# 通信を行うプロセスの移送機能の設計と実装

白木原 敏雄† 金井 達徳†

プロセス移送は分散処理環境において，負荷分散や計算機停止時のプロセスの実行の継続を可能にする．従来，プロセス間通信を行うプロセスの移送は，移送するプロセスのみでなく通信先のプロセスにも影響を与えるため困難であった．本論文では，UNIXで提供しているソケットを用いて通信を行うプロセスの移送をユーザレベルで実現する方法について述べる．本論文のプロセス移送システムはプロセス移送を管理する移送サーバおよびユーザプログラムにリンクされる移送ライブラリからなり，移送時には，これらが協調してプロセスの実行状態の保存・回復を行う．この時，ソケット通信路の状態の保存・回復を行うために，接続指向通信路の場合には，移送サーバは移送前のソケットの代わりに新しく接続したソケットを生成し，それをユーザプロセスの移送ライブラリに渡す．また非接続指向通信路の場合には，移送サーバが移送前のソケット宛のメッセージを移送先に転送する．移送ライブラリは UNIX ソケットと同じインタフェースを持ち，通常の通信時にはソケットの状態の変化を調べるのみで，実際の通信は通常の UNIX のソケット機能を使用する．ユーザプログラムはソースコードを変更せずに移送ライブラリをリンクするのみで移送可能になる．

# Design and Implementation of a Migration System for Communicating Processes outside the UNIX Kernel

Toshio Shirakihara† and Tatsunori Kanai†

In a distributed computing environment, process migration is a useful mechanism for load balancing and non-stop computing. However, it is difficult to migrate a process which communicates with other processes without affecting peer-processes. In this paper, we describe a newly developed migration system for processes which communicate by UNIX sockets. This system is composed of migration servers and migration libraries. When a communicating process is migrated, the migration servers and the process's migration library save and restore the status information of sockets. In case of connected sockets, the migration servers create new connected sockets and deliver them to migrated process and peer-process. Concerning connectionless sockets, the migration servers receive messages addressed to old sockets and forward them to new sockets of migrated processes. Migration libraries have same interface with UNIX socket and conceal implementation details of these mechanisms from user programs. This system is implemented outside the UNIX kernel and any user process can be migrated without rewriting source programs.

## 1. はじめに

近年，UNIX ワークステーション等をネットワークで接続した環境での分散処理が広く行われている．このような環境において，ある計算機にはプロセスが多数存在するために負荷が重く，別の計算機ではその計算機資源がほとんど使用されていない状態が頻繁に起こる．また，ある計算機が保守等で停止する場合には，その計算機上のプロセスが他の計算機に対して行っているサービスが停止してしまう．

† (株)東芝 研究開発センター 情報・通信システム研究所 第二研究所
Research Lab. II, Communication and Information Systems Research Laboratories, Research and Development Center, Toshiba Corporation

プロセス移送 (Process Migration)[1] とは，ある計算機上で実行中のプロセスを，現在の実行状態を保存して他の計算機上に移送し，実行を継続する技術である．上記の問題に対してプロセス移送を用いることにより，以下のように計算機資源の有効利用を計ることができる．

- 負荷の大きい計算機上で動作しているプロセスを他の計算機へ移送することにより，分散システム全体の負荷を均衡させる．
- 定期保守や障害のために計算機が停止する場合，その計算機上で動作しているプロセスを他の計算機へ移送することにより，そのプロセスが行っているサービスを継続できる．

プロセス移送の実現方法は以下の2つに分類でき

る．

1. オペレーティングシステム (OS) のカーネルでサポートする方法[2)~5)]
2. OS カーネルを変更せずにユーザレベルでサポートする方法[6)~8)]

1はプロセスの実行状態の保存・回復が容易に行えるが，実現方法が特定の OS に依存する．一方，2は OS カーネルの変更を必要としないため，既存の OS 上で実現することができ，また1に較べて移植性も高い．現在広く使われている UNIX[9),10)] のカーネルはプロセス移送をサポートしていない．そのため標準的な UNIX においてプロセス移送による負荷分散や計算機停止時のプロセスのサービスの継続を行うためには，2の方法に基づいてユーザレベルでプロセス移送をサポートする必要がある．

ユーザレベルでプロセス移送を実現する場合，ベースとなる OS はプロセスの実行状態をユーザレベルで保存・回復するための機能を提供していないことが多いため，移送できるプロセスには入出力等に関する制約があった．従来の研究では，通常ファイルへの入出力を行うプロセスの移送を行うために，ファイルのオープン時にそのファイル名を保存し，移送時にはファイル内のアクセス位置を保存する．移送後には保存したファイル名を元に再オープンし，アクセス位置を回復することにより移送後のファイルへのアクセスを可能にしている[8)]．

プロセス間通信は通常ファイルに対する入出力と異なり，同一計算機上あるいはネットワークで接続された計算機上の他のプロセスに対する入出力である．そのためプロセス間通信を行うプロセスの移送は，複数のプロセスの実行状態を保存・回復の対象にする必要があり，従来のプロセス移送方式では困難であった．

本論文ではこの問題を解決し，プロセス間通信を行うプロセスの移送を UNIX ユーザレベルで実現する方法を提案する．プロセス間通信には接続指向の通信路を使用するものと非接続指向の通信路を使用するものがあり，本方式では各々の場合に対して以下の方法により通信路の保存・回復を可能にする．

- 接続指向の通信路はプロセス移送とともに切断されるため，プロセス移送システムが新たに接続した通信路を生成し，それをユーザプロセスに渡す．
- 非接続指向の通信は通信ごとに通信先が指定されるため，既に移送されたプロセスへのメッセージはプロセス移送システムが移送先に転送する．

