情報処理学会論文誌　コンピューティングシステム　Vol. 1　No. 1　12–27 (June 2008)

ユーザ透過に利用可能な高性能・耐故障マルチリンクEthernet結合システム

岡本高幸[†1,*1]　三浦信一[†1,*2]　朴泰祐[†1,†2]
塙敏博[†2]　佐藤三久[†1,†2]

本論文では，Gigabit Ethernet を複数同時に用いることで高バンド幅と耐故障性の両者を実現する PC クラスタ向けネットワークである RI2N のシステムレベル実装である RI2N/DRV を提案する．本システムは仮想的な Ethernet デバイスとして機能し，既存の TCP/IP レイヤからユーザ透過に利用できる．また，RI2N/DRV 内部でパケット順序の復元を行うことで，比較的短いメッセージから安定して高いスループットが得られる．本システムを Linux 上に実装し，Gigabit Ethernet を 2 リンク利用した環境で評価を行った．ドライバレベルでの実装によってレイテンシの上昇を抑えつつ，スループットとしては最大で 222 MB/s というシングルリンクのほぼ 2 倍の性能が得られた．既存の Linux Channel Bonding と比較して，より短いメッセージで高いスループットが得られることも確認できた．また，いずれかのリンクが故障した場合にも冗長リンクを使って通信を継続できることを確認した．

User-transparent Ethernet Multilink Bonding System for High Performance and Fault-tolerance

Takayuki Okamoto,[†1,*1] Shin'ichi Miura,[†1,*2]
Taisuke Boku,[†1,†2] Toshihiro Hanawa[†2]
and Mitsuhisa Sato[†1,†2]

RI2N is a concept of commodity interconnection network for HPC PC clusters to unify the high performance and reliability with multiple links of network. In our previous work, we have developed an implementation of RI2N as a user-level communication library named RI2N/UDP, which requires a slight modification on the application written in TCP/IP manner as well as recompilation. In this paper, we propose a user-transparent implementation of RI2N named RI2N/DRV. Although it is similar to Channel Bonding technology in standard Linux, it provides a packet sequencing function to improve the throughput on any message size solving the problem on unmatched feature of multi-link use and TCP/IP protocol. We implemented RI2N/DRV as a pseudo Ethernet device on Linux and evaluated its performance and fault tolerance function. As a result, it provides 222 MB/s of maximum throughput with dual-link Gigabit Ethernet as well as the link failure detection and recovery functions. It is also shown that RI2N/DRV provides lower latency and higher throughput than Linux Channel Bonding with any practical message size.

1. はじめに

クラスタ向きの高性能・高信頼ネットワークである InfiniBand や Myrinet，また，Gigabit Ethernet（以下，GbE）の次世代となる 10 Gigabit Ethernet などはその価格がまだ高価であり，小中規模の PC クラスタへ導入するにはハードルが高い．そのため，現在も PC クラスタの多くでは GbE が利用されている．その一方で，近年のマルチコアプロセッサの普及により，PC クラスタにおいてもマルチコア，マルチソケットが標準になりつつある．これにより，1 ノードあたりの計算能力は飛躍的に向上しており，それらをつなぐネットワークの性能に対する要求も高くなっている．

そこで我々は GbE を複数本同時に利用することによって高いバンド幅と耐故障性を実現する RI2N（Redundant Interconnection with Inexpensive Network）というコンセプトを提唱し，それを実現するシステムを提案，実装してきた[1)]．RI2N とは，安価な複数本の Ethernet リンクとソフトウェアの拡張のみで高性能化と信頼性の向上を実現しようとするものである．具体的には各ノード間に複数本の Ethernet リンクを設置し，正常時にはデータのストライピングによってスループットを向上させる．そしてリンクが故障した場合には冗長なリンクを利用して通信を継続させる．しかし，これまでの実装システム RI2N/UDP [2),3)] は UDP/IP を利用したユーザレベルライブラリであったため，ソースの改変やリコンパイルが必要であった．そこで，本論文ではシステムレベルの実装によってユーザ透過に利用でき

†1 筑波大学大学院システム情報工学研究科
Graduate School of Systems and Information Engineering, University of Tsukuba
†2 筑波大学計算科学研究センター
Center for Computational Sciences, University of Tsukuba
*1 現在，富士通株式会社
Presently with Fujitsu Limited
*2 現在，筑波大学計算科学研究センター
Presently with Center for Computational Sciences, University of Tsukuba

る新たなシステムを提案する．本システムは Linux kernel 2.6 における仮想的な Ethernet デバイスとして動作し，既存の TCP/IP レイヤをそのまま利用して高バンド幅化と耐故障性を実現する．本論文では，まず Ethernet のマルチリンクを用いたシステムについて関連する研究を紹介する．そして，マルチリンク Ethernet 結合システムにおける性能の問題について述べ，これを解決するシステムとして RI2N/DRV を提案する．その後，RI2N/DRV の実装について述べ，実環境での評価について述べる．最後に考察について述べ，まとめとする．

2. 関連研究：マルチリンク Ethernet の利用

Ethernet におけるマルチリンクの利用をノード間にも適用するドライバソフトウェアとして Linux Channel Bonding [4] がある．元々HPC クラスタ向けに開発されたものであり，デフォルト設定のモードである balance-rr（round-robin）モードは本研究の目的とする姿に近い．しかし実際には，パケットの到達順序の入れ替わりが頻繁に発生してしまうために TCP/IP では性能を十分に発揮することは難しいという指摘もなされている[4]．この問題については 3 章で詳しく述べる．また，Linux Channel Bonding ではモジュールロード時に設定する少数のノードに対して定期的に各リンクで ARP 要求を送って，ARP 応答が得られるかどうかでリンクの故障判断をしている．これは既存の仕組みをうまく利用した方法ではあるが，検出可能な故障の種類や箇所に制限が多く，耐故障性については十分に配慮されていないと考えられる．

IEEE802.3ad [5] は Ethernet のスイッチ間を複数のリンクで接続し，それを 1 つの仮想的なリンクとして利用することによってスイッチ間バンド幅を拡張する規格である．本来スイッチ間の規格であるが Linux Channel Bonding を利用することでノード–スイッチ間でも利用することができる．しかし，IEEE802.3ad の制限から 1 つのコネクションに属するパケットを複数のリンクに分散させることができないため，多数のノード間のランダム通信でなければすべてのバンド幅を利用できない．また，結合した複数のリンクは単一のスイッチにすべて接続されていなければならない．そのため，スイッチが single point of failure となり，1 つのスイッチが故障するだけで通信が停止することになる．

PM/Ethernet [6),7] は HPC クラスタにおけるノード間通信のための軽量な通信ライブラリである．複数のリンクを同時に利用する機能（PM/Ethernet Network Trunking，以下，PM/Ethernet NT）も持っており，レイテンシ，スループットの面で高い性能を示している．PM/Ethernet NT ではあらかじめ到着順序の入れ替わりが頻繁に発生することが考慮されており，システム内部で順序制御を行うため，Linux Channel Bonding で TCP/IP を用いた場合のような性能低下はない．しかし，PM/Ethernet では性能を追求する一方，専用の API を用いることで可搬性を低下させている．また，現在のところ耐故障性を向上させる機能は持っていない．

本研究では IEEE802.3ad のような標準規格から離れてもより高い性能と耐故障性も目指す．しかし，PM/Ethernet のように完全に専用の API を用いることは，アプリケーションの可搬性を低下させることになるため避けたい．そのため本研究では，Linux Channel Bonding のように既存の API を維持したまま高性能化を実現させることを目指す．また，特定のスイッチのハードウェア/ファームウェア規格に依存せず，一般的な Layer-2 スイッチの集合で構成される汎用ネットワーク上で動作することを目指す．

3. マルチリンク Ethernet の性能問題

複数の物理的なリンクを有効に使って高いスループットを実現する最も単純な方法は，それぞれのリンクを均等に利用するラウンドロビン方式でパケットを送出するというものである．しかし，実際のネットワーク上でラウンドロビン方式でパケットを送出するとパケット順序の入れ替わりが頻繁に発生する．TCP/IP などネットワーク層のプロトコルは，下位レイヤで順序の入れ替わりが少ないことを想定して作られているため，TCP/IP とマルチリンク Ethernet 結合システムを組み合わせた場合，マルチリンクによる物理レベルでのバンド幅の向上を効率的に利用できない．

本章ではマルチリンク Ethernet でパケット順序の入れ替わりが頻発する原因と，そのときの TCP の振舞いについて説明する．

3.1 パケット順序入れ替わりの原因

マルチリンクでのラウンドロビン送出によってパケット順序の入れ替わりが起こる原因は 2 つある．1 つは，スイッチのパケット処理速度や信号の伝搬遅延などが経路ごとにわずかに違うために，経路ごとにパケットの転送に要する時間が変化する “物理的な到着順序の入れ替わり” である．これは物理的に異なる経路を利用しているため完全に防ぐことは難しい．しかし，HPC 向け PC クラスタのネットワークは非常に短距離のネットワークであり，スイッチのホップ数も少なく，スイッチや NIC などのハードウェアもすべて同じ機種が用いられていることが多い．また本システムでは，複数のリンクが平行に張られていて，論理的に等価な経路でネットワークが構成されることを想定している．このような場合，各リンクでの通信時間の差は非常に小さくなると考えられ，その差が 1 パケット分（MTU1500,

