Vol.2010-OS-115 No.15
2010/8/4

軽量なプロセスマイグレーションを可能とするフレームワーク

○上野 康平[†1] 笹田 耕一[†1]

筆者らは，軽量プロセスマイグレーションを用いた分散コンピューティングフレームワークである Izna を開発している．高速なインターネット回線の普及に伴うサービスの多様化，それに伴うサーバシステムの複雑化により，サービスの負荷分散や耐障害性の確保には様々な課題が生じている．Izna では，クライアントからのリクエストを軽量プロセスに結びつけ，これを負荷に応じてサーバ間で移送することで，柔軟な負荷分散を実現した．また，軽量プロセスにリクエストの処理状態を全て持たせることで，サーバの死活に依存しない強い耐障害性を持たせた．今回，Izna フレームワークのプロトタイプ上で実際に現実のウェブアプリケーションと同等のシステムを動作させ提案手法がサービスの負荷分散及び耐障害性の向上に有用であることを確認した．

Framework enabling lightweight process migration

Kouhei Ueno [†1] and Koichi Sasada[†1]

A distributed computing framework based on lightweight process migration has been developed. Providing load distribution and fault tolerance are critical issues in growing diversity of services and complexity of server systems providing them. In our framework, *Izna*, queries from client are each bound to a structure called lightweight process. A flexible load balancing mechanism is provided by *migrating* these lightweight processes over servers. The fault tolerance of the system is ensured by packing full context data needed to process each query onto a lightweight process structure. We have created a small web application on top of our *Izna* framework to show that the framework transparently incorporates load distribution and fault torelance features.

†1 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo

1. はじめに

インターネット回線の普及，高品質化により多様なサービスが膨大なユーザに提供されている．ウェブページの数は 1 億を超え，高精細な画像や動画などを数千万人に提供するメディアになっている．また，画一的な配信だけではなく，一人ひとりに合わせた情報のカスタマイズや，多人数によるリアルタイム情報共有等，双方向の通信を行うなど多様な活用がされている．

これに伴いサーバシステムは複雑化の一途をたどっている．現在のサービスは一般的に複数のコンポーネントによって成り立っている．例えば，画像配信サービスを例にとってみても，実際の画像データを保存するストレージサーバ，視聴者の端末に合わせて画像形式を変換するコンバートサーバ，負荷分散のためのキャッシュサーバ，実際にクライアントからの接続を受け付けるフロントエンドサーバといったように多数のコンポーネントの集合体になっている．

しかし，現状のシステムでは，サービスの負荷分散や耐障害性の確保には課題が多い．サービスを構成するコンポーネントはそれぞれ物理サーバに固定されており，その負荷状況に応じて動的にサーバの割り当てを変えることは困難である．上の画像配信の例では，ストレージを扱うサーバは常にストレージのみを扱い，仮にフロントエンドサーバに負荷がかかっていても余ったリソースを振り分けることはできない．また，コンポーネント間の接続は原則としてコンポーネントを提供するサーバを指定する形で行われ，サーバの故障やサービス構成の変更に柔軟に対応できない．例えば，フロントエンドサーバとキャッシュサーバ間の接続は，フロントエンドサーバ側からキャッシュサーバのホスト名を指定する形で行われるが，サーバの増減はネットワーク設定の変更が必要になる．

筆者らは，軽量プロセスという構造を基本単位とした分散コンピューティング環境である Izna フレームワークを開発している．軽量プロセスは，軽量な処理コンテキストである．Unix 等 OS のプロセスとは違い，物理リソースに束縛されず，最小限の情報のみを保持することで，高速な移送を実現している．軽量プロセスを用いて，Izna では様々な分散システムに特有の問題をシステム側で吸収し，ロジックの実装に専念できる環境を目指している．

軽量プロセスは高速な移送をサポートしており，コンピュータ間の通信は全てプロセス移送によって行われる．クライアントからのリクエストも例外ではなく，リクエストとその処理状況は 1 つの軽量プロセスに対応する．クライアントはリクエストを記述したプロセスをサーバに移送することで計算を委託する．

