2008－HPC－116 (31)
2008／8／7

# 広域計算環境用のスケーラブルな高性能通信ライブラリ

斎藤秀雄[†,††] 田浦健次朗[†]

我々が開発している広域計算環境用の通信ライブラリ SSOCK の進捗について報告する．SSOCK は通常の Socket と類似した API を提供するが，WAN における接続性やスケーラビリティの問題を解決するために LAN 間の通信を透過的に中継デーモンのオーバレイネットワークを介して行う．一つの実験では，11 クラスタからなりファイアウォール及び NAT を含む広域環境で，SSOCK を用いることによって任意のノード間で接続が確立できることを確認した．別の実験では，広域環境で多数のプロセスが同時にコネクトを行ったときに，通常の Socket を用いると多数のパケットが落ちて 189 秒でタイムアウトしてしまうのに対して，SSOCK を用いると 170 ミリ秒で全対全で接続が確立できた．

# A Scalable High-performance Communication Library for Wide-area Environments

HIDEO SAITO [†] and KENJIRO TAURA[†]

We report on the progress of SSOCK, the communication library for wide-area environments that we are developing. SSOCK provides a Socket-like API, but transparently routes inter-LAN traffic through forwarding daemons in order to solve the connectivity and scalability problems of WANs. In one experiment, we confirmed that connections could be established between every pair of nodes in an 11-cluster environment with firewalls and NAT. In another experiment in which many processes simultaneously tried to establish connections, SSOCK was able to establish connections between all pairs of processes in 170 milliseconds, while regular Sockets suffered from a large number of packet drops and timed out after 189 seconds.

## 1. はじめに

近年，Wide Area Network (WAN) の帯域の増加に伴い，異なる位置に存在する複数のクラスタを接続して並列計算を行う機会が増加した．複数のクラスタを用いることによって，単一のクラスタを用いる場合より多くの計算力，メモリ，ストレージが利用可能になる．一方，WAN にはファイアウォールやプライベートアドレスに起因する接続性の問題があり，これらを解決しなければ複数のクラスタを接続することはできない．

これまでに WAN における接続性の問題を解決するための手法がいくつか提案されてきた[1),2),5)]．それらの手法は通常通信できないノードが通信できるようにするが，大規模な並列アプリケーションを支援するためには，さらに以下に説明するスケーラビリティ及び性能に関する要求を満たす必要がある．

**スケーラビリティ** クラスタの数及びノード数が増えてもシステムの様々な資源の制限に触れない必要があり，そのためには多数の接続を確立するような単純な手法を用いてはならない．特に，WAN の接続は資源の消費量が多いので，多数確立する

**表 1** いくつかのファイアウォールの同時セッション数 (ファイアウォールの規模の指標としてスループットも示す)
Table 1 Number of concurrent sessions for some common firewalls (throughput is also given as a measure of the scale of the firewall)

| ファイアウォール | 同時セッション数 | スループット |
|---|---|---|
| WatchGuard Firebox®Edge X55e | 10,000 | 100 Mbps |
| Juniper Networks NetScreen-208 | 128,000 | 375 Mbps |
| ITOS CR1000i | 400,000 | 1 Gbps |
| Fortinet FortiGate-3600 | 1,000,000 | 4 Gbps |

† 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
†† 日本学術振興会特別研究員
Research Fellow of the Japan Society for the Promotion of Science

ことを避けなければならない．例えば，TCP の接続は高いスループットを実現するために帯域遅延積以上のバッファメモリを必要とするが，WAN の接続には帯域・遅延が共に大きくてメモリを大量に要するものがある．また，Network Address Translation (NAT) を行うサーバが処理できるセッションの数はおよそ 65,000 (ポートの数) に限られている．さらに，ステートフルインスプクションを行うファイアウォールが処理できるセッションの数にも限りがある (**表 1** にいくつかのファイアウォールの同時セッション数を示す)．

**性能** WAN で多数の接続を確立するとスケーラビリティのみならず通信性能にも悪影響を与えることがある．例えば，多数の接続が協調せずに同時に通信するとパケットが落ちて通信効率が下がる場合がある．オーバレイネットワークを構築すれば接続数を削減することができるが，ネットワークのトポロジーや性能を考慮して中継を行わないと，やはり通信性能が落ちてしまう．

本稿では以上を踏まえて我々が開発している広域環境用の通信ライブラリ Scalable Sockets (SSOCK) の進捗について報告する．今後 Socket API の主要な部分を実装して WAN を意識していない既存のミドルウェアを WAN で高速に動作させることを考えているが，現在は Socket API そのものではなくそれに類似したものを実装している．