この結果，プロセスは移送された後も移送前と同様にプロセス間通信を行うことができる．プロセスが使用するプロセス間通信の手段としては，標準的な UNIX でサポートされもっとも広く使われているソケット[11)]を対象とした．本プロセス移送システムはソケットと同一のインタフェースをユーザプログラムに提供し，その内部で通信路の保存・回復のための処理をユーザプログラムから隠蔽して行うため，通信を行うプロセスをソースコードの変更なしに移送できる．

本論文の構成は次のとおりである．第2章ではユーザレベルでのプロセス移送方式について述べ，第3章ではソケットの実行状態について説明する．第4章，第5章では接続指向および非接続指向ソケットの保存・回復方式について述べる．第6章では実装法および性能について説明する．

## 2. プロセス移送方式

本プロセス移送機能の実行状態の保存・回復方式は，森山ら[8)]と同様の方式を採用し，それに対して新たにソケットを用いた通信に関係する実行状態の保存・回復の機能を拡張したものである．ソケットには UNIX ドメインのものと INET ドメインのものがあるが，本プロセス移送機能では異なる計算機間の通信と同一計算機上の通信の両方に用いることができる INET ドメインのソケットを対象とした．ソケットの通信プロトコルは接続指向の TCP プロトコルと非接続指向の UDP プロトコル[12)]を対象とした．

ソケットの入出力には通常入出力と非閉塞入出力[9),10)]がある．通常入出力はソケット層およびプロトコル層の処理が終了した時点でプロセスに処理が戻る．一方非閉塞入出力では，ソケットによるデータ受信および接続受付時にデータや接続要求がない場合もプロセスを休止させず処理を戻す．また，データ送信および接続要求時にはソケット層での処理が終了後，プロトコル層に要求を渡す時点で処理をプロセスに戻す．そのため，プロセスに処理が戻った後もプロトコル層での処理が継続され，後述するようにソケット実行状態のユーザレベルでの保存が困難になる．本システムではカーネル内部のソケット実行状態の保存・回復を容易にするために非閉塞入出力については対象外とした．

**図1**に本システムの概要を示す．プロセス移送システムはユーザプログラムにリンクされる移送ライブラ

リと，各計算機上でプロセス移送を管理する移送サーバ migd からなる．計算機1から計算機2にプロセスを移送する場合下記の手順で処理を行う．

1. migd A は移送するプロセスAにシグナルを送る．
2. シグナルを受けたプロセスAは，移送ライブラリ内のシグナル処理ルーチンでプロセスの実行状態を獲得し，それを実行可能な形式でファイルに保存する．
3. 移送ライブラリは実行状態を保存した保存ファイル名を migd A に通知する．
4. migd A は移送先の移送サーバ migd B にプロセスを移送することを通知する．
5. migd B は通知された保存ファイルを実行するプロセス A′ を生成する．なお，保存ファイルはネットワークファイルシステム[13]等を介して参照する．
6. 移送されたプロセス A′ は手順2で獲得・保存した実行状態を回復し実行を再開する．

上記の移送処理において移送ライブラリと migd の行う処理は以下のようになる．

**実行開始時および実行再開時の処理**

移送ライブラリは実行開始時および移送後の再開時に migd に対して実行を開始することを通知し，migd はそのユーザプロセスをテーブルに登録する．その後，実行開始時にはユーザプログラムの main 関数を呼び出し，実行再開時にはプロセスの実行状態を回復する処理を行う．

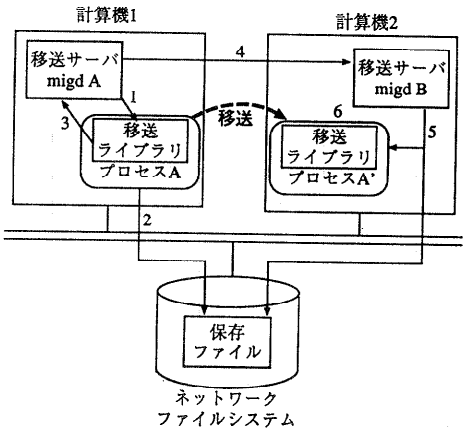



図 1 プロセス移送システムの概要
Fig. 1 Overview of the process migration system.

**実行中の処理**

移送ライブラリは通常ファイルのオープン・クローズを行う専用の関数を提供する．オープン時には，移送後の再オープンのためにそのファイル名を保存した後，通常の open システムコールを実行する．クローズ時には保存したファイル名の情報を消去し，通常の close システムコールを実行する．これらの関数は，UNIX で提供されている関数と同様のインタフェースを持つ．ソケットインタフェース関数についても移送のために状態を保存するが，これらの詳細は後述する．

**プロセス移送時の処理**

移送ライブラリはハードウェアレジスタおよびシグナルマスク，現在のディレクトリ，シグナル処理関数，ファイルのアクセス位置をプロセスのデータ領域に保存し，テキスト，データ，スタック領域の内容を実行可能な形式でファイルに格納する．移送後は migd が保存ファイルを fork/exec するため，テキストおよびデータ領域はシステムにより回復される．スタック領域などの情報は移送ライブラリが実行開始時に回復する．

## 3. ソケットの実行状態

UNIX ソケットの実行状態は，ユーザ空間に保持されるファイルディスクリプタとカーネル内のソケット層およびプロトコル層の状態からなる[10]．ソケットはそれを保持するプロセスおよび通信先プロセスからの入出力を受けるため，カーネル内の状態はさらに以下の2つに分類できる．

**(1) システムコールによって変化する状態**

ソケットはシステムコールによりその状態が変化する．例えば，connect システムコールにより未接続の状態のソケットは接続済みの状態に変化する．

**(2) 他のプロセスからの通信により変化する状態**

他のプロセスからメッセージが送られてきた場合，それらはカーネル内のバッファに保存される．また，他のプロセスが接続要求を行った場合，それらに対応した新しいソケットがカーネル内のキューとして管理される．