ⓒ 2010 Information Processing Society of Japan

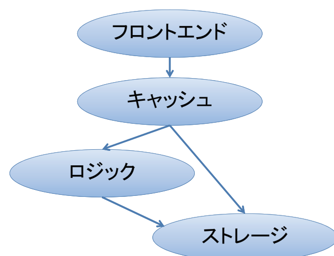

図 1　Web サービスをコンポーネントに分解した例

筆者らは，Izna フレームワークにプロセス移送による透過的な負荷分散及び耐障害機能を実装した．Izna サーバは周辺のサーバの負荷を相互に監視しており，定期的に負荷が高いサーバから低いサーバへプロセスを移送することにより，負荷の平滑化を実現している．耐障害性の確保は，移送先の選択を動的に行うことによって解決している．コンポーネント間の接続はサーバを直接指定するのではなく，次のコンポーネントのみの指定によって行われ，サーバの選択は相互監視にもとづきその時活きているサーバのみが選択される．

本論文では，Izna に実装した軽量プロセスモデル，及びそれを利用した負荷分散，耐障害手法について述べる．また，実際に Izna 上でアプリケーションが構成でき，透過的な負荷分散，耐障害性の確保が可能であることをプロトタイプを利用した評価によって示す．

2. 処理モデル

ここでは，Izna による処理モデルの概要について述べる．

Izna はサービスを提供するプラットフォームである．ここでいうサービスは，ユーザからリクエストを受け，何らかの処理を行い，リプライを返すシステムである．例えば，Web サービスはユーザからのリクエストに応じ，ウェブページを生成し，リプライとして生成したウェブページを返す．

サービスは Izna 上のコンポーネントの集合体として実装される．コンポーネントはサービスを構成する要素であり，それぞれが一つの独立した機能を実現する．例えば，典型的な Web サービスはフロントエンド，キャッシュ，ロジック，ストレージによって成り立っているが，Izna ではこれらの要素をそれぞれコンポーネントとして記述する（図 1）．

サービスの処理の実行は軽量プロセスという構造を用いて行われる．軽量プロセスは，軽量なコンテキストである．Unix 等の OS におけるプロセスとは違い，この軽量プロセスは

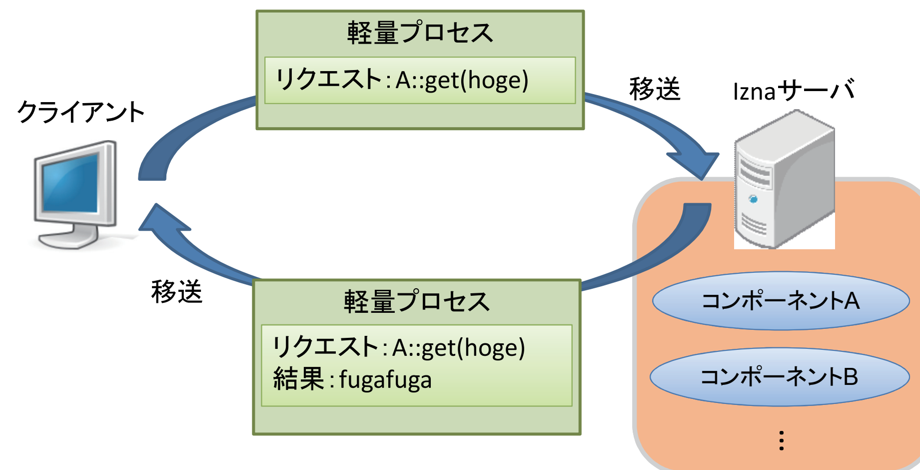

図 2　Izna によるリクエスト処理例

メモリなどのマシンリソースと直接対応してはおらず，内容は環境非依存な形で記述される．Izna は，この軽量プロセスをコンポーネント間でやりとりすることで処理を行う．

コンポーネントは，この軽量プロセスを介して機能を提供するプログラムである．コンポーネントは軽量プロセスを受け入れ，軽量プロセスのコンテキストを基に処理を行い，処理結果を軽量プロセスに追記した後，別のコンポーネントに軽量プロセスを転送する．Izna を用いたサービスは，軽量プロセスをやり取りする複数のコンポーネントの集合体として提供される．

