実時間シグナルを用いたポーリングI/Oライブラリの設計

河 合 栄 治[†] 門 林 雄 基[††] 山 口 英[††]

数千から数万もの同時コネクションを扱うネットワークサーバにおいて，高い性能を得る 1 つの重要なポイントは，I/O の多重処理管理である．現在多くのサーバで，select() や poll() によるポーリング I/O が I/O の多重処理に用いられている．しかし，このポーリング I/O は，同時コネクション数の増加にともなう性能劣化の問題が広く知られている．本研究では，POSIX 実時間シグナルを用いた高速な I/O ポーリング機能の実現を目的とする．POSIX 実時間シグナルは，高い I/O 多重処理性能を有する効率的なイベント通知機構として期待されている．しかし，その I/O モデルおよびプログラミングインタフェースが従来のポーリング I/O のものと大きく異なるため，利用が難しいという問題がある．本稿で提案する手法は，ポーリング I/O のインタフェースを提供する一方で，実装では POSIX 実時間シグナルを利用して各ソケットの状態をあらかじめテーブルで管理し，処理の高速化を図る．

The Design of a Polling I/O Mechanism with Real-time Signals

Eiji Kawai,[†] Youki Kadobayashi[††] and Suguru Yamaguchi[††]

To achieve high performance on network servers that handle thousands or tens of thousands of concurrent connections, efficient management of the multiplexed network I/O is a key. We discuss an application of POSIX real-time signals for faster polling I/O, i.e., poll() and select(). Although several research groups have revealed the real-time signal-driven I/O model can achieve high performance scalability, real-time signals are utilized by few network servers because of the completely different I/O model and programming interface. This paper describes the design of a communication management mechanism that provides the traditional polling I/O interface and implements its functionality by managing the state transitions of each connection notified with real-time signals.

1. は じ め に

近年のネットワークの高速化により，ネットワークサーバの負荷が大きな問題となってきている．そのため，ネットワークサーバにおいては，数千から数万ものコネクションを効率良く管理する，性能を重視したプログラミングモデルが重要になってきている．

従来，高い性能を目的としたネットワークサーバでは，select() や poll() によって実装されるポーリング I/O モデル[1) と呼ばれる I/O 機構がよく用いられる．これは，複数の記述子における I/O の即時完了性を 1 回の呼び出しでチェックするもので，単一の実行エンティティ（プロセスあるいはスレッド）で複数の記述子を並行して扱うことが可能となる．そのため，それぞれの記述子に実行エンティティを割り当てるマルチプロセスモデルやマルチスレッドモデルでは問題となる実行エンティティ管理コスト（生成，切替えのコスト）が発生せず，効率的な通信管理が可能となる．一方で，ポーリング I/O モデルは，扱う記述子の数の増加に対するスケーラビリティが低く，性能劣化の問題が広く知られている[2)]．

そこで近年着目されているのが，明示的な通信イベント通知機構を用いた通信管理方式である[2)~6)]．これは，ユーザプロセスが記述子集合の状態チェックをカーネルに依頼するポーリング I/O とは異なり，カーネルが個々の記述子を監視し，状態が I/O 可能状態に変化したときに，特別なインタフェースを通じてその記述子と状態変化情報をユーザプロセスに通知するモデルである．こうした機構の実装はこれまでに複数リリースされているが，それらの中で，POSIX 実時間シグナルを用いたイベント通知機構は，POSIX 実時間機能の拡張（IEEE 1003.1b-1993）において標準

† 奈良先端科学技術大学院大学附属図書館研究開発室
Research Division of Digital Library, Nara Institute of Science and Technology
†† 奈良先端科学技術大学院大学情報科学研究科
Graduate School of Information Science, Nara Institute of Science and Technology

化され，現在では多くの Unix オペレーティングシステムに実装されている機能である．しかし，この機構をそのまま用いるには，プログラミングモデルが従来のポーリング I/O と大きく異なる点や，例外処理の煩雑さなど課題が多い．実際にこの機能を用いているネットワークサーバは非常に少ないというのが現状である．

本研究では，この POSIX 実時間シグナルを用いて擬似的に従来のポーリング I/O モデルを実現する通信管理ライブラリを提案する．具体的には POSIX 実時間シグナルから得られる情報およびノンブロッキング I/O の実行結果を用いて，各ソケットの状態遷移を管理し，ポーリング I/O の呼び出しにおいては記録しておいた情報を返すというものである．本方式により，個々の記述子のポーリング処理におけるオーバヘッドが軽減され，より高速なポーリング I/O が実現される．

2. ネットワーク I/O の多重化とプログラミングモデル

ネットワークサーバは，複数のクライアントに並行してサービスを提供しなければならない．すなわち，複数のソケット I/O を並行して処理（多重処理）しなければならない．そこで，この多重処理のために，次にあげるようなプログラミングモデルがこれまでに確立されている．

- マルチプロセスモデル
- マルチスレッドモデル
- ノンブロッキング I/O モデル
- ポーリング I/O モデル
- 実時間シグナル駆動モデル
- システム依存の明示的イベント通知モデル

本章では，これらのモデルについてそれぞれ利点と欠点をまとめる．

2.1 マルチプロセスモデル

マルチプロセスモデルは，inted デーモンなどでも用いられ，プログラミングモデルとして単純な構造を持つ．そのため，サーバの実装が容易であるという利点を持ち，実際に広く利用されている．しかし，実行エンティティ（プロセス）の生成や管理のオーバヘッドが大きく，高い性能が得られにくいという問題がある．そのため，高い性能の達成を目的としたネットワークサーバでは利用されないのが一般的である．

2.2 マルチスレッドモデル

マルチスレッドモデルは，実行エンティティ（スレッド）の生成や管理のオーバヘッドがマルチプロセスよりも小さいという利点を持つ．これまでに，スレッド実装モデルはいくつか開発されているが，ユーザスレッドモデルを用いることによって，このオーバヘッドを可能な限り小さくし，高いスケーラビリティを達成する手法[7]も提案されている．具体的には，後で述べるノンブロッキング I/O および Linux の epoll 機構により I/O イベントを管理し，コルーチンを基礎にしたユーザレベルスレッドの切替えにより I/O の多重化処理を行っている．これにより，I/O がブロックしてしまった場合にプロセス全体の実行が中断してしまうという，ユーザスレッド特有の問題を解決している．しかし，ユーザレベルスレッドはマルチプロセッサシステムへの対応が難しく，カーネルによってスレッドを管理するモデルの方が普及している．

