同期
「クロック、論理クロック」

佐藤

分散システムのクロック（時計）

- 単一システムでは、時計は一つ
- 分散システムでは、それぞれのコンピュータに時計があるために、同じ「時計」を用いることは困難

- 物理クロック
  - 実際の（計算機の中の）時計

- 論理クロック
  - 物事の順序をあらわす「時計」

何が困るか、例えば…

- makeコマンド
  - makeコマンドでは、xxx.oとxxx.cの更新された時刻を比較して、xxx.oの方が古ければ、コンパイルする
  - 同じマシン上で仕事をしている分には、困らない
  - だが、NFSでファイルシステムを共有していて、xxx.cのファイルを更新するマシンAとxxx.oがあるマシンBの時刻が、異なっていたら、このコマンドはうまくうごかない。
  - (計算機にもGPSがほしい…)
  - NTPD (Network Time Protocol Demon)で時計を合わせる

物理クロック

- 各マシンでは、時計を持っている
  - 水晶発振(Quartz Crystal), Line Clock (50Hz/60Hz)
  - カウンターを読む、一定時間で割り込みなど
- 必ず、マシンによってクロックのずれ(Clock Skew)が生じる
- 問題点
  - 物理クロックを現実の正確な時間と合わせること
  - 複数のクロックの時刻を合わせること

物理クロック

- 実際の時間とは…
  - 天文学的時間
    - 太陽日（南中間の時間差）1/86400を1秒として、計算
    - 地球の自転周期が一定ではない（うるう秒）
  - 原子時計の時間
    - セシウム133原子の状態遷移の数を数えることで計測（9,192,631,770回の状態遷移を1秒と定義）
    - International Atomic Time (TAI)（国際原子時）
  - うるう秒によって修正された時刻: Universal Coordinated Time (UTC)（協定世界時）
    - TAIと太陽秒の差が800ミリ秒に達する毎に、TAIにうるう秒(leap second)を導入
    - 昔はGMTだったが、いまはUTC

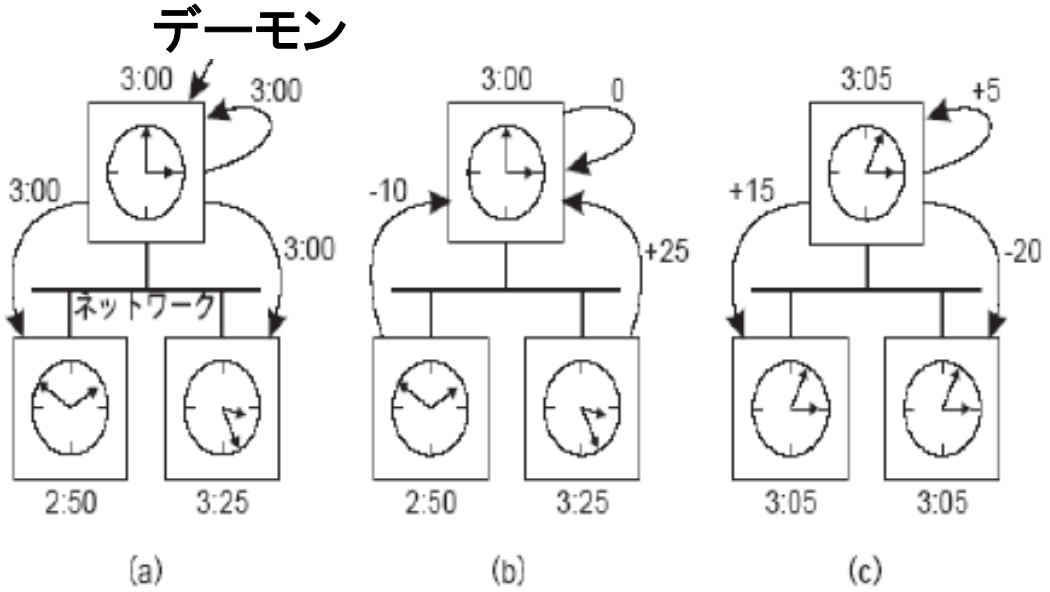
Cristianのアルゴリズム

- 一つのマシン(時間サーバ(time server))が正確な時刻（UTC）を持っている
- 他の全てのマシンをそのマシンの時刻に同期

- 手順
  - 各マシンは定期的に時刻サーバに時刻を問い合わせる
  - 時刻サーバは問い合わせに対して現在時刻をできるだけ早く応答
  - クライアントは得られた時刻を用いてクロックを設定
  - そのまま設定すると時刻が後戻りする可能性があるので時間を遅くしながら（加える刻みを調整）合わせる
  - 通信遅延を(T1-T0)/2で、相殺

Berkeleyアルゴリズム

- 正確な時刻サーバを仮定しない
- 時間サーバが他の全てのマシンに対して定期的にメッセージを送信し、それらのマシンの現在時刻の情報を集計
- 各マシンの時刻の平均値を計算し、各マシンに対してどれだけ時計を進めればよいか、または、遅らせればよいかの情報を通知