1 Gbps で 12 μs）を超えるほどあるとは考えにくい．そのため，本研究が対象としているHPC クラスタのネットワークではこの種の原因によるパケット順序の入れ替わりは少ないと考えられる．

もう 1 つの要因は，バッファの利用によって引き起こされる順序の入れ替わりである．通信経路上では多数のバッファを用いるが，その中でも影響が最も大きいと考えられるのが，受信側の NIC のバッファから OS の受信キューにパケットを挿入する部分である．一般的な PC クラスタにおいて典型的に用いられていると考えられる Intel 製 GbE NIC のドライバでは，NIC のバッファに複数のパケットがあればそれらはまとめて 1 度に処理され，その順序のまま OS のキューに挿入されることになる．また，多くの GbE カードではパケット到着時に OS に対してハードウェア割込みをかけるのを意図的に遅らせる（interrupt coalescing）機能を持っている．これにより，バッファにより多くのパケットを蓄積し少ない割込み回数でまとめてパケットを処理することで CPU 負荷を軽減することができる．しかし，このバッファ機能とマルチリンクのラウンドロビン送出が同時に用いられると，**図 1** のようにパケット順序の入れ替わりを引き起こす．たとえ，実デバイスにパケットが届く順番が 101 (NIC1), 102 (NIC2), 103 (NIC1) ... のように送出順番通りになっていたとしても，それらは一度 NIC ごとに用意されたバッファに蓄積され，まとめて処理されることになる．そのため，OS の受信キューに挿入される時点では 101, 103, ... のように各 NIC バッファ内での順序で処理されることになり，送信時と受信時でパケット順序の入れ替わりが生じる．このようなバッファによる順序のずれが本質的な問題であると考えられるが，バッファリングを排除することはできない．通信経路上のハードウェアバッファはもちろんのこと，NIC のバッファリング処理も，これがなければ著しい性能低下と CPU 負荷の増大をまねく．

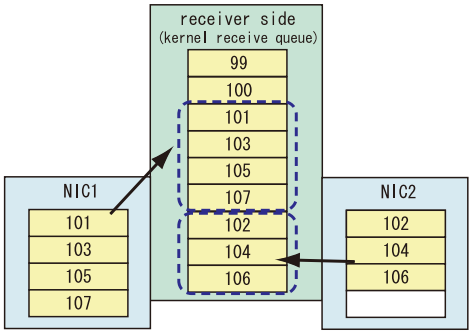

図 1　バッファによるパケット順序の入れ替わり
Fig. 1　Packet misordering on multiple buffers.

3.2　TCP の動作

パケット順序が入れ替わった場合の TCP の振舞いについて述べる．TCP のデータの受信側では，順番の入れ替わったパケットを受信すると即座に ACK パケットを生成，送信する．これは送信側でこの ACK 情報をもとにパケットロスの検出やフロー制御を速やかに行うためである．そのため，最初の 1 回だけではなく不連続なパケットが届くごとに毎回 ACK が送信される．データの送信側（ACK パケットを受信する側）では，ACK 番号が同じパケットを一定数（=*reordering*）以上受信すると，その番号のパケットが消失したと判断する．*reordering* は初期の TCP では 3 という固定値であった．そのため，4 回以上の ACK はすべてパケットロスであると判断され輻輳ウインドウサイズの低下を招いていた．しかし，現在の拡張された TCP では SACK（選択的確認応答）[8),9)] の導入などによって，パケットロスと順序の入れ替わりを分離することができるようになった．そのため現在では，送信側の TCP 層でネットワーク中の順序入れ替わりの大きさを推定し，*reordering* を適宜最適な値に変更している[10)]．そのため，順序の入れ替わりが頻繁に起こった場合にも現在の TCP では高いスループットを発揮することができる．

しかし，実際には *reordering* の推定が有効に動作せずスループットが低下する場合がある．断続的に通信を行う場合やメッセージサイズが頻繁に増減するような場合がこれに該当する．これらの通信ではスループットが一定ではなく，結果としてパケット順序のずれの大きさが変動しやすくなる．推定した *reordering* よりも実際のずれが大きかった場合，パケットは消失したと判断され輻輳ウインドウサイズが縮小されスループットが低下する．バースト転送などスループットが一定である程度長い時間継続する通信では *reordering* は精度良く推定されるため，高いスループットが期待される．その一方で，現実的なメッセージサイズにおける通信では *reordering* の推定が十分に機能する前に通信が終了することが考えられ，その場合，十分なスループットは得られない可能性が高い．このため，順序入れ替わりが頻繁に起こる環境で TCP を用いた場合，ping-pong 通信での半性能長が非常に大きくなると考えられる．

また，Linux Channel Bonding で TCP を利用した場合，通信速度には直接影響しないが，CPU やネットワークなどシステムに対する負荷が大きくなる．これは，処理負荷の大きな SACK オプションがほぼつねに利用されること，ほぼすべてのデータパケットに対し

て ACK パケットが即座に返されることなどが原因である．SACK オプションは順序入れ替わりの多いネットワークで性能を向上させるために必要不可欠な機能であるが，本来の TCP の確認応答機構である累積確認応答（cumulative ACK）に比べて処理が複雑で負荷が大きい．また，ACK の数が増大する問題は，“順序がずれたパケットが到着すると即座に ACK を返し送信側でフロー制御を行う”という現在の TCP のフロー制御方式の基本部分が原因であり，現在の TCP の拡張ではこの問題に対処することは難しいと考えられる．

3.3 解 決 方 法

前述のようにパケット順序が入れ替わる原因は明らかであるが，それ自体を取り除くことは難しい．また，現在の拡張された TCP 実装であっても性能を得るためにはメッセージサイズの制限が大きく，CPU やネットワークに大きな負荷をかけることになる．

このような，マルチリンク Ethernet 結合システムでの問題を解決する方法として，順序が入れ替わったパケットをドライバレベルで並べ替えて，パケット順序の入れ替わりを解消するという方法が考えられる．TCP よりも下のレイヤで順序の入れ替わりを吸収し，TCP では順序のそろったパケットとして処理する．Ethernet リンク結合システムであればドライバモジュールとして実装することも可能であり，その内部で新たに順序制御を行うこともできる．

他の解決方法として，TCP/IP などのプロトコル処理に改良を加えてマルチリンクでの性能を向上させることも考えられる．しかし，TCP レイヤはカーネルと深く結び付いており，ユーザ透過性を失わずにプロトコルの改良を行うためには，カーネルソースを改変する必要がある．そのため，Ethernet リンク結合システムによる解決の方がより良い方法であると考えられる．

4. RI2N/DRV の設計

4.1 提　　案

現在の Ethernet のマルチリンクを利用したシステムでは，PM/Ethernet NT のように専用の API を用いなければマルチリンクの性能を十分に使いきることができない．しかし前述の解決方法のようにドライバ内部で順序制御を行うことによって，標準の TCP/IP を利用しながら高い性能を発揮するマルチリンク Ethernet 結合システムが実現可能であると考えられる．そこで我々は，この仕組みを利用しユーザ透過に利用可能でより高性能，かつ耐故障性をサポートした Ethernet リンク結合システム RI2N/DRV を提案する．

仮想ネットワークデバイスとしてローダブルモジュールによって導入可能なシステムとする．システム全体の基本的な仕組みは Linux Channel Bonding で balance-rr モードを用いた場合に類似する．**図 2** に RI2N/DRV の基本的な仕組みを示す．TCP/IP のパケットが RI2N/DRV によって作成された仮想的なデバイス（以下 RI2N デバイス）に送られる．RI2N デバイスではこのパケットを実際に存在する複数の NIC に分散して送信する．受信側では各 NIC で受信されたパケットを RI2N デバイスで集約し IP 層にパケットを渡す．このような動作は Linux Channel Bonding と非常に近い．しかし，Linux Channel Bonding や IEEE802.3ad でのリンク結合のように非常に単純な処理のみを行うのではなく，RI2N/DRV では順序入れ替えのような比較的複雑な処理を行いある程度のオーバヘッドがあることを前提として性能改善を目指す．

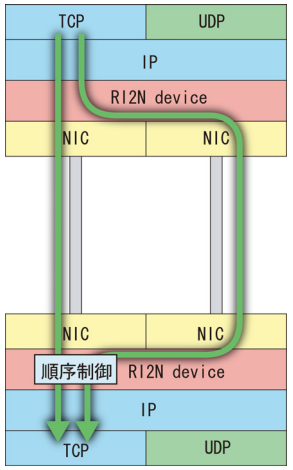

図 2　RI2N/DRV の仕組み
Fig. 2　Packet transfer with RI2N/DRV.

