■6 群（コンピュータ - 基礎理論とハードウェア）- 7 編（ディペンダブルコンピューティング）

2 章 分散システムのディペンダビリティ

（執筆者：）

■概要■

【本章の構成】

■6 群 - 7 編 - 2 章

2-1 モデル

（執筆者：土屋達弘）[2008 年 6 月 受領]

2-1-1 分散システム

(1) 基本モデル

分散システムを，ネットワークを介して互いにメッセージを送受信することで通信するプロセスの集合として考える．つまり，メッセージパッシングによる通信を仮定する．

通信方法については，一つのプロセスから一つのプロセスへの通信，すなわち，一対一の通信のみを考える．また，メッセージ送信は，非ブロッキング送信のみを考える．これは，実行後，メッセージが受信されたか否かにかかわらず，すぐ次の処理に進むような送信である．反対に，送られたメッセージが送信先のプロセスに受信されるまで待たされる場合は，ブロッキング送信と呼ばれる．ブロッキング送信は，ACK（受信通知）の返信を行うことで，非ブロッキング送信を用いて実現できる．

送信されたメッセージは，0 より大きい時間経過した後，送信先のプロセスを実行しているプロセッサに到達する．メッセージの到達に必要なこの時間をメッセージ遅延と呼ぶ．プロセスがメッセージを受信できるのは，自分のプロセッサにメッセージが到達した時刻以降である．

あるプロセスの相対速度を，ほかのすべてのプロセスが少なくとも 1 命令を実行する間に，実行することのできる最大命令数と定義する．また，プロセスを実行する計算機それぞれが時計を有し，プロセスがその時計の値を参照できることを仮定することがある．この時計を局所時計（local clock）と呼ぶ．

(2) 属　性

文献 1) では，分散システムのモデルを特徴づける属性として，(a) ネットワークトポロジー，(b) メッセージバッファリング，(c) 故障，(d) 同期性という 4 項目をあげている．以下では，(a) ネットワークトポロジー，及び，(b) メッセージバッファリングについて説明する．(c) 故障と (d) 同期性に関しては，それぞれ本章 2-1-2 節と 2-1-3 節で議論する．

(a) ネットワークトポロジー

ネットワークトポロジーは，どのプロセスがどのプロセスに直接メッセージを送信できるかを規定する．ネットワークトポロジーは，プロセスを頂点とするグラフで表現できる．ここで頂点 p_i から p_j への辺は，プロセス p_i が p_j にメッセージを送信できることを表す．したがって，このグラフにおける辺を通信リンクと呼ぶ．プロセスは双方向に通信可能な場合が普通であるので，無向グラフを考えることが多い．

ただし，メッセージルーチングを行う通信ネットワークプロトコルを仮定することで，ネットワークトポロジーを陽に扱わないことが多い．その場合，ネットワークトポロジーは完全グラフとなる．

反対に，そのようなルーチング機構を仮定せず，ネットワークトポロジーを明示的に扱う場合の例としては，直接網を有す並列計算機や，無線ネットワーク，あるいは，オーバレイネットワーク上のシステムを想定した場合をあげることができる．

(b) メッセージバッファリング

各リンクにおける受信メッセージのバッファリングについては，バッファ長，及び，メッ

セージの到着順序に関して，いくつかの異なる種類の仮定が考えられる．

まず，バッファ長については，無限長の場合と，収納できるメッセージ数に制限がある場合を考えることができる．後者の場合，バッファがいっぱいになった際に，メッセージを送信した場合の動作に関する仮定が必要である．例えば，メッセージ消失やエラー通知の有無を仮定する．ただし，現実的には，バッファは十分大きいと考えることができる場合が多い．

メッセージの到着順序に関しては，送信順に到着を仮定する FIFO モデルと，そのような順序を仮定しないモデルがある．

2-1-2 故障モデル

分散システムにおける典型的なプロセスの故障モデルを以下にあげる．

- クラッシュ故障（crash fault）：故障が起こるまで正常に動作し，故障が起こった後は停止する．
- オミッション故障（omission fault）：メッセージの送受信に失敗する．つまり，実行した送信，受信命令の一部しか実際には成功しない．
- タイミング故障（timing fault）：時間に関する仮定に違反する．具体的には，命令の実行速度や時計の精度に関して，それらの仕様が守られない．
- ビザンチン故障（Byzantine fault）：故障プロセスの動作に仮定をおかない．つまり，プロセスは任意の動作を実行可能である．

通信リンクについても同様なモデルが設定できる．

- クラッシュ故障: 故障が起こるまで正常に動作し，故障が起こった後はメッセージの配送を停止する．
- オミッション故障: メッセージを消失する．
- タイミング故障: 時間に関する仮定に違反する．つまり，メッセージ遅延が，仕様よりも早く，あるいは，遅くなる．
- ビザンチン故障: 任意の動作を実行可能とする．したがって，偽のメッセージの配送やメッセージの改ざんなどが生じうる．

プロセスのクラッシュ故障の変種として，故障した後に回復し，正常な動作を再開する可能性を考慮したモデルもよく検討されている．これは，クラッシュ・回復モデル（crash-recovery model）と呼ばれる．通常のクラッシュ故障を仮定した場合と異なり，クラッシュ・回復モデルでは，クラッシュによって失われたメモリ上の情報や，停止中に受信できずに消失したメッセージについて考慮する必要が生ずる．したがって，このモデルで動作するアルゴリズムは，通信ネットワークのオミッション故障にも耐性をもつことが普通である．

2-1-3 同期性

分散システムは，プロセスの処理速度やメッセージ遅延の大きさによって，同期システムと非同期システムに大別される．これらの要素に関する仮定は，分散システムにおける主要な諸問題を解くアルゴリズムの存在を左右する条件となるので，極めて重要である．

(1) 同期システム

同期システムは，以下の条件を満たす分散システムである．

- プロセスの相対速度に上限が存在する．
- メッセージ遅延に上限が存在する．

局所時計を仮定する場合は，その精度に関し以下の条件を設ける．

- 実時刻 t においてプロセス p_i が参照した時刻を $C_i(t)$ で表記する．定数 $\rho(\geq 0)$ が存在して，局所時計による時刻と実時刻との間に以下の条件が成り立つ（ρ をドリフト率（drift rate）と呼ぶ）．

$$(1+\rho)^{-1} \leq \frac{C_i(t) - C_i(t')}{t - t'} \leq 1 + \rho \qquad (2\cdot1)$$

同期システムでは，すべてのプロセスが，あたかも完全に同期して動作する仕組みを実現できる．これは，同期ラウンドモデルと呼ばれる．同期ラウンドモデルでは，全プロセスは同期して，ラウンドと呼ばれる処理の単位を逐次的に実行する．各ラウンドは，メッセージの送信，そのラウンドで送信されたメッセージの受信，そして，受信したメッセージに基づく計算処理という，3 段階により構成される．

(2) 非同期システム

同期システムでないシステムは，非同期システムである．同期性の度合いに応じ，様々な非同期システムモデルを考えることができる．仮定する同期性の度合を強めると，その分システムの設計は容易になるが，モデルが適切に表現することのできる現実的な状況は限定されてしまう．逆に同期性に関する仮定を設けなければ，そのモデルを前提としたアルゴリズムは一般性が高く，頑強なものとなる．

しかしながら，故障の可能性を仮定した場合，完全な非同期システムでは，多くの重要な問題に対しアルゴリズムが存在しないことが知られている．これは，同期性が故障検出に本質的な役割を果たすためである．例えば，完全な非同期システムでは，故障しているプロセスと，実行速度の遅いプロセスを区別することができない．したがって，ある程度の同期性に関する仮定を導入した，適切なモデルを設定することが必要となる．以下に例をあげる．

(a) 部分的同期モデル (partially synchronous model) [2)]

以下の二つのモデルを考える[2)]．

- メッセージ遅延，プロセスの相対速度に上限が存在しているが，その値は分かっていない．
- システムが安定状態にある場合，メッセージ遅延，プロセスの相対速度の上限が存在

し，大きさも分かっている．しかし，安定状態は将来いつか成り立つとしかいえない．

また，これらを組み合わせることで，以下のモデルも設定できる[3]．

- システムが安定状態にある場合，メッセージ遅延，プロセスの相対速度に上限が存在するが，その値は分かっていない．また，安定状態は将来いつか成り立つとしかいえない．

また，安定状態が間欠的に成り立つという仮定を置いたモデルもある．時間非同期モデル（timed asynchronous model）はそのような例である[3]．

(b) 同期性の抽象化

同期性をプロセスの故障検出能力として抽象化することで，システムモデルから明示的な同期性に関する仮定を取り払ったモデルが，近年では広く受け入れられている．このモデル化のアプローチでは，プロセスの相対速度やメッセージ遅延に関する仮定を設けない代わりに，各プロセスは障害検出器（failure detector）をもち，ほかのプロセスの状態を予測することができるものとする[3]．障害検出器に基づくアルゴリズムは，個々のシステムモデルに依存したアルゴリズムと比べ，一般性が高く，設計・理解が容易であるという利点がある．障害検出器については以下で詳しく述べる．このほかに，同期性を表現する，障害検出器よりも更に抽象度の高い概念として，通信述語（communication predicate）が提案されている[7]．

(3) 障害検出器

(a) 障害検出器のクラス

障害検出器は，各プロセスに装備された機構で，故障と推測されるプロセスの集合を出力する．障害検出器の性質は，故障していないプロセスを故障と推測しないという正確性（accuracy）と，故障したプロセスはいずれ故障と推測されるという完全性（completeness）に大別して考えることができる．プロセスのクラッシュ故障を仮定したとき，これらの性質は以下のようにより詳細に分類できる．

- 強（strong）（弱（weak））正確性: すべての（少なくとも一つの）プロセスは故障する前に故障と推測されない．

- 将来的（eventually）強（弱）正確性: ある時刻以降，正常なすべての（少なくとも一つの）プロセスは故障と推測されない．

- 強（弱）完全性: ある時刻以降，正常なすべての（少なくとも一つの）プロセスは，故障したすべてのプロセスを故障と推測し続ける．

これらから表 2･1 に示すように，8 種類の障害検出器のクラスが定義できる．

また，元来の障害判定器の概念を拡張した，これらのクラスに属さない障害判定器も多く提案されている．それらのなかで最も重要なものが，Ω 障害判定器である[5]．Ω の出力は，故障と推測されるプロセスではなく，故障していないと推測される一つのプロセスである．Ω は次の性質を有す．

- ある時刻以降，故障していないすべてのプロセスの故障判定器において，故障していないあるプロセスが出力され続ける．

つまり，Ω はリーダプロセスの選択を行っているといえる．

表 2·1　障害検出器のクラス

完全性	正確性			
	強	弱	将来的強	将来的弱
強	$\mathcal{P}$	$\mathcal{S}$	$\Diamond\mathcal{P}$	$\Diamond\mathcal{S}$
弱	$\mathcal{Q}$	$\mathcal{W}$	$\Diamond\mathcal{Q}$	$\Diamond\mathcal{W}$

障害検出器を前提とすることで，障害検出器の実現方法は考慮せず，そのクラスの能力のみに注目してアルゴリズムを設計することが可能となる．

(b) 障害検出器の実現

8 種類の障害検出器のうち，最も非力な $\Diamond\mathcal{W}$, $\Diamond\mathcal{S}$ であっても，完全な非同期システムでは実現することができない．そのため，有用な障害検出器を実現するためには，ある程度の同期性が必要である．例えば，先に見た 3 種類の部分的同期モデルでは，定期的に生存メッセージを送信するという単純なアルゴリズムで，$\Diamond\mathcal{P}$ が実現できる[3)]．一方，障害判定器 $\mathcal{P}$, $\mathcal{Q}$, $\mathcal{S}$, $\mathcal{W}$ は，これらの部分的同期モデルにおいては実現できない[6)]．Ω は，部分同期モデルよりも更に弱い仮定を用いて実現できる[8)]．

■参考文献

1) L. Lamport and N. Lynch, “Distributed Computing: Models and Methods,” Handbook of Theoretical Computer Science, Elsevier, pp.1158-1199, 1990.
2) C. Dwork, N. Lynch, and L. Stockmeyer, “Consensus in the presence of partial synchrony,” J. ACM, vol.35, no.2, pp.288-323, 1988.
3) F. Cristian and C. Fetzer, “The timed asynchronous distributed system model,” IEEE Trans. Parallel and Distributed Systems, vol.10, no.6, pp.642-657, 1999.
4) T.D. Chandra and S. Toueg, “Unreliable failure detectors for reliable distributed systems,” J. ACM, vol.43, no.2, pp.225-267, 1996.
5) T.D. Chandra, V. Hadzilacos, and S. Toueg, “The weakest failure detector for solving consensus,” J. ACM, vol.43, no.4, pp.685-722, 1996.
6) M. Larrea, A. Fernandez, and S. Arevalo, “On the implementation of unreliable failure detectors in partially synchronous systems,” IEEE Trans. Computers, vol.53, no.7, pp.815-828, 2004.
7) M. Hutle and A. Schiper, “Communication predicates: A high-level abstraction for coping with transient and dynamic faults,” Proc. 37th Int’l Conf. on Dependable Systems and Networks, pp.92-101, 2007.
8) A. Mostefaoui, M. Raynal, C. Travers, “Time-free and timer-based assumptions can be combined to get eventual leadership,” IEEE Trans. Parallel and Distributed Systems, vol.17, no.7, pp.656-666, 2006.

■6 群 - 7 編 - 2 章

2-2 合意問題

（執筆者：泉　泰介，DÉFAGO Xavier，林原 尚浩，FETZER Christof）

［2009 年 4 月 受領］

分散システムにおいて，多くの問題は複数のプロセス間で原子的な動作を行うために合意を形成することを必要とする（例えば，リーダ選択問題（leader election），全順序メッセージ配送（total order broadcast），原子コミット（atomic commit）など）．

本章では，よく用いられる合意問題のなかでも特に，合意問題における静的な問題としてのコンセンサス（consensus problem）と動的な問題としての時間同期（clock synchronization）に焦点を当てる．まず，コンセンサスについては同期システム，非同期システムの両方の場合について述べる．

次に，時間同期の概念とそのアルゴリズムについて概説する．

2-2-1 合意問題と応用

（執筆者：DÉFAGO Xavier，林原 尚浩）［2009 年 4 月 受領］

複数のプロセス間で協調動作を行うためには何らかの合意が必要なため，合意問題は様々なアプリケーションに用いられている．ここでは，よく用いられるいくつかの合意問題とそのアプリケーションについて概説する．

(a) 能動的多重化

能動的多重化（state machine replication）の概念は，分散相互排除問題（distributed mutual exclusion）を解くメカニズムとして，Lamport [48)]によって提案された．このアイディアは，分散相互排除問題において各プロセスはリクエスト・キュー（request queue）の複製をもち，アルゴリズムはそのキューの内容の一貫性を保持する．あるプロセスがキューに入れたリクエストが，キューの先頭に到達したときに，そのプロセスはクリティカルセクションに入ることができる．

キューの一貫性保持は次にあげる状態を前提としている．(1) キューを表現する状態機械は決定的である，(2) すべてのキューの複製は同一の初期状態から始まる，(3) すべての状態遷移はすべてのキューの複製に同じ順序で起こる．そのため，すべての複製は状態変化が一致して起こる．

Lamport は，自身が提案した手法が相互排除問題よりも一般的であることに言及しており，この手法を改良したものを能動的多重化の概念として議論している [11)]．能動的多重化の様々な手法については，Schneider によって調査されている [10)]．

相互排除問題に以外にも能動的多重化は多くのアプリケーションに用いられている．特に，いくつかのプロセスが故障しても，それに対する耐故障性をもつ分散システムやそれによって提供されるサービスに用いられることが多い．能動的多重化は，複製された複数のプロセスがその状態の一貫性を保持することを保証し，システム内部で起きたプロセスの故障などを隠ぺいすることができる．