平均アルゴリズム

- 分散型(decentralized)
- すべてのマシンのクロックを一度同期化
- ある一定間隔ですべてのマシンがクロックを同報通信
- 同時に他のマシンからのブロードキャストを一定期間受信
- すべての値から新しい時刻を計算
  - 平均値
  - m個の最高値と最低値を除いた平均値
  - 伝播時間の概算値を用いた補正

- 最も普及しているアルゴリズム：Network Time Protocol(NTP)

論理クロック

- 実際の時刻と無関係に、システム内部での一貫性を持つ時計
  - 事象の順序を表す「時計」
  - 事象（イベント）の発生順序
    - makeコマンドの例では、ファイルが修正された時刻の前後関係のみが重要（時刻の値自体は関係ない）
- Lamportの論理クロック
- Vector Clock

事前発生関係(happens before)

- 論理クロックを同期させるために、分散システムのイベント間に“事前発生（happens before）”関係を定義
- 以下の場合に a→b (=「aはb以前に発生」) が成立
  - 1. a,bがともに同じプロセスによるイベントで、かつ、aはbより前に起こるとき（プロセス内での実行順序）
  - 2. aがあるプロセスによるメッセージ送信イベントで、bは別のプロセスによるそのメッセージの受信イベントであるとき（送受信による因果関係）
- a→bは推移的関係: a→bかつb→cならばa→c

クロック時刻値 C(a)

- どのイベントaに対してもすべてのプロセスで一貫した時刻値C(a)を割り当てる
- a→bならばC(a)<C(b)
- クロック時刻Cは常に前進していて決して後退しない
- 時刻の修正はクロックに正の値を加えて修正し、値を引いてはいけない

