分散システム

第6回 同期

双見 京介（FUTAMI Kyosuke）
高田秀志（TAKADA Hideyuki）

2025年11月

はじめに

- 分散システムにおいてプロセスがどのように同期を行うのかは重要
  - 複数のプロセスが同時に共有資源にアクセスすることは許されない
  - 複数のプロセスはイベントの順序（例えば，どのメッセージが先に送られたのか）について合意することが必要
- 分散システムにおける同期は，単一プロセッサやマルチプロセッサのシステムにおける同期に比べて，かなり困難であることが多い

時計の同期

- プロセスは，OSへの関数呼び出しによって時刻を知ることができる
  - 集中システムにおいては，時刻は呼び出しの順に並ぶ
  - 分散システムにおいては，マシン間の時計のずれにより，時刻が呼び出し順に並ばないこともある
- 各マシンが自身の時計を持っている場合，あるイベントの後に起こったイベントに，より早い時刻が割り当てられることがあり得る
- 分散システムにおいて，すべての時計を同期させることは可能なのだろうか？

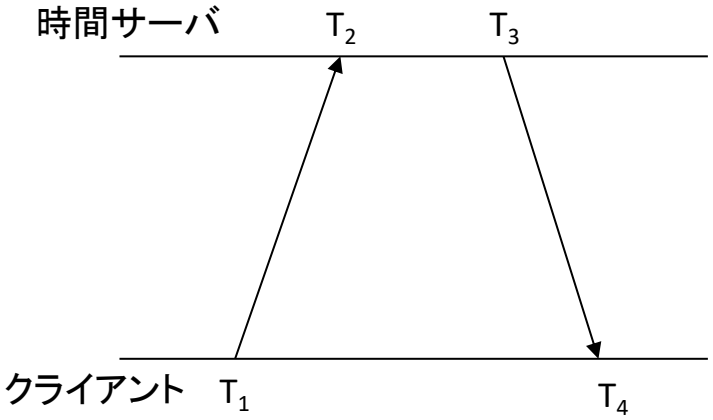
物理（Physical）クロック

- コンピュータには，時刻を保持する回路（「クロック」（clock）と呼ぶ）が内蔵されている（実は「タイマ」（timer）の方が適した語）
- タイマは1秒間に決まった回数の割り込みを起こし，割り込み処理により，ソフトウェア的に保持しているクロック（チック数）に1を加える
- クロックは，特定の過去の日時（1970年1月1日0時0分など）からのチック数を表す
- 時計のズレを表す最大ドリフト率を ρ とすると，2つの時計は，最後に同期してから Δt が経過すると，最大 $2\rho\Delta t$ だけずれる
- システムとして時計のズレを δ 以内に抑えたい場合は，最低 $\delta/2\rho$ 毎に同期しなければならない

Network Time Protocol (NTP)

- クライアントは，正確な時刻を提供可能なサーバに問い合わせを行う
- クライアント側の相対的な時刻のずれ θ は，以下の式により計算できる

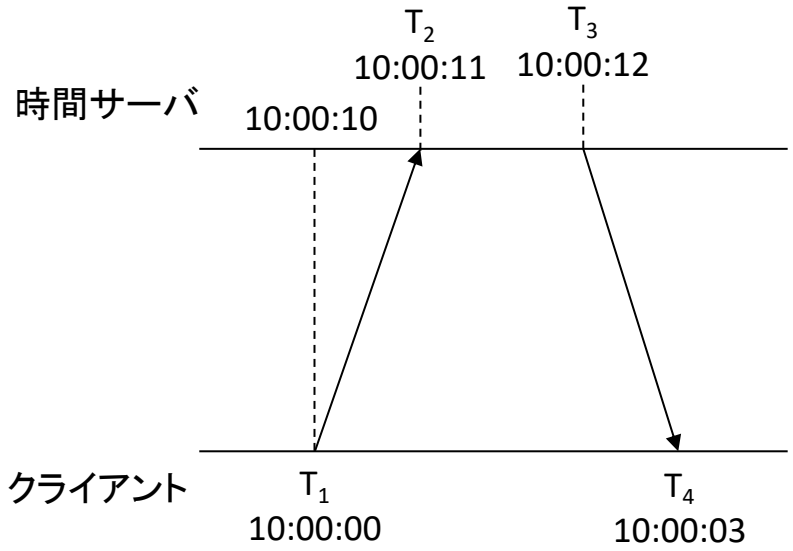
$$\theta = T_2 - \frac{(T_2 - T_1) + (T_4 - T_3)}{2} - T_1 = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}$$



- クライアントは，自身の時刻印 T_1 とともにサーバに要求を送信
- サーバは，自身の時刻印 T_2（受信時刻）を記録し，T_2 と送信時刻 T_3 をクライアントへ返信
- クライアントはサーバからの返信の受信時刻 T_4 を記録

