Local Majority Coterie とそれを用いた資源配分の為の分散アルゴリズム

橋本 悟†, 和田 裕‡, 程 子学†

†: 会津大学コンピュータ理工学部 ‡: 会津大学大学院コンピュータ理工学研究科

分散システムにおける資源配分問題は, 現在まで数多くの研究がなされてきており, 様々な解法が提案されてきた. 資源配分問題の解決の為には, 排他制御の実現に加え, プロセスがお互いに他のプロセスによって既に占有されている資源を要求しあい, いずれかのプロセスも資源の利用が出来ないデッドロックの状態と, あるプロセスが永久に資源を獲得出来ない飢餓状態の双方を回避する分散アルゴリズムが必要とされる. 本論文ではよりメッセージ通信量が少なく, 耐故障性に優れた Local Majority コテリーの定義をあたえ, それを構成する方法を提案する. 次に, Local Majority コテリーを用いてデッドロックや飢餓状態を回避しながら資源配分問題を解決するアルゴリズムを提示する. 最後にそのアルゴリズムのメッセージ複雑度の考察を行う.

Local Majority Coteries based Distributed Algorithm for Resource Allocation

Satoru Hashimoto†, Yutaka Wada‡, Zixue Cheng†

†: School of Computer Science and Engineering, University of Aizu

‡: Graduate School of Computer Science and Engineering, University of Aizu

To solve the resource allocation problem in distributed system, many studies have been done so far, and various solutions for the problem have been suggested. It is necessary to guarantee the mutual exclusive use of a resource in a solution for the resource allocation problem. Furthermore, this solution should avoid both deadlock and starvation. In this paper, we define the Local Majority Coterie that is more efficient in term of message complexity and robuster to network failure, and propose a method to create a Local Majority Coterie. Then, we show an algorithm, which guarantee the mutual exclusive access to a resource, deadlock-free and starvation-free, to solve the distributed resource allocation problem, by using Local Majority Coterie. Finally we discuss the message complexity of the algorithm.

1 まえがき

分散システムにおける資源配分問題は, 現在まで数多くの研究がなされてきており, 様々な解法が提案されてきた. その概要をまとめると以下のようになる. 分散システムには, 複数の資源と複数のプロセスが存在する. 各プロセスは任意の資源の一部を要求し, 要求された資源を獲得した時にこれを利用する. 各資源は 1 つのプロセスにのみ割り当てられるため, 資源を要求するプロセスはこれを排他的に利用しあう. これを排他制御 (mutual exclusion) といい, この実現が資源配分問題の解決の為に必要とされる. その上で, プロセスがお互いの既に獲得している資源を要求しあい, いずれかのプロセスも資源の利用が出来ないデッドロック (deadlock) の状態と, あるプロセスが永久に資源を獲得出来ない飢餓状態 (starvation) の双方を回避する分散アルゴリズムが必要とされる. 資源配分のためのアルゴリズムの構成にコテリーの概念を利用する方法があり, コテリーを用いた相互排除問題の為の分散アルゴリズム [4] や, コテリーを資源配分問題に適用したローカルコテリーを用いた分散アルゴリズム [2] が提案された. 本論文では, 角川の分散アルゴリズム [2] で述べられたローカルコテリーの構成法よりもクォーラムサイズについてより優れる, Local Majority コテリーの構成法を提案する. そして, Local Majority コテリーを用いてデッドロックや飢餓状態を回避しながら資源配分問題を解決するアルゴリズムを提示し, 最後に Local Majority コテリーを用いたアルゴリズムの通信量についての考察を行う.

