数理解析研究所講究録
950巻 1996年 101-105

複数プロセス故障を許した耐故障分散相互排除アルゴリズム*

武川 茂樹　若林 真一　小出 哲士
広島大学工学部

1 まえがき

分散システムの基本的な問題の一つに相互排除問題がある．相互排除問題は，複数の競合するプロセスの中から一つのプロセスに特別な権利を与える問題で，例えばプリンタやファイルなどの共有資源を使用する場合に生じる．

相互排除問題を解く最も簡単なアルゴリズムは，ある1つのプロセスが調停者となる手法 [1] である．しかし，分散システムでは，システムを管理する特別なプロセスは存在しないことが一般的である．そこで，各プロセスが同等の責任を持ちながら問題を解くことが望まれる．文献 [3, 5] では，全てのプロセスが調停者となるアルゴリズムを提案している．つまり，資源を要求するプロセスは全てのプロセスから許可を得れば資源を使用できることとして，相互排除問題を解く．しかし，これらのアルゴリズムでは，システム中のプロセス数を N とすると，資源の獲得に $\Omega(N)$ のメッセージが必要となる．コータリ [2] を用いる手法 [4, 6] では，最大のコーラムサイズを c とすると $O(c)$ のメッセージで相互排除を実現できるという特徴を持つ．

一方，近年，分散システムの規模は大きくなってきており，接続されているすべての計算機や通信リンクが無故障であると仮定することは現実的ではない．従って，分散アルゴリズムを設計する際，システムの一部に故障が存在する場合においても目的を達成する耐故障性を考慮する必要がある．上記に述べた手法 [3, 5] では，1つのプロセスが故障するだけで相互排除問題を正しく解くことができなくなる．一方，コータリを用いる手法では，あるコーラム中の全てのプロセスが正常ならば相互排除を実現できる．しかしこの場合，アルゴリズム実行中にはプロセスは故障しないという仮定をおく必要があり，任意の時刻に故障が起きると問題が解けなくなる．

そこで著者らは，任意の時刻に高々1つのプロセスが故障すると仮定した場合の相互排除アルゴリズムの提案を行なった [7]．この手法は，文献 [4] のアルゴリズムに対し，タイマを用いて故障プロセスを検出しコータリを動的に更新することで，システムを故障のない状態に導くことにより相互排除を実現した．

本稿では，[7] のアルゴリズムを拡張し，故障するプロセスの個数を任意とした場合の相互排除アルゴリズムを提案する．各プロセスは，コータリ中の故障プロセスに対する置き換えプロセスを定義する更新テーブルを保持し，コータリとこの更新テーブルを動的に更新することでシステムを故障のない状態に導く．

2 準備

2.1 分散システムのモデル

対象とする分散システムでは，以下の仮定が成り立つものとする．

- 各プロセスは共有メモリを持たず，プロセス間の情報のやりとりはメッセージパッシングのみとする．
- 同期システムを仮定する．
- メッセージは送信された順番で受信されるとする（FIFO 型システム）．
- ネットワークの形状は，完全グラフとする．
- プロセスの故障は停止故障とする．
- 通信リンクは故障しない．

2.2 諸定義

定義 1 コータリ [2]

U をプロセスの集合とする．$C = \{Q_1, Q_2, \cdots, Q_m\}$ が以下の条件を満たす時，C をコータリと呼ぶ．コータリ中の各 Q_i はコーラムと呼ばれ，$Q_i \subseteq U$，$Q_i \neq \emptyset$ であり，且つ次の2つの条件を満たす．

1. 全ての対 i と $j(1 \leq i, j \leq m)$ に対して，$Q_i \cap Q_j \neq \emptyset$.
2. 全ての対 i と $j(1 \leq i, j \leq m, i \neq j)$ に対して，$Q_i \not\subseteq Q_j$. □

定義 2 コータリの更新

プロセスの集合を U_1, U_2 とし，$x, y \in U_1$，$U_2 = U_1 - \{x\}$ とする．C_1 は U_1 の下でのコータリ，$G, H \in C_1$ はコーラムとする．