時刻合わせの例

クライアントのクロックがサーバよりも10秒遅れていると仮定



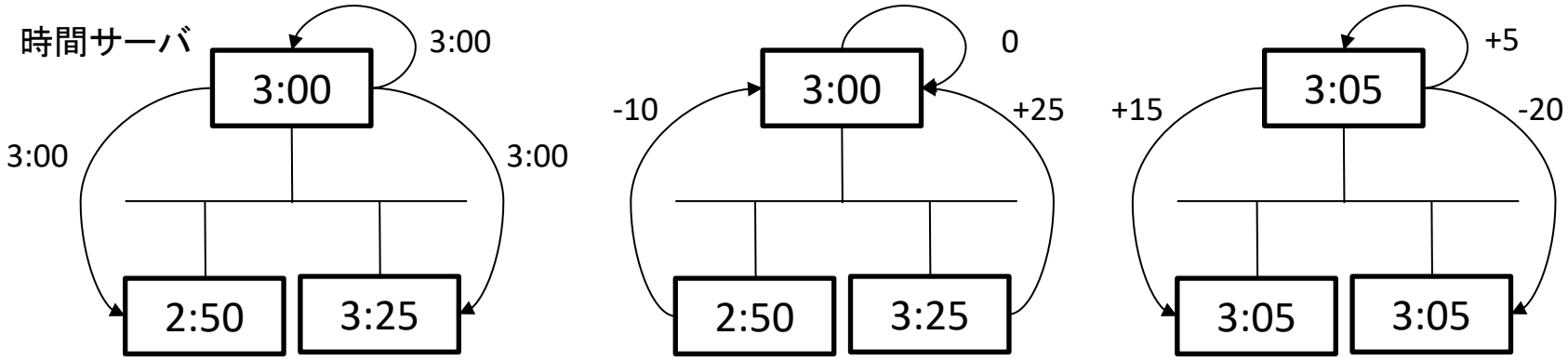
例えば
- クライアントとサーバ間のメッセージ送信に1秒かかる
- サーバ側で要求を処理するのに1秒かかる

とした場合には，以下のように計算できる

$$\theta = \frac{(T_2 - T_1) + (T_3 - T_4)}{2} = \frac{(10{:}00{:}11 - 10{:}00{:}00) + (10{:}00{:}12 - 10{:}00{:}03)}{2}$$
$$= 10$$

