社団法人　情報処理学会　研究報告
IPSJ SIG Technical Report

2005－OS－99（11）
2005／5／26

サーバの透過的な移動のためのOS拡張

横山陽介[†]　大山恵弘[†]　米澤明憲[†]

本研究では、ネットワーク上で動作するサーバのためのプロセスマイグレーションの手法を提案する。クライアント側に移動を意識させずにサーバをホスト間で移動することを困難にしている要素をいくつか挙げ、それらのOSの拡張による解決法とLinux上での実装を説明し、有用性を確認するための予備実験の結果を報告する。

OS Extension for Transparent Migration of Server Processes

Yosuke Yokoyama ,[†] Yoshihiro Oyama [†]
and Akinori Yonezawa[†]

We propose a method of process migration for servers. There are some difficulties in achieving process migration for servers which are not recognized by clients. We explain an OS extension and the implementation on Linux for solving the difficulties. Then, we explain the results of preliminaly experiments.

1. はじめに

ネットワーク上で動作しているサーバプログラムは、停止させないことがしばしば求められる。しかし、サーバを動作させているホストのメンテナンスのためにシャットダウンする時には、サーバを停止せざるを得ない。また、ホスティングなどで複数のサーバを同一ホストで動作させているとき、負荷分散のために負荷の高いサーバを他のホストへ移動させる動的な負荷分散が必要になることがある。しかしこの場合も、サーバを一度停止させて移動先で再開させる必要がある。

これらの問題を解決するには、サーバを動作させたまま他のホストへ移動させることができればよい。これを実現する上で重要なのは、サーバ、クライアント双方にとって移動が透過的であることである。すなわち、サーバ、クライアント双方が移動を意識した実装をすることなく、移動しても接続中の通信が切断されずクライアントにはサーバの位置の変化していないように見せることができる方法が優れている。この方法により、既存のサーバ、クライアントをそのまま用いてサーバの移動を実現することができるからである。

また、導入のコストが低いことも重要である。具体的には、サーバ側のみの最低限の拡張で実現することにより、クライアント側には何も導入する必要がなくてすむ。

サーバの移動を実現すための方法として、プロセスマイグレーション[8]がある。プロセスマイグレーションは、あるホストで動作中のプロセスをその動作状態を保ったまま他のホストへ移動させ、そこで実行を継続させることである。プロセスマイグレーションを実現する方法として、プロセスの動作状態をファイル等に保存し、移動先のホストで状態を復元する方法がある。プロセスの保存、復元を行うシステムをチェックポイントシステムといい、多くのものが提案されてきた。

しかし、サーバプログラムのような複雑かつ大規模なプログラムをこの方法で移動させるのは困難である。その理由は、サーバはファイルやネットワークといった多数の資源を使用しているため移動後に動作環境が変化すると正しく動作しなくなる可能性が高いこと、複数のプロセスか協調して動作しているためその状態の保存、復元が困難であることなどが挙げられる。

これらの問題の解決法はいくつか提案されている。通信やOS内の資源を仮想化し、プロセスには資源やネットワークが変化していないように見せる環境の構築方法が提案された。また、VMware[16]のような仮想機械を利用して、OS全体の環境を移動することでサーバ移動を可能にする仕組みもいくつか提案された。しかし、これらは大規模な仮想環境を利用したシステムであり、また前者には、サーバとクライアント双方

† 東京大学
University of Tokyo

を仮想環境内で動作させる必要がある、後者には、OS全体の環境を移動するため移動にかかるコストが膨大であるといった欠点がある。

そこで我々は、移動にはできるだけ既存の OS の機能を使い、不足する機能に関しては OS の最低限の拡張で補うという方針で、プロセスマイグレーションによるサーバ移動の困難さを解決する方法が有用であると考えた。

本研究では、広く使われている Web サーバである Apache[1] のようなサーバを透過的に移動させるために必要な条件を挙げ、それを OS の最低限の拡張によって実現することを提案する。移動するサーバに変化しない環境を提供し、クライアントとの接続を維持したまま移動するための最低限の仮想化機能を OS に追加するという方式で実現する。また、本方式の Linux 上での実装を説明し、仮想機械を使ったシステムや、大規模な分散環境による実現方法との比較を通じて本研究の利点欠点を議論する。

