コーラムコンセンサスを用いた分散相互排除命令ライブラリの設計と評価

丸山英明[†] 伊東靖英[†] 藤田聡[†]

共有資源に対するプロセス間の相互排除問題は，分散システムにおける重要な問題の 1 つである．本稿では，コーラムコンセンサスと呼ばれる分散相互排除アルゴリズムに着目し，このアルゴリズムを実際の分散ネットワーク環境下で実現することを目的として試作したクラスライブラリについて報告する．ライブラリは Java で記述され，相互排除を行う機能は Java のクラスが持つメソッドとして実装される．このライブラリでは，プロセスの追加や削除といったシステムの動的な変化が起こるような状況においても継続的に相互排除が行えるようにするための機能が提供されている．

A Design and Evaluation of Distributed Mutual Exclusion Libraries Based on Quorum Consensus Protocol

Hideaki Maruyama,[†] Yasuhide Ito[†] and Satoshi Fujita[†]

In distributed computer systems, several processes are frequently required to achieve a mutual exclusion for accessing shared resources. This paper reports the detail of a class library, that was designed and implemented for achieving a mutual exclusion in a distributed manner based on the quorum consensus protocol. The library is written in Java language, and the function of mutual exclusion is implemented as Java classes. As a remarkable feature, it supports a continuous execution of mutual exclusions even if it occurs a dynamic change of the process state, such as creation, deletion, and permanent failures of the processes.

1. はじめに

分散システム上にはさまざまな資源が存在する．それらの資源の多くは，ネットワーク接続された計算機上で動作する複数のプロセスによって共有され，各プロセスの自律的な動作によってさまざまな処理が分散的に進められる☆．ネットワーク上に存在する共有資源に対するアクセスの相互排除（mutual exclusion）をできるだけ低いコストで効率良く実現する問題は，分散システムにおける基本的な問題の 1 つであり，これまでに数多くの解法が提案されてきている[1),4),7)]．本稿ではこの問題を分散的に解くことを考える．ここで分散的に解くとは，情報を集中管理するための特定のサーバを置かず，プロセス間のメッセージ交換のみによって相互排除を実現することを意味している．

相互排除問題を分散的に解くためのアルゴリズムは，大きく分けてトークン（token）方式と許可（permission）方式とに分類される[6)]．トークン方式は，共有資源に対するアクセス権をトークンという形でネットワーク上に巡回させる方法であり，各時点でトークンを所有しているプロセスに対して，プログラム中の臨界領域（critical section: CS）を実行する権限が与えられる．一方，許可方式は，共有資源を使用するために必要な許可の断片をあらかじめネットワーク上に分散しておき，その断片の中のいくつかを集めたプロセスに対して CS を実行する権限が与えられるというものである[1)]．なお，この方式によって相互排除が保証されるためには，許可の収集方式がコーラムシステムと呼ばれる数学的な構造に従っていることが必要であることが知られている[2)]（コーラムシステムの定義については後述する）．

コーラムシステムに基づく（許可方式の）分散相互排除アルゴリズムは，一般にコーラムコンセンサスと呼ばれる．これまでに，コーラムシステムの具体的な構成法[5)]や数学的な性質の解明[9)]などに関する研究が

† 広島大学大学院工学研究科
Graduate School of Information Engineering, Hiroshima University

☆ 分散システム上の共有資源の例としては，プリンタやディスクのような物理的なもののほかに，ネットワーク上に分散配置される計算サーバ群に対するアクセス権や，分散化されたデータベース上の顧客情報[8)]などがある．

数多くなされている．また，交換されるメッセージ数や待ち時間の意味で良好な性質を示すプロトコルの提案もいくつか行われている．本稿の目的は，コーラムコンセンサスが実際の広域分散ネットワーク上で有効に機能する分散相互排除アルゴリズムであるという認識にたち，このアルゴリズムを現実のネットワーク上に実装する際に起こるであろう課題とそれに対する対応方法とを，具体的な試作を通してあきらかにすることである．具体的には，システムの状態が動的に変化するような状況のもとでも相互排除を継続的に実行するためのメカニズムを提案し，そのアイデアを Java のクラスライブラリとして実装する．クラスライブラリの性能は，実環境下での実験により評価される．

本稿の構成は以下のとおりである．まず 2 章ではコーラムコンセンサスの概要を説明する．3 章では，今回実装したクラスライブラリの詳細について述べる．4 章では，プロセスの動的な状態変化に対する対応方法について説明する．試作したライブラリの性能は，5 章で実験的に確認される．最後に 6 章では今後の課題について述べる．