2 コテリーとローカルコテリー

2.1 分散システムの資源配分モデル

分散システムの資源配分モデルは, 二部グラフ $G(V,E)$ で表される. ここで V は, 分散システムのプロセスの集合 P と資源の集合 R からなる集合 $V = P \cup R$ である. また E はプロセスと資源についての隣接関係を示す辺の集合を示し,

辺で結ばれた隣接する資源についてプロセスは使用を要求する．このときあるプロセス $p \in P$ について，p が使用を要求する資源の集合を $\mathcal{A}_p$ で表す．各プロセスはそれぞれが局所的な時刻系をもつため，プロセス間の時間の整合性は論理時計の概念 [3] を導入して解決する．あるプロセスが資源の使用を要求する場合，それぞれの排他制御を行うプロセスにおいて合意が得られたときにはじめて資源の使用が許可されるものとする．資源の使用を許可されたプロセスは資源を使用するが，それは有限時間内に終わるものとする．また，制御ノードが同時に複数の要求ノードに対して資源の使用を許可することはない．分散システムにおいてプロセスは，資源の使用を要求する要求ノードとしての役割を果たすだけでなく，資源配分の為の排他制御を行う制御ノードとしての役割も果たす．各制御ノードは自らの保持する局所的な資源配分の情報のみを元に排他制御に関する決定を下す．排他制御を行うためのメッセージは直接他のプロセスに送られるものとし，そのメッセージの転送時間は不定であるが，有限時間内に必ず届くものとする．

2.2 コテリー

コテリーを用いた分散アルゴリズム [4] は，排他制御を行う制御ノードの組合わせをうまく選ぶことによりメッセージ通信量の削減を図ったものである．
集合を要素とする集合，つまり集合族 $\mathbf{C}$ について，その任意の集合要素 $Q \in \mathbf{C}$ が以下の条件をみたすとき，$\mathbf{C}$ をコテリー (coterie)，Q をクォーラム (quorum) と呼ぶ．[1, 2]

- 空集合であるクォーラムは存在しない
  $(^\forall Q \in \mathbf{C}, Q \neq \emptyset)$
- クォーラム同士は必ず一つ以上の共通要素を持つ
  $(^\forall Q_i, Q_j \in \mathbf{C}, Q_i \cap Q_j \neq \emptyset)$
- あるクォーラムの部分集合であるクォーラムは存在しない
  $(^\forall Q_i, Q_j \in \mathbf{C}, Q_i \not\subseteq Q_j)$

コテリーを用いた分散アルゴリズムでは，クォーラム Q は任意の要求ノード (プロセス) に対してその制御ノード集合になる．
このとき，Q を $\mathbf{C}$ に含まれる全てのプロセスの過半数集合 (Majority) とするとクォーラムを容易に求めることが出来る．コテリー $\mathbf{C}$ に含まれる全要素数を $N_\mathbf{C}$ とすると，その過半数集合の要素数 maj は $maj = \lfloor N_\mathbf{C}/2 \rfloor + 1$ で表される．Majority コテリーにおいて，排他制御の為に問い合わせる制御ノードの数はすべてのノードが制御ノードとなる場合よりも少なくなるため，排他制御を行うためのメッセージ量は削減される．また，すべての制御ノード集合は互いに共通する制御ノードをもつので，いずれの制御ノード集合に資源の問い合わせをしても，それぞれの集合が資源の使用を許可する要求ノードは複数にはならない．

2.3 ローカルコテリー

ローカルコテリーを用いた分散アルゴリズム [2] は，相互排除問題に適用したコテリーを資源配分問題に適用させたものである．ローカルコテリーは各プロセスごとに固有のコテリーを構成するが，そのクォーラムは注目したプロセスが要求する資源について、それを共有する全てのプロセスの集合から求められる．これにより，制御ノード集合は注目したプロセスと資源を共有するプロセスのみで構成されるので，資源を共有しないプロセス同士が互いの排他制御に干渉することはない．ローカルコテリーは以下の様な条件を満たす．[2]

