘッドの間にトレードオフがあるため，アプリケーションの整合性担保のためにも，実現すべき性質をよく議論すべきである
- 次回：Amazon Web Servicesなどの実例を通し，クラウドの設計思想を議論する