分散システム 第7回
― 同期 ―

大連理工大学・立命館大学 国際情報ソフトウェア学部

大森 隆行

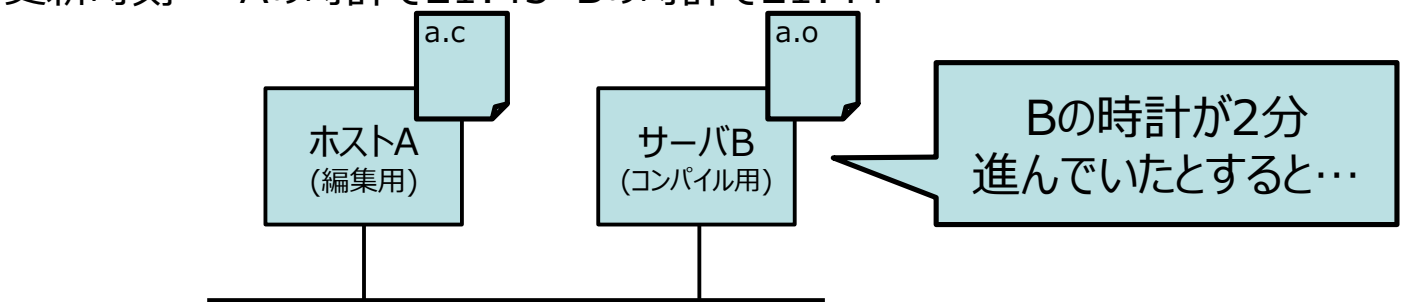
講義内容

- 同期
  - 同期と時間
  - クロック同期
  - 論理クロック
  - 排他制御

同期 (synchronization)

- 分散システム内のプロセスをいかに同期させるか
  - 同期：タイミングを合わせること
- 同期の必要性
  - 共有リソースへのアクセス
    - 排他制御
    - 矛盾した書き込み・呼び出しの防止
  - イベント、メッセージの順序の保持

(例) make (タイムスタンプに基づくソースコードのコンパイル)



物理的クロック

- コンピュータ内部の時刻保持の仕組み
  - 水晶振動子(特定の周波数で振動する)に基づくクロックを利用
  - 電源を落としても時刻を保持するために電池でバックアップしたCMOS RAMに保存
  - 水晶ごとに振動周波数は異なる
    - クロックスキュー(clock skew)：クロックが同期からずれる現象
- 課題
  - 互いに異なるクロック時間をどのように同期させるか
  - 本当の時刻(時計)とどのように同期させるか

標準時間

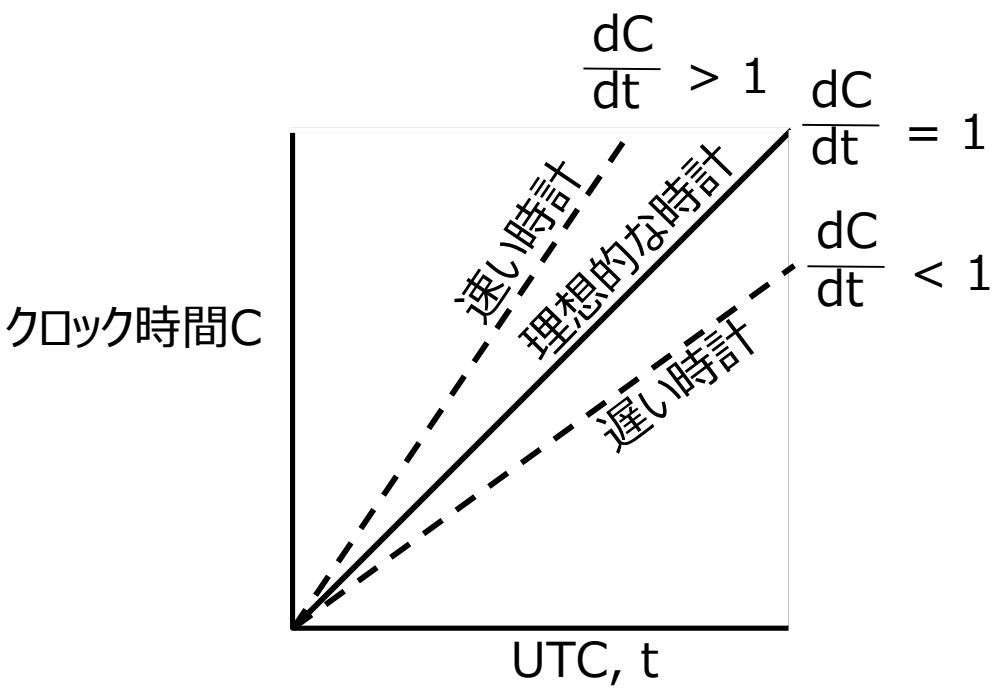
- セシウム時計
  - Cs133を使って発生させたマイクロ波(約91億Hz)を使用 → 1秒の基準とする
- 国際原子時間 (TAI：International Atomic Time)
  - パリの国際時間局で1958年1月1日0時からのセシウム時計のティック数に基づく
  - 平均太陽日(太陽の南中時刻に基づく測定)より約3ミリ秒短い
    → 太陽時間との差が800ミリ秒より大きくなったときに閏秒を導入(1秒ずらす)
  - ⇨ 協定世界時 (UTC: Universal Coordinated Time)