2.3 ノンブロッキング I/O モデル

ノンブロッキング I/O は，対象となるソケットに `fcntl()` を用いて `O_NONBLOCK` フラグをセットすることで有効となり，I/O 要求に際して呼び出し側をブロックしない方式である（同様の操作は `ioctl()` を用いても可能である）．いい換えると，I/O 要求時に即時処理可能な部分のみを実行するものである．たとえば，読み込むべきデータが何も到着していない場合は，データの読み込み要求に対して何もデータを返さずに呼び出しスレッドに実行を戻す．このように，ノンブロッキング I/O ではスレッドをブロックしないため，複数のソケットを並行して扱うにはソケット群に対して順に I/O を呼び出すだけでよい．

一方で，本方式は，読み込むべきデータが到着していない時点で読み込み I/O 要求が発行されたり，書き込むカーネルバッファスペースがないのに書き込み I/O 要求が発行されたりし，I/O 要求が不必要に呼び出されるという問題がある．特に I/O 要求はシステムコールで実装され，オーバヘッドが大きい．そのため，I/O の多重化を目的としてノンブロッキング I/O が単独で用いられることはほとんどなく，通常は他のプログラミングモデルと併用される．たとえば，次節で述べるポーリング I/O モデルにより，ある記述子の書き込みバッファが空いていると判明している場合でも，実際の空き容量は不明であることから，大きなデータを書き込もうとした場合にブロックされる可能性がある．こうした事態を回避するために，ノンブロッキング I/O が併用されることが多い☆．

☆ 一方で，ポーリング I/O が書き込み可能と判断する最小の空きバッファスペースを空きバッファ容量として用いることで，ブロックする可能性を回避することもできる．その場合は，バッファ空き容量を実際よりも小さく見積もることになり，書き込み

2.4 ポーリング I/O モデル

ポーリング I/O は，select() や poll() によって実装されており，記述子集合を引数にとり，それぞれの記述子の状態を返す機能である．ポーリング I/O を用いることにより，単一の実行エンティティで複数のソケットを並行処理するプログラミングが可能となる．

ポーリング I/O は，その記述子集合を走査するオーバヘッドが大きく，スケーラビリティに欠けるという問題[2)]がよく知られている．このオーバヘッドは，次にあげる 2 つの要因によって発生する[8)]．

(1) サーバが並行して管理する総ソケット数の増加にともない，リスト走査における処理コストが増加する．

(2) 通信の高速化にともない，記述子が I/O 可能に変化するイベント発生率が増加し，リスト走査の回数が増加する☆．

前者の問題を本質的に解決するためには，記述子の走査を不要にする新しいプログラミングモデルの開発が必要である．こうしたプログラミングモデルには，本研究で着目した実時間シグナルを用いたものや，新しく開発されたシステム依存のイベント通知機構などがある．それらについては，2.5 および 2.6 節で述べる．

また，後者の問題を解決しようとしているものには，ポーリング I/O の呼び出し頻度を直接制御し，サーバプロセスがビジーループ状態になるのを防ぐ手法[8)]がある．この方式の欠点は，スレッドを半強制的にブロックするため，場合によっては遅延時間が増加してしまうことや，並行ソケットの総数が非常に大きく（1 万を超えるような場合）かつその大半がアイドルであるような状況では，前者の問題，すなわちリスト走査の処理コストの問題が支配的になることである．

2.5 実時間シグナル駆動モデル

シグナル駆動型 I/O とは，ソケットが I/O 可能状態に変化する際に発行されるシグナルを用いたプログラミングモデルである．具体的には，fcntl() の F_SETSIG コマンドで，対象となるソケットと，状態変化の通知に用いるシグナルを指定し（指定しなかった場合は SIGIO が用いられる），同じく fcntl() の F_SETOWN コマンドでプロセス記述子を設定することでシグナルによる通知が有効となる．

このシグナル駆動型 I/O において，POSIX 実時間シグナルを用いるモデルが近年検討されている[5),6)]．実時間シグナルは情報を持ち，カーネルはイベントが発生したソケット記述子およびイベントの内容（読み込み可能，書き込み可能，エラー発生など）を通知することができる．また，実時間シグナルはキューに置かれるため，サーバプログラムはサービス処理をシグナル発生に対して非同期的に行うことができる．

このように実時間シグナルを用いたプログラミングモデルは非常に魅力的に見えるが，一方でプログラミングモデルが他と大きく異なるため，導入のコストが大きい．また，実時間シグナルを受信する際に，個々のシグナルに対してシステムコール呼び出しが必要となる問題[5)] や，シグナルキューの溢れの問題[6)] もある．特に後者の問題は，キューが溢れたときにイベント情報が失われるため，何らかの対処が必要となる．実装では，対象となる実時間シグナルのハンドラをデフォルト（SIG_DFL）に設定することでシグナル駆動型の I/O モデルを中止し，ポーリング I/O モデルへ移行することが多い．しかし，シグナルキュー溢れはシステム負荷の高いときに発生することが多く，さらに回復手段は実時間シグナル駆動なプログラミングモデルよりもコストが高いため，そのような対処を行うことは一般的に困難である．

2.6 システム依存の明示的なイベント通知機構を用いるモデル

ポーリング I/O における記述子集合の走査を不要にする，新しいプログラミングモデルが近年開発されている[2)〜4)]．実際に採用されているものには，Solaris や IRIX の poll デバイス（/dev/poll）[9)] や，FreeBSD の kqueue[10)]，Linux の epoll[11)] などがある．

これらの方式では，カーネルにあらかじめ記述子を登録しておき，カーネルがそれらの記述子において状態の変化を検出すると，特別なインタフェースを通じてプログラムに通知することにより，リスト走査を不要にしている．基本的には，実時間シグナル駆動モデルと同じ方式であり，実時間シグナルの改良と位置づけることができる．しかし，こうしたプログラミングインタフェースは近年開発されたものであり，標準化もされておらず，広く普及しているとはいいがたい．また，実時間シグナル駆動モデルの場合と同様に，従来のプログラミングモデルからの移行コストが大きいという問題もある．さらには，オペレーティングシステム依存のため，移植性に欠ける．

