# 分散システム基礎とクラウドでの活用

# 第2回：基本的なプロトコル

国立情報学研究所
石川　冬樹
f-ishikawa@nii.ac.jp

# 今回の内容

- 同期や一貫性，耐故障性における基本的な考え方を確認する
  - 簡単な例題を通して，定義や分析，議論の進め方自体を確認する
  - 広く用いられているプロトコルである，Two-Phase Commitment，特に障害への対応について議論する

# 目次

- 基本例題の議論
- 考え方の確認
- 分散コミット

# 基本例題：概要

- サーバにある処理を依頼する（RPC）
  - プロトコル例
    - クライアントはリクエストをサーバに送信
    - サーバは処理を実行
    - サーバは確認メッセージを送り返す
  - 問題：サーバがクラッシュするとどうなる？
    - 後に復帰し，復帰した旨を伝える
    - まず，メッセージ配信は成功すると仮定する

# 基本例題：クラッシュの影響

- クラッシュが起きるタイミングごとに，サーバ側で起きることを考えてみる
  - クラッシュ（未印刷，未確認）
  - 印刷 → クラッシュ（未確認）
  - 印刷 → 確認メッセージ → クラッシュ
- クライアントから見ると，1個目と2個目の状況は確認メッセージが来ないという同じ状況に見える
  - 再送すると，2個目では2回印刷してしまう
  - 再送しないと，1個目では印刷されない

# 基本例題：対応方針

- 確認できない場合，メッセージを再送し，再依頼する（あきらめる，よりは）
- サーバ側には，（処理が完了していても）二回以上同じメッセージが到達する可能性がある
  - ミドルウェア・フレームワーク側で重複受信を判定し，アプリケーションには高々1回しかメッセージを渡らないようにする
  - 何度実行されても結果が同じとなる（idempotent，冪等）処理であることを確認しておく
    - 例：送付された情報を指定ファイルに書き込む

# 基本例題：ネットワークの考慮

- メッセージ配信の失敗を考えると，さらにややこしくなる
  - 確認がないことが，サーバのクラッシュなのか，ネットワークの問題なのか，一般にはクライアント側からはわからない
  - 処理が終わったかどうかを知ることができないという点はもとよりそうで，対応方針は同じとなる

# 目次

- 基本例題の議論
- 考え方の確認
- 分散コミット

# 考え方：プロトコルの再利用

- 同期や一貫性，耐故障性の実現に関する，既知の設計が多く提供されている
  - 各ノード（プロセス）がとるべき振る舞い（約束事）を定めたプロトコルの形式が多い
    - 「（手順を定めた）アルゴリズム」，「（実装の基になる）スケルトン」などとも見なせる
  - 設計のうちある側面を抜き出し，実装非依存な形で，一般的に再利用可能としたもの
  - 多くの場合，ミドルウェア・フレームワーク側が実装，提供すべき基盤技術
    （ただし，使うだけの立場でも理解は必要）

# 考え方：プロトコルとしての再利用（例）

**クライアント：**
あらかじめ合意された識別子を
含めてリクエストを送信

Ackを待つ

受け取った
　=> 処理成功として終了
タイムアウトした
　=> 処理結果不明として終了

**サーバ：**
リクエストを待つ

識別子を確認し，
処理済みと記録されている
　=> Ackを送信
処理済みと記録されていない
　=> 処理を行い，
　　処理済みと記録し，
　　Ackを送信

# 考え方：プロトコルとしての再利用

- 「知見」としてのプロトコルは一般的，抽象的に定められていることが多い
  - 対象問題に対し，詳細はうまく決められる（ケースバイケースで決める必要がある），と仮定した上での定義になっている
    - 例：ネットワークレイヤにおける通信方法は特に規定しない
    - 例：通信失敗の判断（タイムアウトの長さやその他の検出方法）については，必要なら実装で定める
    - 例：他の大きなプロトコルでは，このRPCのようなプロトコルを，適宜用いることと仮定される