4.2 物理ネットワークの構成

IEEE802.3ad でマルチリンクネットワークを構成する場合，**図 3** のような構成となる．これは Linux Channel Bonding と合わせてノード–スイッチ間のバンド幅も拡張した場合の例である．8 ポート以上のスイッチであれば図のように 2 つのスイッチを利用せずとも接続は可能であるが，これを 1 台にまとめた場合でも，この図のように 2 つに分けた場合でも，スイッチがいずれか 1 台でも故障するとすべての通信が停止する．このように，IEEE802.3ad で構成したマルチリンクでは 1 つのスイッチが single point of failure となる．

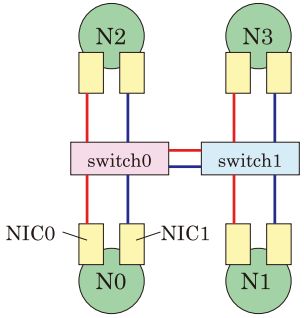

図 3　IEEE802.3ad でのネットワーク構成
Fig. 3　Network construction with IEEE802.3ad.

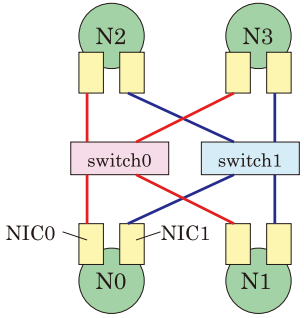

図 4　RI2N/DRV でのネットワーク構成
Fig. 4　Network construction with RI2N/DRV.

このようなことから，RI2N/DRV では複数のネットワークリンクが平行に接続されたネットワーク構成を想定する．**図 4** に構成例を示す．この図では switch0 と switch1 のそれぞれに接続された 2 つの独立したネットワークがすべてのノードにそれぞれ接続されている．このようなネットワーク構成であれば，それぞれのネットワーク系統が独立しており，いずれかの系統が完全に停止したとしても他の系統に影響を与えない．そのため，前述のような single point of failure はなく，このような構成は耐故障性の面で有効に機能する．RI2N/DRV ではこのような構成でネットワークを構築できるようにする．

4.3　複数リンクの利用

RI2N/DRV ではスループットの向上と耐故障のために複数のリンクを同時に利用する．

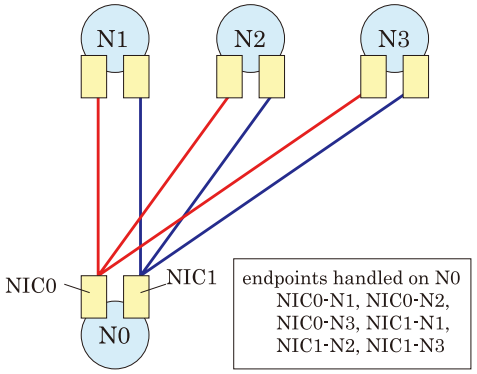

図 5　ノード N0 に対応する endpoint
Fig. 5　Endpoints handled on N0.

Linux Channel Bonding の balance-rr と同様に，複数のリンクを偏りなく利用するためラウンドロビン方式でリンクを選択する．これにより 1 つのコネクションにおけるバンド幅を効率的に上昇させることができる．IEEE802.3ad では 1 つのコネクションを 1 つのリンクに固定することを推奨しているため，本システムのように 1 つのコネクションにおけるバンド幅を向上させることはできない．

ラウンドロビンで利用するリンクを選択する場合，いずれかのリンクが故障したとしても TCP の再送によって通信は継続することができる．しかし，そのままでは性能が著しく低下するため，早期に故障を検出し故障したリンクは利用を停止する必要がある．

4.4　故障と回復の検出

RI2N/DRV では，あるノードにおける通信相手と利用するリンクの組合せを endpoint と呼び，endpoint ごとに故障情報の管理を行う．**図 5** に endpoint の例を示す．この図の例ではノード N0 における endpoint は合計で 6 つとなる．

RI2N/DRV では通常の通信パケットを使って故障の検出を行う．故障検出には RI2N/UDP [2] と同様に，受信パケット数の偏りを評価する方法を用いる．この方法は，スループットが高い状態，つまり積極的に通信を行っている場合に即座に故障の検出ができるという利点がある．endpoint ごとに独立したカウンタを用意し，対応する endpoint から受信したパケット数を記録する．いずれかのカウンタが一定数（=*FT_HTHRESH*）を超えたとき，同じ通信相手に属する endpoint のカウンタのうち値が一定数（=*FT_LTHRESH*）

以下であるものに対応するリンクを故障と判断する．

また，正常な状態でもバッファの挙動によって短時間の偏りが発生し，誤ってこれを故障と判断する可能性がある．これを防止するため，endpoint ごとにパケット受信時刻を記録し，最後にパケットを受信してから一定時間（=*SLOW_TOUT*）の間は故障と判断しない．最後にパケットを受信してから *SLOW_TOUT* 以上の時間が経過し，かつカウンタ値が一定数（=*FT_LTHRESH*）よりも小さい endpoint を故障と判断する．この処理が受信パケット数の変化に対する low-pass filter となり故障の誤検出を抑制する．

RI2N/DRV ではハートビートによる故障検出も行う．TCP/IP ではパケットロスがあった場合に急激にスループットが低下する．そのため，前述の受信パケット数に偏りが生じる前にパケットの送出がほぼ停止した状態となる可能性がある．このような状態では故障検出に必要な受信パケット数（*FT_HTHRESH*）を超えることが難しい．また，この状態ではデータパケットの大部分が再送タイムアウトによって再送される状態となる．TCP の再送タイムアウト時間は 1 秒以上であるため，故障検出に要する時間は数秒から数十秒単位と大きくなる．このように，著しくスループットが低下した状態を避けるためには，ハートビートによるアクティブな故障検出が必要となる．そこで RI2N/DRV では，過去に通信を行ったノードを記憶しておき，そのノードに対して一定間隔（=*HBSPAN*）でハートビートパケットを送信する．そして，このパケットを受信パケットカウンタに加えることで過去に通信を行ったすべてのノードとの間でハートビートによる故障検出が可能となる．しかし，ハートビートパケットをむやみに増加させることを抑制するため *HBSPAN* には数秒程度の値を設定することを想定している．そのため，前述の問題と同様に，故障と判断できるだけのパケットが集まるまで非常に長い時間がかかる可能性がある．そこで，*HBSPAN* はそのままで，ハートビートパケット 1 パケットあたりのカウンタ増加量を *HBWEIGHT* にすることで故障の検出をより短時間で行えるようにする．

故障が検出された endpoint については故障したことを示すフラグを立て，パケット送信時にはその endpoint を選択しない．

また，故障からの回復も自動的に検出を行う．これにはハートビートを利用する．故障検出のために利用しているハートビートをそのまま利用し，故障しているかどうかに限らずすべての endpoint にハートビートパケットを送る．ハートビートパケットが受信できた endpoint は，故障していない，もしくは故障から回復していると判断できる．回復を検出した endpoint からは故障フラグを取り除き，次回のパケット送信から通常のデータパケットでの利用を再開する．

4.5　順 序 制 御

3.3 節で述べたようにようにマルチリンク環境で高いスループットを維持し続けるにはいずれかの層でパケット順序の入れ替わりを吸収しなければならない．RI2N/DRV ではパケットの並べ替えをドライバ内で行うことでマルチリンク環境でも安定した高いスループットを実現する．

RI2N/DRV では各パケットにシーケンス番号をつけて送信し，それを使って受信側でパケットの並べ替えを行う．シーケンス番号は通信相手ノードごとにそれぞれ独立に管理する．RI2N/DRV の順序制御は性能改善を目標としたものであり，TCP の順序制御のように順序の保証をするためのものではない．そのため，性能低下を招く可能性のある確認応答や再送までは行わない．“順序の保証” には再送処理が必要であり，そのためには到達確認と到達確認が到着するまで送信側でパケットを保持しておくことが不可欠である．これらは TCP ですでに行われていることであり，ドライバ内で再度これを実装することは無駄にオーバヘッドを増やすことになる．そのため RI2N/DRV では，パケットロスにより再送が必要な場合には TCP にその処理を任せる．

しかし，RI2N/DRV 自体は再送を行わないため，パケットロスが発生すると，どれだけ待ってもパケット順序がそろわなくなる．TCP で再送が行われたとしても，RI2N/DRV のシーケンス番号が異なるためこのパケット列の復元はできず，この問題は回避することができない．そのため，各パケットの到着時刻をそれぞれ記録しておき，一定時間（=*FAST_TOUT*）以上経過したパケットは順序が揃わない場合にも上位層へ渡すこととする．このようにすることで，順序が揃わないことによって発生する通信の停止を避けられる．

4.6　パケットロスの早期検出手法

前節で述べたとおり，RI2N/DRV では順序制御の際にロストパケットを待ち続けて通信が停止することを防止するため，受信後 *FAST_TOUT* 以上が経過したパケットを順序によらず上位層へ渡す．これで通信の停止は避けられるが，パケットロスがあるたびに *FAST_TOUT* だけ待つ必要がある．そのため，パケットロスの多い状況では著しく性能が低下する可能性がある．

そこで，時間計測によらず解析的にパケットロスを検出する手法を提案する．1 つのリンクではパケットの追い越しがないと仮定し，シーケンス番号 A のパケットが届いていない場合に，シーケンス番号が A+1 以降のパケットが受信側のすべての NIC で受信できればパケット A は消失したと判断するという方法である．**図 6** の例で説明する．図の左から順にパケットが届いている場合に 103 番のパケットが消失したとすると，102 番の次に到着

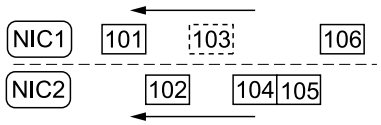

図 6　パケットロスの早期検出
Fig. 6　A fast-detect method for packet losses.