2. 関 連 研 究

サーバプログラムとして、以下の要素を持ったプログラムを考える。

- 複数のプロセスからなり、互いにシグナル等で通信している。
- TCP/IP でネットワークを介して他のホストと通信を行う。接続待ち状態 (`TCP_LISTEN`) ソケット、接続確立状態 (`TCP_ESTABLISHED`) のソケットを持つ。
- 特定のディレクトリ以下に必要なデータをいくつかのファイルとして保持している。

これらの要素を持つプログラムを既存のシステムで移動させる時の問題点を挙げる。

2.1 チェックポイントシステム

チェックポイントシステムは、プロセスの状態や使用している資源の状態 (レジスタ、メモリ等) をファイルに保存し、後で復元できるようにするシステムである。チェックポイント機能を持つライブラリを利用するもの[11)17)]、プログラム変換によるもの[12)]、カーネルにチェックポイント機能を組み込むものといった[15)18)]、様々な仕組みが提案されてきた。これらを用いることで、プロセスの移動を

(1) 移動元のホストにおいて、プロセスの状態や使用している資源の状態をファイル等に保存する。
(2) 移動先のホストにおいて、、新しくプロセスを生成し、保存された状態を復元する。

という方法で実現できる。

しかし、これらの仕組みはあらゆるプロセスを保存、復元ができるわけではない。例えば、プロセス ID は復元のために新しくプロセスを生成すると保存したものから変化してしまうといったように、保存、復元が困難な要素が OS に存在し、これが原因で正しく復元できないプログラムが存在する。また、サーバプログラムの場合、TCP/IP が移動に対応していないため、クライアントとの通信は移動すると切断されてしまうという問題がある。

2.2 分 散 OS

プロセスマイグレーション機能を持つ分散 OS である Mosix[3)] は、移動したプロセスのシステムコールは移動前のホストで実行されるのでチェックポイントシステムにあるような問題点は存在しない。しかし、システムコールが頻繁に実行されると移動前と移動後のホスト間での通信が多くなりパフォーマンスに問題がある。

2.3 仮想機械による移動システム

仮想機械を利用した移動システムとして VMotion[16)]、vMatrix[2)] 等が提案されている。これらは、仮想機械上で動作するゲスト OS の状態をイメージファイルに保存し、移動先のホストで仮想機械を復元することで移動を実現する。仮想機械はホスト環境によらず同一の環境を提供しプロセスを OS ごと移動させることが可能なので、どんなプロセスでも移動可能である。しかし、イメージファイルの容量はしばしば大きなものとなり、移動時にファイル転送のためにネットワークに負荷をかけ、移動時間を増大させるという問題がある。また、移動により仮想機械の位置が変化することでネットワーク接続は切断されてしまう。そのため VMotion には同一ネットワーク内でしか移動できないという制限があり、vMatrix は接続の維持については考慮していない。

2.4 移動に対応した TCP/IP の拡張

TCP/IP ネットワークはホストの移動に対応していないため、サーバの移動により IP アドレスが変化するとクライアントとの接続は切断されてしまう。この問題の解決法として、移動に対応した仮想ネットワーク層を入れる[14)]、トンネリングにより移動先にパケットを転送する[10)]、クライアントとの間にプロキシを入れる[7)] 等の方法が提案されている。これらは接続を維持したままサーバを移動するのに有用であるが、サーバプログラムの移動機構は含まれていない。そこで本研究ではこれらの拡張と組み合わせてサーバを移動させることを考える。

2.5 資源の仮想化

Zap[9] は、ファイルやネットワークといった移動の難しい資源を仮想化し、移動前後で変化しない仮想環境を実現している。その中でプロセスを動作することでプロセスマイグレーションによるサーバプログラムの移動を可能にした。しかし、ネットワークが完全に仮想化されていることを前提としているため、サーバ、クライアント双方が仮想化されたネットワークを使う必要がある。そのため、クライアント側のホスト全てに対して仮想化機構を導入しなければならないという欠点がある。本研究では、サーバ側の拡張のみで移動を可能にする方法を提案する。

3. 設　　計

本章では、前章で挙げたような性質を持つサーバプログラムの移動を OS の最低限の拡張で行う方法について述べる。必要な OS の拡張は、プロセスの状態保存機能の追加、プロセス ID やファイルの仮想化である。また、クライアントとの通信を移動後も継続するために、クライアントにはサーバが移動していないように見せる機構も必要である。