# 考え方：正しさに関する議論

- 基本的には，数学的な証明という形で，成り立つ性質（正しさもしくは限界）が議論されている
  - 障害の発生タイミングなど様々な場合分けの基で，（しばしば形式言語を用いて）理屈により議論，保証されている
- こういった厳密な議論では，結果を保証するためには仮定が必要となる

  （「・・・ならば，・・・を保証できる」という言い回し）

# 考え方：正当性に関する議論（例）

- 先のサーバ呼び出しにおいて成り立つ性質
  - クライアントがackを受け取った場合，サーバの処理が完了したとクライアントは正しく断言することができる（なぜなら，サーバがackを送るのは処理が終わった後だけだから）
  - クライアントがackを受け取らなかった場合にクライアントは，サーバの処理が完了していないと断言することはできない
    （なぜなら，処理が完了してもその後クラッシュや，ack配信失敗が起きる状況がありうるため）

# 考え方：正当性と妥当性・有用性（１）

- 自明なことはわざわざ議論しない
  （しばしば仮定も暗黙的）
  - 「もしも最初からサーバが落ちていてずっと復帰しなかったら・・・」（どうしようもない！）
- 正当性があればそれでよいというわけではない
  - オーバーヘッドなど他の品質側面から，不十分かもしれない
  - うまくいくための仮定が強すぎて，現実の環境には適用できないかもしれない

# 考え方：正当性と妥当性・有用性（２）

- 本当に達成したい性質は，「必ず成り立つ」という形式では保証できないことが多い
  - 例：「クライアントのリクエストに対し，サーバはいつか必ず処理を行う」
- が，「成り立たない状況がありえる」のであっても，実用上困らないことも多い
  - 例：「通常の運用状況では十分に低い確率でしか起きない」，「例外的な障害状況では，システム全体として別途特殊な対応を定義する」といった議論のみ可能で，それで十分たりうる

# 目次

- 基本例題の議論
- 考え方の確認
- 分散コミット

# コミット問題：概要

- 複数のプロセスすべてが，ある動作を行うか否かについて合意できるようにしたい
  - 一部のプロセスはその動作を行うことができない可能性がある
- 参加するプロセスのすべてが，動作を行えるかどうかをまず確認することが必要
  - すべてのプロセスが行えることが確認できるならば，commit（約束する）
  - 1個以上のプロセスが行えなかったり，連絡が付かなかったりするならば，abort（中断する）

# コミット問題：障害への対処

- 前述の性質を保証できるために必要な仮定：

  各プロセスは，動作を行えると確認したならば，もしもクラッシュし，後に復帰したとしても，その動作を行うことができる

  - 例：永続的な（クラッシュで失われない）二次記憶領域など（の一時領域）に書き込んでから，動作を行えると宣言する

# ACID

- Atomic（原子性）：外部の世界から見て不可分（途中過程や部分的に実施された状態は見えない）
- Consistent（一貫性）：不変条件を満たす
- Isolated（孤立性）：複数のトランザクションは互いに干渉しない
- Durable（耐久性）：一度確定（コミット）すると，その変更は永続的に反映される

# コミット問題：プロトコル

- 通常，動作を依頼し，合意のためのやりとりを行う調整者（coordinator）がいることを想定する
- 調整者と各参加者がとる振る舞いを規定したプロトコルがよく知られている
  - 2PC: Two-Phase Commitment Protocol
  - 3PC: Three-Phase Commitment Protocol
  - 他のプロトコルに，同様の考え方を埋め込むことも多い
- 以降プロトコルに従った一回一回の振る舞いの集まりを「実行」と呼ぶ（「インスタンス」がよい？）

# 2PC：プロトコルの基本的な流れ

1. 調整者はプロトコルの実行開始を呼びかけ
   - 識別子，動作内容，参加者リスト
2. 各参加者はcommitまたはabortに投票
   - commitの場合は前述の通り，そのことを永続的に記録
3. 調整者は，すべての参加者からの投票が集まればそれに基づいて決定，もしくはタイムアウトし，決定を各参加者に送信
   - 全参加者がcommitならばcommitと決定，送信
   - それ以外の場合abortと決定，送信