I/O が不必要に断片化される可能性がある．

☆ ポーリング I/O は，引数に与えられたソケット集合のうち，1 つでも I/O 可能状態になると終了する．一般的に，通信の高速化は，ソケットが I/O 可能状態に変化するイベントの発生率の増加を意味し，ポーリング I/O の呼び出し頻度の増加を引き起こす．そのため，ポーリング I/O を用いたサーバは，一定以上の負荷を与えられると，CPU アイドル時間のないビジー状態となる．

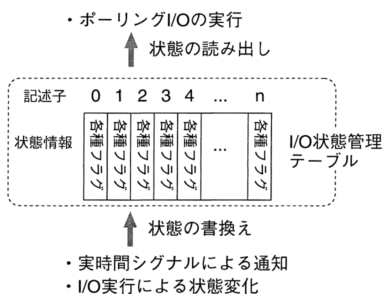


図 1　ソケット状態管理の基本構造
Fig. 1　Socket state management table.

3. 実時間シグナルを用いた通信管理機構の設計

本研究で提案する通信管理機構では，実時間シグナルおよびノンブロッキング I/O を用いて，擬似的にポーリング I/O の機能を実現する．本章では，その基本的な通信管理の仕組みと，ポーリング I/O 機能の実現について述べる．

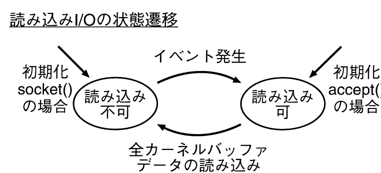
3.1　実時間シグナルによる通知イベントの管理

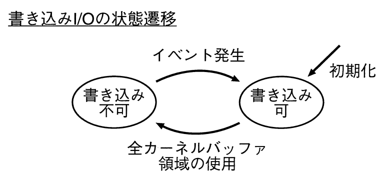
ポーリング I/O は，管理する記述子集合に対して個々の状態を得るものである．一方で，実時間シグナルは，個々の記述子が I/O 可能に変化する際に発行されるものである．また，I/O 不可への変化は，ノンブロッキング I/O の実行結果から検出することができる．そのため，ポーリング I/O を実現するためには，実時間シグナルやノンブロッキング I/O によって検出される状態変化を静的な記憶領域に記録しておき，要求されたときに返せばよい．**図 1** は，各ソケットの状態を保持する I/O 状態管理テーブルを示したものである．

一方，実時間シグナルによるイベントとソケットの状態遷移の様子をまとめたものが**図 2** である．このように，I/O 実行による状態変化の記録では，読み込み，書き込み，エラーの状態について管理することができる．

3.1.1　読み込み I/O

読み込み I/O では，読み込み可と判明している記述子に対してのみ実際のシステムコールをともなう I/O を実行する．そこで，用意したユーザバッファのサイズより小さいデータしか読み込めなかった場合に，そのソケットは読み込み不可に変化したことが分かる．また，状態の初期値については，`socket()` 呼び出しで得られたソケットの場合，読み込み不可としておけばよい．サーバにおいて `accept()` を呼び出して得られた新しいソケットについては，`accept()` 終了後，実時間シグナルをセットするまでの間にデータが到着してしまう可能性がある☆．そのため，`accept()` によって得られたソケットについては，読み込み可として初期化することでこの問題を回避する．一方で，データが届いているか未確認のまま読み込み可に設定してしまうため，1 回余分に読み込み I/O が実行される可能性があるというオーバヘッドがある．しかし，この I/O によりバッファの状態が正しく検出されるため，次からは無駄に I/O が実行されることはない．





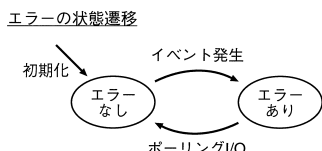


図 2　イベント通知とソケット状態の遷移
Fig. 2　Socket state transitions.

3.1.2　書き込み I/O

書き込み I/O については，書き込もうとしたデータサイズよりも小さいサイズしか実際には書き込めなかった場合，そのソケットは書き込み不可に変化したことが分かる☆☆．ここで注意しなければならないのは，実時間シグナルは I/O 不可状態から I/O 可能状態への変化をイベントとして通知する点である．そのため，書き込みバッファの状態については，状態の初期化において書き込み可に設定する必要がある．

3.1.3　エ　ラ　ー

エラーについては，実時間シグナルはその発生時に一度通知してくれるだけである．そのため，対処によってエラー状態が解除されても，知ることができない．

☆ クライアントでは，TCP の 3-way handshake において，2 番目の SYN パケットが到着したら，3-way handshake 最後の ACK パケットを送信するとともに，そのコネクションは確立されたものとしてデータの送信が可能となる．そのため，クライアントの実装依存ではあるものの，サーバではコネクション確立後すぐデータが到着する可能性が高い．

☆☆ I/O ではノンブロッキング I/O を用いていることに注意．

一方で，エラー情報が通知されれば，サーバプログラムによって何らかの対処がなされると考えられる．本通信管理機構では，エラーはポーリングI/Oによりサーバプログラムに通知されたときに解除している．

3.2 ノンブロッキングI/Oの併用

提案する通信管理機構では，実際のI/Oにおいてノンブロッキング I/O を併用している．ノンブロッキング I/O を用いる理由は，以下の3つの理由による．

1つは，I/Oにおける例外的なブロックを排除することである．ブロッキング I/O では，それがポーリング I/O などにより I/O 可能であると判断される場合でも，バッファの空き容量を超えるデータの書き込みを試みた場合など，ブロックされる場合がある．

2つめの理由は，I/Oによりソケットの状態を管理することである．特に書き込みI/Oにおいては，ブロッキング I/O を用いる場合，カーネルバッファの空き容量が足りない場合でも，ブロックすることで最終的には全I/Oが実行されるため，状態を知ることができない．ノンブロッキング I/O では，即時可能な分のみ実行されるため，その結果によりバッファの空き具合いを知ることができる．