南中：太陽が真南に来ること

クロック同期

- クロック時間C
- 時間(UTC) t
- 最大ドリフト率ρ: クロックスキューがどの程度許容されるか

$$1-\rho \leqq \frac{dC}{dt} \leqq 1+\rho$$



最大ドリフト率ρの
2台のマシンの間では
t秒経過後には
最大2 ρ tだけ時計がずれる

時計のズレをδ以内に
抑えたい場合は、少なくとも
δ/2 ρごとに再同期が必要

確認問題

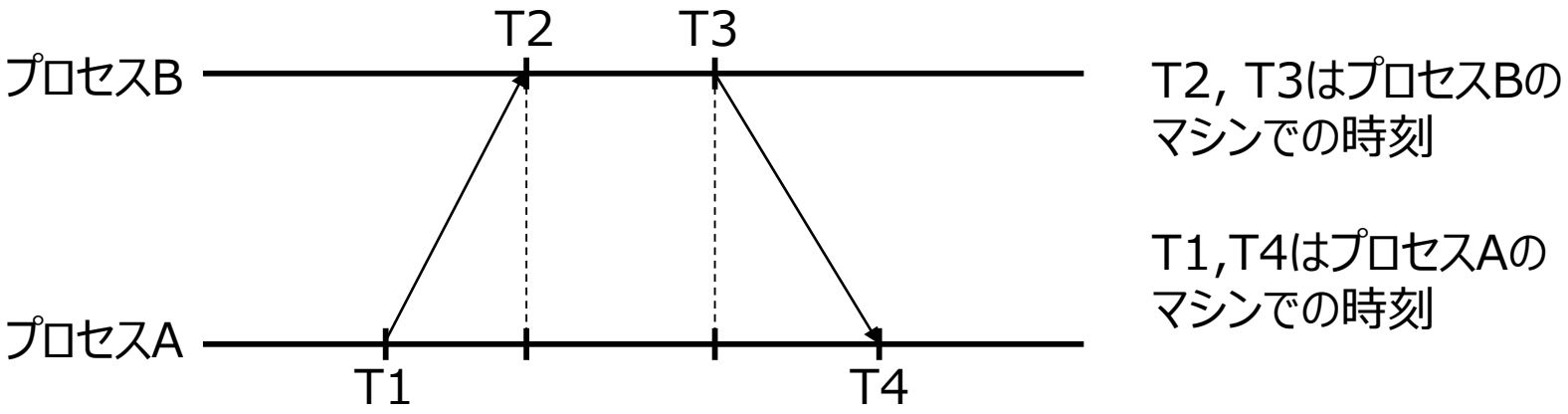
- 次の説明に合う語句を答えよ。
  - 複数のマシンやプロセス間で処理のタイミングを合わせること。
  - クロックが同期からずれる現象。
  - 国際原子時間と太陽時間のズレを調整するために追加あるいは削除される1秒。
  - 国際原子時間に由来する、世界的に使用される基準時刻。UTCのこと。
- 最大ドリフト率ρの2台のマシンの間で時間のズレをδ以内に抑えたい場合に最低限必要な再同期の周期を答えよ。

講義内容

- 同期
  - 同期と時間
  - クロック同期
  - 論理クロック
  - 排他制御

Network Time Protocol (NTP)

