多人数協調型分散システムにおける アクセス負荷分散ミドルウェア

松本 直樹 北野 智広 梅津 高朗 山口 弘純 東野 輝夫

大阪大学大学院情報科学研究科

本稿では，多数 (例えば数千規模) のクライアントによる協調型の分散システムを容易に実現するための設計支援法について述べる．提案手法では、設計者が各クライアントの動作を Java により記述し，クライアント間の同期を簡潔に指定するために提供されたメソッドを用いる．ミドルウェアはこのプログラム群を複数の負荷分散用マネージャからなるアーキテクチャ上で動作可能なプログラムに変換し、多数のクライアントで効率の良い協調動作を可能にする。

Middleware for Load Balancing in Collaborative Distributed Systems with Many Clients

Naoki Matsumoto Tomohiro Kitano Takaaki Umedu
Hirozumi Yamaguchi Teruo Higashino

Graduate School of Information Science and Technology, Osaka Univ., Japan

In this paper, we propose a novel approach to help designers in developing a collaborative distributed system with thousands of clients. In the proposed technique, developers can describe clients' programs in Java using special methods specifying synchronization with the other clients. Then we translate the given programs into a set of collaborative programs running on the distributed architecture which consists of a set of machines and a load balancer.

1 まえがき

近年，インターネットへの常時接続環境が一般的となり，加えて計算機の処理能力が飛躍的に向上したことから，オンラインゲーム・遠隔講義支援システムをはじめとする多人数参加型分散アプリケーションの今後一層の需要が予想される．このような多人数参加型分散アプリケーションでは，レイヤ 4 の負荷分散スイッチ／レイヤ 3 の DNS ラウンドロビンなどの汎用負荷分散アーキテクチャに加えて，アプリケーションレベルの負荷分散や効率化が極めて重要である．

しかし，このようなシステムは，クライアント間の同期条件判定や，共有リソースへのアクセスを多数含む複雑なプログラムの集合体であり，不用意な記述や実行は，単一サーバへの集中による共有リソースの I/O ボトルネックや，同期待ちによるレスポンスタイムの増大など，システムのパフォーマンス低下を招く可能性が高い．そこで本論文では，数千規模のクライアントを含む分散協調アプリケーションを対象とした設計支援法により，自動的にパフォーマンスの高い分散協調プログラムを生成する手法を提案する．

提案手法においては，設計者は，実際にはサーバが何台で構成されるかを意識することなく，システムを多数のクライアント及びそれらのクライアントから任意にアクセス可能な共有オブジェクトを管理する単一のサーバからなるとみなして仮想プログラムを記述する．そのように与えられた仮想プログラムに対し，提案手法は，複数のサーバからなる分散環境で負荷を分散しながら効率よく実行するための具体的なサーバへの通信方法を含む動作プログラムへ変換する．

負荷分散は与えられた仮想プログラムの静的な負荷見積もりをもとに静的スケジューリングを行うことで実現する．具体的には，クライアントプログラムとサーバプログラムのソースコードからそれらの制御フローを抽出し，動作の実行順序を制御フローグラフで表現する．このグラフから，(1) 同時に実行される可能性のあるジョブの集合，(2) 同時に一つのジョブの実行を要求するユーザの最大数，(3)I/O ジョブが同時に実行される可能性 などの情報を抽出する．この情報と指定された実際のサーバ台数から，どの動作をサーバに行わせるかをスケジューリングポリシに基づき決定する．この決定に基づき，仮想プログラムは，各クライアント及びサーバがどのタイミングでどのサーバにアクセスすべきかが指定された動作プログラムに変換される．

現在サーバの負荷分散を実現する方法として，ビジネス向けでは負荷分散ハードウェアを導入することが一般的であり，主にジョブのリダイレクションを行うことで 1 つのサーバへの負荷集中を回避している．また，このような機能をソフトウェアで実装している例もあり，アプリケーションに依存しない汎用スケジューラの研究が進められている．アプリケーションレベルでスケジューリングを行うシステムとしては Condor[1] や LSF[2] などが知られているが，いずれもジョブの実行順序などの条件指定に基づき動的スケジューリングを行うもので，同時に多数のアクセスが集中する状況を予め回避するように静的なスケジューリングを行うためには設計者が自らタスクフロー解析を行う必要がある．