(b) コンセンサス問題

コンセンサス問題において，各プロセスは何らかの初期値をもち，すべての正常なプロセスは最終的に同一の値に合意しなければならない．この問題は，以下の性質を満たす．

 © 電子情報通信学会 2010

合意性 すべての故障していないプロセスは同じ値に合意しなければならない．

正当性 もし，ある故障していないプロセスが，ある値 v を決定するならば，その値 v は，いずれかのプロセスの初期値である．

停止性 すべての故障していないプロセスは最終的に値を決定する．

これらの性質は，プロセスがどのような値をとったとしても保持される．

通常のコンセンサスの定義は，プロセスのクラッシュ故障 * を前提としている．しかし，前述の性質はプロセスが任意の故障（例えば，ビザンチン故障）を起こす場合にも適用される．このような場合のコンセンサス問題を特にビザンチン・コンセンサスという．

コンセンサス問題は，より特化した様々な合意問題を形成するために用いられる抽象度の高い問題である．例えば，コンセンサス問題はグループメンバシップ，全順序ブロードキャスト（total order broadcast），能動的多重化，原子レジスタ（atomic register）などを実装するために用いられる．コンセンサス問題のバリエーションの一つはまた，受動的多重化の実装に用いられている [4)]．

(c) ビザンチン合意問題

ビザンチン合意問題はプロセスが任意の故障を起こす可能性のあるシステムのために定義されている．この問題の共通の定義において，ある一つのあらかじめ決められた**ソースプロセス**と呼ばれるプロセスが初期値をもち，ビザンチン合意プロトコルを開始する．この定義は，前述のコンセンサス問題と同じ合意性と停止性をもち，それに加えて以下の正当性をもつ．

正当性 もし，ソースプロセスが故障していないならば，すべての故障していないプロセスはソースプロセスの初期値に合意しなければならない．

本稿において，ビザンチン合意問題は，しばしばビザンチン・コンセンサス問題として記述される場合がある．この二つの問題の違いは，ビザンチン・コンセンサス問題においては，dedicated プロセスがなく，ビザンチン合意問題にはあるという点があげられるが，明確な定義があいまいであるため，混同される場合がある．

(d) ブロードキャストプロトコル

多くの異なるブロードキャスト（もしくは，マルチキャスト）の定義が存在する．これらの違いはどのような性質を保証するかという点にある．Hadzilacos と Toueg はこれらの定義やアルゴリズムの分類を行っている [19)]．本項では，その一部である二つのブロードキャストである，高信頼ブロードキャスト（reliable broadcast）と全順序ブロードキャスト（total order broadcast）† の定義を取り上げる．これらは，合意問題の実践的側面において代表的な定義であるといえる．

高信頼ブロードキャストは，二つのプリミティブ R-broadcast(m) と R-deliver(m) によって定義される．このブロードキャストは，送信プロセスがメッセージ m をブロードキャス

---

* クラッシュ故障とは，プロセスが停止もしくは終了し，その状態から回復することはないという定義の故障を指す．

† 全順序ブロードキャスト（total order broadcast）はまた，原子ブロードキャスト（atomic broadcast）とも呼ばれる．

トしている間に故障が起きた場合に，すべての正常なプロセスは m を受け取るか，もしくはどのプロセスも受け取らないという原子的な動作を行うことを保証する．つまり，形式的には以下の性質を満たす．

正当性 もし故障していないプロセスがメッセージ m を R-broadcastしたら，そのメッセージ は最終的に R-deliverされる．

一律合意 もしプロセスがメッセージ m を R-deliverするならば，すべての故障していないプロセスは最終的にそのメッセージを R-deliverする．

一律妥当性 任意のメッセージ m に関して, メッセージ m がその送信者によって R-broadcast された場合に，かつそのときに限り，すべてのプロセスはそのメッセージをたかだか 1 回 R-deliverする．

高信頼ブロードキャストは複数のプロセスに公平に情報を送信するために便利なメカニズムである．一律合意という性質によって定義されている原子的な動作は，耐故障システムにおいてプロセス間の命令伝達に非常に有効である（例えば，分散システムの資源管理など）．

全順序ブロードキャストは，TO-broadcast (m) と TO-deliver (m) によって定義される．このブロードキャストは，高信頼ブロードキャストの上位プロトコルにあたり，高信頼ブロードキャストにおいて定義された性質を継承している．これに加え，全順序ブロードキャストはメッセージを受信するすべてのプロセスが同じ順序でメッセージを受容することを保証する．この定義を以下に示す．

全順序 あるプロセス p_i がメッセージ m' をメッセージ m の後に受容するならば，ほかのプロセス p_j も m を受容した後にしか m' を受容しない．

全順序ブロードキャストは様々な方法で定義，記述されている．Défago らによってこれらの調査，分類，比較が行われた [18)]．また，Baldoni らはこれらの仕様について分類を行っている [23)]．全順序ブロードキャストの応用で最も主要なものは，前述の能動的多重化である．

(e) グループメンバシップ

グループメンバシップは, あるグループのメンバであるプロセスはすべて同じビュー（view）をもつようなグループメンバの管理を行う．これは, グループメンバの管理と前述のブロードキャストのプリミティブを組み合わせたビュー同期通信（view synchronous communication）と呼ばれるプリミティブによって実現されている．

グループメンバシップの仕様記述は，環境に強く依存するために，非常に困難であるといわれている．Chockler らは，多くのグループメンバシップの仕様とそれに関連する通信プリミティブについて調査と分類を行っている [5)]．

Schiper は，グループメンバシップの仕様や定義が困難である原因について,『ビューの一貫性を保持する際に，プロセスの明確なグループへの参加（join）／脱退（leave）イベントと故障によってグループから脱退するイベントの区別が付かないことである』と強く主張している [21)]．Schiper は，グループメンバシップにおいてメンバシップ管理と故障検出は互いに相容れないため，それぞれを分けた仕様や定義を推奨している．

グループメンバシップとビュー同期は，Quicksilver [24]，Spread [2]などの耐故障ミドルウェア・プラットフォームのための抽象モデルとして提案された．

2-2-2 コンセンサス問題（同期モデル）

（執筆者：泉　泰介）[2008 年 7 月 受領]

同期システムは，一般的には，プロセスの相対速度及びメッセージ遅延に上限を設けたモデルとして定義されるが，合意問題を考える場合は，通常，より簡略化された同期ラウンドモデルが利用される場合が多い．以下，本節では同期ラウンドモデル上での合意プロトコルについて説明する．なお，多くの同期ラウンドモデル上の合意プロトコルにおいては，以下の 3 条件が暗黙的に仮定されており，本節もそれに従うものとする．(1) すべてのプロセスは初期ラウンド（ラウンド 1）開始時にプロトコルを実行する．(2) 提示しうる値の集合は全順序集合である（すなわち，任意の 2 提示値の間で大小関係が定義されている）．(3) 故障が起こるプロセス数には上限が存在し，その値は既知である．

以降の議論において，システム内のプロセスの数を n，故障を起こすプロセス数の上限を t で表すものとする．また，故障はプロセス故障を指す．

（1）同期モデル上でのクラッシュ故障耐性を有する合意プロトコル

(a) Floodset プロトコル

同期ラウンドモデルにおける耐故障合意問題の研究は分散システム研究の初期より盛んに研究されており，これまでに数多くのプロトコルが提案されているが，本節では，最も基本的なプロトコルの一つである，Floodset プロトコル[27]を紹介する．

Floodset プロトコルの動作は以下のとおりである．

- 初期ラウンドにおいて，各プロセスは，自身の提示値をほかの全プロセスへと放送する．
- 2 ラウンド目以降は，自身がこれまでに受信したすべての提示値の集合を放送する．
- $t+1$ ラウンド目まで上記の通信を繰り返した後，最終的に，受信した提示値のなかから最小の値を選び，決定値として出力する．

クラッシュ故障が起こる同期ラウンドモデルにおいては，メッセージ送信時にプロセスが停止故障を起こす可能性がある．このとき，故障前に送信が完了しているメッセージは正しく受信プロセスに届けられるが，故障以降に送信する予定であったメッセージは配送されない．そのため，共通のメッセージをほかの全プロセスに配信しようとしても，送信者の停止故障により一部のプロセスのみがメッセージを受信する場合があり，このことが合意プロトコル実現の妨げとなる．Floodset アルゴリズムでは，決定値の合意を保証するために，$t+1$ ラウンドの実行のうち，故障が生じないラウンドが少なくとも一つ存在するという事実を利用している．このとき，故障が起きないラウンド以降において，全プロセスの収集した値の集合が等しくなることが保証される．

（2）早期決定アルゴリズム

前述の Floodset アルゴリズムでは，プロトコルの実行ラウンド数は故障の最大数 t にのみ依存しており，実行中において実際に起こる故障の個数にかかわりなく $t+1$ ラウンドを要する．一方，Floodset を改良することで，実際に実行中に起こる故障プロセスの数 f に

対して $f+2$ ラウンドで正しく決定を下すプロトコルを構成することが可能である[29]．このような，プロトコルの実行ラウンド数が実故障数 f によって決まるプロトコルは早期決定プロトコル（early-deciding protocol）と呼ばれる．一般に，実システムにおいては故障が起きる頻度はそれほど高くない（すなわち，$f \ll t$）ため，早期決定プロトコルは実用上は非常に高速に動作する．なお，Floodset に早期決定のプロトコルの機構を組み込むことで，最悪時ラウンド数 $\min\{f+2, t+1\}$ のプロトコルを構成することが可能であるが，このプロトコルはラウンド数の点で最適であることが示されている[26]．

（3）オミッション故障耐性を有する合意プロトコル

オミッション故障を考慮した場合，故障プロセスも含めたかたちで合意を達成することは不可能である（直感的には，ある故障プロセスが送受信するメッセージがすべて消失した場合，そのプロセスがほかのプロセスと共通の決定値を出力することは明らかに不可能である）．そのため，通常の合意問題の定義においては，オミッション故障プロセスに関して，その出力に関して一切の保証が要求されない．一方，より強い定義として，合意性のみ保証する（すなわち，故障プロセスについては決定値を必ずしも出力する必要はないが，出力された場合は合意が達成されている）かたちで合意問題を定義する場合もある．後者の定義に基づく問題は一様合意問題（Uniform Consensus）と呼ばれる．一様合意問題は通常の合意問題より真に困難な問題であることが知られており，オミッション故障のもとで，通常の合意問題が任意の最大故障数 t について可解であるのに対し，一様合意問題は $t < (n/2)$ のときのみ可解であることが示されている[29]．なお，一様合意問題について，ラウンド数に関して最適な（最悪時ラウンド数が $\min\{f+2, t+1\}$ の）アルゴリズムが存在することが知られている[28]．

2-2-3 コンセンサス問題（非同期モデル）

（執筆者：DÉFAGO Xavier, 林原 尚浩）[2009 年 4 月 受領]

非同期システムは，通信や処理の遅延時間の上限に制約がないシステムのモデルである．このモデルにおいて，プロセス群のなかの一つでも故障する可能性がある場合，コンセンサス問題の解決は不可能であることが **FLP 不可能性**として知られている [9]．この不可能性は，非同期システムにおいてプロセスがクラッシュ故障を起こしているか，とても遅いが正常に実行しているかを判別することが全くできないことが原因となっている．

(a) ランダム化コンセンサス

FLP 不可能性は, 非同期システムにおいてコンセンサス問題の **決定的**（deterministically）な解決策がないことを意味している．しかし，これは**確率的**な解法の存在を否定するものではなく，実際にいくつかの確率的耐故障アルゴリズムが提案されている．そのなかでも最も有名なものが，Rabin と Ben-Or によって提案されたランダム化コンセンサスアルゴリズムである [3, 15]．これらのアルゴリズムは，コンセンサス問題における合意性と正当性の性質を常に満たすことを保証しており，これに加え，停止性は確率的に満たすことを保証している．これは，FLP 不可能性を否定するものではない．なぜなら，コンセンサスアルゴリズムの永久に止まらない場合にも言及しているが，このような可能性は，アルゴリズムの実

行時間が増加するに伴い 0 に収束する *．FLP 不可能性の現実的な影響としては，複数のプロセスが合意に達するまでの上限時間を決めることが不可能であるということがあげられる．システムが非同期であるため，アルゴリズムは少なくともプロセスの過半数は故障していない，という仮定を置かなくてはならない．

(b) 部分同期モデル

ほかの手法として，部分同期システムというモデルが提案されている．Dwork らは，いくつかのクラスの部分同期システムを定義した [8)]．これらの一つとして，『通信遅延には上限が存在するが，その上限を知ることはできない』というシステムモデルも含まれている．定義された各クラスには，それぞれ決定性アルゴリズムが存在し，それらの様々な遅延の上限に関する興味深い考察が与えられている．

(c) 故障検出器

部分同期システムのような非同期システムの拡張とは異なる手法として，非同期システムに故障しているプロセスの情報を与えるオラクルとしての故障検出器 (failure detector) を導入したものがある．Chandra と Toueg は非信頼性故障検出器 (unreliable failure detector) という抽象モデルを用いてコンセンサス問題の解法を提案した．彼らは故障検出の精度に応じて，いくつかのクラスの故障検出器を定義した．そのなかでも最も重要なものが $\Diamond\mathcal{S}$ と呼ばれるクラスの故障検出器で，以下の様に定義されている．

完全性 故障しているプロセスは，いずれすべての故障していないプロセスに検出*される．

正確性 ある時刻以降，誤検出されない正常な（つまり，故障していない）プロセスが存在する．

Chandra と Toueg は，非同期システムにおいて，この $\Diamond\mathcal{S}$ を用いてコンセンサス問題を解決するアルゴリズムを提案した [7)]．このアルゴリズムは，正常なプロセスが半数以上存在することを前提としている．アルゴリズムの概要は以下のとおりである．

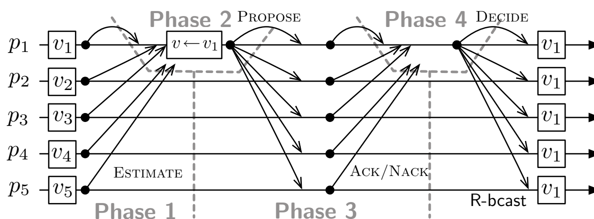


図 2･1 $\Diamond\mathcal{S}$ 故障検出器を用いたコンセンサスアルゴリズム; 単一ラウンドの実行.

---

* この問題のアナロジとして「二人のジャンケン」があげられる．二人でジャンケンを行う場合，理論的には「あいこ」の無限の系列は存在するが，その様な可能性はジャンケンの回数が増えるごとに 0 に近づき，いずれは勝敗が決する．

* 非同期システムにおいて，プロセスの故障を完全に判別することができないため，故障検出器がほかのプロセスを故障と判断（推測）する際，英文では suspect と表現される．本稿では，検出という表現を用いる．

このアルゴリズムは，非同期ラウンドの系列として実行が進む．各ラウンドはシーケンス番号によって一意に識別することができる．また，すべてのメッセージもそれらがどのラウンドのものかをラウンドの番号によって識別される．ラウンドの遷移は非同期に行われるため，論理的にはラウンドの系列によって実行が進んでいるにもかかわらず，実際は複数のプロセス間で見ると，いくつかのラウンドが同時に実行されている可能性がある．各ラウンドは Π_S のなかの一つのプロセスがラウンドの調停者として定義される．Π_S の構成は変更されることがなく，アルゴリズムの開始時点ですべてのプロセスがその情報を得ているものとする．したがって，ラウンド r の調停者は $c^r = ((r-1) \bmod n) + 1$ という式によって決定論的に指定される [†]．そのため，各ラウンドの調停者は Π_S のプロセスが持ち回りで務める．この調停者の決定方法は，循環調停者パラダイムとして知られている．

プロセスは何らかの初期値 v_0 をもち，*propose* イベントからコンセンサスの実行を始める．各プロセスはいくつかの値を保持している．そのなかで最も重要なものは，(1) 現在実行中のラウンド番号，(2) コンセンサスにおける決定値の推定値，(3) 推定値の論理的なタイムスタンプである．プロセスは次のように値を設定してコンセンサスアルゴリズムの最初のラウンドを開始する: (1) 1，(2) v_0，(3) 0.

