クラウドシステム基礎

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，耐故障性に関する考え方を議論する
  - 基本的な（単純）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，特にその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

- サーバに印刷処理を依頼する（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
    - まずは，メッセージ配信は確実に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
- 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さないようにする
  - 何度実行されても結果が同じとなる冪等（べきとう，idempotent）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名のファイルに書き込む（ファイル名は固定で，追記ではない）

f(f(x))=f(x)

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，実際に起きうることの場合分けは増える
  - 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
  - しかし，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：

1. 識別子を含めてリクエストを送信

2. Ack受信を規定時間待つ
2.1 規定時間内に受け取った
  => 処理成功として終了
2.2 初回試行でありタイムアウトした
  => 最初に戻って再試行
2.3 再試行済みでありタイムアウトした
  => 処理結果不明として終了

サーバ：

1. リクエストを受信する

2. リクエスト内のタスク識別子に
  対する処理記録を確認する
2.1「処理済み」と記録されている
  => Ackを送信
2.2 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），と仮定して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，実装で適宜定める
    - 例：より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
- 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）

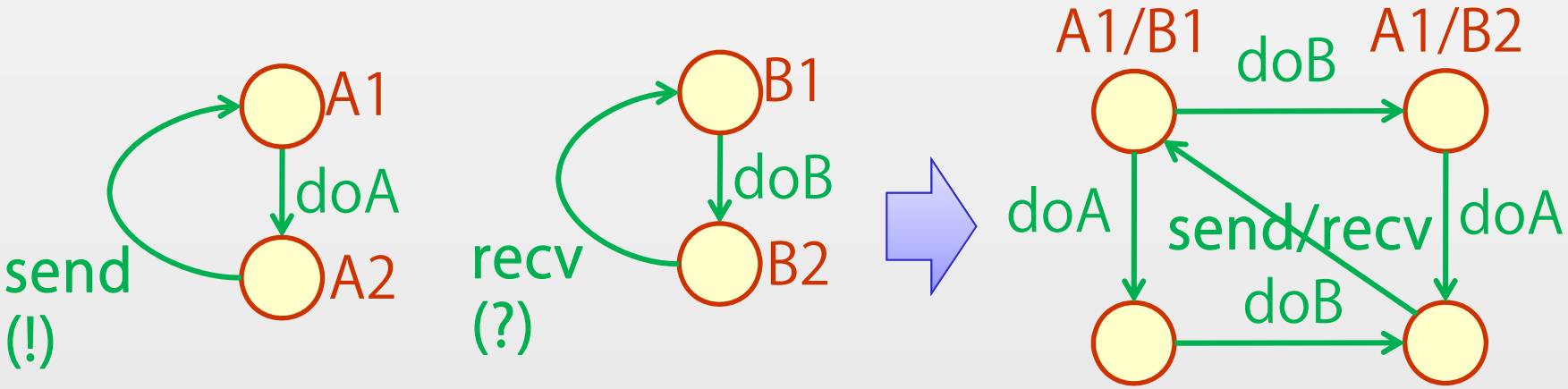
- 自明なことはわざわざ書かないことが多い（しばしば仮定も暗黙的）
  - 「もしも最初からサーバが落ちていてずっと復帰しなかったら・・・」（どうしようもない！）
- 正しく何か性質を示せればそれでよいというわけではない
  - オーバーヘッドなど他の品質側面から，不十分かもしれない
  - うまくいくための仮定が強すぎて，現実の環境には適用できないかもしれない

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」といった結果の意味を議論することで十分たりうる

補足＆デモ1：汎用SEツールでの分析例

- 付録の.pml.txtファイル：モデル検査器SPINを用いて分析する場合（「設計モデル検証」講義）
  - if/do内の :: では非決定的に起きうることを列挙している（doの場合はループする）
  - 並行システム全体としての振る舞いを合成



http://spinroot.com/spin/whatispin.html

補足＆デモ1：汎用SEツールでの分析例

- （続）
  - ltlから始まる行は検証したい項目
    （ある命題が真となるかどうか）
  - <>[] は，「最終的には・・・となって落ち着く（変化がなくなる）」と思っておけばよい

例（ファイル内p2）：サーバーが最終的に成功と記録しているならば，クライアントも成功と記録しているか？（どんな実行列になった場合でも必ず？）

補足＆デモ2：汎用SEツールでの分析例

- 付録の.pm.txtファイル：確率モデル検査器PRISMを用いて分析する場合
  - 起きうる状態遷移のモデルに対し，確率を付与し，ある性質を満たす確率を求める
  - 従来人工知能やOR（Operations Research）の技術と分類されることも多い技術だが，ここでは「確率モデル検査」を名乗るツールを用いている

http://www.prismmodelchecker.org/

補足＆デモ2：汎用SEツールでの分析例

- ネットワーク障害あり（10%の確率）版
  - 例：50%の確率で処理成功するが，その通知が10%の確率で届かない
    （タイムアウト・再依頼は1回）

    →「処理が成功しているが，クライアントがそのことを把握できない」確率は約1.3%

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，動作を行えることを宣言する

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が見えることはない）
- Consistent（一貫性）：（アプリケーションが定める）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