2. 準　　備

2.1 諸 定 義

n 個のプロセスの集合 $P = \{p_1, p_2, \ldots, p_n\}$ を考える．P のべき集合 2^P の任意の部分集合 $\mathcal{C} = \{Q_1, Q_2, \ldots, Q_m\}$ は，以下の条件を満たすときコーラムシステム（quorum system）と呼ばれる（$\mathcal{C}$ の各要素はコーラムと呼ばれる）：任意の 2 要素 $Q_i, Q_j \in \mathcal{C}$ に対し，

$$Q_i \cap Q_j \neq \emptyset.$$

さらに，以下の条件を満たすコーラムシステムを**コータリ**（coterie）と呼ぶ：任意の 2 要素 $Q_i, Q_j \in \mathcal{C}$ に対し，

$$Q_i \not\subset Q_j.$$

ここで P 上に定義されるコータリは，一般に 1 つとは限らないことに注意されたい．たとえば，$P = \{p_1, p_2, p_3\}$ のとき

$\mathcal{C}_1 = \{\{p_1, p_2\}\}$

$\mathcal{C}_2 = \{\{p_1, p_2\}, \{p_2, p_3\}, \{p_3, p_1\}\}$

はいずれもコータリである．$\mathcal{C}_1$ は単一の要素しか持たないためシングルトン（singleton）コータリと呼ばれ，$\mathcal{C}_2$ は P 上で過半数の要素を持つ部分集合からなることから多数決（majority）コータリと呼ばれる．本稿では議論を簡単にするため，多数決コータリのみを取り扱う（一般的なコータリの構成法については文献 5)，9) を参照されたい）．

2.2 基本プロトコル

コータリの条件から，共有資源に対する相互排除問題を，以下のようなプロトコルを用いて分散的に解くことができることが分かる（以下，この解法のことをコーラムコンセンサス（Quorum Consensus: QC）と呼ぶ）：まずはじめに，P 上に定義されるコータリ $\mathcal{C}$ を任意に 1 つ固定し，P 上の各プロセスに対してトークンを 1 つずつ割り当てておく．ここでいうトークンとは，各プロセスがプログラム中の臨界領域（CS）に入るための権限を複数のプロセスに分割して割り当てたものであり，先にあげたトークン方式のトークンとは異なることに注意されたい．以下では，CS を実行する各プロセスをクライアントと呼び，P の各要素をサーバと呼ぶ（クライアントは P の要素であってもよいし，なくてもよい）．相互排除を行うためのプロトコル QC は以下のように記述される：

protocol QC

(1) クライアント p は $\mathcal{C}$ から任意のコーラム Q を選択し，Q に属するすべてのサーバ q に対して q が初期状態で所持していたトークンを p に渡すように要求する．

(2) 要求を受けた各サーバ q は，もしトークンを所持していればそれを p に渡し，そうでなければ以下を実行する：q が最後にトークンを渡したクライアントを r とする．q はまず p と r の優先度を比較する☆．1) もし p の方が高い優先度を持てば，q は r にトークンを返却するように要求し，返却されたトークンを p に渡す．このとき r の要求は q の保持する待ち行列（キュー）に入れ直される☆☆．2) もし r の方が高い優先度を持てば，p の要求はすぐには受け付けられず，q の保持するキューに入れられる．

(3) クライアント p は自分が要求を出したすべてのサーバからトークンを受け取れば CS に入る．

(4) クライアント p が CS から出るときは，集めた全トークンを持ち主に返却する．トークンを返却された各サーバ q は，もしキューの中にトークン待ちの要求があれば取り出してトークンを渡す．

このプロトコルを用いることで，プロセス間のメッセージの送受信と各サーバ内の局所的な判定のみに

☆ 優先度については後述する．

☆☆ r がすでに CS に入っているときは，r の CS の実行が終了するまでは r からのトークンは返却されない．その場合は，r が再びキューの中に入れ直されることはない．

よって，分散的に相互排除が実現されることになる．

2.3 各要求の優先度

QC プロトコルでは，複数のトークン要求が同時に発生した場合などのために，要求を蓄えておくためのキューが各サーバ上に用意されている．ただしここでは通信路に関して何も仮定をおいていないため，単純な FIFO 制御ではデッドロックや飢餓状態を回避することはできない．したがって，要求に対して優先度をつけ，それに従って適切に処理を進めることが必要となる．

本稿では文献 1) の方法に従った実装を行う．すなわち，要求の優先度は要求を出したクライアントに対してつけられ，その優先度は，クライアントがそれまでに CS に入った回数とクライアントのプロセス ID とを元に決定される．具体的には優先度は，CS に入った回数が少ないほど高く，回数が同じ場合はプロセス ID が若いほど高いものとする．