- コータリ中の要素（プロセス）x を y に置き換える関数 $T_{xy}(C_1)$

$$T_{xy}(C_1) = \{CT_{xy}(G) \mid G \in C_1\}$$

$$CT_{xy}(G) = \begin{cases} G - \{x\} & if\ x \in G \wedge y \in G \\ (G - \{x\}) \cup \{y\} & if\ x \in G \wedge y \notin G \\ G & otherwise \end{cases}$$

- $C_3 = T_{xy}(C_1)$ から $G' \subseteq H'(G' \in C_3,\ H' \in C_3 - G')$ なる集合 G' を取り除く関数 $Q(C_3)$

$$Q(C_3) = \{C_3 - DQ(G', H') \mid G' \in C_3, H' \in C_3 - G'\}$$

$$DQ(G', H') = \begin{cases} G' & if\ G' \subseteq H' \\ \emptyset & otherwise \end{cases}$$

---

* "A Coterie-Based Fault-Tolerant Algorithm for Mutual Exclusion in a Synchronous Distributed System," by Shigeki TAKEKAWA, Shin'ichi WAKABAYASHI and Tetsushi KOIDE, Faculty of Engineering, Hiroshima Univ., e-mail:waldy@ecs.hiroshima-u.ac.jp.

この時，$C_2 = Q(T_{xy}(C_1))$ を**更新後のコータリ**と呼ぶ．□

定義 3
システム中のプロセス集合を $U = \{1, 2, \ldots, N\}$，$|U| = N$，故障プロセスの上限を $K(< N)$，故障プロセスの集合を U_K とする．□

定義 4 各リンク中の通信遅延の最大値を σ とする．□

定義 5 各プロセスは，2つのタイマ T_{max}，T_d を持つとする．各プロセスがメッセージを受信してから返信するまでの処理時間の最大値を λ とすると $T_d = 2\sigma + \lambda$ とする．□

定義 6 各プロセスは，配列 $update[i \in U] \in U$ を持つ．この配列を**更新テーブル**と呼ぶ．□

定義 7 有向グラフ $G = (V, E)$，$(V \subseteq U, E = \{(v_i, update[v_i]) | v_i \in V\})$ を，$V \subseteq U$ の下での**更新グラフ**と呼ぶ．□

定義 8 更新テーブル $update[i \in U]$ が，全ての $|U_j| \leq K$ なる集合 $U_j \subset U$ に対し $update[i \in U_j] \notin U_j$ なる $update[i]$ が存在するならば，この更新テーブルを **K–耐故障**であると呼ぶ．□

2.3 前川のアルゴリズム

コータリを用いる相互排除アルゴリズムとして，前川のアルゴリズム [4] がある．

各プロセス P_i は，あるコーラム Q_i 中のすべてのプロセスから資源の使用を許可されれば資源を使用する．即ち，各プロセス P_i はあるコーラム Q_i 中の全てのプロセスに要求メッセージ *Request* を送信する．この *Requset* メッセージを受信したプロセスは，もし他のプロセスに許可を送信していない場合 (送信している場合そのプロセスにロックされているという)，即座に許可メッセージ *Locked* を送信する．あるコーラム中の全てのプロセスから許可メッセージ *Locked* を受信すれば資源を使用できる．資源の使用終了後は資源の使用終了メッセージ *Release* をそのコーラム中の全てのプロセスに送信しロックを解除する．

このアルゴリズムが相互排除を保証することは，以下に示す事実から明らかである．

- 各コーラムは他の全てのコーラムと少なくとも 1 つのプロセスを共有している．
- 許可は一つのプロセスにしか与えないとし，一度許可を与えるとそのプロセスの許可なしでは許可を取り消すことはできない．

しかし，上記のアルゴリズムではデッドロックに陥る可能性がある．そこで前川のアルゴリズムでは，要求メッセージ *Request* にタイムスタンプ [3] を付けることによりプライオリティをつける．許可メッセージ *Locked* を送信しているプロセス (他のプロセスにロックされているプロセス) がプライオリティの高い要求メッセージを受信した場合，先に送信した許可メッセージを取消し，そのプライオリティの高い要求メッセージを送信したプロセスに許可を送信する．これによりデッドロックを解除する．