3つめの理由は，記述子の状態変化の曖昧性の問題を回避することである．記述子の状態変化の曖昧性とは，以下のように説明される．記述子からデータを読み込む際に，アプリケーションバッファのサイズと同じサイズのデータが得られた場合，本通信管理機構ではまだデータがカーネルバッファに残っていると判断し，読み込み可の状態を保持している．しかし，読み込みI/O前にカーネルバッファに格納されていたデータのサイズが，アプリケーションバッファのサイズと同じであった可能性もある．実時間シグナルは，読み込み可の状態から読み込み不可の状態へ遷移した場合には発行されない．そのため，さらなるデータの読み込みを実行しなければこの判断はできない．ここで，ブロッキング I/O を用いると，カーネルバッファにデータが残っていない場合にブロックされるため，非効率的である．同様の問題は，データの書き込みの際にも発生する．データ書き込みにおいて，用意したアプリケーションバッファのデータをすべて書き込むことができた場合でも，ちょうどカーネルバッファが一杯になり，書き込み不可の状態に遷移している場合がある．本通信管理機構では，こうした問題を回避するために，ノンブロッキング I/O を用いている．

一方で，ブロッキング I/O を用いているネットワークサーバにノンブロッキング I/O を基本とする本通信管理機構を導入する場合，I/Oにおいてセマンティクスが異なるため，注意しなければならない点がいくつかある．1つは，EAGAIN エラー（システムによっては EWOULDBLOCK の場合もある）に対処することである．たとえば，読み込むべきデータがないのに読み込みI/Oを呼び出した場合などに，このエラーが返される．その場合，単にI/O処理を中断し，その時点でのI/O実行状況を保持したまま次の記述子の処理に移ればよい．また，こうしたエラーは，本通信管理機構において，記述子の状態をI/O可能状態からI/O不可状態に遷移させるため，連続して発生することはない．

もう1つの注意すべき点は，書き込みI/Oにおいて，アプリケーションバッファのサイズより小さいデータしか書き込まれなかった場合への対処である．一般的に，ブロッキング I/O を用いている場合にはこのような現象は起こりえず，エラーとして認識されるべきである．ノンブロッキング I/O では，こうした事態は当然起こりうることであり，サーバでは書き込んだデータのサイズ分だけポインタを進めるなどすればよい．

3.3 クローズされた記述子の扱い

実時間シグナルを用いたイベント通知では，クローズされた記述子の扱いには注意が必要である．シグナルキューにイベント情報が残っている状態で，その記述子がクローズされた場合，そのイベント情報は誤ったものになる．文献3)では，FreeBSD の kqueue における同様の問題が議論されている．こうしたイベント情報は，記述子がクローズされている間に取り出されれば対応が可能である．しかし，記述子がクローズされた後にただちに再利用される場合，問題である．

この問題は，記述子をクローズする際に，シグナルキューを走査し，該当するイベント情報を取り出すか無効にすることで解決することができる．しかし，そのためにはオペレーティングシステムへの変更が必要であり，現実的には移植性の問題などが発生する．そこで本通信管理機構では，以下の手順によりこの問題を回避する．

(1) 記述子をクローズする場合には，shutdown() を用いてソケットを閉じ，その後別のリストでその記述子を（クローズしないで）管理する．
(2) シグナルキューから情報を取り出す際に，イベントがキューに残っていないことを検出した場合に，リストに保管されている記述子集合をクローズする☆．

☆ 実際には，4.1 節で述べるように，これらの記述子をクローズ可能とマークし，さらに別のスレッドによって遅延クローズを行っている．

本方式が正しく動作するためには，シグナルキューにイベントを残さないように，可能な限り高速に取り出すようにしなければならない．そのため実装においては，このイベント取り出しを行う別のスレッド（シグナル取得スレッド）を用意し，その実行優先度を高く設定している．シグナル取得スレッドは，シグナルを取得し，その情報に応じて 3.1 節で述べた I/O 状態管理テーブルを更新する．このシグナル取得スレッドの実行優先度を高く設定することは，低遅延の処理のためだけでなく，次節で述べるシグナルキューの溢れの問題を回避するためにも重要である．

3.4 シグナルキュー溢れへの対処

実時間シグナルを用いる場合，シグナルキューの溢れの問題がある．しかし，2.5 節で述べたように，この問題への対処は一般的に困難とされている．

本提案手法では，この問題を回避するために，シグナルキューからのイベント取り出しを最優先で行うように設計している．しかし，それでも不十分なほどシステムが過負荷状態になることも考えられる．そこで，シグナルキューの溢れに対して，全ソケットを I/O 可能状態（読み込み，書き込みともに可）に設定することで対処している．これにより，読み込み I/O および書き込み I/O がアプリケーションの必要に応じて実行され，その結果により I/O 状態管理テーブルの情報が正しく再設定されることになる．本方式は次の利点を有している．

- 処理を I/O に集中することにより，スループットを向上させ，一時的な過負荷状態をすばやく解決することができる．
- 他の I/O モデルへの移行が不要であり，継続して実時間シグナル駆動モデルを利用することができる．

一方で，こうした I/O 状態の設定は，不必要な I/O 実行を招き，逆に非効率となる可能性もある．一般的に，シグナルキュー溢れにより失われた情報は回復されなければならないが，その回復が遅延したとしても処理の遅れにつながるだけで，致命的な問題にはならない．そのため，**図 3** に示すように，実装ではこの I/O 状態の操作を設定により与えられた一定時間（たとえば 1 秒間）遅延させて行い，その間のシグナルキュー溢れを集約して対応するようにしている．

3.5 ブロッキング I/O の実現

提案する通信管理機構では，すべてのソケットをノンブロッキング化している．一方で，ブロッキング I/O が必要になる場合もある．そこで，各ソケットに条件変数を用意し，読み込みおよび書き込みが可能になるイベントを待つことができるようにした．実時間シグナルにより，ソケットが I/O 可能状態に変化したことを検出すると，そのイベントを待っているスレッドがあるかどうかチェックし，あればそのスレッドのブロックを解除する処理を行っている．

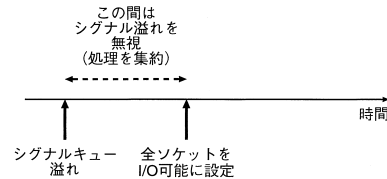


図 3 シグナルキュー溢れとその遅延処理
Fig. 3 Delayed processing of signal queue overflow.

