2008－OS－109 (5)
2008／8／7

P-BusにおけるOSカーネル間通信機構の設計と実装

藤田 肇[†] 平野貴仁[††] 山本和典[††]
松葉浩也[†] 石川裕[††,†]

分散環境においてOSカーネル同士が通信を行うためのインタフェースIKC(Inter-kernel Communication)を提案する．IKCは様々な通信プロトコルや通信デバイスの差異を吸収し，共通のインタフェースでの通信を行えるようにする．またIKCはOSカーネル内での使用に配慮し，全ての操作をノンブロッキングで行い，パケット受信のような非同期に発生するイベントはコールバック関数の形で通知する．本稿ではIKCの設計と，Linuxカーネル上でのSCTPを用いたIKCドライバの実装方針について述べる．

Design and Implementation of Inter-kernel Communication Mechanism in P-Bus

Hajime Fujita,[†] Takahito Hirano,[††] Kazunori Yamamoto,[††]
Hiroya Matsuba[†] and Yutaka Ishikawa[††,†]

In this paper, we propose IKC (Inter-kernel Communication), a communication interface for operating system kernels. IKC is designed to absorb differences between various communication devices and protocols. Also, IKC specifies every operation to be implemented as a non-blocking function so that any kernel codes are able to call IKC interfaces at any moment. Asynchronous events, such as data arrival, are notified by callback functions. We are currently implementing IKC device layer using SCTP. Implementation details of the IKC/SCTP device are also described.

1. はじめに

オペレーティングシステムカーネルはユーザ空間プログラムにネットワーク通信機能を提供するが，それだけでなく，カーネル自信が通信を行う必要が生じることがある．例えば，分散ファイルシステム，分散共有メモリ，プロセスマイグレーションといった機能をカーネルレベルで実現するためには，カーネル内が通信の端点となる必要がある．カーネル内にこのようなコードを実装する理由としては，ユーザ空間に透過な実装を提供したい，コンテキストスイッチのオーバーヘッドを削減したいといったことがある．

OSカーネル内にこれらの通信を利用する機能を実装する方法として，ソケットAPIをカーネル内から利用することが考えられる．しかし，ソケットAPIはユーザ空間に対して機能を提供することを前提に設計・実装されており，カーネル空間内から利用しにくい，あるいは効率が悪い面がある．

まずソケットAPIの最上位層はユーザモードで動作するプログラムのアドレス空間とデータをやりとりすることを前提として書かれており，ユーザアドレス空間とカーネルアドレス空間との間でデータのコピーが行われる．カーネル内からソケットAPIを利用する場合このコピーは本来は不要であり，CPU時間，メモリ帯域，キャッシュといった資源を浪費する．

また，ソケットAPIのコードはユーザプロセスコンテキストで動作するため，状況に応じて休眠(コンテキストスイッチ)するように実装されている．カーネル内のコードには，割り込みコンテキストやソフトウェア割り込みコンテキスト，スピンロックを保持している区間，割り込みを禁止している区間など，休眠が許されない区間が存在する．このような休眠禁止区間からはソケットAPIを利用した通信を行うことはできない．

一方，さきほど例として挙げた分散ファイルシステムや分散共有メモリといった分散OS機能においてはネットワークはなるべく広帯域・低遅延であることが望ましい．このような目的で用いられるネットワークデバイスとしては例えばMyrinet[1]やInfiniBand[2]がある．これらのネットワークデバイスはEthernetデバイスのように標準的なネットワークデバイスとして扱うことも出来るが，本来の性能を生かした通信を

† 東京大学情報基盤センター
Information Technology Center, The University of Tokyo
†† 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

行うためには専用の API を使用する必要がある[3]．しかし，このようなデバイスはどのような計算機にも備わっているとは限らず，専用 API を利用すると環境に応じてデバイスごとに通信機能を書き直さなければならない．

