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

2007－ＱＡＩ－23（２）
2007／5／29

トランスポート端点のポータブルなクラスタ方式

狩野　秀一†　地引　昌弘†

† NEC システムプラットフォーム研究所
神奈川県川崎市中原区下沼部 1753
E-mail: {karino@da, jibiki@bx}.jp.nec.com

あらまし　TCP,SCTP などの端点のポータブルな冗長化方式を提案する．独立した複数台の装置によるクラスタによりサーバ等を冗長化する場合，コネクション指向トランスポートプロトコルの端点は複雑な状態をもつことから，状態コピー等では容易に冗長化することができない．また，トランスポートプロトコル端点はその上位層にアプリケーションがあり冗長化によりアプリケーションの構成等に制約を与えるのは好ましくない．本稿では現用，予備の両系に到達パケットをコピーして入力し，独立にプロトコル処理を行うとともに，Socket API にてアプリケーションの読み書き動作の同期をとる方式を提案する．本方式によりプロトコル状態を複製するオーバーヘッドを減らし，アプリケーションからトランスポート層の冗長化を隠蔽できる．本発表では上記方式の概要および試作ソフトウエアによる評価結果を報告する．

キーワード　transport protocol, redundant system

A Portable Clustering Method for Transport Protocol Endpoints

Shuichi KARINO† and Masahiro JIBIKI†

† System Platforms Research Laboratories, NEC Corporation
1753 Shimonumabe, Nakahara-ku, Kawasaki-shi, 211-8666, Japan
E-mail: {karino@da, jibiki@bx}.jp.nec.com

Abstract　We propose a portable method that endpoints of transport protocols such as TCP, SCTP make be redundant. In construction of redndant servers with clusters of independent nodes, there is a problem that replication of protocol states is difficult because endpoints of connection oriented protocols have complex states. In addition, restricting structure of applications that use those redundant protocol stacks should be avoided. In this paper we propose a method that packets are duplicated and independently processed by each node of clusters and synchronize read/write processing of each node at socket API. Using this method difficulty to replicate states of transport protocols could be reduced and replication mechanism of protocol endpoints dont restrict application's structure. We present the proposing method and evaluation result using a sample implementation.

1. まえがき

TCP/IP 網がインフラとして広く普及するにつれて，多数のユーザー等をかかえる大規模なサービス（たとえばキャリアの VoIP サービスなど）が増えてきた．それらのサービスの基幹を担うノード（たとえば大規模な SIP プロキシサーバなど）は，多くのユーザーが同時に接続するため故障等によるサービス中断の影響が大きい．

これを解決するには，2 台以上のノードを用いる現用・予備構成などの冗長化が一般的だが，多くの TCP/IP 網上のサーバ類は，部分的な冗長化しか実現できていない．とくに，TCP 等のトランスポート（以下 L4 と記す）プロトコル端点の冗長化は，アプリケーションやネットワーク層以下の冗長化と比べてまだ一般的に行われているとは言えない．

L4 プロトコル端点は，アプリケーションに比べてその状態が高頻度で更新されることから，状態の冗長化が難しい．また，プロトコルスタックは通常 OS のカーネル内に実装されていることから，冗長化のための改造もユーザープロセスと比べて高コストである．これらのために，アプリケーションと比べて冗長化が遅れていると考えられる．

本稿では，これらの課題を解決する，次のような特徴をもつ L4 プロトコル端点冗長化方式を提案する．

- AP-プロトコル間の冗長機構分離：Socket APIにてプロトコルスタック冗長化機構をアプリケーションへ隠蔽することによりアプリケーションとプロトコルスタックを分離して冗長化可能とし，プロトコル，アプリケーション双方の冗長化方式への制約を減らす．
- 二重処理状態複製による同期コストの削減：現用系，予備系2台の装置で同報トラフィックを並列処理することで，軽微なプロトコルスタックへの変更のみで，状態コピーを伴わずににプロトコル状態の複製を保持する．

また，提案方式を実装してその性能を評価した結果も報告する．

本稿の構成は次の通りである．2.では，L4プロトコル端点の冗長化方式が満たすべき要件を検討する．3., 4.では，各々提案方式の方式と実装について説明する．5.では，提案方式の性能を評価して実用性を検証する．