Izna サーバは，このコンポーネントの処理を軽量プロセスに対して行うことで，サービスを提供する．サーバには幾つかのコンポーネントを登録することができる．サーバは受け入れた軽量プロセスに対し，自身に登録されたコンポーネントの処理を実行する処理系として働く．

軽量プロセスは移送をサポートしており，コンピュータ間を移動することができる．クライアントはこの機構を用いてリクエストを記述したプロセスをサーバへ移送させ，サーバへの処理の委託を行う（図 2）．サーバは結果を格納した軽量プロセスを再度クライアントに移送させることで，リプライを返す．またサーバ間でも，負荷分散やルーティングの為，より処理をするのに適したサーバへ軽量プロセスの移送が行われる．

ⓒ 2010 Information Processing Society of Japan

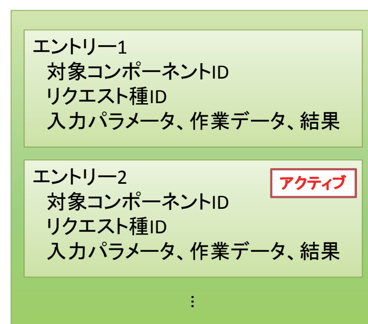

図 3 軽量プロセスの構造

なお，今回の処理モデルの対象は多くとも百台規模のサーバクラスタを前提にしている．フラットなネットワーク構成で接続され，いずれのサーバ間の通信も同じ速度及び接続コストで行われることを前提にしている．

以下では，各要素の詳細な設計について述べる．

2.1 軽量プロセス

軽量プロセスは，Izna における基本的な計算単位である．サービスへのリクエストは全て軽量プロセスを介して行われ，コンテキスト，処理結果等リクエストに関連するデータは全て軽量プロセスの中に記録される．

軽量プロセスは，複数のエントリーからなる（図 3）．それぞれのエントリーは，サービスを構成するコンポーネントへの内部的なリクエストに対応しており，リクエストの対象コンポーネント，種類，入力データ，コンテキスト，及び処理結果が保存される．それぞれの一つは，アクティブとマークされ，次に処理するべき対象となる．コンポーネントがこれをマークする．

アクティブなフラグがあるということをいうだけでもいいきがする

その他，軽量プロセスは，以下に示すサーバ非依存性，高速な移送が可能という特徴を有す．

2.1.1 サーバ非依存性

軽量プロセスは特定の Izna サーバに依存しない．対象のコンポーネントが登録されているサーバであればどこでも処理することができる．

特定のサーバに依存しないことにより，軽量プロセスは高い信頼性を持つ．従来のプロセス移送の多くでは，それぞれオーナーとなるサーバが存在し，そのサーバの死活にプロセスの死活が依存していた．Izna では，軽量プロセスは実行中のサーバの死活のみに依存し，他のサーバの状態に依存することはない．

また，この性質は負荷分散を行う際にも重要である．プロセスは実行中のノードのみに依存するので，必ずしも処理の移送先のノードとの距離を気にする必要はない．移送先サーバの決定は，ノードの負荷状態のみを考慮して行うことができる．

2.1.2 高速な移送

軽量プロセスは，高速な移送が可能である．Izna では，リクエスト毎という細かい粒度で軽量プロセスを移送するため，移送の速度は全体のパフォーマンスに直接影響する．

まず，軽量プロセスの移送にあたっては，移送先で処理を継続するのに必要な最小限のデータのみが転送される．軽量プロセスの移送はコンテキストをサーバ間で転送することによって行われる．ただし，全てのコンテキスト情報を毎回やり取りするのは非効率である．Izna では，高速な移送を実現するため，エントリー単位での遅延転送を導入している．次に処理されるコンポーネントで必要と予測される最近利用されたエントリーのみがまず転送される．その他のエントリーは，参照のみが渡され，必要に応じて遅延転送される．