各ラウンドは以下の四つのフェーズから構成されている．

- フェーズ 1 では，Π_S のすべてのプロセスがそれぞれの推定値をそのラウンドの調停者に送信する．この推定値は書き換えられたラウンド番号によるタイムスタンプが付与されている．

- フェーズ 2 では，調停者が Π_S の過半数のプロセスからの推定値を待っている．調停者は受信した推定値のなかから，最も最近のラウンドで書き換えられた推定値を選択し，その推定値を自身の推定値とする．調停者はまた決定値の案としてその推定値をブロードキャストする．

- フェーズ 3 では，プロセスは調停者からの推定値を待つ．調停者からブロードキャストされた推定値を受信したら，自身の推定値を調停者の推定値に更新し，タイムスタンプを現在のラウンド番号に更新する．その後，調停者に受信した推定値に対する肯定応答（ACK）を返信し，次のラウンドのフェーズ 1 へと移行する．

  あるプロセスが，調停者からの推定値を受信する前に調停者の故障を検出した場合，そのプロセスは，調停者に否定応答（NACK）を送信した後，ラウンド番号をインクリメントし，次のラウンドのフェーズ 1 へと移行する．

- フェーズ 4 では，調停者が過半数のプロセスからの応答（肯定応答か否定応答）を受信するまで待つ．受信したすべての応答が肯定応答の場合は，フェーズ 2 で送信した推定値を決定値として採用する．その後，調停者は決定値をすべてのプロセスへ高信

[†] 正確には，Chandra-Toueg アルゴリズムは $c^r = (r \bmod n) + 1$ という少しシンプルな方法で調停者を決定している．この方法では，直感に反して，プロセス p_2 がラウンド 1，プロセス p_3 がラウンド 2 の調停者をそれぞれ務める

頼ブロードキャストを行い停止する*.

一方，受信した応答のなかに一つでも否定応答があった場合，調停者は直ちにラウンド番号をインクリメントし，次のラウンドのフェーズ 1 へ移行する.

2-2-4 ビザンチン・コンセンサス

（執筆者：DÉFAGO Xavier, 林原 尚浩）[2009 年 4 月 受領]

(a) 最終的リーダシップと Paxos プロトコル

Lamport は Paxos と呼ばれる非同期システムのためのコンセンサスのプロトコルを提案した [12]．Paxos は今まで提案されてきた多くのコンセンサスプロトコルの変型である.

Paxos プロトコルはプロセスを次の三つの役割で識別する：提案者（proposer)，受容者 (acceptor)，学習者（learner)．実際は，プロセスはこの三つの役割のいくつかを兼任する.

提案者 提案者は提案値を支持し，受容者に提案値を受容するように働きかける．複数の提案値が競合した場合，提案者はこれらの提案値の調停を行い，競合を解決する.

受容者 受容者は全体としてプロトコルのメモリのような役割を担う．提案値が受容されるためには，その提案値がクォーラム（quorum）を形成した受容者のサブグループによって受容されなければならない.

学習者 提案値が受容された後，学習者は適切な動作を行う責務がある．複数の学習者をもつことによって，学習者の故障によるシステムの停止を回避することができる.

複数の提案者が競合した場合，提案者間で選ばれたリーダによって調停が行われる．このプロトコルは複数のリーダが存在しても動作するが，プロトコルの進行を保証するためには，唯一のリーダが存在する時間がなければならない．この概念は最終的リーダシップ（eventual leadership）と呼ばれる．この概念を故障検出器として表現される場合，Ω（omega）もしくは，Ω 故障検出器（omega failure detector）と表記される.

近年，Paxos コンセンサスプロトコルは主要なコンセンサス問題の解決方法となっており，多くの変種が存在し，それらのトレードオフも様々なものがある．Lamport によって提案されたオリジナルの Paxos は良性故障（クラッシュ故障とオミッション故障）に耐性をもつためにつくられた耐故障プロトコルであったが，後にビザンチン故障耐性をもつプロトコルの開発が行われた．ビザンチン故障のための Paxos プロトコルの先駆者として Castro と Liskov があげられる．彼らは，少なくとも 2/3 以上のプロセスが正常である場合にビザンチン故障耐性をもつ Paxos プロトコルを開発し，それを基にした能動的多重化プロトコルを提案した [6]．また，Li らは Paxos プロトコルのための統合フレームワークとして Paxos レジスタという概念を提案した [22].

* ほかのプロセスは，いずれのフェーズを実行中であっても，調停者から決定値を受信した場合は停止する.

2-2-5 時計同期

（執筆者：FETZER Christof, 林原 尚浩（訳））[2009 年 4 月 受領]

(1) プロセスとハードウェア時計

分散システムを構成する計算機はそれぞれが時計をもち，その時計の値は物理的な発振器の動作によって加算されていく．この時計をハードウェア時計，または，局所時計と呼ぶ．任意のプロセス p_i は，それを実行する計算機のハードウェア時計を参照することができる．しかしながら，このハードウェア時計の値は異なる計算機間で共有されていないため，時計の値にずれ（ドリフト）が生ずる．

一般的な計算機においてドリフト率 ρ には，たいてい上限 ρ_{max}([1ppm; 100ppm]) が存在する[44]．ハードウェア時計が正しいというのは以下の状態を満たすときである．

$$(t-s)(1-\rho_{max}) \leq C_i(t) - C_i(s) \leq (t-s)(1+\rho_{max}) \qquad (2\cdot2)$$

式 (2･2) において，[s, t] は任意の実時間における間隔である．ハードウェア時計の動作は時計同期アルゴリズム[34]によって定義される．

(2) ソフトウェア時計

ほかの計算機のハードウェア時計 $C_j(t)$ の値は，直接，$C_i(t)$ に適用されるわけではない．その代わりに，ソフトウェア時計が使用される[30, 37, 41, 57, 64]．プロセス p_i のソフトウェア時計 $S_i(t)$ は，ハードウェア時計 $C_i(t)$ からの以下のような変換によって得られる．

$$S_i(t) = C_i(t) + a_i(t) \qquad (2\cdot3)$$

$a_i(t)$ は離散ソフトウェア時計における離散時間関数[37, 57, 52, 65]，もしくは，連続ソフトウェア時計における連続時間関数[35, 61, 63]である．

(a) 離散ソフトウエア時間

時計同期アルゴリズム[34]には，時計の調整を行う同期ラウンドが存在する．離散ソフトウェア時間は，同期ラウンドで使用される時計調整値 A_i を適用する．

(b) 連続ソフトウェア時間

連続ソフトウェア時間は，その動作の連続性ゆえに時計同期アルゴリズムを構成しやすく，離散ソフトウェア時間より優先的に用いられる．連続ソフトウェア時間 S_i の速度 Φ_i は，時計調整値 A_i を用いて以下のように求められる[61]．

$$\Phi_i = \begin{cases} min\left(\frac{A_i}{P}, k\rho\right) & \text{if } A_i > 0 \\ max\left(\frac{A_i}{P}, -k\rho\right) & \text{otherwise} \end{cases} \qquad (2\cdot4)$$

P は同期ラウンドの長さである．k は以下の条件を満たす定数で，連続ソフトウェア時間における時計調整の上限を提供している．

$$k\rho \geq 2\rho + \rho^2 \qquad (2\cdot5)$$

連続ソフトウェア時間の速度 Φ_i は，式 (2･3) における $a_i(t)$ においても以下のように使用される．

$$a_i(t) = a_i + (C_i(t) - C_i) \cdot \Phi_i \tag{2•6}$$

(3) 局所時計予測

時計調整値 A_i を計算するために，プロセス p_i はほかの計算機の局所時計の値と誤差を得なければならない．これらは局所時計予測により得ることができる．局所時計予測には Time Transmission（TT）[30, 64] と Remote Clock Reading（RCR）[35, 40, 44] の二つのアプローチがある．

TT はメッセージ送信遅延の上限 δ_{max} か平均遅延と偏差をあらかじめ必要とするが，RCR はメッセージ送信遅延に関する情報を必要としない．

RCR を用いた Cristian ら手法[33] では，プロセス p_2 の局所時計の値 C_{p_2} を得るために p_1 はメッセージ m_1 を実時間 A に送信し，p_2 は実時間 B に受信する，p_2 は局所時計の値 C_{p_2} を含めたメッセージ m_2 を実時間 B に p_1 へ送信し，p_1 は実時間 D に受信する．各メッセージは送信時間のタイムスタンプが含まれている．m_1 と m_2 の送信遅延の上限は以下のように計算される．

$$up(m_2) = (D - A)(1 + \rho_{max}) - (C - B)(1 - \rho_{max}) - \delta_{mix} \tag{2•7}$$

δ_{mix} は p_1 と p_2 の送信遅延の下限である．

プロセス p_1 は計算された送信遅延の上限 $up(m_2)$ を実時間 $T(T \geq D)$ における C_{p_2} の近似値 $App_{p_2}(T, p_1)$ を得るために用いる．

$$App_{p_2}(T, p_1) = (T - D) + C + \frac{up(m_2)(1 + \rho) + \delta_{min}(1 - \rho)}{2} \tag{2•8}$$

プロセス p_1 は，また p_2 の局所時計の誤差の近似値 $\epsilon_{p_2}(T, p_1)$ を以下のように得る．

$$\epsilon_{p_2}(T, p_1) = 2\rho(T - D) + \frac{up(m_2)(1 + \rho) - \delta_{min}(1 - \rho)}{2} \tag{2•9}$$

(4) 時計同期の限界

ここでは，任意の 2 ノード間の最大偏差 Δ_{max} を用いて，時計同期における精度の限界について言及する．

Lundelius らは，$\Delta_{max} = (\delta_{max} - \delta_{min}) \left(1 - \frac{1}{|N|}\right)$ の条件を満たす時計同期アルゴリズムを提案し，任意の時計同期アルゴリズムが完全結合のネットワークにおいて，以下の条件を保証することに言及している[50]．

$$\Delta_{max} \geq (\delta_{max} - \delta_{min}) \left(1 - \frac{1}{|N|}\right) \tag{2•10}$$

Cristian らは，確率的な時計同期手法を提案した[35]．提案されたアルゴリズムでは，要求した精度での RCR を実行することができ，これによって任意の時計歪み（clock skew）を保証することができる．

$$\Delta_{max} \geq U - \delta_{min} + k\rho_{max}(1 + \rho_{max})W \tag{2•11}$$

Arvind によって提案された確率的時計同期手法において，最大偏差 Δ_{max} は以下のようになる．

$$\Delta_{max} = 2(\Delta_{sync} + (P + \delta_{max} - \delta_{min})\rho_{max}) \tag{2・12}$$

Δ_{sync} は要求された最大時計歪みである．

■参考文献

1) M.K. Aguilera and W. Chen and S. Toueg, “Failure Detection and Consensus in the Crash-Recovery Model,” Distributed Computing, vol.13, no.2, pp.99-125, 2000.
2) Y. Amir and C. Danilov and J. Stanton, “A Low Latency, Loss Tolerant Architecture and Protocol for Wide Area Group Communication,” In Proc. 30th IEEE/IFIP Intl. Conf. on Dependable Systems and Networks (DSN’00), pp.327-336, 2000.
3) M. Ben-Or, “Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols,” In Proc. 2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing (PODC’84), pp.27-30, 1983.
4) K.P. Birman and A. Schiper and P. Stephenson, “Lightweight Causal and Atomic Group Multicast,” ACM Trans. Comput. Sys., vol.9, no.3, pp.272-314, 1991.
5) G. Chockler and I. Keidar and R. Vitenberg, “Group Communication Specifications: A comprehensive study,” ACM Comput. Surv., vol.33, no.4, pp.427-469, 2001.
6) M. Castro and B. Liskov, “Practical Byzantine fault tolerance and Proactive Recovery,” ACM Trans. Comput. Sys., vol.20, no.4, pp.398-461, 2002.
7) T.D. Chandra and S. Toueg, “Unreliable Failure Detectors for Reliable Distributed Systems,” J. ACM, vol.43, no.2, pp.225-267, 1996.
8) C. Dwork and N.A. Lynch and L. Stockmeyer, “Consensus in The Presence of Partial Synchrony,” J. ACM, vol.35, no.2, pp.288-323, 1988.
9) M.J. Fischer and N.A. Lynch and M.S. Paterson, “Impossibility of Distributed Consensus with One Faulty Process,” J. ACM, vol.32, no.2, pp.374-382, 1985.
10) L. Lamport, “Time, Clocks, and the Ordering of Events in a Distributed System,” Comm. ACM, vol.21, no.7, pp.558-565, 1978.
11) L. Lamport, “The Implementation of Reliable Distributed Multiprocess Systems,” Computer Networks, vol.2, pp.95-114, 1978.
12) L. Lamport, “The Part-Time Parliament,” ACM Trans. Comput. Sys., vol.16, no.2, pp.133-169, 1998.
13) L. Lamport and R. Shostak and M. Pease, “The Byzantine Generals Problem,” toplas, vol.4, no.3, pp.382-401, 1982.
14) M. Pease and R. Shostak and L. Lamport, “Reaching Agreement in the Presence of Faults,” J. ACM, vol.27, no.2, pp.228-234, 1980.
15) M. Rabin, “Randomized Byzantine Generals,” In Proc. 24th Annual ACM Symp. on Foundations of Computer Science (FOCS), pp.403-409, 1983.
16) F.B. Schneider, “Implementing Fault-Tolerant Services Using the State Machine Approach: a Tutorial,” ACM Comput. Surv., vol.22, no.4, pp.299-319, 1990.
17) X. Défago and A. Schiper, “Semi-passive replication and Lazy Consensus,” J. Par. Distr. Comp., vol.64, no.12, vol.1380-1398, 2004.
18) X. Défago and A. Schiper and P. Urbán, “Total order broadcast and multicast algorithms: Taxonomy and survey,” acmcs, vol.36, no.4, pp.372-421, 2004.