3.1 プロセスの保存、復元

プロセスの状態 (レジスタ、メモリ、使用中のファイルの状態等) の保存は OS を拡張してカーネル内で行い、復元はユーザプログラムで実現することとする。保存をカーネル内で行わなくてはならない理由は 3 つある。

1 つ目は既存の実行バイナリに手を加えずに状態を保存するためにはカーネルに保存機能を追加することが不可欠であるという理由である。OS カーネルを利用せずにプロセスの状態の保存を実現するには、保存対象のプロセスが自身の状態をを保存する機能を持つ必要がある。そのため、保存対象のプログラムソースの変換や実行ファイルにチェックポイント用ライブラリをリンクするといったことが必要となる。一方、カーネル内からは既存の実行バイナリに手を加えることなくプロセスの状態の保存が可能である。

2 つめは、プロセスの状態を全て取得するためである。プロセスの状態はユーザプログラムから全て見えているわけではない。例えば、パイプ、ソケット内の状態やメモリマップの状態をユーザプログラムから全て取得することは不可能であるため、カーネル内から取得する必要がある。一方、プロセスの状態は、大部分をユーザプログラムで復元可能である。プロセスの状態は、ユーザプログラムが命令を実行したりシステムコールを発行したりして変化させていくため、これらの操作で保存時の状態を復元することはほぼ可能であると言える。そのため、プロセスの状態保存のための OS 拡張は、保存機能のみでよい。ただし、プロセス ID のように復元が不可能な要素やネットワークのようにプロセス外と関係している要素の復元は OS の拡張によって行わざるをえない。これは後述する。

3 つめは、複数のプロセスの状態を保存したり、ファイルも同時に保存したりする時に互いの整合性をとるためである。これを実現するためには、カーネル内でユーザプロセスの動作を一時的に停止させて状態を取得しファイルを保存する必要がある。

復元の大部分をユーザモードで行うことで、OS の拡張を少なくすることができる。また、復元動作のカーネルに依存する部分を減らすことができ、異なるバージョンのカーネル上での再開が容易になる、再開時のユーザ権限内の動作しか許可されないため復元したプロセスに利用されたくない資源にアクセスされることを防ぐことができるといった利点もある。

3.2 ファイルの保存、仮想化

サーバプログラムはファイルに必要なデータを保持しているため、移動によりファイルシステムが変化することにより正しく動作しなくなる。そのため、サーバプログラムには移動前後で変化しないファイルシステムを見せる必要がある。これを OS の機能を利用して実現するには、chroot のような機能を利用すれば良い。すなわち、サーバの動作に必要なファイルを一時ディレクトリに展開し、そこをルートとする環境でサーバを動作させ、移動するときはそのディレクトリごと移動先ホストへ持っていくという方法が考えられる。しかし、サーバを動作させるための環境を一時ディレクトリに構築するのは手間がかかることや、構成するファイルが多くなりがちで移動のコストが大きいといった問題がある。

これを解決するために、移動後のホストに既に存在しサーバの動作中に変化しないファイル (ライブラリ、実行バイナリ等) は、chroot 環境の中にマップできるようにし、移動すべきファイルのみを chroot 環境に入れておくという方針をとる。そのためのマップ機能が OS 拡張として必要である。

また、ファイルをプロセスと同時に整合性を確保して移動させるため、プロセスが保存された時点のファイルを tar 等で 1 つのファイルにまとめて保存するときには、ファイルを保存している間は保存対象のプロセスがファイルに書き込みをさせてはならない。その機構も OS 拡張として必要である。