例

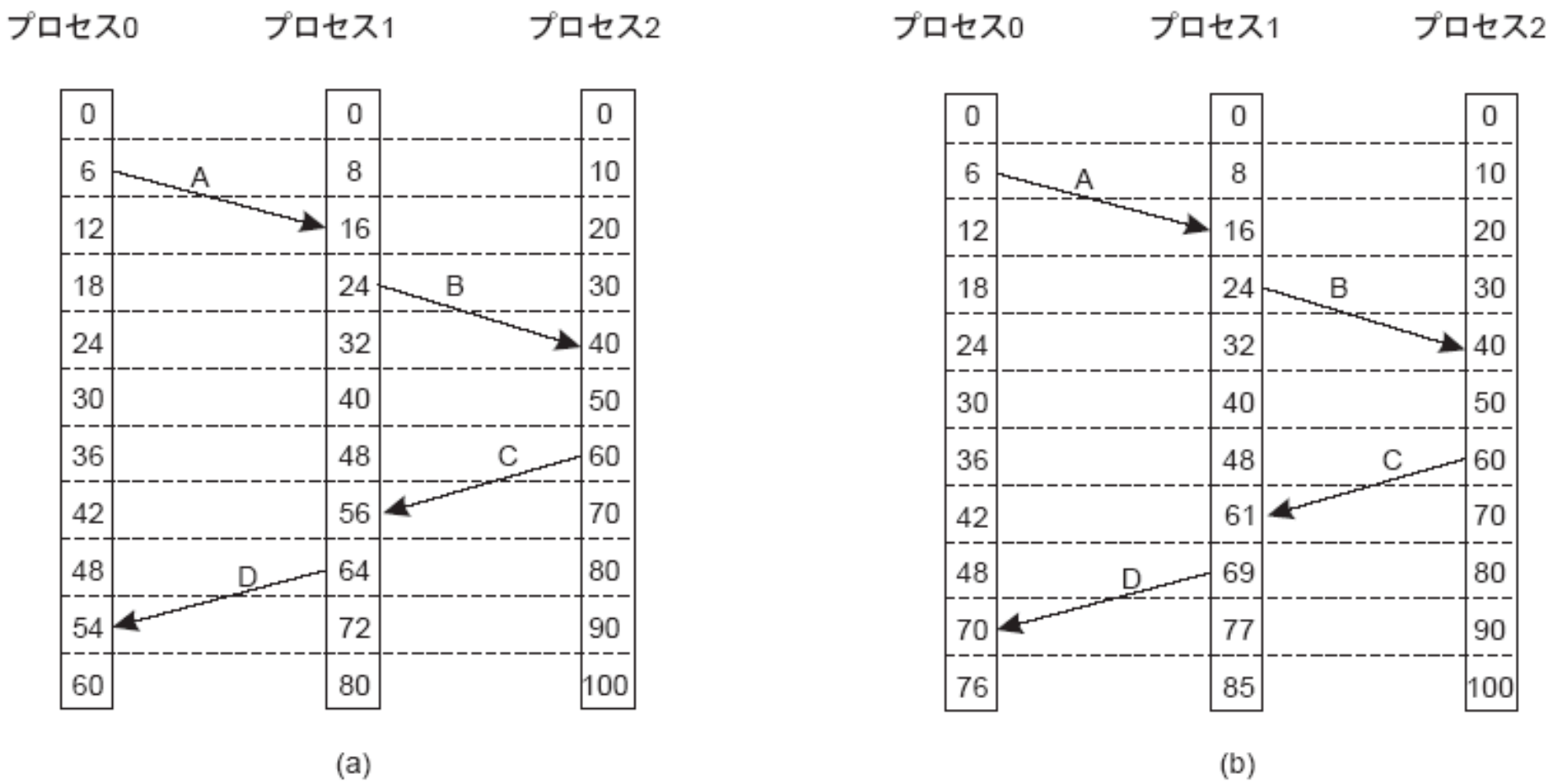


図 5-7　（a）それぞれが自己のクロックをもつ 3 つのプロセス。クロックは異なる速度で動作している、（b）Lamport のアルゴリズムがクロックを修正する

Lamportの論理クロックのアルゴリズム

- それぞれのプロセスPiは、ローカルなカウンタＣｉを以下のように管理する
  - １．イベント（メッセージの送受信等）を実行する前には、カウンタを１増やす。すなわち、Ci ←Ci+1
  - ２．Piがメッセージmをに送信する時、それ以前ステップを実行した後のCiと等しい、タイムスタンプts(m)をメッセージmに付加する
  - ３．メッセージmを受け取ったプロセスPjは、ローカルなカウンタCjをCj←max(Cj, ts(m))とする。すなわち、メッセージの送信時刻と自分が受信した時刻を比較し、大きい（早い）方にCjをセット

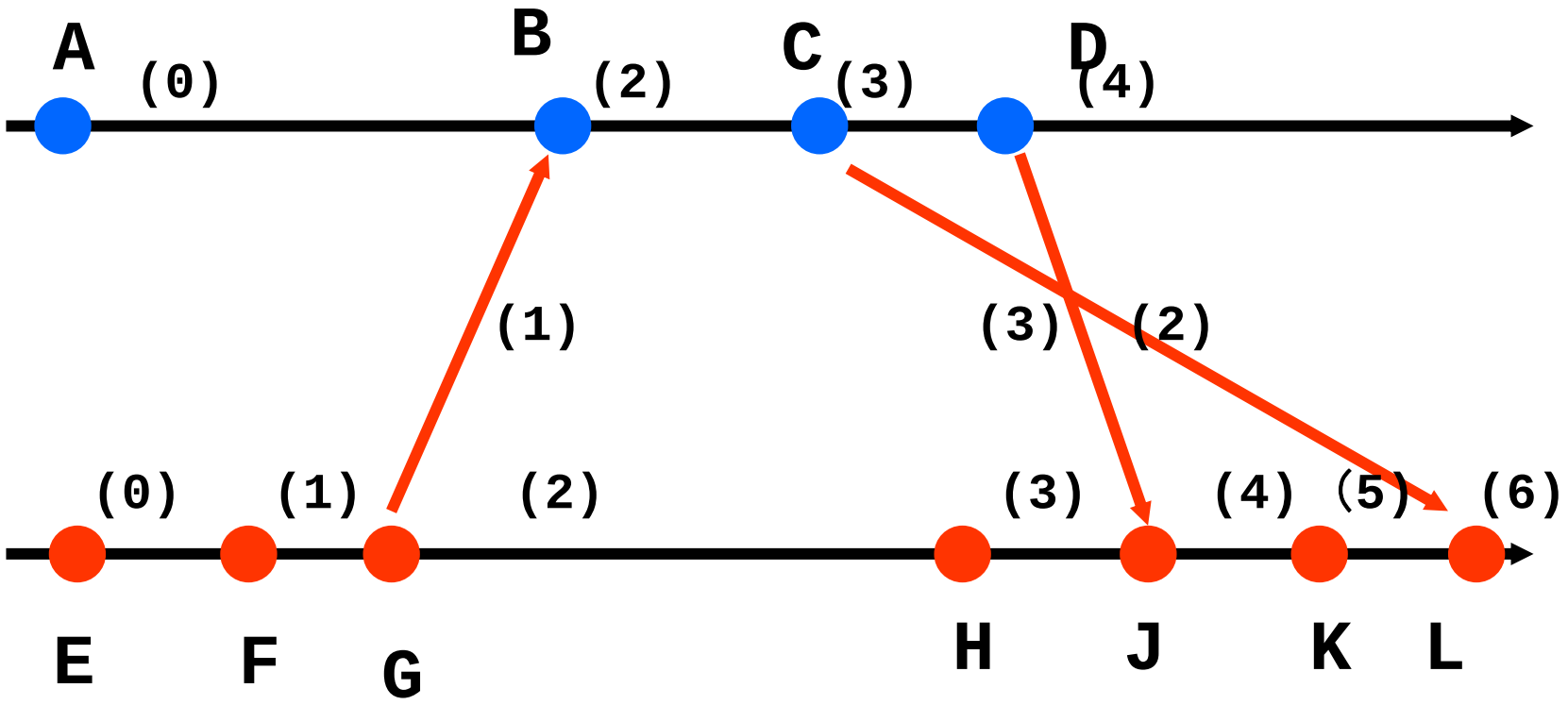
http://funini.com/kei/logos/clock.shtmlより