19) V. Hadzilacos and S. Toueg, “A Modular Approach to Fault-Tolerant Broadcasts and Related Problems,” type-tr, no.94-1425, cornell, cornell-addr, 1994.
20) Jim Gray and Leslie Lamport, “Consensus on transaction commit,” ACM Transactions on Database Systems, vol.31, no.1, pp.133-160, 2006.
21) André Schiper, “Dynamic group communication,” Distributed Computing, vol.18, no.5, pp.359-374, 2006.
22) H Li and A Clement and A Aiyer and L Alvisi, “The Paxos Register,” In Proc. 26th IEEE Intl. Symp. Reliable Distributed Systems (SRDS’07), pp.114-126, 2007.
23) Roberto Baldoni and S Cimmino and C Marchetti, “A classification of total order specifications and its application to fixed sequencer-based implementations,” Journal of Parallel and Distributed Computing, vol.66, pp.108-127, 2006.
24) K. Ostrowski and K. Birman and D. Dolev, “Quicksilver Scalable Multicast (QSM),” In Proc. 7th IEEE Intl. Symp. Network Computing and Applications (NCA’08), pp.9-18, 2008.
25) 米田友洋，梶原誠司，土屋達弘, “ ディペンダブルシステムー高信頼システム実現のための耐故障・検証・テスト技術ー, ” 共立出版，2005.
26) I. Keider and S. Rajsbaum, “A simple proof of the uniform consensus synchronous lower bound,” Inf. Process. Lett., vol.83, pp.47-52, 2003.
27) N. Lynch, “Distributed Algorithms,”, pp.872, 1996.
28) P.R. Parvédy and M. Raynal, “Optimal Early Stopping Uniform Consensus in Synchronous Systems with Process Omission Failures,” Proc.SPAA, pp.302-310, 2004.
29) M. Raynal, “Consensus in Synchronous Systems: A Concise Guided Tour,” Proc.PRDC, pp.221-228, 2002.
30) K. Arvind, “Probabilistic clock synchronization in distributed systems,” In IEEE Transactions on Parallel and Distributed Systems, vol.5, no.5, pp.474-487, 1994.
31) A. Bavier and M. Bowman and B. Chun and D. Culler and S. Karlin and S. Muir and L. Peterson and T. Roscoe and T. Spalink and M. Wawrzoniak, “Operating system support for planetary-scale network services,” In Proc. of the First Symposium on Network Systems Design and Implementation (NSDI), pp.19-19, 2004.
32) S. Biaz and J.L. Welch, “Closed form bounds for clock synchronization under simple uncertainty assumptions,” Inf. Process. Lett., vol.80, no.3, pp.151-157, 2001.
33) F. Cristian and C. Fetzer, “Fault-tolerant internal clock synchronization,” In Proc. of the Thirteenth Symposium on Reliable Distributed Systems (SRDS’94), pp.22-31, 1994.
34) F. Cristian and C. Fetzer, “Fault-tolerant external clock synchronization,” In Proc. of the 15th International Conference on Distributed Computing Systems (ICDCS’95), pp.70, 1995.
35) F. Cristian, “Probabilistic clock synchronization,” Distributed Computing, vol.3, no.3, pp.146-158, 1989.
36) D. Dolev and J. Halpern and H.R. Strong, “On the possibility and impossibility of achieving clock synchronization,” In Proc. of the Sixteenth ACM Symposium on Theory of Computing (STTOC’84), pp.504-511, 1984.
37) C. Fetzer and F. Cristian, “An optimal internal clock synchronization algorithm, ” In Proc. of the 10th IEEE Conference on Computer Assurance (COMPASS’95), pp.187-196, 1995.
38) C. Fetzer and F. Cristian, “Integrating external and internal clock synchronization,”

Journal of Real-Time Systems, vol.12, no.1, pp.123-171, 1997.
39) C. Fetzer and F. Cristian, “Building fault-tolerant hardware clocks from COTS components,” In Proc. of the 7th IFIP International Working Conference on Dependable Computing for Critical Applications, pp.67-86, 1999.
40) C. Fetzer and F. Cristian, “A fail-aware datagram service,” In I. Bate and A. Burns, editors, IEE Proc. of Software Engineering, vol.146, pp.58-74, 1999.
41) R. Fan and N. A. Lynch, “Gradient clock synchronization,” Distributed Computing, vol.18, no.4, pp.255-266, 2006.
42) FlexRay Consortium, “FlexRay Communications System Protocol Specification,” version 2.1 revision a edition, 1995.
43) J.Y. Halpern and B. Simons and R. Strong and D. Dolev, “Fault-tolerant clock synchronization,” In Proc. of the 3rd ACM Symposium on Principles of Distributed Computing (PODC’84), pp.89-102, 1984.
44) Zbigniew Jerzak and Robert Fach and Christof Fetzer, “Fail-aware publish/subscribe,” In Sixth IEEE International Symposium on Network Computing and Applications (NCA 2007), pp.113-125, 2007.
45) H. Kopetz and A. Ademaj and A. Hanzlik, “Integration of internal and external clock synchronization by the combination of clock-state and clock-rate correction in fault-tolerant distributed systems,” RTSS, 00, pp.415-425, 2004.
46) H. Kopetz and G. Grunsteidl, “TTP - A Protocol for Fault-Tolerant Real-Time Systems,” Computer, vol.27, no.1, pp.14-23, 1994.
47) H. Kopetz and A. Kruger and D. Millinger and A. Schedl, “A synchronization strategy for a time-triggered multi-cluster real-time system,” In Proc. of the 14th IEEE Symposium on Reliable Distributed Systems (SRDS’95), pp.154-161, 1995.
48) L. Lamport, “Time, clock, and the ordering of events in a distributed system,” Communications of the ACM, vol.21, no.7, pp.558-565, 1978.
49) B. Liskov, “Practical uses of synchronized clocks in distributed systems,” Distributed Computing, vol.6, no.4, pp.211-219, 1993.
50) J. Lundelius and N.A. Lynch, “An upper and lower bound for clock synchronization,” Information and Control, vol.62, no.2/3, pp.190-204, 1984.
51) E.L. Lloyd, “Handbook of Wireless Networks and Mobile Computing,” chapter Broadcast Scheduling for TDMA in Wireless Multihop Networks, pp.3-12, 2002.
52) L. Lamport and P.M. Melliar-Smith, “Synchronizing clocks in the presence of faults,” J. ACM, vol.32, no.1, pp.52-78, 1985.
53) L. Lamport and R. Shostak and M. Pease, “The byzantine generals problem,” ACM Trans. on Programming Languages and Systems, vol.4, no.3, pp.382-401, 1982.
54) D.L. Mills, “Internet time synchronization: the network time protocol,” IEEE Trans. on Communications, vol.39, no.10, pp.1482-1493, 1991.
55) D.L. Mills, “Simple network time protocol (SNTP),” version 4 for IPv4, IPv6 and OSI, 1996.
56) D.L. Mills, “A brief history of ntp time: memories of an internet timekeeper,” SIGCOMM Comput. Rev., vol.33, no.2, pp.9-21, 2003.
57) H. Moser and U. Schmid, “Optimal clock synchronization revisited: Upper and lower bounds in real-time systems,” Principles of Distributed Systems, vol.4305, pp.94–109, 2006.
58) A. Olson and K.G. Shin, “Probabilistic clock synchronization in large distributed

systems," IEEE Trans. on Computers, vol.43, no.9, pp.1106-1112, 1994.

59) J. Postel, "User datagram protocol," 1985.

60) M. Pease and R. Shostak and L. Lamport, "Reaching agreement in the presence of faults," J. ACM, vol.27, no.2, pp.228-234, 1980.

61) F. Schmuck and F. Cristian, "Continuous clock amortization need not affect the precision of a clock synchronization algorithm," In Proc. of the 9th ACM Symposium on Principles of Distributed Computing (PODC'90), pp.133-143, 1990.

62) K.G. Shin and P. Ramanathan, "Transmission delays in hardware clock synchronization," IEEE Trans. on Computers, vol.37, no.11, pp.1465-1467, 1988.

63) T.K. Srikanth and S. Toueg, "Optimal clock synchronization," J. ACM, vol.34, no.3, pp.626-645, 1987.

64) J.L. Welch and N. Lynch, "A new fault-tolerant algorithm for clock synchronization," Information and Computing, vol.77, no.1, pp.1-36, 1988.

65) J. Widder and U. Schmid, "Booting clock synchronization in partially synchronous systems with hybrid process and link failures," Distributed Computing, vol.20, no.2, pp.115-140, 2007.

■6 群 - 7 編 - 2 章

2-3 原子動作

（執筆者：増澤利光）[2008 年 7 月 受領]

原子動作（atomic action）とは，それ以上分割できない最小単位の動作のことである．分散システムでは複数プロセスが互いに干渉しながら並行に動作するが，プロセスまたはプロセス群が実行する一連の操作列を原子動作として扱うことにより，操作列の一部のみが実行された過渡的な状態を考慮する必要がなくなる．つまり，原子動作として扱われる操作列は瞬時に実行されるとみなすことができる．このことは，実際にはより詳細な複数の操作で構成されるオペレーションでも，それを原子動作として実現することにより，プロセス間の相互干渉やシステム障害の影響はこのオペレーションの粒度で考えれば十分であることを意味する．このため，原子動作はディペンダブル分散システムの設計にとって重要な概念である．

2-3-1 一貫性制御

複数のプロセスに共有されるデータは，効率や故障耐性のために，複製を用いて実現されることが多い．同じデータの複製をいくつかの計算機に分散配置しておけば，近くの複製を利用することにより効率的なアクセスが可能となり，また，一部の計算機が故障しても共有データへのアクセスが可能となる．このとき，データへのアクセスは一つ以上の複製へのアクセスで実現されるが，同じデータに対する読出しや書込みを複数のプロセスが実行した場合，読み出される値が満たすべき制約のことをデータの一貫性（consistency）という．共有データが原子的であることを保証するための一貫性として，逐次一貫性（sequential consistency）と線形化可能性（linearizability）が用いられる．

複数のプロセスが同じデータにアクセスする場合を考える．逐次一貫性は，複数プロセスの共有データへのアクセスの結果（読み出された値）が，それらをある逐次的な順序で実行した場合の結果と一致し，更にその順序が各プロセス内での実行順序に矛盾しないことを保証するものである．線形化可能性は，逐次一貫性の制約を更に強めたものであり，逐次一貫性の制約に加えて，並行していない（一方の命令の実行終了後に他方の命令の実行が開始された）命令の組について，逐次一貫性を保証する逐次的な順序が，これらの並行していない命令の実行順序に矛盾しないことを保証する．

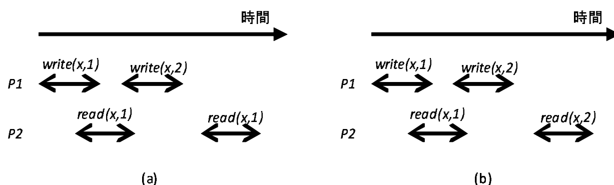


図 2･2 逐次一貫性と線形化可能性

図 2･2 に，プロセス $P1$, $P2$ が共有データ x に対して並行にアクセスしている例を示す．ここで，$write(x,a)$, $read(x,b)$ は，それぞれデータ x に対する値 a の書込み，値 b の読出しを表している．(a) の実行結果は $P1 : write(x,1)$，$P2 : read(x,1)$，$P2 : read(x,1)$，$P1 :$

$write(x, 2)$ の順の逐次実行に一致するので逐次一貫性を満たす．しかし，$P2$ の二つ目の $read(x, 1)$ は $P1 : write(x, 2)$ の終了後に開始しているにもかかわらず，逐次実行では $P1 : write(x, 2)$ に先行している．つまり，逐次実行は並行していない命令の実行順序に矛盾しているため，(a) の実行結果は線形化可能性を満たさない．一方，(b) の実行結果は $P1 : write(x, 1)$，$P2 : read(x, 1)$，$P1 : write(x, 2)$，$P2 : read(x, 2)$ の順の逐次実行に一致しており，逐次一貫性，線形化可能性をともに満たす．

逐次一貫性に関しては，読出しと書込みのいずれか一方だけは，一つの複製のみにアクセスする実現法が知られている．つまり，すべての計算機がデータの複製をもてば，読出しと書込みのいずれか一方だけは，ほかの計算機との通信を行うことなく，局所的なアクセスのみで実現可能である．例えば，読出しが書込みよりも多く実行される応用では，読出しを局所的に実行することにより，システム全体の処理効率を向上させることができる．しかし，読出しと書込みのうち，少なくとも一方に関しては，局所的アクセスのみでは実現不可能であることが知られており，計算機間の通信遅延の影響を避けることができない．一方，線形化可能性については，読出しと書込みのいずれについても通信遅延の影響を避けられないことが知られている．線形化可能性の重要な利点は，分散システムの各データが線形化可能性を満たせば，分散システム全体でも線形化可能性を保証できることである．つまり，分散システムのすべてのデータに対するアクセスの結果が，並行していない命令についてはその順序を保持しながら，ある逐次的な順序で実行した場合の結果と一致する．この性質は逐次一貫性に対しては成立しない．

逐次一貫性あるいは線形化可能性を保証する共有データの実現は，計算機間の通信遅延の影響を避けることはできない．より効率的な実現を可能にするために，より制約の弱い一貫性も考えられている．例えば，因果一貫性（causal consistency）は，書込みに関する逐次的な順序がプロセスによって異なることを許すことで逐次一貫性の制約を緩和しているが，原子動作として実現しているわけではない．

2-3-2 並行制御

原子動作としてもっとも古くから考察されてきたのは，データベースにおけるトランザクションである．例えば，銀行システムで口座 A から一定額を減じ，同額を口座 B に加えるという送金トランザクションは，口座 A，B それぞれに対する読出しや書込みというより詳細な命令の列で実現される．しかし，システム障害などにより実行が途中で停止した場合でも，口座 A だけが減額されていたり，口座 B だけが増額されているという状態が結果として生ずることは許されない．つまり，トランザクションは原子動作として実現する必要がある．一般的には，トランザクションのもつべき特性として，次の ACID 特性（ACID property）と呼ばれるものがよく知られている．

・原子性（atomicity）：トランザクションがそれ以上分割できない最小単位であること．つまり，トランザクションはすべてを完全に実行される（コミット）か，全く実行されないか（アボート）のいずれかである．

・一貫性（consistency）：トランザクションの実行によって，システムが矛盾した状態に陥ることがないこと．つまり，トランザクションの実行が，システムが満たすべき整合性を保存すること．

・独立性（isolation）：各トランザクションは，並行に実行されているほかのトランザクションの影響を受けないこと．
・耐性（durability）：コミットされたトランザクションの結果は，システム障害などが生じても失われることがないこと．

独立性はトランザクションの実行途中の過渡的な状態がほかのトランザクションに見えることはなく，システムの応答はコミットしたすべてのトランザクションをある順で逐次的に実行した場合に一致することを意味しており，直列化可能性（serializability）と呼ばれることもある．これは共有データの逐次一貫性に類似の制約であるが，トランザクションでは複数のデータにアクセスする場合もあり，逐次一貫性よりも大きな粒度を扱っている．以下では，並行動作するトランザクションが直列化可能性を満たすことを保証するための並行制御（concurrency control）の方法を示す．

（1）2 相ロッキング法

同じデータに対する異なるプロセスの書込みは相互排他的に実行する必要がある．また，一方のプロセスが読出しを行い，他方のプロセスが書込みを行う場合にも相互排他的に実行する必要がある．このように相互排他的に実行する必要がある命令は競合するという．ロッキングは，同じデータにアクセスする命令を相互排他的に実行するための方法である．つまり，プロセスがデータを使用する場合，そのデータが競合する命令でロックされていなければそのデータをロックしてアクセスする．既に競合する命令でロックされている場合は，ほかのプロセスが排他的なアクセス権を有していることとなり，そのロックが解放されるまでそのデータにアクセスできない．ただし，異なるプロセスの読出しを並行に実行することは問題なく，読出しの並行実行を可能とするためには，読出し命令によるロック（読出しロック）と書込み命令による（書込みロック）を区別する必要がある．

複数のデータにアクセスするトランザクションの実行が直列化可能性を保証するための手法として，2 相ロッキング（two-phase locking）がよく用いられる．2 相ロッキングは成長相とそれに続く縮退相から構成されるが，成長相ではロックの取得のみを行い，縮退相ではロックの解放のみを行う．つまり，2 相ロッキングではトランザクションの実行に必要なデータのロックをすべて取得した後でのみ，ロックの解放が行われる．2 相ロッキングの各トランザクションの実行結果は，それぞれのトランザクションを縮退相の開始時刻の順に逐次的に実行したときの結果と一致しており，直列化可能性を保証している．

2 相ロッキングではロックを順次解放することが許されており，あるデータのロック解放後にトランザクションがアボートし，そのデータに対する書込みが取り消されることがある．このとき，書込みが取り消されたデータを読み込んだトランザクションもアボートする必要があり，連鎖的なアボートを引き起こす可能性がある．厳密 2 相ロッキングでは連鎖的なアボートを避けるために，各トランザクションは終了時まですべてのロックを保持し，終了時にすべてのロックを同時に解放することを義務づけている．

2 相ロッキングでは複数のトランザクションがほかのトランザクションのロックの解放を互いに待ち続けるというデッドロック（deadlock）状態に陥る可能性がある．デッドロックを避けるためにデータのロッキングの順序を制約する方法や，デッドロックを検出するための様々な方法が考えられている．