- 時間サーバ(time server)との交信のための
  プロトコル



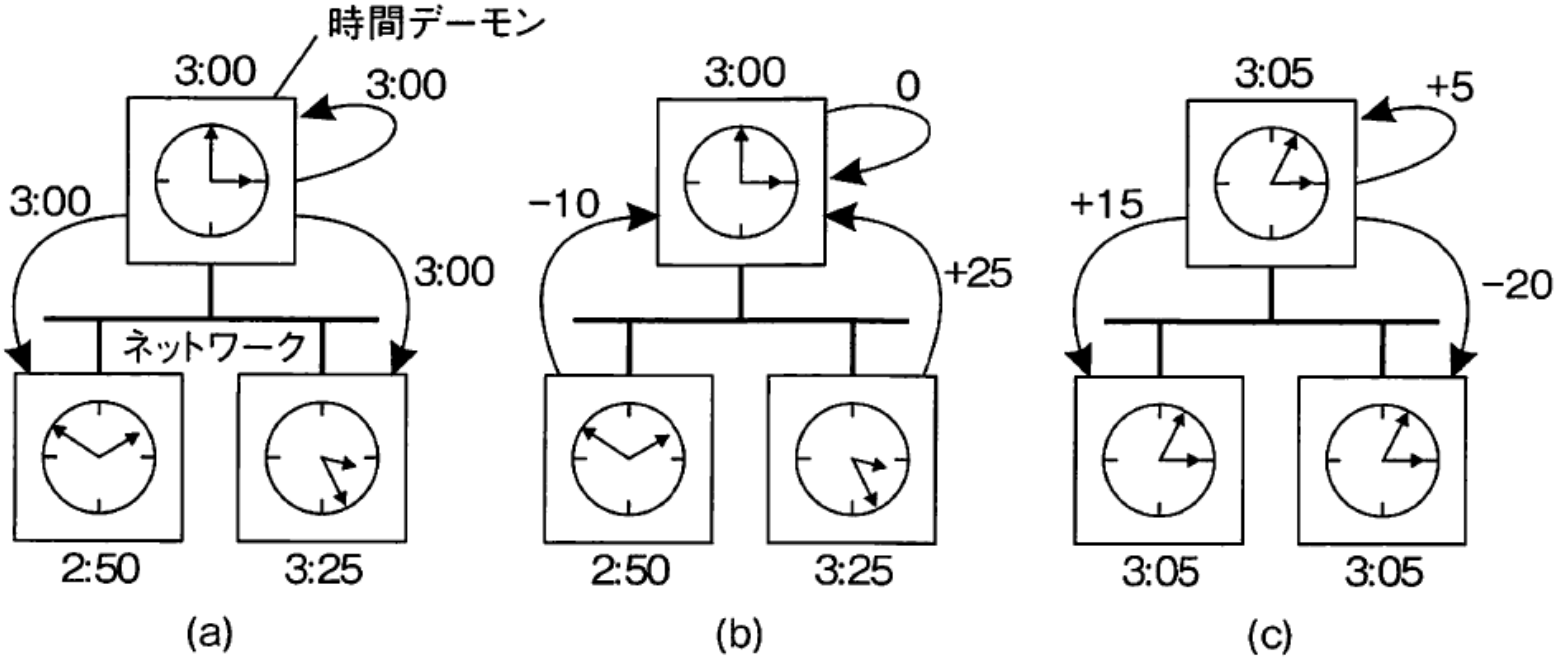
- 伝達遅れ T2-T1 と T4-T3 がほぼ同じと仮定
- Aの時計が遅れていた場合 → Bの時刻に合わせる
- Aの時計が進んでいた場合 → 徐々に時刻調整

Network Time Protocol (NTP)

- NTPのサーバ階層
  - 標準時計(reference clock)を持つサーバ
    ＝ 階層-1サーバ
  - 階層-kサーバ の時刻をもらうサーバ
    ＝ 階層-(k+1)サーバ
    (階層が高いサーバが相手サーバに合わせる)
  - 同じ階層のサーバ同士は相互に時刻を
    問い合わせる (AがBに問い合わせると同時に
    BもAに問い合わせ)

Berkeleyアルゴリズム

- 時間デーモン(time daemon)が能動的に各コンピュータに問い合わせ、平均時刻を計算



確認問題

- 以下の各文は正しいか。○か×で答えよ。
  - クロック同期の際、ネットワークの遅延は無視できる
  - NTPでは往路と復路で発生する遅延の大きさは同じだと仮定される
  - NTPでは、階層-2サーバは階層-1サーバの時刻に合わせる
  - Berkeleyアルゴリズムによるクロック同期では、調整後の時刻と現実の時刻は必ずしも一致しない