ソケットを持つプロセスをユーザレベルで移送する場合，(1)(2)に示したカーネル内の状態を移送システムで保存・回復する必要がある．

(1)に示したように，システムコールによって状態が変化する過程ではさまざまな途中状態を経る．例えば，TCP プロトコルでは write システムコールによるデータの送信時にパケットを送信し，それに対する応答を待つ応答待ち状態が存在する[11]．システムコールが通常入出力の場合にはプロトコル層の処理が完了した時点でプロセスに処理が戻るため，このような途中状態は考慮する必要がなくユーザレベルでソケットの状態が容易に把握できる．しかし，非閉塞入出力の場合にはプロトコル層に要求が渡る時点で処理が戻り，プロセス実行中にプロトコル層の状態が変化することがあるため，ソケットの状態をユーザレベルで把握することが困難になる．本システムでは非閉塞入出力は対象外とし，システムコールにより変化したソケットの状態（**ソケット状態**）を移送ライブラリ内に保存する．

(2)の他のプロセスからの通信により変化する状態については，通常の実行時にはそのソケットを持つプロセスによって(2)のバッファやキューがクリアされるため考慮する必要はない．例えば，カーネル内に保存されたメッセージは read システムコール等による読み出しによりクリアされる．しかし，移送開始時およびプロセス移送中には(2)のバッファやキューにデータが存在する可能性があるため（**移送中の通信**），これらについても保存・回復を行う必要がある．

以上のことにより本システムではソケットの実行状態のうち，**ファイルディスクリプタ**，**ソケット状態**，**移送中の通信**についての情報を保存・回復の対象とする．

## 4. 接続指向通信路の保存・回復

接続指向のソケットの状態は，接続済みの状態と未接続の状態に分類できる．接続済みとは2つのプロセスがそれぞれ accept, connect システムコールにより双方のソケットを接続した状態であり，それ以前の状態は未接続である．接続済みソケットはプロセス移送時にクローズされるため，本システムでは移送後に新しく生成したソケットを接続した状態に回復する．未接続のソケットはさらに接続を受け付ける接続待ちソケットと，他のプロセスに対して接続を要求する接続要求ソケットに分類できる．接続待ちのソケットを持つプロセスを移送する場合，そのソケットに接続しようとするプロセスは移送前のソケットアドレスを指定するため，そのソケットアドレスで接続を行えるような状態に回復する必要がある．接続要求ソケットについてはソケットアドレスが変化しても接続には問題ないため，ファイルディスクリプタのみを保存する．本章では接続済みおよび接続待ちソケットの状態の保存・回復処理の方式を示す．

### 4.1 接続済みソケット

接続済みのソケットを持つプロセスを移送する場合，移送先の migd と通信先のプロセスが存在する計算機上の migd 間で新たにソケットを接続し，それを移送後のプロセスおよび通信先のプロセスに渡すことによりソケットの状態の保存・回復を行う☆．この処理は，ソケットの実行状態の保存，移送プロセス側のソケットの実行状態の回復，通信先プロセス側のソケットの実行状態の回復の3つの処理にわかれる．以降ではこれらの処理について詳細に説明する．

**図2**は計算機1上で動作しているプロセスBを計算機3に移送する場合の接続済みソケットの保存・回復の手順を示したものである．ここでは，プロセスBは移送前にソケット$\beta$を計算機2上のプロセスAのソケット$\alpha$と接続していたとする．各計算機上にはプロセス移送を管理する migd およびソケットアドレスの登録・参照が行われる共有メモリが存在する．共有メモリは，

(addr 1, addr 2, newfd)

の組になった情報を管理するテーブルで，addr 1, addr 2 は移送前に接続されていたソケット対のソケットアドレス，newfd は移送後に新たに接続されたソケットのファイルディスクリプタである．各プロセスの移送ライブラリは通信を行う前に共有メモリを調べる．この時，使用するソケットおよび接続先のアドレスが共有メモリに登録されていれば，通信先のプロセスが移送されたかもしくは移送中と判断し，移送中ならばそれが完了するまで通信を保留する．移送が完了したかどうかは管理テーブルの newfd が登録されたかどうかで判断する．

**(a) ソケットの実行状態の保存**

図2(a)はプロセスBが計算機3に移送され，プロセス B′ になった状態である．プロセスBを移送する前に移送ライブラリは接続済みのソケット $\beta$ と接続先のソケット $\alpha$ のソケットアドレスを計算機1上の migd に通知する．これを受けた migd は，ソケット $\alpha$ のソケットアドレスから計算機2のネットワークア

☆ UNIX では sendmsg, recvmsg システムコールにより，同一計算機上のプロセス間でソケットなどのファイルディスクリプタの受渡しを行うことができる．

ドレスを知り，計算機2上の migd にこのソケットアドレス対の通信路が切断されることを通知する．計算機2上の migd はソケットアドレス対 $(\alpha, \beta)$ を共有メモリに登録することにより，接続先のソケット $\alpha$ による通信を抑制する．その後，プロセスBの移送ライブラリはソケット $\beta$ 宛のメッセージでカーネル内に保持されているものを読み出し，データ空間のメッセージバッファに保存する．

**(b) 移送プロセス側のソケットの実行状態の回復**

図2(b)は移送後のプロセス B′ が実行開始時にソケットの接続を回復するための処理を示している．この時，プロセス B′ の移送ライブラリは移送前のソケットアドレス対 $(\alpha, \beta)$ を migd に知らせる．migd は計算機2上の migd と新しくソケット $\beta'$ を接続し，それをプロセス B′ に渡す．計算機2上の migd は計算機3上の migd からの接続を受け付け，接続したソケット $\alpha'$ のファイルディスクリプタを共有メモリ上のソケットアドレス対 $(\alpha, \beta)$ に対応した newfd として登録する．これらの処理によりプロセス B′ はソケット $\beta$ のかわりにソケット $\beta'$ を使用して通信を続けることができる．なお，ソケット $\beta'$ からの読み出しを行う時，メッセージバッファにメッセージが残っている場合にはそこから読み出しを行う．