OS カーネルを安全かつ柔軟に拡張するための枠組みとして，我々は P-Bus[4)~6)] を提案している．我々は P-Bus の API の一部として，OS カーネル内に通信を利用した機構を実装するためのカーネル間通信インタフェース IKC(Inter-kernel Communication) を設計している．

IKC はデバイスに非依存で信頼性のある一対一通信を提供する．IKC の送受信操作は内部で休眠しないように実装されており，カーネル内の任意の箇所から呼び出すことができる．本稿では，IKC の設計について延べた後，SCTP を用いた IKC デバイス実装の Linux カーネル上での実装方針について述べる．

2. 設　　計

本節では，P-Bus がカーネル内プログラミングのために提供するネットワーク関連機能の構成について述べる．

IKC はデバイスに依存しない，OS カーネル内で利用可能な通信インタフェースを提供する．このインタフェースの目標は次の通りである．

- 実際の通信デバイスによらず同じインタフェースで通信可能なこと
- 信頼性のある通信を提供すること
- カーネル内のあらゆる箇所から利用可能なこと (休眠する操作を含まないこと)
- 通信にあたって可能な限りメモリのコピーを避けること

2.1 通信相手の特定

IKC は LAN 内に構成されたクラスタのように，ある閉じた計算機の集合内部での通信に用いられることを想定している．IKC は各ノードをノード番号と呼ぶ整数値で識別する．ノード番号は IKC を用いて通信しうる範囲内で唯一であることが必要である．ノード番号は計算機 1 台につき 1 つ定まる．ある計算機が複数のネットワークデバイスを持ち，複数のネットワークアドレスを持っていても，IKC のレベルでは共通の 1 つのノード番号を用いる．

また，一般に 1 台の計算機上では複数のサービスが同時に動作しうる．例えば分散共有メモリ機構とプロセスマイグレーション機構が同時に動作することが考えられる．このため通信相手を識別するためにはノード番号の他にサービスを識別するための識別子が必要である．IKC ではこの識別子をタイプ (type) という整数値で表現する．

2.2 通信の特性

IKC は以下に示すような通信を提供する．

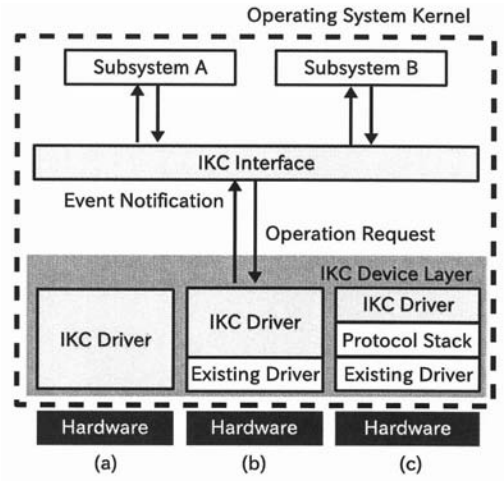

図 1 IKC の概要．デバイス層の構成は (a) デバイスドライバを含めて IKC ドライバが実装 (b) 既存のハードウェアデバイスドライバの上に IKC ドライバが IKC プロトコルを実装 (c) 既存のプロトコルスタックの上に IKC ドライバインタフェースを実装 の 3 通りが考えられる．

- 信頼性のあるコネクション指向の一対一通信
  通信を行うためには通信相手とのコネクションを確立する必要がある．通信相手はノード番号とタイプの組で表現される．また，通信路に送出されたデータは送出された順番で受信され，データの欠落や重複はないことが保証される．
- メッセージ境界の保存
  1 回の送信操作で送信されたデータは 1 回の受信操作で全て受信される．
- 通信の多重化
  あるノードとタイプの組との間に同時に複数のコネクションを張ることができる．それぞれのコネクションは独立のものとして扱われ，通信データが混じり合うことはない．