以降，2 章で関連研究を紹介する．その後，3 章で SSOCK の設計と実装について説明して，4 章で実験結果について述べる．最後に，5 章でまとめと今後の課題を述べる．

## 2. 関連研究

ファイアウォールを越えて通信するために広く用いられている技術として SOCKS[4] などで用いられるプロキシサーバがある．プロキシサーバは直接通信できないノードに対するアクセスを代理で行うことによって通信を可能にする．しかし，プロキシサーバを複数経由しなければ通信できない場合もあるので，多数のノードが全対全で通信できるように経路を設定するのはとても煩雑である．また，プロキシサーバを用いて多数のノードを接続した場合，WAN では多数の接続が確立されてしまう．

WAN で接続性のことを気にすることなくソケットを用いることを可能にするライブラリとして SmartSockets[5] がある．SmartSockets は Java の `Socket` クラス及び `ServerSocket` クラスを拡張したものを提供する．それらのクラスは，透過的に逆向きコネクトを行ったり中継デーモンを用いたりすることによって，通常接続を確立できないノードが接続を確立できるようにする．SmartSockets の中継デーモンは自動的にルーティングテーブルを構築するので SOCKS のような煩雑さはないが，やはり WAN で多数の接続が確立されるという問題はある．

多数のノードをスケーラブルに接続するためには，Chord[9] や Pastry[7] を始めとすると分散ハッシュテーブル (Distributed Hash Table: DHT) を用いるとができる．DHT を用いたオーバレイネットワークでは，$n$ 個のノードがそれぞれ $O(\log n)$ 個の他のノードに関する情報を保持することによって，互いに平均 $O(\log n)$ ホップで通信できる．しかし，DHT は P2P アドレス空間において割り当てられたアドレスを用いてルーティングを行うため，メッセージが物理ネットワークにおいては大きく遠回りをしてしまうことが多い．

Ganguly らによって提案された IP over P2P (IPOP)[2] も P2P オーバレイネットワークを用いて接続性の問題を解決する．IPOP が P2P オーバレイネットワークとして用いる Brunet[1] でも Chord や Pastry と同様にメッセージが大きく遠回りをしてしまうことがあるので，Ganguly らは頻繁に通信するノードの間にはショートカット接続を確立することも提案した[3]．しかし，多数のノード対が頻繁に通信するアプリケーションでは多数のショートカット接続が確立されてしまうという問題がある．また，IPOP が IP トンネリングを行うために用いる仮想インタフェースのオーバヘッドは多くの並列アプリケーションにとっては大きすぎる (クラスタ内の遅延は通常数マイクロ秒から数十マイクロ秒であるが，IPOP の仮想インタフェースのオーバヘッドは数ミリ秒である)．

## 3. 設計と実装

### 3.1 概　　要

Scalable Sockets (SSOCK) は，WAN における接続性及びスケーラビリティの問題を解決しつつ高い性能を実現することを目的とした通信ライブラリである．SSOCK は以下の理由から LAN 内の通信は直接行い，LAN 間の通信は中継デーモン (`ssockd`) のオーバレイネットワークを介して行う．

- LAN 内ではすべてのノードが互いに直接通信できることが多い．一方，LAN 間では直接通信できないノードもあるが，中継を行うことによって間接的に通信できるようになる．
- LAN 内で中継を行うと遅延が大きな影響を受け