2. 課題と要件

本節では，L4プロトコル端点冗長化における要件を検討する．主に下記3点の必要性について順に検討する．

- アプリケーションと冗長機構が分離できること
- できる限りプロトコル依存が少ないこと
- アプリケーション及びプロトコル実装への変更が軽微なこと

2.1 AP-プロトコル間の方式分離

L4プロトコル端点の上ではほとんどの場合アプリケーションが動作するため，L4プロトコル端点の冗長化は，アプリケーションも含めて考える必要がある．ただし，アプリケーション（及びアプリケーションプロトコル）と，下位層のプロトコルとでは，プロトコル状態の更新頻度が大きく異なる．また，下位層のプロトコルは高々数種類だが，アプリケーションは種類が多く多様なため，それらをまとめて単一の方式で冗長化するのは難しい．

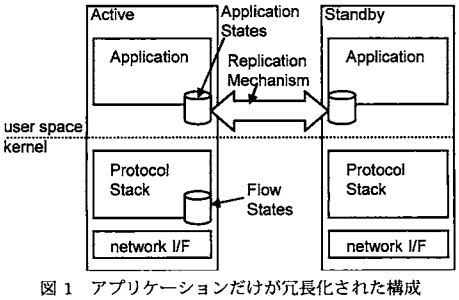

図1 アプリケーションだけが冗長化された構成

L4端点を含み，アプリケーションが動作するノードには，しばしば，UNIX等のマルチプロセスOSを用いられている．それらの環境では，一般にアプリケーションの方が，下位層プロトコルよりも容易に変更／拡張できるため，アプリケーションのみが先に冗長化されている場合もある（図1）(注1)．

(注1)：フェイルオーバ時は，アプリケーションの状態は引き継がれるが，トランスポートプロトコルのコネクションは張り直しになる

これらのことから，少なくともアプリケーションと下位層のプロトコル処理は，独立に冗長化できるべきであり，また，下位層プロトコルの冗長化は，既成アプリケーションを無変更で実現できる方式が望ましいといえる．

2.2 プロトコル独立な冗長化機構

トランスポートプロトコルはプロトコルとしては種類は少ないが，たとえばTCPは継続的かつ頻繁に更新・拡張されている[3]．また近年策定されたSCTP[4]やDCCP[5]は，今後仕様の微調整がしばらく続くことが予想される．これらの仕様変更の都度冗長化方式の再検討やが必要になるのは望ましくない．すなわち，冗長化方式はできる限りプロトコル非依存でるべきでる．

2.3 実装上のポータビリティ

典型的ホストノードでは，アプリケーションより下位層のプロトコルは，OSのカーネル内に実装されている．カーネルの開発はユーザー空間のそれよりも一般に手間がかかり，また，カーネル自体も複雑であるため，L4プロトコル端点の大規模な変更は高コストになる．よって，なるべく既存の実装に手を加えずに冗長化が実現できる方式が望ましい．

以上の条件をふまえて，ネットワークサブシステムのどの部位に冗長機構を組み込むべきかを検討すると，次のようになる．プロトコルに依存した処理が比較的少ないのは，API部分（多くのシステムではバークレイソケット）と，データリンク-ネットワーク層間のネットワークインタフェース層である．また，アプリケーションに独立な，下位層プロトコル冗長化機構を設けるには，API以下でそれを実現する必要がある．

また，プロトコル実装への変更が軽微な冗長化方式として，筆者らは以前ルータ向けに二重パケット処理冗長方式を提案した[1]．同方式では，データリンク層にてパケットを複製して，現用・予備の両系に同報し，独立にプロトコル処理を行わせ，送出前に予備系のみでパケットを破棄する（図2）．これをホストノード向けに適用できれば，比較的プロトコル依存が少ない冗長化方式を実現できると考えられる．