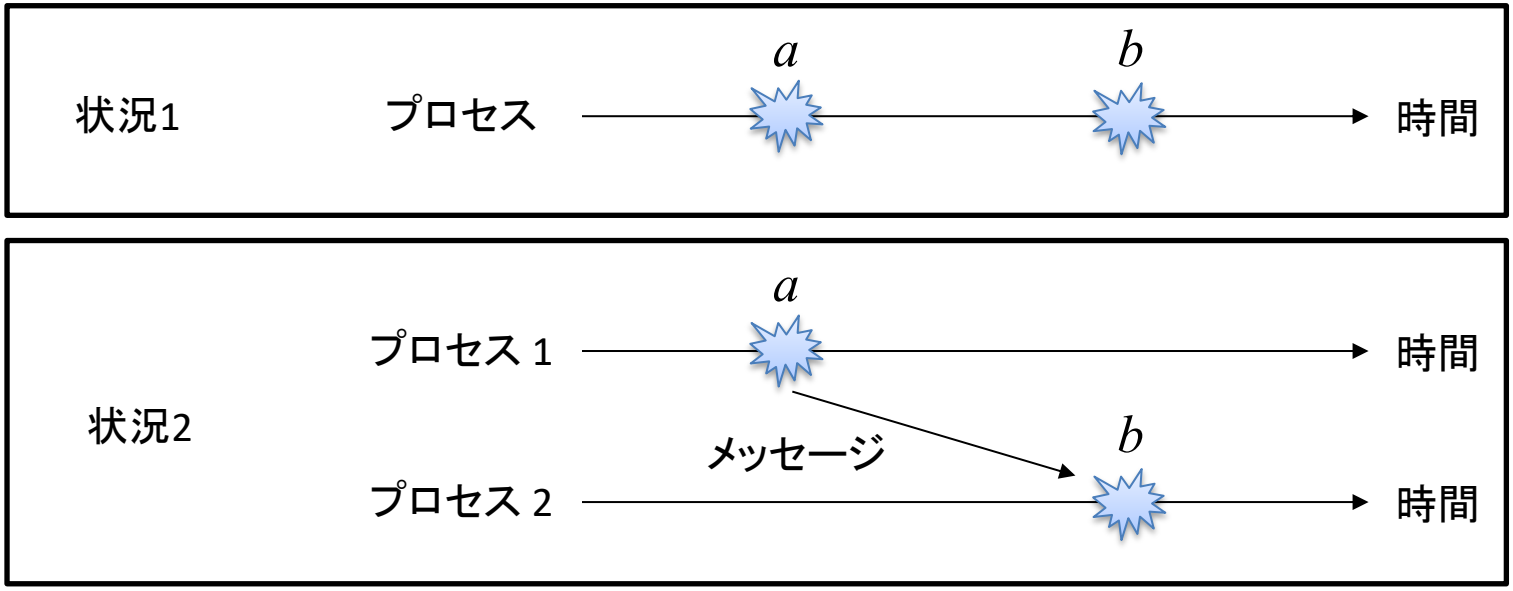
Berkeleyアルゴリズム

- 正確な時刻を保持しているマシンが存在しない場合に利用
- 動作手順は以下のとおり
  (a) 時間サーバは定期的にすべてのマシンに対して保持している時刻を知らせるように要求
  (b) 時間サーバは，各マシンからの返答に基づき，時刻の平均を取る
  (c) 時間サーバは，各マシンに対してどれだけ時刻を補正すべきかを送信
- 時間サーバの時刻は管理者により定期的に手動で修正

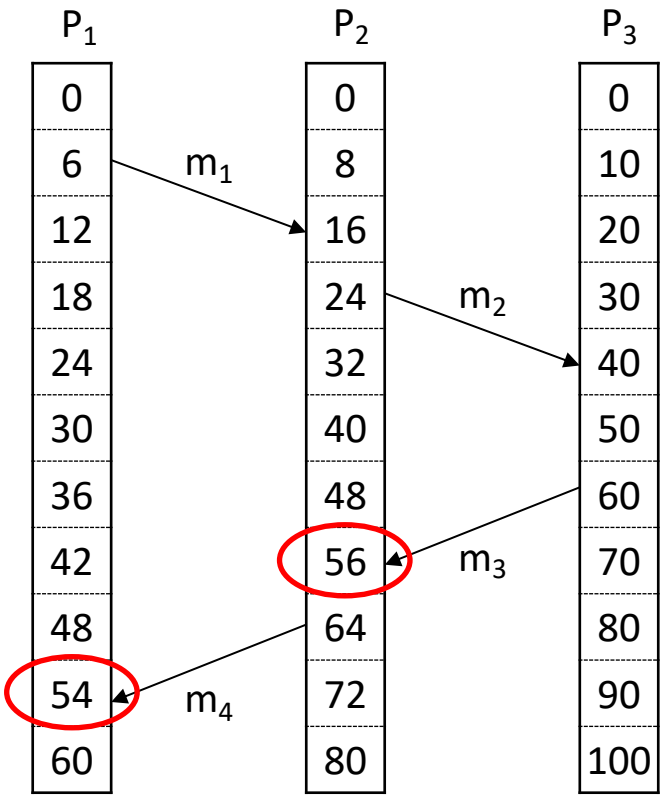


論理（Logical）クロック

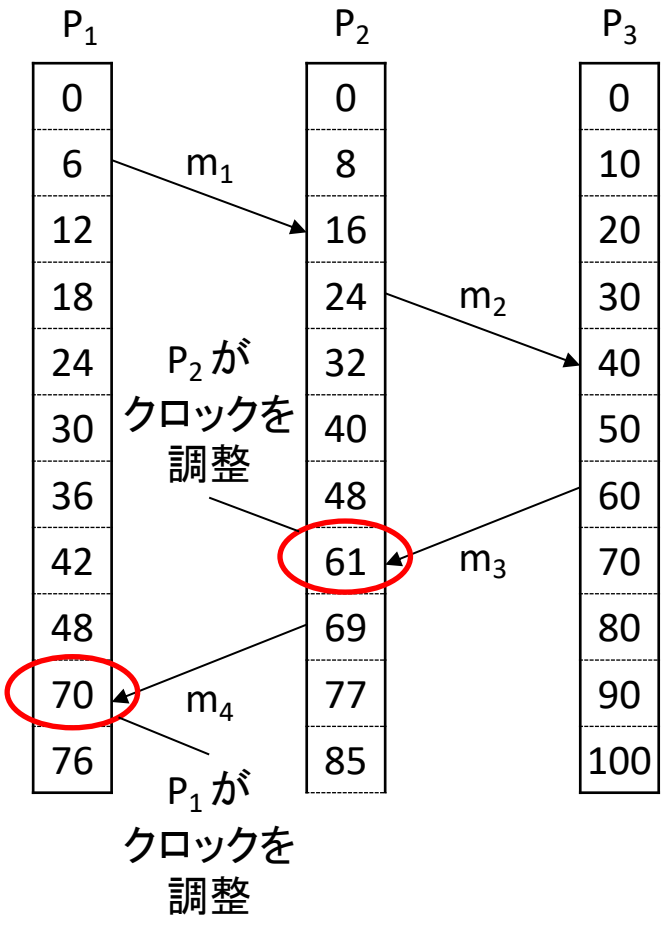
- 複数のプロセスが「イベントの発生した順序」に合意する
- **Lamportの論理クロック**
  - 事前発生（***happens-before***）という関係を定義
  - $a \rightarrow b$ は「a は b より前に発生」と読む