講義内容

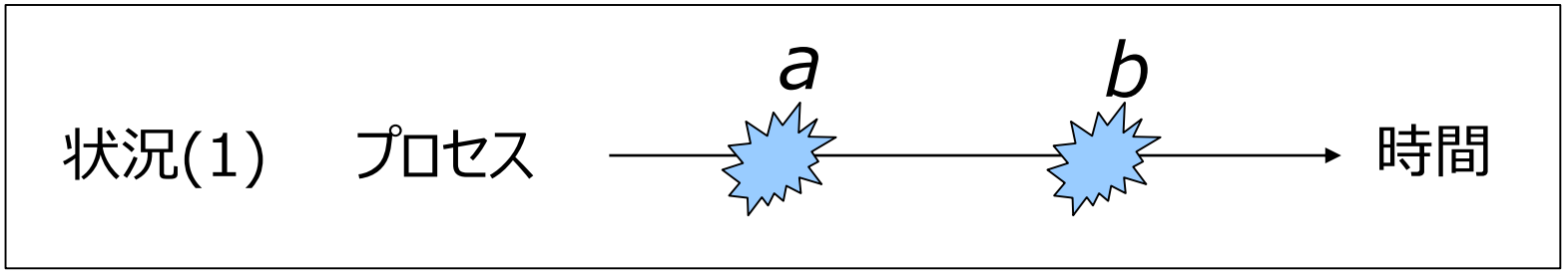
- 同期
  - 同期と時間
  - クロック同期
  - 論理クロック
  - 排他制御

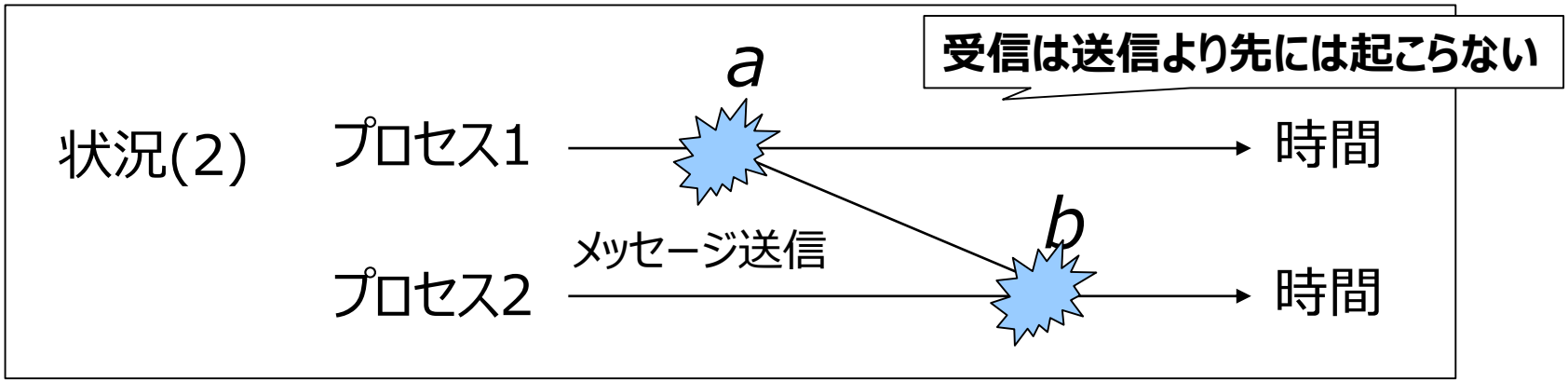
論理クロック

- マシンが保持する時間は実時間と関連がなくても十分な場合が多い
- イベントの順序にさえ同意していれば十分
  - (例) a.cとa.oのどちらが新しいか？
- 論理クロック(logical clock) ：イベントの発生順序に基づくクロック