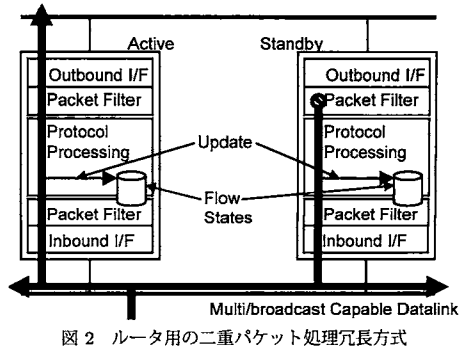

図2 ルータ用の二重パケット処理冗長方式

3. 冗長化方式

前節で検討した要件にもとづいて，本節では，二重処理に基づくL4プロトコル端点の冗長化方式を提案する．APIとして

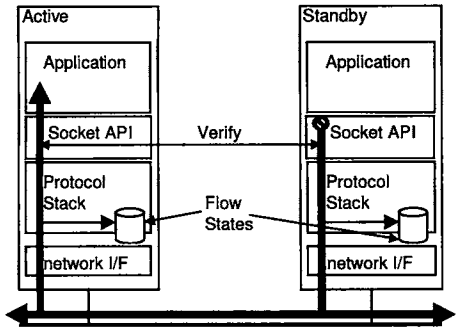

図 3　提案方式での受信トラフィックの流れ

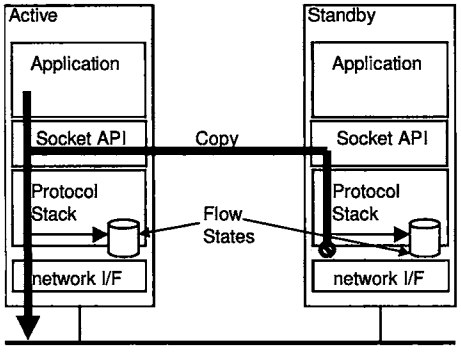

図 4　提案方式での送出トラフィックの流れ

バークレイソケットが，トラフィックを同報可能なデータリンクが各々利用可能であるとする．

下記の方針に基づいて Socket API とネットワークインタフェース層で冗長化処理を実現する．

（1） Inbound のトラフィックは，データリンクにおいて複製し，現用系，予備系の両ノードにて並列に受信処理を行う．その後，Socket API での読み込み処理時に，両ノードで同一データが到達していることを確認したうえで，アプリケーションへデータを渡す（図 3）．

（2） Outbound のトラフィックは，アプリケーションから受け渡されたデータを Socket API での書き込み処理時に複製して現用系，予備系の両ノードで独立にプロトコルスタックへ渡し，送信処理を実行する．重複したパケットの送出を防ぐために，予備系ノードでのみ伝送路へ出る前にパケットを破棄する（図 4）．

（3） 上記の二重処理でプロトコル状態を正しく複製できない場合には，個別に状態同期処理を行う．

以下，Socket API およびネットワークインタフェース層における処理の流れと，3 項の例外処理の概要を説明する．

3.1 Socket API における冗長化処理

Socket API に対しては，API は変更せずに，システムコール毎に，プロトコルスタックが現用系と予備系で冗長に動作するよう拡張を施す．ここでは主要な Socket API システムコールについて処理方法を説明する．

本稿では，アプリケーションは現用系でのみ動作しているものとして，現用系でのみアプリケーションからシステムコールが発行されるものとした．すなわち予備系では，アプリケーションは存在しない（系切り替え後に稼働する）か，または待機状態であるとする．

典型的には，Socket API は次のように用いられる [2]．

- TCP 等では，コネクション待ち受け（サーバ）側は socket() にてファイル記述子を取得し，次に bind(), listen(), accept() を実行して，確立したセッションにたいするファイル記述子を取得する．以降，同記述子を使って読み書き動作を行い，close() 等を呼び出してセッションを終える．発信側は socket() で得た記述子で connect() を実行してセッションを確立させ，以降その記述子を使って読み書きを行う．
- UDP 等では，セッション確立の動作がなく，待ち受け側で bind() を実行して端点を固定すれば，すぐに読み書きが可能になる．

以降この動作に沿って冗長化処理を説明する．

3.1.1 待ち受け側でのセッション開設

待ち受け側での socket() 処理から accept() 処理までの流れを図 5 に示す．

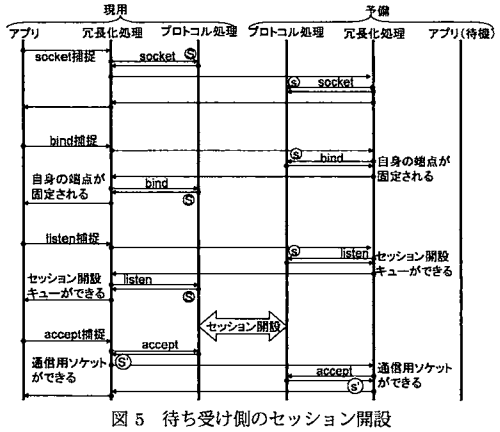

図 5　待ち受け側のセッション開設

まずアプリケーションが socket() をコールすると，現用系に加えて予備系でも socket システムコールを実行する．その結果得られる記述子の番号は，一般に現用系と予備系では異なっている（図では各々S と s で示した）．以降の冗長処理で，操作対象のセッションを識別するため，ここで得られた現用系と予備系の記述子の対応関係を保持しておく．