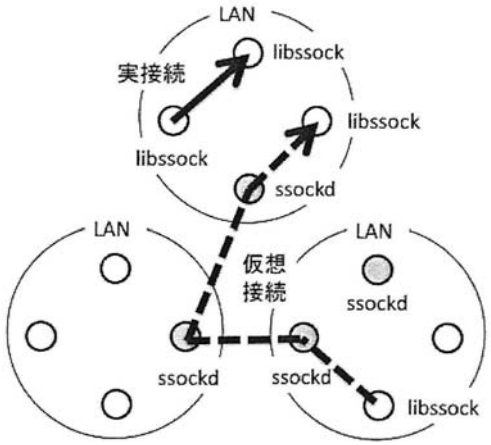


**図 1** 実接続と仮想接続
Fig. 1 Real and virtual connections

てしまう．例えば，ノード間の遅延が一様であると仮定すると，中継オーバヘッドを無視しても，一回中継した場合の遅延は一回も中継しない場合の遅延の二倍になってしまう．

- LAN 間で多数の接続を確立すると多くの資源を消費するが，LAN 間の接続を少数の中継デーモンの間でのみ確立すれば資源の消費量は大幅に削減できる．

通信を直接行うか中継デーモンを介して行うかは，ライブラリ (libssock) がアプリケーションには透過的に決定する．libssock は**図 1** に示すように，LAN 内の接続は実接続として確立し，LAN 間の接続は ssockd によってルーティングされる仮想接続として確立する．この設計は SmartSockets の設計に似ているが，高いスケーラビリティを達成するために LAN 間では直接通信することが可能でもしないという点で異なる．また，各 LAN に複数の中継デーモンを配置できるようになっているという点でも異なる．複数のデーモンを用いた実験はまだ行っていないが，LAN 間の帯域を一個のデーモンで使い切れない場合や負荷分散が必要な場合には，複数のデーモンを用いることができることは重要である．

以降，3.2 節から 3.4 節で libssock と ssockd とそれらのブートストラップに用いられる bootserv について説明する．

### 3.2 Libssock

libssock は SSOCK のライブラリ部分であり，SSOCK を用いるアプリケーションにリンクされる．最終的には Socket API の主要な部分を提供する予定であるが，現在は Socket API そのものではなくそれに類似したものを提供している．具体的には，Socket API の connect や send などと同じ引数と返り値を持つ ss_connect や ss_send などの API を提供している．

初めて libssock の関数が実行されたときに，libssock は自分が接続すべき ssockd を bootserv に問い合わせ，その ssockd に接続する．それ以降，LAN 内のノードに対して ss_connect が呼び出された場合はそのノードに対して connect を呼び出すことによって実接続を確立し，LAN 外へのノードに対して ss_connect が呼び出された場合は ssockd 経由でそのノードにメッセージを送ることによって仮想接続を確立する．ss_send は，与えられたソケットが実接続と仮想接続のどちらのものであるかに応じて，send を呼び出して宛先ノードに直接データを送るか ssockd 経由で宛先ノードにデータを送るかを決定する．ss_accept や ss_recv などの他の API も同様の動作をする．

### 3.3 Ssockd

ssockd は SSOCK のデーモン部分であり，メッセージの中継を行う．ssockd は起動されると最初に bootserv と接続を確立する．それ以降，bootserv を通して他の ssockd のエンドポイントを知り，それらのエンドポイントに対してコネクトを試みることによってオーバレイネットワークを構築する．アプリケーション実行時に自分に接続している libssock からメッセージが送信されてきたら，それを他の ssockd に中継する．また，他の ssockd からメッセージが送信されてきたら，自分に接続している libssock 宛であればその libssock に中継し，そうでない場合は更に別の ssockd に中継する．

現在は各 ssockd は他のすべての ssockd にコネクトを試み，確立できた接続をすべて用いる．この手法でも計算に参加するすべてのプロセスが互いに接続を確立しようとする場合よりは WAN の接続を大幅に削減できるが，ssockd の数が多い場合は ssockd 間の接続さえ削減する必要がある．そこで，今後は 8) で提案されているような手法を用いて選択的に接続を確立して，更にスケーラビリティを高めることも検討したい．

また，ssockd は定期的に互いにリンクステートメッセージを送ることによってルーティングを行うので，ssockd を同時に起動する必要はなく，後から ssockd を追加したり落としたりしてもかまわない．ただし，メッセージの到着確認や到着順序の保証はしていない

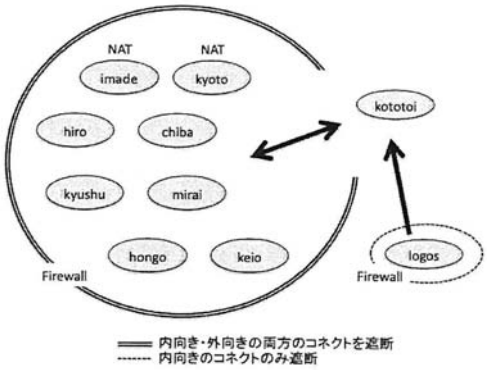


**図 2** 実験環境
Fig. 2 Experimental environment

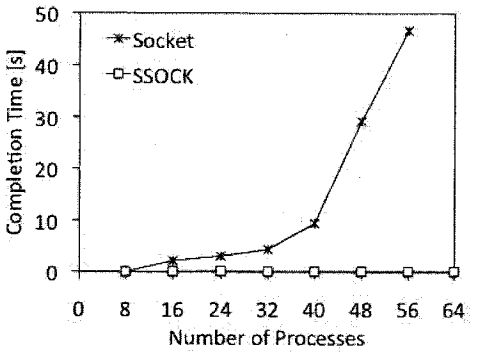


**図 3** 全対全のノンブロッキングコネクト要した時間
Fig. 3 Completion time of all-to-all non-blocking connect

**表 2** 各クラスタのノード数及びコア数
Table 2 The number of nodes and cores in each cluster

| クラスタ名 | ノード数 (コア数) | クラスタ名 | ノード数 (コア数) |
|---|---|---|---|
| chiba | 58 (116) | okubo | 13 (26) |
| hiro | 11 (88) | keio | 10 (80) |
| hongo | 14 (28) | kototoi | 22 (88) |
| imade | 29 (58) | kyoto | 28 (56) |
| logos | 4 (14) | kyushu | 10 (80) |
| mirai | 6 (48) | | |