このようなジョブスケジューラは近年特にグリッドコンピューティングの研究分野でニーズが高まっており，ミドルウェアとして負荷分散やセキュリティなどの機能を提供する方法の研究が進められている．グリッドベースのスケジューラとしては AppLeS[3][4]・NetSolve[5] などが知られているが，ジョブの実行順序や負荷などを考慮した静的なスケジューリングは行っていない．

本論文は以下のように構成する．2 章では提案手法の概要を述べ，入力プログラム (仮想プログラム) の記述方法や出力プログラム (動作プログラム) について述べていく．3 章では

負荷の見積もり及び動作のスケジューリング方法を述べる．4章では静的スケジューリングを行った場合と単純なラウンドロビンによるスケジューリングでどの程度の性能差が生じるかを簡単な例を用いて検証する．最後に5章で本論文のまとめを行う．

2 提案手法の概要

対象とするのはクライアント間での協調動作を含む分散アプリケーションであり，例としては位置情報を利用した街頭でのコミュニケーションシステム，携帯を利用したオンラインのロールプレイングゲーム，遠隔講義システムなどがある．このようなシステムでは，ある特定のタイミングでユーザの同期をとる・共有リソースに対してアクセスするといった処理をJavaのRMIやCのRPCによってサーバに対して行う．このようなサーバ側のメソッドの呼び出しの単位をジョブの投入と呼ぶことにする．

先にあげたようなシステムでは，位置情報の更新のようなデータサイズの小さな通信から動画ファイルの転送のようなデータサイズの大きな通信まで，優先度や負荷に大きな差が存在するため，負荷分散を考慮に入れた静的スケジューリングが望ましい．また，対象とするクライアント数が多い場合は効率よく同期をとる機構も必要と思われる．これら分散アプリケーション全般に求められる機能を自動で実現し，設計の負担を軽減するのが本研究の目標である．

2.1 アプリケーション例

複数の種類のクライアントプログラムとサーバプログラムからなる遠隔講義システムを考える．提案手法が目指すプログラム変換の概念図を図1に示す．

C_1,C_2,C_3 は各々講義受講・テスト・会議用のクライアントプログラムとし，対応する S_1,S_2,S_3 はサーバサイドのサービスを記述したプログラムとする．また，これまでの運用により，それぞれのクライアントプログラムの利用者がおおよそ1800人・20人・400人であることが分かっているとする．設計者は，各クライアントの種別ごとにプログラムを記述し，クライアント数はパラメータとして与える．

このように与えられた仮想プログラム (図1(a)) に対し，提案手法では利用可能なサーバ台数を与えるだけで，そのクライアント数及びサーバ数に適した分散動作を行うプログラムに自動的に変換する．

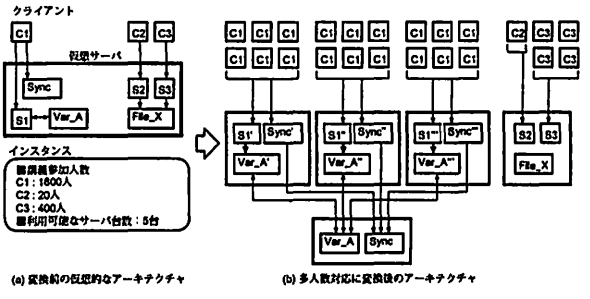

図 1: 提案手法の概念図

2.2 入力/出力プログラムの概要

入力プログラムにはクライアントで動作するプログラムと単一の仮想サーバで動作するプログラムの2種類があり，本研究ではJavaで記述されたプログラムを仮定する．

- クライアントのプログラムは以下専有クラスと呼ぶ．専有クラスは個々のクライアントのみが利用するプライベートなメソッド群と変数群からなる．前節における3種類のクライアントのように，動作が異なるクライアントは個別の専有クラスとして記述する．後述する共有クラスの変数及びメソッド呼び出しも記述することができる．さらに，クライアント間の直接通信も指定することができる．
- 仮想サーバのプログラムには仮想サーバが提供するサービス (クライアント間同期やファイルアクセスなどを行うメソッド群と変数群) を記述し，以下これを共有クラスと呼ぶ．共有クラスには識別子 shared を付与する．共有クラスのメソッド群や変数群は特にプライベート指定がない限りは，全てクライアントからアクセス可能な共有リソース及び共有メソッドである．この共有クラスを用いて，サーバ再度の変数やファイルなどのリソースへのアクセス，さらにクライアント間のサーバでの待ち合わせ (同期) などを記述することができる．
- インスタンス定義ファイル
  専有クラスごとの数・サーバ数を規定する．