通信路はチャネル (channel) と呼ぶ抽象的データ構造により表現され，1 つの通信路に 1 つのチャネルが対応する．チャネルには通信相手のノード番号とタイプ番号といった情報や，IKC デバイス固有の通信に必要なデータが格納される．

2.3 基 本 構 造

IKC は，**図 1** に示すような階層構造を取る．図中に Subsystem と書かれた部分が，分散共有メモリやプロセスマイグレーション機構といった通信を行うサブシステムである．一方実際に通信機能を実装している部分を IKC デバイスと呼び，実際に通信に用いられるデバイス/プロトコルごとに異なる IKC デバイスが使用される．

IKC インタフェースはサブシステムと IKC デバイスとの橋渡しをする．サブシステムが IKC デバイス

を用いた操作を実行する際には，IKC を経由して行う．逆に IKC デバイスがサブシステムにイベントを通知する際にも IKC を経由して通知する．これによって，サブシステムは実際に使用されるデバイスに依存せずに，デバイスはイベントの通知先のサブシステムに依存せずにデータやイベントの受け渡しができる．このように IKC 層はサブシステムと IKC デバイスとを接続しているのみで，通信における本質的な処理は介在していない．通信に関わるあらゆる処理，例えば信頼性確保やフローコントロールのような処理は全て IKC デバイス層の責任で行う．

既に述べたように IKC は全ての基本操作を休眠なしで行うことを大前提としている．しかし，ネットワーク越しの操作には本質的に操作の開始から完了までの時間が長いものがある．例えば相手側ホストにコネクション開始要求を出してから実際にコネクションが確立されるまでには，今日の CPU の実行速度からすると非常に長い時間を要する．そのため，IKC は全ての操作を操作要求の開始点と定義し，操作の完了時にはあらかじめ登録しておいた関数がコールバックされることとした．

IKC の通信インタフェースの特徴はデータの受信が非同期なコールバックとして受動的に行われる点である．ソケット API の `recv` システムコールのように能動的に到着データを取得するインタフェースは存在しない．この点で IKC のインタフェースはソケット API と大きく異なっている．

2.4 コールバック

コールバックは非同期に起こる．コールバックはプロセスコンテキストないしソフトウェア割り込みコンテキストの任意のコンテキストから呼ばれうる．ソフトウェア割り込みコンテキストで走行する可能性があるため，コールバック関数内で休眠する操作を行うことはできない．またマルチプロセッサ環境上のカーネルでは，コールバック関数は任意の CPU 上で実行されうる．したがってコールバック関数と競合する可能性のある箇所はロックをかけるとともにソフトウェア割り込みを禁止する必要がある．

2.5 通信バッファの表現と扱い

IKC では，通信バッファとして `pbus_netbuf_t` という専用のバッファ型を用いる．この型は，各 OS カーネルが提供するネットワーク送受信用のバッファ構造体 (Linux であればソケットバッファ`struct sk_buff`) にマップされる．

このような特殊な型を用いる理由は 2 つある．1 つめの理由は，既存のネットワークデバイスドライバが理解するバッファ構造に合わせるためである．図 1 の (b) や (c) のような実装形態を取る IKC デバイスの場合，最下層で送受信を行うデバイスドライバに適したバッファ形式としておかないと，途中で変換やコピーが生じる可能性がある．

2 つめの理由は，物理ページを扱えるためである．Linux のソケットバッファや FreeBSD の mbuf では大量のデータを送信するときのために物理ページのディスクリプタをバッファ構造体内に格納することができる．これによって，物理ページの内容を転送することが目的の場合にページをカーネル仮想アドレス空間にマップする必要がなくなる．

通信バッファは，送信時は送信を要求するサブシステムによって確保され，送信を実行する IKC デバイスによって解放される．一方受信時には IKC デバイスによってバッファが確保され，それがサブシステムに渡されてくる．この操作はポインタの受け渡しであり，コピーは発生しない．受信バッファを解放するのは受信したサブシステムの責任である．