するのは 104 番，105 番のパケットである．1 つのリンクではパケットの追い越しがないと仮定すると，これら 103 番よりも後のシーケンス番号を持ったパケットが NIC2 から受信できたため，NIC2 では 103 番のパケットはこれ以降も受信できないことが分かる．次に，106 番が NIC1 で受信される．このことから，103 番はこれ以降 NIC1 でも受信されないことになる．NIC1 でも NIC2 でも受信されないことが確認されたので 103 番は消失したと判断することができる．

この手法は受信側のみの処理で実現することができる．ただし，この方法でも図 6 の 105 番，106 番のように各リンクの最後のパケットが消失した場合には検出することができない．そのため，時間計測による検出も併用する必要がある．

5. 実　　装

5.1 概　　要

RI2N/DRV では 1 つの仮想 Ethernet デバイス（RI2N デバイス）を OS に追加し，そのデバイスに上位層から渡されたパケットを複数の実デバイスに分散させて送信する．そして，受信側では複数のデバイスで受信されるそれらのパケットを 1 つの RI2N デバイスから受信されたかのような形にして上位層へ渡す．

また，物理デバイスのドライバやカーネル本体にも変更を加える必要はなく，ローダブルモジュールの形で導入する．アプリケーションの通信としては基本的に TCP が利用されることを想定するが，UDP や ICMP，ARP などでも利用できる．

RI2N/DRV は Linux 上でのデバイスドライバとして動作するため次のような実装方法を利用する．

パケットの扱い　Linux のネットワークプロトコル処理では物理デバイスからソケットまで sk_buff 構造体という共通のフォーマットでパケットを扱っている．そのため，プロトコル階層間のパケットの受け渡しもメモリコピーを介さず，この構造体の参照情報を受け渡すことで実現されている．RI2N/DRV ではこの sk_buff 構造体 1 つを単位として処理を行う．送受信の処理も，sk_buff ごとにそれぞれのハンドラが起動される．

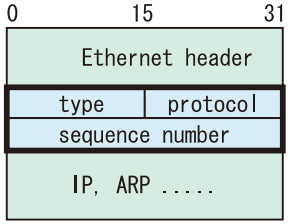

図 7　ヘッダの拡張
Fig. 7　Expansion of a packet header.

時間の扱い　RI2N/DRV では時間経過によって一部の処理を行う．時間の測定にはカーネルのタイマ割込みカウンタである jiffies の値を利用する．記録したい時点でそのときの jiffies の値を記録しておき，現時点での jiffies との差分を求めることにより経過時間を得る．jiffies の分解能はカーネルのタイマ割込み周期となる（今回の評価で利用したカーネルでは 1 ミリ秒単位となっている）．

5.2 RI2N ヘッダ

RI2N/DRV ではパケット順序を受信側のドライバ内部で並べ替えて，順序どおりに直したうえで上位層にパケットを渡す．そのためには送信時の順序を正確に知る必要がある．RI2N/DRV では各パケットにシーケンス番号を付与することでこれを実現する．そのため，RI2N/DRV では**図 7** のように Ethernet ヘッダとペイロードの間に 8 byte のヘッダ領域を新たに設ける．これを RI2N ヘッダと呼ぶ．*type* フィールドはパケットの種類を示すもので，ハートビートパケットと通常のパケットを区別するために用いる．*protocol* フィールドはネットワーク層のプロトコル番号を格納する領域である．通常は Ethernet ヘッダ内に格納されるものであるが，RI2N/DRV では Ethernet ヘッダ内には RI2N/DRV のパケットを示す固有の番号を格納する必要がある．その代わりに RI2N ヘッダ内に実際の上位層のプロトコル番号を保存する．また，*sequence number* は送信時のパケット順序を示すシーケンス番号で，1 パケットごとに 1 ずつ増やす．

RI2N ヘッダをつけたパケットも Ethernet パケットとして規格に沿ったパケットであり，同一の Layer-2 スイッチ上で既存の Ethernet による通信と共存することができる．しかし，IP や ARP などネットワーク層のパケットとしては扱うには RI2N ヘッダを処理する必要があり，RI2N デバイスを持たないノードとの通信では利用することができない．

また，RI2N ヘッダの挿入によってヘッダ領域が 8 byte 増加することになる．そのため，RI2N/DRV では実デバイスよりも MTU が 8 byte 小さくなり，実デバイスの MTU が 1,500 byte の場合最大スループットが 0.5%ほど低下する．本ネットワークシステムは高バンド幅を志向しており，ある程度長いメッセージ長において高性能が達成されることを目的としている．したがって，このヘッダ長の拡張は無視できると考えられる．

5.3　送信時の処理

送信処理は RI2N デバイスの送信ハンドラとして実装する．ネットワークデバイス構造体（net_device）の hard_header 関数と hard_start_xmit 関数がこれに相当する．前者はそのデバイスのヘッダを作成する関数である．この関数が実行された後，後者の関数で最終的な送信処理を行う．

RI2N デバイスの送信処理では主に次のような処理を行う．

ヘッダの構築　ネットワーク層のパケットを受け取って RI2N ヘッダと Ethernet ヘッダを構築する．このとき，Ethernet ヘッダに格納するプロトコル番号は RI2N/DRV のパケットであることを表す番号（0x1004）に変更し，本来のネットワーク層のプロトコル番号は RI2N ヘッダ内の protocol フィールドに格納する．

送出デバイスの選択　デバイスロード時に指定したものの中からラウンドロビンで選択する．通信相手ノードごとに最後に送信に利用したデバイスを記録しておき，次回の送信時にはその情報をもとに別のリンクを利用する．選択したリンクに該当する endpoint に故障フラグがセットされていた場合やそのリンクの送信キューが詰まっていた場合は，さらにその次のリンクを選択する．

実デバイスのキューへ挿入　選択した実デバイスのキューへ dev_queue_xmit 関数によってパケットを挿入する．送信可能なリンクが見つからなかった場合はパケットを破棄する．

5.4　受信時の処理

RI2N/DRV の受信側では故障検出のためのパケットカウントやパケット順序の並べ替えを行わなければならない．しかし，受信側で実デバイスに到着したパケットはハードウェアの専用ドライバによって受信処理が行われ，そのまま OS のプロトコルスタックに渡される．このままでは，RI2N デバイスにパケットが渡されることはなく，受信処理を行うことができない．そこで RI2N/DRV では RI2N ヘッダを処理するためのプロトコルハンドラ（ri2n_skb_recv）を OS に追加しその中で受信処理を行う．これは Linux の VLAN ドライバでも用いられている方法である．**図 8** にプロトコルハンドラによる受信処理の様子を示す．NIC で受信された RI2N のパケットは NIC から直接 OS の受信キューに挿入される．

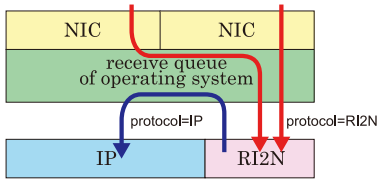

図 8　プロトコルハンドラによる受信処理
Fig. 8　Packet reception mechanism with a protocol handler.