bind, listen システムコールは，現用系のアプリケーションが呼び出しをおこなうと，まず予備系でシステムコールを実行する．これは，外部に対してトラフィックが送出されない予備系で先に端点を作成することでシステムコールの実行時差の影響をなくすためである．

accept システムコールについては現用系で先にシステムコールを実行し，同様に予備系でも実行して，socket の場合と同様新しく得られた記述子（図では S' と s'）を保持しておく．

以上で待ち受け処理の冗長化が行える．

3.2 発信側でのセッション開設

socket システムコールについては，サーバの項で説明したのと手順は同様である．connect システムコールのの冗長化は，次のようになる．

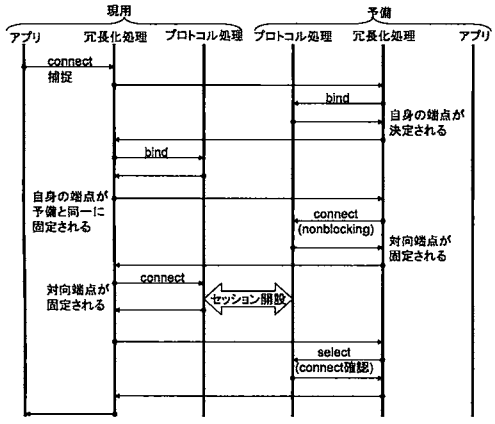

図 6　クライアント側のセッション開設

図 6 に，クライアント側の処理シーケンスを示す．アプリケーションが connect システムコールの呼び出しをすると，まず現用，予備両方の系で自ノード側の端点を固定する処理をおこなう．これは，並列処理のためには，現用系と予備系で同じポート番号の割り当てを要するためである．端点割り当ては，現用系と予備系とで bind() を使って行うことでプロトコル独立な処理として実現できる．

続いて connect システムコールを実行して対向端点にたいしてセッションを確立する．bind(), listen() の場合と同様，システムコールの実行時差の影響をなくすために，まず予備系側で，システムコールを実行する．ただし，予備系では実際にはパケットが送出されないために，セッション確立処理が途中で止まる場合がある（たとえば TCP の SYNACK 待ちなど）．よって，connect() は非同期モードで実行しておき，現用系での connect システムコールの実行に移る．

次に，現用系側で connect システムコールを実行して，セッション確立処理を実際におこなう．現用系でセッション確立処理をおこなうと，実際にパケットが送出され，セッションが確立される．このとき，予備系が対向端点からの応答待ちの状態であれば，対向端点からの応答（現用系と予備系に同報される）によりセッション確立処理が進行する．

3.2.1 読み込み系の動作

図 7 に，受信と読み込み処理のシーケンスを示す．図では read() のみを例にしているが，recvfrom() など他のシステムコールでも同様である．

図 3 に示したように，現用系と予備系のプロトコルスタックは並列に入力トラフィックを処理する．Socket API への変更を軽微にするため，現用と予備の両系で read システムコールを実行して，プロトコルが処理した受信データを読み込む．

