フォールトトレラント性

教科書7章

フォールトトレラント性

- 分散システム: どこか一部分が故障しても、全体を停止させずに障害から回復できるように設計する必要　→ フォールトトレラント性（耐障害性）
- フォールトトレラント性の技術的側面を紹介
  - 背景
  - 耐障害性を持つプロセスの構築法
  - 高信頼マルチキャスト
  - 分散コミット
  - 障害からの回復

高信頼性

- 高信頼性（Dependability）とは、以下の要求を満たすこと
- 可用性（Availability）
  - システムを利用したいときに高い確率で利用可能
- 信頼性（Reliability）
  - ある程度の期間、正常に稼動
- 安全性（Safety）
  - 障害時に重大な問題が生じない
- 保守性（Maintainability）
  - 故障したシステムを容易に回復可能

基本概念
フォールトトレラント性

- フォールトトレラント性
  - システムに障害がおきてもサービスの提供を続けられる性質
- 障害(fault)の分類
  - 過渡障害（transient fault）
    - 一度だけ発生し、消滅する障害
      - ノイズによるビット損失など
  - 間欠障害（intermittent fault）
    - 一時的な障害が繰り返される障害
      - コネクタの接続不良など
  - 永久障害（permanent fault）
    - 修復されるまで存在し続ける障害
      - ディスククラッシュなど

障害モデル

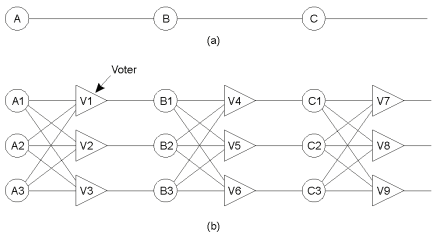
- さまざまなタイプの障害

障害の種類	説明
クラッシュ障害（crash failure）	サーバが停止、ただし、停止するまでは正常動作
欠落障害（Omission failure） *受信欠落（Receive omission）* *送信欠落（Send omission）*	サーバがリクエストへの応答に失敗 サーバがメッセージの受信に失敗 サーバがメッセージの送信に失敗
タイミング障害（Timing failure）	サーバが指定時間内に応答できない
応答障害（Response failure） *値障害（Value failure）* *状態遷移障害（State transition failure）*	サーバの応答が正しくない 応答の値が間違っている サーバが正しい制御フローから逸脱している
任意障害（Arbitrary failure） （又は、**ビザンチン障害**）	サーバが任意のタイミングで任意の応答を生成

冗長性による障害の隠蔽

- 障害に冗長性を持たせることで隠蔽
  - 情報的冗長性
    - 情報の冗長性
      - エラー訂正符号など
  - 時間的冗長性
    - 動作のやり直しによる冗長性
      - トランザクション処理など
  - 物理的冗長性
    - 余分なデバイスやプロセスを持たせる冗長性
      - RAIDなど

冗長性による障害の隠蔽



- 3重モジュールによる冗長性
  - A1,A2,A3のうち1つが故障しても耐えられる
  - B1,B2,B3のうち1つ、C1,C2,C3のうち1つが故障しても同様
  - 投票モジュールViの故障はその出力先モジュールの故障と同等

プロセスグループの設計課題

- プロセス障害の隠蔽への主要なアプローチ
  - グループの中に同一プロセスを複数作成
  - そのプロセスへのメッセージはグループメンバ全員が受信
    - → どれかが故障しても他のプロセスがタスクを継続可能

プロセスグループの設計課題

- プロセスグループ構築のアプローチ
  - 平坦なグループ(flat group)
    - 特別なプロセスなし，決定は合議
  - 階層的なグループ(hierarchical group)
    - 調整プロセス(coordinator)が他のプロセスに指示を与える

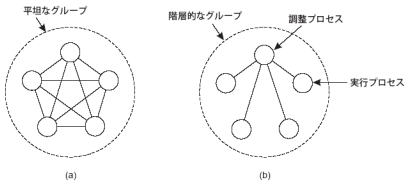


図7.3 (a) 平坦なグループにおける通信，(b) 単純な階層的グループにおける通信

グループメンバ管理

- プロセスのグループへの参加・離脱、グループ作成・削除などの管理手法
  - グループサーバで集中管理
    - 集中型 → 故障の単一箇所性
  - 分散管理
    - 高信頼マルチキャストの利用
      - グループの参加要求を全てのグループメンバに送信
    - 問題
      - クラッシュしたプロセスの離脱処理
      - 参加・離脱時のデータメッセージの同期
        - グループ宛の全てのメッセージ--- 参加した瞬間から受信する必要があり、かつ、離脱した瞬間から受信してはいけない
      - 多くのマシンがダウン → グループ再構成のプロトコルが必要

障害隠蔽とレプリケーション

- システムの障害隠蔽の解決法の一つ
  - （障害が起こりやすい）プロセスの複製をいくつか作り、グループを形成
  - 2つのアプローチ(6章参照)
    - プライマリベースプロトコル --- 階層的グループ
      - フォールトトレラント性の保証→ バックアップを作成(プライマリバックアッププロトコル)
      - プライマリが故障 → 他のバックアップから新しいプライマリを選出
    - レプリカ書き込みプロトコル --- 平坦なグループ
      - グループ内の1つが故障が全体の分散協調に支障を及ぼさない

