システムソフトウェアと
オペレーティング・システム 88－5
（2001．7．27）

User Level Flow Control API をもつ並列ライブラリの実装

神里 健司‡ 河野 真治†
†琉球大学工学部情報工学科
‡琉球大学理工学研究科情報工学専攻

概要

現在MPI、PVM等、通信にTCPを用いた並列ライブラリがあるが、ネットワークレベルの情報は全てカーネル以下のレベルに隠蔽され、ユーザレベルでフローコントロールを行うことはできない。しかし並列プログラミングでは、やりとりするデータの性質に応じた細かなフローコントロールをプログラムが動的に行いたい場合がある。本稿では、これまでに我々が実装してきたUDPベースの並列ライブラリにフローコントロールAPIを付加し、プログラムが細かなフローコントロールを行えるようにするのが目的である。さらに、並列ライブラリに通信の一時中断・再開のAPIを加え、超分散的な処理も行えるようにする。

Parallel Library with User Level Flow Control API

‡Takeshi Kamizato †Shinji Kono
†Department of Information Engineering, University of the Ryukyus.
‡Specialty of Information Engineering, University of the Ryukyus.

Abstract

Currently, MPI and PVM are used for parallel library. These are using TCP to communicate. If we use them, we can't do flow control in program because acknowledge is invisible from user level. In parallel programming, however, we want to do flow control to adapt communication types. So, we implement user level flow control API on the UDP based parallel library. And we add disconnect/reconnect API for super distributed computation environment.

並列プログラミングでは、ユーザプログラムが通信するデータの性質や各ノードの負荷に応じて動的に通信のフローコントロールを行えば、より効率的な負荷分散が可能である。

MPIの主な実装やPVMは、通信はTCPを用いて実装されている。しかし、TCPを用いた実装では、フローコントロールは全てカーネル内のTCPルーチンでおこなわれ、ユーザプログラムはTCPのオプションを操作することしかできず、動的で細かなフローコントロールは行えない。アプリケーションをUDPを用いて実装した場合、信頼性を自前で付加するためフローコントロールは全てユーザプログラムが行える。

UDPを用いたコネクションレス通信では、最小トランザクション時間がTCPよりも小さい。また、アプリケーション間でデータサイズについて合意がなされている場合、TCPに実装されているカーネルレベルでの流量制御は必要ない場合がある。しかし、データグラムに信頼性を付加する処理やフローコントロールを行う機構を実装するのは、非常に繁雑なプログラミングとなる。

そこで、UDPに信頼性とユーザレベルのフローコントロールAPIを付加し、ユーザレベルからのフローコントロール可能な通信ライブラリを提案する。

本稿では、これまでに実装した、信頼性の付加されたUDPベースの並列ライブラリにユーザレベルのフローコントロールAPIを付加し、ユーザ

プログラムが動的にフローコントロールを行える並列プログラムを環境を提供しする。

1 UDPライブラリ

1.1 UDPとTCP

UDPとTCPを比較した場合、一般に次のようなことが利点UDPの利点としていわれる。[6]

- ブロードキャスト・マルチキャストアドレスのサポート
- コネクションレス

UDPが持たないTCPの機能として次のようなものがあげられる。

- コネクションの確立・開放
- 信頼性
- sliding windowによる流量制御
- 輻輳回避

TCPによるこれらの機構は通常カーネルに全て隠蔽されユーザプログラムからはflow controlalは行えない。これらのTCPの機能は、とくにコネクション数が1000を越えるような超分散的な環境ではオーバーヘッドになると考えられる。

また、アプリケーションによってはTCPの提供する信頼性やフロー制御は必要ない、もしくはアプリケーション側で制御したほうがよい場合もある。例えば、動画を転送したい場合などは、ある程度の信頼性は犠牲にしてもよいが、through-putは必要である。逆にデータベースの細かな更新等は、responseが必要とされる。

また、TCPはpeer-to-peerの通信しかサポートしておらず、超分散的な環境では通信の要求にしたがってそれぞれの相手にコネクションを確立する場合のオーバヘッドが無視できない大きさになる場合もあると考えられる。

このようにアプリケーション毎にもとめられる通信品質は異なり、TCPのオーバヘッドが無視できないような分散環境では、UDPを用いた実装によりユーザレベルで再送やフローコントロールを行ったほうがよい。

しかし、安易にUDPで実装すると、容易に受信側がオーバランするしやすい。また、アプリケーション毎にAcknowlegdeや再送等の信頼性を付加するルーチンを付け加えることは、非常に大変な作業である。