このような優先度を用いることで，任意の要求間に順序関係が与えられ，トークン待ちのループによって引き起こされるデッドロックを防ぐことができる．また要求を出しているにもかかわらず CS に入ることができないクライアントは，いずれ他のクライアントよりも（相対的に）高い優先度を持つようになることから，特定のクライアントが飢餓状態に陥ることも防ぐことができる[1)]．

2.4 状態の動的な変化に関する仮定

次章以降で説明するプロトコルの実装では，サーバやクライアントの状態が動的に変化することを想定している．具体的には，システムの状態の変化に関して以下のことを仮定する．

- システム内には有限個のホストが存在し，各ホスト上には有限個のサーバとクライアントが存在しうる．
- 初期状態では存在していなかったサーバやクライアントが，任意の時点で QC プロトコルに新しく参加することができる．ただし各クライアントは参加するときに，自分が必要とする資源の名前と，QC に参加しているホスト集合の中の，少なくとも 1 つのホストを知っているものとする（これについて 4.2.2 項で具体的な参加の方法を示す）．
- 任意のサーバと任意のクライアントは QC プロトコル実行中の任意の時点で故障しうる．また，故障したサーバやクライアントが復旧した場合は，それまでとは異なる ID を持つ新しいプロセスとして改めて QC に参加することができる．

以下ではまず次章で基本プロトコルの実装の概要について説明する．その後，上で述べたような状態の動的な変化に適応させるための実装上の留意点について述べる．

3. 基本プロトコルの実装

プロトコルの実装は Java を用いて行った．これは，分散ネットワーク環境では異なるプラットホームを持つマシンが接続されている場合が多く，Java の持つ特質の 1 つであるマルチプラットホーム性が有効に機能すると予想されるためである．

3.1 クラス構成

まず今回実装したクラスについて説明する．

`QCClient` は QC におけるクライアントに対応するクラスであり，サーバとの通信を行うためのインタフェース（後述）が実装されている．ユーザは `enter_CS` と `quit_CS` という 2 つのメソッドを呼び出すことで共有資源に対する相互排除を行う（この 2 つのメソッドではさまれた部分がプログラム上の臨界領域（CS）になる）．`enter_CS` メソッドは CS に入るときに呼び出され，サーバに要求を出して必要なトークンを収集する．呼び出しプロセスの実行は，実際に CS に入れるようになるか，あるいはサーバ故障などの原因によって CS に入れないことが分かるまでブロックされる．もしメソッドが正常に終了すれば，メソッドからの処理が復帰した時点で自分が CS に入っていることが保証される．サーバの故障などの理由により正常に終了しない場合の対応方法については 4.1 節で述べる．一方 `quit_CS` メソッドはプロセスが CS から出るときに呼び出され，対応する `enter_CS` 内で集められたトークンをそれぞれの持ち主に返却する．なおクライアントのインスタンス生成時には，臨界領域で使用される資源の指定を行うことができる．これにより，複数の共有資源が存在する場合に対しても，複数の `QCClient` インスタンスを持つことで対応することができる．

`QCServer` は QC におけるサーバに対応するクラスであり，`QCClient` と同様クライアントとの通信に用いるインタフェースが実装されている．ただしこのクラス自体にはユーザから直接呼び出されるメソッドはなく，クライアントに対してトークンを提供する機能のみが与えられている．また `QCClient` と同様，インスタンスを生成する際に特定の資源を指定することで，そのサーバの提供するトークンと資源とを結び付けることができる☆．なお各サーバは，自分と同じ資

☆ なお `QCClient`，`QCServer` ともに，指定された資源以外のクライアントおよびサーバのインスタンスの動作には影響されない．

源を管理する（すなわち同じ資源に対するトークンを持つ）他のすべてのサーバのリスト（後述）を持っており，このリストを参照することで，どのホスト上にどのサーバが存在するかを知ることができる．サーバの追加・削除は，すべてのサーバのリストに対して反映される．リスト間の一貫性の保持方法については後述する．

Messenger は各種のメッセージを送信する機能を提供するクラスである．Messenger クラスには同期メッセージ送信用の call メソッドと非同期メッセージ送信用の send メソッドとが実装されている．ここで同期メッセージ送信では，送信側の処理は受信側から返事があるまでブロックされるが，非同期メッセージ送信は別スレッドで処理されるため，送信側の処理はブロックされないことに注意されたい．本実装では，非同期メッセージを送る場合には，だれがそのメッセージを送信したのかを受信するプロセスが認識できるように，送信するプロセスが自分のプロセス ID をメッセージに付加しておくことにする．