**(c) 通信先プロセス側のソケットの実行状態の回復**

図2(c)はプロセスAがソケット $\alpha$ を使用してプロセスBと通信を行おうとした状態を示したものである．プロセスAの移送ライブラリは通信を行う前に，そのソケットおよび接続先のソケットのアドレスが共有メモリに登録されているかをチェックする．登録されていなければ通常の通信を行う．登録されていれば通信相手が移送されたことを示すので，計算機2上の migd に対して新しいソケットを要求し獲得する．この時，ソケットアドレス対 $(\alpha, \beta)$ に対する新しいソケット $\alpha'$ のファイルディスクリプタが登録されていない場合には通信相手が移送中であるため，登録されるまで保留する．

これらの処理により，プロセス A，B′ はソケット $\alpha$，$\beta$ にかわる新しいソケット $\alpha'$，$\beta'$ を使用して通信を続けることができる．なお移送ライブラリは，migd から獲得したソケットのファイルディスクリプタを dup2 システムコールにより元のファイルディスクリプタに変換するため，各プロセスは同じファイルディスクリプタを用いて通信することができる．

### 4.2 接続待ちソケット

他のプロセスが保持する接続待ちのソケットに接続を行いたいプロセスは，接続要求時に相手のソケットアドレスを指定する．プロセスを移送した場合，そのプロセスが持っている接続待ちソケットのソケットアドレスは変化してしまい，移送前後のソケットアドレスをユーザレベルで同一にすることは不可能である．そのため，本システムでは接続待ちソケットを持つプロセスが移送された後も，移送前のソケットアドレスに対する接続要求を受け付けることを可能にする．

**図3**は，計算機2上のプロセスAが，ソケット $\alpha$ を既に移送されたプロセスBのソケット $\beta$ に接続しようとした状態を示している．接続待ちのソケット $\beta$ を持っているプロセスBの移送ライブラリは，移送前にソケット $\beta$ をクローズすることを計算機1上の migd に通知する．これを受けた migd はこのアドレスを migd 内のテーブルに保存する．このアドレスは移送後新し

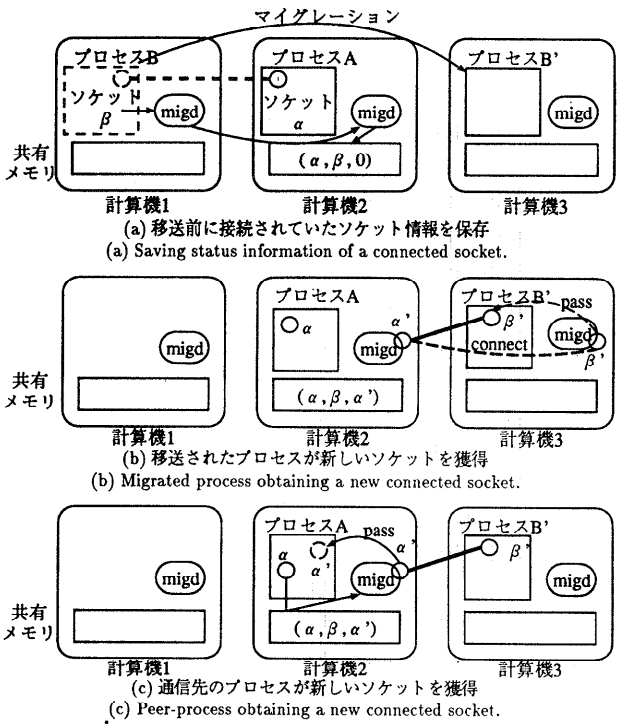



**図2** 接続済みソケットの移送処理
Fig. 2 Migration procedure of a connected socket process.

いソケットが登録されるまで保持する．移送後，プロセスB′の移送ライブラリはソケットβに対応したソケットβ′を作成し，ソケットβ,β′のアドレスを計算機3上のmigdに通知する．これを受けたmigdはこのアドレス対をmigd内のテーブルに保存する．

プロセスAがプロセスBの持っていたソケットβに接続しようとした場合，既にソケットβは存在しないため接続は失敗する．このとき，ソケットβをもつプロセスが移送された可能性があるため，プロセスAの移送ライブラリはソケットβの状態を同一計算機上のmigdに問い合わせる．これに対してmigdはすべてのmigdにソケットβのアドレスが変化しているかを問い合わせる．図3の場合には，計算機3上のmigdがソケットβの変化の状態を保持しているので，それを計算機2上のmigdに返す．計算機2上のmigdはソケットβがソケットβ′に変化したことを知り，それをプロセスAの移送ライブラリに返す．プロセスAの移送ライブラリは新たにソケットβ′に対して接続を行う．また，プロセスBが移送中の場合は計算機1上のmigdが移送中であることをプロセスAに通知し，プロセスAは移送が完了するまで接続処理を保留する．

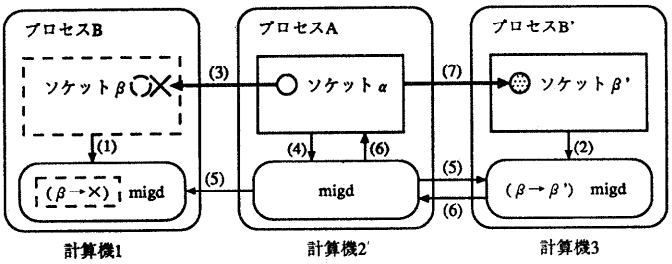
図3ではプロセスBの持つ接続待ちのソケットが移送のためにクローズされた後に，プロセスAがconnectシステムコールによって接続を要求した場合の処理の様子を示した．しかしこのような場合以外に，プロセスBが移送される前にプロセスAがconnectシステムコールを実行して接続を要求したが，プロセスBがそれに対するacceptシステムコールをまだ行っていない状態で移送される場合が考えられる．この場合計算機1のカーネル内部にはプロセスAのconnectシステムコールにより接続したソケットの情報が存在する．プロセスBがacceptシステムコールを行っていない時点でソケットβをクローズするとこれらの情報は放棄されてしまう．そのため本システムではこのような状態のソケットは保存できない．