一方出力として得られるのは，入力として与えられたクライアント群のプログラムの共有クラスへのアクセスを以下で述べる通信クラスの同名変数及びメソッドに変更したもの，分散されたサーバプログラム群，及びサーバ再度で通信クラスからのアクセスを処理する同期クラスからなるプログラムである．クライアント側では通信クラス，サーバ側では同期クラスによって通信を隠蔽している．

- 通信クラス
  専有クラスからの共有変数へのアクセスを横取し，スケジューリングによりあらかじめ決定されている適切なサーバにアクセスする，クライアントサイドのクラス．通信クラスはいわゆるクライアントスタブである．
- 同期クラス
  サーバ側の共有クラスの通信部分を隠蔽し，通信クラスからの通信を処理する．同期クラスはいわゆるスケルトン (サーバスタブ) に相当する．

図2に通信クラスと同期クラスによる共有変数アクセス及び共有メソッド呼び出しの実現方法を示す．クライアントプログラムは通信クラスの変数にポーリングを行うことで処理の完了を知る．

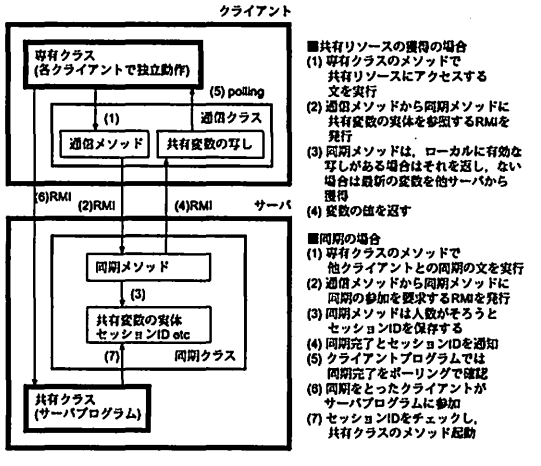

図 2: 分散環境での共有変数及び共有メソッドの実現方法

2.3 専有クラス (クライアントプログラム)

各クライアントで独立動作をするメソッドを持つ専有クラスは以下の条件を満たすものとする．

- 共有リソースへのアクセスを行うフローはシーケンシャルなプログラムで，分岐は可能だが並行処理は行わないものとする．ジョブを投入しないフローを並行処理とし

て指定することは可能であるとする.
- 共有リソースへのアクセスについては指定のメソッド呼び出しのフォーマットに従う.

2.3.1 共有クラスへのアクセスにおけるクライアントでの明示的な排他制御指定

共有変数は，同時に複数のクライアントが読み書きをする可能性があるが，提案手法ではどの演算及び代入文においてもその原子性は保障しない．したがって，クライアントが原子性を保障したいブロックに対しては，sychronized ブロックまたは synchronized メソッドを用いて明示的に原子性を宣言させる．synchronized ブロック内はクリティカルセクションとして扱われ，他のクライアントに値が汚されないことが保証される．

以下の例は，変数 x に対し if 文全体を不可分として指定した例である．

```
public class Client{
    shared Data x;
    public static void main(String args[]){
        synchronized (x) {
            if(x.value != 1000) x.value++;
        }
    }
}
```

このように，shared 宣言された共有リソースのロックを明示的に記述した場合，ブロック内部のコードはサーバ上で動作するのが望ましいため，提案手法ではこのブロックを単純な RMI 呼出し文に置き換える．これにより if 文の条件判定と x のインクリメントはサーバ上で不可分に行われる．

2.3.2 共有クラスの呼び出し方法

設計者がサーバプログラムとして実装した共有クラスのメソッドをクライアント側から呼び出す場合は，以下のフォーマットに従う．

```
shared RemoteClass obj = new RemoteClass();
rmi(obj);
```

RemoteClass とは設計者が用意した共有クラスであり，そのオブジェクトを引数として与えて呼び出すことで簡単にサーバサイドのサービスを利用できる．rmi というメソッドは通信クラスで引数のオブジェクトの型に応じて多重定義される．メソッド rmi では，その型に応じて実際のジョブの投入方法が決定され，クライアントごとに投入先のサーバが異なるよう実装される．

また，典型的な機能を効率的に実装するために，特別な実装方法を行うよう指示するためのメソッドを用意した．例えば，インクリメント&チェックメソッドとして指定された共有変数が指定された値未満であればインクリメントを行って true を返し，値を超えていれば行わず false を返すメソッドなどである．