アプリケーションへは現用系の Socket API で読み込んだデータを渡せば良いため，予備系では，受信データは不要であ

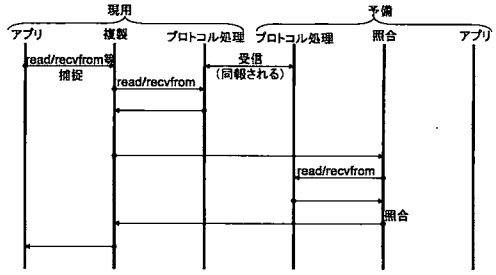

図 7　受信-読み込み

るが，同一データを処理していることを確認するため，読み込み処理の後，現用系と予備系の読み込みデータを照合する．

3.2.2 書き込み系の動作

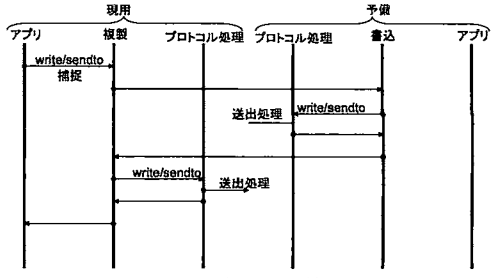

図 8　書き込み-送信

図 8 に，書き込みと送信処理のシーケンスを示す．図では write() のみを例にしているが，他のシステムコールでも同様である．

本システムでは，アプリケーションが現用系のみで動作するため，図 4 に示したように，書き込みは現用系のみで行われる．アプリケーションが write() を呼び出すと，まず，予備系で，次に現用系でシステムコールを実行し，各々の系でプロトコルに送信処理を実行させる．予備系で先にシステムコールを実行するのは，bind(), listen() 等の場合と同様，システムコールの実行時差の影響を低減するためである．

3.3 データリンク及びネットワークインタフェース層における冗長化処理

図 3 に示したように，データリンク層では，外部から到達したパケットを同報して現用と予備の両系に受信させる．同報処理は，たとえば Ethernet のマルチキャスト等が利用できる．その場合，具体的には現用系，予備系で共有する IP アドレス A を用意し，A への ARP 要求にマルチキャスト MAC アドレス m を格納して応答させる．現用，予備両系で m を受信するよう設定しておけば，同一パケットを受信できる．

また，図 4 に示したように，予備系においては，送出用のパケットを，伝送路へ送出する前に破棄する処理を行う．全送出パケットを破棄すると管理用通信などに支障が出るため，先述の代表アドレス A を発信元とするパケットのみ破棄する．多くの OS では，ネットワークインタフェース層付近にパケットフィルタが用意されているため，同フィルタを用いてこの処理を実装できる．

3.4 二重処理以外の状態冗長化

これまでに説明した二重処理方式の枠組みでは，一貫した冗長処理が実現できない以下のような場合がある．これらは，プロトコル等に依存する問題のため，個別に対処することにした．

3.4.1 乱数等由来のプロトコルデータ

TCP の初期シーケンス番号等，乱数や装置内の時刻，タイマー等により決定され，送出パケット等に付加される情報は，現用系，予備系で独立に生成すると，一般に異なる値になる．

このようなプロトコルデータは，現用系と予備系で同じ値になるよう，パケット送出時にパケットを加工するか，または，予め同じ値に揃えておく必要がある．提案方式では，実装の容易さを考えて，現用系で設定された値を取得して予備系に伝達し，予備系でも同一の値を設定するようにした．

3.4.2 著しい同期解離の防止処理

現用系と予備系の間でプロトコル処理に著しい進捗の差異が出ると，冗長処理が継続できなくなる場合がある．たとえばTCP では，予備系がパケットの送出動作を行う前に同一パケットを現用系が送出し，その確認応答が届いてしまうと，予備系がそのあと該当パケットを送出しても多くの場合確認応答は届かないため，それ以降予備系の送出処理が継続できなくなる．

なるべくプロトコル実装に変更を加えず，このような状態を回避するため，先述のネットワークインタフェース層付近のパケットフィルタにて，所定のパケットを保持させるか，通過を遅延させる処理を行う．上記の例では，予備系がパケットを送出するまで，該当応答を保持しておき，必要に応じて該当パケットを予備系へ再送するようにした．

3.5 系切り替え

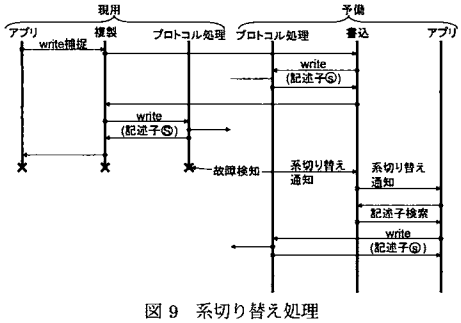

図 9 系切り替え処理

図 9 に処理シーケンスの例を示す．一般に，系切り替えは故障検知などを契機として実行される．本稿では，故障検知機構については言及しないが，死活監視プロトコルやソフトウエアは多数あり（たとえば [10]），それらを利用して実現できる．