障害隠蔽とレプリケーション

- いくつ複製を作ればよいか？
  - K-フォールトトレラント性(K fault tolerant)
    - 高々K個のプロセスが故障してもシステムが正常に動作し続ける性質
  - K-フォールトトレラント性の保証
    - 故障したプロセスが単に停止するのみ → K+1個のレプリカで十分
    - プロセスがビザンチン障害(任意障害)を持つ場合
      - プロセスがランダムなメッセージを送信する可能性 → 少なくとも2K+1個のレプリカが必要 （過半数のプロセスは正しい）
    - 前提条件：全てのリクエストは全レプリカに同じ順序で到着
      - アトミックマルチキャスト問題(atomic multicast problem)

障害システムにおける同意

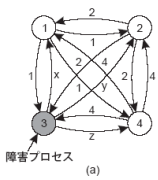
- プロセスグループで同意を形成する必要
  - コーディネータの選出、(分散)トランザクションのコミット、同期など
  - 通信とプロセスに障害が起こりうる状況下では問題

障害システムにおける同意

- ビザンチン将軍問題(Byzantine generals problem)
  - n人の将軍がお互いの兵力の情報を交換したい
  - そのうちm人の将軍は裏切り者であり、嘘の情報を流して合意に至るのを邪魔する
  - 忠実な(裏切り者でない)将軍たちは合意に至ることができるか？

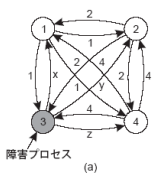
障害システムにおける同意

- Lamportのアルゴリズム[Lamport 1982]
  - 4ステップから構成
  - ステップ1:各将軍は自分の兵力情報を他の全ての将軍に告知



障害システムにおける同意

- Lamportのアルゴリズム[Lamport 1982]
  - ステップ2:各将軍は自分が受け取った情報をベクトル形式でまとめる

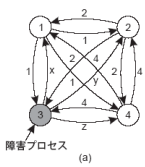


将軍1は(1, 2, x, 4)を獲得
将軍2は(1, 2, y, 4)を獲得
将軍3は(1, 2, 3, 4)を獲得
将軍4は(1, 2, z, 4)を獲得

(b)

障害システムにおける同意

- Lamportのアルゴリズム[Lamport 1982]
  - ステップ3:各将軍は自分が受け取ったベクトルを他の全ての将軍に転送

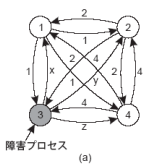


	将軍1の獲得情報	将軍2の獲得情報	将軍4の獲得情報
将軍1は(1, 2, x, 4)を獲得	(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
将軍2は(1, 2, y, 4)を獲得	(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
将軍3は(1, 2, 3, 4)を獲得	(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)
将軍4は(1, 2, z, 4)を獲得			

(b) (c)

障害システムにおける同意

- Lamportのアルゴリズム[Lamport 1982]
  - ステップ4:各将軍は自分が受け取ったベクトル群の各要素を比較
    - 過半数が同じ値ならば、その値を正しい値として記録



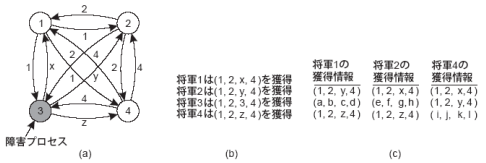
	将軍1の獲得情報	将軍2の獲得情報	将軍4の獲得情報
将軍1は(1, 2, x, 4)を獲得	(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
将軍2は(1, 2, y, 4)を獲得	(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
将軍3は(1, 2, 3, 4)を獲得	(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)
将軍4は(1, 2, z, 4)を獲得			

(b) (c)

→ 将軍1:(1,2,*,4)、将軍2:(1,2,*,4)、将軍4:(1,2,*,4)

障害システムにおける同意

- Lamportのアルゴリズム[Lamport 1982]
  - 障害プロセスがm個のとき、正常プロセスが2m+1個以上、全体で3m+1個以上のとき、正常プロセス同士で合意に至ることが可能
    - 以下の例はm=1, n=4>=3m+1なのでOK

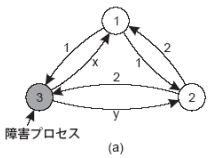


	将軍1の獲得情報	将軍2の獲得情報	将軍4の獲得情報
将軍1は(1, 2, x, 4)を獲得	(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
将軍2は(1, 2, y, 4)を獲得	(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
将軍3は(1, 2, 3, 4)を獲得	(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)
将軍4は(1, 2, z, 4)を獲得			

→ 将軍1:(1,2,*,4)、将軍2:(1,2,*,4)、将軍4:(1,2,*,4)

障害システムにおける同意

- Lamportのアルゴリズム[Lamport 1982]
  - m=1,n=3ではこのアルゴリズムで合意に至ることは出来ない



	将軍1の獲得情報	将軍2の獲得情報
将軍1は(1, 2, x)を獲得	(1, 2, y)	(1, 2, x)
将軍2は(1, 2, y)を獲得	(a, b, c)	(d, e, f)
将軍3は(1, 2, 3)を獲得		

→ 将軍1:(*,*,*)、将軍2:(*,*,*)

高信頼クライアントサーバ間通信

- 信頼性のある2地点間通信
  - TCPのような高信頼通信プロトコルにより構築
  - (通信路の)クラッシュ障害は隠蔽されない
- 障害を考慮したRPC
  - RPCで起こりうる障害の分類
    - クライアントがサーバの位置を特定できない
    - クライアントからサーバへの要求メッセージが喪失
    - サーバが要求を受けたあとにクラッシュ
    - サーバからクライアントへの応答メッセージが喪失
    - クライアントにおいて要求メッセージを送信後に障害発生
  - RMIの場合も同様