この前川のアルゴリズムは，アルゴリズム実行中にプロセスが故障すると正しい動作が保証されず，次の 3 つの問題が生じる．

1. メッセージを永久に待つ．
2. 全てのコーラムが故障プロセスを含む時，相互排除が実現できない．
3. 一度送信した要求を取り消す必要がある．

1. は，故障プロセスからのメッセージを待っていたプロセスはそのメッセージを無限に待ってしまうことを示す．2. については，各プロセスは資源を獲得する際あるコーラム中の全てのプロセスから許可メッセージを受信する必要がある．しかし，全てのコーラムが故障プロセスを含む場合あるコーラム中の全てのプロセスから許可を受信できないので，相互排除ができない．また 3. については，図 1を用いて説明する．図 1(a) はあるプロセス P_i が資源の獲得のためあるコーラム G_1 中の全てのプロセスに要求を送信している場合である．この時 G_1 中の 1 つのプロセスが故障していたとする．P_i は資源獲得のため別のコーラム G_2 に要求を送信する必要があるが，図 1(b) の様に以前に G_1 中の全てのプロセスに送信していた要求メッセージを取り消す必要がある．

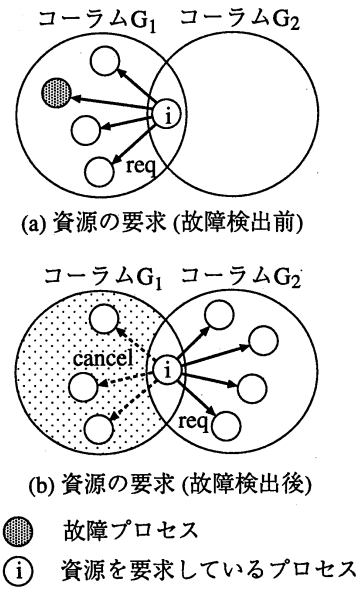

図 1 資源の要求メッセージの取消

そこで，本稿ではこれらの問題点を解決するアルゴリズムの提案を行なう．

3 提案アルゴリズム

3.1 アルゴリズム

本稿で提案する耐故障分散相互排除アルゴリズムの概要について述べる．アルゴリズムは，(1) 故障プロセスの検出，(2) メッセージ待ちからの解除，(3) コータリの更新，の 3 つのステップからなり，これらにより 2.3 で述べた文献 [4] のアルゴリズムの問題点を解決する．プロセスの故障が存在しない場合，アルゴリズムは文献 [4] のアルゴリズムと同じ動作をする．以下に提案アルゴリズムの概要を示す．

1. 故障プロセスの検出
   故障を検出するために 2 つのタイマ T_{max} と T_d を用いる．資源の使用を要求するプロセス P_i は，あるコーラム $G \in C$ 中の全てのプロセスに要求メッセージ *Request* を送信する．P_i は G 中の全てのプロセスから T_{max} 時間，許可メッセージ *Locked* を受信するのを待つ．もしタイムアウトすればそのプロセス (P_j とする) に *Is_allright* を送信し，P_j が故障しているかどうかを直接調べる．*Is_allright* を受信したプロセス P_j は，正常ならば即座に *Allright* を返信する．もし，P_i が T_d 時間以内に *Allright* を受信しなかった場合，P_i は P_j を故障しているとみなす．一方，P_i が *Allright* を受信した場合は，P_j から *Locked* を受信するのをもう T_{max} 時間待つ．以上と同様の操作を，許可 *Locked* を送信して資源の使用終了メッセージ *Release* を受信する場合，および許可を譲渡して再度許可を受信する場合についても行なう．

2. メッセージ待ちからの解除
   故障を検出したプロセス P_i は，故障プロセスを示す故障検出メッセージ *Down* を全てのプロセスに送信する．*Down* を受信したプロセスは，故障プロセスによるメッセージ待ちの解除を行ない，もし故障プロセスから要求を受信していた場合は削除する．