2.6 関　　数

IKC が定義する関数には次の種類がある．

- サブシステムから IKC を呼ぶ関数
- IKC からサブシステムを呼ぶ関数
- デバイスから IKC を呼ぶ関数
- IKC からデバイスを呼ぶ関数

このうち IKC から呼び出す形の関数がコールバックである．なお，実際の関数名および型名には接頭辞 `ikc_` がつくが，本稿ではスペースの関係上省略した．

2.6.1 サブシステムから IKC を呼ぶ関数

これらの関数はサブシステムが IKC を呼び出す形で利用するものである．

- `int type_register(int type, struct callback_ops *ops)`
  新しいタイプ番号を登録する．`ops` は IKC からサブシステムをコールバックする関数ポインタ群を指定した構造体である．
- `int type_unregister(int type)`
  タイプを抹消する．
- `int channel_open(int type, node_t dst, struct channel **chan)`
  ノード番号 `dst` で与えられるホスト上のサービスに対して新たなコネクションの開始を要求する．通信相手のサービスは `type` によって識別される．新たなチャネルが `chan` に返されるが，`channel_open` が完了した時点ではデバイス層での実際の接続は完了していない可能性がある．
- `int send(struct channel *chan, pbus_netbuf_t *nbuf, void *id)`
  `nbuf` に指定されたバッファの内容を `chan` で指定されたチャネルに送信する．`id` は個々の送信処理を識別するための任意のデータであり，送信者が自由に設定できる．
- `int channel_close(struct channel *chan)`
  コネクションを切断しチャネル構造体を解放する．

2.6.2 IKC からサブシステムを呼ぶ関数

これらの関数はサブシステムが非同期に発生するイ

ベントを IKC 経由でサブシステムが受け取るためのものである．これらの関数はサブシステムが実装し type_register でタイプごとに登録する．

- `void channel_open_complete(int result, struct channel *chan)`
  channel_open で要求したコネクション要求が完了したことを通知する．コネクション確立に失敗した場合も呼ばれ，その場合には result にエラーの原因が通知される．
- `void send_complete(int result, struct channel *chan, void *id)`
  送信処理が成功であれ失敗であれ完了したことを示す．id は send 時に指定したものである．
- `void data_arrived(struct channel *chan, pbus_netbuf_t *nbuf)`
  データが到着した際に呼ばれる．
- `void channel_closed(struct channel *chan)`
  コネクションが相手によって切断されたことを示す．
- `int accept(struct channel *newch)`
  新しいコネクションが開かれた際に呼ばれる．newch は新しいコネクションを表すチャネルである．コールバック関数は 0 以外のエラー値を返すことでコネクションを拒否することができる．

2.6.3 デバイスから IKC を呼ぶ関数

- `int dev_register(int priority, nodeset_t availability, struct dev_ops *ops, struct device **dev)`
  新しいデバイスを登録する．ops には IKC からデバイスを呼ぶための関数ポインタ群が登録されている．availability はこのデバイスを用いて通信が可能なノード番号の集合を表す．priority はデバイスの優先度である．同一ノードに対して複数のデバイスが選択可能な場合は優先度の高いものが用いられる．新しいデバイス構造体が dev に返される．
- `int dev_unregister(struct device *dev)`
  デバイスの登録を抹消する．

デバイスから IKC を呼ぶ関数を関数はこれらに加え，2.6.2 に示したコールバック関数を呼び出すための IKC インタフェースがある．IKC デバイスは各タイプのコールバック関数を直接呼ぶのではなく，IKC インタフェースを通じて呼び出す．

2.6.4 IKC からデバイスを呼ぶ関数

IKC からデバイスを呼び出すための関数は，2.6.1 節に述べた channel_open, send, channel_close と同じ形式の関数群と，デバイスの初期化，終了処理のための関数群をまとめた関数ポインタ群である．この構造体は dev_register に渡され，デバイスの登録時に使用される．