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:

1. マルチキャスト *commitok?*

2. 返信を待つ
2.1 全参加者から *ok* 受信
  => マルチキャスト *commit*
2.2 それ以外(タイムアウトも含む)
  => マルチキャスト *abort*

参加者:

1. *commitok?* 受信

2. 指定された動作が可能か確認
2.1 対応可能
  => 変更を一時領域に保存,
    返信 *ok*
2.2 対応不可能(以降ここは省略)
  => 返信 *ng*, 終了

3. 指示を受信
3.1 *commit* 受信
  => 一時変更を永続的に反映
3.2 *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
- 調整者や各参加者が，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
  - 適当に消してしまった結果，commitに投票したのに動作を行わないようなことがあってはならない
- 孤立性保証のために関連する他のトランザクションをブロックするため，性能に留意が必要

2PC：参加者のみの障害（１）

- 初期状態での障害
  - 投票依頼を受け取れない場合，調整者はabortに決定するということが起きる（これは重要でない）
- commit準備ができ投票した状態での障害
  - 投票済みであるため，復帰後に調整者による決定を知り，必要ならcommit処理を行う必要がある
- 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
  - 一時領域からのファイルコピーなら高速で冪等（障害時の状態を気にせず再試行可能）

2PC：参加者のみの障害（２）

- 参加者による障害からの復帰後処理（前頁）
- 調整者が落ちない状況に絞って考えると，復帰後に調整者に問い合わせればよい
  - 参加者は二次記憶にプロトコルの進捗状況を覚えるなどして，復帰後に必要な行動を判断できる必要がある（完了後に削除）
  - 調整者は，すべての参加者がプロトコルを完了するまで結果を覚えておく必要がある
  - 参加者は完了時に調整者に報告（到達確認付き），調整者は完了状況を把握

2PC: 参加者のみの障害への対応版

調整者:
1. マルチキャスト *commitok?*
2. 返信を待つ
2.1 全参加者から *ok* 受信
  => マルチキャスト *commit*
2.2 それ以外(タイムアウトも含む)
  => マルチキャスト *abort*
3. 参加者からの受信
3.1 投票結果問い合わせを受信
  => 投票結果を通知
3.2 完了通知を受信(全参加者揃わず)
  => 完了した参加者を記録
3.3 完了通知を受信(全参加者揃った)
  => 記録を削除し終了

参加者:
1. *commitok?* 受信
2. 変更を一時領域に保存,
  ログ1, 返信 *ok*
3. 指示を受信, ログ2
3.1 *commit* 受信
  => 一時変更を永続的に反映
3.2 *abort* 受信
  => 一時変更を削除
4. 完了通知を送信

復帰時(ログ1): 結果問い合わせ3へ
復帰時(ログ2): 3.1か3.2から再試行

演習1：課題

- 調整者の一時的な障害に対応できるように，プロトコルの変更を検討せよ
  - 重要な状況として，「投票が集まり決定を行った後に調整者がクラッシュ」という状況に焦点を当てて検討せよ
    - マルチキャストが「順次送信」である場合，一部の参加者にだけ結果が通知された時点でのクラッシュがある可能性も考慮せよ

（続きあり）

演習1：進め方

- グループ演習
  - 必要ならここまでのプロトコルを整理し，疑問・疑念，暗黙の仮定，妥当性・実用性の懸念などを出し合い，議論してもよい
  - 対応策の「思いつき」を出し合う
  - できたプロトコルを皆でレビューし，疑問・疑念，暗黙の仮定，妥当性・実用性の懸念などを出し合い，議論する

（次スライドにヒント）

演習1：ヒント

- ヒント
  - 調整者の復帰を待つようにすれば，確実
    （それでも復帰後のやりとりは決める必要がある）
  - 調整者を待たずに参加者だけで対応することが，
    どういう状況ならどうできるのか，検討してもよい
  - これまで出たものを使うかも
    - 障害があっても，復帰した後に障害前のことを思い出し，処理を再開するための方法
    - 確実に実行させるために同じメッセージが2回届いたとしても，重複処理を避ける方法