イベントへの時刻の割り当て

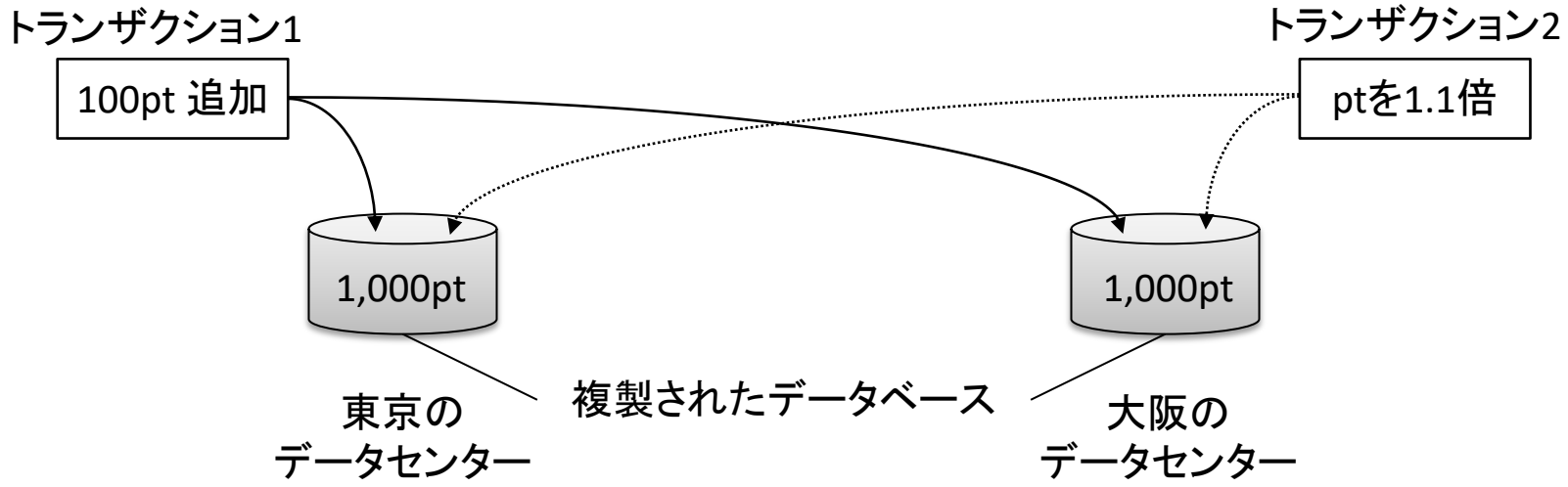


クロックの進み具合が少しずつ異なる状況で，3つのプロセスがメッセージの送信をしあうと，例えば「時刻60に送ったメッセージが時刻56に到着する」（過去に到着する）ということが発生する