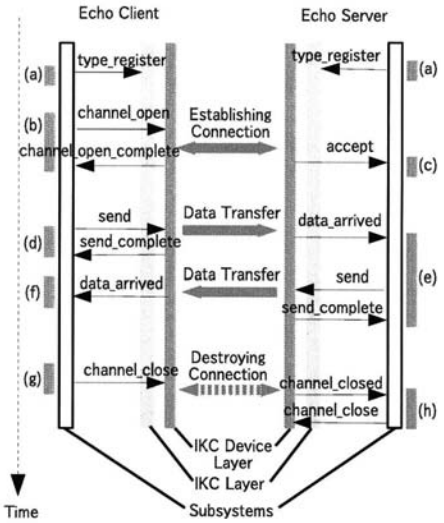

図 2 IKC を用いた echo サービスの実現例．

2.7 サンプル: echo

IKC を用いて通信を行う様子を図 2 に示す．これはデータを送信するサブシステム (echo client) とそれを受け取って送り返すサブシステム (echo server) とが通信する様子である．通信に必要なデバイス層は既に登録され初期化されているものとする．

まず，(a) で双方が同じタイプを IKC に登録する．次に，(b) でクライアントがサーバへのコネクションを要求する．このとき，図では細部が省略されているが，実際にはまずクライアントサブシステムは IKC 層の channel_open 関数を呼ぶ．すると IKC 層ではチャネルに対し適切なデバイスを選択し，その IKC デバイスの channel_open 関数を呼ぶ．

コネクションが確立されると，クライアント側では channel_open_complete コールバック関数が呼ばれる (c)．一方サーバ側ではコネクションの確立後 accept コールバック関数が呼ばれる．接続が確立すると，クライアントはデータを準備し (d) でチャネルに対し送信する．(e) では，クライアントから送信されたデータが data_arrived によってサーバに渡される．サーバサブシステムは即座に同じデータを送り返す．そして送り返されたデータをクライアントは (f) で受信する．

最後にクライアントは channel_close を呼んでコネクションを切断する．クライアントによってコネクションが切断されたことは (h) でサーバ側に通知される．サーバ側も channel_close を呼んでチャネル構造体を解放する．

3. 実　　装

我々は現在，Linux 2.6.24 上に IKC の実装を進めている．本節では，IKC の動作検証用に実装している

ITC/SCTP の実装方針について述べる．

3.1 IKC/SCTP

IKC/SCTP は通信のために SCTP[7] を利用する IKC デバイスである．この IKC デバイスは図 1 の (c) にあたる方式で実装されている．SCTP は

- コネクション指向であり，
- メッセージ境界を保存し，
- 信頼性のある

通信機能を提供するため，IKC のセマンティクスと親和性が高い．

3.2 P-Bus 上での実装

我々は IKC を P-Bus API の一部として定義することを考えているが，IKC の実装そのものは本質的には OS カーネルに依存しないものであり，IKC の実装は P-Bus 上のコンポーネント (P-Component) として記述可能である．

現在，IKC インタフェース層 (図 1 の"IKC Interface") は P-Bus の API のみを用いて記述されているが，IKC/SCTP(図 1 の"IKC Device Layer") は Linux カーネルの API を用いて記述されている．現在の実装では，P-Bus API を用いて記述されたコードも最終的には Linux のローダブルカーネルモジュールにコンパイルされるため[5]，これらのコードをリンクして使用することが可能である．将来的には，ハードウェア操作までを含めた IKC デバイス全体を P-Bus API で記述できるようにし，IKC デバイスも含め P-Component 化する予定である．