例

- メッセージのやりとりが無いとき、クロックはイベントの発生に従って単調増加する
- メッセージを送信するときは、送信直前のクロックを添付して送り、その後クロックを増加させる
- メッセージを受信したら、メッセージのクロックと受信直前のクロックを比較し、大きい方より大きい値を次のクロックとする（そして、1加える）

◆ Lamportの論理クロック

ここで、(0)とメッセージのスタンプ(1)と比較して(1)、それに1を加えると(2)



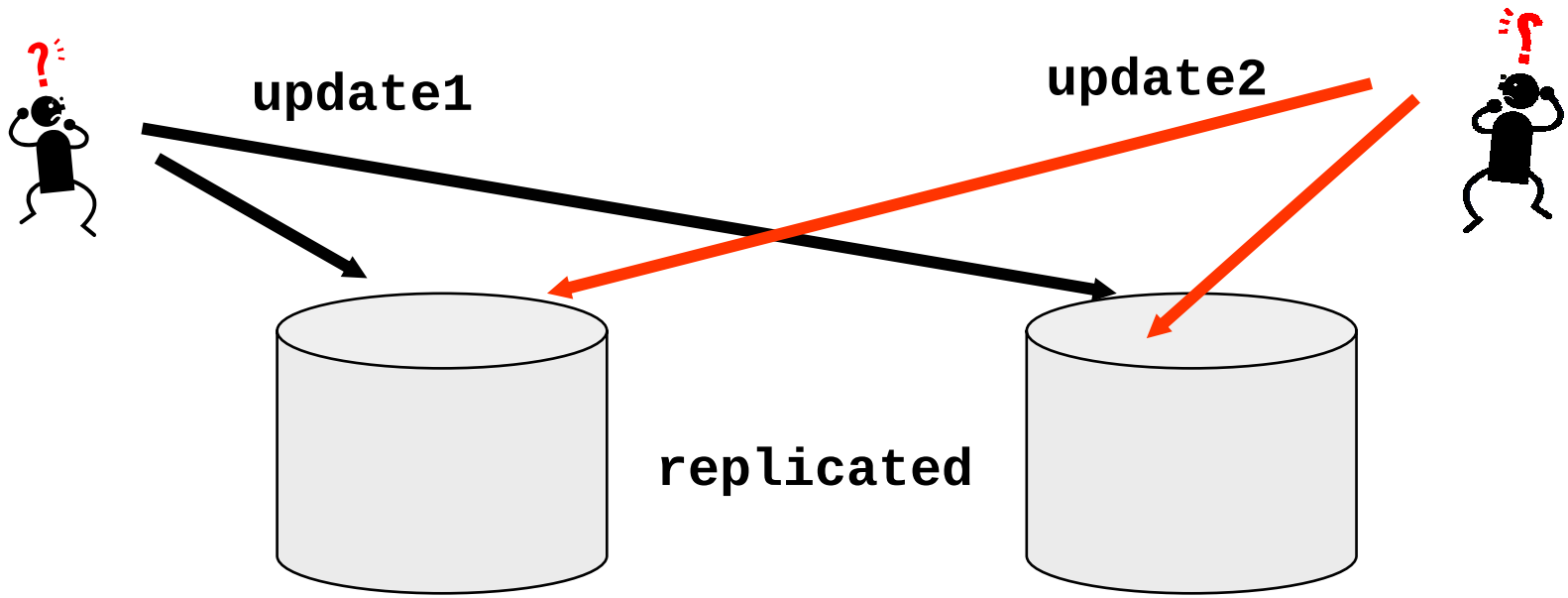
便利な性質

- a≠bならばC(a)≠C(b)であることを保証
  - 1イベントの処理に少なくとも1 tickかかる
- 分散システムの全イベント間に全順序をつけることが可能
  - 多くの分散アルゴリズムでは、曖昧性を避けるためにイベントの全順序を仮定する必要

