分散システム基礎とクラウドでの活用

第6回：演習

国立情報学研究所
石川　冬樹
f-ishikawa@nii.ac.jp

今回の内容

- これまで学んだ内容の復習と，それらに関する演習，議論を行う

目次

- 演習：順序制御の必要性
- 演習：故障への対応
- これまでのまとめ
- 演習：アプリケーションとの照らし合わせ

あらためて，本講義の位置づけ

- クラウドにおいて，大規模環境での性能やスケーラビリティ確保のため，「一貫性」をある程度犠牲にするという傾向（第5回）
- クラウド上でのアプリケーション構築（やクラウドの構築）の際には，トレードオフを踏まえた難しい選択が求められる
- 「一貫性」の定義や意義の理解（第2～4回）
  - 犠牲にしてよいものなのか，判断できるように
  - 必要と判断されたなら使いこなせるように
    （今後「一貫性」の選択肢が増えていく？）

おまけ：講師の個人的な見解

- Key-Value Storeなどが話題に
  - AmazonやGoogleが，自分たちのために，作り使っていた（たぶん過去形）ものを外部に発表，提供
  - 「真に」必要とするところはすぐ活用（SNSなど）
- ➡ 求められる発想転換への反動
  - 今まで当然と思っていた一貫性保証のあきらめ
  - これまでの知識・経験が役に立たなくなる
- ➡ 正しい理解と適切な融合・使い分けが主流に
  - 一貫性保証やRDB風APIも追加されてきている
  - これまでの技術を選ぶべき状況も多数

演習6：複製を用いたデータストアの実現

- 複製を用いたデータストアの実現を題材として，異なる種類のマルチキャストによる様々な実現方式を議論してみる
  - 一つ一つの実現方式の選択肢に対して，時間をとるので，自分自身で理解し，評価を行ってみる

題材：複製を用いたデータストアの実現

- アプリケーション（例えばFirst-Tierプロセスやクライアントプロセス）が様々な操作（API）を実行し，それを複製プロセスの1つが受け取り処理する
  - 例：UPDATEであれば全体に反映するようマルチキャストなどを行う
  - こういったAPIの実現方法や利用方法を議論する
- 下記方針でデータストアの実現を考える
  - UPDATE操作は複製全体に反映
  - READ操作は複製どれかから読み出す
    （READの割合が多くその性能が重要な場合）

題材：複製を用いたデータストアの実現

- LOCK操作が必要になることもある
  - 例：READして状況をチェックし，その結果に応じてUPDATEする（その間に値が変わると不整合）
  - 例：一連のUPDATEを一括して反映する（一部のみ他UPDATEの結果が混ざると不整合）
- 明示しない場合，障害は考えなくてよい
  - 落ちていると思われるプロセスは落ちたとして扱う
  - 必要なら新規プロセスを立ち上げ，状態をコピー
  - 第3回で概説したGMSなど（詳細はいずれの教科書にもあり）

演習6－1：LOCK操作の実現

- READ-LOCKの実現方式

  （取得操作があるわけではなく暗黙的に対応）

  - 各プロセスはREAD操作メッセージを受け取ったら，その対象へのWRITE-LOCKが発行されていなければ，READ-LOCKを発行する
  - そしてそのままREAD操作を実行し，完了後READ-LOCKを解除する

  （たいてい一時ファイルに書き込み最後にコピー，などできるので，READ-LOCKはいらないことも）

演習6－1：LOCK操作の実現

- (WRITE-)LOCK操作の実現方式1
  - WRITE-LOCKを取得しようとするプロセスは全プロセスにLOCKリクエストをマルチキャストする
  - 各プロセスは下記の条件が満たされたら，LOCKリクエストを受け取った順に許可メッセージを返す
    - その対象へのREAD-LOCKが発行中でない
    - その対象へのWRITE-LOCKを許可したがUNLOCKメッセージを受け取っていない，という状態でない
  - 各プロセスはすべてのプロセスから許可を受け取ったなら，必要な処理を行い，UNLOCKメッセージを全プロセスにマルチキャストする