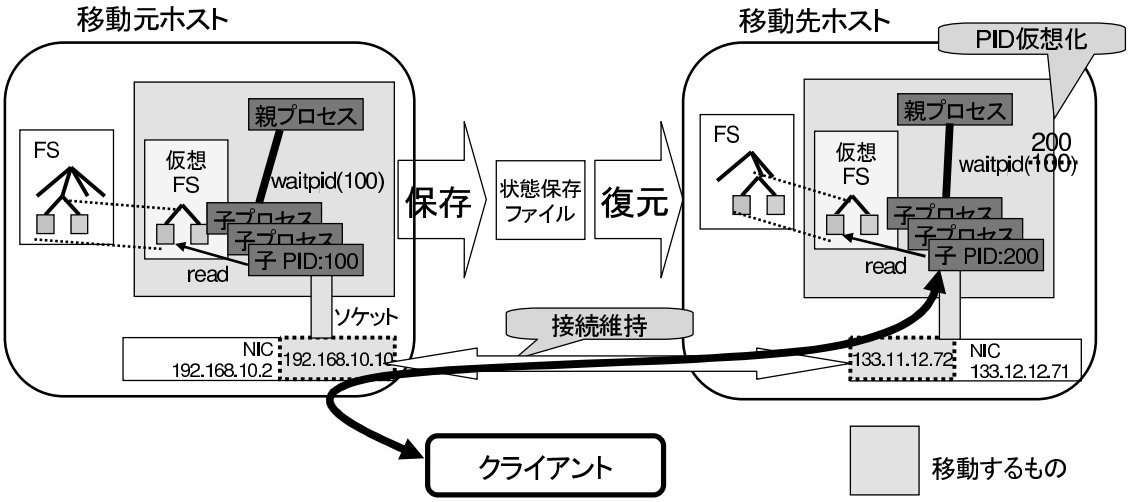

図 1 提案方式

3.3 プロセス ID の仮想化

多数のプロセスからなるプログラムがシグナル等で通信する場合、移動後にプロセス ID が変化すると正しく通信が行えなくなる。プロセス ID を復元する機能を OS に追加することもできるが、移動後のホストですでに動作していたプロセスと ID が重複していた場合復元は不可能である。そこで、移動後のプロセス ID は変化するものとし、移動前のプロセス ID がシステムコールで使用されたときに移動後の ID に変換してシステムコールハンドラに渡す OS 拡張が必要である。

3.4 TCP/IP ソケットの移動

TCP/IP ソケットの移動は、ソケットの状態の保存、復元とクライアントには移動を意識させず移動後も接続を維持する仕組みの 2 つに分けて実装するという方針をとる。前者は OS の拡張により実装する必要があるが、後者には 2.4 節で述べた様々な既存研究があり、状況に応じて異なるシステムと組み合わせることを可能にするためである。

ソケットの状態の保存、復元は既存の OS の機能では不可能であるので、OS の拡張が必要である。また、移動後も同じ IP を使い接続を維持する仕組みがいくつか提案されている。そこで、サーバには専用の IP を与えて移動前後で同じ IP を使わせるようにし、これらと組み合わせて接続を維持するという方針をとる。図 1 には、実装で採用した転送を使った接続の維持のイメージを記した。

4. 実　　装

我々は、Linux 2.4.27 上で本機構を実装した。OS の拡張はカーネルモジュールを用いて行った。

4.1 プロセス状態の保存・再開

プロセスの状態の保存はカーネルモードで行う。保存には CHPOX[15] というチェックポイントシステムを拡張したものを用いた。CHPOX を利用するのは比較的新しい Linux のカーネルに対応している、複数プロセスの同時保存が可能である、レジスタ、メモリ、ファイルディスクリプタ、パイプ、シグナルといったプロセスの多くの要素を保存できるといった利点があり、小さな拡張で Apache を始めとする様々なサーバを保存することが可能であるからである。我々は CHPOX を拡張し、TCP/IP ソケットの状態、プロセス ID、プロセスが使用しているがディレクトリツリー上からは削除されたファイルの保存機能を追加した。

再開はユーザモードで行う。我々は、CHPOX によって保存されたプロセスの状態を読み取り、保存したプロセスの状態を復元するユーザプログラムを実装した。メモリは mmap システムコール, ファイルは open, lseek, dup システムコールというように OS のシステムコールを利用して状態を復元するものである。ただし、ネットワークの接続状態は復元することができないため、カーネル内の機能拡張により実現した。詳しくは 4.3 節で述べる。

4.2 ファイル、PID の仮想化

ファイル空間、PID の仮想化は SoftwarePot[6] を

用いて行う。SoftwarePot は仮想ファイルシステムを持つサンドボックスであり、ソフトウェアをファイルシステムごと Pot ファイルと呼ばれるファイルに固めて配布し、サンドボックス (Pot 空間) 内に閉じ込めた状態で実行することができる。Pot 空間内の仮想ファイルシステムには 2 種類のファイルがある。1 つは、Pot ファイル内のファイルシステム上に実体があるファイル (static ファイル)、もう 1 つは、Pot 空間外、すなわち実行するホストのファイルシステム上に実体があり、実行時に Pot 空間内にマップされるファイル (map ファイル) である。これらのファイルへのアクセスは、システムコールの引数にファイルパスがある場合、Pot 空間内の仮想ファイルシステム上のパスから実体ファイルへのパスに書き換えることで実現される。この仮想ファイルシステムの機能により、移動対象のプロセスに移動前と移動後のファイルシステムが同一であるように見せることが実現された。