また，大容量のデータに関しては，分散メモリ機構が活用できる．実際のデータの代わりに，データへのポインタを格納することで，各エントリーのデータ量を抑えている．

前節で述べたサーバ非依存性も，軽量プロセスの高速な移送に貢献している．ホストノードが不在なので，軽量プロセスの移送時には実行ノード変更等の連絡は不要である．よって，軽量プロセスの移送において発生する通信は，移送元と移送先でのコンテキストのやり取りのみである．

2.2 コンポーネント

コンポーネントは Izna におけるサービスの構成単位であり，軽量プロセスを介して機能を提供するプログラムとして記述される．

コンポーネントは，自身を対象とした軽量プロセスに対し処理を行う．コンポーネントは軽量プロセスに記述された一種類もしくは複数種のリクエストを受け付け，対応する機能を提供する．軽量プロセスに記述されたパラメータを読み取り，処理を実行し，実行結果を軽量プロセスに追記し，自身の呼出元を軽量プロセスの対象と書き換える．

コンポーネントの処理は他のコンポーネントに依存することもある．例えば，ウェブサービスの例（図 1）でのキャッシュは，キャッシュミス時の応答データの作成をロジックやストレージに依存している．このような場合，コンポーネントは自身の継続を軽量プロセスに

ⓒ 2010 Information Processing Society of Japan

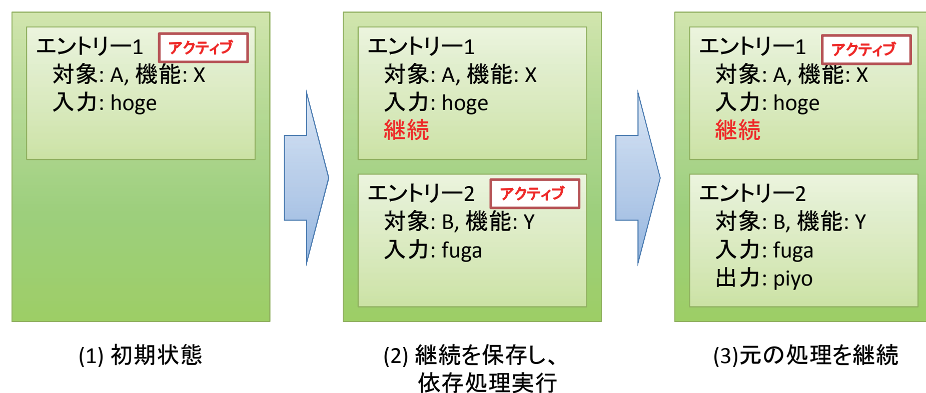

図 4 依存処理実行時の軽量プロセス

保存し，依存コンポーネントに一旦軽量プロセスの実行を移す（図 4）．

コンポーネント A の提供する機能 X がコンポーネント B の機能 Y に依存している場合を考える．まず，コンポーネント A は軽量プロセスに自身の継続を保存し，コンポーネント B 用の新しいエントリーを作成する（図 4(2)）．コンポーネント B は処理実行後，コンポーネント A のエントリー 1 を再度アクティブにし，軽量プロセスは再度 A の処理対象とする（図 4(3)）．コンポーネント A は軽量プロセスに保存された継続から処理を続行する．

2.3 Izna サーバの動作

ここでは，サーバが実際にどのようにリクエストを処理するかを述べる．

Izna サーバは，リクエストが記述された軽量プロセスを受け取り，自身に登録されたコンポーネントを用いて計算処理を行う．まず，軽量プロセスの中で，アクティブなエントリーを見つけ出し，対応するコンポーネントの処理を行う．この時，もし継続情報が軽量プロセスに存在した場合，その時点から処理の継続を行う．コンポーネントでの処理によって，軽量プロセスには処理の結果もしくは継続情報，及び次の処理対象コンポーネントが追記される．処理の終了後，Izna サーバはシステムレベルで軽量プロセスの再スケジューリングを行う．これは，後述するサービスの負荷分散に用いられる．スケジューリングにおいて，実行中のサーバが次のコンポーネントの処理を行うにあたって最適なサーバではないと判断された場合，プロセス移送が行われる．

HTTP 等による，軽量プロセス以外の形でのユーザからのリクエストは，対応するゲートウェイにより吸収される．ユーザからのリクエストはゲートウェイによって軽量プロセスに変換され，Izna サーバへ移送される．サーバ上で処理が行われた後，リプライを追記された軽量プロセスは再度ゲートウェイに移送され，ユーザへリプライとして返される．