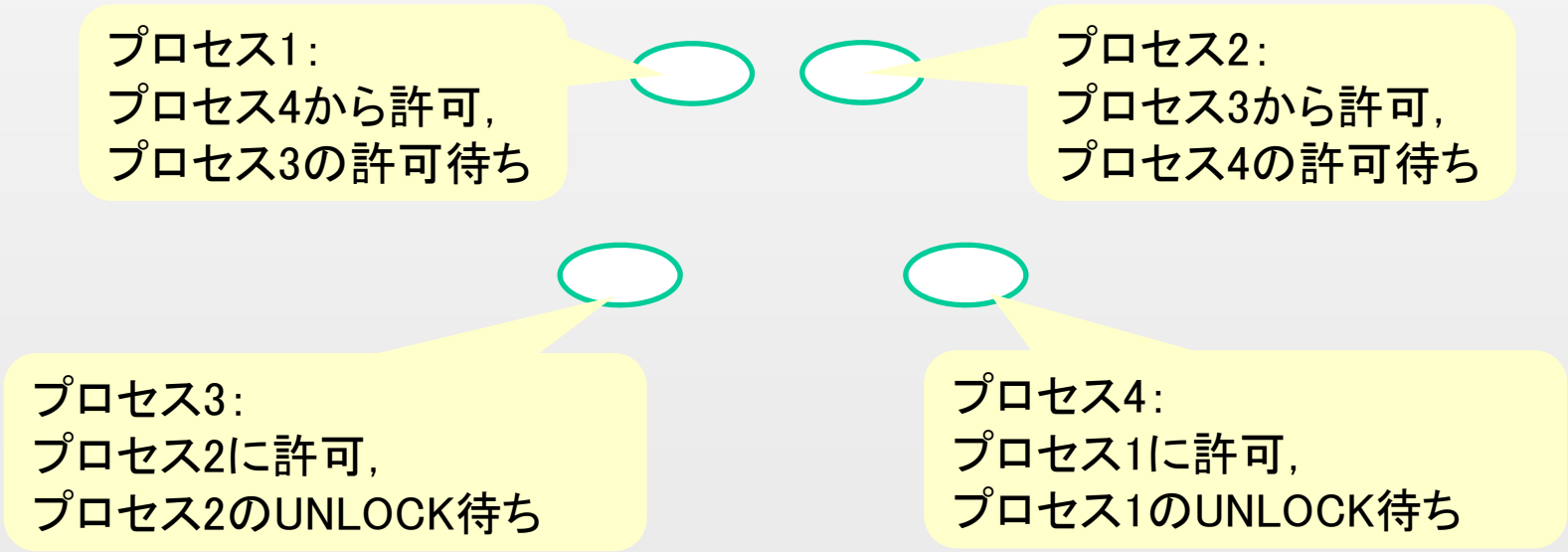
演習6－1：LOCK操作の実現

(WRITE−)LOCK操作の実現方式1において，

- LOCKリクエストの送信には全順序マルチキャストを用いる必要があることを確認せよ
  - 全順序マルチキャストを用いない場合に，不適切な状況に陥るような例や理由を挙げてみよ

演習6－1：解答例

- 複数のLOCKリクエストが，プロセスによって異なる順序で届いてしまう場合に，どのプロセスもロックを取得できないことがありうる



演習6－1：補足

- ロックを取得している間のUPDATE操作のメッセージが，UNLOCKのメッセージより後に到達すると意味がない
  - 同一プロセスからのメッセージなので，FIFOで届いていればよい

演習6－2：LOCK操作の実現（改）

- (WRITE-)LOCK操作の実現方式2
  - WRITE-LOCKを取得しようとするプロセスは全プロセスにLOCKリクエストをマルチキャストする
  - 各プロセスはすべてのプロセスから許可を受け取ったなら，必要な処理を行い，「受け渡し可能」状態として待機する
    （立ち上げ時には「受け渡し可能」状態である
    初期プロセスを1つ選んでおく）