ファイルの保存は、現在の Pot 空間のファイルを Pot ファイルに保存する機能を拡張することで実現した。あらかじめ保存すべきディレクトリやファイルを指定しておき、それらに加えてプロセスの状態を保存したファイル、再開を行うプログラムの実行ファイルを保存する仕組みを実装した。また、プロセスの状態を保存するときに開いているファイルの flush を行い、ファイルの保存はプロセスの状態を保存した直後に、保存対象のプロセスの動作を停止させて行うことで整合性をとるようにした。

また、プロセス ID の仮想化は SoftwarePot のシステムコールの引数の書換え機能を拡張することで実現した。移動前の状態を復元するために移動後のホストでプロセスを生成したときに移動前と移動後の ID の対応付けを記録しておき、プロセス ID を引数に取るシステムコールが発行されたときに移動前のプロセス ID を使用していたら移動後のものに書換えるようにした。プロセス ID を返り値にとるシステムコールも同様の事を行うようにした。getpid, kill, wait, waitpid システムコールについて書換えを行うよう実装した。

4.3 ネットワーク

ソケットの状態の保存をするためには、TCP/IP の接続状態のうち最低限必要なもの (自身のアドレス、ポート、通信相手のアドレス、ポート、次に送受信するシーケンス番号) の保存とソケット内の送受信バッファの保存が必要である。これらはカーネル内からのみ取得できるので、プロセスの保存時にこれらを取得するような実装を行った。また、保存したソケットの状態の復元は、TCP_LISTEN 状態のソケットであればシステムコールのみで復元可能であるが TCP_ESTABLISHED 状態のソケットを復元するためには、自身のポート、次に送受信するシーケンス番号、ソケット内のバッファといったソケット内のデータの書き換えが必要である。そこで、ソケットとこれらのデータを OS に与えると書き換えが実行される仕組みを実装した。

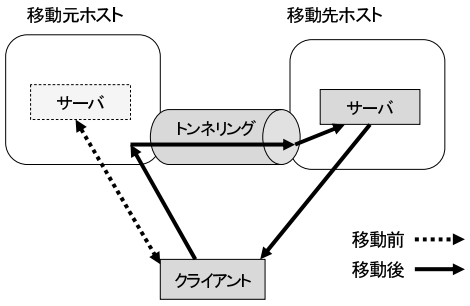

図 2 転送による接続維持

クライアントに移動を意識させず移動後も接続を維持する仕組みは、今回は MobileIP[10] で使用されている IPIP トンネリング[13] によるパケットの転送を用いて実装した。まず、サーバ毎に専用の IP アドレスを割り振っておき、移動前後で同一 IP をサーバのアドレスとして利用できるようにしておく。移動前のホストに送られてきたパケットは移動後のホストに転送され、図 2 のような経路をたどり通信が継続される。

4.4 移動全体の流れ

保存、復元の大まかな流れは以下の通りである。まず、保存対象のプロセスからのネットワーク通信を一時的に遮断する。次に CHPOX に保存対象のプロセスの ID を通知し、そのプロセスに特定のシグナルを送ると CHPOX によって現在のプロセスの状態がファイルに保存される。その後 CHPOX は SoftwarePot に保存完了の通知を行い、通知された SoftwarePot は現在の Pot 空間の状態を Pot ファイルに保存する。また、CHPOX がファイルの状態を記録するとき、Pot 空間ではなく実体のファイルパスを記録するので、CHPOX が保存したファイルパスと Pot 空間内のファイルパスとの対応も保存する。次に、移動元ホストから移動先ホストへのパケットの転送を開始する。最後に、移動後のホストで保存された Pot ファイルを実行することでサーバプロセスの復元を行う。復元を行うプログラムが保存された Pot ファイルに含まれており、その Pot ファイルを実行すると保存されたファイルシステムが復元され、Pot 空間内で復元を行うプログラムが最初に実行され、復元が完了する。