3. コータリの更新
   故障検出メッセージ *Down* を受信したプロセスは，更新テーブル *update* を更新し，さらにこのテーブルを基にコータリの更新を行なう．また，この *Down* メッセージ受信後 σ 時間資源の使用を禁止する．σ 時間後，資源の使用禁止を解除する．*Down* を受信する前にあるコーラム $G \in C$ に資源の要求をしていた場合，更新後のコーラム G' 中のプロセスでまだ要求を送信していないプロセス $(G' - G)$ に対して要求を行ない，G' 中の全てのプロセスから許可を受信するのを待つ．資源の禁止状態を解除後，あるコーラム中の全てのプロセスから許可を受信している場合，資源を使用する．

以降では，3. のコータリの更新において更新テーブルを用いた更新方法について詳しく述べる．

3.2 更新テーブル

更新テーブル $update[j]$ は，プロセス j が故障した場合コーラム中のプロセス j を $update[j]$ で置き換えることを示す．故障プロセス j を検出したことを示すメッセージ $Down(j)$ を受信したプロセスは，まず次に示す手続きによりこの更新テーブルを更新する．

[テーブル更新アルゴリズム *update_table*]
全ての $update[i] = j$ なる $i \in U$ に対し，$update[i] = update[j]$. §

なお，この後このテーブルを基に次の節で示すコータリ更新関数 $C' = Q(T_{xy}(C))$ によりコータリの更新を行なう．

次に，このテーブルの初期値について議論する．更新テーブル *update* は，故障プロセスをどの正常プロセスで置き換えるかを示している．更新テーブルを更新する際，$update[i] = i$ なるプロセス i が存在していて i が故障した場合，更新を行なうことができない．$i = 4$ の時の更新テーブルの例を図 2に示す．従って，K回の更新を行なう際このような状態が起こらないように初期更新テーブルを作成する必要がある．次の重要な補題がある．

1	2	3	4	5	6	7
5	6	6	4	2	1	5

図 2 テーブルの更新ができない例

補題 1 更新テーブルの初期状態が K−耐故障ならば，任意の K個のプロセス故障に対し，更新テーブルにおいて $update[i] = i$ なる i が存在しプロセス i が故障するという状態は起こらない．

(証明) Uの下での更新グラフについて議論する．更新テーブルとそれにより求まる更新グラフの例を図 3に示す．更新テーブルは K−耐故障なので，更新テーブル中のサイクルで枝の数が K以下のものは存在しない．一度の更新で更新グラフから取り除かれるのは一つの節点とその節点の外向枝のみである．従って，更新中に自己ループ ($update[i] = i$ なる i が存在することに対応) が存在するのは，少なくとも $K + 1$ 回目の更新時のみである． □

従って，更新テーブルが K−耐故障になるように初期値を決めれば良い．この条件を満たす更新テーブルの初期値を決めるには，色々な手法が考えられるが，例えば次のように更新テーブルを設定することで $(N-1)$-耐故障である更新テーブルを作成することができる．これは，$N-1$ 個のプロセスが故障してもテーブルの更新が可能であることを示している．

$$update[i \in U] = \begin{cases} i+1 & (1 \le i \le N-1) \\ 1 & (i = N) \end{cases}$$

このように設定された更新テーブルを以降では**初期更新テーブル**と呼ぶ．

1	2	3	4	5	6	7
5	6	6	1	2	4	5

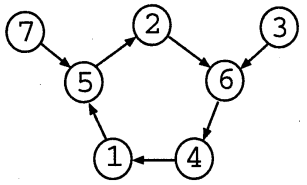

図 3 更新テーブルとその更新グラフ

補題 2 初期更新テーブルは，$N-1$ 個のプロセスの故障に対しても更新を行なうことができる．

(証明) 初期更新テーブルとその更新グラフは図 4のようになる．即ち，更新グラフは N個の節点と N本の枝からなる 1 つのループである．毎回の更新は更新グラフにおいて 1 つの節点とその節点からの外向枝を取り除くことに対応する．この操作は高々$N-1$ 回しか存在しないので，自己ループが存在するのは $N-1$ 回目の更新後のみである． □