（次頁に続く）

演習6－2：LOCK操作の実現（改）

- (WRITE-)LOCK操作の実現方式2（続）
  - 各プロセスはリクエストを受け取った順にキューに入れておく
  - 「受け渡し可能」状態のプロセスは，キューの先頭にある（自身にとって最も古い）LOCKリクエストに対し，許可メッセージをマルチキャストする
  - 許可メッセージのマルチキャストを受け取った各プロセスは，その対象へのREAD-LOCKが発行中でなければ，許可メッセージを該当プロセスに送る

演習6－2：LOCK操作の実現（改）

(WRITE−)LOCK操作の実現方式2において，

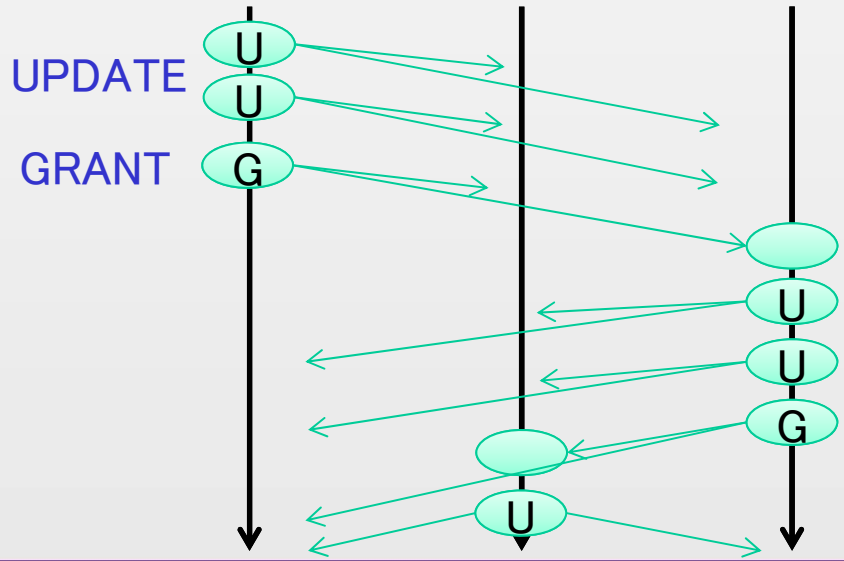
- LOCKリクエストの送信および，LOCKを確保している間のUPDATE操作において，因果順序マルチキャストを用いれば十分であることを確認せよ
  （全順序マルチキャストを用いなくてよい）
  - LOCKリクエストが各プロセスに異なる順序で届いても，実現方式1のときのように不適切な状況に陥らないことを確認せよ
  - UPDATE操作も因果順序マルチキャストに従うべきであることを確認せよ

演習6－2：解答例

- 実現方式2では，LOCKを保持して「受け渡し可能」状態にあるプロセス1つだけが，次にLOCKを取得するプロセスを指定する
  - 他のプロセスはその指定に従い，許可を出すだけ
  - 実現方式1のように，全プロセスが同じように「次のプロセス」を認識する必要はない

演習6－2：解答例

- 実現方式2では，因果順序により以下の順序は全プロセスで共有される
  - ノードNiがロックを取得して行ったUPDATE
  - その後の，ノードNiからノードNjへの許可