系切り替え後は，予備系が単体の系として動作するようになれば良い．本方式では，予備系のプロトコルスタックも現用系と同様に稼働している．系切り替え時には，プロトコル状態の引き継ぎは不要で，Socket API とパケットフィルタの冗長処理部分を変更する．具体的には，SocketAPI ではソケット記述子の引き継ぎ，パケットフィルタでは予備動作の解除のみ行えば良い．

a） ソケット記述子の引き継ぎ

通信中セッションのソケット記述子は，予備系の Socket API 冗長処理部分に保持されている．アプリケーションは，同処理部よりオープンされたソケット記述子番号を取得できれば通信を継続できる．オープンされたソケット記述子の番号だけでは，それがどのセッションを指しているのか判断できないが，getpeername() 等を用いて識別できる．

b） パケットフィルタの設定

送出パケットを破棄するよう，予備系で設定されているパケットフィルタは，送出パケットを遮断しない（すなわち何もしない）ように設定を変更する．そのほか，冗長化のために行っていた処理があれば，すべて停止させる．

4. 実　　装

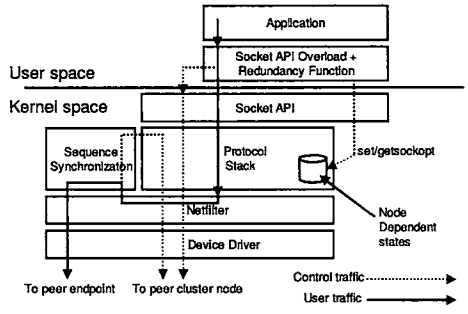

図 10 提案方式の実装

提案方式を Linux 上に実装した．図 10 に構成の概要を示す．冗長処理を実現する部位のうち，Socket API の拡張はライブラリとして，パケットフィルタは netfilter [9] モジュールとして実装した．また故障検知は heartbeat [10] を利用した．

- ソケット API 冗長化部分は，システムコールを捕捉して追加処理をおこなう動的リンクライブラリとして実装した．すなわち，ソケット API に加えた拡張は，同ライブラリ内に実現し，カーネル内の Socket API システムコールのコードへは変更は加えていない．

予備系では，現用系で動作するアプリケーションと対になって，Socket API の処理のみを行うプロセスを用意し，同プロセスに上記ライブラリをリンクするようにした．系切り替え後は，同プロセス上でアプリケーションを稼働させて処理を継続できるようにした．

- パケットフィルタでは，同フィルタでは，予備系における送出パケットの遮断，同期解離防止のためのパケット再送処理等を，TCP および SCTP を対象に実装した．また同期解離処理には現用系，予備系間の通信が必要になるため，ICMP にてその通信機構を実装した．

5. 評　　価

提案方式は，プロトコルおよびアプリケーションへの変更は比較的少ないが，プロトコル処理を現用系と予備系で並列に行い，Socket API とパケットフィルタにて同期処理を行うため，

冗長処理のコストが高く，また，系間の通信遅延による性能低下も懸念される．提案方式がどの程度性能に影響を与えるかを実測により評価した．

5.1 評価環境・方法

上記冗長化の影響を評価するため，前節にて説明した実装を用いて，単体ノードの性能と提案方式にて冗長化したノードの性能を，処理可能なスループットの計測により評価した．スループットの計測は，十分高速な対向ノードとの間で方方向のみトラフィックを能力一杯まで流し，アプリケーションから見た単位時間あたりのデータ転送量を計測することで行った．

図 11 に評価環境の構成を示す．評価に用いた装置の諸元は表

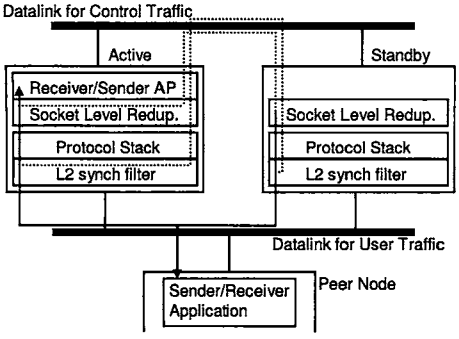

図 11 評価環境の構成

1 の通りである．ネットワークデバイスはすべて Fast Ethernet,

表 1 評価に用いた装置の諸元

項目	現用系	予備系	対向
CPU	Celeron 900MHz	Pentium3 930MHz	Celeron1100MHz
メモリ	256Mbytes	256Mbytes	256Mbytes
OS	CentOS 3.6	CentOS 3.6	CentOS 4.4
kernel	2.4.33	2.4.33	2.6.9