- 全てのローカルコテリーは空集合ではない
  $(^\forall p \in P, \mathbf{C}_p \neq \emptyset)$
- 資源を競合するプロセスのクォーラム同士は必ず 1 つ以上の共通要素を持つ
  $(^\forall p_i, p_j \in P, \mathcal{A}_{p_i} \cap \mathcal{A}_{p_j} \neq \emptyset \Rightarrow$
  $^\forall Q_i \in \mathbf{C}_{p_i}, ^\forall Q_j \in \mathbf{C}_{p_j}, Q_i \cap Q_j \neq \emptyset)$
- あるローカルコテリーのクォーラムについて、部分集合であるようなクォーラムが存在しない
  $(^\forall p \in P, ^\forall Q_i, Q_j \in \mathbf{C}_p, Q_i \neq Q_j \Rightarrow$
  $Q_i \not\subseteq Q_j)$

前述の分散アルゴリズム [2] で提案されたローカルコテリーの構成法では，ローカルコテリーは 1 つのクォーラムのみからなる $(^\forall p \in P, |\mathbf{C}_p| = 1)$．この構成法で求められたローカルコテリーは，資源を共有しないプロセス同士が互いの排他制御に干渉することはない．しかし，各プロセスのローカルコテリーに属するクォーラムが唯一つであるため，耐故障性については考慮されてはおらず，またそのクォーラムは競合する全てのプロセスからなるため，排他制御のための効率についても考慮されていない．

3 LM(Local Majority) コテリー

3.1 LM コテリーの概要

本論文で提案する LM(Local Majority) コテリーは，角川の分散アルゴリズム [2] で述べられたローカルコテリーの構成法を改善するために，Majority コテリーと同様複数のクォーラムでローカルコテリーを構成するものである．

3.2 LM コテリーの構成

LM コテリーは，次の様な流れで構成される．分散システムの各資源 $r \in R$ について r を共有するプロセスの集合 $\mathcal{S}_r$ を構成し，それを用いて Majority コテリー $\mathbf{M}_r$ をつくる．全てのプロセス $p \in P$ について，以下の事を同様に行う．p が利用する各資源 $r \in \mathcal{A}_p$ について $\mathbf{M}_r$ を選ぶ．前述の $\mathbf{M}_r$ 全てに対し，各 $\mathbf{M}_r$ に属するクォーラムを一つずつ選び，それらの論理和をとる．このときこの論理和をクォーラム候補とする．このように全てのクォーラムの組み合わせに対し同様に論理和をとり，その集合を $\mathbf{T}_p$ とする．最後に $\mathbf{T}_p$ から，全てのクォーラム候補について，その部分集合であるクォーラム候補が存在するようなクォーラム候補を除外したものが求める LM コテリー $\mathbf{C}_p$ である．以下はその詳細である．

when making LM coteries
begin
  for *each* $r \in R$ **do**
    begin
      $\mathcal{S}_r := \{p \in P \mid r \in \mathcal{A}_p\}$;
      $maj := \lfloor |\mathcal{S}_r|/2 \rfloor + 1$;
      $\mathbf{M}_r := \{m_i \mid {}^{\forall} m_i, m_j \subseteq \mathcal{S}_r, |m_i| = |m_j| = maj, m_i \cap m_j \neq \emptyset, m_i \not\subseteq m_j\}$;
    end
  for *each* $p \in P$ **do**
    begin
      $\mathbf{T}_p := \{Q_p \mid Q_p = \bigcup_{\forall r \in \mathcal{A}_p, {}^{\exists} m \in \mathbf{M}_r} m\}$;
      $\mathbf{C}_p := \{Q_{pi} \mid {}^{\forall} Q_{pi}, Q_{pj} \in \mathbf{T}_p, Q_{pj} \not\subseteq Q_{pi}\}$;
    end
end.