（2）時刻印法

論理時計による時刻印（timestamp）を利用して，トランザクションの実行の直列化可能性を保証することもできる．この方法では各トランザクション T に時刻 $tt(T)$ を付し，基本的には，時刻印の小さい順にトランザクションを実行する．また，直列化可能性を保証するために，各データ A には A が読み出された直近の時刻 $tr(A)$ と A が書き込まれた直近の時刻 $tw(A)$ を付しておく．具体的には以下のように処理するが，この手法ではデッドロックが生じない代わりに，トランザクションのアボートが多くなる可能性がある．

(1) トランザクション T がデータ A を読み出すときには，$tt(T) > tw(A)$ ならば A を読み出すことができ，$tr(A)$ を $\max(tt(T), tr(A))$ に設定する．一方，$tt(T) < tw(A)$ のときは，トランザクション T をアボートする．T を実行するには，新たな時刻印で処理を開始する必要がある．

(2) トランザクション T がデータ A を書き込むときには，$tt(T) > \max(tw(A), tr(A))$ ならば A に書き込むことができ，$tw(A)$ を $tt(T)$ に設定する．一方，$tt(T) < tr(A)$ のときは，T をアボートする．$tr(A) < tt(T) < tw(A)$ のときは，T をアボートせず，T は A への書込みを行わずに以降の操作を続行する．これは T が A に書き込んだデータはどのトランザクションにも読み出されることなく，時刻印が $tw(A)$ のほかのトランザクションに上書きされるという逐次的実行に一致する．

（3）楽観的方法

楽観的方法（optimistic method）では，競合がまれにしか生じないという期待の下で処理を進める．トランザクションがデータにアクセスする際には，直列化可能性を保証するためのロッキングや時刻印チェックを一切行わず，操作を実行する．ただし，データへの書込みについては，共有データに直接書き込まずにプライベート作業領域に値を保持しておく．トランザクションの一連の操作が完了した時点で，競合が生じたかどうかを検査し，競合が生じていなければプライベート作業領域の値を共有データに書き込んでコミットする．一方，競合が生じている場合はアボートする．競合の有無は，例えば時刻印を用いて判定できる．

2-3-3 回復処理

トランザクション実行中に障害が生じた場合でも，トランザクションは原子性を満たさなければならない．つまり，障害にかかわらず，トランザクションは完全に実行を完了するか，その影響を全く残さないかのいずれかでなければならない．トランザクション実行中の障害に対応する方法として，プライベート作業領域を用いる方法と，ログを用いる方法がある．プライベート作業領域を用いる方法は，楽観的並行制御で述べた手法と同様に，書込み時には共有データに直接書き込まずプライベート作業領域に値を保持する．このプライベート作業領域の値は，トランザクションがコミットするときに共有データに書き込む．一方，トランザクションがアボートしたときには，単に，プライベート作業領域中のコピーは廃棄される．これにより，トランザクションがアボートしたときには，そのトランザクションの影響がシステムに残ることはない．

ログを用いる方法では，データへのアクセスは共有データに直接行われる．しかし，書込み時には，書き込む前のデータの値をログとして保存しておき，障害などでトランザクションがアボートしたときには，このログの記録を用いて，共有データの値を書込み前の値に戻

す（アンドゥと呼ばれる）．これにより，トランザクションがアボートしたときには，その影響をシステムに残さない．

2-3-4 原子コミット

分散システムにおけるトランザクションでは，一般に複数の計算機に分散した共有データにアクセスする．このとき原子性を実現するには，関連するすべての計算機でこのトランザクションをコミットするかアボートするかを合意しなければならない．すべての計算機で処理を完了できるならトランザクションはコミットされる．つまり，関連するすべての計算機でコミットする．一方，いずれか一つの計算機でも処理を完了できない場合はトランザクションをアボートすることとなり，すべての計算機でアボートする必要がある．すべての計算機でコミットかアボートかを合意することを原子コミット（atomic commit）と呼ぶ．

原子コミットは 2 相コミットプロトコル（two-phase commit protocol）によって実現できる．2 相コミットプロトコルでは，調停プロセスの存在を前提としており，投票相とそれに続く決定相の 2 相から以下のように構成される．また，システム障害によるプロセス停止に対応するためにタイムアウトを用いている．

(1) 投票相：調停プロセスがトランザクションに関連する全計算機にコミットの可否を問い合わせるために，メッセージ `request` を送信する．`request` を受信した計算機は，その計算機で実行されたすべての操作がコミット可能なら `yes` を，そうでなければ `no` を調停プロセスに返信する．`no` を送信するプロセスはこの時点でアボートする．なお，`yes` を送信する場合は，送信後の故障に対応するために処理をログに記録しておき，故障復旧後にログを用いてコミットできるようにしておく必要がある．

各計算機において，`request` の待受け中にタイムアウトが生じた場合，調停プロセスの故障と判断し，`no` を調停プロセスに送信してアボートする．

(2) 決定相：調停プロセスはすべての計算機から `yes` または `no` を受信するまで待つ．全計算機から `yes` を受信したときだけコミットに決定し，一つでも `no` を受信したときはアボートに決定する．そして，その決定をメッセージ `commit` か `abort` を全計算機に送信して通知する．各計算機では，受信した `commit` か `abort` に従ってコミットかアボートする．

調停プロセスが `yes` か `no` の待受け中にタイムアウトが生じた場合，アボートに決定し `abort` を全計算機に送信する．一方，各計算機で `commit` か `abort` を待受け中にタイムアウトが生じた場合，処理は少し複雑になる．この場合，もし調停プロセスが決定後に故障しているなら，その決定に従う必要がある．そこで，まず `commit` か `abort` を受信した計算機の有無を問い合わせ，`commit` か `abort` を受信した計算機が存在すればそれに従う．一方，`commit` も `abort` も受信した計算機が存在しなければ，調停プロセスが決定前に故障したか決定後に故障したかが不明となり，調停プロセスの復旧を待つ必要がある．

上述のように，決定相で `commit` か `abort` を待受け中にタイムアウトが生じた場合，各計算機は調停プロセスの復旧を待つことになる場合がある．このため，2 相コミットプロトコルはブロッキングプロトコルと呼ばれる．

ブロッキングを生じさせない原子アトミックプロトコルとして，より強力なブロードキャストプロトコルを使用する 2 相コミットプロトコルや，コミットをする前にコミット準備状態を導入した 3 相コミットプロトコルなどが考えられている．

■6 群 - 7 編 - 2 章

2-4 プロセス／データ／通信多重化

（執筆者：土屋達弘）[2008 年 6 月 受領]

2-4-1 プロセス多重化

多重化（レプリケーション（replication））は，最も典型的な耐故障性の実現方法である．プロセス多重化は，複数のプロセスを協調させることで，プロセスが故障した場合でも残りの正常なプロセスによって処理を継続することが可能な，耐故障サーバを実現する．多重化されたプロセスそれぞれはレプリカと呼ばれる．

状態をもたず応答が命令にのみに依存するようなサーバは，ステートレス（stateless）であるという．このようなサーバについては，レプリカ間の状態の整合性を気にする必要がないので，多重化の適用は容易である．一方，状態を有しているサーバを，ステートフル（stateful）サーバと呼ぶ．プロセス多重化でステートフルサーバを実現することは，レプリカ間で状態を常に同一に保つことが必要となるため，必ずしも容易ではない．以下では，ステートフルサーバの多重化について説明する．

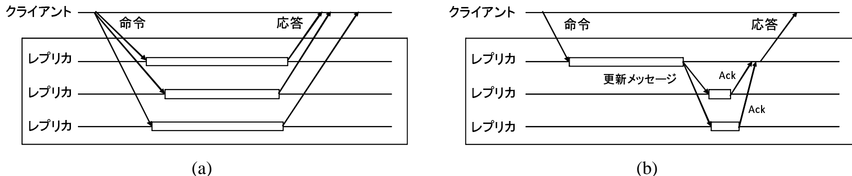


図 2·3　プロセス多重化．(a) **能動的多重化，**(b) **受動的多重化**

(1) 能動的多重化

能動的多重化（active replication，または，**状態機械アプローチ**（state machine approach））[10)]では，クライアントからの命令はすべてのレプリカに送られる（図 2·3(a)）．レプリカはそれぞれ同じ命令を実行して，応答をクライアントに返送する．すべてのレプリカは全く同じ処理を行い，状態も完全に一致させることで，同じ応答の出力が可能となるとともに，プロセスの故障が発生しても，クライアントに故障を意識させることなしに，正常なプロセスで処理を継続できる．

レプリカの状態を完全に一致させるためには，まず，プロセスの動作が決定的（deterministic）であることが必要である．これは，プロセスの状態が初期状態，及び，実行した命令列から一意に決定されることを意味する．非決定的動作の要因としては，マルチスレッド化や，シリアル番号のような個々の計算機に依存する情報の使用などがあげられる．

更に，クライアントからの命令の実行については，全順序性と原子性という 2 条件が満たされなければならない．全順序性とは，すべてのレプリカが同じ順序で同じ命令を実行するという条件である．原子性とは，命令が実行されるのであれば，すべての正常なレプリカでその命令が実行されなければならないという条件である．したがって，受信した命令をすぐに実行するのではなく，これらの 2 条件が満たされるようにレプリカプロセスの間で協調作業を行った後で，命令の実行を行うことが必要となる．

原子ブロードキャスト（atomic broadcast, total order broadcast）は，この協調作業

を行い，複数のプロセスに対して同順序でメッセージを配送する通信機能である．この機能は，プロセスを実行しているノードが受信したメッセージをいったんバッファリングし，協調作業を終えてからプロセスに配送することで実現される．様々な原子ブロードキャストの方法が提案されている．よく知られた方法の一つは，コンセンサスアルゴリズムを用いるものである[3)]．この方法では，受信したメッセージの集合を提案値としてコンセンサス問題を連続して解くことで，その都度プロセスに配送するメッセージとその順序について，ノード間で合意を達成する．

(2) 受動的多重化

次に**受動的多重化**（passive replication，または，プライマリバックアッププローチ（primary/backup approach））[2, 7)]について説明する（図 2･3(b)）．この手法では，要求された命令に対する実際の処理を行うプロセスは，一つのプロセスのみである．このプロセスを主プロセス（primary process）と呼び，それ以外のプロセスをバックアッププロセスと呼ぶ．主プロセスは要求された命令を実行後，その計算結果をバックアッププロセスに送信する．このメッセージを更新メッセージと呼ぶことにする．バックアッププロセスは更新メッセージの情報を用いて，主プロセスと同じ状態に状態を更新し，主プロセスに ACK を返信する．主プロセスは ACK 受信後，応答をクライアントに返送する．

受動的多重化では，主プロセスのみが命令を実行し，クライアントへの応答とサーバの新しい状態を決定するので，プロセスの動作が非決定的であっても，プロセス間で状態の不一致は起こらない．この特徴は，能動的多重化と比較した場合，実装上の制約が大きく緩和されるので，重要な長所といえる．また，バックアッププロセスの負荷が小さい点でも優れている．ただし，クライアントにとって，バックアッププロセスの故障は透過的であるが，主プロセスの故障はそうではない．クライアントの命令を実行中に主プロセスが故障した場合は，クライアントは新しい主プロセスの情報を得て，再度，同じ命令の実行要求を行う必要が生ずる．

また，サーバ側におけるプロセスの故障時の対応は容易ではない．特に，主プロセスが故障した場合，レプリカ間で状態の整合性を維持するためには，一部のバックアッププロセスのみに更新メッセージが配送されることがないことを保証しなければならない．

更新メッセージの送信に適した通信機能が，**仮想同期**（virtual synchrony）[1)]である．仮想同期では，ビューと呼ばれるシステムを構成するプロセス群が，故障や故障の推測に応じて時間とともに動的に変化していく．そして，ビュー全体に対しメッセージを送信した場合，そのメッセージがビューに含まれるプロセスすべてに配送されないのであれば，どのプロセスにも届かないという，原子性が保証される．ビューを多重化サーバを構成するレプリカの集合とすることで，仮想同期を用いて更新メッセージを全バックアッププロセスに配送することが可能となる．

また，能動的多重化を折衷した方式として，**準受動的多重化**（semi-passive replication）[4)]がある．この方式では，更新メッセージを直接バックアッププロセスに送信するのではなく，コンセンサス問題の提案値とすることで，コンセンサスを解くことによってレプリカに伝搬する．クライアントからの命令は能動的多重化のようにレプリカすべてに送られ，実行される．主プロセスとバックアッププロセスは明示的に区別されず，決定された値を提案したプロセスが実質的に主プロセスの役割を果たす．

2-4-2 データ多重化

データ多重化において満たすべき重要な性質は，物理的なコピー（レプリカ）が複数存在していることにかかわらず，各データ項目が論理的には一つの実体として，多重化を行っていない場合と同様に扱うことができることである．このような性質は，**単一コピー等価性**（one-copy equivalence）と呼ばれる．

（1）データベースの多重化

ここでは複数のノードにデータベースが多重化されており，トランザクション単位での制御によって，データ多重化を実現する方法について述べる．データベースの多重化手法は，イーガー（eager）手法とレイジー（lazy）手法に大別される[6)]．イーガー手法では，すべてのノードで同期してトランザクションの実行結果を反映させる．そのため，どのレプリカにおいてもデータが同一に保たれ，単一コピー等価性が達成できる．逆にレイジー手法では，一つのノードが単独でトランザクションをコミットすることが許される．更新内容はその後でほかのノードに伝搬され反映されるが，一時的なレプリカ間でのデータの不一致が生ずる可能性があり，単一コピー等価性は保証されない．性能の面では明らかにレイジー手法が優れているが，データの整合性を保証しないので，以降は，イーガー手法に限定して説明する．

データベースの動作が決定的である場合，プロセスの能動的多重化と同様，原子ブロードキャストでトランザクションを各ノードに配送し，ノードは配送された順序でトランザクションを実行することで，多重化が実現できる．

データベースの動作が非決定的な場合でも，各トランザクションは一つのノードで実行し，コミットする前に更新内容を原子ブロードキャストで配送することで，能動的多重化が実現できる[9)]．更新内容を受け取ったとき，そのトランザクションが競合を引き起こすかどうかを判定し，競合が起こるのであればトランザクションをアボートし，そうでなければコミットする．どのレプリカでも，同じ順序で同じトランザクションのコミット，アボートを決定するので，データベースの状態の整合性が保たれる．

また，プロセスの受動的多重化に相当する手法は，**主コピー**（primary copy）[6)]と呼ばれる．この手法では，すべてのトランザクションは，主コピーと呼ばれる特定のノードによって実行される．コミットする前に，更新内容が主コピーからほかのノードにブロードキャストされ，受信したノードはその内容を反映させる．最後に，全ノードで原子コミットを行い，そのトランザクションをコミットするかアボートするかを決定する．

（2）データ項目の多重化

データ項目を多重化し，読出し，書込み命令レベルでデータの整合性を保つ方法について紹介する．これにより，トランザクションに限らず，個別のファイルへのアクセスといったより基本的なレベルで，耐故障性を実現することができる．

単一読出し全体書込み（Read One Write All（ROWA））は，書込みはすべてのデータレプリカに行い，読出しは任意のノードから行う手法である．すべてのレプリカは完全に同じ状態に保たれる．

多数決投票（majority voting）は，単一読出し全体書込みとは異なり，すべてのレプリカにアクセスすることなく，書込み操作も実現することが可能である．この手法では，各データ項目には，整数値のバージョン番号が与えられる．また，各ノードには 0 以上の整数値が票として割り当てられる．

読出し操作を行う場合，バージョン番号と票を各ノードから受け取り，あらかじめ決められているしきい値 r 以上の票を取得できたならば，最も大きいバージョン番号を返したノードから読出しを行う．書込み操作の場合は，書込み操作に対するしきい値 w 以上の票を取得できた場合，要求に応じたノードのレプリカすべてに書込みを実行するとともに，それらのバージョン番号を更新する．この新しいバージョン番号は，返送されたバージョン番号の最大値に 1 を加えたものとする．