そこで我々は、現在実装されているデータグラムに信頼性を付加したUDPライブラリにフローコントロール用APIを加え、ユーザレベルからフローコントロール可能なライブラリを実装した。

1.2 UDPライブラリの特徴

今回ユーザレベルフローコントロールAPIを追加したUDPライブラリ[1]は以下のような特徴をもっている。

- 完全結合型の通信を、各ホスト毎のUDPソケットひとつで提供する。
- 通信相手先毎にblocking/de-blockingのためのqueueをもつ。
- 送出されるパケットは、ライブラリによって付加されたseqence番号をもつ。
- 再送のタイミングはユーザが指定する。

図1.2はこのUDPライブラリがアドレス毎にもつのデータ構造である。図1.2は、このUDPライブ

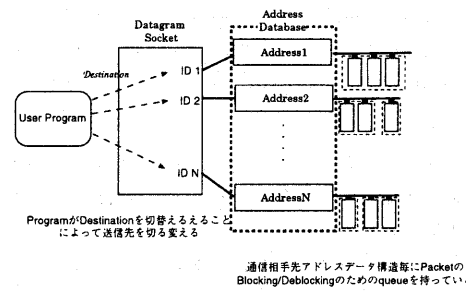

図 1: UDPライブラリの構成

ラリのセグメントヘッダである。このUDPライブラリは、データグラムに到着順序の保証と信頼性の付加するため、データグラムパケットにseqence番号を付加し、de-blocking/blocingのためにブロックのこのデータグラムパケットがどのブロックの

パケットであるかを、そのブロックの先頭パケットの sequence 番号によって区別する。Beginning of block フィールドはそのパケットが属するブロックの先頭パケットの sequence 番号が入る。

Destination ID と From ID は、アドレスデータ構造に登録されているアドレスとセットになった ID が入る。送信先／送信元は ID で指定される。

TCP パケットと違い、Acknowledge 番号のフィールドを独立にはもたない。コントロールフィールドの Ack 有効な場合、このパケットは Acknowledge として扱われデータは無視される。このとき、sequence 番号が Acknoledge 番号となる。Acknowledge パケットとデータパケットを別々にするこの方式は、全 2 重通信で双方から頻繁にデータパケットがやりとりされるときにはパケットが増えるという欠点があるが、パケットヘッダの大きさ自体が小さいため、どちらか一方のデータ転送が主で、受信側は Acknowledge を返すだけの場合、パケットヘッダのオーバヘッドを減らすことができる。

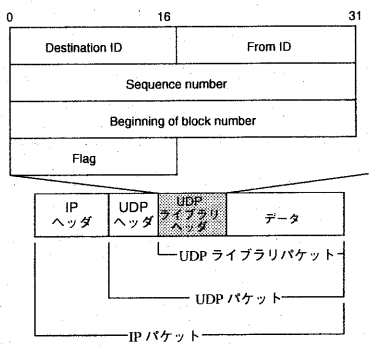

図 2: UDP ライブラリのセグメントヘッダ

1.3 UDP ライブラリの評価

TCP、UDP、UDP ライブラリを用いて簡単なトランザクションに対するレスポンスを計った。小さなファイルの要求/転送のプログラムをそろぞれで実装し、トランザクション数を増やしいった。図 1.3 はトランザクション数と全てのトランザクション終了までにかかった時間の関係である。転送したデータの大きさは、ファイル転送の要求の大きさが 128Byte 程度のメッセージ、実際のデータ大きさは 100KByte 程度である。TCP の場合、それぞれのトランザクション毎に TCP コネクションを張り直す必要があるが、UDP ライブラリの場合、送信先を切替える手間だけであり、ほとんどパケットの処理もオーバヘッドとならない。

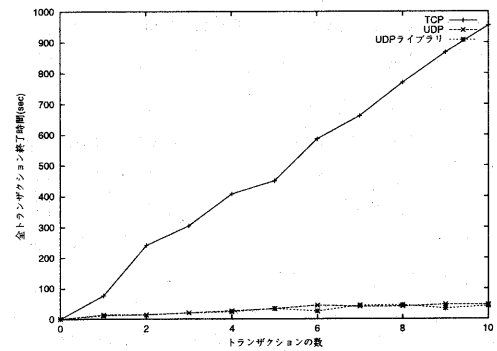

図 3: 処理時間とトランザクション数 (10M Ethernet,CPU=Pentium 200MHz,OS=BSD/OS4.2,NIC=3com Fast Etherlink XL, Switching HUB=Cisco Systems Catalyst1900)