## 5. 非接続指向通信路の保存・回復

非接続指向のソケット通信は，接続指向の通信とは異なり接続を確立しないため，サーバプロセスは任意の時点で任意のプロセスからメッセージを受ける．ソケットアドレスはOSによって与えられるため，移送前のソケットと移送後に再オープンしたソケットのアドレスは異なる．そのため，移送前のプロセスのソケットアドレスを指定して送られたメッセージは移送後のプロセスに届かないという問題が生じる．

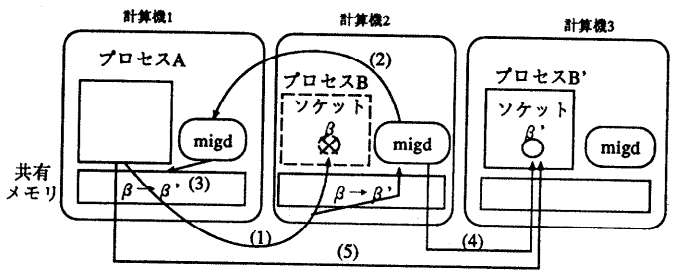
本システムでは，移送されたプロセスへのメッセージをネットワークドライバから直接読み出し，移送先へ転送することにより，上記の問題を解決し非接続指向のソケット通信の実行状態の保存・回復を行う．

図4に非接続指向のソケット通信の場合の処理の様子を示す．ここではソケットβを持つプロセスBが計




(1) βのクローズを通知
(2) β→β'を登録
(3) connectシステムコール失敗
(4) migdに問い合わせ
(5)他のmigdに問い合わせ
(6) β→β'を通知
(7) β'にconnect

図3 接続待ちソケットの移送処理
Fig. 3 Migration procedure of a connection-waiting socket process.




(1) β宛にメッセージを送信
(2) β→β'を通知
(3) β→β'を登録
(4) β'宛にメッセージを転送
(5)2回目以降はβ'宛に通信

図4 非接続指向ソケットの移送処理
Fig. 4 Migration procedure of a connectionless socket process.

算機2から計算機3へ移送されたとする．接続済みソケットの場合と同様に，移送時にはソケットβ宛のメッセージでカーネル内に保持されているものを読み出してデータ空間のメッセージバッファに保存する．プロセスBは移送後プロセスB′になり，ソケットβに対応したソケットβ′を作成する．その後，プロセスB′はソケットβ′のソケットアドレスをソケットβのアドレスと共に計算機3上のmigdに通知する．計算機3上のmigdはそれを計算機2上のmigdに通知し，計算機2上のmigdはそれを共有メモリに登録する．このように，そのソケットが最初にアドレスを獲得したノード（ホームノード）のmigdは移送に伴うソケットβのソケットアドレスの変化を管理する．これは，ソケットβ宛のメッセージが必ずそのホームノードに送られるからである．図4の場合，ソケットβのホームノードは計算機2である．

このような状態でプロセスAがソケットβ宛のメッセージを送った場合，プロセスBはすでに計算機2上に存在しないため，計算機2上のmigdがネットワークドライバからそのメッセージを獲得し，共有メモリを参照して移送先のソケットアドレスが登録されていれば，メッセージを移送先へ転送する．図4の場合にはソケットアドレスβ′が登録されているため，ソケットβ宛のメッセージはmigdによりソケットβ′宛に転送される．また，プロセスBの移送処理中にソケットβ宛のメッセージが送られてきた場合，ホームノードのmigdがそのメッセージを保存し，新しいソケットアドレスが通知された時点で移送先に転送する．

また，2回目以降の通信からmigdの転送機構を使用せず直接メッセージを送信するために，計算機2上のmigdはメッセージを転送する前に計算機1上のmigdにソケットアドレスの変化を通知する．計算機1上のmigdは通知されたソケットアドレスの変化を共有メモリに登録する．プロセスAの移送ライブラリは，非接続指向のソケットを使用した通信を行う前に送り先のソケットアドレスの変化が共有メモリに登録されていないかをチェックし，登録されている場合には変化後のソケットアドレスにメッセージを送る．これにより，1回目のメッセージはmigdにより転送されるが，2回目以降はmigdの転送機構を使用することなく直接移送先へメッセージを送信できる．

この方法によってメッセージが転送できるが，上記の方法ではソケットアドレスの変化を常にそのホームノードで管理しているため，その計算機が保守や障害等で停止する場合にメッセージの転送を行えなくなる．この場合，本プロセス移送システムでは，ホームノード上のmigdにかわって，任意の計算機上のmigdが転送を行う．これを実現するために，どの計算機宛のメッセージかに関わらずネットワーク上に流れるすべてのパケットを拾うネットワークドライバの機能（promiscuous機能）を使用する．

**図5**にホームノードが停止した場合のメッセージの転送方法を示す．図5は図4においてソケットβのホームノードである計算機2が停止した状態である．図では計算機2のmigdにかわって，ホームノードを失ったソケットを持つプロセスが存在する計算機3上のmigdが転送を行う．計算機3上のmigdはソケットβのアドレスの変化を管理し，β宛のメッセージを監視する．この状態でプロセスAが計算機2上にあったプロセスBのソケットβ宛に出したメッセージは，計算機3上のmigdによって獲得される．計算機3上のmigdは新しいソケットアドレスβ′とメッセージの内容をそのメッセージの発信元である計算機1上のmigdに送ることにより再送を要求する．計算機1上のmigdは再送要求を受けるとソケットβ′宛にメッセージを再送する．