P: set of processes
R: set of resources
$\mathcal{A}_p$: $\mathcal{A}_p = \{r \mid p \text{ requires } r\} \subseteq R$

このようにして求められた LM コテリーは第 2.3 節で示されたローカルコテリーの条件を満たす．

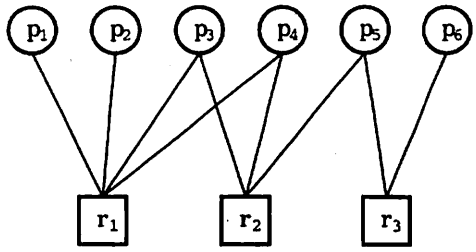

図 1: 分散システムの例

3.3 LM コテリーの例

図 1 の様な分散システムを考える．この分散システムはプロセス $P = \{p_1, p_2, p_3, p_4, p_5, p_6\}$ 及び資源 $R = \{r_1, r_2, r_3\}$ から構成され，プロセス $p \in P$ の使用する資源 $\mathcal{A}_p$ は以下の通りに定められる．

- $\mathcal{A}_{p_1} = \mathcal{A}_{p_2} = \{r_1\}$
- $\mathcal{A}_{p_3} = \mathcal{A}_{p_4} = \{r_1, r_2\}$
- $\mathcal{A}_{p_5} = \{r_2, r_3\}$
- $\mathcal{A}_{p_6} = \{r_3\}$

この分散システムの P の全てのプロセス p について LM コテリー $\mathbf{C}_p$ を求めると，

- $\mathbf{C}_{p_1} = \mathbf{C}_{p_2} = \{\{p_1, p_2, p_3\}, \{p_1, p_2, p_4\}, \{p_1, p_3, p_4\}, \{p_2, p_3, p_4\}\}$
- $\mathbf{C}_{p_3} = \mathbf{C}_{p_4} = \{\{p_1, p_3, p_4\}, \{p_2, p_3, p_4\}, \{p_1, p_2, p_3, p_5\}, \{p_1, p_2, p_4, p_5\}\}$
- $\mathbf{C}_{p_5} = \{\{p_3, p_5, p_6\}, \{p_4, p_5, p_6\}\}$
- $\mathbf{C}_{p_6} = \{\{p_5, p_6\}\}$

となる．ここで p_2 , p_6 の LM コテリー $\mathbf{C}_{p_2}$, $\mathbf{C}_{p_6}$ から分かるとおり，資源を共有しないプロセス同士は互いの資源配分のための排他制御に干渉する事はない．また，LM コテリー $\mathbf{C}_{p_3}$, $\mathbf{C}_{p_4}$ から分かるとおり LM コテリーのクォーラムについて，その要素数は一定でない場合がある．本論文では，ある LM コテリー $\mathbf{C}$ における要素数が最大であるクォーラムを Q_{max}, 最小であるクォーラムを Q_{min} とする．

4 LM コテリーを用いた分散アルゴリズムの動作

4.1 アルゴリズムの概要

本論文の目的は，同一分散アルゴリズムについて LM コテリーを用いた場合の優位性の主張である．したがって角川のアルゴリズム [2] よりも

簡略化した単純な排他制御アルゴリズムを用いて，比較を行う．アルゴリズムを具体的に説明すると以下のようになる．
自らに隣接する資源について，任意の複数資源を獲得することが出来る角川のアルゴリズム [2] とは異なり，要求ノードが資源の使用を要求する場合，自らに隣接する全ての資源についてこれを要求する．

4.2 アルゴリズムの詳細

以下はアルゴリズムの詳細である．
各プロセスは，次の内部変数をもつ．

- $time_p$—プロセス p の論理時刻．
- $queue_p$—プロセス p に送られてきた *inqury* メッセージのタイムスタンプが記憶される配列．*reply* メッセージを既に送ったプロセスのタイムスタンプは配列から取り除かれる．
- $permit_p$—*permission* メッセージを送ったプロセスについてタイムスタンプを記憶する．*permission* メッセージを送ったプロセスから *release* または *dispose* メッセージが返されたならばその内容は破棄される．