データの整合性を保つため，しきい値 r, w は，全ノードの票の合計を N とした場合，（条件 A1）$2w > N$，及び，（条件 A2）$w + r > N$ を同時に満たさなければならない．条件 A1 により，異なる書込み操作が同時に実行されることが防止される．また両条件により，読出し操作，書込み操作に関与するレプリカのなかに，必ず最新の値が存在することが保証される．

重み付き投票では，読出し，書込み操作でアクセスする必要のあるノードの集合を，票の合計数としきい値によって間接的に指定している．**クォーラム**（quorum）は，これらのノード集合を明示的に指す用語であり，多数決投票を更に一般化することができる．

読出し，書込みでアクセスが必要なノード集合をそれぞれ読出しクォーラム（read quorum），書込みクォーラム（write quorum）と呼ぶ．読出しクォーラムを $R_1, R_2, \cdots$，書込みクォーラムを $W_1, W_2, \cdots$ とすると，（条件 B1）任意の W_i, W_j について $W_i \cap W_j \neq \emptyset$，及び，（条件 B2）任意の R_i, W_j について $R_i \cap W_j \neq \emptyset$ という 2 条件が満たされなければならない．これら 2 条件は，重み付き投票における 2 条件に対応する．これらの条件を満たすクォーラムの構成に関して，様々な構成法が提案されている．しかし，現実的なトランザクションやレプリカ数を考えた場合，単一読出し全体書込みが性能面，及び，実装・管理の容易性から優れていることが多い[8)]．

2-4-3 通信多重化

分散システムの基盤となる通信ネットワークでは，多重化に基づく様々な障害対策が取られている．例えば，通信経路の多重化は，典型的な耐故障手法である．プロセスがルータの役割を果たすことで実現される**オーバレイネットワーク**（overlay network）でも，同様の手法を用いることが可能である．オーバレイネットワークは仮想的なネットワークであり，典型的には，ピアツーピアアプリケーションで用いられる．例えば，Tapestry オーバレイネットワーク[12)] では，バックアップ通信リンクによる通信経路の多重化や，ノードに対する識別子（オーバレイネットワーク上のアドレス）の多重割当てといった様々な多重化により，耐故障性を実現している．

明示的な通信路を用いないメッセージ多重化手法として，**エピデミック**（epidemic，あるいは**ゴシップ**（gossip））[5)] がある．これは，ブロードキャスト手法であるが，すべてのノードにメッセージが届くという保証は与えない．プロセスは，ブロードキャストするメッセージを受信した場合，いくつかのプロセスをランダムに選択し，それらにそのメッセージのコピーを転送する．各プロセスが転送するメッセージコピーの数が多ければ，それだけ冗長にメッセージがシステム中に拡散することになるため，故障プロセスの割合が大きくても，非常に高い確率でメッセージを届けることが可能になる．

また，TCP や HTTP のコネクションを，プロセス多重化の技術によって二重化してお

き，実際に接続されているサーバ側のプロセスに故障が起こった場合でも，ユーザに意識されることなく，コネクションを別のレプリカプロセスに付け替える方法も提案されている[11)]．受動的多重化では主プロセスの故障がクライアントに対し透過的でないと述べたが，このような通信の多重化技術を用いれば，主プロセスの故障についても透過性を実現することができる．

■参考文献

1) K.P. Birman and T.A. Joseph, “Exploiting virtual synchrony in distributed systems,” SIGOPS Symposium on Operating Systems Principles (SOSP), pp.123-138, 1987.
2) N. Budhiraja, K. Marzullo, F.B. Schneider, and S. Toueg, “The primary-backup approach,” In S. Mullender, editor, “Distributed Systems,” pp.199-216, ACM Press/Addison-Wesley Publishing Co., 1993.
3) T. D. Chandra and S. Toueg, “Unreliable failure detectors for reliable distributed systems,” J. Assoc. Comput. Mach, vol.43, no.2, pp.225-267, 1996.
4) X. Défago and A. Schiper, “Semi-passive replication and lazy consensus,” J. Parallel and Distributed Systems, vol.64, no.12, pp.1380-1398, 2004.
5) P.T. Eugster, R. Guerraoui, A.-M. Kermarrec, and L. Massoulié, “Epidemic information dissemination in distributed systems,” IEEE Computer, vol.37, no.5, pp.60-67, 2004.
6) J. Gray, P. Helland, P. O’Neil, and D. Shasha, “The dangers of replication and a solution,” In Proc. International Conference on Management of Data (SIGMOD’96), pp.173-182, 1996.
7) R. Guerraoui and A. Schiper, “Software-based replication for fault tolerance,” IEEE Computer, vol.30, no.4, pp.68-74, 1997.
8) R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and B. Kemme, “Are quorums an alternative for data replication?,” ACM Trans. Database Systems, vol.28, no.3, pp.257-294, 2003.
9) F. Pedone, R. Guerraoui, and A. Schiper, “The database state machine approach,” Distributed and Parallel Databases, vol.14, no.1, pp.71-98, 2003.
10) F.B. Schneider, “Implementing fault-tolerant services using the state machine approach: A tutorial,” ACM Computing Surveys, vol.22, no.4, pp.299-319, 1990.
11) D. Zagorodnov, K. Marzullo, L. Alvisi, and T.C. Bressoud, “Engineering fault-tolerant TCP/IP servers using FT-TCP,” In Proc. Int’l Conf. on Dependable Systems and Network (DSN 2003), pp.22-27, 2003.
12) B.Y. Zhao, L. Huang, J. Stribling, S.C. Rhea, A.D. Joseph, and J. Kubiatowicz, “Tapestry: A resilient global-scale overlay for service deployment,” IEEE J. Selected Areas in Communications, vol.22, no.1, pp.41-53, 2004.

■6 群 - 7 編 - 2 章

2-5 リカバリプロトコル

（執筆者：福本　聡）[2009 年 4 月 受領]

分散システムの誤り回復機構を実現するプロトコルあるいは分散アルゴリズムを総称して，ここでは**リカバリプロトコル**（recovery protocol）と呼ぶ．本節ではその概要について述べる．

2-5-1 誤り回復の原理と前提

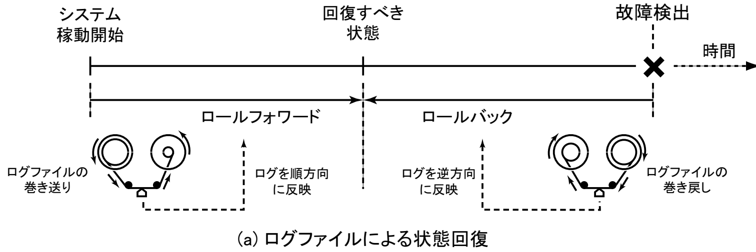
分散システムに限らず，コンピュータシステムの誤り回復の基本原理は，**前方誤り回復**（forward error recovery）と**後方誤り回復**（backward error recovery）に大別できる[1)]．

前方誤り回復による手法は，誤りが検出されたシステム状態に何らかの補正を加えることで，あるいは誤りを何らかの冗長性によってマスクすることで，正しいシステム状態をつくりだす．この手法では，誤りが発生する以前に達成された処理結果を破棄することなく，処理を継続することが可能である．誤り補正の代表的な実現方法としては，誤り訂正符号などの情報冗長の採用がある．また，誤りマスクのための具体的な冗長性としては，ハードウェアやソフトウェアの論理を三重化して多数決をとることで単一の誤りをマスクする TMR（triple modular redundancy）などがあげられる．分散システムの多数決による合意については，本章 2-2 節を参照されたい．

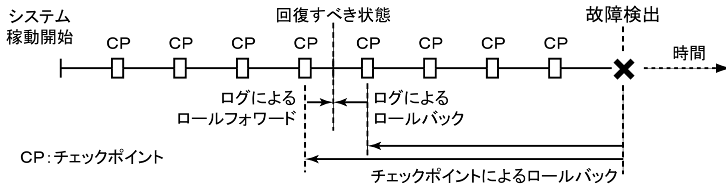
一方，後方誤り回復による手法は，誤りが発生する以前の正しい状態にシステム状態を戻す．伝統的に，システムの任意の過去の状態を復元するために，状態遷移を決定するすべての情報，すなわち**ログ**（log）を保存する方法が用いられてきた．これは**ロギング**（logging）と呼ばれる．システムの初期状態に，それ以降に取得されたログ情報を時間的に順方向に反映することで必要な状態を再構築できる．また，システムが稼働して到達したある状態に，ログ情報を時間的に逆方向に反映することで，その状態から見た過去の状態へさかのぼることができる．それらの操作をそれぞれ，**ロールフォワード**（rollforward; 巻送り）及び**ロールバック**（rollback; 巻戻し）という（図 2･4(a) 参照）．しかし多くの場合に，ロールフォワード，ロールバックともに，状態を再構築するために膨大なログ情報を走査する必要がある．そのようなオーバヘッドを削減するため，**チェックポイント**（checkpoint）と呼ばれる状態情報の控えが定期的に取得される．これは**チェックポインティング**（checkpointing）と呼ばれる．チェックポイントを用いて，回復すべき状態の近傍の状態へ直接遷移し，それを起点にロールフォワードまたはロールバックによって最終的な状態を再構築する（図 2･4(b) 参照）．これによってログへのアクセスコストを減らし，高速な状態回復が可能になる．なお，ロールフォワード及びロールバックという用語は，かつてロギングのための大容量記憶媒体として磁気テープを使用したことに由来すると考えられる．しかし，現在ではその意味は薄れ，上記のチェックポイントを用いた直接的な状態遷移も，ログによる逆方向走査と同様にロールバックと呼ばれる．

さて，本節で取り上げる分散システムのリカバリプロトコルは，上記の誤り回復についてのより具体的なモデル化を前提とする．以下，それに関して述べる．

システム内の各ノードにはそれぞれプロセッサと主記憶装置（メモリ）及び二次記憶装置が存在する．プロセスが応用プログラムを実行することで，そのプロセッサ内のレジスタと主



(a) ログファイルによる状態回復



(b) チェックポイントとログファイルによる状態回復

図 2·4　後方誤り回復

記憶の内容が持続的に更新される．プロセスは上記のログとチェックポイントを二次記憶装置に保存する．プロセスの故障によって応用プログラムの処理結果が影響を受けるが，そのとき想定される影響の程度は誤り回復の原理によって異なる．前方誤り回復では，レジスタや主記憶にある応用プログラムの処理結果の一部に誤りを含んだ状態が仮定され，この状態に対して誤りの訂正やマスクが試みられる．後方誤り回復では，レジスタや主記憶にある応用プログラムの処理結果は完全に失われるものと仮定され，二次記憶装置のログとチェックポイントによって回復が試みられる．ただし，故障が検出された直後の状態は，回復操作の起点として使えないため，チェックポイントへ直接的にロールバックしたのち，ログによってロールフォワードする．二次記憶に保持するチェックポイントの世代数は適用するリカバリプロトコルによって異なる．またそれに対応して，ロールフォワードのために必要となるログの範囲も異なる．

ノードを結ぶネットワークについてもモデル化を行う．まず，通信リンクは故障によって分断されることはないと仮定する．通信プロトコルの信頼性については，応用プログラムや使用するリカバリプロトコルによって前提が異なる．信頼性を保証する通信プロトコルを仮定する場合はメッセージの消失はなく，保証しない場合には消失があり得る．

以下では，分散システムの前方誤り回復と後方誤り回復それぞれについて解説していく．

2-5-2 分散システムの前方誤り回復

(1) 分散リカバリブロック

一つのノードの誤りを，分散システムに配置した**リカバリブロック**（recovery block）でマスクする前方誤り回復[2)] について述べる．

リカバリブロックは，ソフトウェアの誤りをマスクするための冗長構成である．同一の機能を果たす複数バージョンのソフトウェアブロックを異なる論理で実装し，主ブロックと予備ブロックに分ける．主ブロックでの実行結果を**受入テスト**（acceptance test）で審査して，合格しない場合は予備ブロックで再実行するという手順を合格するまで繰り返す．ここでは，ソフトウェアブロックを実行ブロック A と実行ブロック B の二つのバージョンに限定して，その両方を二つのノード X と Y に配置する．ただし，図 2･5 に示すように，ノード X とノード Y では主ブロックと予備ブロックのバージョンが異なるように実行する．また，受入テストには，処理結果の正しさを審査する論理受入テストと，処理のリアルタイム性を保証するための時間受入テストがあり，それぞれのノードに配置されている．

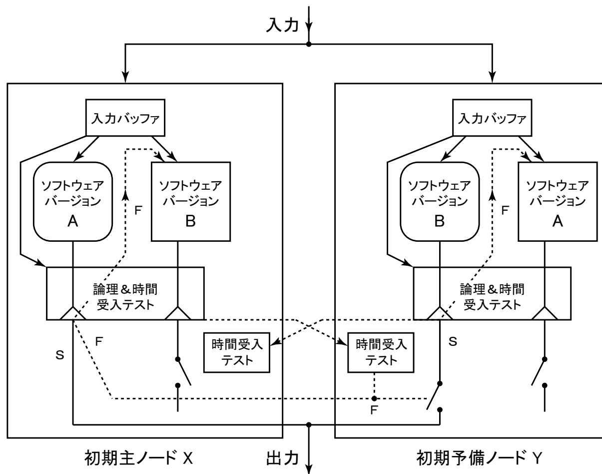


図 2･5　分散リカバリブロックの構成

初期の構成では，ノード X が主ノードとなって，ブロック A を主ブロックとして処理を実行する．同時にノード Y が予備ノードとなって，ブロック B を主ブロックとして処理を実行する．誤りが発生しないときには，主ノードの処理結果が出力される．もし誤りが検出された場合は，主ノードはそのことを予備ノードへメッセージで通知すると同時に，予備ブロック B によって再実行して処理を継続する．メッセージを受け取った予備ノードはブロック B による実行で得られた処理結果を出力する．更に，これを契機に，これまでの予備ノード Y はノード X に替わって主ノードとなり，ノード X はノード Y に替わって予備ノードとなる．このとき，ノード X は再実行で使用したブロック B を主ブロックに変更し，ノード Y は X とは逆にブロック A を主ブロックとすることで誤りの発生以前と対称な構成をとる．

分散リカバリブロックでは，単純な二重化と異なり，片側ノードの誤りで両側ノードの処理結果が無効になることを回避できる．しかしながら，複数バージョンのソフトウェアの開発・保守は難しいことが知られており，本手法の実現は必ずしも容易ではない．なお，受入テストの詳細については文献 2) などを参照されたい．

（2）プロセスペアと時間冗長による誤りマスク

二つのノードをペアにしてプロセスを二重化することで誤りを検出し，しかも，誤った側のプロセスの処理結果をマスクする前方誤り回復方式が提案されている[3)]．

このリカバリプロトコルでは，分散システムを複数のプロセスペアとそれらが共用する一つの検算用のプロセスで構成する．ペアとなった二つのプロセスは，メッセージ交換によってタスクごとの処理結果を比較し，互いの正常動作を確認する．誤りが検出された場合，検算用プロセスによって誤りを発生したプロセスを特定することで，正常プロセスが処理結果を破棄することなく継続的に次のタスクを実行できる．具体例によってこのプロトコルを説明する．