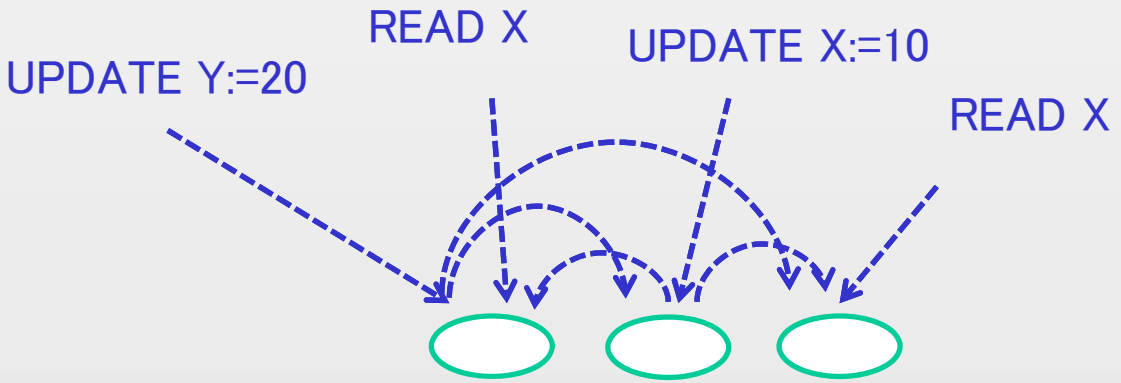


目次

- 演習：順序制御の必要性
- **演習：故障への対応**
- これまでのまとめ
- 演習：アプリケーションとの照らし合わせ

方針確認：UPDATE操作の実現

- UPDATE操作の実現方式
  - READ操作は，適当に分散して，いずれか1つのプロセスから読ませることにする
  - UPDATE操作を，全複製プロセスに対するマルチキャストとして実現する
    （マルチキャストの種類を指定可能とする）



演習6－3：ASYNC-UPDATE操作の実現

- ASYNC-UPDATE操作の実現方式
  - UPDATE操作をアプリケーションプロセスから受け取った複製プロセス1つが，アプリケーションをブロックせずに，他の複製プロセスへの反映を行う
    - 自身が持つデータに反映し，すぐ操作にアプリケーションに応答を返す（非同期）
    - マルチキャストでUPDATEのリクエストを全複製プロセスに送信する
    - 各プロセスにおける更新反映はメモリ上で行うことを想定する

演習6－3：ASYNC-UPDATE操作の実現

- ASYNC-UPDATE操作における耐故障実現方式
  - UPDATEのマルチキャストを行う複製プロセスは，下記の到達確認を行う
    - TCPなどを用い到達確認ができるようにする
    - 全プロセスへの到達が確認できたら，そのことをマルチキャスト
  - 各プロセスは受け取ったメッセージを保存しておく
    - タイムアウトした場合，上記プロセスの代わりに再度マルチキャストを行い，到達確認を行う
      （二重配信しないようにメッセージIDチェックもする）
    - 全プロセスへ到達が確認できたらメッセージ削除

演習6－3：ASYNC-UPDATE操作の実現

ASYNC-UPDATE操作の実現方式において，

- 障害のために，下記のような問題が発生する可能性があるかどうか検討せよ
  - 一部のプロセスにはマルチキャストが受信され，一部のプロセスには受信されない（非atomic）
  - 操作の効果が，システム内で（生きているどの複製にも）一切残らないままとなる（非durable）
- この問題が発生しない（atomic，durableである）ことを保証するような，別のUPDATE操作の実現方針を簡単に説明せよ

演習6－3：解答例

- 下記のような場合
  - 最初にマルチキャストを行うプロセスが，その途中でクラッシュ
  - 一部のプロセスには届き，反映やそれに基づいた処理が行われるが，それらも（タイムアウトして再送を試みる前に）クラッシュ

演習6－3：解答例

- Atomicity/Durabilityの実現：
  2PCのようなアプローチ
  - 二次記憶などを用いクラッシュ後にも状況を認識
  - 全体の確認が取れるまで反映しない
    （障害などがある場合，皆が反映しない）
  - 全体ではなく特定のQuorum値や過半数のプロセスに確認をとるようにしてもよいが，READにおいても特定数のプロセスへの確認が必要になる

Durability実現のコスト

- 教科書「Guide to …」によれば，いい実装で，
  - ASYNC-UPDATEは8.5万/sec，
    READは数百万/sec
  - Quorum READ/UPDATEは合計で100程度/sec

FLUSH操作

- FLUSH操作を提供するという選択も多い
  - 自身が送った，または受け取ったマルチキャストが，全プロセスに届いていることが確認できるまで待つ（届いていなければ届ける）
  - これまでに起こったマルチキャストが行き渡るまで待つ（行き渡らせる）
  - 必要な際にアプリケーションが明示的に呼び出す
  - メッセージ配信が順調ならほとんどブロックしない
  - 複数マルチキャストに対し，まとめて行うこともできる（個々に行うより効率的）