Lamportの論理クロック

- 事前発生(happens-before)の関係
  - 「まずaが発生し、次にbが発生する」ということにすべてのプロセスが同意
  - a→b と表記

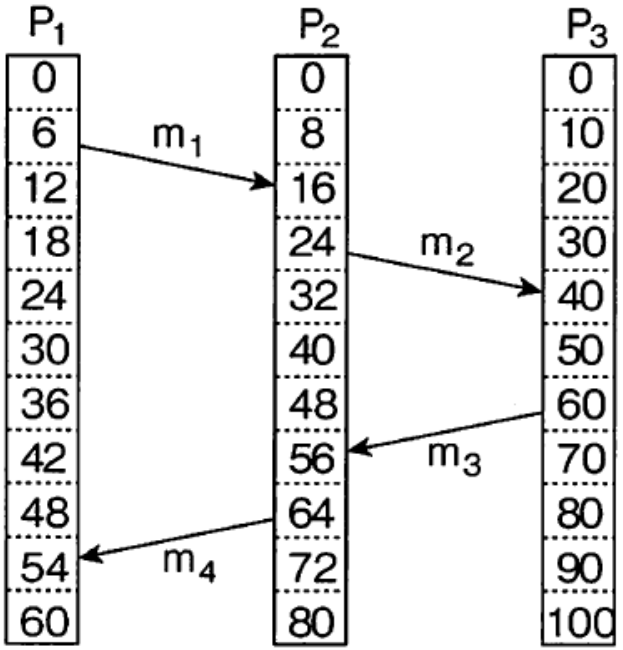




Lamportのアルゴリズム

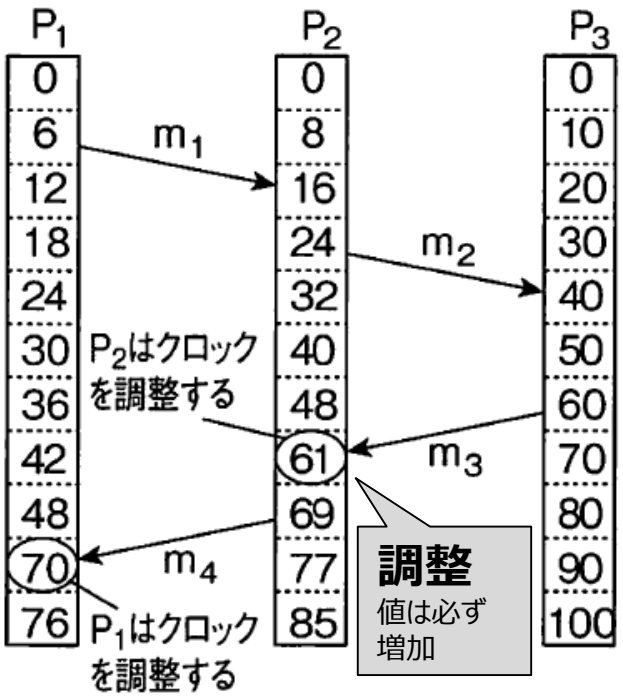
- a→b ならば C(a)→C(b)
  を満たすようにクロックを調整

C(a)はイベントaが
発生したときの
クロック時間



(a)