受信キューでは各パケットのプロトコル番号によって次にどのような処理を行うかを決定している．RI2N デバイスから送信されたパケットの Ethernet ヘッダには RI2N のパケットであることを示すプロトコル番号が格納されているため，これらのパケットはすべて RI2N のプロトコルハンドラに渡される．ここで RI2N/DRV で必要な受信処理を行った後，プロトコル番号を IP など本来のネットワーク層の番号に変更して再度 OS の受信キューへ挿入する．今度はパケットのプロトコル番号が IP など通常のプロトコルのものになっているので OS はプロトコルごとにそれらに必要な処理を行う．

受信側で行う主な処理は次のとおりである．

RI2N ヘッダの処理　受信ハンドラが呼び出された状態では，パケットのデータ領域ポインタの示す位置に RI2N ヘッダが存在する．このポインタを RI2N ヘッダの終わりまで移動させ，IP など本来のネットワーク層のヘッダを指すようにする．

プロトコル番号の書き換え　ネットワーク層のプロトコル番号が RI2N/DRV のパケットを表すものに書き換わっているため，これを本来のプロトコル番号に戻す．このとき Ethernet ヘッダのプロトコル番号領域に加えて，sk_buff 構造体の protocol フィールドも同様に書き換える．

受信デバイスの書き換え　sk_buff 構造体にはどのデバイスで受信されたかが記録されているため，これを RI2N デバイスで受信されたかのように書き換える．この情報を書き換えることで OS の ARP 処理がそのまま利用できる．

故障検出のための処理　どの NIC で受信され，どのノードから送信されたものであったかに合わせて，該当する endpoint のパケットカウンタを増加させる．また，このときの時刻を記録する．

順序の並べ替え　RI2N ヘッダのシーケンス番号によってパケット順序を並べ替える．

5.5　順序並べ替えの手順

RI2N/DRV の受信処理はプロトコルハンドラとして実装するため，OS からは sk_buff 構造体でパケットが渡される．RI2N/DRV では，この sk_buff を再度 OS のキューに挿入することで，IP など本来のネットワーク層の処理を行う．このときに OS のキューへ構造体を渡す順序を変更することで順序の並べ替えを行う．順序は RI2N ヘッダのシーケンス番号によって決定する．通信相手ごとにこれまでに処理の終わった最大のシーケンス番号を保持しておき，その番号の次の番号を持ったパケットが来た場合にはすぐに OS の受信キューへ渡す．そうでない場合は，通信相手ごとに用意したリンクリストにその sk_buff 構造体を追加し受信処理を終了する．このリンクリストはシーケンス番号順に接続されており，sk_buff 構造体に対する参照情報と jiffies から取得した受信時刻を保持する．

OS の受信キューに sk_buff 構造体を渡した後は，リンクリストを参照して順番が揃ったパケットがないかを確認する．順番が揃ったパケットがあればそれも同様に OS の受信キューへ渡し，以降同様にリンクリスト中の順番が揃ったパケットがなくなるまで繰り返す．またこの際，パケットを受信してから *FAST_TOUT* 以上の時間が経過しているパケットは順番が揃っていなくても OS の受信キューへ渡し，処理済みシーケンス番号もこのパケットのシーケンス番号まで増加させる．そのため，このパケットよりもシーケンス番号が小さいパケットも順番が揃っているかどうかによらず OS の受信キューへ渡される．また，*FAST_TOUT* 間隔で起動するタイマを OS に登録し，全通信相手のリンクリストに対して同様の処理を行う．

前述のとおり RI2N/DRV では sk_buff 構造体を参照するリンクリストによってパケットの並べ替えを行う．そのため，パケットのデータサイズに依存したメモリ領域の確保を必要としない．しかし，実際にはカーネル内に処理待ち状態の sk_buff 構造体を増加させることになるため，リンクリストに保持されるパケットの数については考慮する必要がある．そこで RI2N/DRV では，リンクリストに保存できるパケット数を相手ノードあたり最大 *MAXBUF* に制限する．これ以上のパケットを保存しようとした場合には，1 度すべてのパケットを OS の受信キューに渡してリンクリストをクリアする．

5.6　アドレス情報の扱い

RI2N デバイスにはそれぞれ独立した MAC アドレスを割り当てる．この MAC アドレスは実デバイスと重なっていても RI2N/DRV としては問題なく動作するが，MAC アドレスの重なった実デバイスを RI2N/DRV と独立に利用することはできなくなる．そのため，ユーザにより明示的な割当てがなければカーネルの random_ether_addr 関数によって割り当てられたランダムなアドレスとして利用する．この MAC アドレスは通常の Ethernet デバイスと同様に ARP によって IP アドレスと対応付けられるため，参加ノード間であらかじめ MAC アドレスを交換しておく必要はない．そのため，RI2N/DRV の利用にはクラスタ全体の IP アドレスや MAC アドレスを把握するための設定ファイルなどは必要ない．

5.7　ハートビート

RI2N/DRV では故障と回復の検出にハートビートを利用する．ハートビートパケットはアプリケーションからの要求とは独立に RI2N/DRV が自発的に送信するパケットである．カーネルタイマに *HBSPAN* 間隔で起動するハンドラを追加し，そのハンドラ内でハートビートパケットを送信する．送信先は RI2N デバイスが保持しているすべての endpoint となる．ハートビートパケットは数秒間に 1 回という，通常のパケットよりも非常に低い頻度でやりとりされるため性能に対する影響は低いと考えられる．

受信側では，ハートビートパケットが送られてきた endpoint の故障フラグをオフにする．元々故障フラグがオフの場合，endpoint の受信パケットカウンタを *HBWEIGHT* だけ上昇させる．

5.8　リソース消費量

RI2N/DRV では複数のテーブルを参照し，故障情報の管理やパケット順序制御，リンクの選択などを行っている．その中で最もメモリ消費量の大きな部分は，順序制御のための受信パケットのバッファであると考えられる．ただし，この領域は RI2N/DRV によって明示的に確保する領域ではなく，NIC ドライバで作成され OS 内部で管理されている sk_buff 構造体のデータ領域を利用する．sk_buff 構造体のデータ領域の大きさは，受信されたデバイスの MTU サイズにそのデバイスのヘッダサイズを加えたものである場合が多い．そのため，Ethernet では 1 つの sk_buff 構造体あたり 1,514 byte になる．また，sk_buff 構造体自体の大きさが 232 byte となっており，RI2N/DRV 内部でのリンクリスト構造体も合わせて 1 パケットあたり全体としておおむね 1,800 byte を消費していることになる．GbE 2 本でのバースト転送では定常的に 12 パケット程度の順序のずれが発生しており，バッファ全体の容量は 20 KB 程度となる．また，このバッファは通信相手ノードごとに用意するため同時に通信を行うノード数が増えると消費するバッファの容量も増加する．

故障情報や通信相手ノードのリスト，ノードとリンクの組合せとなる endpoint のリストなどは統合して管理している．まず，通信相手ノードを表す node 構造体のハッシュテーブルがあり，MAC アドレスをキーとしている．node 構造体には，シーケンス番号や該当ノードの MAC アドレス，endpoint のリストに対する参照，受信時の並べ替えバッファへ

表 1　測定環境
Table 1　Evaluation environment.

項目	スペック
CPU	Intel Xeon 5110 1.6 GHz dual core
memory	DDR2 2048 MB
kernel	linux 2.6.22
NIC	Intel PRO1000PT dual port 1000 base-T
NIC driver	Intel PRO/1000 Network Driver 7.3.20-k2-NAPI
MPI	OpenMPI 1.2.4
bonding driver	Ethernet Channel Bonding Driver, v3.1.3
switch	Dell Power Connect 5324

の参照などを持っており，48 byte で構成される．endpoint 構造体は，パケットカウンタやリンク（実デバイス）への参照，故障情報，最終受信時刻を保存するためのもので 32 byte で構成される．全 node 構造体の合計メモリ使用量は node 構造体の数に比例する．それに対して，全 endpoint 構造体の合計メモリ使用量は，ノード数と RI2N/DRV が 1 ノードあたりに利用する NIC の数の積に比例する．図 4 のように 4 ノードのクラスタを 2 つの Ethernet リンクを使った RI2N/DRV で接続した場合に 4 ノードが全対全で通信したとすると，$4 \times (48 + 32 \times 2) = 448$[byte] が消費される．これは，MTU（1,500 byte）にも満たない値であり，受信パケットの並べ替え時に一時的に保管しなければならない sk_buff 構造体の大きさに比べればほとんど無視できる量であると考えられる．

6. 評　　価

実装した RI2N/DRV の通信性能の評価を行う．RI2N/DRV はデバイスレベルでの実装であるため，TCP，UDP を用いる幅広いネットワークアプリケーションに利用可能であるが，ここでは TCP/IP のソケット API を直接用いて作成した測定用のプログラムで評価を行う．比較対象として，単一の Ethernet リンクをそのまま利用した場合（以下，シングルリンク）と Linux Channel Bonding ドライバを用いた場合の測定も行う．それぞれの場合において，2 ノード間のスループットとレイテンシを測定する．また，通信中に意図的に故障を発生させ故障中にも冗長リンクを使って通信が継続できることを確認する．

測定環境は**表 1** に示す Xeon サーバ 2 台である．このサーバ 2 台を**図 9** のように GigabitEthernet スイッチで構成した独立した 2 つのネットワークで接続する．実デバイスの MTU は 1,500 byte とし，RI2N デバイスの MTU は 1,492 byte とする．Linux Channel Bonding ではヘッダの拡張が行われないためその仮想デバイスの MTU は実デバイスと同じく 1,500 byte である．本文中で述べた各種パラメータは**表 2** の値を用いた．今回利用した NIC，Intel PRO/1000 ではデフォルトの設定で割込み遅延機能が有効になっている．この設定によって性能が大きく変化するため，レイテンシとスループットの測定についてはデフォルトの設定（遅延割込みが有効）での測定と割込み遅延機能を無効にした場合の測定の両方を行った．割込み遅延機能を無効にするときのモジュールオプションは，RxIntDelay=0，RxAbsIntDelay=0，TxIntDelay=0，TxAbsIntDelay=0，InterruptThrottleRate=0，RxDescriptors=1024 とした．

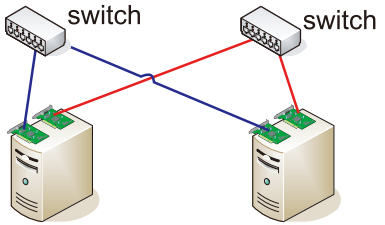

図 9　ネットワーク構成
Fig. 9　Network construction.