目次

- 演習：順序制御の必要性
- 演習：故障への対応
- これまでのまとめ
- 演習：アプリケーションとの照らし合わせ

ここまでで準備できた（できそうな）API

- READ
  - どの複製1個から読んでもよい
- FLUSH + READ
  - これまでのUPDATEが反映して最新の値になってからの読み出し
- （FLUSH +）SEARCH
  - 各複製に異なる部分を担当させての検索も可能

ここまでで準備できた(できそうな)API

- ASYNC-UPDATE
  - ブロックしない更新
  - ほとんどの場合，生きている全複製に反映される
  - 順序制御の追加も可能
    - FIFO：同じアプリケーションプロセスからのUPDATEはその順に処理される（シーケンス番号を用いて実現）
    - 因果順序：同じアプリケーションプロセスからのUPDATEや，送受信イベントのつながりから前後関係が決まるUPDATEは，その順に処理される（論理クロックを用いて実現）

ここまでで準備できた(できそうな)API

- ASYNC-UPDATE + FLUSH
  - 結果が全複製に反映されるのを待つUPDATE
- TOTAL_ORDER-UPDATE
  - 全複製が同じ順序で受け取るようなUPDATE
    (調整者を設けるなどして実現)
- ATOMIC-DURABLE-UPDATE(TOTAL_ORDER)
  - クラッシュと再起動があっても，全複製に持続的に反映されるか，いずれにも反映されないかというUPDATE
  - 自然と全順序になる

ここまでで準備できた(できそうな)API

- LOCK, UNLOCK
  - あるデータに対するUPDATEの排他的な制御
- CONDITIONAL-XXX-UPDATE
  - データの値を確認して，ある条件を満たせば書き換える，ということを原子的に行う
    (間で他のUPDATEが割り込まないようにする)
  - 条件を満たさない場合はその内容を報告する
- その他全体に対する合計や関係組み合わせの演算など

ここまでで準備できた(できそうな)API

- Key-Valueの話はしていない

  (データの中身を，ファイルなどではなくKey-Valueにするかどうかだが，今までの議論は実装非依存)

  - 性能・スケーラビリティが大きく上がる
  - 基本はREADと順序なしASYNC-UPDATEのみ
    - 一定時間内に値が全複製に反映
    - CONDITIONAL-ASYNC-UPDATEもたいてい可能

  一貫性など機能的な(クライアントなどの体験の観点からの)正しさの議論に今は集中

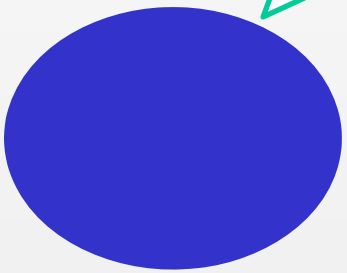
アーキテクチャ

- ありがちなクラウドの層分け（再）

いかに負荷を
前で吸収できるか？

First Tier
リクエスト処理を
行うWebページ

Second Tier
スケーラブルな
Key-Valueデータ



Inner Tiers/Back-end
データベースや
インデックス，
バッチ分析処理など

「とにかくレスポンスを速く，大量に」
（Consistencyを犠牲に？）という議論の対象

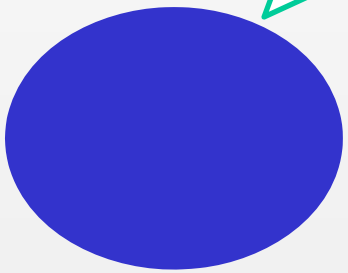
アーキテクチャ

- 多くの場合にあり得る
  これまで通りの選択

First Tier
リクエスト処理を
行うWebページ

Second Tier
スケーラブルな
Key-Valueデータ

いかに負荷を
前で吸収できるか？



Inner Tiers/Back-end
データベースや
インデックス，
バッチ分析処理など

Consistencyを検討するかも
Second Tierは必要ないかも

目次

- 演習：順序制御の必要性
- 演習：故障への対応
- これまでのまとめ
- アプリケーションとの照らし合わせ

演習6－4：アプリケーションの議論

- 周りにある様々なアプリケーションにおいて，これまでのAPIをどう選び利用するか検討せよ
  - 常識的に or 自身の要求に基づいて検討する
  - 「軽い，速い」選択だと，一貫性などの観点で不適切な状況が発生しそうかどうかを主に検討する
    - 因果順序のASYNC-UPDATEでも遅いから，順序を付けない，といった判断は具体的にはできない
    - 発生するとしても，受け入れてよいのでは，アプリケーション側で対応できるのでは，という考察歓迎