解説資料は別途配付

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ことがある
- 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:
1. マルチキャスト *commitok?*
2. 返信を待つ
2.1 全参加者から *ok* 受信
 => マルチキャスト *precommit*
2.2 それ以外(タイムアウトも含む)
 => マルチキャスト *abort*
3. 障害がない参加者からの
 precommit 受信通知を集める
 => 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:
1. *commitok?* 受信
2. 変更を一時領域に保存,
 返信 *ok*
3. 指示を受信
3.1 *precommit* 受信
 => precommit状態へ, 受信通知
3.2 *abort* 受信
 => 一時変更を削除, 受信通知, 終了
4. *commit* 受信 =>
 一時変更を永続的に反映, 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*

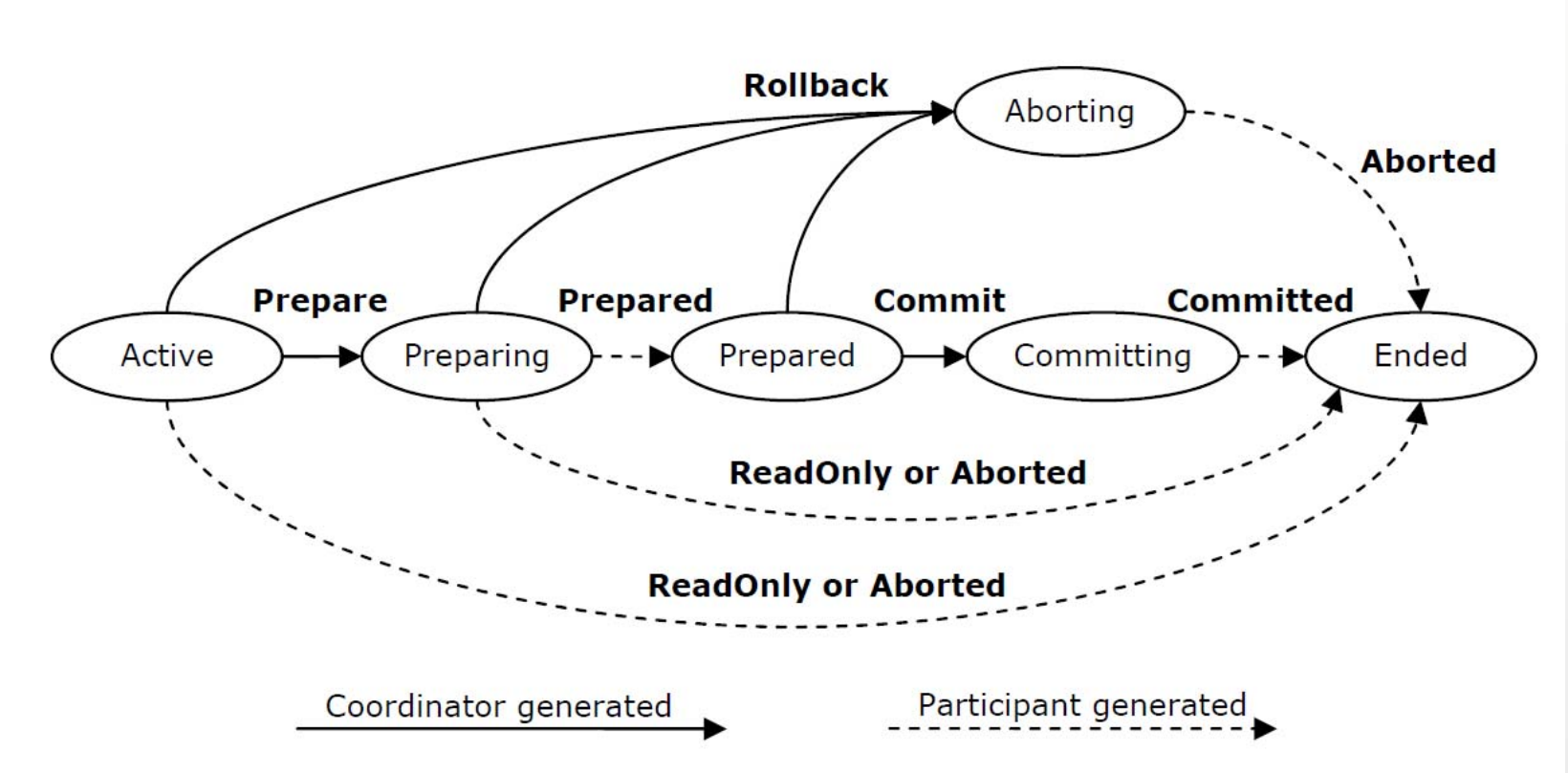
3PC：実用上の位置づけ

- 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，前述の通り限定的（可能性が低い）
  - 3PCはそれを避けている（他の参加者に確認したときに，判定不能ということがない）
  - メッセージ量を多くして複雑にすることに見合うか・・・？