最後に Checker は，プロセスの故障に対応するための各種メソッドを提供する．プロセス故障時の具体的な処理については 4.1 節で述べる．

3.2 プロセス間通信

Messenger クラスによって提供されるサーバ・クライアント間の通信は，具体的には，送信プロセスが受信プロセスのメッセージハンドラメソッドを呼び出すことによって実現される．ただし分散環境においては一般に，クライアントとサーバが異なるホスト上に存在するため，リモートメッセージハンドラの呼び出しには Java の RMI（Remote Method Invocation）が用いられている（すなわち各メッセージハンドラは，RMI リモートインタフェースとして定義される）．

プロトコルで取り扱われるメッセージは，トークン操作のためのものとコータリ情報更新のためのものとにわけられる．メッセージの種類と対応するメッセージハンドラ名，およびその意味を**表 1** にまとめる．

トークン操作のための 4 種類のメッセージについてみると，これらのうち RET_TKN_REQ のみが同期型であり，残りはすべて非同期型であることに注意されたい．トークン操作メッセージによって呼ばれるメソッドには，パラメータとしてそのメッセージを送信するプロセスの情報（プロセス ID，存在するホスト名，優先順位）が収められた文字列が渡される．また，非同期型である returnTokenRequest だけがトークンが返却されたかどうかを表す返り値を持つ．他の 3 種類のメソッドには返り値はない．

表 1 メッセージの意味と対応するハンドラ
Table 1 Messages and the associated handlers.

クライアント上でハンドラが実装されているメッセージ
GET_TKN
void getToken(String server)
クライアントからの要求に対してサーバから送られるトークンを表す．
RET_TKN_REQ
boolean returnTokenRequest(String server)
優先度の高い要求からの割り込みによって，一度渡したトークンの返却を求める．
サーバ上でハンドラが実装されているメッセージ
GET_TKN_REQ
void getTokenRequest(String client)
トークン要求時にサーバに送られる．
RET_TKN
void returnToken(String client)
優先度の高い要求からの割り込みによって，クライアントから返却されるトークン．
GET_COTERIE_INFO
CoterieInfo getCoteireInfo(void)
コータリ情報を持つサーバに対して送られるコータリ情報送信要求．
SET_COTERIE_INFO
void setCoterieInfo(CoterieInfo)
あるサーバが更新したコータリ情報を，他のサーバに知らせるときに送られる．

サーバのコータリ情報を更新する際に用いるメソッドについては，getCoterieInfo はパラメータがなく，コータリ情報を保持する CoterieInfo が返り値として得られる．また setCoterieInfo は，コータリ情報をパラメータとしてとり，返り値はない．

4. システムの状態の動的な変化に対する適応

4.1 既存プロセスの故障

まず最初に，すでに QC に参加しているプロセスが故障した場合の対応方法について説明する．故障の検出は，RMI においてリモートメソッドを呼び出そうとしたときにスローされる例外をとらえることで行う．したがって故障の検出は，メッセージ送信時に行われることになる．

4.1.1 クライアントの故障

クライアントの故障はサーバに対して影響を与える．故障に対する対応方法は，クライアントが故障したときにサーバがどのような状態にいるかによって異なる．具体的には以下の 2 通りの場合がありうる．ここで故障したクライアントを C とする：

ケース 1： C のトークン要求がサーバのキューに入っている場合，すなわち C によってサーバの getTokenRequest メソッドが呼び出された後．

ケース 2： C がサーバからのトークンを受け取ってい

る場合，すなわちサーバによって C の getToken メソッドが呼び出された後．

ケース 1 では，単に C からの要求を無効にして，キュー内に残っている次候補クライアントにトークンを渡せばよい．それに対してケース 2 では，C とともにトークンまで紛失してしまうことになるため，トークンを明示的に回復する必要がある．そのため本実装では，サーバがトークンを渡したクライアントの生存を `Checker` クラスを用いて定期的に検査するようにしている．検査の結果故障が検出されれば，ただちに検査用のスレッドからトークン返却メッセージが送信されてトークンが回復される．

4.1.2 サーバの故障

サーバの故障はクライアントに対して影響を与える．故障に対する対応方法は，サーバが故障したときにクライアントがどのような状態にいるかによって異なる．具体的には以下の 3 通りの場合がありうる．ここで故障したサーバを S とする：

ケース 1： クライアントが S にトークンを要求する前の場合．

ケース 2： クライアントからのトークン要求が S のキューに入っている場合，すなわちクライアントによって S の getTokenRequest メソッドが呼び出された後．