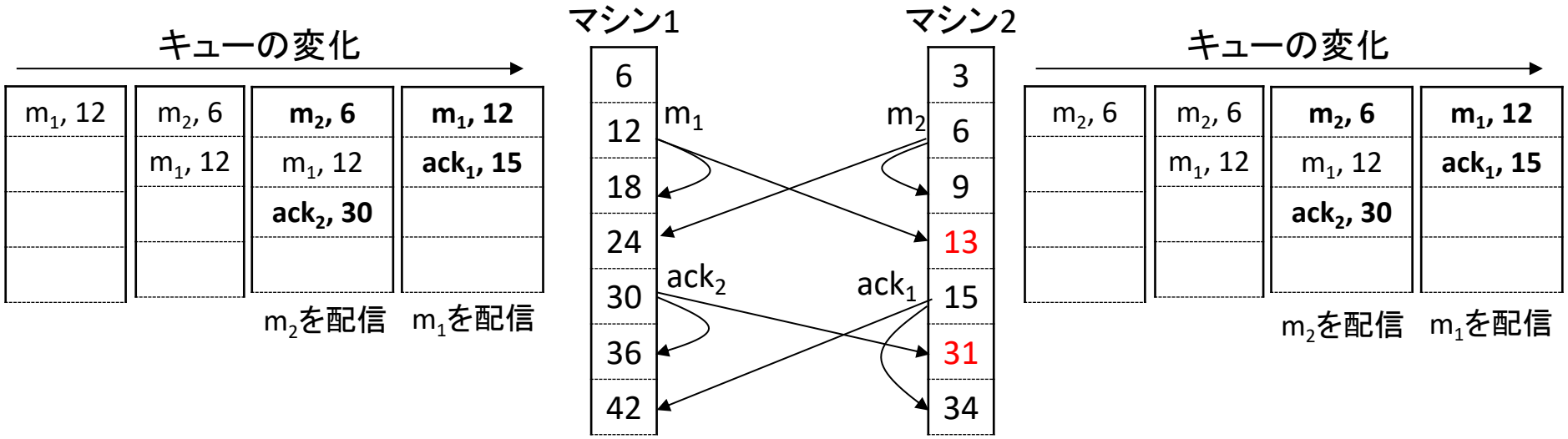
Lamportのアルゴリズムによりクロックを修正

イベントの順序付け



- 地理的に離れたデータセンターに複製されたデータベースを想定
- ほぼ同時に複数のトランザクションが発生すると，実行順序により結果に不整合が生じる可能性がある
  - 東京：100pt追加→1.1倍→結果は1,210pt
  - 大阪：1.1倍→100pt追加→結果は1,200pt
- すべてのデータセンターで同じ順序でトランザクション（イベント）を実行する必要がある

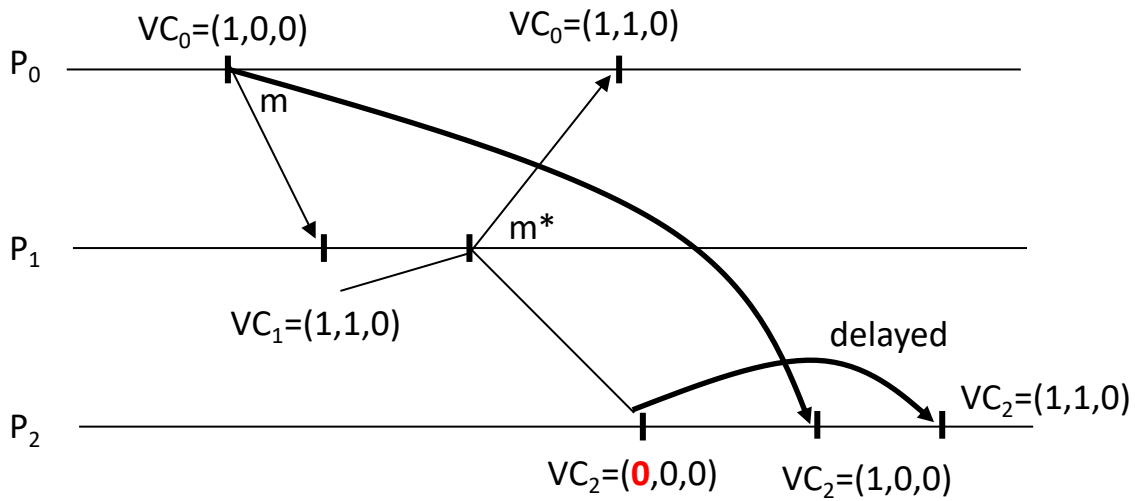
全順序マルチキャスト



- 各マシンは，メッセージとその受信通知を論理クロックとともにすべてのマシンに送る
- 各マシンは，自身のキューにメッセージと受信通知を論理クロック順に並べる
- すべてのマシンからの受信通知が届いたメッセージをキューから取り出してアプリケーションに配信する

ベクタークロックと因果通信

- イベント間の「**因果関係**」をとらえる
- 因果的に先に受信されているべきメッセージがすべて受信された場合にのみメッセージを配信する



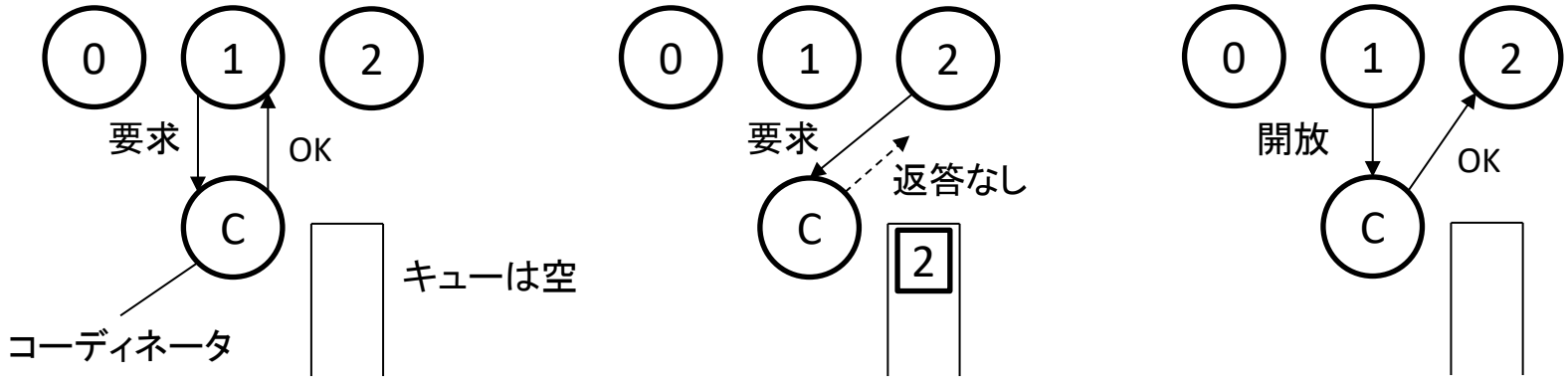
P_0 は時刻(1,0,0)にメッセージ m を他のプロセスに送信。P_1 はそれを受信後，m*を送信することになり（ m の受信と m* の送信は因果関係がある），それは m よりも先にP_2 に配信された。この時点で，m* の P_2 への配信は m が受信されるまで延期される。