障害を考慮したRPC

- クライアントがサーバの位置を特定できない場合
  - 考えられる原因
    - サーバがダウン
    - クライアントスタブのバージョンが古く、サーバスタブと互換性がない
    - など
  - 解決策
    - 例外を発生し、適切に処理
      - Javaなどは例外処理機構を持つ
      - C言語ではシグナルハンドラにて実現
    - 問題点
      - 全ての言語が例外・シグナルを持つわけではない
      - 透過性が実現できない
        - 手続きがローカルかリモートかで例外処理内容が全く異なる

障害を考慮したRPC

- 要求メッセージが喪失した場合
  - タイムアウト・再送メカニズムで対処可能
  - 何回再送しても確認通知(ACK)がない場合 → サーバの位置を特定できない場合に帰着

障害を考慮したRPC

- サーバでクラッシュが起きた場合
  - 2つの可能性
    - 要求を受信し、手続き実行後にクラッシュした場合(図(b))
      - クライアントに障害を通知する必要
    - 要求を受信し、手続き実行前にクラッシュした場合(図(c))
      - クライアントから要求を再送するのみでよい
    - クライアント側からは両者を区別不能(タイムアウトのみ観測)

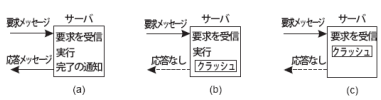


図 7-6 クライアントサーバ間通信でのサーバの動き。
(a) 正常な場合、(b) 実行後に障害、(c) 実行前に障害

障害を考慮したRPC

- サーバでクラッシュが起きた場合
  - 対応方法
    - サーバが再起動するまで待ち、同じ処理を再実行
      - 手続きが少なくとも1回は実行されることを保証（最低1回セマンティクス(at-least-once semantics)）
    - すぐ要求をあきらめて障害発生を通知
      - 手続きがまだ実行されていないかもしれない（最高1回セマンティクス(at-most-once semantics)）
    - 何も保証しない
      - この立場に立つと、実装が容易
    - 理想的な対応方法：正確に1回実行されることを保証（正確1回セマンティクス(exactly once semantics)）

障害を考慮したRPC

- サーバでクラッシュが起きた場合
  - 理想的な対応方法：正確に1回実行されることを保証（正確1回セマンティクス(exactly once semantics)）
    - 実装方法は一般に存在しない

障害を考慮したRPC

- 印刷を実行するRPCの例：
  - サーバはクライアントからテキストファイルの印刷要求を受信して、プリンタに対して印刷の指示を出す
  - サーバは要求受信後、確認通知（ACK）を返信
  - サーバは処理の完了通知をいつ送信するか？
    - 実際の印刷の指示を出す前
    - 実際の印刷が終了した後
  - サーバがクラッシュし、後に復旧したとする
  - クライアントは要求を再送すべきか？（ただし、正確に1回だけ印刷したい）

障害を考慮したRPC

- 印刷を実行するRPCの例：
  - サーバ側では完了メッセージの送信（M）、テキストの印刷（P）、クラッシュ（C）の3 つの出来事が起きるとする
  - 発生の順番は以下の6通り
    1. M→ P →C：完了通知送信・印刷の後にクラッシュが起きた
    2. M→C(→ P)：完了通知の後にクラッシュが起きて、テキストの印刷は済んでいない。
    3. P →M→C：印刷・完了通知の後にクラッシュが起きた。
    4. P →C(→M)：印刷の後にクラッシュが起きて、完了通知は済んでいない。
    5. C(→ P →M)：印刷・完了通知以前にクラッシュが起きて何もできなかった。
    6. C(→M→ P)：完了通知・印刷以前にクラッシュが起きて何もできなかった。

障害を考慮したRPC

- 印刷を実行するRPCの例：
  - いかなる場合を考慮しても、正確に1回だけ印刷するのは不可能

クライアント　　サーバ

要求再送方法	通知後印刷する場合 MPC	MC(P)	C(MP)	印刷後に通知する場合 PMC	PC(M)	C(PM)
常に行う	DUP	OK	OK	DUP	DUP	OK
行わない	OK	ZERO	ZERO	OK	OK	ZERO
ACK受信時のみ	DUP	OK	ZERO	DUP	OK	ZERO
ACKを受信しない時	OK	ZERO	OK	OK	DUP	OK

OK =テキストは1回印刷される
DUP =テキストは2回印刷される
ZERO =テキストは全く印刷されない

図7-7　サーバで障害が起きた場合の、クライアントとサーバの対応の組み合わせに応じた印刷結果

障害を考慮したRPC

- 応答メッセージの喪失
  - タイムアウト・再送による解決
    - 手続きが2回実行される可能性
    - 何回繰り返しても良い手続き(繰り返し等価、べき等(idempotent))ならば問題ない
    - 繰り返されると困る場合
      - 銀行口座からの送金手続きなど
    - 解決法：
      - 全ての手続きを繰り返し等価にする → 本質的に繰り返し等価でない処理（送金の例）もある
      - クライアントからの全ての要求に一意な番号を振る
        - 要求が1回目か2回目かをサーバ側で把握 → 2回目の実行を抑止