2.3.3 クライアント間通信

クライアント間通信を行う場合は，ターゲットとなるクライアントまたはグループの識別子 (ディレクトリネーム) とオブジェクトの参照を渡す．同期クラスのメソッド rmi では，引数を解析して適切なターゲットに対して RMI で通信を行う．

```
shared RemoteClass obj = new RemoteClass();
rmi("mcast://lecture01/group/TEAM_A",obj);
rmi("unicast://lecture01/usr/n-matumt",obj);
```

2.4 共有クラス (サーバプログラム)

サーバ側で動くプログラムとしては採点を行う遠隔メソッド・会議の投稿を参加者に配布する遠隔メソッドなどが考えられる．以下の制限を課す．
- サーバで行う処理内容のみを記述し，クライアント間の同期処理は記述しない．クライアントの同期を利用する場合は，同期クラスの変数を参照する (図 2(7))
- あるサーバからあるサーバへジョブを転送するなどの処理は許可しない (サーバプログラム上での直接通信は許可せず，共有変数を用いる).

2.5 プログラムの出力方法

2.5.1 出力プログラムの概要

クライアントプログラムの通信をサポートする通信クラス，サーバサイドで同期の待ち合わせやリソースの一貫性制御を行う同期クラスが導出される．

提案手法により各クライアントごとに 1 つの通信クラスが自動導出され，設計者が入力として記述したクライアントプログラムから利用される．このクラスの通信メソッドでは解析結果に基づいて RMI の呼び出し先サーバをクライアント毎に定めており，これによって単一サーバへのアクセスの集中を回避する．また，入力プログラム上で共有リソースに直接読み書きを行っていた文は，RMI の呼出し文に置き換えられる．実行結果は，通信クラス内にある特定の変数をポーリングして得る (図 2 の (5)).

同期メソッドでは同期の待ち合わせの要求があれば人数のカウントを行い，待ち合わせが完了するとセッション ID を発行してクライアントに返す.

2.5.2 導出方法の概要

入力であたえられたプログラムを合成し，参加人数・サーバ台数などの情報に基づき解析することによって，プログラムのどの部分にどれだけの数のクライアントが同時にアクセスをする可能性があるかを静的に求めることができる．図 3 のように入力として参加人数をあたえ，各領域でアクセスする可能性のある変数やファイル・共有メソッドを列挙することによって，それら共有リソースをどのサーバに配置するかを決定する．この配置に基づいて，通信クラスで規定するジョブの送信先を決定する．なお，共有リソースのサーバへの配置方法に関しては第 3 章で詳細を述べる.

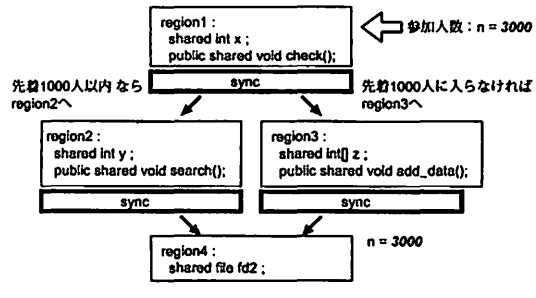

図 3: アクセス負荷の解析

2.5.3 共有リソースの一貫性問題

前節で述べたように仮想的な共有リソースの実体は各サーバに分散するため，それらの一貫性を同期クラスでとる必要がある．共有変数・共有ファイルの一貫性制御には，共有メモリ方式のキャッシュの一貫性をとるプロトコルを利用する．プロトコルには無効型・更新型があり，1 つのプロセッサが連続してアクセスする傾向が強い変数に関しては無効型が有利

で，多くのプロセッサにより頻繁にデータ交換が行われる変数に対しては更新型が有利といわれている．どちらが適しているかはアプリケーションにも依存するので，ライブラリとして提供することで選択可能にする．

3 静的スケジューリングの導出

本章では具体的にどのようなスケジューリングを行うかについて説明する．

対象とするアプリケーションが長時間サーバを占有するI/O要求などを含む場合，他のクライアントが要求したイベント通知のような短時間で完了するジョブの待ち時間が長くなる可能性がある．本研究ではそのような状況を生じさせないように予め静的スケジューリングを行う．具体的には同期要求のような短時間ジョブとファイルI/Oのような長時間ジョブを明示的に区別し，長時間ジョブの後に短時間ジョブが多数サーバ側に送られることが解析結果から予め分かっている場合は，それらを処理するサーバを分ける．