次にアルゴリズムの動作について述べる．

1. **プロセス p_i が資源の使用を要求するとき**
   任意のクォーラム $Q \in \mathbf{C}_{p_i}$ を選び，資源の使用を終えるまでこのクォーラムのプロセスを要求ノード p_i の制御ノードとする．Q に属する全ての制御ノードに対して $inquiry(p_i, time_{p_i})$ メッセージを送り，その各々からの *permission* メッセージを待つ ($(p_i, time_{p_i})$ は p_i が *inquiry* メッセージを発行した時点での論理時刻 $time_{p_i}$ を含んだタイムスタンプである.) 要求ノード p_i は全ての制御ノードから *permission* メッセージを返されたときに初めて資源の使用を開始する．
2. **プロセス p_i が資源の使用を終えたとき**
   資源の使用を終えたならば，Q に属する全ての制御ノードに対して $release(p_i, time_{p_i})$ メッセージを送り，資源を解放した旨を通知する．
3. **プロセス p_i が プロセス p_j から *inquiry* $(p_j, time_{p_j})$ メッセージを受け取ったとき**
   プロセス p_j からの *inquiry* メッセージを受け取った時点で，まだ他の要求ノードに対し *permission* メッセージを送っていない場合 ($permit_{p_i}$ にタイムスタンプが格納されていない場合)，プロセス p_j に対し $permission(p_i)$ メッセージを送り，$permit_{p_i}$ に $(p_j, time_{p_j})$ を格納する．既にプロセス p_k に *permission* メッセージを送っていた場合は，p_j のタイムスタンプ $(p_j, time_{p_j})$ を待ち行列 $queue_{p_i}$ に入れる．次にプロセス p_j とプロセス p_k のタイムスタンプを比較し，プロセス p_k の優先度が低ければ p_k に対し *cancel* メッセージを送り，p_k からの *dispose* もしくは *release* メッセージを待つ．
4. **プロセス p_i がプロセス p_j から *permission* (p_j) メッセージを受け取ったとき**
   全ての制御ノードから *permission* メッセージを受け取ったならば，資源を使用する．そうでなければ，全ての制御ノードから *permission* メッセージが送られてくるのを待つ．
5. **プロセス p_i がプロセス p_j から $release(p_j, time_{p_j})$ メッセージを受け取ったとき**
   $permit_{p_i}$ に格納されているタイムスタンプを破棄し，待ち状態を解除する． もし $queue_{p_i}$ に プロセス p_i からの *permission* メッセージを待つ要求ノードのタイムスタンプが存在すれば，そのもっとも優先度の高い要求ノード p_k に対して $permission(p_i)$ メッセージを送り，$permit_{p_i}$ に $(p_k, time_{p_k})$ を格納する．
6. **プロセス p_i がプロセス p_j から $dispose(p_j)$ メッセージを受け取ったとき**
   $permit_{p_i}$ に格納されているタイムスタンプを待ち行列 $queue_{p_i}$ にいれる．続いて 5. と同じ手順を踏む．
7. **プロセス p_i がプロセス p_j から $cancel(p_j)$ メッセージを受け取ったとき**
   もし既に資源の使用を開始しているならばプロセス p_j からの *cancel* メッセージを無視する．そうでなければ $dispose(p_i)$ メッセージをプロセス p_j に送る．

5 アルゴリズムの解析

5.1 正当性の証明

本論文のアルゴリズムは，デッドロックと飢餓状態を回避しながら排他制御を行うアルゴリズム，つまり以下の全ての条件を満たすアルゴリズムであるため正当である．

5.1.1 排他制御の保証

[命題] 制御ノードは要求ノードに対して排他的に資源使用の許可を与えるため, 排他制御は保証される.