## 6. 実装および性能評価

### 6.1 実　　装

本システムはSunOS 4.1.1[14]上に実装を行った．移

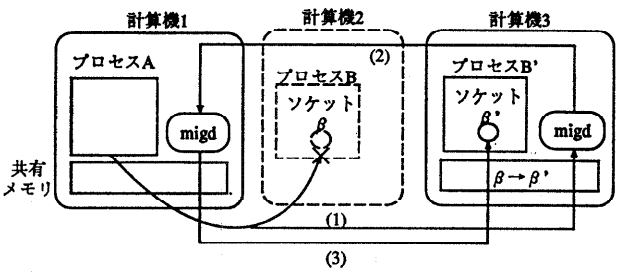



**図5** 非接続指向ソケットの移送処理（ホームノードが存在しない場合）
Fig. 5 Migration procedure of a connectionless socket process without a home-node.

送サーバ migd は非接続指向ソケットのメッセージを転送するためにネットワークドライバに対するデータの読み書きを行う必要がある．その手段として SunOS が提供している NIT デバイスを利用した．NIT デバイスを用いることにより，その計算機に送られてきたすべてのイーサネットパケットの読み出しや，任意のパケットの送出ができる．また，migd はホームノードが停止したソケット宛のパケットを転送するために，ネットワークに流れるすべてのメッセージを拾う機能を必要とする．この機能は NIT デバイスの promiscuous 機能を用いて実現した．promiscuous 機能を用いるとどの計算機宛のメッセージかに関わらず，接続されているサブネット[11]上を流れるすべてのパケットを拾うことができる．さらに SunOS で提供されている NIT デバイス用のフィルタにより，実際に読み出しを行うパケットのフィルタリングが行える．本システムではこのフィルタ機能を用いて UDP パケットのみを読み出すように設定している．

今回の実装ではあるソケットのホームノードが停止した場合，そのソケットを持っていたプロセスの移送先計算機上の migd が転送を行うようにした．NIT デバイスの promiscuous 機能では，プロセスがゲートウェイを越えて他のサブネット上に移送された場合，移送元のネットワーク宛のメッセージを拾うことができずメッセージの転送ができない．そのため各プロセスの移送先は同一サブネット内に制限している．この制限は各サブネットごとにホームノードが停止したソケットのための転送を行う migd を1つずつ配置することにより回避することができる．

### 6.2 性能評価

本プロセス移送システムの方法において，ユーザプロセスの動作のオーバヘッドとなるのは次の2点である．

- プロセスの移送処理に要する時間
- 接続指向および非接続指向通信時に移送システム内の処理に要する時間

ここではこの2点がユーザプロセスに与える影響を明らかにする．測定は各プロセスを実行するワークステーションとして SparcStation 1+(CPU : 25 MHz, メモリ 64 MB) を使用した．各プロセスのプロセスイメージを保存するディスクを持つワークステーションとして SparcServer 470 (CPU : 33 MHz, メモリ 32 MB) を使用した．

#### 6.2.1 移送処理のオーバヘッド

プロセス移送時の処理は4つの手順からなる．

1. プロセスの実行状態を獲得
2. 実行状態を実行可能な形式でファイルに出力
3. 移送後，migd が保存ファイルを fork/exec
4. 保存したプロセスの実行状態を回復

スタック領域サイズを増加することによってプロセスサイズを変化させた場合の各処理に要する時間の変化を**図6**に示す．保存ファイル作成に要する時間はプロセスサイズの変化に比例して増加している．プロセス実行状態の獲得時間は 17 ms とほぼ一定である．fork/exec に要する時間も平均値は 310 ms でほぼ一定である．実行状態の回復に要する時間はプロセスサイズに比例して増加している．これは実行状態の回復時にスタック領域を回復するためである．

移送対象のプロセスが持つソケットの数を変化させた場合の実行状態の獲得・回復時間を**表1**に示す．表中の TCP および UDP は，プロセスが持っているソ

**表 1** ソケット数に対する実行状態の保存・回復時間
Table 1 Save and restore time of process context when the number of socket varies.

単位（ミリ秒）

| | ソケット 0 | TCP×1 | TCP×2 | UDP×1 | UDP×2 |
|---|---|---|---|---|---|
| 実行状態保存 | 19.1 | 45.7 | 55.2 | 33.9 | 42.6 |
| 実行状態回復 | 131.5 | 138.3 | 149.2 | 147.3 | 163.4 |

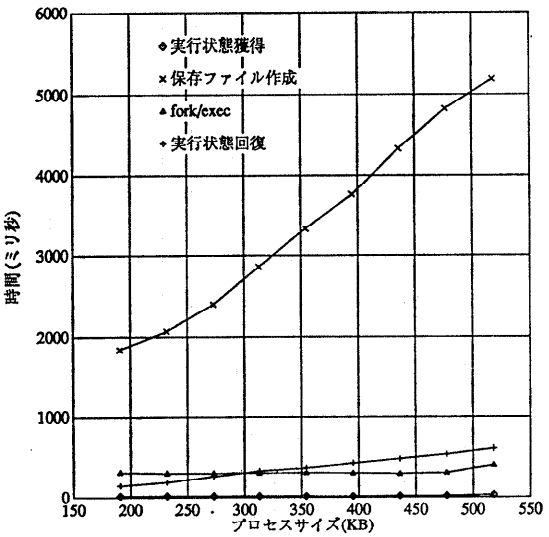



**図 6** プロセスサイズに対するプロセス移送時間
Fig. 6 Migration time when process size varies.

ケットがそれぞれ接続指向および非接続指向であることを示す．例えば，“TCP×1”は移送するプロセスが接続指向のソケットを1つ持っていることを意味する．この結果によれば，接続指向および非接続指向ソケットの実行状態獲得のオーバヘッドはソケット1つにつき約 20 ms および約 7 ms である．また，実行状態回復のオーバヘッドについては約 13 ms および約 17 ms である．