3.1 制御フロー解析の概要

負荷分散は，設計者が用意したプログラムを解析して静的なスケジューリングを行うことによって実現する．

（1） 専有クラス $C_1..C_n$，共有クラス $S_1..S_m$ の制御フローをそれぞれ抽出する．ここでは抽出された制御フローを入力とみなす．

（2） それらを合成して全体の制御フローを得る．

（3） n 種類のクライアントプログラム各々の最大人数 $k_1..k_n$ に対し具体的な数値を与え，共有変数へのアクセスで同時に最大何人のクライアントからアクセスがくるかを求める．

（4） 制御フローを解析し，サーバ上の処理に対する負荷を求める．後述する方法でサーバプログラムの各々の投入されたジョブの重み (実行負荷) を計算し，その和をホスト1台の処理能力で割ることで，必要な台数の見積もりを行う．

（5） 制御フローを解析して，同時に発生しない処理の組合せでより大きいものを求める．

（6） 解析結果に基づき，サーバプログラムをサーバに配置していく．その配置結果に応じてジョブの送信先を決定することができ，出力である通信メソッド $J_1..J_n$ および同期メソッド $B_1..B_m$ が得られる．

3.2 負荷の解析

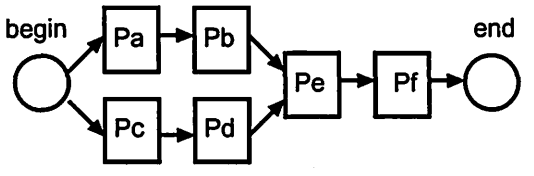

図 4: 制御フローの例

システムの制御フローはクライアントプログラムとして記述されており，それに含まれるジョブの処理時間はサーバプログラムのモジュールを単体で実行したときの実行時間で得られる．本手法では，クライアントプログラム群を合成することにより必要と予想されるサーバの台数を決定する．

ここで，例として図4のような制御フローを考えることにする．P_a，P_b，P_c，P_d，P_e，P_f はそれぞれサーバに投入されるジョブであり，それらを単独で実行したときの処理時間がそれぞれ T_a，T_b，T_c，T_d，T_e，T_f であるとする．また，合成結果よりそれぞれ最大 N_a，N_b，N_c，N_d，N_e，N_f 人のクライアントから同時に処理要求が来ると仮定すると，ジョブの負荷 W は $W = NT$ で定義する．

次に全体で必要となるホストの最大台数の見積方法について示す．ここでは簡単のためサーバの性能は全て均一であるとし，その処理能力を定数 $Spec$ で表すことにする．

まず，制御フロー上で逐次的に処理される部分については P_a と P_b，P_c と P_d，P_e と P_f がそれぞれ同時に起こることはないので，負荷は逐次部分のモジュールの中で最大のものである．すなわち $Max[W_a, W_b]$，$Max[W_c, W_d]$，$Max[W_e, W_f]$ である．

逐次処理の部分をひとつにまとめた後，並列処理されている部分に注目する．並列処理されている部分は同時に起こる可能性を考慮し，負荷はそれぞれの和で求める．

このようにして逐次処理部分・並列処理部分の負荷を計算していく．処理の重みは実行時間で計算しているので，このモジュールを処理するのに必要な台数は

$$\frac{Max[(Max[W_a, W_b] + Max[W_c, W_d]), Max[W_e, W_f]]}{Spec}$$

で与えられる．

3.3 ジョブの送信先の決定

本節では，解析の結果得られた同期の最大クライアント数やジョブ単体の処理時間 (負荷) の情報を元に，スケジューリングを導出するまでに至る流れを説明する．

まず，図5(a) のように合成した制御フローが得られたとする．図において，それぞれの遠隔メソッドに割り振られた数は，クライアント数を表している．この例では各ジョブで同期に参加する人数が静的に分かっており，A・B・C，3つのフロー (それぞれ人数は 100・200・200) がある．また，各ジョブは処理時間が異なっており，それらの値はそれぞれ W_1，W_2，W_3，W_4，W_5，W_6，W_7，W_8 で与えられるとする．