ので，アプリケーションの実行と実行の間に ssockd を増減させるのはかまわないが，アプリケーションの実行中に ssockd を増減させることはできない．

### 3.4 Bootserv

bootserv は libssock や ssockd のためのランデブポイントである．ssockd に接続されたときは，その ssockd のエンドポイントを教えてもらい，逆にその ssockd に他の ssockd のエンドポイントを教える．libssock に接続されたときは，その libssock に接続すべき ssockd のエンドポイントを教える．bootserv は SSOCK で唯一全ノードから接続できなければならないプロセスであり，bootserv のエンドポイントは SSOCK で唯一設定しなければならない項目である．

## 4. 実験結果

### 4.1 接続性

最初の実験では，ファイアウォール及び NAT を含む広域環境で接続テストを行った．用いた実験環境を**図 2** 及び**表 2** に示す．OS はすべてのノードで Linux 2.6.18 であった．図 2 の二重線は内向き・外向きの両方のコネクトを遮断するファイアウォールを表しており，その中にある 8 個のクラスタは互い及び kototoi とのみ通信できる．また，点線は内向きのコネクトのみ遮断するファイアウォールを表しており，その中にある logos から kototoi にコネクトをすれば接続を確立できるが，kototoi から logos にコネクトを行っても接続は確立できない．また，imade と kyoto は NAT を用いており，各ノードはプライベートアドレスしか持っていない．

各クラスタに ssockd を 1 個立ち上げた状態で接続テストを行ったところ，SSOCK を用いることによって任意のノード間で接続を確立することができた．クラスタ内では実接続が確立されたのに対して，クラスタ間ではいくつかの ssockd を経由する仮想接続が確立された．例えば，chiba-logos 間では 3 個の ssockd (chiba, kototoi, logos) を経由する仮想接続が確立され，imade-kyoto 間では 2 個の ssockd (imade, kyoto) を経由する仮想接続が確立された．

### 4.2 同時接続確立

次の実験では，広域環境で多数のノードが同時にコネクトを行ったときにかかる時間を測定し，通常の Socket と SSOCK とで比較を行った．Socket ではファイアウォールや NAT を越えて接続を確立することはできないので，この実験では図 2 のクラスタのうち直接通信できる 8 個のクラスタ (chiba, hiro, hongo, keio, kototoi, kyushu, mirai, okubo) のみ用いた．各クラスタで同数のプロセスを立ち上げ，各プロセスはノンブロッキングコネクトを用いて他のすべてのノードに接続を確立しようとした．

プロセスの数を 8 から 64 まで変化させたときの結果を**図 3** に示す (56 プロセス以上のときは mirai クラスタでは一部のノードで 2 プロセスずつ立ち上げた)．Socket を用いた場合は 56 プロセスでは 47 秒かかり，64 プロセスでは 189 経過しても一部のコネクト

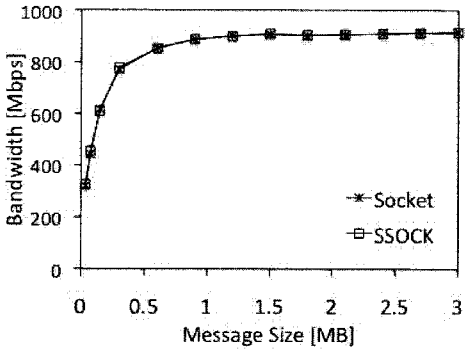


図 4 クラスタ内のピンポン性能
Fig. 4 Intra-cluster ping-pong performance

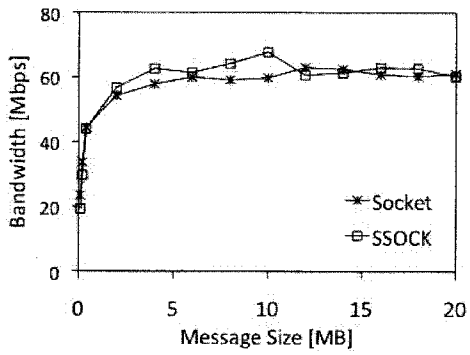


図 5 クラスタ間のピンポン性能
Fig. 5 Inter-cluster ping-pong performance

が完了せずタイムアウトしてしまった☆. これに対して SSOCK を用いた場合は 64 プロセスでも 170 ミリ秒ですべての接続が確立された. SSOCK を用いる場合は事前に ssockd 起動する必要があるが, ssockd の起動時間の 8 秒を考慮しても SSOCK の方が速かった.