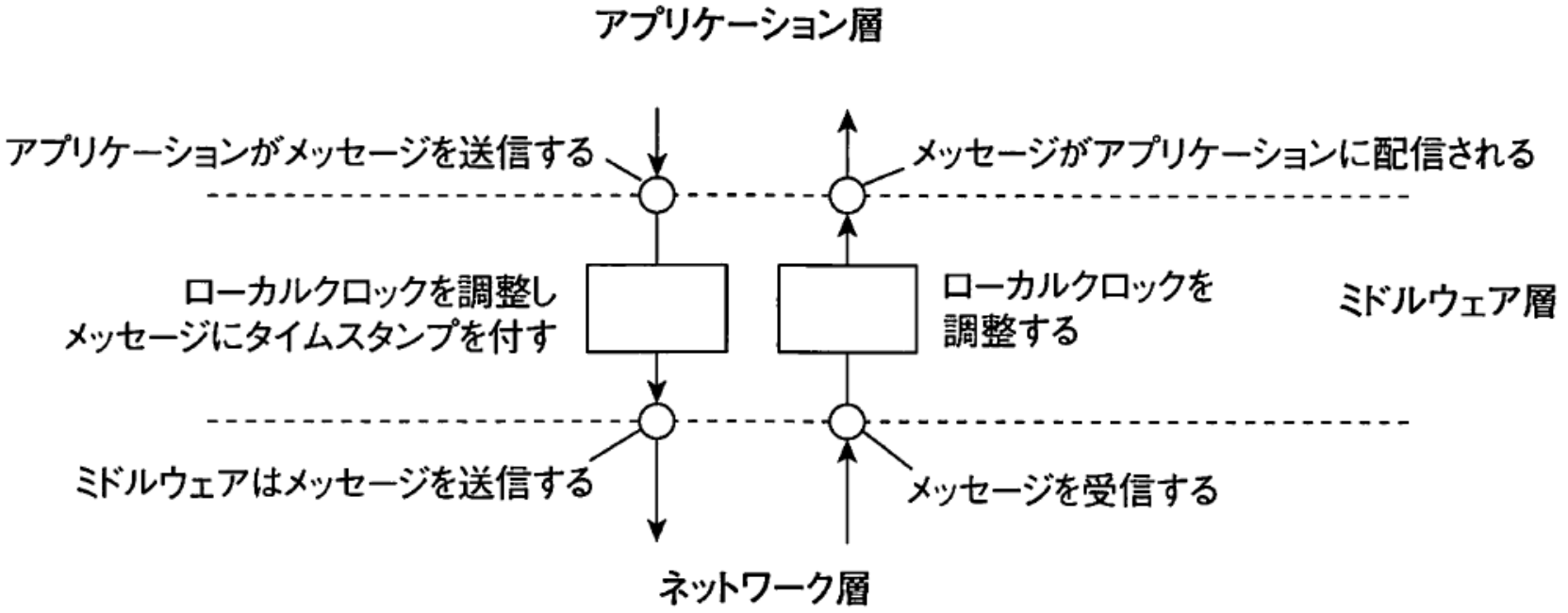
各マシンが自己のクロック時間を
持っている



(b)

イベントの前後関係に合うように
クロック時間を調整

Lamportの論理クロックの実装



- アプリケーション層とネットワーク層の中間(ミドルウェア層)で実装
  - 受信メッセージを処理する前にローカルクロックを調整

(例) Lamportの論理クロックの応用

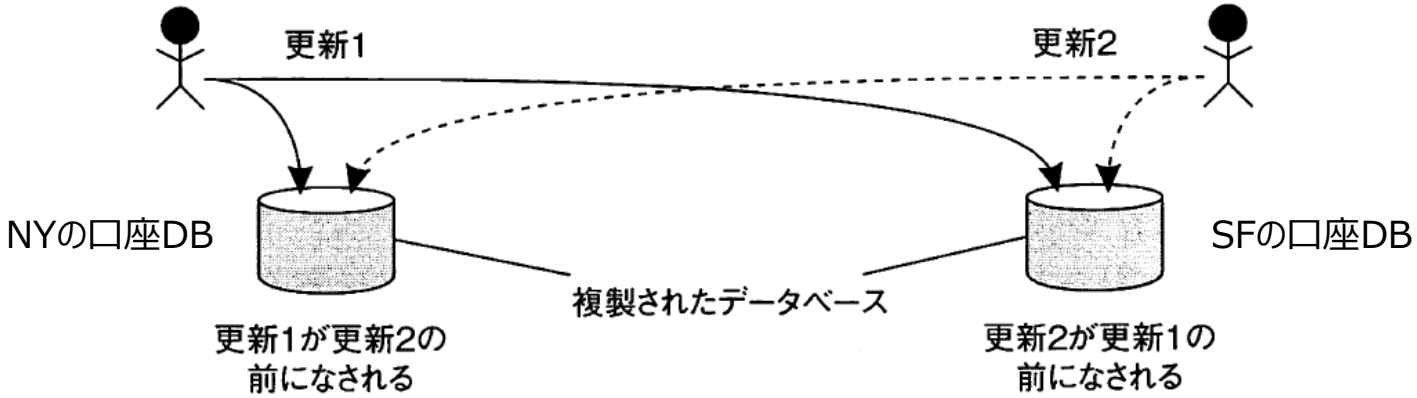
- 銀行預金口座のデータベースがNYとSFで複製されている状況
  - 口座残高$1000 → SFで100ドル預金 (残高$1100)
    → NYで金利追加処理 (残高$1111)
  - 口座残高$1000 → NYで金利追加処理$1010 (残高$1010)
    → SFで100ドル預金 (残高$1110)

値が異なるのはまずい

- 全順序マルチキャスト
  - 更新要求をマルチキャスト → 受信確認もマルチキャスト
  - 受信キュー内でメッセージを送信タイムスタンプ順に並べる
  - キューの先頭で、受信確認が揃ったものからアプリケーションに渡す