次に人数を横軸，処理時間を縦軸に取ってそれぞれの遠隔メソッドをタイリングし，図5(b) を得る．次のステップで各遠隔メソッドをサーバに割り振るが，単純にフローごとサーバに割り振る方法では図の網掛け部分で示したように部分的に無駄な領域が生じる．そのような場合は，横のタスクモジュールと組にして，それらの面積の比を横の長さの比にすることで無駄な部分を有効に利用できるようにする (図5(c))．最後に与えられたサーバの台数で全体を分割し，図5(d) が得られる．ただし，横に並んだ2つのタスクモジュール境界線の部分に分割の線が合わなかった場合は，境界線と分割線の間を両方のタスクを実行する領域として定める．

4 シミュレーション

本章では簡単なサンプルプログラムを作成し，提案方式がどの程度負荷分散を実現できるかをシミュレーションを通じて考察した．ここでは対象とする分散アプリケーションで考えられる典型的なケースを取り上げ，単純なラウンドロビンスケジューリングとの性能比較を行う．

一般にジョブの送信先を決定する方法は3通りあり，それらは (1) 予めジョブの送信先の候補になっているホストの中からランダムで1つを選出する方法 (ラウンドロビン)，(2) 各ホストの負荷を定期的にチェックしジョブの送信先を決定する方法 (動的スケジューリング)，(3) 予めプログラムの解析を行うことで負荷を予測し静的にジョブの送信先を決めておく方法 (静的スケジューリング)，である．(1) の手法では実行時間が大きく異なるジョブでの負荷分散が難しいという問題，(2) の手法はサーバに対してジョブの投入先を問い合わせる部分でボトルネックとなるといった問題が知られている．

そこで，(1) 及び (2) に対して本研究で提案する (3) の手法を用いることでどの程度の改善を見込めるかを確認するため，2つの実験を行った．

4.1 実験 1

特に処理時間に大きな差がある2種類のジョブ (リモートオブジェクト上の共有変数を操作するジョブとリモートホス

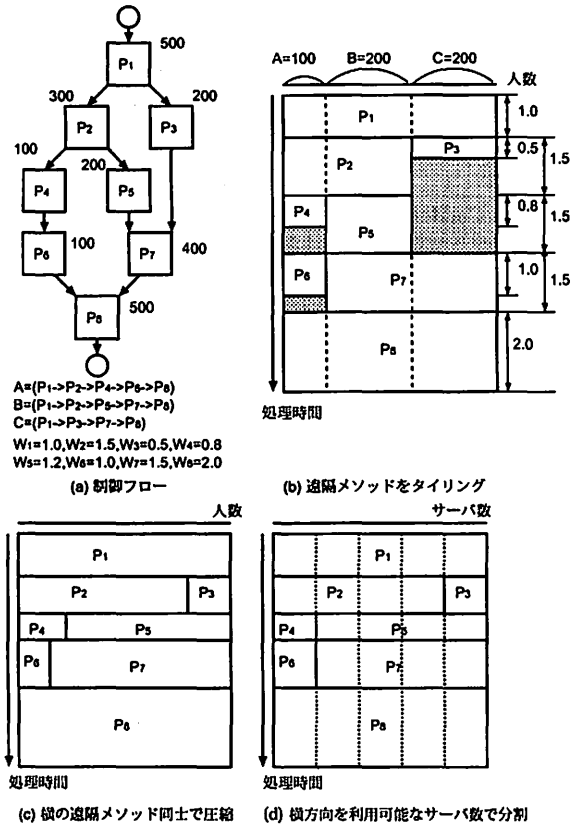

図 5: 静的スケジューリングの流れ

ト上のファイルを操作するジョブ) を用意し，いくつか異なるシナリオを想定して静的スケジューリングの効果を検討する．以下の仮定をおく．

- サーバを 5 台用意し，別のクライアントホストから多数のジョブを送信する．送信するジョブは 2 種類あり，一方は数ミリ〜数十ミリ秒でレスポンスが得られる軽いジョブ，もう一方は数秒〜20 秒程度処理に時間がかかるファイル入出力処理を伴う重いジョブである．
- (1) 軽いジョブと重いジョブを交互に発生させた場合 (2) 重いジョブを送信して 2000msec 後に軽いジョブを送信した場合 の 2 通りのシナリオを用意する．また，重いジョブで扱うファイルを (A) サイズが比較的小さい画像ファイル (B) サイズが大きい画像ファイル の 2 通りで計測し，軽いジョブに対する重いジョブの処理時間の比率がどのように影響するかについても調査した．
- クライアントホストで動かすプログラムは 2 種類あり，1 つは 5 台のサーバからランダムでジョブの送信先を 1 つ選択するラウンドロビンスケジューリング，もう 1 つは予め指定した 4 台のサーバにサイズの大きいジョブ (ファイル入出力の伴うジョブ) を割り当てて残りの一台に軽いジョブを割り当てる静的スケジューリングに従う，というものである．
- 測定するのは全てのクライアントに結果が返るまでの時間 (すなわち全体の処理時間) と，個々のジョブの待ち時間の平均 (2 種類のジョブで別々に求める) である．