次に、TCP、UDP ライブラリの 10Base-T Ethernet における through-put を測定した。TCP はソケットオプションは全てデフォルト値であり、受信バッファサイズ、送信バッファサイズはともに 8192Byte である。現在、UDP ライブラリは、ネットワークにデータグラムパケットを送出する際に、MTU にあわせて (10MBEthernet の場合は 1400Byte)blocking /de-blocking を行う。この実験ではライブラリレベルでの blocking/de-blocking の手間を少なくするため、write ブロックを 1400 としている。(表 1) この実験では read したデータはユーザレベルでなにも処理しない。

参考までに、TCP/IP の 10M イーサネット上のおける理論上の最大データ転送速度は 1,185,840Byte であることをあげておく。

表中の UDP ライブラリのスループットは 70KByte でパケット数がウィンドウサイズちょうど 50 パケットの時に測定した値である。このとき、再送要求やパケットの喪失は確認されてない。UDP ライブラリは、ライブラリが blocking/deblocking を

通信ルーチン	スループット
TCP/IP	1,067 KByte/sec
UDP ライブラリ	1,206 KByte/sec

表 1: 最大スループットの比較 (10M Ethernet,TCP/UDP ソケットの送信/受信バッファサイズ=8192Byte,CPU=Pentium 200MHz, OS=BSD/OS4.2, NIC=3com Fast EtherLink XL,Swiching HUB=Cisco Systems Catalyst 1900)

行うが、このユーザレベルの blocking/deblocking は大きなオーバーヘッドとなっている。wirte block のサイズを 2000byte にし、ライブラリが MTU サイズのデータグラムパケットへの分割を行う場合、through-put は 76Kbps にまで落ちる。しかし、通常 blocking/deblocking による手間よりも、メッセージの処理 (演算、コピー) の方が手間になるため、これがボトルネックになることはないと考える。

UDP ライブラリは、受信側のオーバランが発生しても、ユーザプログラムが再送を行えば信頼性は確保される。しかし、一度再送要求が起こると容易に輻輳が発生する。ライブラリ中のウィンドウサイズを 50 として、100 のパケットを高速に再送 write した時、再送要求が 7 つ発生し、through-put が 7Kbps まで減少する。このとき、転送したパケットの総数は再送パケットとあわせて 163 個であった。これは再送のタイミングをユーザレベルで指定しているためで、転送が終了しない場合、ユーザプログラムがライブラリに再送要求を非常に短い間隔で繰り返したためと考えられる。ユーザレベルで再送のタイミングを指定する場合、このように容易に輻輳が起こるので非常に注意深いプログラミングが必要になる。しかし、これはライブラリがある程度自動化するべきである。

ウィンドウサイズの調整やパケットのフラッシュのタイミング、再送のタイミングの調整はフローコントロールと同等である。この UDP ライブラリはある程度まではアプリケーションのフローコントロールの欠落を吸収するが、性能を引き出すには、UDP と同様プリケーションレベルでフローコントロールを行わなければならない。その半面、フローコントロールの機構をカーネルレベルに隠蔽している TCP に比べ、UDP ライブラリはユーザプログラムから動的に調整することが可能である。アプリケーションが持ち得る情報をもとに行うフローコントロールと、ライブラリが自動的に行うべきフローコントロールをすりあわせ、ユーザレベルにどのようなフローコントロール API を提供することを考えることが重要である。

2 ユーザレベルコネクション管理

フローコントロールはデータグラム型には存在せず、コネクション指向通信の上にしか存在しない。

現在の UDP ライブラリもデータグラムに sequence が付加され、単純なデータグラムにコネクションの概念があるが、ユーザレベルからコントロールする方法がない。そこで、ユーザレベルでコネクション管理を行うために、 TCP の 3 way hand shake と同様の方法を使い UDP パケットの sequence 番号を同期する機能を追加した。3 つのパケットにより双方の sequence 番号の同期がとれた時点でコネクションを確立できたとする。図?? はコネクション確立の一例である。まず、コネクション確立を試みる側がが SYN(同期) のフラグがたったパケットを送信する。この時パケットには、同期したい sequence 番号を sequence 番号フィールドにセットされている。このパケットをまだコネクションが確立していないアドレスから受け取ると、sequence をセットし、SYN と ACK のフラグが立ったパケットを送信する。UDP ライブラリのパケットは Acknowledge 番号のフィールドを独立に持たないため、SYN と ACK のフラグが立ったときには、SYN 用の seqence 番号は、データフィールドの最初の 4byte に格納される。こうしなければ、SYN/ACK パケットへの Acknowledge か独立な Acknoledge と区別できなくなる。