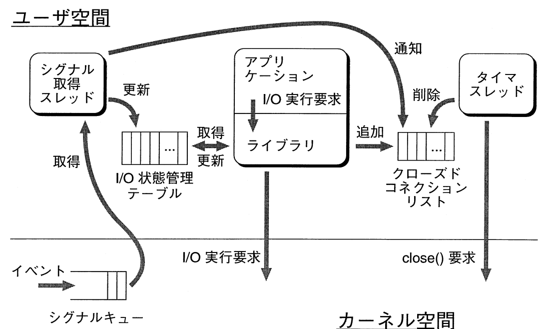


図 4 ライブラリにより生成されるスレッドとその役割
Fig. 4 Threads generated by the library.

4. 実 装

本研究では，提案する通信管理機構のプロトタイプ実装をライブラリとして実装した．実装に使用した言語は C 言語で，全体で 4,000 行程度のコンパクトなものである．本章では，その基本的な構造および API について述べる．

4.1 スレッド構造

本ライブラリは，直接リンクするプログラムのほかに，2 つの外部スレッドを生成する．スレッドを用いている理由は，本ライブラリを用いるプログラムとは非同期的な処理を行うためである．生成されるスレッドは，シグナル取得スレッドと，タイマスレッドである．これらのスレッドの関係を**図 4** に示す．

シグナル取得スレッドは，まず各記述子の状態を記録しておくテーブルのためのメモリ領域を用意し，シグナルハンドラとしての設定を行う．その後，シグナルキューからシグナルを取り出し，シグナルに格納された情報に依存した処理を行う．また，シグナルキューが空であることを検出した場合，3.3 節で述べたように，クローズ可能な記述子を記録する．

タイマスレッドは，100 ミリ秒間隔の断続的な処理サイクルを繰り返すスレッドである．その実行時には，閉じられたコネクションの記述子リストを走査し，クローズ可能なものに対して close() を呼び出す．

なお，I/O 状態管理テーブルの状態更新については，相互排他ロック変数を用いて排他制御を行っている．特に，I/O 可能状態であるにもかかわらず，I/O 不可状態に情報を上書きしてしまうと，場合によっては I/O が永遠に実行されない状況が発生する．こうした状況を回避するために，アプリケーションにおける I/O 実行と，その結果による I/O 状態管理テーブルの更新は，同一のクリティカルセクション内に実装している．

4.2 API

本ライブラリは，システムにより提供される I/O 機能に代わるいくつかの API を提供している．具体的には，ufdSocket()，ufdAccept()，ufdPoll()，ufdSelect()，ufdRead()，ufdWrite()，ufdClose() などである．また，ブロッキング I/O 用のインタフェースとして，ufdAcceptBlock()，ufdReadBlock()，および ufdWriteBlock() を用意している．

5. マイクロベンチマークによる評価

提案する通信管理機構の性能を評価するために，マイクロベンチマークを行った．本章ではその結果について述べる．

5.1 マイクロベンチマークプログラム

本稿で提案する通信管理機構の対象は，数千から数万もの同時コネクションを扱うサーバである．特に Web サーバは，永続コネクション（persistent connection）の導入により，多数のアイドルコネクション（データのやりとりのないコネクション）と，少数のアクティブコネクション（データのやりとりがあるコネクション）が混在する状態であることが多い．そこで，本研究ではそのような状態を再現するマイクロベンチマークプログラムを実装して評価を行った．

具体的には，5 台のクライアントホストと 1 台のサーバホストが同一セグメントにギガビットイーサネットで接続され，クライアントからサーバへ多数のコネクションをあらかじめ確立しておき，それらの上で擬似的なサービス要求を送信するものである．実験環境を**図 5** に示す．

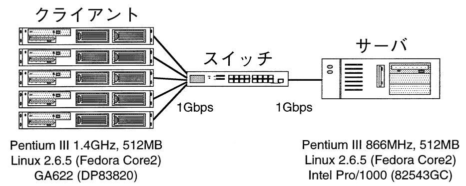


図 5 マイクロベンチマーク実験環境
Fig. 5 Microbenchmark test environment.

実際のサービス処理は，クライアントが 1 KB のデータを要求に見立てて送信し，10 KB のデータを応答に見立てて受信するというモデルを実装している．提案する通信管理機構の性能を純粋に計測するため，データの内容については何ら処理を行っていない．また，大半のコネクションがアイドルコネクションであるという状況を再現するために，クライアントにおける各コネクションにおいて，要求の送信開始から応答の受信完了までをアクティブ期間と定義し，このアクティブな期間にあるコネクションの総数を各クライアントで制限している．ベンチマークテストでは，コネクション総数およびアクティブコネクションの全コネクションにおける割合（ACR: Active Connection Ratio）の 2 つをパラメータとしている．具体的には，コネクション総数を 5,000 から 40,000 とし，それぞれの場合においてアクティブコネクションの割合を 0.5%から 4.0%まで変化させテストした．サーバにおけるポーリング I/O には，従来の poll() および我々の開発した ufdPoll() を用いている．

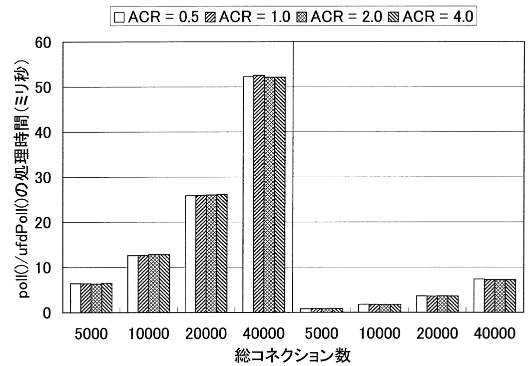


図 6 ポーリング I/O 呼び出しあたりの処理時間（ミリ秒）．左半分のグラフは poll()（従来手法）の場合，右半分のグラフは ufdPoll()（提案手法）の場合
Fig. 6 Mean processing time of a polling I/O invocation. The left half is for traditional poll() and the right half is for our ufdPoll().