# 2PC：基本版

**調整者：**
マルチキャスト *ok to commit?*

返信を集める
　全参加者 *ok*
　　=> 送信 *commit*
　それ以外
　　=> 送信 *abort*

**参加者：**
*ok to commit* =>
　変更を一時領域に保存，
　返信 *ok*

*commit* 受信 =>
　一時変更を永続的に反映
*abort* 受信 =>
　一時変更を削除

# 2PC：要考慮事項

- 調整者および各参加者の，障害時の振る舞い
  - 様々な状態での障害を検討する必要がある
- 調整者および各参加者が，一時的に保持する情報の保持期間
  - 「ごみ」が無限に溜まっていってはならない
  - 適当に消してしまい，commitに投票したのに動作を行わない，といったことがあってはならない

# 2PC：参加者の障害(１)

- 初期状態での障害
  - 投票依頼を受け取らず，調整者はabortに決定
- commit準備ができ投票した状態での障害
  - 復帰後プロトコルの結果（調整者による決定）を知る必要がある
- commit/abort処理を行う状態での障害
  - 復帰後commit/abort処理を完了する必要がある（idempotentであるようにし，何度でも行う）

下2つの場合，他の処理をブロックすべきことも

# 2PC：参加者の障害(２)

- 実装での対応の例
  - Idempotentな方法としてファイルコピーを利用
  - 永続的な二次記憶などに，プロトコルの各実行に関するログを記録
  - 成功したときだけログを削除
- 調整者が落ちないと状況を絞って考えると，
  - 結果は調整者に問い合わせればよい
  - 各参加者は完了後にackを送るようにする
  - すべてのackが集まるまでは，調整者はその実行に関する情報を保持するあd

# 2PC：参加者障害対応版

**調整者：**
マルチキャスト *ok to commit?*

返信を集める
　全参加者 *ok* =>
　　送信 *commit*，結果をログ
　それ以外 =>
　　送信 *abort*

各参加者からのackを集める
情報を削除する

**参加者：**
*ok to commit* =>
　変更を一時領域に保存，
　返信 *ok*

*commit* 受信 =>
　一時変更を永続的に反映
*abort* 受信 =>
　一時変更を削除

障害からの復帰時 =>
　調整者に結果を問い合わせ，
　応じて上記のcommit/abort処理

# 2PC：調整者の障害

- 調整者が復帰するまで参加者が待つのであれば，修正点は少なくて済む
  - 決定（commitまたはabort）を送る前に，永続的な二次記憶などにその決定を記録する
  - 障害から復帰した際にはそれを参加者に送る
  - 参加者は，二回目以降の結果受信に対してはackを返すだけで動作は実行しない
  - 参加者皆が処理を終えたらその決定ログを消す

※ 参加者はプロトコル実行が終わるまで，依存関係がある行動を止めるべきで，オーバーヘッド大

# 2PC：調整者障害対応版

**調整者：**
マルチキャスト *ok to commit?*
返信を集める
　全参加者ok =>
　　決定を永続領域にログ，
　　送信commit
　それ以外 =>
　　送信abort
各参加者からのackを集める
情報を削除する

障害からの復帰時 =>
　結果を再送し，上記のack集め，
　および情報削除を行う

**参加者：**
*ok to commit* =>
　変更を一時領域に保存，
　返信ok
commit 初回受信 =>
　一時変更を永続的に反映，ack
abort 初回受信 =>
　一時変更を削除，ack
二回目以降の受信 =>
　ack

障害からの復帰時 =>
　調整者に結果を問い合わせ，
　応じて上記のcommit/abort処理

# 2PC：調整者の障害（より積極的な対応）

- 参加者の結果待ちにタイムアウトを設け，他の参加者に問い合わせるようにする
  - もしも調整者からの結果が届いていれば結果を知ることができる
  - 他の参加者全員の投票が把握できれば，結果を
  - 決めることができる