3. 負荷分散，耐障害性の確保

提案システムを用いて，システムの負荷分散，及び耐障害性を実現する方法を述べる．

3.1 負 荷 分 散

システムの負荷分散は定期的な軽量プロセスの再スケジューリングによって行われる．各 Izna サーバは自身の周辺のサーバの負荷状況を監視しており，もし周りのサーバに比べて自身の負荷が高いようであれば，軽量プロセスを積極的に他のサーバに移送する．軽量プロセスは原則として物理リソースに束縛されず，マイグレーションコストも低いため，ほぼ負荷情報のみを元に移送先を決定することができる．

サーバの負荷情報はシステム全体で共有される．今回はネットワーク構成がフラットなため，負荷情報の共有は UDP マルチキャストにより行うことができる．各サーバは定期的に自身の CPU，メモリ，ネットワーク負荷をマルチキャストし，この情報をもとに軽量プロセスのスケジューリングを行う．これは，サーバの死活監視も兼ねており，後述する耐障害性の確保にも貢献している．現在の実装では，これらのデータをもとに 1 次元のノード負荷率を計算し，それを指標として用いている．

再スケジューリング処理はコンポーネントの処理ごとに行われる．軽量プロセス移送が発生する条件は以下のいずれかである：

一つは，自身の負荷率が周りのサーバに比べて著しく高い場合である．この場合，システムは負荷の不均衡が起きていると判断し，該当サーバは未処理のものを含め積極的に軽量プロセスを他サーバに移送する．これは，サーバが重い処理を引き受けてしまった場合などに発生する．

もう一つは，アクティブなエントリーが，自身の保持するコンポーネントを対象としていない場合である．サーバは該当するコンポーネントを保持する別のサーバに軽量プロセスを移送し，処理を委託する．

移送先のサーバは，次の処理対象であるコンポーネントを保持するサーバを候補として選択される．まず，負荷情報が最近発信されていないサーバは，非常に高負荷状態にあるか，もしくは異常が発生しているとして，候補から除外される．移送先サーバは，候補をノード負荷により重み付けされた上でランダムに選択される．一見，ここでランダムにノードを選

© 2010 Information Processing Society of Japan

択することにより偏りが発生するように思えるが，再スケジューリングは軽量プロセスごとの細かい粒度で行われるため問題にならない．

3.2 耐障害性の確保

耐障害性の確保は，このシステムにおいて透過的に実現されている機能である．

プロセスの処理先の選択は，前節で述べたサーバ間の相互監視により，活きていることが確認されているサーバのみが対象となる．仮に応答を停止しているサーバが誤って選択されてしまっても，移送が失敗し，活きているサーバが再選択される．個々のコンポーネントには，各リクエスト固有の情報は一切保存されないため，サーバがなんらかの原因で停止した場合でも，他のサーバで処理されているリクエストに影響を及ぼすことはない．ただし，停止したサーバに未処理のプロセスが残されていた場合，その軽量プロセスの内容は失われる．現在は，ゲートウェイでタイムアウトを検知し，リクエストを再送する形で対処している．

4. 予備評価

Izna フレームワークのプロトタイプ実装を用いた評価を行った．実際に小規模なウェブアプリケーションを作製し，提案手法によるサービス構築が実用的であることを確認した．また，提案手法により提供されたサービスの負荷分散，及び耐障害性に関して評価した．

Izna フレームワークのプロトタイプは，Ruby 言語を用いて実装した．上で述べた Izna サーバの機能を全て実装しており，1400 行程度の規模である．ノード間通信には TCP を用いており，Ruby 処理系標準の IO モジュールを使用している．

評価は，Intel Xeon X3440 プロセッサ及び 4GB DDR3 メモリを搭載し，1GbE を用いて接続された実機マシン 3 台を用いた．OS には FreeBSD 8.0 を用い，プロトタイプ実装は Ruby 1.9.1 処理系上で動作させた．