(証明) 第 4.2 節から, ある制御ノードに対し一つ以上の *inquiry* メッセージが送られた場合, 制御ノードは最小のタイムスタンプを持つ *inquiry* メッセージを発行した要求ノードに対して *permission* メッセージを送る. また, その要求ノードから *release*, *dispose* メッセージが送られない限りは他の要求ノードに対して *permission* メッセージを送ることはない. つまり資源の使用権を与えた要求ノードが, 使用権を (一時的にでも) 手放さない限りは, 資源の使用を要求する他の要求ノードに対して使用権を与える事はない. この時第 2.3 節及び第 3.1 節の条件より, 資源使用の競合を起こしうる要求ノード同士について, その制御ノード集合であるクォーラムは必ず一つ以上の共通要素をもつ. この条件及び論理時刻 [3] から, ある制御ノードに送られた競合する *inquiry* メッセージは一意的に優先度が決定される. したがって排他制御に参加する全ての *inquiry* メッセージについてその全順序が決定され, 制御ノードは各要求ノードに対してその優先度の高い順に排他的に資源の使用権を与える. よって排他制御は保証される.

5.1.2 デッドロックの回避

[命題] 最も優先度の高い要求ノードが全ての資源使用の許可を得るためデッドロックは回避される.

(証明) ある要求ノードに対して *permission* メッセージを送り資源の使用権を与えた制御ノードに, 優先度の高い他の要求ノードから *inquiry* メッセージが到着すると, 既に資源の使用権を与えた要求ノードに対して *cancel* メッセージを送る. *cancel* メッセージを受け取った要求ノードは, 資源を使用していない場合 *dispose* メッセージを送り資源の使用権を一時放棄する. *dispose* メッセージを受け取った制御ノードは, *permission* メッセージを最も優先度の高い要求ノードに送り, この要求ノードが資源の使用権を獲得する. ここで全順序関係 (第 5.1.1 節参照) より, 最小のタイムスタンプを持つ (最も優先度の高い) *inquiry* メッセージを発行した要求ノードのみが全ての制御ノードから *permission* メッセージを受け取る. したがって資源の競合が発生している状態において, 優先度のもっとも高い要求ノードが全ての資源について使用権を獲得する. したがってデッドロックは回避される.

5.1.3 飢餓状態の回避

[命題] 任意の要求ノードについて, 有限時間内に必ず資源の使用が許可されるため飢餓状態は回避される.

(証明) アルゴリズムにおいて, *inquiry* メッセージを発行した要求ノードは全順序関係 (第 5.1.1 節参照) にしたがい, そのタイムスタンプが小さい順に制御ノードから *permission* メッセージを受け取る. 第 2.1 節より資源の使用は有限時間内に終了し, その後資源を使用していたプロセスは直ちに *release* メッセージを送り資源を解放する. タイムスタンプが最小のプロセスが資源を使用し, その後再び資源の使用を要求する場合を考える. この時発行された新たな *inquiry* メッセージのタイムスタンプは, 以前に他のプロセスに対し発行されたどの *inquiry* メッセージよりも新しい (大きい) タイムスタンプを有する. したがって全ての要求ノードについて有限時間内に必ず *permission* メッセージが返される順番が回ってくる. よって, 有限時間内に必ず資源の使用権が得られるので飢餓状態の発生は有り得ない.

5.2 通信量の考察

以下に本論文で示したアルゴリズムの通信量について, 最良のケースと最悪のケースを提示する.

最良のケースは, 要求ノード p が LM コテリーの最小クォーラム Q_{min} を制御ノード集合として選択し, その各制御ノードで競合を起こさずに全ての資源を獲得した場合である. このとき要求ノード p と Q_{min} の各制御ノードの間で *inquiry*, *permission*, *release* の 3 つのメッセージが一度ずつやり取りされる. したがってそのメッセージ通信量は $3|Q_{min}|$ で示される.

最悪のケースは以下のような状況に陥った時である. ある要求ノード $p \in P$ について, p と資源を共有し合うプロセス (要求ノード) の集合を $\mathcal{P}_p$ とする. この時 $\mathcal{P}_p$ は p 自身を含む ($p \in \mathcal{P}$).