Socket を用いた場合に時間がかかったのは, 同時にコネクトを行うとセキュリティのために SYN パケットを落とすルータがあったためである. 多数のプロセスでもコネクトがタイムアウトしないようにするためには, 同時に行うコネクトの数を制御したりブロッキングコネクトを用いたりする必要がある.

これに対して SSOCK を用いた場合は, LAN 間では ssockd 間で事前に確立した接続を通常のメッセージが通っただけなので, パケットが落とされることなかった. SSOCK でも ssockd を同時に立ち上げると ssockd 間で同時にコネクトが行われるが, ssockd の数は末端プロセスの数より少ないという点が異なる.

### 4.3 一対一通信性能

最後の実験では, ピンポンテストを行って一対一の通信性能を通常の Socket と SSOCK とで比較した.

まず, ピンポンプロセスを単一クラスタ (kototoi) 内に立ち上げた場合の結果を**図 4** に示す. Socket と SSOCK の性能はほぼ一致したが, これは SSOCK を用いてもクラスタ内では Socket を用いた場合と同様に実接続が確立されたためである.

次に, ピンポンプロセスを異なるクラスタ (hongo と okubo) に立ち上げた場合の結果を**図 5** に示す. 今度は SSOCK を用いた場合は仮想接続が確立されてメッセージは二つの ssockd によって中継されたが, やはり Socket に対して性能低下は観測されなかった.

☆ Linux の connect は 189 秒でタイムアウトする.

## 5. おわりに

本稿では, 我々が開発している広域計算環境用の通信ライブラリ SSOCK の進捗について報告した. LAN 間の通信を中継デーモンのオーバレイネットワークを介して行うという SSOCK の設計について説明し, 接続性, 同時接続確立, 一対一通信性能の三点について行った実験の結果について述べた.

今後の課題としては, 以下のことが挙げられる.

- Socket API そのものを提供するようにし, WAN を意識せずに設計されている既存のミドルウェア (例えば MPICH2[6]) を WAN で動作させて性能評価を行うこと.
- 多数のノード対が同時に通信したときの性能評価.
- 各 LAN に立ち上げるべき ssockd の数を自動的に決定する方法の考案.

**謝辞**

本研究の一部は文部科学省科学研究費補助金特定領域研究「情報爆発に対応する新 IT 基盤研究プラットフォームの構築」の助成を得て行われた.

## 参考文献

1) Brunet Software Library: Online at http://brunet.ee.ucla.edu/brunet/.
2) Ganguly, A., Agrawal, A., Boykin, P. and Figueiredo, R.: IP over P2P: Enabling Self-configuring Virtual IP Networks for Grid Computing, *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2006).
3) Ganguly, A., Agrawal, A., Boykin, P. and Figueiredo, R.: WOW: Self-Organizing Wide Area Overlay Networks of Virtual Worksta-

tions, *Proceedings of the 15th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pp. 30–41 (2006).
4) Leech, M., Ganis, M., Lee, Y., Kuris, R., Koblas, D. and Jones, L.: SOCKS Protocol Version 5, IETF RFC 1928 (1996).
5) Maassen, J. and Bal, H.: SmartSockets: Solving the Connectivity Problems in Grid Computing, *Proceedings of the 16th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pp. 1–10 (2007).
6) MPICH2: High-performance and Widely Portable MPI: Online at http://www.mcs.anl.gov/research/projects/mpich2/.
7) Rowstron, A. and Druschel, P.: Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems, *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pp. 329–350 (2001).
8) Saito, H. and Taura, K.: Locality-aware Connection Management and Rank Assignment for Wide-area MPI, *Proceedings of the 7th International Symposium on Cluster Computing and the Grid (CCGrid)*, pp. 249–256 (2007).
9) Stoica, I., Morris, R., Karger, D., Kaashoek, M. and Balakrishnan, H.: Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications, *Proceedings of the 2001 Conference on Applications, Technologies, Architectures and Protocols for Computer Communication (SIGCOMM)*, pp. 149–160 (2001).