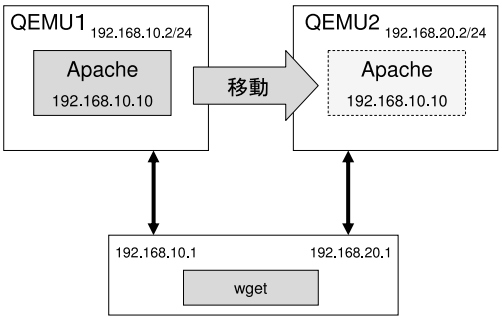

図 3 実験環境

5. 実　　験

我々は、本機構の有用性を確認するための予備実験を行った。CPU エミュレータを用いた仮想ネットワーク環境を構築し、Web サーバ Apache の保存、復元にかかるだいたいの時間と、移動すべきファイルの大きさについて測定した。実験環境として、CPU エミュレータ QEMU[4] を用いて Linux の実環境上に 2 つの仮想 Linux 環境を用意し、各々を別の仮想ネットワークに配置した (図 3)。実験に用いた Apache はバージョンが 1.3.33 のものである。

本実験では、一方の仮想 Linux 環境で Apache を動作させて実環境のクライアントから接続し、ファイル転送中に Apache を他方の仮想 Linux 環境へ移動することを行った。その結果、移動前後で接続は切断されずファイルの転送は継続されたものの、スループットが半分程度に落ちてしまった。その原因は接続の復元時に MSS(max segment size) が小さくなってしまうことであった。また、図 2 のようにパケットが転送されていることを tcpdump を用いて確認した。

本実験において、保存、復元が必要なプロセスは 8 個であった。保存時間の正確な測定は行っていないものの、これらのプロセスのの保存、復元にかかる時間はそれぞれ 1 秒以下であり、保存、復元動作においては、移動に必要なファイルの保存、復元にかかる時間が大部分であることが分かった。

また、Apache を移動させる際に移動すべきファイルの大きさも測定した。移動に必要なのは、サーバプロセスの状態を保存したファイル、実行バイナリやプロセスが使用しているライブラリ、HTML ファイル等のデータファイルである。データファイルは、用途に応じて大きさが変化するためここでは考えない。また、実行バイナリやプロセスがマップしているライブラリは移動先に同一のファイルがある場合は移動させる必要がなく、プロセスの状態のみを移動させればよいのでプロセス状態記録ファイルの大きさも測定した。

その結果は、Apache のプロセス状態記録ファイルの容量はおよそ 7 メガバイトであり、それに実行バイナリと使用しているライブラリを加えるとおよそ 10 メガバイトであった。

6. 論　　議

プロセスを移動させる際に問題となるのは、移動にかかる時間と移動機構の導入によるパフォーマンスの低下である。

移動にかかる時間は、保存、復元の時間とホスト間でのデータ転送の時間である。本研究において、保存、復元の時間は必要なファイルを 1 ファイルにまとめる時間が大部分を占めると考えられる。これは移動に必要なファイルの容量に比例する。

また、ホスト間でのデータ転送時間も移動に必要なファイルの容量に比例するが Apache を移動させる際の通信量は 10 メガバイトにデータファイルの容量を足したものであり、データファイルの容量が小さければ転送時間はあまり問題にならないと考えられる。一方、仮想機械を用いた移動システムでは、ディスクを含めた VM イメージを移動するため数百メガバイトから数ギガバイトになることがあり、分割転送等の仕組みが必要となる。

移動機構の導入によるパフォーマンスの低下として考えられるのが、ファイルや PID の仮想化のためのシステムコール引数書き換えによるプロセス動作に対するオーバヘッドと、移動後にパケット転送を行うことによるネットワーク通信に対するオーバヘッドである。前者に関しては参考文献によると 6 から 20%程度に抑えられている。後者に関しては実環境での性能評価をしていないが、レイテンシの増大やトンネリングのオーバヘッドが問題になると思われる。

本研究と類似したシステムとして Zap があり、Zap は本システムと同様に OS の拡張によってサーバを移動可能にしている。しかし、TCP/IP を独自拡張した仮想ネットワークを利用しているため、サーバだけでなくクライアント側にも同一の OS 拡張が必要であり、導入の負担が大きいといった問題がある。