4.1.1 計測結果及び考察

軽いジョブを 1200 個・重いジョブを 120 個用意し，(1) 軽いジョブと重いジョブを交互に発生させた場合 (2) 重いジョブを 2000msec 先に送信してから軽いジョブを送信した場合 それぞれについて (A) 重いジョブでサイズの比較的小さめなファイルを扱った場合 (B) 重いジョブでサイズの大きいファイルを扱った場合 の合計 4 通りで全体の処理時間 R_T および個々のジョブの平均待ち時間 W_{Light}, W_{Heavy} を求めた (RR: ラウンドロビン，SS:静的スケジューリング，W_{Light}:軽いジョブの平均待ち時間，W_{Heavy}:重いジョブの平均待ち時間)．単位は msec(ミリ秒) である．

上の表と下の表で数値を比較すると，全体の処理時間は RR の場合と SS の場合で大差がないが，SS は軽いジョブの待ち時間が安定して低いということが指摘できる．

RR	(1)-(A)	(2)-(A)	(1)-(B)	(2)-(B)
R_T(ms)	79356	30777	65656	81227
W_{Light}(ms)	9360	368	215	1260
W_{Heavy}(ms)	5679	404	439	787

SS	(1)-(A)	(2)-(A)	(1)-(B)	(2)-(B)
R_T(ms)	26788	28961	69375	81574
W_{Light}(ms)	6	5	6	5
W_{Heavy}(ms)	6004	945	716	506

ファイル入出力を含むような重い処理と単純で非常に軽い処理が混在し，同時に 2 種類のジョブが到着する状況 (1) を考えると，ラウンドロビンスケジューリングではロードバランスが大幅に崩れてしまう可能性がある (図 6(a))．そこで，処理の重さを考慮した上で，軽いジョブは重いジョブの後ろのキューにつながないような配慮をした静的スケジューリングを行うことで，図 6(b) のように全体的に均一なキューの長さにすることができた．

静的なスケジューリングを行うことによって，ラウンドロビンに基づくスケジューリングよりも全体の処理時間の短縮につながったのは，特にジョブのサイズの格差が大きい場合 (B) であった．

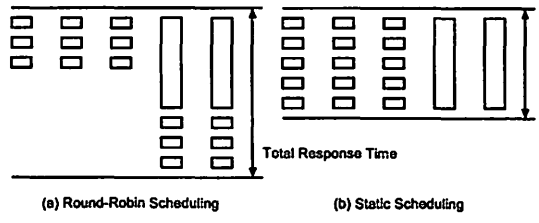

図 6: 2 種類のジョブがほぼ同時に到着する場合

また，ラウンドロビンで割り当てを行う場合は重いジョブのキューの後ろに軽いジョブがつながれ，軽いジョブの待ち時間が実行時間と比較して極めて長くなってしまう可能性がある．それに対し明示的に軽いジョブと重いジョブを分離する静的スケジューリングを行っている場合は，軽いジョブの平均待ち時間が大幅に改善されていることが計測結果から確認できる．

4.2 実験 2

動的スケジューリングでは，ジョブを投入する前に投入先の問い合わせを行う．そこで同等の軽いジョブを複数のクライアントから多数投入し，どの程度までレスポンス時間が悪化するかを確認するため，能力的にほぼ均一な 4 台のクライアントからジョブを投入し，最大待ち時間及び全体の処理時間を計測した．

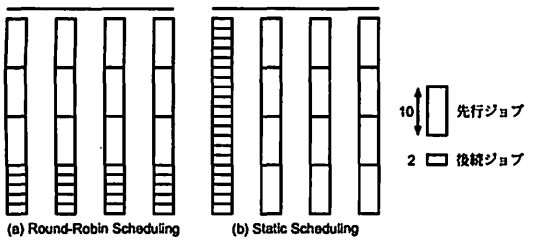

図 7: 2 種類のジョブの到着に差がある場合

4.2.1 計測結果及び考察

計測結果は以下の表の通りになる．4 台のホストそれぞれで 1800 以上のジョブをサーバに投入すると，クライアントホスト 4 台のうち 1〜2 台の要求に対してはエラーメッセージが返された．これより，共有変数にアクセスする程度の比較的軽いジョブであれば一度に 5000 程度まで処理可能であると判断できる．以下の表において，N は各クライアントホストでジョブの送信を行う数，$Max(W)$ は投入されたジョブの待ち時間の中で最大の値，W_{total} は全てのクライアントホストに全てのジョブの結果が返されるまでの時間である．