障害を考慮したRPC

- クライアントがクラッシュした場合
  - サーバに要求後、結果を受け取る前にクライアントがクラッシュ
  - 要求自体は生きているが、依頼主が居ない状態（孤児、orphan）
    - サーバは、孤児のために無駄なCPUやリソースを使用
    - クライアントが再起動し、要求を再発行した直後、孤児からの応答を受信 → 混乱が発生

障害を考慮したRPC

- クライアントがクラッシュした場合
  - 4つの解決方法[Nelson 1981]
    1. 根絶(extermination)
       - RPC呼び出しのログをクライアントでディスクに保存→復帰後ログを参照→ 孤児を特定し、明示的に処分
       - 欠点：ディスクの浪費、孤児自身がRPCを発行することにより、孫孤児が発生し、追跡困難になる可能性など
    2. 再生(reincarnation)
    3. 温和な再生(gentle reincarnation)
    4. 期限切れ(expiration)

障害を考慮したRPC

- クライアントがクラッシュした場合
  - 4つの解決方法[Nelson 1981]
    1. 根絶(extermination)
    2. 再生(reincarnation)
       - 時間をタイムスロットに分割→各々に番号（エポックナンバ）を割り振る→ クライアントが復帰後、新しいエポックナンバの開始をブロードキャスト→サーバはそのクライアントの代わりに孤児を処分
    3. 温和な再生(gentle reincarnation)
    4. 期限切れ(expiration)

障害を考慮したRPC

- クライアントがクラッシュした場合
  - 4つの解決方法[Nelson 1981]
    1. 根絶(extermination)
    2. 再生(reincarnation)
    3. 温和な再生(gentle reincarnation)
       - 再生の変形。サーバは新しいエポックナンバーの開始を知らせるブロードキャストを受け取ると、RPCリクエストが来ているか探し、もしあればその親（要求元）を探し、存在しなければ孤児として処分
    4. 期限切れ(expiration)

障害を考慮したRPC

- クライアントがクラッシュした場合
  - 4つの解決方法[Nelson 1981]
    1. 根絶(extermination)
    2. 再生(reincarnation)
    3. 温和な再生(gentle reincarnation)
    4. 期限切れ(expiration)
       - RPC処理の期限Tを設ける→ サーバは期限までに処理を完了できなければ更に時間Tを要求 → クライアントは障害発生後、復帰するまで時間Tだけ待てば、孤児は期限切れにより消滅

障害を考慮したRPC

- クライアントがクラッシュした場合
  - 4つの解決方法[Nelson 1981]
    1. 根絶(extermination)
    2. 再生(reincarnation)
    3. 温和な再生(gentle reincarnation)
    4. 期限切れ(expiration)
  - 実際には、これらの方法では不十分
    - 孤児がリソースをロックしている場合など、困難な課題が存在

高信頼グループ通信

- プロセスの複製による耐障害性の実現
  - 高信頼マルチキャストが重要
    - プロセスグループのメンバ全員にメッセージが届くことを保証
- プロセスグループに対する高信頼メッセージ配信に関して議論する

高信頼マルチキャストの基本手法

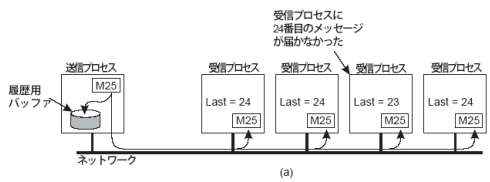
- 高信頼マルチキャスト(reliable multicast)
  - プロセスグループ宛のメッセージは必ず各メンバに配送
  - マルチキャスト中に参加・離脱が起こる場合の対処も規定
    - 障害プロセスがある場合
      - 正常なグループメンバのみがメッセージ受信
      - 困難な点：メッセージ配信前に、グループ全体の状態が現在どうなっているかに関して合意しておく必要
    - メンバ全員が正常な場合

高信頼マルチキャストの基本手法

- 高信頼マルチキャスト(reliable multicast)
  - プロセスグループ宛のメッセージは必ず各メンバに配送
  - マルチキャスト中に参加・離脱が起こる場合の対処も規定
    - 障害プロセスがある場合
    - メンバ全員が正常な場合
      - グループメンバは分かっているとし、参加離脱もない場合
        - 全てのメッセージが全てのメンバに行き渡ればよい
        - → 受信者が少数ならば、実装は容易

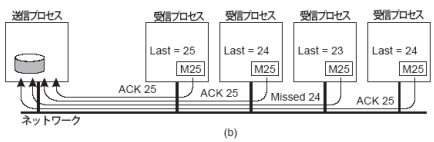
高信頼マルチキャストの基本手法

- 高信頼マルチキャスト(reliable multicast)の単純な実装
  - 各メッセージにシーケンス番号を付加
  - →受信プロセスは届かなかったメッセージを検出可能



高信頼マルチキャストの基本手法

- 高信頼マルチキャスト(reliable multicast)の単純な実装
  - 正常に受信したプロセスは確認通知(ACK)を返信
  - メッセージ損失に気づいた受信プロセスは否定的確認通知(NACK)を返信 → 送信プロセスは再送を行う



高信頼マルチキャストにおけるスケーラビリティ

- 単純な高信頼マルチキャストの問題点
  - 多数の受信者をサポートできない点
    - N人の受信者 → N個の確認通知(ACK)を受信する必要
    - → 送信者は大量のACK/NACK(=フィードバック)を処理できない可能性（**フィードバック爆発**）
    - 受信者が広域ネットワークに広がっていることも考慮する必要