ケース 3： クライアントが S のトークンを渡してもらっている場合，すなわち S によって，クライアントの getToken メソッドが呼び出された後．

ケース 1 では要求を出すサーバを変更し，ケース 3 ではトークンを返却せずにそのまま破棄すればよい．それに対してケース 2 では，故障したサーバとともにトークン要求まで紛失してしまうことになるため，要求の回復などの明示的な対応が必要となる．そのため本実装では，クライアントの方も自分が要求を出したサーバの生存を `Checker` クラスを用いて定期的に検査するようにしている．チェックによってサーバが応答しないことが分かるとそのサーバへの要求を無効にし，新しくサーバを再選択して要求をだす．このとき，集めなければならないトークン数よりも生存しているサーバ数の方が少ないならばどのクライアントも新たに CS に入ることができないため，`QCClient` はそのことを表す例外を `enter_CS` メソッドの外にスローしておく（したがって `enter_CS` を呼び出すときには，必ずこの例外をとらえる必要がある）．

4.2 新規プロセスの追加

次に，新規のプロセスを QC に追加する場合の対応方法について説明する．

4.2.1 サーバリスト

本稿では多数決コータリのみを考えているため，サーバの集合 P を特定すれば，その上で定義されるコータリ $\mathcal{C}$ を一意に決定することができる（一般のコータリの場合は，P のほかに $\mathcal{C}$ の記述も与える必要がある）．本実装では，資源 r_i に関する相互排除に関わるサーバの集合 P_i は，リストの形で保持される．このリストをサーバリストと呼ぶ．サーバリストには 0 から始まるシーケンス番号が付加されており，シーケンス番号はサーバリストが更新されるたびに 1 ずつ増えていく．以下では，リスト L のシーケンス番号を $seq(L)$ と記す．

各クライアントは，自分の関与する資源 r_i ごとにサーバリスト L_i を個別に管理する．最新のサーバリストは L_i に属するすべてのサーバによってメンテナンスされ，後述のように，適切なタイミングでサーバからクライアントに転送される．以下ではプロセス p によって保持されている L_i のコピーを L_i^p と記す．なお以下では，表記を簡潔にするため，文脈からあきらかな場合は添字を省略することにする．

各サーバ p 上には，そのサーバが関与する各資源 r_i ごとに L_i^p，$\bar{L}_i^p$ という 2 種類のサーバリストが保持される．L_i^p はクライアントからのサーバリスト要求に応じてただちに転送されるリストであり，$\bar{L}_i^p$ はサーバの追加に関する情報を保存しておくために用いられるリストである．各リストの使用方法については後述する．

また，すべてのサーバリストは表 1 で定義されている `GET_COTERIE_INFO` および `SET_COTERIE_INFO` により，他のプロセスからアクセスされる．L_i^p と $\bar{L}_i^p$ はメッセージにつけられたオプションで判別する．

4.2.2 追加手続き

まず新規クライアントを追加する場合について説明する．先に述べたように，各クライアントはその起動時に，自分の使用したい共有資源名と，QC に参加しているホスト名 1 つ以上とを知っている．したがって自分の知っているホストに直接問い合わせることにより，使用する共有資源に関するサーバリスト L_i を入手することができる．ただし，サーバが見つからない，あるいはホストが応答しないなど，正常にサーバリストを取得できなかった場合は，例外がスローされてインスタンスは生成されない．

一方，サーバを追加するときは，もしそれが資源 r_i に対する最初のサーバである場合は，サーバが自らその資源に対する新しいサーバリストをつくり，それに自分自身を加えて持っておくだけでよいが，そうでな

いならば，資源 r_i のトークンを管理する既存のサーバに対して，自分が新たに QC に参加することを知らせなくてはならない．結果の整合性を保持するためには，この手続きは相互排除的に行う必要がある．したがってここでは，新規サーバをリストに追加するための権限自体を 1 つの共有資源と見なし，それに対する相互排除を QC プロトコルを用いて分散的に行うという手法をとる（したがって `QCServer` は `QCClient` をメンバとして持つことになる）．

なお，この時点で更新されるのは，クライアントからの要求に応じてただちに転送される可能性のある L_i ではなく，$\bar{L}_i$ のみである．またサーバリストを更新する権限は通常のクライアントによって共有される資源とは別のものとして取り扱われ，したがってクライアントの共有資源へのアクセスを妨げるものではないことに注意されたい．