補足：Webサービスのトランザクション仕様

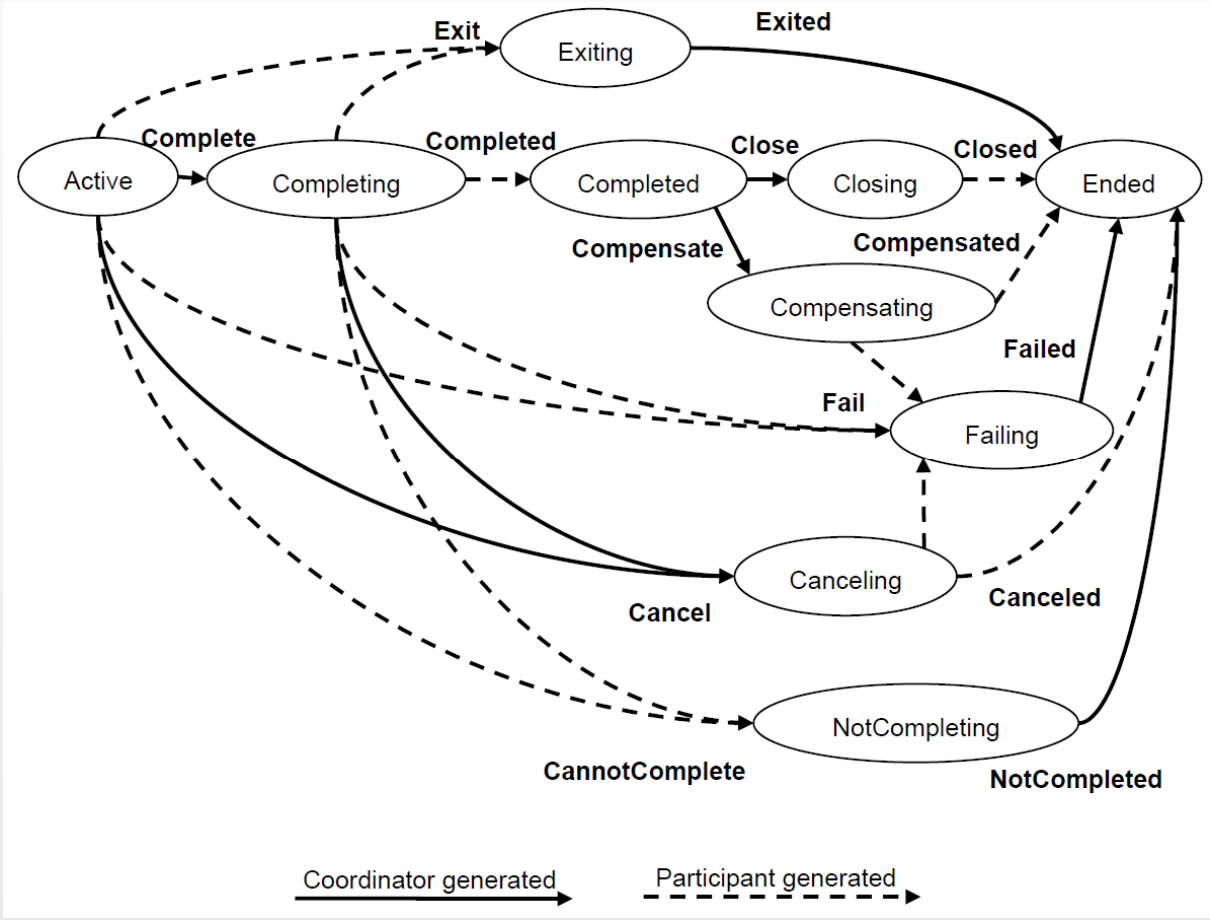
- WS-Coordination
  - プロトコルに依らず，トランザクションプロトコル実行に関する制御を行うための仕様
- WS-AtomicTransaction
  - 2PCの仕様
- WS-BusinessActivity
  - 組織をまたがるなど影響範囲が広い長期のトランザクションでは，参加者をブロックさせたくない
  - 各参加者は処理を完了してしまい，その後に必要なら補償（compensate，取り消し動作など）する

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]

楽観的アプローチ・悲観的アプローチ

- 悲観的アプローチ
  - 2PCやロックに基づいた排他制御など
  - 競合の発生を前提として，それらを避けるためのアプローチ（基本的にブロッキング）をとる
- 楽観的アプローチ
  - とにかく書き込んでみて，その際に競合が見つかったらあきらめる，あるいは競合や失敗が後でわかったらロールバックするなど
  - ブロッキングを避けることを重視する

今回のまとめ

- 同期や一貫性，耐故障性については，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
  - プロトコルとして，あるいは再利用できる典型的な「考え方」も存在する
- 次回：全プロセスで同一状態を維持しなくとも，全体として進み続けるための仕組みを議論する