高信頼マルチキャストにおけるスケーラビリティ

- 一つの解決策:
  - 受信者に確認通知(ACK)を通知させない
    - NACKのみフィードバック
    - → スケーラビリティがより高まる([Towsley et.al. 1997]など)
  - フィードバック爆発が決して起こらないことを保証するのは困難
  - 送信者は、(理論的には)再送のために過去の送信済みメッセージを全て保存しておかなければならない
    - どれほど古いメッセージに対する否定応答(NACK)が来るかわからない
      - 全受信者にメッセージが届くまでの時間が一般に予測不能
    - 確認通知(ACK)を受信しないので破棄できない

高信頼マルチキャストにおけるスケーラビリティ

- 高信頼マルチキャストにスケーラビリティを持たせる手法
  - スケーラブル高信頼マルチキャスト(Scalable Reliable Multicasting: SRM)プロトコル
    - フィードバックメッセージの数を抑制する手法(feedback suppression)
    - 基本アイデア: ACKは返信せず、NACKは受信者のうち代表者一人のみが返信する

高信頼マルチキャストにおけるスケーラビリティ

- SRMプロトコルの動作概要
  - 受信者は成功した配送には応答しない
  - 受信に失敗したときのみNACKを返信(フィードバック)
  - NACKは他の受信者にもマルチキャスト
    - → 他の受信者のフィードバックを抑制
  - NACKに対する再送は常に受信者全員にマルチキャスト
    - → 再送要求は受信者のうち誰か一人が出せばよい

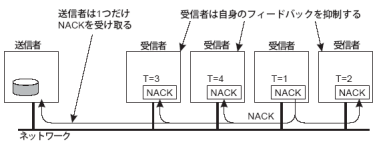


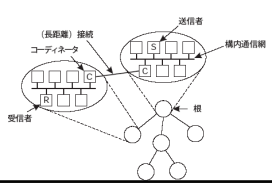
図7.9 何人かの受信者は再送要求を保持する。最初の再送要求によって他の再送要求が抑制される

高信頼マルチキャストにおけるスケーラビリティ

- 受信者のうち一人のみがフィードバックを行うようにしたい
  - 解決法:
    - 各受信者は、送信しようとしているNACKメッセージをまずランダムな時間保留する
      - その間に他の受信者がNACKを返信すれば、自分にもマルチキャストされるのでそのことが分かる → 自分はNACKを送信しない
    - その間、誰もNACKを送信しなければ、自分がNACKを送信者および他の受信者にマルチキャスト
  - 問題点
    - 伝送遅延の大きい広域ネットワークでは受信者同士の同期が困難 → どうしても複数プロセスが同時にフィードバックを行う可能性が残る

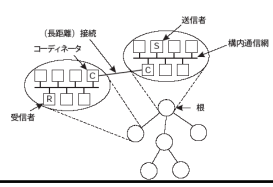
階層的なフィードバック抑制

- フィードバック抑制問題に対する階層的なアプローチ
  - 受信者のグループ → 幾つかのサブグループに分割
  - サブグループは木状に配置
    - 送信者を含むサブグループは木のルートを形成
  - 各サブグループはローカルコーディネータを指定
    - サブグループ内の受信者の再送要求を処理する責任を持つ
    - 自分自身が受信に失敗したら、親のサブグループのコーディネータに再送要求



階層的なフィードバック抑制

- 問題点:木の構築
  - 動的に構築する必要
    - 既存のマルチキャスト木を利用 → ネットワーク層におけるマルチキャストルータがコーディネータの役割を果たせるように機能強化する必要 → 容易ではない



アトミックマルチキャスト

- アトミックマルチキャスト(atomic multicast)
  - メッセージが全てのプロセスに配送されるか、どのプロセスにも配送されないか、どちらかであること(アトミック性)を保証
  - メッセージが全てのプロセスに同じ順序で配送されること(全順序性)を保証

アトミックマルチキャスト

- アトミックマルチキャストの利用例
  - 分散データベースのレプリカ構築
  - 前提
    - 高信頼マルチキャストを利用可能
    - レプリカはプロセスグループを形成可能
    - 更新操作は全てのレプリカへのマルチキャスト(アクティブレプリケーション)
  - 障害シナリオ
    - ある更新を実行中にレプリカがクラッシュ
      - 他のレプリカは更新される
    - クラッシュしたレプリカが復旧 → 他のレプリカと同じ状態に更新する必要

アトミックマルチキャスト

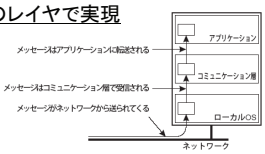
- アトミックマルチキャストの利用例
  - 分散データベース(アクティブレプリケーション)
  - アトミックマルチキャストの使用
    - クラッシュ直前の更新操作 --- 正常な全てのレプリカで実行されているか、全く実行されていないかのいずれか
    - あるレプリカがクラッシュ → グループから除外することに他のメンバが合意した後、更新実行
    - クラッシュしたレプリカが復旧 → もう一度グループに参加(このとき、状態を他のメンバと同じにする) → 更新操作の受信を再開
    - アトミック性、および、グループ参加時の調停により、一貫性を保証

仮想同期マルチキャスト

- マルチキャストグループが動的に変化する場合の耐障害性
  - 送信者がマルチキャスト中にグループメンバが変化したらどうするか？
    → 常に送信時のグループメンバ全員にメッセージが届くようにしたい
  - 送信者がメッセージ送信後、グループメンバ全員が受信する前に送信者がクラッシュしたらどうするか？
    → 常にメンバ全員がそのメッセージを破棄することを保証したい
- このような高信頼マルチキャスト ＝ **仮想同期マルチキャスト(Virtually Synchronous Multicast)** [Birman and Joseph,1987a][Birman and Joseph,1987b]