図6，表1から分かるように，プロセス移送に要する時間の大半は保存ファイルを作成する時間である．すなわち移送に要する時間はそのプロセスサイズに依存する．UNIX プロセスの大半はそのプロセスサイズが高々数 100 KB であるため移送に要する時間も数秒であり，ユーザに対する応答性を考えた場合十分に短い時間である．しかし，メモリ上のデータを大量に使用するシミュレーションプログラム等のプロセスサイズは数 MB に及ぶことがあり，このような場合の移送時間は数十秒とかなり長くなる．サイズの大きいプロセスを頻繁に移送すると，CPU，ディスク，ネットワーク等の負荷が高くなるため，負荷分散を目的としてプロセスを移送する場合は，プロセスのサイズを考慮する移送戦略が必須である．保存ファイル作成時間はプロセスサイズに比例して増加するため，プロセスサイズから移送時間を予測できる．移送時間の見積りは，どこにどのプロセスを移送するかを決める手段の1つとして有効に利用できる．また，本システムは従来のユーザレベルのプロセス移送方式にソケットの実行状態の保存・回復機能を拡張したものであり，プロセスの保存ファイル作成についてはテキスト領域を含むすべての領域を保存している．そのためテキスト領域は変わらないものとして，データおよびスタック領域のみを保存する等移送の高速化を計る必要がある．

#### 6.2.2 接続指向通信

ここでは本システムが接続指向通信に及ぼす影響を測定した結果を示す．測定に用いたプログラムはクライアントとサーバが最初に connect, accept システムコールによりソケットを接続し，read, write システムコールを用いて通信を行うものである．クライアントがサーバに 16 バイトのメッセージを送り，サーバから 16 バイトの応答が返って来るまでの時間を測定した．

測定結果を**表2**に示す．接続指向の通信の場合には migd は通常の通信時には介在せず，通信時のオーバヘッドは移送ライブラリによるもののみである．移送ライブラリのオーバヘッドは表2より 0.2〜0.3 ms であり，通信時間と比較して小さい．

#### 6.2.3 非接続指向通信

非接続指向通信に対して本システムが及ぼすオーバヘッドの要因は以下の3つである．

- 移送ライブラリは通信を行う前に共有メモリを参照し，通信相手が移送されたかどうかを調べる．
- migd は NIT デバイスを使用して UDP のメッセージを絶えず監視しているため，転送の必要がない UDP メッセージも処理を行う．
- 既に移送されたプロセスに送られてくるメッセージは migd により転送される．

これらのオーバヘッドを明らかにするために以下の各条件で通信時間の測定を行った．

**通常モード**：移送ライブラリをリンクしていない通常のプログラムの通信時間

**通常＋NIT**：migd が NIT デバイスをオープンし UDP メッセージを監視する状態での，通常のプログラムの通信時間

**移送モード**：移送ライブラリをリンクしたプログラムの通信時間

**転送**：プロセス移送直後，migd によりメッセージが転送される場合の通信時間（図4の場合）

**PROMISC モード**：ホームノードが停止した場合の通信時間（図5の場合）

測定に用いたプログラムは，クライアントとサーバが最初にソケットを生成後，sendto, recvfrom システムコールを用いて通信を行うものである．クライアントがサーバに 16 バイトのメッセージを送り，サーバから 16 バイトの応答が返って来るまでの時間を測定した．測定結果を**表3**に示す．表中のリモート，ローカルはサーバとクライアントが異なる計算機上に存在するか同一計算機上に存在するかを示す．

移送ライブラリ自体のオーバヘッドは，NIT デバイスによる UDP パケット監視処理の影響が及ばない“通常モード”のローカルと“移送モード”のローカルの比較より 0.2 ms である．移送ライブラリでは通常

**表2** 接続指向の通信時間
Table 2 Elapse time of connection-oriented communications.

単位（ミリ秒）

| | ローカル | リモート |
|---|---|---|
| 通常モード | 1.7 | 2.1 |
| 移送モード | 2.0 | 2.3 |

表 3 非接続指向の通信時間
Table 3 Elapse time of connectionless communications.

単位（ミリ秒）

| | ローカル | リモート |
|---|---|---|
| 通常モード | 1.9 | 2.0 |
| 通常+NIT | 1.9 | 3.0 |
| 移送モード | 2.1 | 3.1 |
| 転　　送 | — | 10.7 |
| PROMISC モード | — | 15.4 |

の通信の場合，共有メモリを調べ通常の通信を行うのみであり，そのオーバヘッドは小さい．

NIT デバイスによる UDP パケット監視処理のオーバヘッドは“通常モード”と“通常+NIT”の比較より約 1 ms である．このオーバヘッドはサーバ，クライアントが存在する双方の計算機上で，migd が NIT デバイスからメッセージを獲得し処理を行うために生じる．

メッセージの転送および PROMISC モードの処理は，表 3 によれば各々 10.7 ms，15.4 ms とオーバヘッドが大きくなっている．しかしこれらの処理は移送直後の 1 回目のみで，それ以降は余分な転送を必要せず，直接移送先と通信する．そのため，大半の通信は移送モードの時間で行われ，転送によるオーバヘッドが性能に与える影響は小さい．

以上のことから，非接続通信のオーバヘッドの大半は NIT デバイスによる UDP パケット監視処理に起因するものである．現在の実現方式では，SunOS で提供されているフィルタを使用して UDP パケットのみを NIT デバイスから獲得している．この方式では，UDP パケットが送られて来ると必ず migd の処理が行われるため常にオーバヘッドが生じる．そこで，アドレスを変換するためのテーブルを持ち，それに従ってメッセージの転送を行う高機能なフィルタをカーネル内に組み込めば，転送処理はカーネル内で行われ，migd の処理が起動されなくなるため，NIT デバイスに起因するオーバヘッドを押さえることができる．