4.1 ウェブアプリケーションの作製

Izna フレームワークの実用性を評価するため，実際に簡易的なウェブアプリケーションを作製した．このアプリケーションでは，mimetex[1) という軽量な数式レンダラを利用し，TEX 記法で記述された数式の描画を行う．

アプリケーションはキャッシュ及びロジックの 2 つのコンポーネントにより構成される（図 5）：

キャッシュ リクエストのキャッシュ機能を提供する．今回，キャッシュのバックエンドには NFS で実装された簡易共有メモリを使用している．以前あったリクエストと同じリクエストが来た場合，共有メモリ上のキャッシュから画像を配信する．新しいリクエストの場合，ロジックコンポーネントにより画像を生成し，共有メモリ上にキャッシュを保存する．

ロジック mimetex により数式のレンダリングを行う．軽量プロセスからユーザ入力の TEX 数式を読みとり，mimetex を実行する．出力画像を軽量プロセス上に保存し，要求元に応答する．

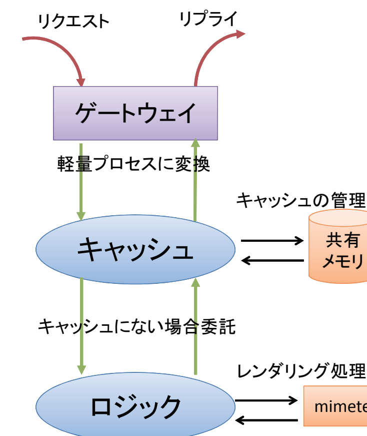

図 5 評価用ウェブアプリケーションのコンポーネント構成

また，これらの Izna 上で実装されたアプリケーションは，軽量プロセスの形でリクエストを受け付けるため，前段にゲートウェイを配置し，クライアントからのリクエストを軽量プロセスに変換する．

軽量プロセスセクション，及び軽量プロセスに対する応答処理の記述により，それぞれのコンポーネントの実装を行った．Izna サーバ上にこれらのコンポーネントを登録し，クライアントから送信したリクエストがサーバによって適切に処理されることを確認した．

4.2 負荷分散の評価

作製したウェブアプリケーションを用い，アプリケーションを動作させるサーバ台数の変化に伴うパフォーマンスへの影響を計測することで，負荷分散の評価を行った．

1 台のマシン上で負荷を生成し，Izna サーバ上で動作させたアプリケーションに対してリクエストを行った．専用のベンチマークスクリプトを作成し，10 並列でそれぞれ 100 リ

ⓒ 2010 Information Processing Society of Japan

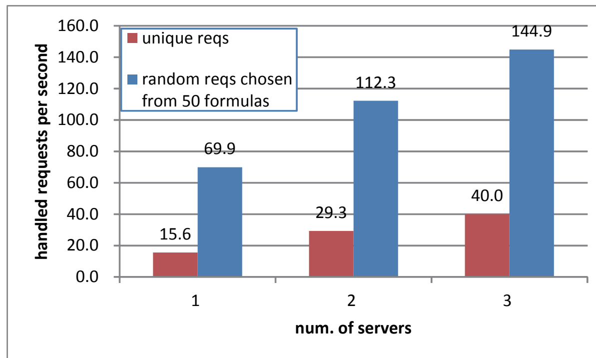

図 6 負荷分散の評価

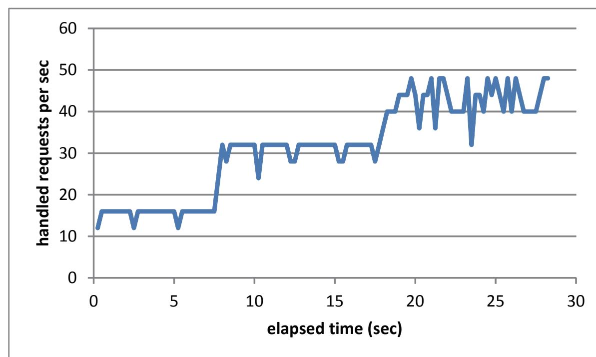

図 7 動的ノード追加に伴うパフォーマンスの変化

クエスト，合計 1000 リクエストを送信し，全てのリクエストの処理が終わるまでの時間を計測した．また，全て異なるリクエストを送った場合と，50 種類のリクエストをランダムに送信した場合に対して評価した．