N	$Max(W)$(ms)	W_{total}(ms)
200	2193	3167
600	5069	6171
1000	10756	13172
1400	21164	27217
1800	34256	34785

4 台のクライアントホストそれぞれで 1000 以上のジョブの投入を行った場合，最大待ち時間が 10 秒程度になっている．これはすなわちジョブの投入先を決めるだけで 10 秒程度かかることを意味する．静的スケジューリングに従ってジョブの投入先を決めている場合，この 10 秒が不要になるため，特に多人数参加型アプリケーションの場合は動的にスケジューリングを決定してしまうと，ジョブの投入先を決める部分だけで相当の時間がかかってしまう (あるいは問い合わせが失敗してしまう) 可能性が高いといえる．

5 まとめ

本稿では，多人数参加型の分散アプリケーションを設計するにあたり，アプリケーションレベルで負荷分散を行うプログラムを自動的に提供する手法を考案し，そのためのアーキテクチャの提案を行った．設計者は仮想的な共有リソース (変数やファイル) を用いたプログラムを設計し，提案手法でそのプログラムの制御フローを解析して負荷分散を考慮した静的スケジューリングを行う．その結果に基づき，共有リソースへの仮想的なアクセスの文をリソースへの実体へのアクセスの文に置き換える．これにより，仮想的な 1 つのサーバで動作するよう記述していたアプリケーションを，比較的容易に多人数に対応させることができる．

簡単な実験結果より，単純にラウンドロビンでジョブの送信先を決定するレイヤ 3 レベルの負荷分散と比較し，処理時間の小さいジョブのレスポンス時間を安定して短くできた．特にネットワークを介して動画などの大容量ファイルを転送するようなジョブは，イベント通知のような軽いジョブと比較してサイズがはるかに大きく，ジョブの実行順序を考慮して予め一部のサーバを投機的に空けておき，軽いジョブが直ちに処理されるように静的なスケジューリングを決めておけば，イベント通知が大幅に遅れて全体の処理時間の遅れにつながるという状況を未然に防ぐことができると考えられる．

今後の課題としては，設計者の用意したプログラムから負荷分散に必要な情報を抽出するために制御フローを構築する方法を考え，そのために必要十分なクラス制限を行い，実際に抽出を行うプログラムの実装を行うことを考えている．また，一部のユーザに動的スケジューリングを行うことによって全体の負荷がどのように変わるかについても，実装及びシミュレーション結果を踏まえて検討していく予定である．

参考文献

[1] The Condor Project. http://www.cs.wisc.edu/condor/

[2] Platform Computing. http://www.platform.com/

[3] Grid Computing Laboratory(GCL). http://apples.ucsd.edu/

[4] H. Casanova, G. Obertelli, F. Berman and R. Wolski : "The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid", Proc. Super Computing, 2002.

[5] NetSolve. http://icl.cs.utk.edu/netsolve/

[6] K. G. Shin and C. J. Hou, "Allocation of Periodic Task Modules with Precedence and Deadline Constraints", IEEE Trans. Computers, vol.46, pp.1338-1356, 1997.

[7] M. Harchol-Balter, C. Li, T. Osogami, A. Scheller-Wolf and M. S. Squillante, "Analysis of Task Assignment with Cycle Stealing under Central Queue", Proc. 23rd Int. Conf. on Distributed Computing Systems (ICDCS'03), pp.628-637, 2003.

[8] D. Durand, R. Jain and D. Tseytlin, "Parallel I/O Scheduling Using Randomized, Distributed Edge Coloring Algorithms", Journal of Parallel and Distributed Computing, vol.63, pp.611-618, 1997.

[9] 甲斐島 武, 市川 昊平, 高坂 貞弘, 伊達 進, 水野 (松本) 由子, 下條 真司: "グリッド技術を用いた効率的並列処理による脳機能解析システム", 情報処理学会 SWoPP2003.