例: Totally Ordered(全順序)マルチキャスト

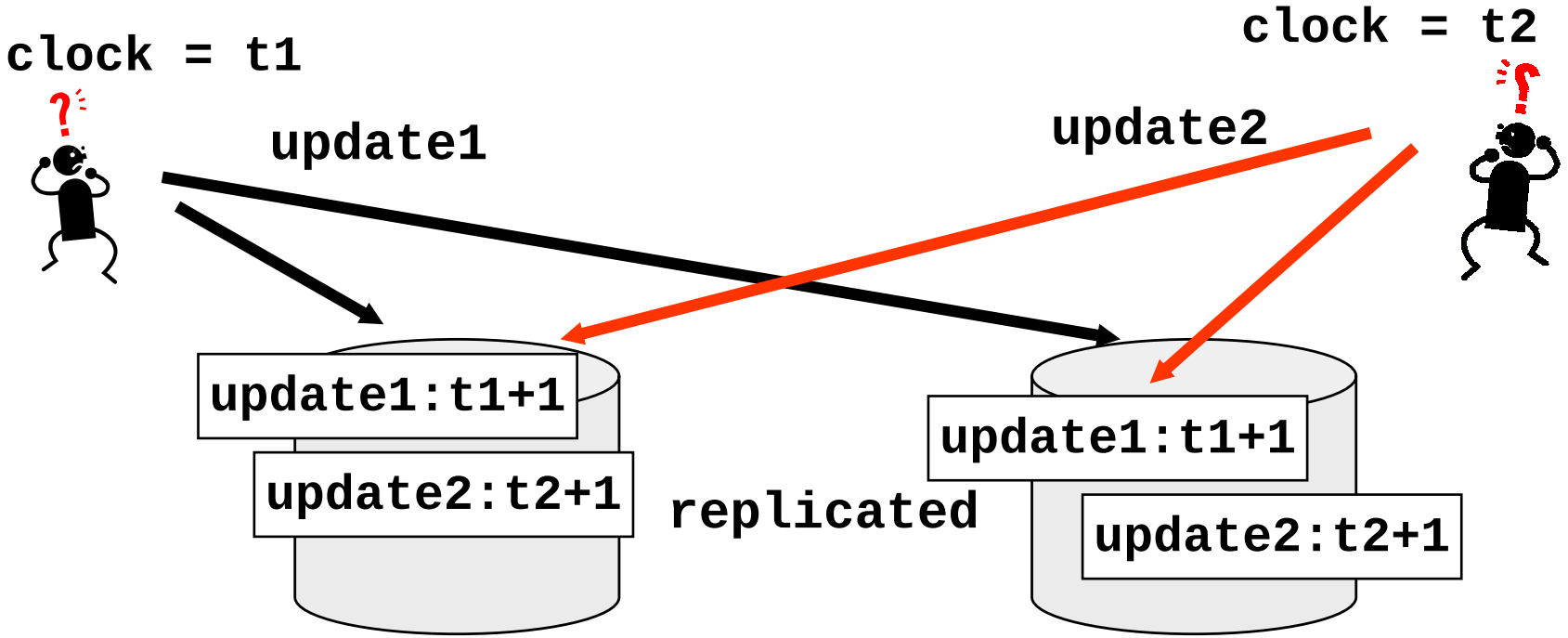
- 2つのデータベース
  - 検索は近い方で更新
  - その代わり更新にコストがかかる(両方のデータベースを一貫性を保ちつつ更新する必要)

ある口座に現在$1,000の残高があり、サンフランシスコからその口座に$100の振込があったとする
同時に、ニューヨーク支店の銀行員がその口座に1%の利息を加えようとしたとする
もし2つのデータベースで上記の更新操作の順序が異なると、2つのデータベースの一貫性が保てない
サンフランシスコでは$100の振込の後、利息1%加算
　残高は$1,111.-
ニューヨークでは利息1%加算の後、$100の振込
残高$1,110.-