※ ただし，復帰を待つ必要がある状況はありうる

- 他の参加者に問い合わせたところ，参加者の1つだけが応答なし．他は皆commitだと，その参加者または調整者が復帰するまで結果がわからない

# 2PC：調整者の障害（より積極的な対応）

- 参加者の結果待ちにタイムアウトを設け，他の参加者に問い合わせるようにする（続）
  - この方法がうまくいくためには，各参加者は自分の処理が終わっても結果を覚えておく必要がある
  - ➡ 調整者からのack（情報削除通知）を最後に追加
  - 処理を終えた参加者は，このackが一定時間来ない場合，結果を他の参加者に通知してもよい
  - 情報削除は，システム全体において複数のプロトコル実行分をまとめてたまに行うようにしてもよい（2PCを用いて！）

# 2PC：調整者障害積極対応版

**調整者：**
マルチキャスト *ok to commit?*
返信を集める
　全参加者ok =>
　　決定を永続領域にログ，
　　送信commit
　それ以外 =>
　　送信abort
各参加者からのackを集める

障害からの復帰時 =>
　結果を再送し，上記のack集める

システム全体で定期的に完了したプロトコル実行を問い合わせ，それらのログ削除を2PCを用い行う

**参加者：**
*ok to commit* =>
　変更を一時領域に保存，返信ok
commit 初回受信 =>
　一時変更を永続的に反映，ack
abort 初回受信 =>
　一時変更を削除，ack
二回目以降の受信 => ack
障害からの復帰時 =>
　調整者に結果を問い合わせ，
　応じて上記のcommit/abort処理

*ok to commit* 後にタイムアウト=>
　他の参加者に問い合わせ，
　可能ならcommit/abort決定，処理
　不可能なら待ち

# 3PC：概要

- 2PCでブロックが起きてしまう状況を回避したい
  - 調整者と一部の参加者が落ちているときに，生きている参加者は決定ができない
- 考え方
  - 2PC同様の投票をするが，すべての参加者が合意してもすぐにcommitを実行しない
  - 調整者はackを求め，生きている参加者すべてからackが来たらcommit決定，処理してしまう
    （落ちていた参加者は復帰後にcommit処理）

# 3PC：概要

**調整者：**
マルチキャスト *ok to commit?*
返信を集める
　全参加者 *ok*
　　=> ログ，送信 *precommit*
　それ以外
　　=> 送信 *abort*
障害がない参加者からの*precommit*に関するackを集める
　=> ログ，送信 *commit*

処理完了のackを集める
情報削除のプロトコルを実行

**参加者：**
*ok to commit* =>
　変更を一時領域に保存，返信 *ok*
*precommit* 受信 =>
　precommit状態へ，ack
*commit* 受信 =>
　一時変更を永続的に反映，ack
*abort* 受信 =>
　一時変更を削除，ack

障害からの復帰時 =>
　他の参加者に問い合わせ，皆が*precommit*状態か，一つでも*commit*済みの参加者がいれば*commit*
　そうでなければ*abort*