7. まとめと今後の課題

本研究では、OS の最低限の拡張によってサーバを透過的に移動する方式を提案し、Linux 上での実装を示た。また、Apache を移動させる予備実験を行い、仮想機械を使った方式と比較して移動すべき容量が小

さいことを確認した。

現在はCPUエミュレータを用いた予備実験しか行っていないので、実環境でのパフォーマンス測定をする必要がある。また、スループットが落ちた原因である接続の復元時にMSSが小さくなる問題を改善しなくてはならない。MSS値は接続の開始時に決定されるので、移動に関係するホスト間で共通して使うことができる最適なMSS値を決定しておくことも必要になると考えられる。

現在の実装では、移動時に全ファイルをまとめて移動しているため移動の終了まで再開が行えず、移動によるサーバの停止時間が長くなるという問題がある。この改善策として、再開に最低限必要なファイルのみを先に移動することや、あらかじめ投機的にファイルを移動しておき、その後プロセス移動の時にはファイルへの書き込みにより変化した差分のみを移動するといった改善策が考えられる。

本研究ではネットワーク接続の移動をトンネリングによるパケット転送で実装したが、この方法にはオーバヘッドやセキュリティの問題が存在する。また、移動前のホストも転送にかかわるので、シャットダウンすることができない。これらを解決するために、この手法の改善となるMobileIPv6[5]や他の接続の移動手法を組み合わせ、評価を行うことも考えている。

参 考 文 献

1) Apache Software Foundation: Apache. http://www.apache.org/.
2) Awadallah, A. A. and Rosenblum, M.: The vMatrix: Server Switching, *IEEE 10th International Workshop on Future Trends in Distributed Computing Systems* (2004).
3) Barak, A. and Wheeler, R.: MOSIX: How Linux Clusters Solve Real World Problems, *An Integrated Multiprocessor UNIX, Proceedings of the USENIX Winter 1989 Technical Conference* (1989).
4) Bellard, F.: QEMU. http://fabrice.bellard.free.fr/qemu/.
5) Johnson, D., Perkins, C. and Arkko, J.: Mobility Support in IPv6, RFC 3775 (2004).
6) Kato, K. and Oyama, Y.: SoftwarePot: An Encapsulated Transferable File System for Secure Software Circulation, *Software Security – Theories and Systems, volume 2609 of Lecture Notes in Computer Science*, pp. 112–132 (2003).
7) Maltz, D. A. and Bhagwat, P.: MSOCKS: An Architecture for Transport Layer Mobility, *In Proceedings of the 17th Annual Joint Conference of the IEEE Computer and Communications Societies* (1998).
8) Milojíčić, D. S., Douglis, F., Paindaveine, Y., Wheeler, R. and Zhou, S.: Process migration, *ACM Computing Surveys*, Vol. 32, No. 3, pp. 241–299 (2000).
9) Osman, S., Subhraveti, D., Su, G. and Nieh, J.: The Design and Implementation of Zap: A System for Migrating Computing Environments, *5th Symposium on Operating Systems Design and Implementation*, pp. 361–376 (2002).
10) Perkins, C.: IP Mobility Support for IPv4, RFC 3344 (2002).
11) Plank, J. S., Beck, M., Kingsley, G. and Li, K.: Libckpt: Transparent Checkpointing under Unix, *Usenix Winter Technical Conference*, pp. 213–223 (1995).
12) Ramkumar, B. and Strumpen, V.: Portable Checkpointing for Heterogenous Architectures, *Symposium on Fault-Tolerant Computing*, pp. 58–67 (1997).
13) Simpson, W.: IP in IP Tunneling, RFC 1853 (1995).
14) Su, G. and Nieh, J.: Mobile Communication with Virtual Network Address Translation, Technical Report CUCS-003-02, Department of Computer Science, Columbia University (2001).
15) Sudakov, O.O. and Meshcheryakov, E.S.: CHPOX - CHeckPOinter for linuX. http://www.cluster.kiev.ua/tasks/chpx.html.
16) VMware Inc.: VMware, VMotion. http://www.vmware.com/.
17) Zandy, V. C.: ckpt: A process checkpoint library. http://www.cs.wisc.edu/ zandy/ckpt/.
18) Zhong, H. and Nieh, J.: CRAK: Linux Checkpoint / Restart As a Kernel Module, Technical Report CUCS-014-01, Department of Computer Science, Columbia University (2001).