相互排他（Mutual Exclusion）

- 共有リソースに対して，複数のプロセスが同時にアクセスすることを防止
- 単一マシンにおいては，セマフォによる制御が可能
- 分散システムにおいては，プロセス間のメッセージ通信を利用して実現する必要がある

集中アルゴリズム

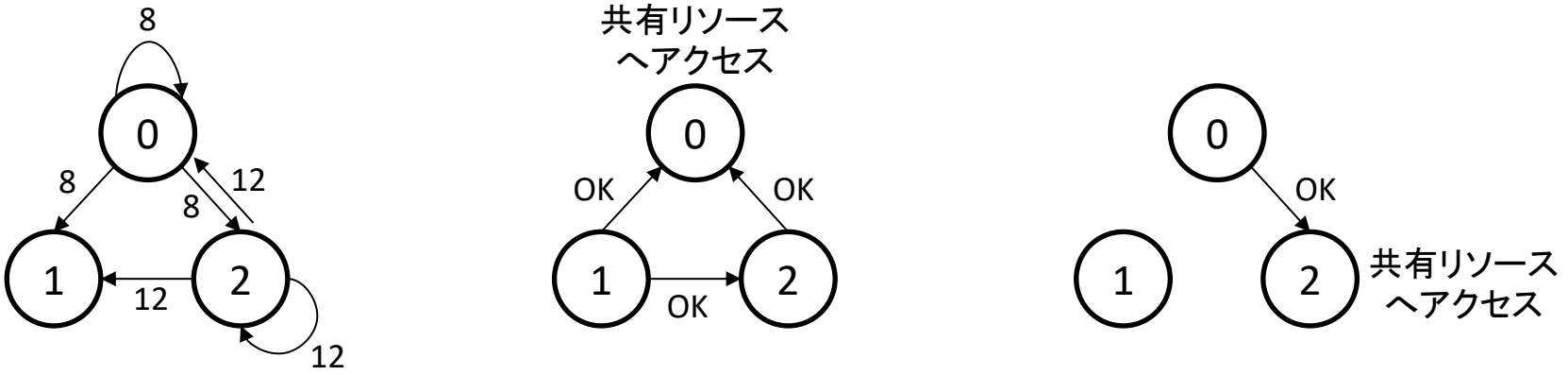
- 単一プロセッサによる相互排他を模擬
- コーディネータが故障するとアルゴリズムが停止するため，コーディネータが「単一障害点」（***single point of failure***）になる



(a) プロセス1がコーディネータに共有リソースへのアクセスを要求。許可される。
(b) 次に，プロセス2が同じ共有リソースへのアクセスを要求。コーディネータは**返答せず**（アクセスが許可されないことを返答するのではないことに注意），プロセス2をキューに入れる。
(c) プロセス1が共有リソースを開放すると，コーディネータはプロセス2にアクセス許可を返答。

分散アルゴリズム

- コーディネータの存在を仮定しない
- いずれかのプロセスが故障するとアルゴリズムが動作しなくなるという「故障n箇所性」（***n points of failure***）が存在



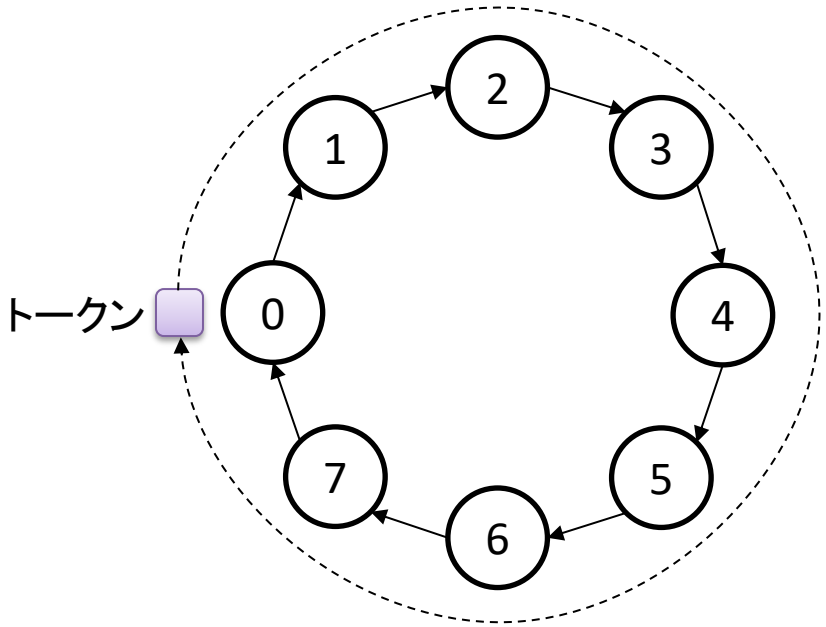
(a) プロセス0とプロセス2が同時に共有リソースをアクセスしようとし，それぞれが自分の論理時刻をブロードキャスト。