4.2.3 サーバリストの更新

新規サーバ q が追加される段階で，まず各サーバが所持している $\bar{L}$ が更新される．前述のようにこの更新は q がメンバとして持っている `QCClient` を用いて相互排除的に行われる．CS 中で q は，任意の既存サーバ p から $\bar{L}^p$ のコピーを取得してこれを $\bar{L}^q$ とし，自分のエントリをその中に作成する．このとき $seq(\bar{L}^q) = seq(\bar{L}^p) + 1$ となることに注意されたい．その後，$\bar{L}^q$ 中のすべてのサーバ r に対して，それらの $\bar{L}^r$ を更新するよう指示する．ここで，サーバの故障などの原因によって q による CS の実行が行えない場合は，既存サーバの $\bar{L}$ を安全に更新できないため，追加処理を中断する．

$\bar{L}$ に登録された新規のサーバが実際にクライアントからのトークン要求を受けることができるようになるためには，サーバやクライアントが持つサーバリスト L を更新する必要がある．サーバ上での L の更新方法は以下のとおりである．

(1) クライアント p は，サーバ q からトークンを受け取る際にシーケンス番号 $seq(\bar{L}^q)$ も同時に受け取り，その値を記録しておく．またトークンを集めている間に故障しているサーバを発見した場合は，それらについても記憶しておく．

(2) p が CS から出る際には，CS に入る前にトークンを受け取ったすべてのサーバ q から再び $seq(\bar{L}^q)$ を受け取る．

(3) もしそれらのシーケンス番号が一致しており，かつトークン収集の間に故障サーバを発見していなければ，p はそのまま各サーバにトークンを返却し CS を終了する．
シーケンス番号は一致しているが故障サーバを発見している場合は，p が所持しているサーバリスト L^p は各サーバのリストと同じなので，p は L^p から発見した故障サーバのエントリを削除する（削除の結果得られたリストのシーケンス番号はもとのリストのシーケンス番号より大きくなる）．その後 p は，L^p 中のすべてのサーバ q に対して，その L^q を L^p で置き換えるよう指示する（なお置換えの際にはシーケンス番号も同時に上書きされるものとする）．
故障サーバが発見され，しかもシーケンス番号が一致しない場合は，最も大きな番号を返したサーバ q^* から（最新の）L^{q^*} を取得してこれを L^p とし，それに対して上と同様の操作を行う．ここで，もし自分が発見した故障サーバのエントリがすでに（他のクライアントによって）削除されていれば何もせず，そのまま L^p 中に含まれるすべてのサーバ q に対して，L^q を L^p で置き換えるよう指示する．

一方，クライアント側のサーバリストの更新方法は以下のとおりである．

(1) クライアント p はトークン要求と同時に，シーケンス番号 $seq(L^p)$ をサーバ q に送る．

(2) サーバ q は，各トークン要求ごとに，メッセージに付加されたシーケンス番号を記録する．q がクライアント p にトークンを渡すときには，この番号がシーケンス番号 $seq(L^q)$ よりも小さい（すなわち古い）ならばトークンと一緒に自分の L^q を送り，そうでないならトークンだけを渡す．

(3) クライアント p はトークンと一緒にサーバリストが送られてきた場合は，以降そのリストを使用する．

このように新規プロセスの追加とその後の各プロセスにおけるサーバリスト更新は，プロセス間でサーバリストの情報（シーケンス番号およびリスト本体）を交換することで行われる．このときの情報の交換には表 1 にあげた `GET_COTERIE_INFO` などのメッセージを使用する．

また，この情報交換がトークン送受信などにともなっておこる場合，送信プロセスはそれらのトークン送受信メッセージの中に送信するべきサーバリスト情報を含ませる．たとえばサーバリストの更新処理では，クライアントがサーバからトークンを受け取るときに一緒にシーケンス番号も受け取る．このときはサーバがクライアントに `GET_TOKEN` メッセージを送信する

表 2 実験環境
Table 2 Experimental environment.

CPU	Celeron (500 MHz)
Memory	256 MB
OS	FreeBSD3.4-RELEASE
Runtime	JDK1.1.8
Network	Ethernet 100 Mbps

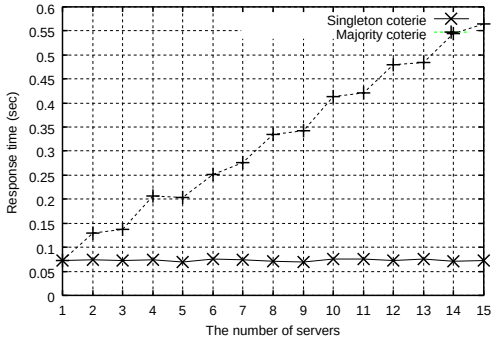

図 1 すべてのプロセスを同一のホストに割り当てたときの応答時間の変化（実測値）
Fig. 1 Effect of the number of servers to the response time under single processing environment.