3.3 操作のマッピング

SCTP 上に IKC のセマンティクスを実現するために，IKC の操作と SCTP の操作の対応付けを定義する必要がある．まず，IKC のコネクションと SCTP のコネクションは一対一対応するものとした．次に，IKC/SCTP は特定の SCTP ポートのみを `listen` し，接続を待ち受けるものとした．新しい接続を確立する際にはまず SCTP コネクションを確立し，接続をアクティブオープンした側 (`channel_open` を呼んだ側) がタイプを通知する．パッシブオープンした側では，通知されたタイプが自分のノードに登録されているかどうか調べ，対応するタイプが登録されていれば新しいチャネル構造体を割り当てて `accept` を呼び出す．

3.4 遅延実行処理の実装

実装を容易にするため，IKC/SCTP の一部はソケット API を利用して実装されている．ソケット API を利用するのは `channel_open`, `send`, `accept` である．これらはソケット API ではそれぞれ `connect`, `sendmsg`, `accept` に対応するが，これらのソケット API を実装しているカーネル内の関数 `sctp_connect`, `sctp_sendmsg`, `sctp_accept` はどれも休眠しうるコードを含んでいるため，IKC デバイスのコールバック関数内で直接実行することはできない．このため，IKC/SCTP は休眠しうる操作を行うためのカーネルスレッドを用意し，休眠する可能性のある処理はこのカーネルスレッドで行っている．

3.5 非同期通知の実装

IP パケット受信を契機とするイベントを拾うことでコールバックを実現できるケースがある．STCP パケットの受信時には，ソケットバッファが受信キューに繋がれたのちに対応するソケットの `sock` 構造体の `sk_data_ready` 関数ポインタが呼ばれる．これを利用し，IKC/SCTP で用いるソケットの `sk_data_ready` 関数ポインタを自前のものに変更しておくことで，データを受信したタイミングを知ることができる．

同様に，`LISTEN` 状態にあるソケットに新しい接続要求が到着した際，もしくは `ESTABLISHED` 状態にある接続において相手側が接続を切断した際には `sk_state_change` 関数ポインタが呼ばれることを利用して，これらのタイミングを捉えることが可能である．

4. 議　　論

本節では，現状の IKC の設計において問題となる箇所について延べ，今後の設計方針について議論する．

4.1 上位層 API

IKC は非同期に発生するイベントをすべて非同期なコールバックとして扱う．これは柔軟なインタフェースであると同時に，そこまでの柔軟性が必要ない場合には実装を困難にする要因でもある．そこで，IKC の上位層に MCAPI[8] のようなコールバック中心でない API をもう一段実装することが考えられる．

また，別の要求として，一対一通信でなく一対多の通信を行いたい場合がある．このような要求に対しては，IKC を用いて全体全のコネクションを張る層を用意することで対応できる．

4.2 デバイス選択

現在の設計及び実装では通信先ノードを指定すると自動で使用されるデバイスが選択される．一般にあるノードと通信するために複数の通信デバイスが利用可能な場合がある．このような場合，通信の目的によってデバイスを使い分けたり，複数の通信経路を使用して耐障害性を高めたりすることができる[9]．

このため，今後はデバイス選択を上位のサブシステムに委ねるよう，あるノードとの通信に対して使用可能なデバイスのリストを返すインタフェースや，`channel_open` の際に使用するデバイスを明示的に選択するインタフェースが必要になると考えられる．

5. 関 連 研 究

PM[10] は高性能計算のための通信ライブラリである．PM が Myrinet のみを通信デバイスとして想定しているのに対して，後継実装である PM2[11] では Myrinet, Ethernet, 共有メモリといった複数の通信デバイスを共通のインタフェースで操作できるようにし

ている．通信デバイス/プロトコルごとにIKCのドライバを作成するというアプローチはPM2に由来するものである．PM，PM2はユーザ空間から通信を行うためのライブラリであり，データの到着をポーリングで待つインタフェースとなっているため，OSカーネル内で使用することは難しい[12]．