5.2 ポーリング I/O の処理速度およびサービススループット

図 6 に，ベンチマークテストにおけるポーリング I/O の呼び出しあたりの処理時間（poll() および ufdPoll() の実行時間）を示す．このグラフに見られるように，ポーリング I/O の処理時間は総コネクション数にほぼ比例しており，アクティブコネクショ

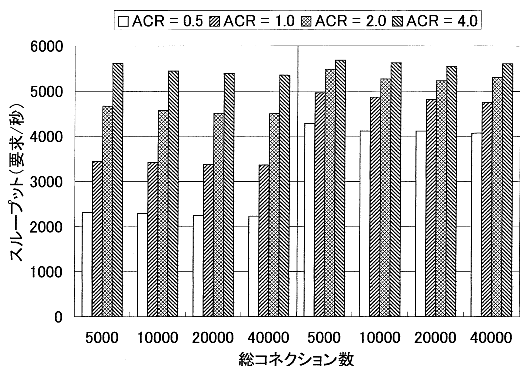


図 7 サービススループット（要求/秒）．左半分のグラフは poll()（従来手法）の場合，右半分のグラフは ufdPoll()（提案手法）の場合

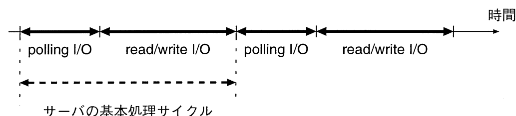
Fig. 7 Service throughput. The left half is for traditional poll() and the right half is for our ufdPoll().

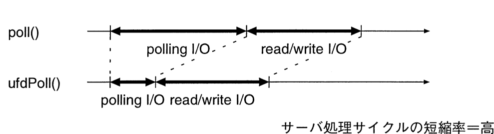
ンの割合はほとんど影響を与えていないことが分かる．ポーリング I/O が行う処理は，引数に与えられた記述子集合の状態を返すことであり，各記述子の状態によって処理負荷が変化するものではないことから，これは自然な結果である．

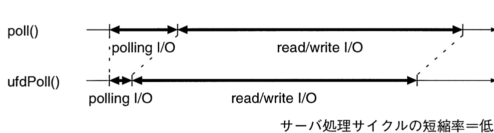
一方で，このポーリング I/O の処理時間は，本稿で提案する通信管理機構により，大きく減少していることが分かる．従来の poll() と比較して，80%以上の削減となっている．これは，従来のポーリング I/O の場合，システムコール呼び出しのためのカーネル内メモリ確保やコピーを行ったり，各ソケットの状態をチェックするために VFS のファイル構造体から各ソケットのポーリング関数を呼び出し，数多くのポインタ経由の操作を行ったりするなど，オーバヘッドが大きいのが原因である．提案手法の場合，各ポーリング I/O の処理では，より少ないメモリコピー，ポインタ操作，ビット演算ですんでいる．

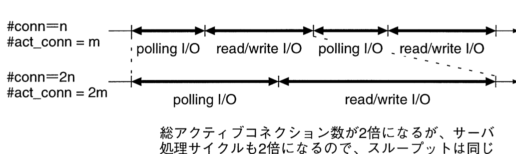
次に，ベンチマークテストにおけるサービススループットを**図 7** に示す．本提案手法を用いた場合と用いなかった場合を比較すると，アクティブコネクションの割合が小さい場合，本提案手法による改善が大きいことが分かる．これは，提案手法が，従来のポーリング I/O と比較してその処理時間を短縮していることが直接の理由である．

この現象を詳しく見るために，サーバの挙動をミクロな視点から説明したものを**図 8** に示す．この図の (1) のように，サーバはポーリング I/O の処理とその結果に基づいた I/O の処理からなる基本処理サイクルを持っている．マイクロベンチマークテストにおいて計測された基本処理サイクルの時間を**図 9** に示す．サーバは，この基本処理サイクルの間に到着した









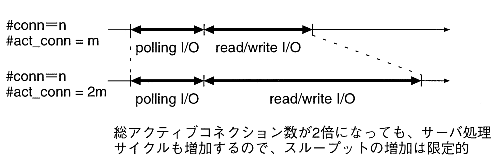


図 8 サーバにおける処理サイクル

Fig. 8 Processing cycle on the server.

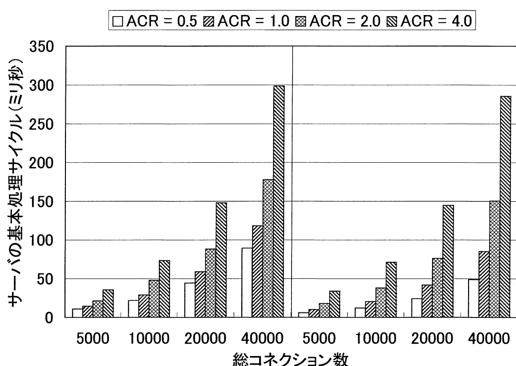


図 9 サーバにおける基本処理サイクルの時間（ミリ秒）．左半分のグラフは poll()（従来手法）の場合，右半分のグラフは ufdPoll()（提案手法）の場合

Fig. 9 Mean server processing cycle. The left half is for traditional poll() and the right half is for our ufdPoll().