→ 全複製データベースで同じ順序で処理可能

※メッセージには送信者のタイムスタンプで送信時刻が付与される
※メッセージの喪失はないものと仮定



確認問題

- 以下の文は正しいか。○か×で答えよ。
  - Lamportの論理クロックを使うことで、実際に要求が発生した順番に処理を行うことを保証できる。

講義内容

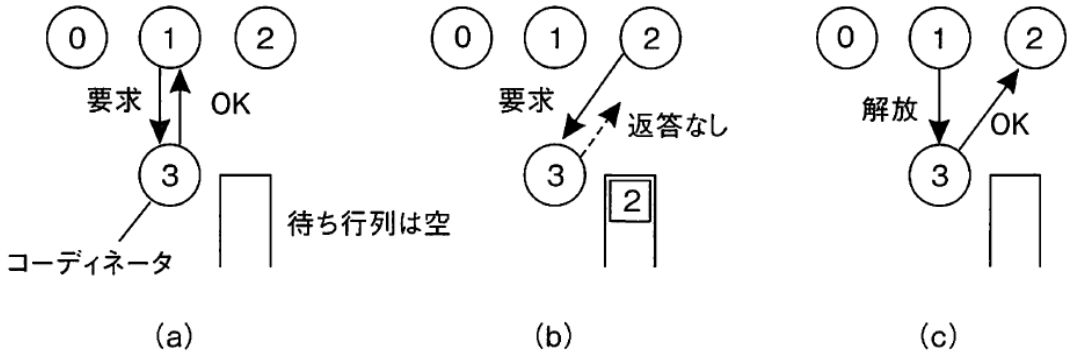
- 同期
  - 同期と時間
  - クロック同期
  - 論理クロック
  - 排他制御

排他制御

- 排他制御は分散システムにおいて生来的な問題
  - 複数プロセスの並行と協調
  - 同じ資源への同時アクセス
- 排他制御の際に避けるべきこと
  - 飢餓(starvation)
    - プロセスが必要なリソースにアクセスできない状態
  - デッドロック(dead lock)
    - 複数のプロセスが互いの終了を待ち、
      いずれも先に進めなくなる状態
- 排他制御の方式
  - トークンベース(token-based)方式
    - トークンを取得したプロセスがリソースを利用可能
  - 許可ベース(permission-based)方式
    - 許可を得たプロセスがリソースを利用可能

集中アルゴリズム

- 一つのプロセスをコーディネータとして専任
- リソースにアクセスするときはコーディネータの許可を得る
- コーディネータが待ち行列を管理
  - リソース使用中なら待ち行列に入れる



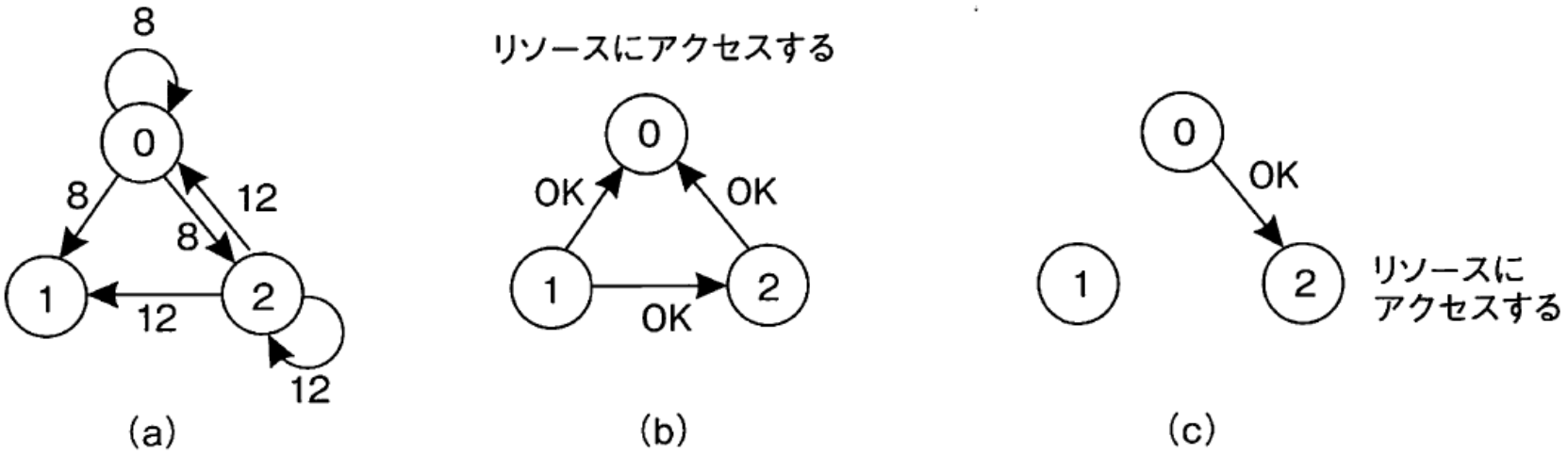
長所
- 先着順→公平
- どのプロセスも永遠に待つことはない
- 実装が容易

短所
- 故障単一箇所性 (Single point of failure)
  コーディネータがダウンすると全体が止まる

分散アルゴリズム

- Lamportの論理クロックを応用
- プロセスがリソースにアクセスするときのメッセージには
  リソースの名前、プロセス番号、現在の論理的時間を含む
- 自分も含め、すべてのプロセスに対してメッセージを送信
- メッセージに対する応答
  - (1) もし受信者がリソースにアクセス中でもアクセスしようとも
    していないのであれば、受信者は送信者にOKメッセージを送る
  - (2) もし受信者がリソースにアクセス中であれば、返答せず、
    要求を自身の待ち行列に入れる
  - (3) もし受信者がリソースにアクセスしようとしているのであれば、
    受信メッセージのタイムスタンプと自身が出した要求のタイム
    スタンプを比較。値が小さい方を優先する
    (OKを返すか、待ち行列に入れる)