1	2	3	•••	N-1	N
2	3	4	•••	N	1

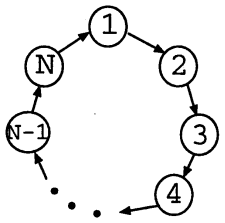

図 4 初期更新テーブルとその更新グラフ

3.3 コータリの更新

故障プロセスを示すメッセージ *Down* を受信したプロセスは，更新テーブル更新後このテーブルに従って，コータリの更新を行なう．更新前のコータリを C，故障プロセスを x とし，$y = update[x]$ であった場合の更新後のコータリ C'は，$C' = Q(T_{xy}(C))$ によって定義する．

更新後のコータリ C'に対して以下の定理が成り立つ．なお，定理の証明については，[7] を参照のこと．

定理 1 更新後のコータリ C'は，コータリの条件 *(定義 1)* を満たす． □

定理 2 $x \in G, G \in C$の時，$(G-\{x\}) \subseteq G'$なる $G' \in C'$ が存在する． □

定理 2より，コータリ更新前に故障プロセスを含むコーラム G に要求を送信していたプロセスは，コータリ更新後 $(G' - (G - \{x\}))$ 中のプロセスにのみ要求を送信し，G'中の全てのプロセスから許可を受信しているか受信すれば資源を使用できる．すなわち，コータリ更新前に送信した要求を取り消す必要はない．

例 1 コータリ更新に関する例を示す．プロセス集合を $U_7 = \{1,2,3,4,5,6,7\}$ とする．初期更新テーブルとコータリの初期値を図 *5(a)* に示す．この状態でプロセス *1* の故障を *Down* メッセージを受信したことで認識したとする．まず，アルゴリズム *update_table* により更新テーブルの更新を行なう．その更新テーブルによりプロセス *1* をプロセス *2* で置き換えることが分かる．更新後のコータリを図 *5(b)* に示す．また同様の操作により，プロセス *5* の故障を認識した場合の更新後の更新テーブルとコータリを図 *5(c)* に示す． □

1	2	3	4	5	6	7
2	3	4	5	6	7	1

```
C={{1,2,3},{2,4,6},
   {3,5,6},{1,4,5},
   {2,5,7},{1,6,7},
           {3,4,7}}
```

(a) 初期更新テーブルとコータリ

1	2	3	4	5	6	7
2	3	4	5	6	7	2

```
C={{2,3},{2,4,6},
   {3,5,6},{2,4,5},
   {2,5,7},{2,6,7},
           {3,4,7}}
```

(b) 故障プロセス1に対する更新

1	2	3	4	5	6	7
2	3	4	6	6	7	2

```
C={{2,3},{2,4,6},
   {3,6},{2,6,7},
         {3,4,7}}
```

(c) 故障プロセス5に対する更新

図 5 コータリ更新の例

4 アルゴリズムの正当性

補題 3 故障プロセス集合 $U_f \subseteq U_K$に対し，どのような順序で更新を行なっても，更新完了後の更新テーブルは同一である．

(証明) 更新グラフにおいて，各節点からの外向枝はただ 1 つなので，各節点は自分を始点としある節点を終

点とするパスを丁度1つ持つ．更新テーブルの更新を行なうことは，更新グラフにおいて故障プロセスの節点とそのプロセスからの外向枝をグラフから取り除くことである．従って，どのような順序で取り除いたとしても各節点を始点としたパスにおける節点の順序は変化しない．このことは，更新後の更新グラフが一致することを示している．よって，更新テーブルは同一である．□

次に，更新後のコータリに関する重要な補題を導く．

補題 4 いかなる順序で更新したとしても，故障プロセス集合 $U_f \subseteq U_K$ 中の全てのプロセスに対し更新を行なった後の更新後のコータリ C'''_f は，同一である．

(証明) 初期コータリを C_0，$|U_f| = f$ 個の故障プロセスに対し更新を完了した更新テーブルを $update_f$，この $update_f$ に基づき関数 T_{xy} のみを f 回適用後の集合を C_f，$C'_f = Q(C_f)$ とする．

コータリの更新は，T_{xy} と Q を毎回適用することで行なわれるが，毎回の更新時に Q を適用しなかったならば，補題3より，$|U_f|$ に対する更新完了後の集合は，C_f に一致する．ここで，$C_0 = \{q_1, q_2, \ldots, q_m\}$ とすると，$C_f = \{q'_i | q'_i = F(q_i), 1 \leq i \leq m\}$ として，1対1に対応できる．