要求，すなわちアクティブコネクション上で送られてきた要求を処理する．そのため，サーバのサービススループットは，この基本処理サイクルの長さと，その中で処理を行う要求の数によって決定される．ここで，アクティブコネクションの割合が小さい場合を考えると，基本処理サイクルあたりに処理する要求が少なくなるため，I/O 処理時間が小さくなり，相対的にポーリング I/O の処理時間が大きくなる．そのため，図 8 (2) に示すように，サーバの基本処理サイクルの短縮率が大きくなり，大きな性能改善が得られるのである．一方で，アクティブコネクションの割合が大きくなると，図 8 (3) に示すように，提案手法による性能改善は小さくなる．

また，図 7 の結果において，アクティブコネクションの割合を固定すると，総コネクション数を増加してもスループットは増加していない．総コネクション数の増加は総アクティブコネクション数の増加を意味するため，直感的にはスループットの増加が期待される．しかし，図 8 (4) で説明されるように，サーバの基本処理サイクルも比例して増加するために，スループットの増加にはほとんど貢献していない．図 9 の実際に計測された基本処理サイクル時間もそのことを裏付けている．

ここで，総コネクション数を固定し，アクティブコネクションの割合を増やしても，それに比例してスループットは増加しないことにも注意が必要である．これは，図 8 (5) に示すように，アクティブコネクションが増加すると，I/O の処理時間が増加し，サーバの基本処理サイクルが増加してしまうためである．

最後に，マイクロベンチマークテスト中に計測されたサービス応答時間を**図 10** に示す．ここでのサービス応答時間とは，クライアントが要求をソケットに書き込む直前から，その応答をソケットから読み取り終わるまでの時間である．グラフに示されているように，応答時間はサーバの基本処理サイクルより若干小さいものの，ほぼ同様の水準であることが分かる．

なお，本研究で行ったマイクロベンチマークテストでは，総コネクション数と，アクティブコネクション数をパラメータとして同一の条件としている．これは，ポーリング I/O の処理時間を提案手法の実装により削減する一方で，I/O 処理時間については，従来のポーリング I/O と提案手法とで同一の実験条件にするためである．一方で，ベンチマークテストの手法として，要求レートを同一の条件にすることも考えられる．これは，サーバの処理能力を上回る要求レートを設定することで，サーバの過負荷状態を調査することができる．しかし，クライアントも容易に過負荷状態になってしまうため，緻密なプログラミングと高い処理能力が求められる．そのため，本研究ではまだ実現しておらず，今後の課題としたい．なお，本研究で行ったベンチマークテスト手法は，与えられた条件に対してサーバの最大スループットを容易に計測することができるという利点を持つ．

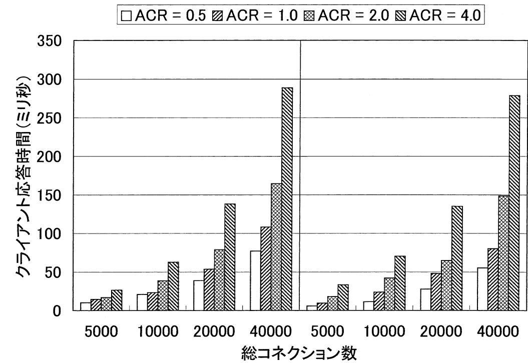


図 10 クライアントで計測されたサービス応答時間（ミリ秒）．左半分のグラフは `poll()`（従来手法）の場合，右半分のグラフは `ufdPoll()`（提案手法）の場合

Fig. 10 Mean service time measured by clients. The left half is for traditional `poll()` and the right half is for our `ufdPoll()`.

6. 議　　論

本章では，本通信管理機構において未解決の問題点と，本研究から得られた知見をもとに，ネットワーク I/O 性能のさらなる向上のための技術展望について述べる．

6.1 実装における Unix ソケットセマンティクス上の制約

本提案手法を既存のサーバ実装に組み込む場合には，次の 2 つの制約が生じる．1 つは，すべてのソケット I/O を本通信管理ライブラリ経由にする必要があることである．本実装では，実時間シグナルから得られる状態変化の情報だけでなく，通常の I/O において得られた結果なども利用しながら，各ソケットの状態を管理している．そのため，オペレーティングシステムや他のライブラリで用意されている I/O 機構が直接呼び出された場合，当然ながら正しい動作を保証することができない．しかし，この問題は，システムで用意されているソケット I/O 機構の種類が一般的に少ないため，対応はそれほど難しくないと考えている．

もう 1 つの制約は，マルチプロセスモデルへの対応である．本実装では，マルチスレッドを用いており，現状ではマルチプロセスモデルには対応していない．

これは，マルチスレッドプログラムにおける fork() 呼び出しのセマンティクスが複雑であることが主な原因である．特に，グローバル変数として管理している各種データ構造の初期化だけでなく，本ライブラリで用いる補助スレッド（シグナル取得スレッドおよびタイマスレッド）の明示的な起動および初期化が，プロセス生成時に必要である☆．これらの実装は今後の課題としたい．

6.2 ネットワーク I/O のさらなる高速化実現に向けて

2 章で述べたように，本研究を含め，ネットワーク I/O の高速化のための技術がこれまでに数多く開発されてきた．それらにより得られた知見は多岐にわたるため，簡単にまとめることはできないが，今後のさらなる高速化実現のために考えるべき点を以下にあげる．

1 つは，イベント管理のオーバヘッド削減だけでなく，I/O 処理能力の向上も重要になってくるということである．本研究で詳細に見たように，I/O 処理そのものに多くの処理能力が割かれるような状況では，従来のポーリング I/O でも比較的高い性能が達成される．文献 6)，12)，13) においても，特に accept() の呼び出し方を工夫することにより，従来のポーリング I/O は実時間シグナルや Linux の epoll と同等に近い性能が達成されることが報告されている．特に今後，ネットワーク上で交換されるデータのサイズは大きくなることが予想され，I/O 処理能力そのものの改善が求められる．これまでに，ゼロコピープロトコルスタック[14] やカーネルモードサーバ[15] など，メモリコピーの削減による性能改善が提案され，実際に市中技術として採り入れられている．一方で，TCP/IP プロトコル処理そのものの高速化はまだまだ不十分である．特に，近年注目されているネットワークプロセッサ技術を用いた TCP/IP オフローディングは，まだ同時コネクション数に対するスケーラビリティに問題があるものの，今後の発展が求められる技術である．

もう 1 つの課題は，高遅延環境におけるネットワーク I/O 性能の向上である．本研究のマイクロベンチマークでは，総トラフィックとして 400 Mbps を超えるスループット☆☆を達成しているが，こうした高いスループットを達成するには，遅延とパケットロスが無視できるほど小さい環境が必要である．実際のインターネット上で運用されている Web サーバでは，今回の実験よりもはるかに低いスループット（場合によっては 10 分の 1 以下）でプロセッサ処理能力の限界に達してしまうことが経験的に知られている．これは，実環境では TCP 輻輳ウィンドウサイズの増加が遅く，再送も発生することから，カーネルが管理する処理中のコネクションの数が大きく増加するためである．すなわち，サーバプロセスが管理するより，はるかに多くのコネクションをカーネルは管理しているのである．この多数の処理中のコネクションにより，プロトコルスタック内部におけるメモリ管理や TCP のイベント管理において，高いオーバヘッドが発生していると推測される．そのため，こうしたオーバヘッドを軽減するプロトコルスタックの実装技術が今後は必要である．