現用系と予備系間の通信用と，対向との接続用に別のデータリンクを用意した．

提案方式では，システムコールの発行の都度，現用系と予備系の間でプロセス間通信が必要になる．このコストを計測するため，システムコールで読み書きする単位データ長を 64〜65536 バイトの範囲で選び，その各場合について上記の性能を計測した．

5.2 評価結果・考察

図 12,13 に各々送信，受信性能の計測結果を示す．

とくにメッセージ長が小さい場合に，TCP において，単体構成と冗長構成の性能差が顕著であることが分かる．TCP ではシステムコールで読み書きするデータの長さによらず，セグメントは MSS 単位で生成できるため，単体動作ではほぼ性能が一定になっており，冗長化によりシステムコール単位でプロセス間通信が必要になったため，性能低下が発生していると考えられる．これらを解決するには，系間の通信頻度を低減させる最適化が必要である．具体的には，同期通信を毎回でなく，可能であれば数回に 1 回に間引く等が考えられる．

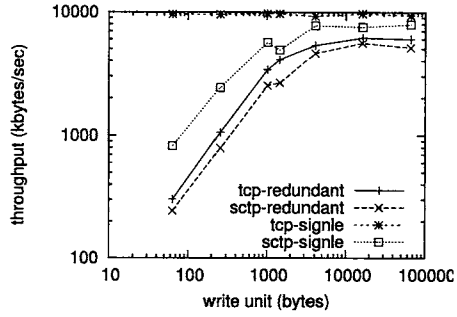

図 12 性能評価結果（送信）

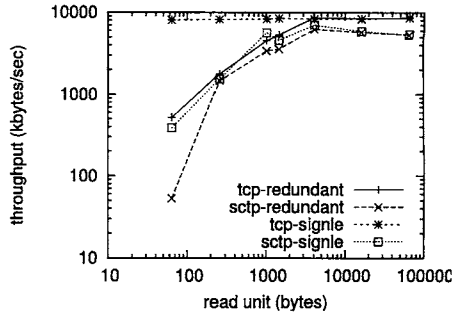

図 13 性能評価結果（受信）

6. むすび

プロトコルスタックにおけるトラフィック二重処理を特徴とする L4 プロトコル端点の冗長化方式を提案した．本方式によれば，Socket API とネットワークインタフェースにて冗長化処理を行うことで，アプリケーション及びプロトコルスタックへの軽微な改造だけで冗長化を実現できた．性能評価を実施した結果，小さいデータを高頻度に送受信する場合に冗長化オーバーヘッドが高いことが分かった．現在上記の場合の性能改善方式について検討を進めている．

文献

[1] 狩野, 地引, “ルータクラスタにおける二重パケット処理冗長方式,” 信学論, Vol.J88-B, No. 10, pp. 1956-1967, Oct, 2005.

[2] W.R. スティーヴンス著, 篠田訳, “UNIX ネットワークプログラミング,” 第 2 版, Vol.1, トッパン, 1999.

[3] M. Duke 他, “A Roadmap for Transmission Control Protocol (TCP) Specification Documents,” RFC4614, Sep 2006.

[4] R. Stewart 他, “Stream Control Transmission Protocol,” RFC2960, October 2000.

[5] E. Kohler 他, “Datagram Congestion Control Protocol,” RFC4340, March 2006.

[6] W.R. スティーヴンス著, 橘訳, “詳解 TCP/IP Vol.1 プロトコル,” ソフトバンク, 1997.

[7] A.S. タンネンバウム, M.V. スティーン著, 水野他訳, “分散システム 原理とパラダイム”, ピアソンエデュケーション, Oct. 2003.

[8] 米田他, “ディペンダブルシステム,” 共立出版, 2005.

[9] netfilter/iptables Project, `http://www.netfilter.org/`.

[10] High-Availability Linux project Web site, `http://www.linux-ha.org/`