ことになるが，ここでサーバはそのメッセージに自分のサーバリストのシーケンス番号を付加してクライアントに送信する．

5. 実　　験

本章では，今回試作したクラスライブラリの性能を評価するために行った実験の概要と結果を示す．実験環境の諸元は**表 2** にまとめてある．

5.1 応 答 時 間

まず最初に，トークン要求の競合がおこらないという条件の下で，各クライアントが `enter_CS` を呼び出してから実際に CS に入れるようになるまでの時間（応答時間）を測定した．

すべてのサーバとクライアントを同一のホスト上に割り当てて QC プロトコルを実行させた場合の実行結果を**図 1** に示す．ここで横軸は集合 P のサイズ，縦軸は応答時間（単位は秒）である．図よりあきらかに，応答時間がサーバ数に比例して増加していることが分かる．これは QC プロトコルの実行時間がコーラムを構成するプロセス数，すなわち集めなくてはならないトークン数にほぼ比例するためである．ここで増加の仕方が階段状になっているのは，多数決コータリでは $n = 2k$ と $2k+1$ $(k \geq 1)$ の場合にコーラムサイズが等しくなるためである．

図 2 には，クライアントをサーバと異なるホスト

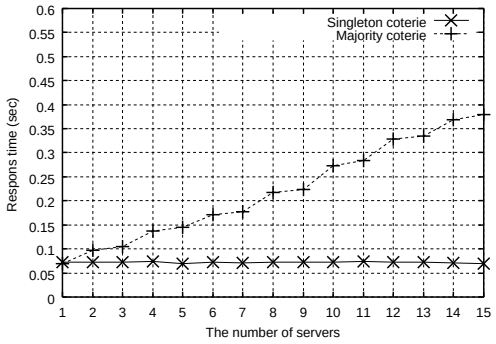

図 2 クライアントをサーバと異なるホストに割り当てたときの応答時間の変化（実測値）
Fig. 2 Effect of the number of servers to the response time under multi processing environment.

上に割り当てて QC プロトコルを実行させた場合の応答時間が示されている（ただし図 1 と同様，サーバはすべて同一のホスト上に存在している）．図よりあきらかに，クライアントとサーバを別々のホストに置いた方が応答時間が短くなることが分かる．その理由は次のように説明することができる：プロセス間のメッセージ送受信はお互いのメッセージハンドラが処理することになる．そのためクライアントとサーバが異なるホスト上に存在すれば，メッセージハンドラが別々のプロセッサで並列に実行されることになり，したがって応答時間は短くなる．

次にクライアントとサーバが別々のホストに存在し，かつ各サーバもそれぞれ異なるホストにおかれる場合について実験した．実験では，使用できるホスト数の制限からサーバ数を 6 に固定し，6 台のホストのそれぞれに 1 つずつサーバをおいた．実験の結果，すべてのプロセスが異なるホスト上で実行されたときの応答時間は約 0.15 秒程度となり，図 2 においてサーバ数 6 のときの応答時間から約 0.02 秒減っている．すべてのプロセスが同じホストにおかれている状態（図 1）からクライアントとサーバを異なるホストにおく状態（図 2）に変えた場合の応答時間の減少が約 0.07 秒程度であったことと比較すると，すべてのサーバを異なるホストにおいた場合の応答時間の減少が少ないことが分かる．これは，サーバをすべて別のホスト上におくことにより各サーバのメッセージ処理（トークンの送信）は並列に行われるため速くなるが，クライアントでの処理（トークンの受信）は逐次的に行う必要があるためである．つまり，サーバがどれだけ速くトークンを送信したとしても，最低でも，クライアントでの受信処理分の時間はかかることになる．

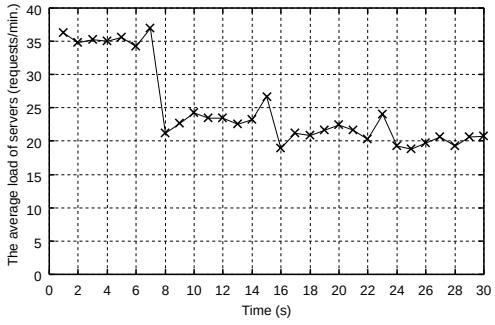

図 3 サーバ数と平均負荷の時間的変化.
Fig. 3 Transition of the average load of servers.