表 2　使用したパラメータ
Table 2　The used parameters.

名称	値	説明（節）
FT_HTHRESH	10	4.4
FT_LTHRESH	1	4.4
SLOW_TOUT	100 ミリ秒	4.4
HBSPAN	2 秒	4.4
HBWEIGHT	3	4.4
FAST_TOUT	20 ミリ秒	4.5
MAXBUF	300	5.5

6.1　レイテンシ

レイテンシの評価のためメッセージサイズを変化させ ping-pong 通信に要する round-trip time を測定した．NIC をデフォルトの設定のままで測定した結果を**図 10** に示す．また，割込み遅延機能を無効にした場合の結果を**図 11** に示す．各測定点において 5 万回の ping-pong に要した時間から平均値を求めた．

メッセージサイズによるレイテンシの変化は利用されるパケット数により大きく異なっ

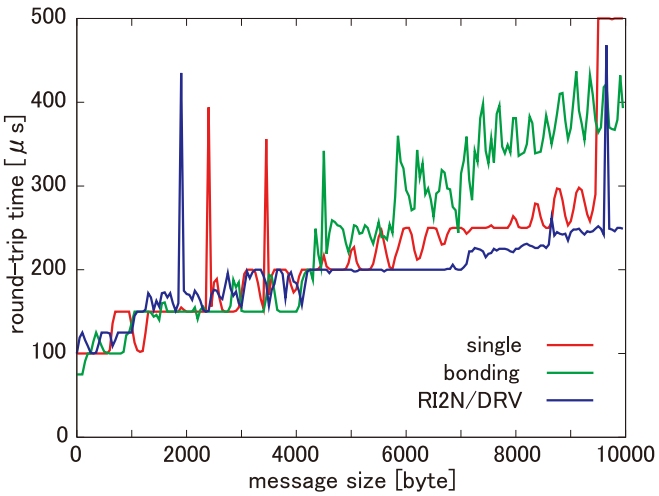

図 10　レイテンシの測定結果（割込み遅延あり）
Fig. 10　The result for latency (with interrupt coalescing).

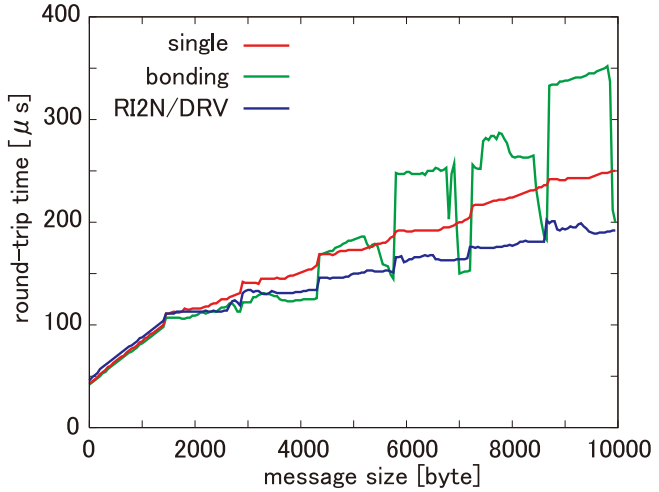

図 11　レイテンシの測定結果（割込み遅延なし）
Fig. 11　The result for latency (without interrupt coalescing).

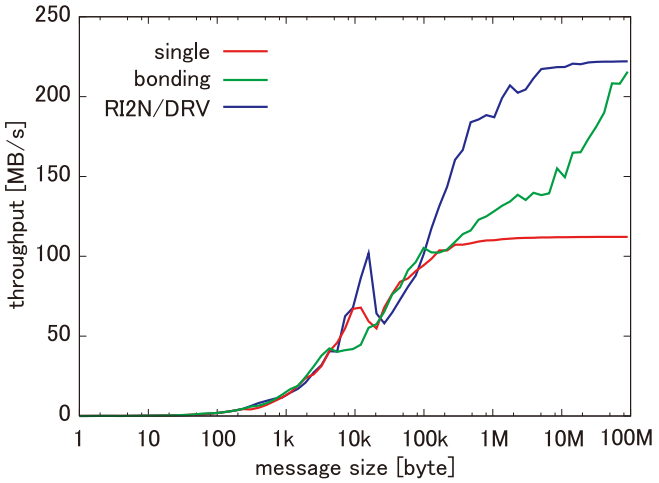

図 12　TCP によるスループットの測定結果（割込み遅延あり）
Fig. 12　Throughput for TCP (with interrupt coalescing).

ている．まず，メッセージサイズが 1,500 byte 以下の範囲では，シングルリンク，Linux Channel Bonding，RI2N/DRV のいずれも，メッセージサイズの増加にともなってほぼ線形に通信時間が増加している．

次の 1,500 byte から 4,500 byte 程度の領域では，2 つのリンクを使う RI2N/DRV と Linux Channel Bonding の方がシングルよりも通信時間が短い．変化の仕方はそれぞれ異なっているがパケット数の変わり目（3,000 byte 程度）で急激にレイテンシが増加する点は共通している．

4,500 byte を超える領域，つまり 4 パケット以上を利用する部分では，Linux Channel Bonding のレイテンシが大きく変化し，比較的長時間を要しているのに対して，RI2N/DRV は安定して高い性能を示している．Linux Channel Bonding のレイテンシは，利用パケット数が増加した直後は通信時間が著しく増加するが，パケット数増加の直前になると RI2N/DRV と同等かそれ以下となっている．

6.2　スループット

スループットについても ping-pong 通信においてメッセージサイズを変えて測定した．NIC をデフォルトの設定で利用した，割込み遅延がある状態での測定結果を**図 12** に示す．また，レイテンシと同様に NIC の割込み遅延を無効にした状態での結果を**図 13** に示す．

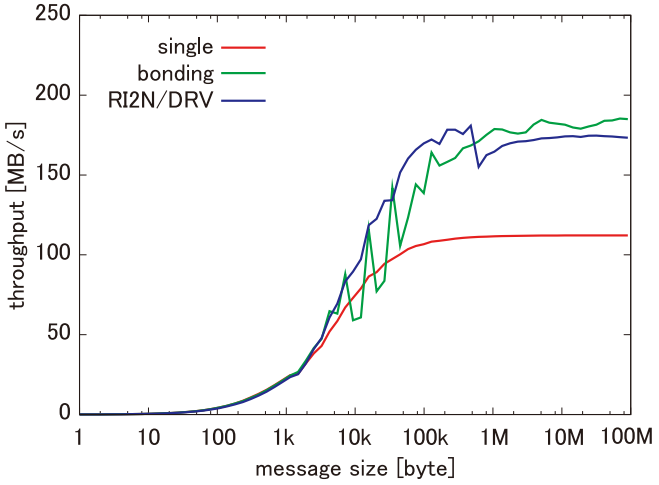

図 13　TCP によるスループットの測定結果（割込み遅延なし）
Fig. 13　Throughput for TCP (without interrupt coalescing).

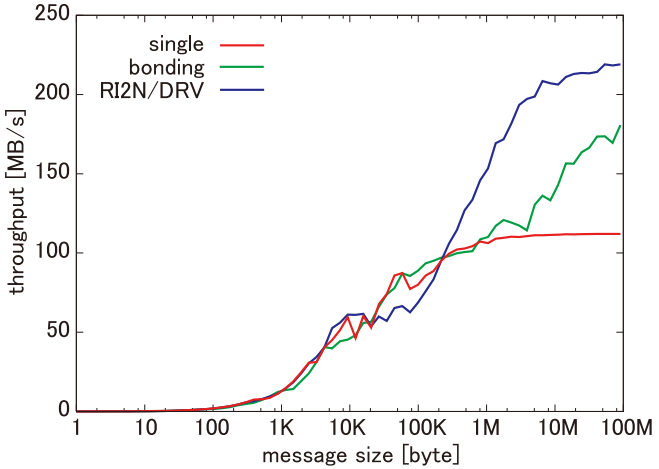

図 14　MPI によるスループットの測定結果
Fig. 14　Throughput for MPI.