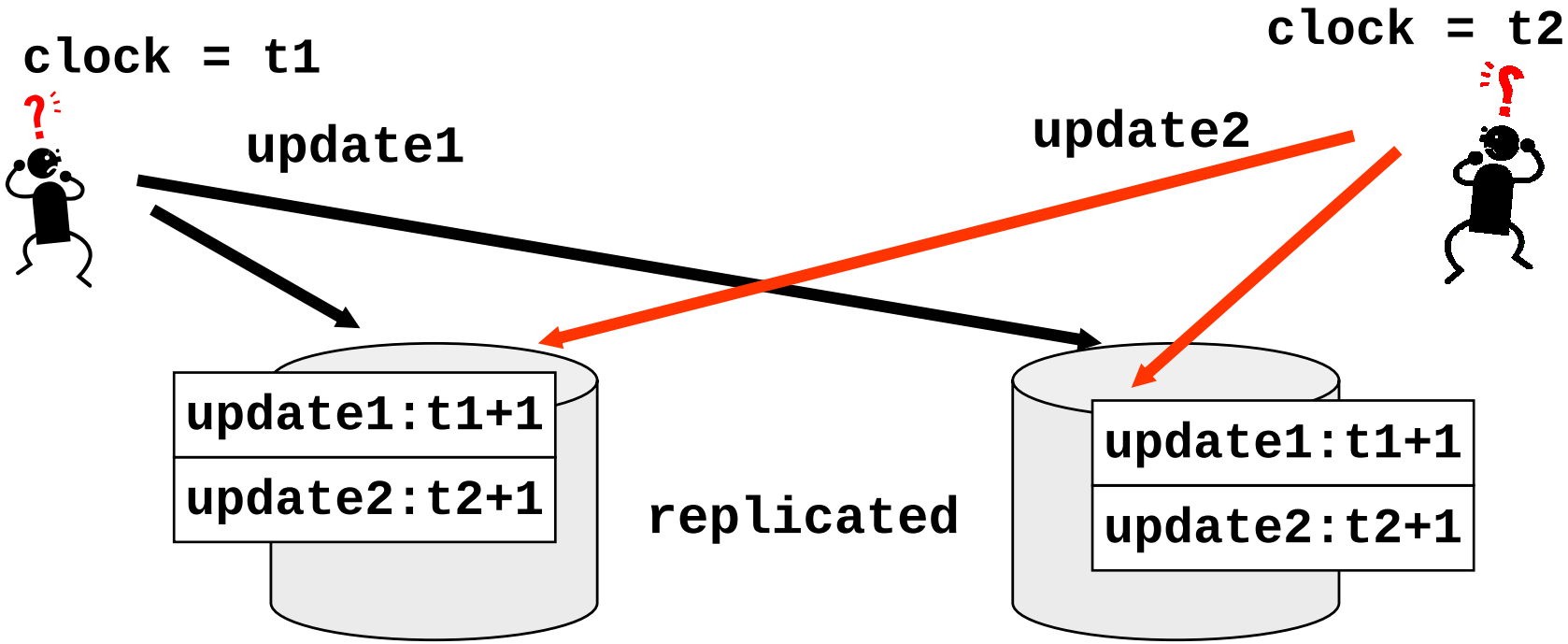
Totally Ordered(全順序)マルチキャスト

- マルチキャストを行うgroupがあることを前提
- マルチキャストを行うsenderのlocal clockで、メッセージにtimestampを付加
- 仮定
  - senderにも同じようにメッセージが送られる
  - senderからのメッセージは同じ順序で到着する
  - メッセージのlostはない



Totally Ordered(全順序)マルチキャスト

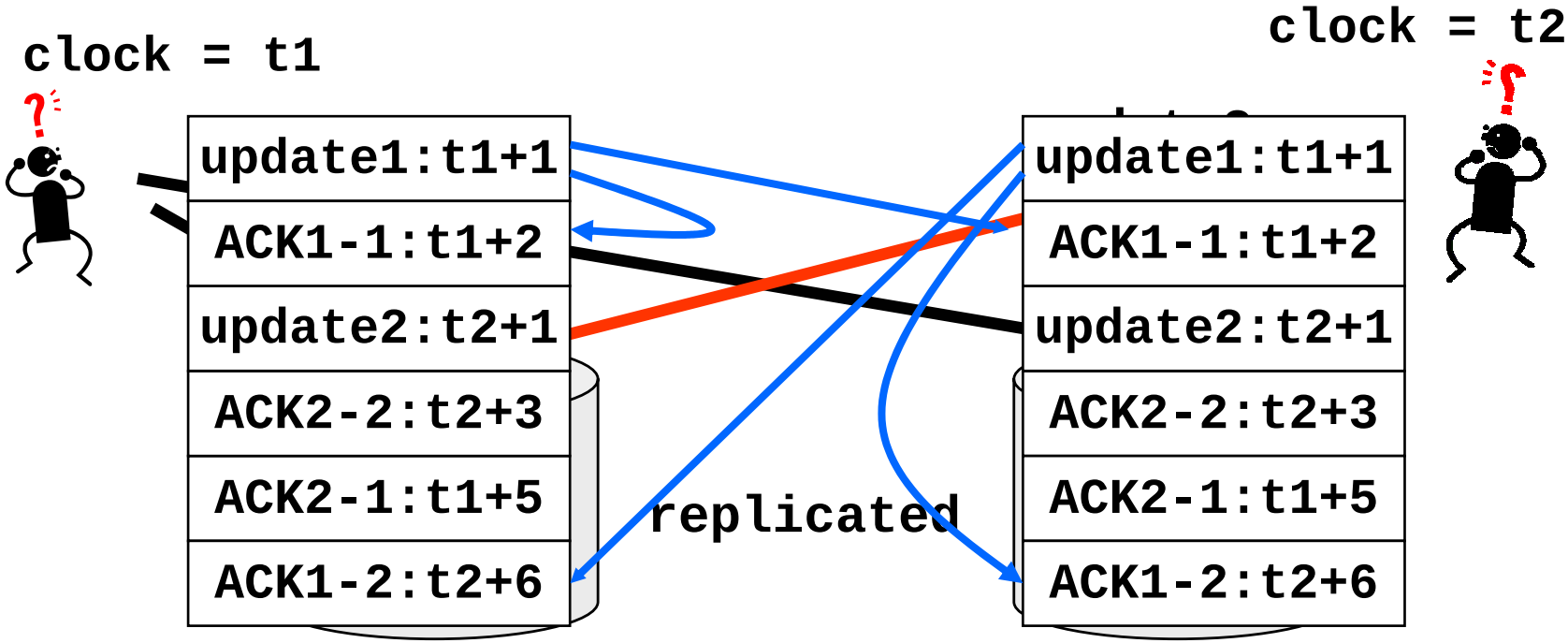
- Receiverは、いったんqueueにいれて、タイムスタンプでソート
  - この場合は、t1 < t2　とすると、下のようになる。