MCAPI(Multicore Communication API)[8]はMulticore Associationによって策定された通信APIである．MCAPIはすべての送受信関数にノンブロッキング版が用意されており，カーネル内の休眠不能コンテキストからも使えるように実装が可能である．またMCAPIでは受信したデータへのポインタを受け取る形の受信操作を行うことができ，受信時のコピー操作が不要という点でIKCに近い．ただしMCAPIはコールバックをサポートしておらず，受信操作は能動的に受信関数を呼び出すことによって行う．その点でIKCの方がより本質的なインタフェースであり，我々はIKCを用いてMCAPIを実装することができると考えている．

6. おわりに

本稿では，P-Busが定義するカーネル間通信インタフェースIKCについて，全体の設計とSCTPを用いたLinux上の試験的実装について述べた．IKCはカーネル内のあらゆるコードに対して通信機能を提供するため，全ての操作を休眠しないものと定義し，非同期なイベントは全てコールバックの形で通知する．IKCはまた複数種の通信デバイスや通信プロトコルを扱えるように設計されており，実際に通信機能を提供するデバイス層に依存せず通信が可能である．今後の課題は，まずIKC/SCTPについての評価を行うことである．また，他の通信デバイスを利用するIKCデバイスも実装し，性能や実装の容易さについて評価する必要がある．

P-Busではカーネル APIを定義するとともに，APIの使い方の正しさをモデル検査器で検査することを目指している[6]．IKCデバイスやIKCを利用するカーネルサブシステムは複数のコンテキストの競合を念頭において実装する必要があるため，IKC周辺の実装をモデル検査のためのテストケースとして用いることを考えている．

謝辞 本研究の一部は，科学技術振興機構 戦略的創造研究推進事業 (CREST) (領域名：実用化を目指した組込みシステム用ディペンダブル・オペレーティングシステム) 技術課題：「高信頼組込みシングルシステムイメージOS」による．

参考文献

1) Myricom: Myrinet. http://www.myri.com.
2) InfiniBand Trade Association: InfiniBand. http://www.infinibandta.org.
3) Myricom: Myrinet Express. http://www.myri.com/scs/MX/doc/mx.pdf.
4) 平野貴仁, 藤田肇, 松葉浩也, 石川裕: PBus：柔軟なカーネル機能拡張のためのインタフェース, *2007-OS-106*, 情報処理学会, pp. 63–70 (2007).
5) 平野貴仁, 藤田肇, 松葉浩也, 石川裕: カーネル機能拡張のための抽象化レイヤP-Busの実装, *2008-OS-108*, 情報処理学会, pp. 17–24 (2008).
6) 藤田肇, 平野貴仁, 松葉浩也, 前田俊行, 菅谷みどり, 石川裕: 安全かつ拡張可能なOS開発基盤の実現にむけて, 第6回ディペンダブルシステムワークショップ (DSW'08summer) (2008).
7) Stewart, R.: Stream Control Transmission Protocol, RFC 4960 (2007).
8) Multicore Association: Multicore Communication API, `http://www.multicore-association.org/` (2008).
9) Miura, S., Boku, T., Sato, M. and Takahashi, D.: RI2N - Interconnection Network System for Clusters with Wide-Bandwidth and Fault-Tolerancy Based on Multiple Links, *ISHPC 2003: High Performance Computing, 5th International Symposium*, Lecture Notes in Computer Science, Vol. 2858, Springer, pp. 342–351 (2003).
10) Tezuka, H., Hori, A., Ishikawa, Y. and Sato, M.: PM: An Operating System Coordinated High Performance Communication Library, *HPCN Europe '97: Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking*, London, UK, Springer-Verlag, pp. 708–717 (1997).
11) Takahashi, T., Sumimoto, S., Hori, A., Harada, H. and Ishikawa, Y.: PM2: High Performance Communication Middleware for Heterogeneous Network Environments, *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Washington, DC, USA, IEEE Computer Society, p. 16 (2000).
12) Matsuba, H. and Ishikawa, Y.: Single IP address cluster for internet servers, *Proceedings of 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS2007)* (2007).