仮想同期マルチキャスト

- 既に受信したメッセージの破棄 → 受信メッセージと配送メッセージの区別により実現
  - メッセージの受信(receipt)
    - ローカルOSにメッセージが送られてくる→ コミュニケーション層により受信
    - アプリケーションはまだメッセージが届いたことを知らない
  - メッセージの配送(delivery)
    - コミュニケーション層からアプリケーションにメッセージが届けられる
- アトミック性は、配送のレイヤで実現

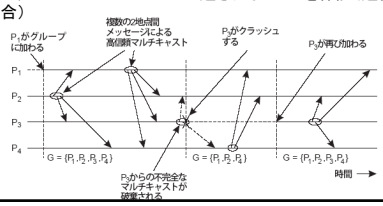


仮想同期マルチキャスト

- **グループビュー(group view)**
  - メッセージmが配送されるべきプロセスのリスト
    - mがマルチキャストされるとき、送信者が持っていた配送リスト(ビュー)
  - リスト上の全プロセスが同じビューを持つ
    - リスト上の全プロセスにmが配送され、それ以外にはmが配送されないことを、リスト上の全プロセスが合意

仮想同期マルチキャスト

- ビューの変更(view change)
  - グループビューへのプロセスの追加、または、削除
- マルチキャストメッセージmを転送中に、ビューが変更された場合、どうするか？
  - ビューの変更通知vcが届く前に、全てのプロセスにmを配送
  - または、全てのプロセスにmが配送されないことを保証（送信者クラッシュの場合）



メッセージ順序の保証

- マルチキャストにおける順序の保証
  - 高信頼無順序マルチキャスト(reliable unordered multicast)
    - 順序に関しては何も保証しない

プロセス P_1	プロセス P_2	プロセス P_3
m1の送信	m1の受信	m2の受信
m2の送信	m2の受信	m1の受信

メッセージ順序の保証

- マルチキャストにおける順序の保証
  - 高信頼無順序マルチキャスト(reliable unordered multicast)
  - 高信頼FIFO順序マルチキャスト(reliable FIFO-ordered multicast)
    - 同じプロセスからのメッセージは、必ず送信順に配送

プロセス P_1	プロセス P_2	プロセス P_3	プロセス P_4
m1の送信	m1の受信	m3の受信	m3の送信
m2の送信	m3の受信	m1の受信	m4の送信
	m2の受信	m2の受信	
	m4の受信	m4の受信	

メッセージ順序の保証

- マルチキャストにおける順序の保証
  - 高信頼無順序マルチキャスト(reliable unordered multicast)
  - 高信頼FIFO順序マルチキャスト(reliable FIFO-ordered multicast)
  - 高信頼因果順序マルチキャスト(reliable causally-ordered multicast)
    - メッセージ間の因果関係を保持して配送

メッセージ順序の保証

- マルチキャストにおける順序の保証
  - 高信頼無順序マルチキャスト(reliable unordered multicast)
    - 順序に関しては何も保証しない
  - 高信頼FIFO順序マルチキャスト(reliable FIFO-ordered multicast)
    - 同じプロセスからのメッセージは、必ず送信順に配送
  - 高信頼因果順序マルチキャスト(reliable causally-ordered multicast)
    - メッセージ間の因果関係を保持して配送

- 全順序配送(total-ordered delivery)
  - 全てのグループメンバに同じ順序で配送
  - **アトミックマルチキャスト＝仮想同期高信頼マルチキャスト＋全順序配送**

メッセージ順序の保証

マルチキャスト	基本的なメッセージの順序	全順序配送か？
高信頼無順序マルチキャスト	なし	いいえ
高信頼FIFO順序マルチキャスト	FIFO順序配送	いいえ
高信頼因果順序マルチキャスト	因果順序配送	いいえ
アトミックマルチキャスト	なし	はい
FIFO順序アトミックマルチキャスト	FIFO順序配送	はい
因果順序アトミックマルチキャスト	因果順序配送	はい

仮想同期高信頼マルチキャストの6通りのバージョン

仮想同期マルチキャストの実装

- 実装例：Isis [Birman et.al. 1991]
  - 実際に使用されているフォールトトレラント分散システム
  - 高信頼ユニキャストであるTCPを用いて実装
- 基本動作
  - 送信者はグループビューGの各メンバにTCPで順番にメッセージ送信
- 解決すべき問題
  - ビューGに対して送信された全てのメッセージを全てのメンバが受信することの保証
    - 送信者が送信中にクラッシュしたらどうするか？
    - ビューが変更されたときどうするか？

仮想同期マルチキャストの実装

- 送信者がメッセージmを送信中にクラッシュした場合
  - まだmを受信していないグループメンバは、既に受信した他のメンバからmを送ってもらう
  - → 受信者はメッセージmを受信し損ねたことをどのように検出するか？

仮想同期マルチキャストの実装

- 受信者はメッセージmを受信し損ねたことをどのように検出するか？
  - コミュニケーション層は、メッセージmがグループメンバ全員に受信されるまでmを保留
    - メンバ全員に受信されたメッセージを安定である(stable)と呼ぶ
    - さもなければ、安定でない(unstable)と呼ぶ
  - 安定なメッセージのみアプリケーションに配送