シングルリンクの最大スループットが 112 MB/s，RI2N/DRV では 222 MB/s，Linux Channel Bonding では 215 MB/s であった．Linux Channel Bonding ではメッセージサイズが 100 Mbyte になっても，スループットは上昇傾向にあり最大値には到達していない．別途行った片方向通信でのスループット測定では RI2N/DRV ととほぼ同じ 222 MB/s を示しており，最大スループットでは RI2N/DRV との差異はほとんどない．

また，スループットについては RI2N/DRV がユーザ透過に利用できることを確認するため OpenMPI [11] を利用した場合の測定も行った．**図 14** に結果を示す．測定の内容は図 12 と同様に ping-pong 通信で，メッセージのやりとりには MPI_Send(),MPI_Recv() を用いた．割込み遅延機能も図 12 と同じく有効にした状態で測定した．図に示すとおり MPI 上からでも RI2N/DRV が利用できシングルリンクの場合に比べてスループットが増加していることが確認できる．

6.3　故障時の通信状況

耐故障機能を評価するため，片方向バースト転送中に 2 リンクのうち 1 つを意図的に故障/回復（人手によってケーブルを抜く/差す）させた場合のスループットの変化を測定した．NIC の割込遅延機能は有効にした状態で測定した．時刻およそ 10 秒でリンクを故障させ，時刻およそ 20 秒で復帰させたところ，**図 15** のようなスループットの変化が現れた．スループットは TCP によるバースト転送を行っている間に，0.1 秒ごとのタイマによりアプリケーション上で取得したものである．

時刻 10 秒で故障を発生させてから 2 秒程度でそれが検出されスループットが回復している．その後は 1 リンクでの最大スループットを維持しており，片方のリンクが故障している間も冗長リンクを使って通信を継続していることが確認できる．

また，時刻 20 秒でハードウェアを回復させた後も 2 秒程度で自動的にそれを検出し，スループットが 2 リンクを使った元のレベルに戻っている．

7. 考　　察

7.1　通 信 性 能

割込み遅延機能を無効にしたレイテンシの測定結果（図 11）から，MTU 以下の短いメッセージで RI2N/DRV の通信にかかる時間が他の方式よりも 5 μ 秒ほど長いことが分かった．しかし，2 パケット以上を用いた通信ではシングルリンクと同等以上の性能であり，また，4 パケット以上では Linux Channel Bonding よりも短い時間で通信を終えている．これに対して，割込み遅延機能を有効にした場合（図 10）には 4 KB 以下の領域で測定結果にばら

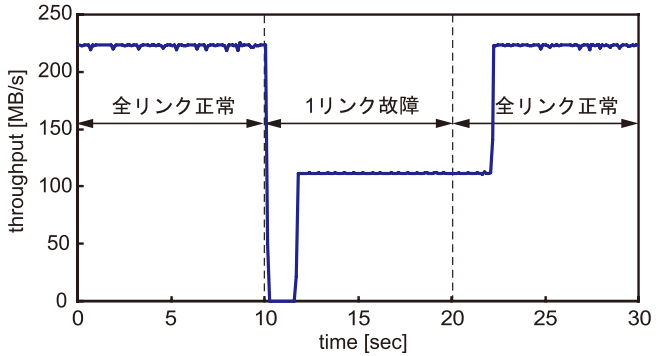

図 15　故障時のスループットの変化
Fig. 15　Throughput before and after failure.

つきがあり比較が難しいが，Linux Channel Bonding が比較的良い性能を示している．しかし，4 KB 以上では方式間の違いが顕著になり RI2N/DRV の性能が 3 方式の中で最も高くなる．

図 10 の測定結果に関しては，RI2N/DRV よりも Linux Channel Bonding の方が興味深い変化の仕方をしている．メッセージサイズが 4.5 KB を超えてからは Linux Channel Bonding の性能はおおむねシングルリンクよりも悪くなっているが，6 KB，7.5 KB，9 KB のパケット数が増加するメッセージサイズの直前でのみ RI2N/DRV を超える非常に高い性能を示している．このように性能が高くなっているのは各パケットの大きさがすべて揃ったためであると考えられる．図 10 からも分かるように 1 つのパケットを処理するために要する時間はそのパケットの大きさに依存している．そのため，大きなパケットの直後に小さなパケットを送った場合，後から送ったパケットの方が先に届く確率が非常に高くなる．また，この評価ではバッファによる順序入れ替わりはほとんど起こらないため，メッセージを構成するすべてのパケットがほぼ同じ大きさになった場合，順序入れ替わりがほとんど起こらなくなる．これにより，特定のメッセージサイズでのみ著しく性能が改善されたものと考えられる．

スループットの測定も割込み遅延機能を有効にした場合と無効にした場合の両方で評価を行った．割込み遅延機能を無効にした場合は，基本的なレイテンシが短くなるためメッセージサイズに対するスループットの立ち上がりが早い．いずれの方式でも 100 KB 程度で最大値までスループットが上昇し，それ以上のメッセージサイズではスループットの変化は少ない．しかし，GbE リンクを 2 つ用いた RI2N/DRV と Linux Channel Bonding でも，最大スループットが 185 MB/s ほどでシングルリンクの 1.6 倍程度にとどまっている．これは，割込み遅延機能を無効にしたことによる多量のハードウェア割込みにより，システムがこれ以上の速度でパケットを処理できないためであると考えられる．これに対して，同様の実験を割込み遅延機能を有効にした状態で行った図 12 では，最大で 222 MB/s までスループットが向上した．これは，シングルリンクのスループットのほぼ 2 倍である．また，RI2N/DRV と Linux Channel Bonding を比較した場合，割込み遅延機能が無効の場合には優位な差は見られないが，機能を有効にした場合にはスループットの立ち上がりの早さに大きな差がある．この結果から RI2N/DRV の順序制御機構がメッセージサイズによらず安定した通信を実現することに大きな効果があることが確認できる．

レイテンシの測定では，割込み遅延機能がない場合の方が高い性能を示したが，スループットの測定では，割込遅延機能がない場合の方が最大スループットが高くなった．この結果からも分かるとおり，割込み遅延機能を利用すべきかどうかは，どのような通信に最適化するかに依存している．本研究では，大規模なデータを扱う科学技術計算やストレージシステムのようにスループットを志向するアプリケーションを想定しており，その場合は割込み遅延機能を有効にした方が性能が良いことが分かる．

今回は MPI ライブラリを利用した通信実験も行った．MPI ライブラリを利用した場合もシングルリンクを利用する場合とほとんど同じ手順で通信をすることができた．図 14 の結果では全体的にメッセージサイズ対するスループットの立ち上がりが遅くなっているものの，図 12 に非常に似た傾向を示している．この評価も割込み遅延機能を有効にした状態で行っているため，RI2N/DRV の順序制御機構の効果が大きい．

7.2　耐故障性

ネットワークの故障には様々な種類のものがある．故障箇所で分類すると，

- ケーブルの破損
- スイッチの故障
- NIC の故障

などがあげられる．また，障害の種類で分類すると，

(1)　パケットの破損
(2)　スループットの低下（一部のパケットが消失）
(3)　すべてのパケットが消失

などがあげられる．RI2N/DRV では上記のいずれの故障箇所に対しても対応することがで

きる．しかし，障害の種類については，すべてのパケットが消失する場合でなければ完全には対応することはできない．

今回の評価ではケーブルの故障を想定し，ケーブルが抜けてそのリンクがまったく使えなくなり，そのリンクのすべてのパケットが消失することを想定した実験を行った．図 15 では，故障発生時に一時的にスループットが大きく低下するものの，約 2 秒という短い時間でその状態から抜け出し，残った 1 つのリンクで効率的に通信を行うことができている．また，リンクを元に戻して故障から回復したときも，2 秒間隔のハートビートパケットが届いたため，時刻 22 秒付近で通信速度がほぼ 2 倍に増加している．

RI2N/DRV ではパケットの内容に対する誤り補正機能を追加してはいない．そのため，パケットの一部が破損するような障害に対しての強度は Ethernet と同等となる．このような障害に対応するためには RI2N/DRV 内で新たに checksum を行う必要がある．checksum によって誤りが検出された場合，そのパケットはロスした場合と同等に扱うことで RI2N/DRV でも対応可能となる．しかし，checksum の計算コストは RI2N/DRV の性能を大きく低下させる可能性があり，この機能の導入には慎重な検討が必要である．

一部のパケットが消失しスループットが低下するような場合は，FT_HTHRESH と FT_LTHRESH を適切に設定しておくことでこれを故障として検出することは可能である．しかし，(2) の状態ではハートビートパケットが正常に送受信できる場合があり，スループットが偏った状態でも回復したと判断する可能性がある．この場合，故障と回復を順番に繰り返し，状態が安定しない．この問題を解決するには，自動回復した場合にも故障があったことをユーザに通知することが重要である．RI2N/DRV の状態を提示することでユーザによって故障の判断がなされる．そのための補助機能として，ユーザレベルで動作するデーモンプログラムの導入が考えられる．イベントが発生するごとに，RI2N/DRV からそのデーモンに対して細かな故障，回復情報を送信する．そしてデーモンが中期的，長期的な情報をもとに実際に故障しているかどうかを判断してそれをユーザに通知する．