いま，ある順序で正しく更新されたコータリを C'''_f とし，f回目の更新において関数 T_{xy} 実行後の集合を C''_f とする (即ち $C'''_f = Q(C''_f)$)．C''_f は，関数 Q を $|f-1|$ 回適用しているので，$C''_f \subseteq C_f$ が成り立つ．従って命題を証明するために，$C'''_f \neq C'_f$ なる C'''_f は存在しないことを示せばよい．

証明は，背理法を用いる．つまり，$C'''_f \neq C'_f$ なる C'''_f は存在する，とする．この場合の C'''_f と C'_f の関係について述べる．ここで，$C''_f \subseteq C_f$ なので，$Q(C''_f) \subseteq Q(C_f)$ より，$C'''_f \subseteq C'_f$ が言えるので，$C'''_f \subset C'_f$ の場合のみを考える．$q'_i \notin C'''_f, q'_i \in C'_f$ なるコーラム q'_i に注目する．C'''_f の更新過程において $h(< f)$ 回目の更新において，$q''_i \subseteq q''_j$ なる関係が成立したことにより，q''_i は削除されたとする．更新の定義より，$q'_i \subseteq q'_j$ は成り立つ．これは，$q'_i \notin C'_i$ であることを示している．よって矛盾する．

従って，$C'''_f \neq C'_f$ なる C'''_f は存在しない．これは，更新後の全てのコータリは同一であることを示している．□

定理 3 提案アルゴリズムは相互排除を満足する．

(証明) 大域時刻 t で資源の使用禁止を解除されているプロセスを P_i，P_j とする．証明は，背理法によって行なう．即ち，異なるコータリを持つ P_i，P_j が存在するとする．

プロセス P_i，P_j が，大域時刻 t までに故障を認識したプロセス集合をそれぞれ D_i，D_j とする．背理法の仮定より，$D_i \neq D_j$ が言える．ここで，$x \in D_i, x \notin D_j$ なる故障プロセス x が存在するとしても一般性を失わない．プロセス P_i は，資源の使用禁止を解除されているので x に関する *Down* メッセージを受信後少なくとも σ 時間経過している．従って，*Down* メッセージはあるプロセスから放送されているので，P_j もこの *Down* を t 以前に受けとっていたことになる．これは，矛盾する．よって，$D_i = D_j$．

従ってこの時，補題4より全てのプロセスのコータリは同一である．このことは，相互排除を満足することを示している．□

5 あとがき

任意の時刻に複数のプロセスが故障する場合の相互排除アルゴリズムを示した．アルゴリズムはコータリに基づいており，故障プロセスを検出後更新テーブルとコータリを動的に更新することで故障プロセスを排除する．今後の課題としては，メッセージ複雑度の解析，故障プロセスが復帰する場合を許した相互排除アルゴリズムの開発等が挙げられる．

文献

[1] P. A. Alsberg and J. D. Day: "A principle for resilient sharing of distributed resources," Proc. of the Second International Conference on Software Engineering, pp. 562–570 (1976).

[2] H. Garcia-Molina and D. Barbara: "How to assign votes in a distributed system," Journal of ACM, Vol. 32, No. 3, pp. 841–860 (1985).

[3] L. Lamport: "Time, clocks and ordering of events in a distributed system," Communications of the ACM, Vol. 21, No. 7, pp. 558–564 (1978).

[4] M. Maekawa: "A $\sqrt{n}$ algorithm for mutual exclusion in decentralized systems," ACM Transactions on Computer Systems, Vol. 3, No. 2, pp. 145–159 (1985).

[5] G. Ricart and A. K. Agrawal: "An optimal algorithm for mutual exclusion in computer networks," Communications of the ACM, Vol. 24, No. 1, pp. 9–17 (1981).

[6] B. Sanders: "The information structure of distributed mutual exclusion algorithms," ACM Transactions on Computer Systems, Vol. 5, No. 3, pp. 284–299 (1987).

[7] 武川, 若林, 小出："相互排除問題を解く耐故障分散アルゴリズム," 情報基礎論ワークショップ論文集, pp. 113–118 (1995).