仮想同期マルチキャストの実装

- メッセージの安定性の保証方法
  - ビューが変更されない（グループメンバが変わらない）時
    - グループの中からコーディネータを選出
    - コーディネータはグループ内の他のプロセスにメッセージmを送信するよう要求
    - 他の全てのプロセスからmを受信すればmが安定であることが保証される

仮想同期マルチキャストの実装

- メッセージの安定性の保証方法
  - ビューが変更される時（誰かがクラッシュしたときなど）
    1. G(i)からG(i+1)へのビュー変更通知を受信した任意のプロセスPは、まず保持している全ての安定でないメッセージをG(i+1)の全メンバにマルチキャスト
    2. その後、Pはフラッシュメッセージ(flush message)をG(i+1)の全メンバにマルチキャスト

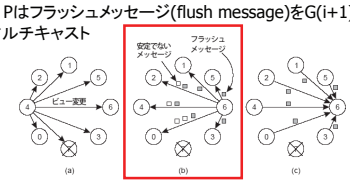


図7.16 (a) プロセス4は、プロセス7がクラッシュしたことに気付き、ビュー変更を送信。
(b) プロセス6は、安定でないメッセージを送信し、続いてフラッシュメッセージを送信。
(c) プロセス6は、他のすべてのプロセスからフラッシュメッセージを受信すると、新しいビューを取り入れる。

仮想同期マルチキャストの実装

- メッセージの安定性の保証方法
  - ビューが変更される時（誰かがクラッシュしたときなど）
    3. Pからフラッシュメッセージを受け取ったプロセスは、Pが安定でないメッセージを持たないことが分かる(TCPを利用 → 送信順に受信)
    4. 全てのプロセスからフラッシュメッセージを受信すれば、安定でないメッセージがなくなったことが保証される → ビューをG(i+1)に変更

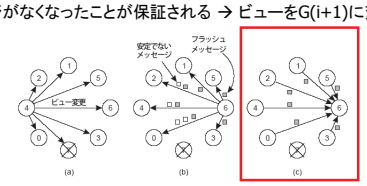


図7.16 (a) プロセス4は、プロセス7がクラッシュしたことに気付き、ビュー変更を送信。
(b) プロセス6は、安定でないメッセージを送信し、続いてフラッシュメッセージを送信。
(c) プロセス6は、他のすべてのプロセスからフラッシュメッセージを受信すると、新しいビューを取り入れる。

分散コミット

- 分散コミット(distributed commit)
  - アトミックマルチキャストの一般化
    - あるオペレーションがプロセスグループのメンバ全てで行われるか、全く行われないかどちらかであることを保証
- 単純な実現法(1相コミットプロトコル:one-phase commit protocol)
  - コーディネータ(プロセスC)を選出 → Cは他のメンバにオペレーションを実行するかしないかを通知
  - → オペレーションを実行不能なメンバが、そのことをCに通知できない
  - 改良版: 2相コミットプロトコル(two-phase commit protocol:2PC)

2相コミットプロトコル

- 2相コミットプロトコル(two-phase commit protocol:2PC)
  - 動作原理:コーディネータは参加者に投票を募り、全員が合意すればコミット、さもなければアボート

図 7-17　(a) 2PC におけるコーディネータの有限状態機械、(b) 参加者の有限状態機械

2相コミットプロトコルの問題点

- 障害が起こりうる場合
  - メッセージの受信待ちのためブロック → 送信者が故障した場合、無限にブロック
  - → タイムアウトメカニズムが必要
  - コーディネータからのGlobal-commitがタイムアウトした場合: 単純にアボートできない
    - 他の参加者はコミットするかもしれない

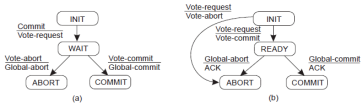


図 7-17　(a) 2PC におけるコーディネータの有限状態機械、(b) 参加者の有限状態機械

耐故障2相コミットプロトコル

- 耐故障版2相コミットプロトコルの実現
  - 単純な方法:コーディネータが回復するまでブロック
  - より良い解決法:
    - 他の参加者に問い合わせ、誰か1人でもアボートに投票していたら、自分もアボート
    - 誰か1人でも、Vote-requestを受信していなければ、同じくアボート
    - 他の参加者もコミットに投票していて、コーディネータからの応答待ちの場合、別の参加者に問い合わせる
    - → 全員が応答待ちならば、コーディネータの回復までブロック

図 7.18　参加者 P が状態 READY にいて、別の参加者 Q にコンタクトしたときに取る動作。

Q の状態	P の動作
COMMIT	COMMIT に遷移
ABORT	ABORT に遷移
INIT	ABORT に遷移
READY	別の参加者にコンタクト

3相コミット

- 3相コミットプロトコル(three-phase commit protocol:3PC)
  - 2PCでのブロッキングを回避
  - コミットの可能性がある場合は、PRECOMMITという状態にまず遷移 → 全員がPRECOMMITに到達したら、コーディネータはコミットを通知

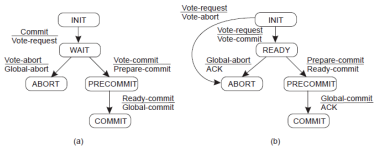


図 7-21　(a) 3PC でのコーディネータの状態遷移を表す有限状態機械
(b) 参加者の状態遷移を表す有限状態機械

3相コミットでの耐故障性

- コーディネータがブロックした場合
  - 状態WAIT: 単にアボートすればよい
  - 状態PRECOMMIT: ある参加者がクラッシュ →しかしその参加者はコミットに投票
  - → コーディネータはコミットを他の全参加者に通知
    - クラッシュした参加者は後で復旧したときにコミット