評価結果は図 6 の通りである．Izna フレームワークに実装された負荷分散機構により，サーバ台数の増加に応じて，同じ時間で処理できるリクエストの数が向上していることがわかる．

また，アプリケーションの動作中にサーバ台数を増加する実験も行った．増加に伴うリクエスト処理数の変化を 1/4 秒ごとにサンプリングして計測した (図 7)．それぞれ，サーバを増加したタイミングで秒間リクエスト処理数が増えており，増加前後でのパフォーマンスの劣化もないことがわかる．

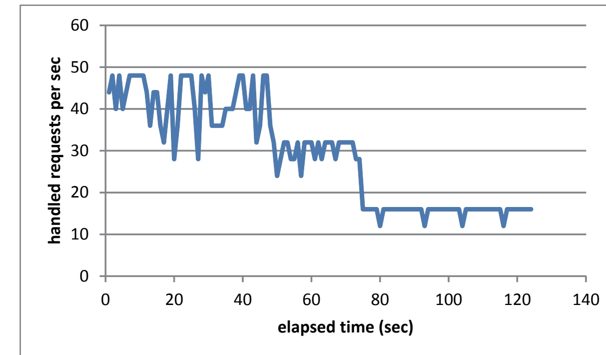

図 8 動的ノード削除に伴うパフォーマンスの変化

4.3 耐障害性の評価

作製したウェブアプリケーションを提供しているサーバを実行中に停止させ，耐障害性の評価を行った．

評価は，前述したベンチマークアプリケーションを実行中に，サーバプロセスを停止することにより行った．ノード増加の際の計測と同様に，リクエスト処理数の変化を 1/4 秒ごとにサンプリングして計測した（図 8）．サーバ停止のタイミングで秒間リクエスト処理数が減っているが，サービス全体の停止には繋がらず，1 台分のパフォーマンスの低下に留まっているのがわかる．また，他システムでよく見られるような，サーバ台数の低下直後のパフォーマンスの大幅な低下も見られない．

5. 今後の課題

今後の Izna フレームワークにおける負荷分散，耐障害性確保において，以下の３つの課題が挙げられる．

１つ目は，トランザクション処理のサポートである．現在の実装では，軽量プロセスのリクエストが完全に遂行されることは，保障されていない．リクエスト処理を行っているサーバが落ちた場合，処理中の軽量プロセスが失われてしまう．この様な場合には，ゲートウェ

ⓒ 2010 Information Processing Society of Japan

イによりリクエストの再送が行われるが，処理の途中で実行が停止することにより発生する不整合が問題になるケースには対処することができない．この問題を解決するために，将来的にはトランザクション機構をサポートする予定である．

２つ目は，新たなサーバへのコンポーネントの動的な追加である．現在は，軽量プロセスレベルでの負荷分散が実装されているが，あくまでコンポーネントが既に登録されているサーバのみを軽量プロセス移送の対象とする．候補となるサーバのいずれもが負荷が高く，コンポーネント登録のコストに見合う場合，他の負荷率が低いサーバに動的にコンポーネントを登録し処理を分散できる対象を増やすことが考えられる．

３つ目は，百台以上の大規模環境への対応である．現実装では，それぞれサーバがネットワークに接続されているサーバ全部の負荷を監視しているため，サーバの数が増えるのに応じて監視のコストが無視できなくなることが予想される．また，ネットワークの構造も重要になり，現在の前提条件である全サーバ間の通信コストが均一という仮定は不適当になると考えられる．これらに対処するため，階層的な負荷分散アルゴリズムを導入する予定である．

6. 関連研究

ここでは，Izna の関連研究について述べる．

6.1 負荷分散，耐障害手法

サービスの動的な負荷分散の為に，今までにも多くの手法が提案されてきた．

多くの場合，アプリケーションレベルで負荷分散の仕組みが用意されている．例えば，Web プロキシサーバの多くはロードバランサを内蔵している．Apache の `mod_proxy_balancer` などではアプリケーションサーバの動的な負荷を考慮した分散が可能である．