(b) 各プロセスは，アクセス要求を許可する場合にはOKメッセージを返信。この場合，プロセス0がより小さい時刻印（8）を持つので，プロセス2よりも先に許可が与えられる。他のすべてのプロセスから許可が与えられると，共有リソースを使用できる。

(c) プロセス0が共有リソースを使用後，プロセス2にOKメッセージを返信。

トークンリングアルゴリズム

- それぞれのプロセスを番号付けしてリング状に配置
- トークンが失われたときや，プロセスが故障したときへの対処が必要



- トークンがリング上を巡回
- 各プロセスは，隣のプロセスからトークンを受信した際，共有リソースへのアクセスが必要かどうかを調べる
- 共有リソースへのアクセスが必要な場合，処理を続け，終了後，共有リソースを開放
- その後，トークンを次のプロセスへ渡す
- 隣のプロセスからトークンを受信した際，共有リソースへのアクセスが必要でない場合は，単にトークンを次のプロセスへ渡す

非集中アルゴリズム

- 各リソースをN個に複製。各レプリカは，アクセスを制御するためのコーディネータを持つ。
- プロセスは，半数（$m > N/2$）を超えるコーディネータから許可が必要。
- コーディネータがクラッシュした場合には，すぐに復旧できるが，クラッシュ前に与えた許可は失われることを想定。つまり，他のプロセスに対して不正な許可を与える可能性がある。
- 一方で，共有リソースへの不正なアクセスが発生する可能性は無視できるほど小さい（例：N=8, m=6, stops 3 秒/時停止の場合，10^{-18} 以下）。

相互排他アルゴリズムの比較

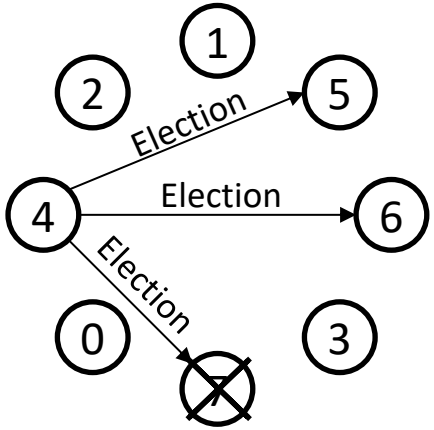
アルゴリズム	アクセス・開放に必要なメッセージ数	アクセスまでに必要なメッセージ数	問題
集中	3	2	コーディネータの故障
分散	$3(n\text{-}1)$	$2(n\text{-}1)$	いずれかのプロセスの故障
トークンリング	1 to ∞	0 to n-1	トークンの喪失, プロセスの故障
非集中	$2mk+m$, k=1,2...	$2mk$	コーディネータの故障

- 集中アルゴリズムが最も単純で効率が良い
- いずれのアルゴリズムもプロセスの故障に対して対応が必要

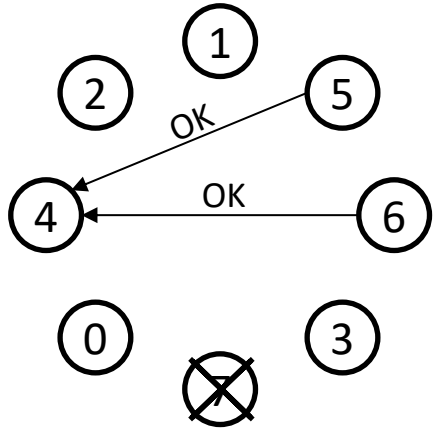
選任アルゴリズム

- 多くの分散システムにおいて，コーディネータなどの特別な役割をもつプロセスが必要
- それぞれのプロセスが一意に識別可能な番号を持つと想定し，選任アルゴリズムによってその中で一番大きな番号を持つもの見つけ出し，コーディネータとする
- 各プロセスは他のプロセスの番号を知っているが，どれが動作中で，どれが停止中かを知らないと仮定