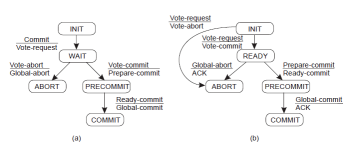


図 7-21　(a) 3PC でのコーディネータの状態遷移を表す有限状態機械
(b) 参加者の状態遷移を表す有限状態機械

3相コミットでの耐故障性

- 参加者がブロックした場合
  - 状態INIT: 単にアボートすればよい
  - 状態READYまたはPRECOMMIT: コーディネータがクラッシュ → 他の参加者に問い合わせて判断
    - 誰もPRECOMMIT状態に居ない → アボート
    - 問い合わせた参加者の大半がREADY状態 → アボート
      - 誰かがPRECOMMIT状態に居たとしても安全にアボート可能
    - 問い合わせた参加者の大半がPRECOMMIT状態 → コミット
      - 故障したプロセスは全てPRECOMMIT又はREADY状態 → それらは既にコミットに合意している → 復旧後にコミット

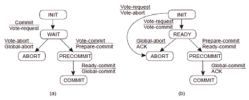


図 7-21 (a) 3PC でのコーディネータの状態遷移を表す有限状態機械
(b) 参加者の状態遷移を表す有限状態機械

回復

- 障害からの回復(recovery)
  - 障害(エラー)が起きた場合に、正しい状態に回復する必要
- エラー回復の2つの形式
  - 後ろ向き回復(backward recovery)
    - 以前の(最も新しい)正常な状態に戻すこと
      - 高信頼通信における再送に対応
  - 前向き回復(forward recovery)
    - 実行継続可能な新しい(正常な)状態に遷移させること
      - 高信頼通信における誤り訂正に対応
- 分散システムでは後ろ向き回復が一般的

後ろ向きエラー回復の課題

- システム全体の状態を巻き戻すのはコストがかかる
  - チェックポイントの作成 → 高コスト
  - 解決法:メッセージログ収集(message logging)
- 巻き戻した後、同じエラーを生じて無限ループに陥る可能性がある → 障害透過性を達成できない
- 元の状態に戻せないオペレーションが存在
  - ファイルを完全消去してしまうなど

メッセージログ収集

- 送信者ベースメッセージログ収集(sender-based message logging)
  - 各プロセスは、自分が送信したメッセージを記録
- 受信者ベースメッセージログ収集(receiver-based message logging)
  - 各プロセスは、自分が受信したメッセージを記録
- クラッシュからの回復処理
  - 最も新しいチェックポイントまで状態を巻き戻す
  - そこから、送受信されたメッセージを再生(replay)
  - → チェックポイント以降の状態に低コストで巻き戻し可能

チェックポイント作り

- チェックポイント
  - 分散システム全体における、過去の一貫したグローバル状態の記録
  - → 分散スナップショット(5章)のアルゴリズムで記録可能
- 各プロセスで独立に記録できないか?
  - 分散スナップショットアルゴリズムはコストがかかるので

独立チェックポイント作り

- 各プロセスが独立に状態を記録 → 一貫性のある状態までロールバックする必要
- 回復ライン(recovery line)
  - 直近の(最新の)一貫性のあるグローバル状態

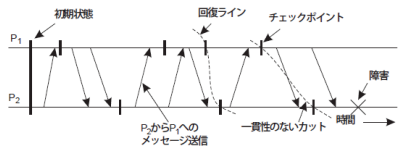


図 7-23 回復ライン

独立チェックポイント作り

- 回復ラインが見つからず、次々とロールバックの連鎖が生じる可能性がある(ドミノ効果) → 解決は困難
- → 各プロセスが協調してチェックポイントを作るのが一般的になってきている

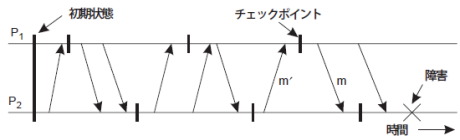


図7-24 ドミノ効果

協調チェックポイント作り

- 分散スナップショットより低コストな方法
  - コーディネータは、CHECKPOINT_REQUESTメッセージを全プロセスに通知
  - それを受信した各プロセスは、現在の処理を中断し、ローカルなチェックポイントを作成し、コーディネータに報告
  - 全てのプロセスから報告を受けたコーディネータは、CHECKPOINT_DONEメッセージを全プロセスに通知し、処理を継続させる
  - 各プロセスはCHECKPOINT_REQUEST受信時からCHECKPOINT_DONE受信時までに受信した他の全メッセージをチェックポイントの記録に追加
  - → 一貫性のあるグローバル状態を記録
    - ただし、5章の分散スナップショットアルゴリズムと異なり、全体の動作を中断させる必要がある

演習問題

1. 教科書図7-2(本スライド7ページの図)の3重冗長モジュールはビザンチン障害を扱えるか？
2. 次の各アプリケーションについて、最低1 回セマンティクスがよいか、それとも最大1 回セマンティクスがよいか考察せよ。
   (a)ファイルサーバからのファイルの読み書き
   (b)プログラムのコンパイル
   (c)遠隔バンキング
3. 教科書図7-14(本スライド57ページの図) において、FIFO 順序アトミックマルチキャストで許される配送順序を示せ。