Totally Ordered(全順序)マルチキャスト

- 受信者は受信メッセージに対するACKをタイムスタンプ付きで他の全てのプロセスにマルチキャスト（このとき、Lamportのアルゴリズムにより、ACKのタイムスタンプは受信メッセージよりも大きいことを保証）

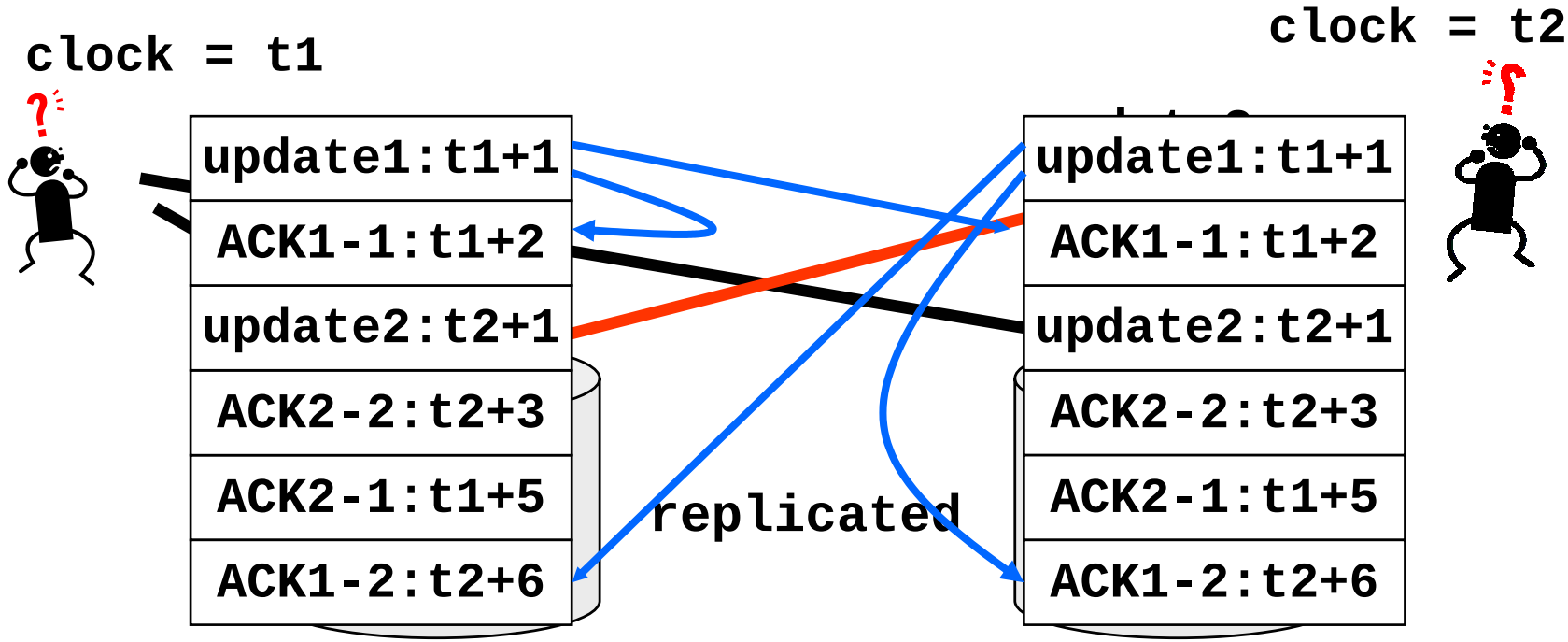
ACKx-y： 更新xに対するプロセスyからのACK



Totally Ordered(全順序)マルチキャスト

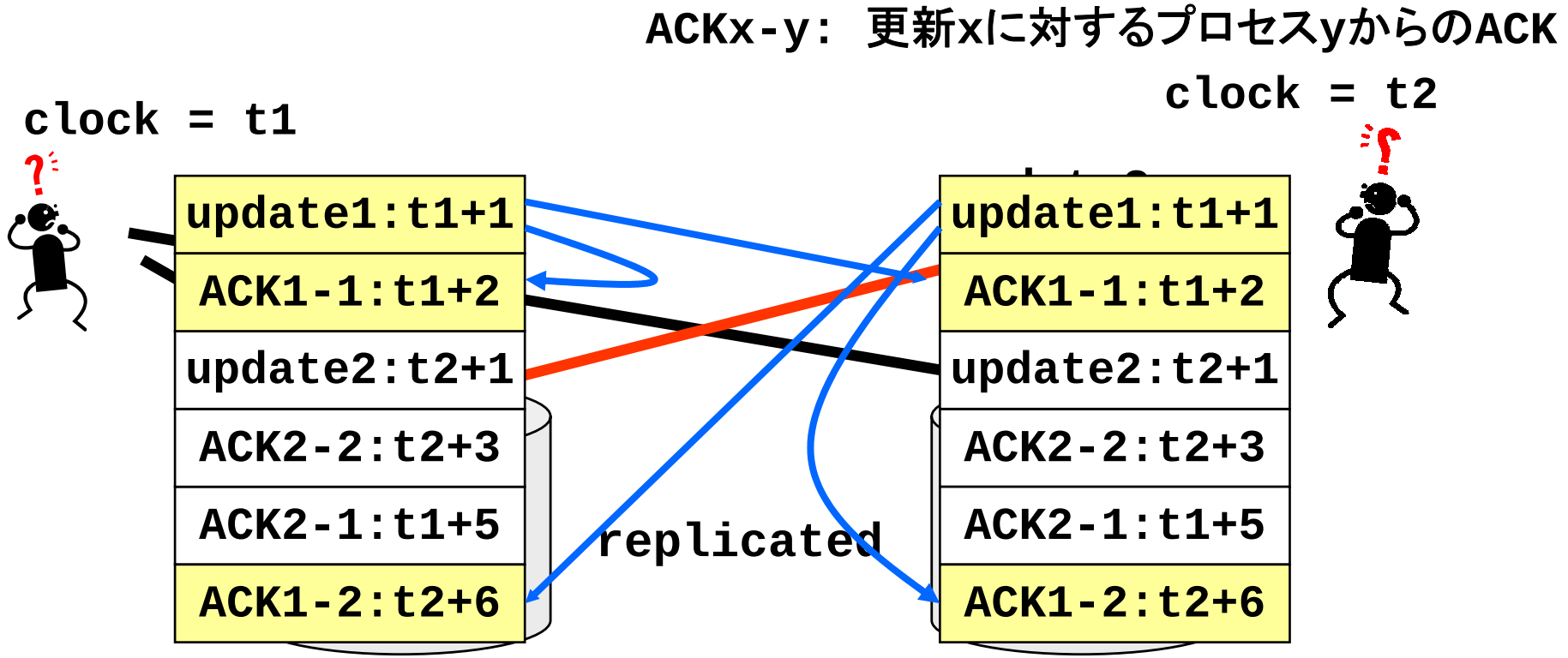
- Lamportのアルゴリズムによって、全てのメッセージが一貫性を持つグローバルなタイムスタンプによって順序付けられることを保証
- 全てのプロセスのキューの内容は（ACKも含めて）いつかは同じになる

ACKx-y: 更新xに対するプロセスyからのACK



Totally Ordered(全順序)マルチキャスト

- 各プロセスは、受信メッセージに対するACKを他の全てのプロセスから受信した段階で、そのメッセージをキューから取り除いてアプリケーションに渡す
  - キューに残ったACKは単に捨てられる



Vector Clock

- Clockを各プロセッサでの論理クロックのベクトルで表現する
  - 初期値はプロセッサによって異なる
  - メッセージのやりとりが無いときは、クロックは変化しない
  - iがjにメッセージを送信するときは、送信直前のクロックを添付して送り、送信後Ci[i]を増加させる