また全ての要求ノードは同一のクォーラム Q_{max} (LM コテリーの最大クォーラム) を制御ノード集合として選択し, *inquiry* メッセージを送る. 各要求ノードが発行した *inquiry* メッセージは優先度の低い順に制御ノードに到着する. ある要求ノード $p_i \in \mathcal{P}$ に対して既に *inquiry* メッセージを送った制御ノードに, より優先度の高い要求ノード $p_j \in \mathcal{P}$ からの *inquiry* メッセージが到着した場合, 制御ノードはアルゴリズムに従って先に資源の使用権を与えた要求ノード p_i に対して *cancel* メッセージを送り, 資源の使用権を一時取り上げる. このとき *cancel* メッセージが前述の要求ノード p_i に届くタイミングは, p_i が資源の使用を開始する前である. 以上の様な状況において, $\mathcal{P}$ に属する全てのプロセスが制御ノードに対して *inquiry* メッセージを送るとする. このような状況下でアルゴリズムに従い排他制御についてのメッセージ通信を行うと, 最終的に最も優先度が低いプロセスが資源の使用を終えるまでに送り合うメッセージの総量は $(3+6(|\mathcal{P}|-1))|Q_{max}|$ になる.

LM コテリーはローカルコテリーの構成法の一つであるため, 排他制御のためのアルゴリズムは同一のものを適用することができる. しかし図 2 の様な極端な例を除けば, 同一の分散システムにおいて LM コテリーのクォーラム (制御ノード集合) は, 明らかに角川のアルゴリズム [2] で示された構成法のそれよりも小さくなり, 結果的に排他制御のためのメッセージ量は少なくなると予想される. 加えて LM コテリーは複数のクォーラムからなるため, プロセスの停止故障に対する耐性の向上も予想される.
以上の様なことは本論文で示したアルゴリズムに LM コテリーを適用した場合だけでなく, 角川のアルゴリズム [2] に LM コテリーを適用した場合にも同様のことが言える.

6 むすび

本論文では, ローカルコテリーの構成法の一つとして Majority コテリーの考えを導入した LM コテリーを提案し, それを用いた分散アルゴリズムが正しく動作することを証明した. 加えて前述のアルゴリズムにおいて, ローカルコテリーの構成法を用いた場合との比較を行い, その優位性を示した. LM コテリーを導入することにより, ローカルコテリーの条件を満たしながら制御ノードの総数を減らす事ができ, これにより排他制御のためのメッセージ量が削減できる. また, 各コテリーに属するクォーラムの数が増えることにより耐故障性が高くなる. 今後の課題としては, LM コテリーのクォーラムサイズについての厳密な定量的解析や Availability の計算等が挙げられる.

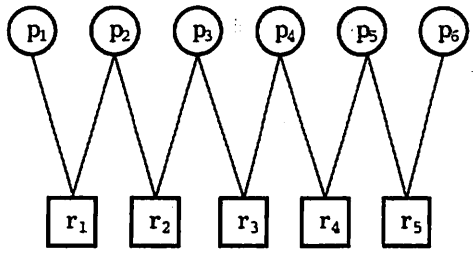

図 2: 分散システムの例 2

謝辞

本研究に関し多くの有益な御討論, 御助言をして頂いている会津大学コンピュータネットワーク学講座の諸氏に感謝致します.

参考文献

[1] Garcia-Molina, H. and Barbara, D.: How to Assign Votes in a Distributed System, *Journal of the ACM*, Vol.32, No.4, pp.841-860 (1985).

[2] Kakugawa, H. and Yamashita, M.: Local Coteries and a Distributed Resource Allocation Algorithm, *Transactions of Information Processing Society of Japan*, Vol.37, No.8, pp.1487-1496 (1996).

[3] Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System, *Communications of the ACM*, Vol.21, No.7, pp.558-565 (1978).

[4] Maekawa, M.: A $\sqrt{N}$ Algorithm for Mutual Exclusion in Decentralized Systems, *ACM Transactions on Computer Systems*, Vol.3, No.2, pp.145-159 (1985).