☆ 具体的には，こうした初期化ルーチンを実装し，pthread_atfork() などによって，fork() のハンドラとして登録することで実現可能である．

☆☆ 1 トランザクションあたり要求 1 KB および応答 10 KB のデータ交換があり，たとえば毎秒 5,000 要求処理したとすると，約 450 Mbps のデータ交換となる．実際には，これにプロトコルオーバヘッドが加わることになる．

7. おわりに

本研究では，実時間シグナルを用いたネットワーク I/O の多重処理を行う通信管理機構を提案した．本機構の利点は，以下のようにまとめることができる．

- ポーリング I/O のプログラミングモデルをそのまま利用することができる．
- サーバにおける並行ソケット数に対するスケーラビリティが高い．
- 実時間シグナルは現在多くの Unix オペレーティングシステムで実装されており，移植性が高い．

今後は，2.6 節で述べた明示的なイベント通知機構（poll デバイス，kqueue，epoll など）もサポートしていきたいと考えている．こうした機構は，現在いくつかのオペレーティングシステムにおいて先進的な機能として実装されており，今後も改良が期待される．最終的には，システムにおける実装の詳細を隠蔽する高レベルかつ高性能な通信管理ミドルウエアとしての実装を目標としている．

謝辞 本研究の一部は，日本学術振興会科学研究費補助金若手研究（B）（課題番号 16700068）による助成を受けている．

参 考 文 献

1) McKusick, M.K., Bostic, K., Karels, M.J. and Quarterman, J.S.: *The Design and Implementation of the 4.4 BSD Operating System*, Addison-Wesley (1996).

2) Banga, G., Mogul, J.C. and Druschel, P.: A scalable and explicit event delivery mechanism for UNIX, *USENIX Annual Technical Conference* (1999).

3) Lemon, J.: Kqueue: A generic and scalable event notification facility, *USENIX Annual Technical Conference* (2001).
4) Provos, N. and Lever, C.: Scalable Network I/O in Linux, *USENIX Annual Technical Conference* (2000).
5) Provos, N., Lever, C. and Tweedie, S.: Analyzing the Overhead Behavior of a Simple Web Server, *4th Annual Linux Showcase & Conference* (2000).
6) Chandra, A. and Mosberger, D.: Scalability of Linux Event-Dispatch Mechanisms, *USENIX Annual Technical Conference* (2001).
7) von Behren, R., Condit, J., Zhou, F., Necula, G.C. and Brewer, E.: Capriccio: Scalable Threads for Internet Services, *19th ACM Symposium on Operating Systems Principles* (2003).
8) 河合栄治，門林雄基，山口　英：ネットワークサーバにおける多重化 I/O の実行間隔制御による性能向上手法，情報処理学会論文誌，Vol.45, No.2 (2004).
9) Acharya, S.: Using the devpoll (`/dev/poll`) Interface, Technical Articles (2002). http://access1.sun.com/techarticles/devpoll.html
10) FreeBSD manual page of `kqueue(2)` (2000).
11) Linux manual page of `epoll(4)` (2002).
12) Brecht, T., Pariag, D. and Gammo, L.: accept()able Strategies for Improving Web Server Performance, *USENIX Annual Technical Conference* (2004).
13) Gammo, L., Brecht, T., Shukla, A. and Pariag, D.: Comparing and Evaluating epoll, select, and poll Event Mechanisms, *Linux Symposium 2004* (2004).
14) Chu, H.K.J.: Zero-Copy TCP in Solaris, *USENIX Annual Technical Conference* (1996).
15) Joubert, P., King, R.B., Neves, R., Russinovich, M. and Tracey, J.M.: High-Performance Memory-Based Web Servers: Kernel and User-Space Performance, *USENIX Annual Technical Conference* (2001).

(平成 16 年 7 月 22 日受付)
(平成 16 年 10 月 30 日採録)

河合　栄治（正会員）

1996 年京都大学理学部数学科卒業．1998 年奈良先端科学技術大学院大学情報科学研究科博士前期課程修了．2001 年同大学同研究科博士後期課程修了．2000 年 10 月より，科学技術振興事業団さきがけ研究 21「機能と構成」領域研究員．2003 年 4 月より，奈良先端科学技術大学院大学附属図書館研究開発室科助手．分散計算環境の構築，インターネットにおける大規模情報配信システムの開発，高速 I/O 指向オペレーティングシステムの研究に従事．博士（工学）．

門林　雄基（正会員）

1996 年大阪大学大学院基礎工学研究科物理系専攻博士後期課程中退．同年大阪大学大型計算機センター助手．1997 年大阪大学大学院基礎工学研究科物理系専攻情報工学分野より博士（工学）取得．1999 年大阪大学大型計算機センター講師．2000 年奈良先端科学技術大学院大学情報科学研究科助教授．WIDE プロジェクトボードメンバ．Content Routing Network Forum 代表．情報通信研究機構セキュリティ高度化グループ短期専攻研究員．レイヤ 7 での QoS 実現を目標とし，セキュリティ，CDN，マルチキャスト等の研究に従事．著書に岩波講座インターネット第 2 巻『ネットワークの相互接続』．

山口　　英（正会員）

1990 年 10 月大阪大学大学院基礎工学研究科情報工学専攻博士後期課程を中退し，大阪大学情報処理教育センター助手として着任．1992 年 10 月奈良先端科学技術大学院大学情報科学センター助教授．1993 年 4 月より，同大学情報科学研究科助教授．2000 年 4 月より，同大学情報科学研究科教授．2004 年 4 月より，内閣官房情報セキュリティ補佐官（併任）．大規模分散処理環境構築，ネットワークセキュリティ等の研究を行う．また，WIDE Project のメンバとして，広域コンピュータネットワークの構築・研究に従事する．工学博士．