分散アルゴリズム



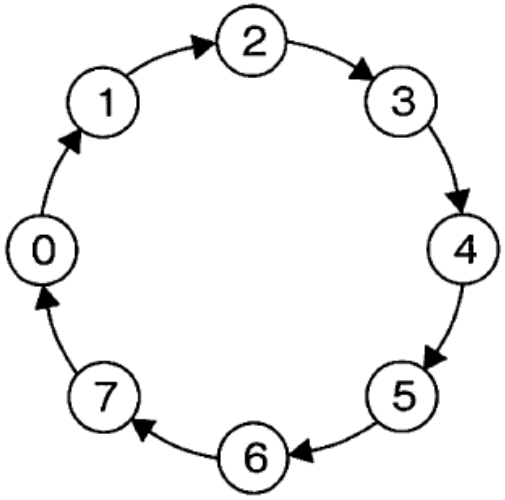
- プロセス0とプロセス2が同時にリソースを要求
  (プロセス0のタイムスタンプは8、プロセス2のタイムスタンプは12)
- プロセス0, 2ではタイムスタンプの値が小さいプロセス0を優先
  プロセス0の待ち行列にプロセス2が入る
  プロセス1はプロセス0, 2両方にOKを送信
- プロセス0のリソース使用終了後、プロセス0は待ち行列に従い、
  プロセス2にOKを返す

分散アルゴリズム

- 長所
  - 飢餓やデッドロックは発生しない
- 短所
  - 故障n箇所性
    - n個のうち1個でも壊れると処理が進まない
      (OKが返って来なくなる)
      → OKかNG(拒否)を返答するようにすれば改善
      (一定時間返答がなければ相手がダウンしたと分かる)
  - マルチキャストで通信する必要
    (or 全メンバのリストを持っていないといけない)
  - 集中型に比べて、複雑で遅い

トークンベース排他制御

- トークンリングアルゴリズム
  - トークン(token)がリングを巡回する
  - トークンを取得したプロセスが
    リソースへアクセス可能



  - 長所
    - 飢餓やデッドロックは発生しない
  - 短所
    - 待ち時間が長くなる可能性はある
    - トークンの喪失の検出が困難
      - トークンを渡した際に確認通知を送るようにすることで対策可
        → ダウンを検出すると、次を飛ばしてトークンを渡せば良い

選任アルゴリズム

- 集中アルゴリズム等におけるコーディネータを
  どのように選ぶか

- ブリーアルゴリズム (bully algorithm)
  - あるプロセスPが、自分より高い数値(e.g., プロセス番号)を
    持つすべてのプロセスに対してELECTIONメッセージを送信
  - もし誰からも応答がなければPがコーディネータとなる
    (Pは各プロセスに自分がコーディネータだと伝える)
  - Pより高い数値のプロセスから応答があれば交代
- リングアルゴリズム (ring algorithm)
  - あるプロセスPが、自分の数値を付けて、リングの次のプロセスに
    対してELECTIONメッセージを送信
  - ELECTIONメッセージを受信すると、自分の方が数値が高ければ、
    自分の数値を付加して送信。そうでなければそのまま送信
  - 自分の数値が付いたメッセージを受信し、それが最大の数値で
    あれば、コーディネータとなる

確認問題

- 次の説明に合う語句を答えよ。
  - プロセスが必要なリソースにアクセスできない状態
  - 複数のプロセスが互いの終了を待ち、いずれも先に進めなくなる状態
- 以下の文は正しいか。○か×で答えよ。
  - 分散アルゴリズムにおける故障n箇所性は、集中アルゴリズムより耐故障性が高いことを意味する。

参考文献

- 「分散システム」
  水野 忠則 監修、共立出版、2015
- 「分散システム 原理とパラダイム 第2版」
  アンドリュー・S・タネンバウム 他 著、
  ピアソン・エデュケーション、2009