また，OS 仮想化の技術を利用した負荷分散手法も多く使われている．多くの仮想化ソフトウェアでは，ライブマイグレーション機能がサポートされており，あるホスト上で動作中の仮想サーバを別ホストにリアルタイムに移送させることができる．Wakame[2) は単体のコンポーネントを提供するサーバを仮想化し，必要に応じて動作させるホスト間で仮想マシンを移送することで動的な負荷分散を実現するためのミドルウェアである．

Izna では，これらのアプローチとは異なり，サービスを構築するフレームワークのレベルで負荷分散及び耐障害性の確保を実現している．この手法の利点は，個別のアプリケーション毎に負荷分散の実装をしなくても済むことである．また，OS 仮想化のアプローチと比較しても，Izna ではリクエスト単位の細かい粒度で調整を行うためよりリソースを効率的に運用できる．さらに，仮想化によるオーバーヘッドや，仮想サーバ移送に伴うダウンタイムなども考慮する必要がない．

6.2 プロセス移送

計算機上で動作しているプロセスを他の計算機上に移送するプロセス移送を利用した負荷分散手法も，従来から研究されている．Barak ら[3) は，Linux 上で動作するプロセスを移送し，負荷分散を行った．Gobelins ら[4) は分散共有メモリ上で移送可能なプロセスを提案している．

また，モバイルエージェントと呼ばれる移送を自律的に行うプロセスモデルも研究されている．Telescript[5) や Agent Tcl[6) では移送にプログラミング言語のレベルで取り組んでいる．他には，Aglets[7) など Java のバイトコード転送を用いたモバイルエージェント環境が数多く提案されている．さらに，Johansen らは Tacoma[8) で独自の強移送用コンテキスト保持方法を実装している．

Izna が他のプロセス移送のアプローチと異なるのは，物理リソースと直接対応しない，高速な移送を前提とした軽量なプロセス構造を用いている点にある．これにより，他の手法と比べてはるかにオーバーヘッドの低いプロセス移送を実現し，柔軟な負荷分散を行っている．

7. まとめ

筆者らは，軽量プロセスを用いた分散コンピューティング環境である Izna フレームワークのプロトタイプ上でサービスを構築し，柔軟に負荷分散，耐障害性の確保が可能なことを示した．まず，コンピュータ間を高速に移送できる軽量プロセスという軽いコンテキストを導入し，クライアントからのリクエストをそれぞれ軽量プロセスに結びつけた．この軽量プロセスを，サーバの負荷に応じて移送することで，透過的な負荷分散を実装した．また，状態を全て軽量プロセス上に載せ，個々のサーバに依存させないことで，高い耐障害性を実現できることを示した．

参考文献

1) John Forkosh Associates, Inc.: mimeTeX quickstart, `http://www.forkosh.dreamhost.com/source_mimetex.html` (2010).
2) Axsh: Wakame, `http://wakame.axsh.jp/` (2010).
3) Barak, A. and La'adan, O.: The MOSIX multicomputer operating system for high performance cluster computing, *Future Generation Computer Systems*, Vol.13, No.4-5, pp.361–372 (1998).
4) Valle, G., Morin, C., Lottiaux, R., Berthou, J.-Y. and Malen, I.D.: Process Mi-

ⓒ 2010 Information Processing Society of Japan

gration Based on Gobelins Distributed Shared Memory, *IEEE International Symposium on Cluster Computing and the Grid*, p.325 (2002).

5) White, J.E.: Telescript technology: mobile agents, pp.460–493 (1999).

6) Gray, R.S.: Agent Tcl: A transportable agent system, *in Proceedings of the Fourth Annual Tcl/Tk Workshop* (1996).

7) IBM Research: Aglets Software Development Kit, `http://www.trl.ibm.com/aglets/` (2002).

8) Johansen, D., Lauvset, K.J., Renesse, R.V., Schneider, F.B., Sudmann, N.P. and Jacobsen, K.: A Tacoma Retrospective, *Software - Practice and Experience*, Vol.32, pp.605–619 (2001).

© 2010 Information Processing Society of Japan