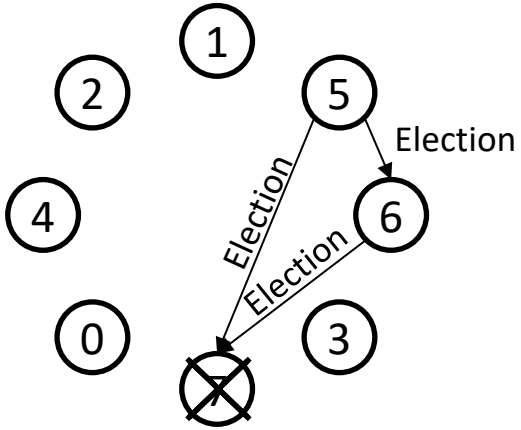
ブリー（bully）アルゴリズム



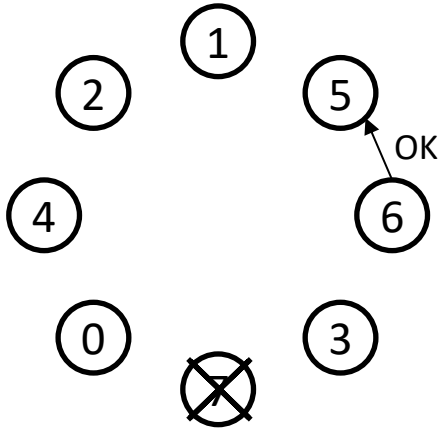
(a) コーディネータが動作していないことを感知したプロセスは，選任を起動。例えば，プロセス4が自分より大きな番号のプロセスへElectionメッセージを送信。



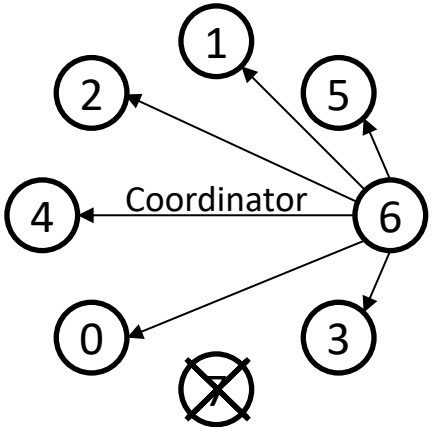
(b) 自分より小さい番号を持つプロセスからElectionメッセージを受信した場合は，OKメッセージを返信。この場合，プロセス5とプロセス6からの返信により，プロセス4は動作を停止。



(c) プロセス5と6が選任を起動し，Electionメッセージを送信。



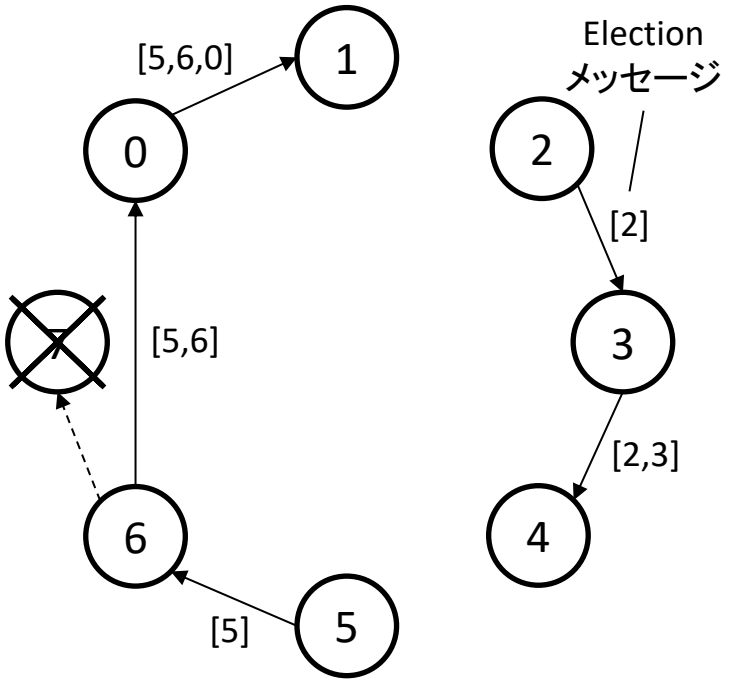
(d) プロセス6からの返信によりプロセス5が停止。



(e) 他のどのプロセスからも返信を受信しなければ，コーディネータとして選任されたものとして，それを他のプロセスに宣言。

リングアルゴリズム

- 最初のプロセスは，自身のプロセス番号を含んだElectionメッセージを隣のプロセスに送信
- もし隣のプロセスが停止している場合は，リング上のその次のプロセスに送信
- 各プロセスは，受信したメッセージに自身のプロセス番号を追加して隣のプロセスに送信
- メッセージが最初のプロセスに戻ってきたら，メッセージの種別を”Coordinator”に変更し，もう一度メッセージをリング上で回して，各プロセスにどのプロセスがコーディネータかを知らせる



- プロセス2と5が，前のコーディネータだったプロセス7が停止していることを発見
- プロセス2と5は，Electionメッセージをリング上で回し始める
- メッセージが一周したら，Coordinatorメッセージに変更してリング上で回す