7.3　ネットワーク構成の制限

RI2N/DRV では 4.2 節で述べたとおり，リンクごとにスイッチを分散させたネットワーク構成を想定している．これはクラスタネットワークの耐故障性を向上させるために重要な部分である．そのため，図 4 のようにスイッチを分散させた構成が可能となるように実装を行った．しかし，現在の実装では，逆に図 3 のように 1 つの RI2N デバイスで使用している複数の物理デバイスを 1 つのスイッチに接続するような構成では正常に動作しない．RI2N/DRV では 1 つの RI2N デバイスに 1 つの MAC アドレスを使用する．物理デバイスが複数あるにもかかわらず 1 つの MAC アドレスしか使用しないため，スイッチの MAC アドレス学習機能により同時には 1 つのデバイスしか有効に動作しない．現在の実装のままでこのような構成を可能にするためには，VLAN など論理的にスイッチを分割する技術を利用する必要がある．

耐故障性のためには複数スイッチにリンクを分散させた構成が有利であるが，RI2N/DRV の適用範囲をより広範なものにするためには自由なネットワーク構成をとれることが望ましい．そこで，RI2N/DRV の改良によってこの問題に対処する方法を検討する．

1 つはあらかじめ設定ファイルなどによって，通信を行うノードの RI2N デバイスの MAC アドレスと物理デバイスの MAC アドレスの対応表を RI2N/DRV に入力する方法である．パケット送受信時にこの表をもとにパケットの送信元，送信先 MAC アドレスを変更する．送信時には RI2N デバイスの MAC アドレスから実デバイスの MAC アドレスに変更し，受信側では逆に RI2N デバイスの MAC アドレスに復元して IP 層に渡す．この方法は静的な事前の設定が必要なので汎用化には適していない．

もう 1 つの方法は，前者のように設定ファイルは用いず，RI2N ヘッダの拡張によってこの問題に対応する．現在の RI2N/DRV では Ethernet ヘッダのプロトコル番号部分の書き換えを行っており，本来の値を RI2N ヘッダに格納している．これと同じ方法で RI2N ヘッダに RI2N デバイスの MAC アドレスを格納しておき，受信側でこの MAC アドレスを実際の Ethernet ヘッダにコピーする．この方法でも，実ネットワーク上では物理デバイスの MAC アドレスを利用し，RI2N/DRV よりも上位の層では RI2N デバイスの MAC アドレスを使用しているかのように仮想化する．先の方法との根本的な違いは，送信元の物理デバイスの MAC アドレスから RI2N デバイスの MAC アドレスをどうやって求めるかである．この方法では，それを RI2N ヘッダから取得するため，設定ファイルを必要とせず動的にノードが参加する場合にも適用可能である．このように，この方法は RI2N/DRV の適用範囲を広げるにあたって利点が多く，導入を検討している．

8. おわりに

ドライバレベルでパケットの順序入れ替わりを修正することによって高い性能を実現するマルチリンク Ethernet 結合システム RI2N/DRV を提案，実装した．

RI2N/DRV は既存の TCP/IP アプリケーションからユーザ透過に利用することができ，TCP を用いた OpenMPI での通信でも何の変更も加えずに利用することができた．性能評価においては，2 リンクの GbE を用いて最大 222 MB/s のスループットが得られ，また，片

方のリンクが故障した状態でも通信を継続する耐故障性を確認した．レイテンシについては他方式よりもわずかに劣る部分があったが実装の改良により削減できる可能性がある．また，4 KB という比較的短いメッセージからシングルリンクや既存の Linux Channel Bonding よりも性能が高くなっており，HPC 向けクラスタにおける MPI での利用など，ある程度長いメッセージでの利用が多いアプリケーションでは，特に高い性能を示すものと考えられる．特に Linux Channel Bonding と比較した場合，メッセージサイズによるスループットの立ち上がりが著しく改善されている．

今後は，実装方法を改良しレイテンシを削減する．そして通信の衝突が頻繁に発生するような環境での評価を行う．また，MPI やクラスタに限らずサーバとストレージとの接続などユーザ透過に利用できることを生かして，他分野での利用についても検討する．

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）研究領域「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」，研究課題「省電力高信頼組込み並列プラットフォーム」による．

参 考 文 献

1) 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*, pp.342–351 (2003).
2) 岡本高幸，三浦信一，朴　泰祐，佐藤三久，高橋大介：Ethernet マルチリンクによる PC クラスタ向け高バンド幅・耐故障ネットワーク RI2N/UDP，情報処理学会論文誌：コンピューティングシステム，Vol.48, No.8, pp.153–164 (2007).
3) Okamoto, T., Miura, S., Boku, T., Sato, M. and Takahashi, D.: RI2N/UDP: High bandwidth and fault-tolerant network for a PC-cluster based on multi-link Ethernet, *The Workshop on Communication Architecture for Clusters with IPDPS2007*, pp.1–8 (2007).
4) Davis, T.: Linux Ethernet Bonding Driver. http://sourceforge.net/projects/bonding
5) IEEE: IEEE 802.3ad “Link Aggregation” (2000). http://www.ieee802.org/3/ad/index.html
6) Sumimoto, S. and Kumon, K.: PM/Ehernet-kRMA: A High Performance Remote Memory Access Facility Using Multiple Gigabit Ethernet Cards, *CCGrid 2003*, pp.326–333 (2003).
7) Sumimoto, S.: A Scalable Communication Layer for Multi-Dimensional Hyper Crossbar Network Using Multiple Gigabit Ethernet, *International Conference on Supercomputing'06* (2006).
8) Mathis, M., Mahdavi, J., Floyd, S. and Romanow, A.: TCP Selective Acknowledgement Options, RFC 2018 (Proposed Standard) (1996).
9) Floyd, S., Mahdavi, J., Mathis, M. and Podolsky, M.: An Extension to the Selective Acknowledgement (SACK) Option for TCP, RFC 2883 (Proposed Standard) (2000).
10) 高橋浩和，小田逸郎，山幡為佐久：Linux カーネル 2.6 解読室，ソフトバンククリエイティブ (2006).
11) Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L. and Woodall, T.S.: Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation, *PVM/MPI*, pp.97–104 (2004).

(平成 19 年 10 月 9 日受付)
(平成 20 年 1 月 30 日採録)

岡本　高幸（正会員）
昭和 58 年生．平成 18 年筑波大学第三学群情報学類卒業．平成 20 年筑波大学大学院システム情報工学研究科博士前期課程修了．同年富士通（株）入社．クラスタおよび分散コンピューティングに興味を持つ．

三浦　信一（正会員）
昭和 54 年生．平成 14 年千歳科学技術大学光科学部光応用システム学科卒業．平成 20 年筑波大学大学院システム情報工学研究科コンピュータサイエンス専攻博士課程修了．博士（工学）．現在，筑波大学計算科学研究センター研究員．クラスタコンピューティング，ネットワークに関する研究に従事．

朴　　泰祐（正会員）

昭和 35 年生．昭和 59 年慶應義塾大学工学部電気工学科卒業．平成 2 年慶應義塾大学大学院理工学研究科電気工学専攻後期博士課程修了．工学博士．昭和 63 年慶應義塾大学理工学部物理学科助手．平成 4 年筑波大学電子・情報工学系講師，平成 7 年同助教授，平成 16 年筑波大学大学院システム情報工学系助教授，平成 17 年同教授，現在に至る．超並列計算機アーキテクチャ，ハイパフォーマンスコンピューティング，クラスタコンピューティング，グリッドに関する研究に従事．平成 14 年度および平成 15 年度情報処理学会論文賞受賞．IEEECS 会員．

塙　　敏博（正会員）

平成 10 年慶應義塾大学大学院理工学研究科計算機科学専攻博士課程修了．博士（工学）．東京工科大学コンピュータサイエンス学部講師を経て，現在，筑波大学計算科学研究センター研究員．計算機アーキテクチャ，並列処理に関する研究に従事．

佐藤　三久（正会員）

昭和 34 年生．昭和 57 年東京大学理学部情報科学科卒業．昭和 61 年東京大学大学院理学系研究科博士課程中退．同年新技術事業団後藤磁束量子情報プロジェクトに参加．平成 3 年通産省電子技術総合研究所入所．平成 8 年新情報処理開発機構並列分散システムパフォーマンス研究室室長．平成 13 年より，筑波大学システム情報工学研究科教授．平成 19 年より，筑波大学計算科学研究センターセンター長．理学博士．並列処理アーキテクチャ，言語およびコンパイラ，計算機性能評価技術，グリッドコンピューティング等の研究に従事．IEEE，日本応用数理学会各会員．