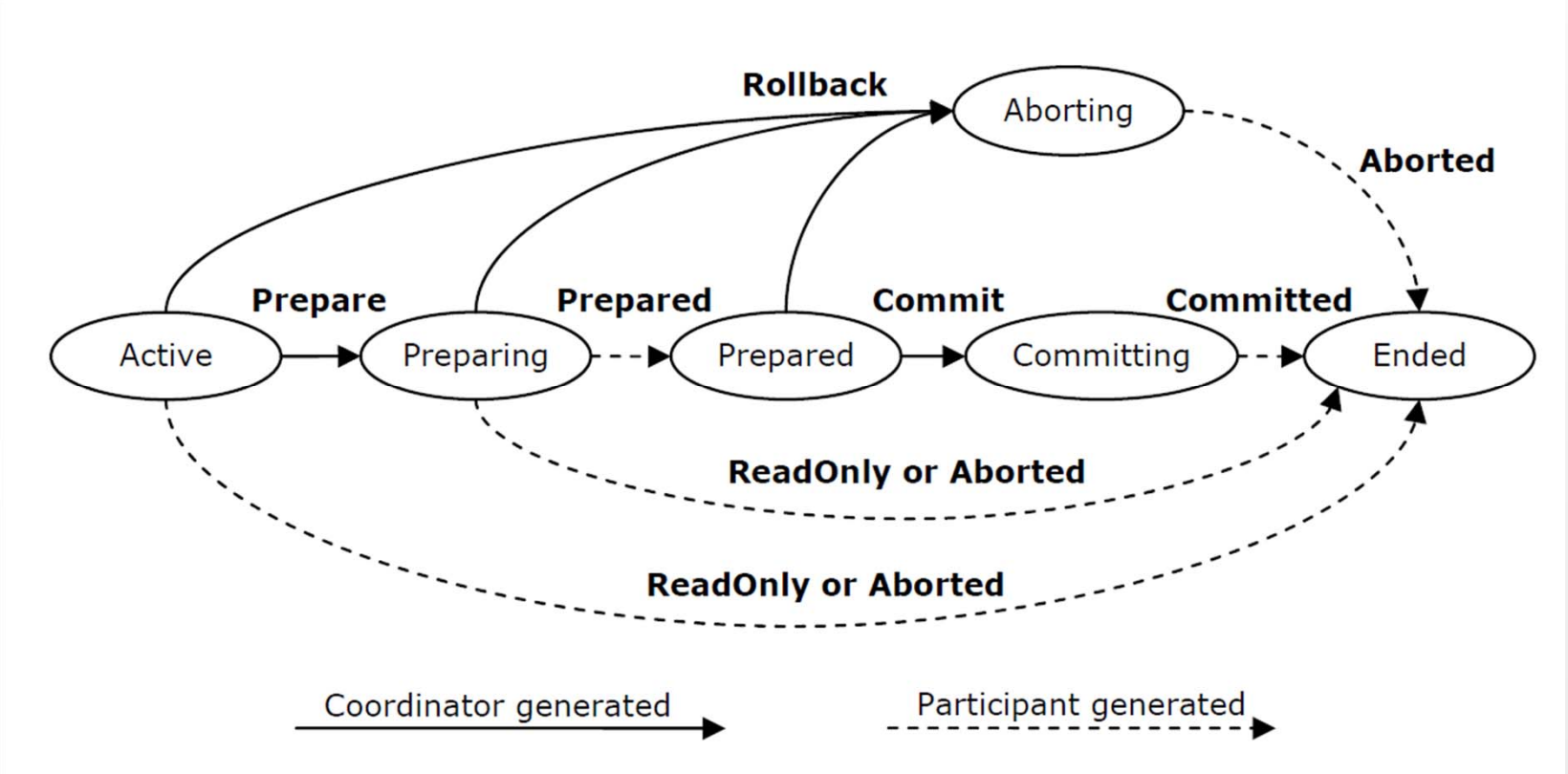
# 3PC：実用上の位置づけ

- 3PCは，「勉強になる」のでよく引用されるが，実際にはあまり用いられない
  - 2PCがブロックする状況が「ある」ことはその通りだが，限定的であり可能性が十分低い
- 3PCに関する証明などは，理論としてはおもしろい，かもしれない