演習6－4：アプリケーションの議論

- アプリケーション例
  - イベントの抽選登録：期間内に抽選登録をしたユーザの中から，一定数が当選となる
    - READ：登録済みユーザが自分の申込有無を取得する，管理者が申込をしたユーザ一覧を取得する
    - UPDATE：ユーザが自身の抽選登録を行う，ユーザが自身の抽選登録をキャンセルする
  - 抽選ではなく「早い者勝ち」で登録，もありうる

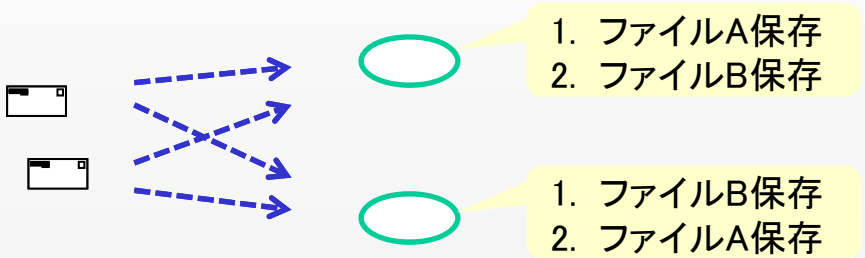
※ 例：どんな一貫性が必要か？

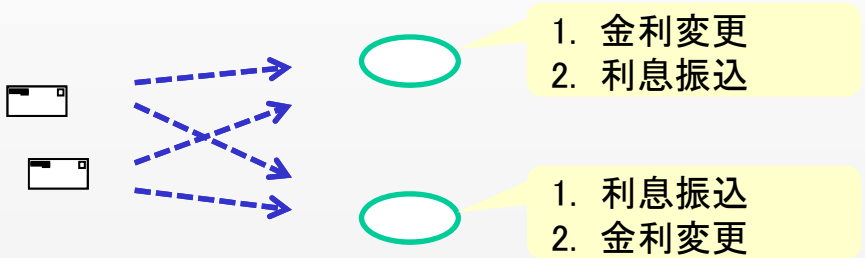
演習6－4：アプリケーションの議論

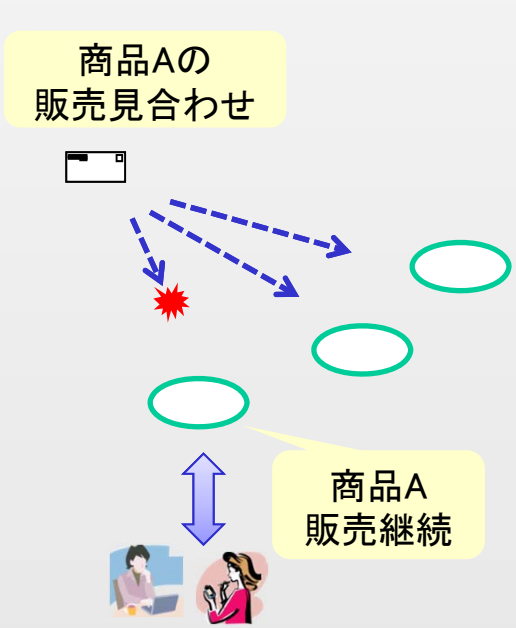
- アプリケーション例
  - オンラインファイル：複数のユーザがそれぞれ自身のファイルや共有のファイルを追加，読み書き，削除できる
    - READ：自身または共有のファイルの中身を読む
    - UPDATE：自身または共有のファイルを追加する，ファイルの中身に追記する，ファイルの中身を置き換える，ファイルを削除する