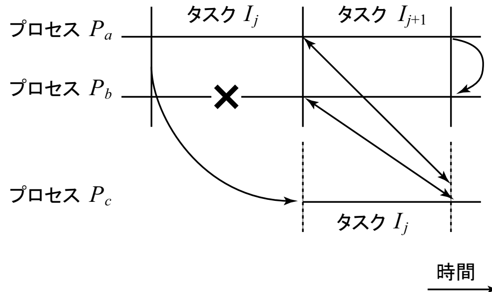


図 2･6 プロセスペアと検算用プロセスによる誤りマスク

図 2･6 は，プロセス P_a と P_b のペア及び検算用のプロセス P_c がタスクを実行する様子を表している．タスク I_j の実行においてプロセス P_b に誤りが発生している．このタスクの実行終了時にプロセス P_a と P_b は処理結果についてのメッセージ交換で誤りを検出する．ここで，このプロセスペアは検算用のプロセス P_c にタスク I_j の再実行を依頼するメッセージを送信する．プロセス P_a 及び P_b は，自らの処理結果を破棄することなく，それを元にそれぞれ次のタスク I_{j+1} を継続処理する．再実行を依頼されたプロセス P_c は，プロセスペアがタスク I_j を実行する前の状態とタスクの内容とをメッセージ交換によって受け取り，検算を開始する．再実行が完了したとき，メッセージ交換が行われ，プロセス P_a, P_b, P_c によるタスク I_j に関する三つの処理結果から，誤りを発生していたプロセス P_b を特定する．この段階で，プロセス P_a と P_b はそれぞれタスク I_{j+1} を完了しているが，プロセス P_b は誤った処理結果を引き継いで実行しているので，その結果は依然として誤っていると考えられる．そこで，プロセス P_a から処理結果をメッセージとして受け取ってこれをマスクする．

なお，上記の例で，プロセス P_a がタスク I_{j+1} を実行する際に誤りが発生し得ることを考慮したリカバリプロトコルも考察されている[3)]．

2-5-3 後方誤り回復における大域的一貫性

分散システムの後方誤り回復によるプロトコルについて説明する前に，それらが回復しようとするシステム状態の一貫性について述べる．ここで具体的に注目したいのは，システム内のプロセスは，メッセージ送信によってそれを受信するプロセスに**依存関係**（dependency）

をもたらすという事実である．

既に本章 2-5-1 節で述べたとおり，後方誤り回復プロトコルは，チェックポイントによる直接的なロールバックとログによるロールフォワードを前提としている．分散システムを構成する各プロセスが，独自に取得したチェックポイントと，故障が発生する直前までのログとを完全に保持していれば，故障したプロセスはほかのプロセスとは関係なく単独で回復処理を実行し，システムの**大域状態**（global state）を故障発生直前のそれに戻すことができる．大域状態とは，各プロセスからそれぞれがとりうる状態を一つずつ集めた集合であり，システム全体の状態を表す．

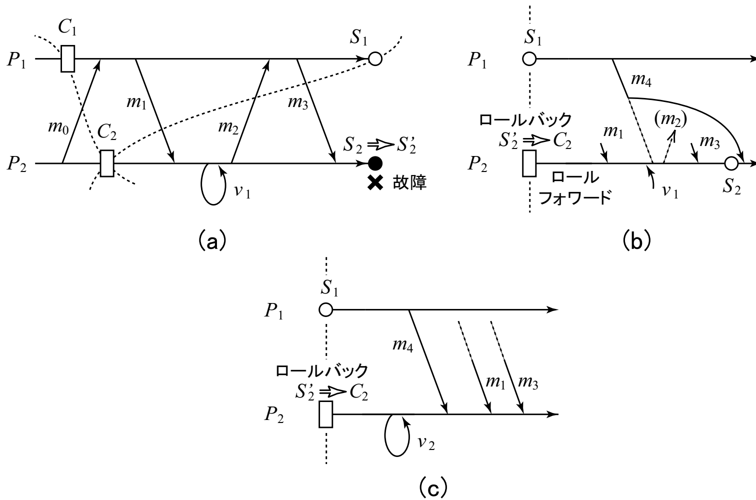


図 2･7　分散システムの後方誤り回復

例えば，二つのプロセスからなるシステムの実行概念図 2･7(a) を参照されたい．P_1, P_2 はプロセスを，C_1, C_2 はそれらのプロセスで取得されたチェックポイントを表す．また，右に方向づけされた水平線はプロセスの実行を表し，斜めに描かれた矢印はプロセス間で送受信されるメッセージを表している．この例では，プロセス P_2 に故障が発生し，本来の状態 S_2 が，応用プログラムの処理結果を失った状態 S_2' に遷移している．プロセス P_2 はチェックポイント C_2 を用いてロールバックする．このとき，システムの大域状態はプロセス P_1 の状態 S_1 とプロセス P_2 のチェックポイント C_2 の状態から構成されている．プロセス P_2 は，C_2 に故障が発生する直前までのログを決定論的に反映させて状態 S_2 を再構築する．具体的には，プロセス P_2 が処理した**決定論的事象**（deterministic event）と P_2 内部で発生した割り込みなどの**非決定論的事象**（nondeterministic event）v_1 及びプロセス P_1 からのメッセージ m_1 と m_3 の受信による非決定論的事象のすべてを故障発生前と同じ時系列で再処理する．プロセス P_2 から P_1 へのメッセージ m_2 の送信も決定論的事象の一つであるが，既にその内容はプロセス P_1 の状態 S_1 に正しく反映されているので再送信する必要はない．この回復処理の間，プロセス P_1 は応用プログラムの処理を停止する必要はない．例

えば図 2･7(b) のように，プロセス P_2 がロールフォワードする間にプロセス P_1 からメッセージ m_4 が送信されたとしても，P_2 が m_4 を受信バッファに保管しておいて，故障発生直前の状態 S_2 が再現されてからそれを反映すれば，システムの大域状態の一貫性は保たれる．

こうした後方誤り回復のプロトコルは，システム外部との入出力が頻繁にある応用に適している．例えば，勘定系などのトランザクション処理では，顧客への出力をコミットしたシステム状態は，いかなる故障が発生しても必ず回復されなければならない．ログによる決定論的な回復では，その準備のためのプロトコルを用意する必要がない．上記のチェックポイントとログによる回復は，**ログ・ベース・リカバリ**（log-based rollback-recovery）と呼ばれる[4]．

しかしながら，ログ・ベース・リカバリでは，プロセスで処理した事象を状態に反映する前にすべてログに書き出さなければならないため，通常処理時のオーバヘッドが非常に大きい．フォールトトレランス機構の設計戦略は，通常処理に対して追加的に発生するコストと故障発生後の回復処理に要するコストとのトレードオフや，信頼性への要求水準とそれを実現するための総オーバヘッドコストなどに基づいて様々に決定され得る．分散システムの後方誤り回復機構においても，そうした視点から種々のリカバリプロトコルが提案されている．

多くの科学技術計算では，回復すべき大域状態は，必ずしも故障の発生する直前の大域状態である必要はなく，正常な動作においてシステムが取り得る，何らかの一貫性のある大域状態であればよい．したがって，決定論的に状態を再構築するためのログを保持しないという選択があり得る．その場合のリカバリプロトコルは**チェックポイント・ベース・リカバリ**（checkpoint-based rollback-recovery）と呼ばれる[4]．

チェックポイント・ベース・リカバリでは，各プロセスが全体として一貫性ある大域状態を得られるようにチェックポイントを使ってロールバックし，その状態から故障前とは必ずしも同じではない新たな処理を開始する．例えば，上記の図 2･7(a) の場合，まずプロセス P_2 がチェックポイント C_2 にロールバックする．このチェックポイント C_2 とプロセス P_1 の状態 S_1 からなる大域状態に一貫性があるならば，各プロセスはこれらの状態からそれぞれ処理を再開する．この大域状態では，メッセージ m_1 と m_3 はプロセス P_1 から送信されているが，プロセス P_2 には未到着である．しかし，これはシステムの正常な動作において起こり得る状況である．仮に，システムがメッセージ送信に関して信頼性を保証するプロトコルを前提としていなければ，このメッセージの未配送は通常時のメッセージ損失と変わりなく，何らかの追加的なプロトコルによって m_1, m_3 の再配送を保証する必要もない．ところが，メッセージ m_2 については，プロセス P_2 がまだ送信していないにもかかわらず，プロセス P_1 は既にこれを受信しているという，正常なシステム動作では起こり得ない状況になっている．したがってこれは処理を再開するための一貫性のある大域状態とはいえない．

m_2 のようなメッセージを**孤児メッセージ**（orphan message），それを受信して状態に反映している P_1 のようなプロセスを**孤児プロセス**（orphan process）と呼ぶ．一方，仮にシステムがメッセージの送信に信頼性を保証するプロトコルを前提としていて，追加的なプロトコルを用いてロールバックに伴うメッセージの再配送をも保証するならば，やがて m_1 と m_3 はプロセス P_2 へ再配送される．しかし，チェックポイント C_2 を起点にして始まるプロセス P_2 の実行は，ログによる決定論的な再処理ではないので，故障発生前の非決定論的事象 v_1 については再処理される保証はなく，メッセージ m_1, m_3 の内容についても故障前

と同じ関係性で P_2 に反映される保証はない．更に，図 2･7(c) のように，プロセス P_1 からのメッセージ m_4 やプロセス P_2 内部で新たに発生した非決定論的事象 v_2 などが P_2 の状態に反映される可能性がある．すなわち，メッセージ m_2 が送信される直前の状態がプロセス P_2 に再現されることがないにもかかわらず，プロセス P_1 の状態 S_1 には既に m_2 が反映されているという矛盾した状況は依然として変わらない．もし，どうしても状態 S_1 とチェックポイント C_2 から処理を再開する必要があるならば，ログによるロールフォワードに相当する何らかの回復操作で，メッセージ m_2 が送信される直前の状態を確実に再現する必要がある．

図 2･7(a) において，状態 S_1 とチェックポイント C_2 からなる大域状態に一貫性がないことは，それらを結ぶ線によって分けられた図の右側の領域から左側の領域へメッセージ m_2（孤児メッセージ）が送信されていることで判断できる．そうして，チェックポイント・ベース・リカバリのプロトコルは一貫性のあるほかの大域状態を捜して，今度はプロセス P_1 をメッセージ m_2 を受信する以前のチェックポイント C_1 へロールバックさせることになる．チェックポイント C_1 と C_2 を結ぶ線によって分けられた右側の領域から左側の領域へ送信されるメッセージは存在しないので，この大域状態には一貫性がある．チェックポイントを結ぶ線が表す大域状態に一貫性があるとき，その線を**回復線**（recovery line; リカバリライン）と呼ぶ．回復線の左側の領域から右側の領域へ送信されているメッセージ m_0 の再配送については，保証されていても，されていなくても一貫性には影響しない．なお，このようにあるプロセスのロールバックがほかのプロセスのロールバックを引き起こすことを**ロールバック伝播**（rollback propagation）という．チェックポイント・ベース・リカバリでは，ロールバック伝播が次々に連鎖的に発生して，回復線がシステムの初期状態となる可能性がある．そのような現象は特に**ドミノ効果**（domino effect）と呼ばれる．

以下では，ログ・ベース・リカバリとチェックポイント・ベース・リカバリそれぞれの具体的なプロトコルについて述べる．

2-5-4 ログ・ベース・リカバリ

ログ・ベース・リカバリのプロトコルは**悲観的ロギング**（pessimistic logging），**楽観的ロギング**（optimistic logging），**因果関係ロギング**（causal logging）の三つに分類できる[4)]．

(1) 悲観的ロギング

悲観的ロギングは，本章 2-5-3 節で示したログ・ベース・リカバリの概念を実現する最も基本的なプロトコルである．メッセージ受信などの非決定論的な事象を応用プログラムの処理に反映させる前に，必ず関連するログの取得を完了する．これは，故障がロギングの前に発生するという悲観的な発想に基づいている．ログは，故障に耐性のある二次記憶媒体に保管されるため，すべての非決定論的な事象の処理を，決定論的な事象の処理との順序関係も含めて，完全に再現することができる．

悲観的ロギングでは，回復のための情報（ログとチェックポイント）は各プロセスごとに保持される．故障が発生したプロセスは最新のチェックポイントへロールバックするが，ほかのプロセスへのロールバック伝播は起こらないため，状態回復は故障プロセス単独で完結する．旧世代のチェックポイントが必要となることはないため，新しいチェックポイントの取得が完了すればそれまで保持していた古いチェックポイントとログを破棄することができ，

二次記憶装置の容量管理は容易である．また，常に故障直前の大域状態を故障プロセスだけで再現できるため，他プロセスと連携して回復に備える特別なプロトコルが必要なく，システム外部からのトランザクションなどを当該プロセスだけでコミットすることができる．

しかし明らかに，悲観的ロギングでは，通常処理中に発生する二次記憶装置への大きなアクセスオーバヘッドが欠点となる．このオーバヘッドを回避するためのプロトコルが提案されている[5)]．そのプロトコルでは，プロセスはログを二次記憶へは書き込まず，メッセージの再配送に必要な情報を，そのメッセージの送信元プロセスの主記憶に保持させる．故障に備えてプロセス間で事前に連携する必要があり，その手順は，(a) 送信プロセスが，メッセージの内容を主記憶に記憶する；(b) 受信プロセスが，メッセージの配送順序を含む受信通知を返す；(c) 送信プロセスは，受信通知から得た配送順序を主記憶のメッセージの内容と合わせて記憶する；というものである．状態回復では，送信プロセスから元のメッセージとその配送順序の情報が故障プロセスに提供されるが，同時に回復処理できるのは一つのプロセスだけである．また，故障プロセスの内部に非決定論的な事象が発生していた場合にはこれに対応することはできない．

(2) 楽観的ロギング

楽観的ロギングは，ロギングによる二次記憶装置へのアクセスオーバヘッドを削減するために，非決定論的な事象に関連するログをそれらの処理とは非同期にまとめて二次記憶へ書き出す．故障はロギングのあとに発生するという楽観的な発想に基づいている．このプロトコルでは，悲観的ロギングより通常処理中のオーバヘッドが小さくなる代償として，二次記憶装置の容量管理や回復処理時のプロトコルが複雑化する．

故障プロセスの状態回復では，関係する全プロセスが回復処理に参加する必要がある．故障が発生したプロセスは，チェックポイントへロールバックしたのちにロールフォワードするが，二次記憶に書き出したログに対応する状態までしか復帰できない．まだ二次記憶に書き出していなかったログの内容は失われ，それに対応した事象の完全な再処理はできない．そのため，本章 2-5-3 節で述べたチェックポイント・ベース・リカバリと同様に，その間に送信されたメッセージを受信したプロセスは孤児プロセスとなる．そこで，故障プロセスだけでなく，孤児プロセスも孤児メッセージの受信がキャンセルされるチェックポイントまでいったんロールバックし大域状態の一貫性を形成できるプロセスの状態までロールフォワードする．

例えば，図 2･8(a) のような依存関係にあるプロセス P_1, P_2, P_3 を考える．$C_{x,\,y}$ はプロセス P_x の y 番目のチェックポイントを表している．また，L_x はプロセス P_x のログが二次記憶へ最後に書き出された時点の状態を表している．プロセス P_2 は，故障を検出したのち，チェックポイント $C_{2,j}$ までロールバックし，ログによって復帰可能な状態 L_2 までロールフォワードする (図 2･8(b))．プロセス P_2 と P_3 の依存関係に注目すると，L_2 と S_3 を結ぶ線によって分けられる右側の領域から左側の領域へメッセージ m_9 と m_{11} が送信されていることから，プロセス P_3 が孤児プロセスとなることが分かる (図 2･8(a))．そこで P_3 は m_9, m_{11} を受信していないチェックポイント $C_{3,k}$ へロールバックし，ログを使って m_9 を受信する直前の状態 $\hat{S}_3$ までロールフォワードする (図 2･8(b))．また同様に，プロセス P_2 と P_1 の依存関係から，P_1 はメッセージ m_7 を受信する直前の状態 $\hat{S}_1$ まで復帰する (図 2･8(b))．ただし，その過程でプロセス P_1 はチェックポイント $C_{1,i}$ ではなく