5.2 サーバ数の変化が平均負荷に与える影響

次にサーバ数の変化がサーバ負荷に与える影響と，その時間的な変化について実験を行った．ここでサーバの負荷とは，サーバが単位時間あたりに他プロセスから受けるメッセージの総数として定義される．以下ではサーバの負荷を，特定のサーバの負荷ではなく，全サーバでの平均負荷によって評価する．ここでは QC の実行中にサーバ数が変化する場合の，サーバの平均負荷の時間的な変化を測定した．結果を**図 3** に示す．この図の実験で用いられたイベント系列は以下のとおりである（時刻の単位は秒）：

- 時刻 0：サーバ 1 つと 15 秒間隔で CS 要求を出すクライアントを 10 個を生成（ただし各クライアントの CS 内での処理時間は 0 秒とする）.
- 時刻 6：サーバを 2 つ追加.
- 時刻 14：サーバを 2 つ追加.
- 時刻 22：サーバを 2 つ追加.

図より，サーバの追加にともなって，新規サーバからの $\bar{L}$ 更新要求やクライアントからの L 更新要求などの発生により一時的に負荷が増加しているが，一連のサーバリスト更新操作が終ると負荷は下がっていることが分かる．またサーバ数を 1 から 3 にするとき（時刻 6）と 3 から 5 にするとき（時刻 14）とでは後者のほうが一時的な負荷の増加の度合が若干ではあるが大きいことも見てとれる．これは，全体のサーバ数が増えるに従って新規サーバが出す $\bar{L}$ 更新要求の数自身が増えているためであると考えられる．

6. おわりに

本稿ではコーラムコンセンサスによって相互排除問題を分散的に解くためのクラスライブラリを Java を用いて試作し，その性能を実験により評価した．本試作では，プロトコル処理中のシステムの状態の動的な変化に対する対応が実装レベルでサポートされている．

本稿では実現を容易にするため多数決コータリのみに絞った考察を行ったが，文献ではほかにもさまざまなコータリが提案されている[3]．これらの中には，クライアントが CS に入るにあたって必要となるサーバとのメッセージ交換の回数がより少ないものや，特殊な条件のついた相互排除を行うことのできるものなどがある．今後の課題としては，このようなコータリを用いる際にどのようにコータリの構造の情報を記述，共有するかという問題があげられる．また，特定のプロセスからのみ要求を受け付けるなどのセキュリティを考慮した機能を追加する必要がある．

参 考 文 献

1) Ricart, G. and Agrawala, A.K.: An imal algorithm for mutual exclusion in computer network, *Comm. ACM*, Vol.24, No.1, pp.9–17 (1981).
2) Garcia-Molina, H. and Barbara, D.: How to assign votes in a distributed system, *J. ACM*, Vol.32, No.4, pp.841–860 (1985).
3) Kakugawa, H., Fujita, M.S. and Ae, T.: A distrubuted k-mutual exclusion algorithm using k-coterie, *Inform. Process. Lett.*, Vol.49, pp.213–218 (1994).
4) Lamport, L.: Time, clocks, and the ordering of events in a distributed system, *Comm. ACM*, Vol.21, No.7, pp.558–565 (1978).
5) Neilsen, M.L. and Mizuno, M.: Nondominated k-coteries for multiple mutual exclusion, *Inform. Process. Lett.*, Vol.50, pp.247–252 (1994).
6) Raynal, M.: A simple taxnomy for distributed mutual exclusion algorithms, *ACM Operating Systems Review*, Vol.25, No.2, pp.47–51 (1991).
7) Srimani, P.K. and Reddy, R.L.N.: Another distributed algorithm for multiple entries to a critical section, *Inform. Process. Lett.*, Vol.41, pp.51–57 (1992).
8) Thomas, R.H.: A majority consensus approach to concurrency control, *ACM Trans. Database Syst.*, Vol.4, pp.180–209 (1979).
9) Ibaraki, T. and Kameda, T.: A theory of coteries: Mutual exclusion in distributed systems, *IEEE Trans. Parallel and Distributed Systems*, Vol.4, No.7, pp.779–794 (1993).

(平成 12 年 12 月 20 日受付)
(平成 13 年 9 月 12 日採録)

丸山 英明

2000年広島大学工学部第二類（電気系）卒業．現在，同大学大学院博士前期課程在学中．システムソフトウェアに興味を持つ．

伊東 靖英

1979年名古屋電気通信工学院卒業，同年広島大学工学部技官．1992年広島大学工学部助手．以後，計算機ネットワーク，分散システムの研究に従事．

藤田 聡（正会員）

1985年広島大学工学部第二類（電気系）卒業．1990年同大学大学院博士課程了．工学博士．現在，広島大学工学部助教授．この間，カナダサイモンフレーザー大学客員研究員（1995），パリ南大学客員研究員（1996）．並列・分散アルゴリズム，組合せ最適化問題に興味を持つ．日本応用数理学会，IEEE，SIAM各会員．