## 7. おわりに

本論文では通信を行うプロセスの移送をユーザレベルで行う方式を述べた．プロセス間通信の手段は標準的な UNIX で提供されているソケットを対象とした．また，この方法により実装したプロセス移送システムのオーバヘッドを明らかにした．

本方式はソケットを使用するプロセスの移送を可能にしたため，SunRPC[14] や X ウィンドウシステム[15] 等のソケット上に構築されたアプリケーションをソースコードの変更なしに移送できる．例えば X ウィンドウシステムでは，画面を表示する X サーバと実際の計算等を行う X クライアントがソケットを使用したプロセス間通信により処理を進める．このようなシステムにおいて，本プロセス移送機能を用いて X クライアントを移送することができた．この際，X サーバ，X クライアントのソースコードは一切変更せず，プロセス移送用のライブラリをリンクするのみで移送可能になった．

接続指向ソケットの移送処理はソケットインタフェースの上位で行ったため，通信プロトコルに関して移植性の高い実現方式になった．また，本方式は TLI[11] などソケット以外の通信用アプリケーションインタフェースに対しても適用可能である．

一方，今回実装した非接続ソケットの移送方式は，NIT デバイスを使用しているため，SunOS に依存した実装になっている．また，イーサネットアドレス，IP アドレスの操作を行うため，ネットワークおよびその通信プロトコルにも依存している．多くの UNIX ではネットワーク上のパケットを監視するために，それらのパケットをユーザレベルで獲得する手段が提供されている．また，標準的な UNIX で提供されている Raw ソケットと呼ばれるソケットにより，TCP や UDP などのトランスポート層の下位に存在するネットワーク層への書き込みが可能である．この 2 つの機能を使用することにより，OS に関してより移植性の高い方法が実現できる可能性がある．

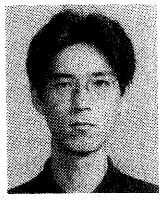
第 2 章で述べたように本移送方式はソケットの非閉塞入出力に対応していない等の制限を持っている．標準的な UNIX 上でこれらの制限を取り除くためにはカーネル内部で解決する必要がある．本論文で述べたソケットの移送処理はソケットの再接続やメッセージの転送を行うが，これらの処理はカーネル内部のソケット層やプロトコル層内部で実現可能である．そのため今後，マイクロカーネル[1] 等による OS のモジュール化が進めば，その通信モジュールに本方式を取り込むことにより，より汎用的で制限のない移送方式が実現可能であると考えている．

## 参考文献

1) 前川　守，所真理雄，清水謙多郎：分散オペレーティングシステム UNIX の次にくるもの，p. 335，共立出版（1991）．

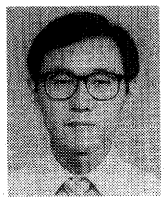
2) Powell, M. L. and Miller, B. P.: Process Migration in DEMOS/MP, *Proc. 9th ACM SOSP*, pp. 110-119 (1983).
3) Butterfield, S. A. and Popek, G. J.: Network Tasking in the Locus Distributed Unix System, *Proc. Usenix Summer '84*, pp. 62-71 (1984).
4) Theimer, M. M., Lantz, K. A. and Cheriton, D. R.: Preemptable Remote Execution Facilities for the V-System, *Proc. 10th ACM SOSP*, pp. 2-12 (1985).
5) Douglis, F. and Ousterhout, J.: Process Migration in the Sprite Operating System, *Proc. 11th ACM SOSP*, pp. 18-25 (1987).
6) Litzkow, M. and Solomon, M.: Supporting Checkpointing and Process Migration Outside The Unix Kernel, *Proc. Usenix Winter '92*, pp. 283-290 (1992).
7) Mandelberg, K. I. and Sunderam, V. S.: Process Migration in UNIX Networks, *Proc. Usenix Winter '88*, pp. 357-363 (1988).
8) 森山茂男，多田好克：利用者レベルで実現したプロセス移送ライブラリ，情報処理学会研究会 OS 研究会報告 91-OS-51, pp. 41-47 (1991).
9) Bach, M. J.: *The Design of the UNIX Operating System*, p. 471, Prentice Hall (1986).
10) Leffler, S. J., McKusick, M. K., Karels, M. J. and Quarterman, J. S.: *The Design and Implementation of the 4.3 BSD UNIX Operating System*, p. 471, Addison-Wesley Publishing Company, Inc. (1989).
11) Stevens, W. R.: *UNIX Network Programming*, p. 772, Prentice Hall (1990).
12) Comer, D. E.: *Internetworking with TCP/IP Vol. I: Principles, Protocols, and Architecture*, p. 534, Prentice Hall (1991).
13) Kleiman, S. R.: Vnodes: An Architecture for Multiple File System Types in Sun UNIX, *Proc. Usenix Summer '86*, pp. 238-247 (1986).
14) *SunOS Reference Manual*, Sun Microsystems, Inc. (1990).
15) Scheifler, R. W. and Gettys, J.: The X Window System, *ACM Trans. Graphics*, Vol. 5, No. 2, pp. 79-109 (1986).

(平成4年10月29日受付)
(平成5年4月8日採録)



**白木原敏雄**（正会員）

1964年生．1987年九州大学工学部情報工学科卒業．1989年同大学大学院修士課程修了．1989年(株)東芝入社．現在，研究開発センター情報・通信システム研究所第二研究所に所属．分散処理のためのオペレーティング・システムなどの研究に従事．



**金井 達徳**（正会員）

1961年生．1984年京都大学工学部情報工学科卒業．1989年同大学院博士課程修了．1989年(株)東芝入社．現在，研究開発センター情報・通信システム研究所第二研究所に所属．分散処理のためのオペレーティング・システム，データベースシステム，プログラミング言語などの研究に従事．電子情報通信学会会員．