# 補足：Webサービスのトランザクション仕様

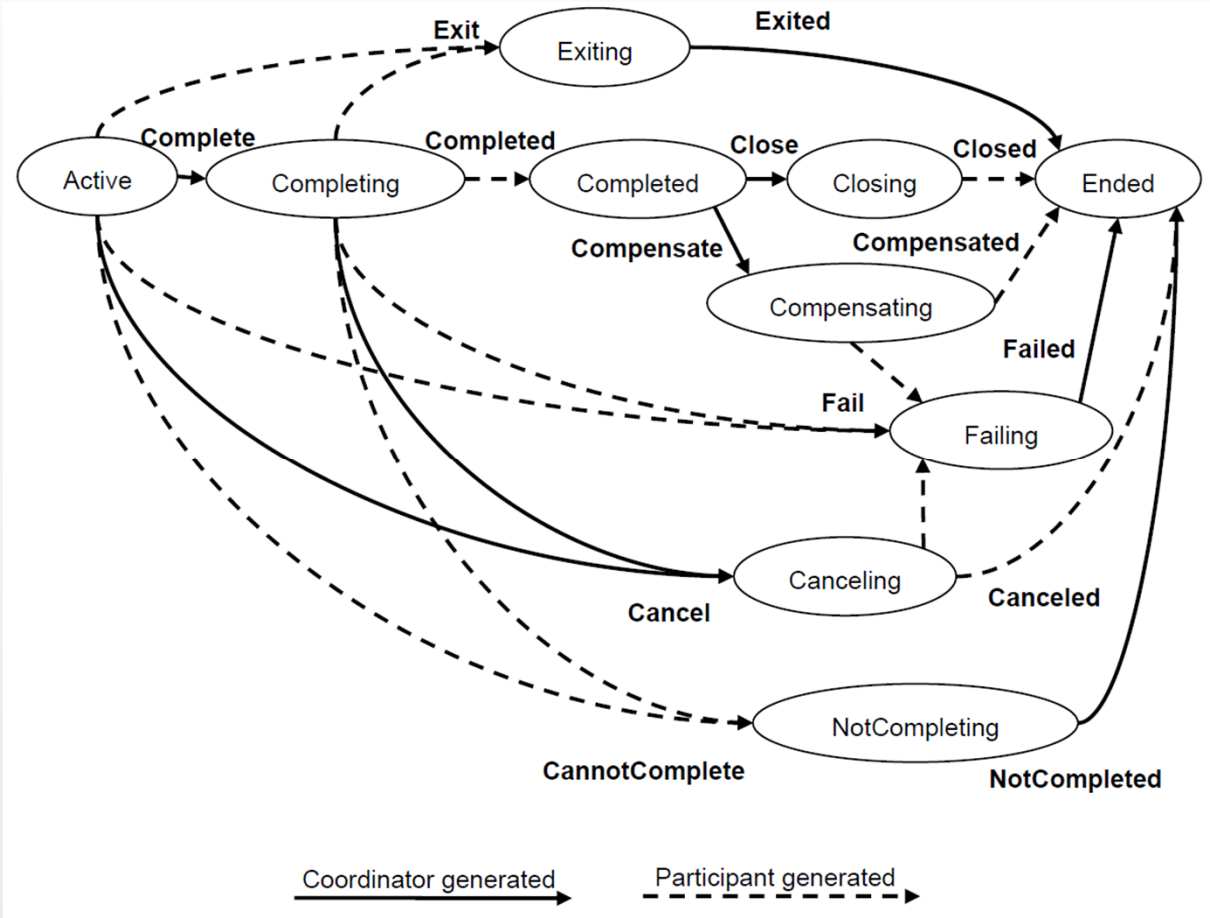
- WS-Coordination
  - プロトコルに依らず，トランザクションプロトコル実行に関する制御を行うための仕様
- WS-AtomicTransaction
  - 2PCの仕様
- WS-BusinessActivity
  - 組織をまたがるなど影響範囲が広い長期のトランザクションでは，参加者をブロックさせたくない
  - 各参加者は処理を完了してしまい，その後に必要なら補償（compensate，取り消し動作など）する

# 補足：Webサービスのトランザクション仕様




[WS-AtomicTransaction 1.1より]

# 補足：Webサービスのトランザクション仕様




[WS-BusinessActivity 1.1より]

# 今回のまとめ

- 同期や一貫性，耐故障性については，再利用可能なプロトコルという形で，実現のための知見が累積されている
  - ある仮定の基で，有用な性質が達成できることが証明されている
  - 保証できる性質，できない限界の明確化とは別に，実用上の意義，意味を議論することが重要
  - 「確認をとって確定する」というコミットプロトコルは典型的なものの1つ
- 次回：順序づけについて議論