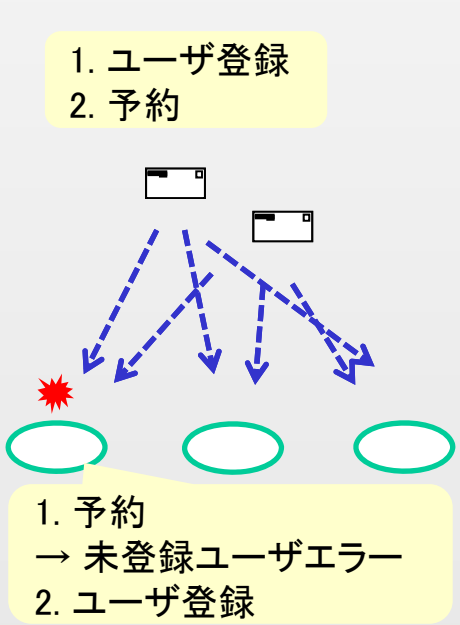
※ 例：個人ファイルへのアクセスは，共有ファイルへのアクセスよりも速くできるのでは

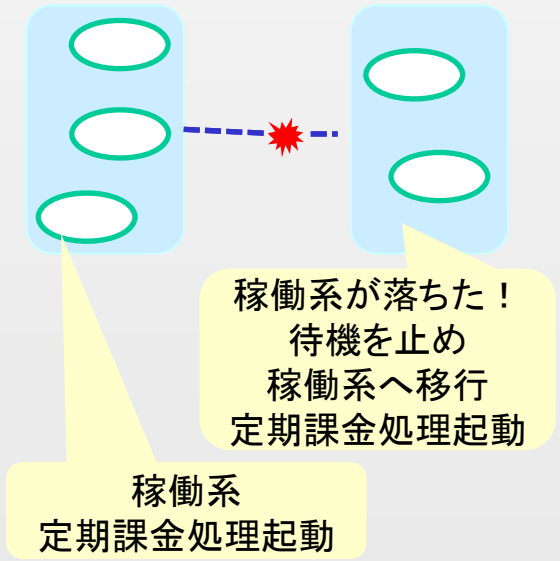
演習6－4：これまで挙げた例（断片）











演習6－4：アーキテクチャに関する問い

- First Tierだけで高速にレスポンス（結果のWebページ作成など）をしてしまうとどうなる？
  - Second TierまたはBackendにリクエストを投げないで
  - または，リクエストを投げるが完了を待たずに
- その選択の方がよさそうなほどのアクセス量，データ量がある例になっている？
  - そうでない場合，そういうものはありそうだろうか（普段自分たちが作らないものも含めて）

演習6－4：アーキテクチャに関する問い

- 二次記憶なしのFirst Tierだけで高速にレスポンスをしてしまうとどうなる？
  - Second TierまたはBackendにリクエストを投げないで
  - またはリクエストを投げるが完了確認を待たずに
  - それらの選択の方がよさそうなほどのアクセス量，データ量がある例は？
- 負荷分散の振り分けロジックなどもうまく定め，マルチキャストは軽くして整合性を確保することができることもある（そういう例はあるだろうか？）

演習6－4：解答例（簡単に）

- 抽選登録
  - 順序の必要性やユーザ間の依存性がない
  - アクセスが多いなら，First-TierでASYNC-UPDATEして速く返事してしまい，裏でRDBなどに登録してもよいかも（idempotent，再送）
    - ASYNC-UPDATEがDurableでない例外は，ちゃんとしたデータセンターなら滅多に起きない？
  - RDBへの登録をして，それをもってユーザに返事する，という作りでも十分そう
    - ユーザも急ぐ必要がなく殺到しない？

演習6－4：解答例（簡単に）

- オンラインファイル
  - 個人ファイルは競合がないように，個人のIDでハッシュして振り分け，そのノードからの順序だけが守られるFIFOでよい
  - 共有ファイルが競合的に複数のユーザにより更新される場合，Wikiのように絶対時間をもって扱うなど，アプリケーション上で解決してもよい
    - CONDITIONAL-UPDATEなどを用いる
    - 複数の絶対クロックのぶれには注意する（人間が気にする精度で因果関係と矛盾する，複数ノード間の時計ずれはどれだけ起きる？）