  - jがiからのメッセージを受信したら、メッセージのクロックと受信直前のクロックを比較し、大きい方より大きい値を次のクロックとする

- Vector Clockを用いると、二つのイベントの時間的関係（因果関係：causality) を決定することができる

Vector clockの利点

- V1とV2の対応する要素について、全ての要素についてV1の方がV2より大きいとき、V1 > V2であるとする
- 地点aとbについて、Vector Clockの値がVa及びVbであるとき、以下のような関係が成り立つ
  - Va < Vb　bはaより先に起こった
  - Va > Vb　aはbより先に起こった
  - Va = Vb　aとbは同じプロセッサで同時に起こった
  - Va || Vb Concurrent State、異なるプロセッサで比較不能な時刻に起こった
- Lamportの論理クロックでは、a→bならば、C(a) < C(b)だが、C(a) < C(b)だからといって、a → bとは限らない

http://funini.com/kei/logos/clock.shtmlより

例

- 初期値はプロセッサによって異なる
- メッセージのやりとりが無いときは、クロックは変化しない
- iがjにメッセージを送信するときは、送信直前のクロックを添付して送り、送信後Ci[i]を増加させる
- jがiからのメッセージを受信したら、メッセージのクロックと受信直前のクロックを比較し、大きい方より大きい値を次のクロックとする

◆ Vector clock