$C_{1,i-1}$ へロールバックすることに注意されたい．このように，楽観的ロギングでは，複数世代のチェックポイントとそれに対応するログを保持する必要がある．また悲観的ロギングとは異なり，システム外部との入出力のコミットには，関係するすべてのプロセスが連携して一貫性ある大域状態を形成するような追加的なプロトコルが必要となる．例えば，図 2･8(a) の故障が発生する前に，プロセス P_1 が状態 S_1 において外部への出力をコミットしようとするならば，プロセス P_1 は自身の非決定論的事象に関するログを二次記憶に書き出し，プロセス P_2 の故障で P_1 が孤児プロセスにならないように P_2 にメッセージ m_7 の送信までのログを二次記憶へ書き出すように要請しなければならない．

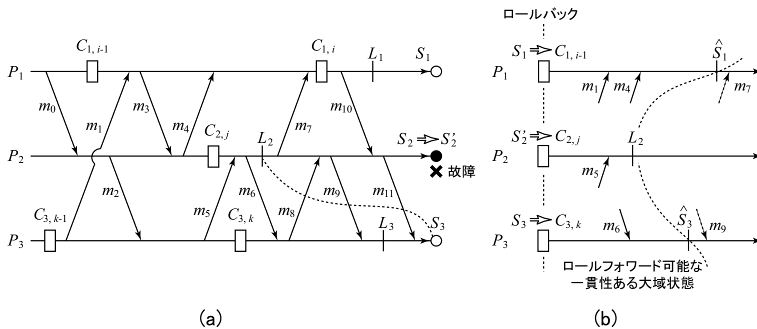


図 2･8　楽観的ロギングにおける回復処理実行例

楽観的ロギングには，各プロセスが全体として復帰すべき一貫性ある大域状態を特定して，それぞれに回復処理を実行する手法が二つある．一つは，故障プロセスがほかのプロセスにロールバックを要請するメッセージをブロードキャストすることから始まり，上記のように順次依存関係にあるプロセスがロールバックしていく手法である[6)]．もう一つは，あらかじめ各プロセスの状態遷移にともなって更新されるインデックス値を関係プロセスが共有してもち，回復処理時には各プロセスがそれぞれ一斉にインデックス値とログ情報から一貫性ある大域状態を算出する手法である[7)]．各プロセスのインデックス値はアプリケーションメッセージに添付して伝達される．

(3) 因果関係ロギング

因果関係ロギングは，悲観的ロギングと楽観的ロギングのそれぞれの特長を併せたプロトコルである[8, 9)]．楽観的ロギングと同様に通常処理中の二次記憶へのアクセスオーバヘッドを削減しながら，悲観的ロギングのほとんどの利点を備えている．

このプロトコルの基本的なアイディアは，各プロセスの主記憶にそのプロセスと依存関係にある全プロセスの非決定論的事象の実行順序に関する情報を保持させて，それらの依存関係にあるプロセスが故障したときのロールフォワードを支援することで，自身が孤児プロセスとなることを回避するというものである．例えば，図 2･9(a) のような実行例を考える．仮に，状態 S_2 にあるプロセス P_2 が故障したとき，状態 S_1 にあるプロセス P_1 が孤児プロセスにならないためには，プロセス P_2 のロールフォワードにおいて，メッセージ m_2 の送信にいたるまでの非決定論的事象，すなわちメッセージ m_1 の受信を正しい因果関係で再現

する必要がある．そのための情報をプロセス P_1 が保持しておけば，プロセス P_2 のログがその主記憶から消失し，二次記憶に記録されていなくても再処理を支援することができる．

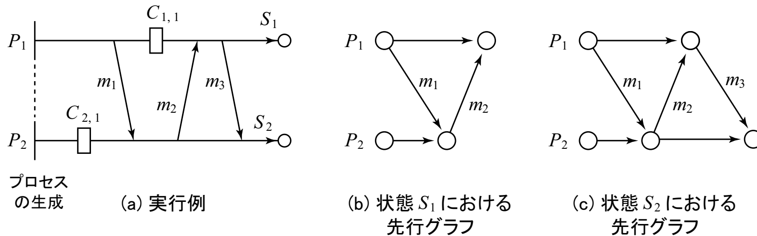


図 2･9　因果関係ロギングの先行グラフ

状態 S_1 においてプロセス P_1 が保持するこのような因果関係の情報は，図 2･9(b) の**先行グラフ**（antecedence graph）で表現できる[8]．グラフのノードが非決定論的事象を表し，矢印が**因果関係**（happened-before relation, causal relation）[10] を表している．ただし，ここではプロセスの生成を，各プロセス内部で発生した非決定論的事象であると考えている．プロセス P_1 は，この先行グラフを主記憶で持続的に更新しながら維持する．具体的には，(a) はじめに P_1 がメッセージ m_1 を送信してその内容と実行順序を記憶する；(b) m_1 を受信したプロセス P_2 はその順序をメッセージ m_2 に添付して P_1 へ送信する；(c) m_2 を受信したプロセス P_1 は得られた因果関係で先行グラフを更新する；という手順をとる．回復操作では，このグラフの示す因果関係をプロセス P_2 に通知する．なお，m_1 の内容の再送は P_1 が実行してもよいし，別の追加的なプロトコルが保証してもよい．一方，仮に，状態 S_1 にあるプロセス P_1 が故障したとき，状態 S_2 にあるプロセス P_2 が孤児プロセスにならないためには，プロセス P_1 のロールフォワードにおいて，メッセージ m_3 の送信前の事象であるメッセージ m_2 の受信を正しい因果関係で再現する必要がある．そのための情報は図 2･9(c) の先行グラフとしてプロセス P_2 の主記憶に構成される．

因果関係ロギングの主な欠点は，回復操作が非常に複雑になることである．しかし，先行グラフを表す情報をログとして主記憶に保持すれば，ロギングのための二次記憶装置へのアクセスオーバヘッドを除去できる．また，チェックポイントは悲観的ロギングと同様に最新世代だけが必要とされるため，二次記憶のオーバヘッドも最小化できる．更に，特別な連携プロトコルを用いなくても各プロセスが孤児プロセスにならないための準備ができるため，システム外部への出力コミットも当該プロセス単独で実行することができる．

2-5-5 チェックポイント・ベース・リカバリ

チェックポイント・ベース・リカバリのプロトコルは**非連携チェックポインティング**（uncoordinated checkpointing）, **連携チェックポインティング**（coordinated checkpointing）, **通信誘導チェックポインティング**（communication-induced checkpointing）の三つに分類できる[4]．

(1) 非連携チェックポインティング

非連携チェックポインティングでは，各プロセスが任意の時点でチェックポイントを取得

する．ほかのリカバリプロトコルのように，一貫性ある大域状態を構築するための事前準備を一切実行しないため，通常処理中のオーバヘッドは最も小さい．しかしながら，故障が発生したときには，各プロセスの最新のチェックポイントを用いて一貫性のある大域状態を形成できるとは限らないため，複数世代のチェックポイントの保持が必要である．また最悪の場合，ドミノ効果が発生してシステムの通常処理結果をすべて失う可能性もある．

回復操作には，全プロセスの参加が必要となる．典型的な手順は，(a) 障害プロセスが始動者（initiator）となってリカバリプロトコルを発動する；(b) 全プロセスからプロセス間依存情報を収集する；(c) 回復線を特定する；(d) 全プロセスに回復線へのロールバックを要請する；である．回復線を特定するための計算には，プロセス間の依存関係を表すグラフが使われる．具体的には，**ロールバック依存グラフ**（rollback-dependency graph）[11]や，**チェックポイントグラフ**（checkpoint graph）[12] による手法が提案されている．

非連携チェックポインティングが実用になるのは，プロセス間のメッセージ交換頻度が比較的小さく，各プロセスが保持できるチェックポイント世代数が十分に大きな場合である．なお，明示的に一貫性ある大域状態を形成するための手順をもたないため，システム外部への入出コミットは実行できない．

(2) 連携チェックポインティング

連携チェックポインティングでは，各プロセスの取得するチェックポイントが一貫性ある大域状態を形成するようにメッセージ交換で制御する．基本的に，すべてのプロセスがチェックポイント生成プロトコルに参加して連携する必要がある．各プロセスが二次記憶装置に保存している直近の状態を使って，直ちに一貫性ある大域状態を再構成できるため，回復処理では，故障プロセスがほかのプロセスにそれらの状態へロールバックするよう要請するだけで基本的な手続きが完了する．

連携チェックポインティングで一貫性ある大域状態を生成する最も素朴な方法は，全体のチェックポイントの取得が完了するまでの間，依存関係をもたらすプロセス間通信を停止するものである[13]．具体的には，(a) 調停プロセスがチェックポイントを取得して，全プロセスにリクエストメッセージをブロードキャストする；(b) メッセージを受け取ったプロセスは応用プログラムの実行とプロセス間通信を中止し，仮のチェックポイントを取得してから，調停プロセスへ ACK(肯定応答) を返す；(c) すべてのプロセスから ACK を受け取ったあとで，調停プロセスがコミットメッセージを全プロセスへ送信する；(d) コミットメッセージを受け取った各プロセスは，これまでの常設のチェックポイントを破棄して，仮のチェックポイントを常設のチェックポイントへ変え，通常処理を再開する；という手順をとる．

上記の手法の欠点は，チェックポイントを生成するごとに，通常処理中に非常に大きなオーバヘッドが生ずることである．このオーバヘッドを削減するため，チェックポイント取得中の通信を停止することなく大域状態を生成する**非ブロッキングチェックポイント連携手法**（non-blocking checkpoint coordination）が提案されている[14]．また，分散システムの緩やかな同期クロックを用いて一貫性ある大域状態を生成する手法[15]や，**2 相プロトコル**（two-phase protocol）と呼ばれる手続きに基づいて，チェックポイント生成に参加するプロセス数を最小化する手法[16]など，連携チェックポインティングに関連する多くのプロトコルが研究されている．

(3) 通信誘導チェックポインティング

各プロセスがほかのプロセスと調停することなく独立にチェックポイントを取得するという点で，通信誘導チェックポイントは非連携チェックポインティングに類似している．そのため，明示的に連携して一貫性ある大域状態を生成するためのオーバヘッドは存在しない．回復線の存在によってドミノ効果は回避できるが，常に最新のチェックポイントでロールバック伝播が停止するとは限らないため，複数世代のチェックポイントを保持する必要がある．また，その回復線は分散システムの処理が進むにつれて古くなるため，回復処理時のオーバヘッドを制限し，チェックポイントための二次記憶容量を許容範囲内に納めるには，一貫性ある大域状態を継続的に再構築して回復線を更新する必要がある．

このプロトコルでは，応用プログラムのメッセージにリカバリプロトコル固有の情報を添付して，プロセス間のメッセージ交換の履歴を伝達する．チェックポイントには，プロセスが自らの都合で取得する**局所チェックポイント**（local checkpoint）と，大域的な一貫性の観点から必要と判断して取得する**強制チェックポイント**（forced checkpoint）とがある．新しい強制チェックポイントは，過去に取得したチェックポイントの時間配置とメッセージ履歴の情報から判断して，回復線が更新されるように各プロセスが独自に取得する．

そのための重要な基準の一つが **Z パス**（Z-path; zigzag path）と **Z サイクル**（Z-cycle; zigzag cycle）によるチェックポイントの有効性の判断である[17)]．Z パスは二つのチェックポイントを結ぶ特殊なメッセージ系列で，始点と終点のチェックポイントが同一ならば Z サイクルとなり，そのチェックポイントを含む大域状態が一貫性をもたないことを示す．

この Z サイクルで検出される無効なチェックポイントを特定して，有効な強制チェックポイントを取得するいくつかの手法がある．**ロールバック依存追跡性**（rollback-dependency trackability）という概念に基づいてチェックポイントとメッセージ履歴のモデル構造を解析する手法[18)]や，受信メッセージに添付されたチェックポイントの**時刻印**（timestamp）によって，同じ時刻印をもつチェックポイントの集合が回復線を形成するように強制チェックポイントを取得する手法[19)]，更に，その強制チェックポイントの数を最小化する手法[20)]などである．

なお，チェックポイント・ベース・リカバリの文献 15, 17, 20) のプロトコルに関しては，文献 21) に和文の解説があるので参照されたい．

■参考文献

1) D.K. Pradhan, “Fault-Tolerant Computer System Design,” Prentice-Hall, 1996.
2) K.H. Kim, and H.O. Welch, “Distributed Execution of Recovery Blocks: An Approach for Uniform Treatment of Hardware and Software Faults in Real-Time Applications,” IEEE Transactions on Computers, vol.38, no.5, pp.626-636, 1989.
3) D.K. Pradhan, N. Vaidya, “Roll Forward Checkpointing Scheme: A Novel Fault-Tolerant Architecture,” IEEE Transactions on Computers, vol.43, no.10, pp.1163-1174, 1994.
4) E.N. Elnozahy et al., “A survey of rollback-recovery protocols in message passing systems,” ACM Computing Surveys, vol.34, no.3, pp.375-408, 2002.
5) D.B. Johnson, and W. Zwaenepoel, “Sender-based message logging,” In Digest of Papers, FTCS-17, The Seventeenth Annual International Symposium on Fault-Tolerant Computing, pp.14-19, 1987.

6) R. Strom, S. Yemini, “Optimistic recovery in distributed system,” ACM Trans. Comput. Syst. vol.3, no.3, pp.204-226, 1985.
7) A. Sistla, and J. Welch, “Efficient distributed recovery using message logging,” In Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing, pp.223-238, 1989.
8) E.N. Elnozahy, W. Zwaenepoel, “Manetho: Transparent Rollback-Recovery with Low Overhead,” Limited Rollback, and Fast Output Commit, IEEE Trans. Comput., vol.41, no.5, pp.526-531, 1992.
9) L. Alvisi, and K. Marzullo, “Message logging: pessimistic, optimistic, causal and optimal,” IEEE Trans. Softw. Eng., vol.24, no.2, pp.149-159, 1998.
10) L. Lamport, “Time, clocks and the ordering of events in a distributed system,” Commun. ACM, vol.21, no.7, pp.588-565, 1978.
11) B. Bhargava, S.R. Lian, “Independent checkpointing and concurrent rollback for recovery ? An optimistic approach,” In Proceedings, Seventh Symposium on Reliable Distributed Systems, pp.3-12, 1988.
12) Y.-M. Wang, “Space Reclamation for Uncoordinated Checkpointing in Message-Passing Systems, Ph.D. Thesis,” University of Illinois, Department of Computer Science, 1993.
13) Y. Tamir, and C.H. Sequin, “Error recovery in multicomputers using global checkpointings,” In Proceedings of the International Conference on Parallel Processing, pp.32-41, 1984.
14) M. Chandy, and L. Lamport, “Distributed snapshots: Determining global states of distributed systems,” ACM Trans. Comput. Syst., vol.3, no.1, pp.63-75, 1985.
15) F. Cristian, and F. Jahanian, “A timestamp-based checkpointing protocol for long-lived distributed computations,” In Proceedings, Tenth Symposium on Reliable Distributed Systems, pp.12-20, 1991.
16) R. Koo, and S. Toueg, “Checkpointing and rollback-recovery for distributed systems,” IEEE Trans. Softw. Engin., vol.13, no.1, pp.23-31, 1987.
17) R.H. Netzer, and J. Xu, “Necessary and sufficient conditions for consistent global snapshots,” IEEE Trans. Parallel and Distributed Systems, vol.6, no.2, pp.165-169, 1995.
18) Y.-M. Wang, “Consistent global checkpoints that contain a set of local checkpoints,” IEEE Trans. Comput., vol.46, no.4, pp.456-468, 1997.
19) D. Briatico, A. Ciuffoletti, and L. Simoncini, “A Distributed Domino-Effect Free Recovery Algorithm,” Proc. IEEE Int’l Symp. Reliability Distributed Software and Database, pp.207-215, 1984.
20) J.-M. Helary, A. Mostefaoui, R.H. Netzer, and M. Raynal, “Preventing useless checkpoints in distributed computations,” In Proceedings, Sixteenth Symposium on Reliable Distributed Systems, pp.183-190, 1997.
21) 米田友洋，梶原誠司，土屋達弘, “ ディペンダブルシステム, ” 共立出版, 2005.