コネクション確立時には TCP と同様の手間がかかるが、このコネクションはユーザプログラムが管理することができる。

コネクションの確立は通常ライブラリ自動で行うが、sequence 番号のリセットや、通信の一時中断/再開用の API を用意し、ユーザプログラムが管理することもできるようにした。これによって、超分散的な処理を行うことも可能である。

3 ユーザレベルフローコントロール

TCP のフローコントロールは、受け取れる Window size の受信側から送信側への通知によって行われている。

このフロー情報をユーザレベルから制御することにより、より効果的なユーザプログラムが可能になると考えられる例を示そう。

3.1 例:並列検証系

タブロー法による時相論理式の検証は [2]、時相論理式であらわされた仕様記述をタブロー法に従って展開していく。この展開処理は、ある時刻のでの状態から次の時刻の状態への遷移に対応する。このように、状態論理式をノードとする多数の 2 分決定木を構成していく。この木の生成処理を分散して並列に行わせる。並列化の手法として生成した状態式全てに固有の ID をふり、ID をもとにハッシュ値を計算し各展開サーバに状態式の展開処理を割り当てる。状態式の ID は全体で共有されいなければならず、その管理は部分項サーバで行う。図 3.1 は並列検証系の構成である。

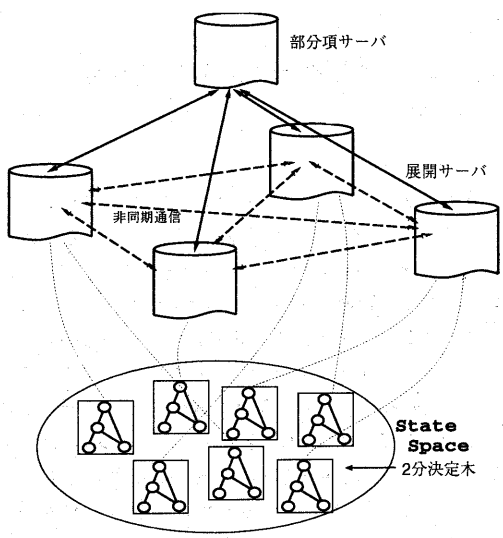

図 4: 並列検証系

我々が実装した並列検証系 [2] の場合、各展開サーバは自立並列的に展開を進めていく。ある時間をすぎると通信密度は小さくなるが、ランダムなタイミングで展開サーバ間同士の通信が発生するため、TCP のコネクションを各展開サーバ間でオーバヘッドが大きく、コネクションレスの UDP で完全結合型通信を実現するのが自然である。並列時相論理検証系では、展開サーバ間ではランダムなタイミングで、さまざまな大きさのデータが通信されるため、TCP のよる実現はサーバの数の 2 乗に比例するソケットのコネクションを持たなければならない。この場合、通信データの量が少ないホスト間での TCP コネクションに使用するリソースは無駄が大きい。したがって UDP によるコネクションレス通信によって、完全結合型通信を実現するのが自然であった。

上記の並列検証系の場合、部分項サーバと各展開サーバ間の通信は新たに生成された状態式の ID を交換を行うのみである。この通信はデータ量は比較的多くないが、系全体で状態式と ID のセットは共有されていなければならないのと、状態式が新たに生成される度に部分項サーバと各展開サーバ間の通信が発生するため遅延が問題となる。逆に、各展開サーバ間の通信は、新たに生成された状態式の展開を他の展開サーバに依頼する、状態式そのものの通信であるため、比較的大きなサイズ単位 (最大数キロ byte) の通信が必要である。展開処理そのものは各展開サーバで独立に行われるため、多少遅延があっても問題ない。また、負荷の集中している展開サーバに、さらに巨大な状態式を転送すると、展開処理と通信処理リソースの奪いあいが起こる。

これらは、ネットワークの帯域や Round Trip Time 等のネットワークのパラメータには無関係なフローコントロールが必要である。ある程度の帯域とレスポンスが保証されれば、並列プログラムはユーザが流量制御など行わなくても動作する。しかし、先程の並列検証系のように、多くのホスト間で互いに自由な通信が行われ、しかも通信の内容が動的に変化する場合、アプリケーション中のフローコントロールによって効率化できる部分は大きくなると考えれる。したがって、このような通信の流量制御をユーザプログラムコントロールからコントロールできればより効率的な負荷分散が可能になると考えられる。つまり、通信データ

の性質により求められる通信品質(QoS)は異なり、従って流量制御に必要な情報は、ネットワークからえられるパラメータだけでなくユーザプログラムにも存在するといえる。

3.2 ユーザレベルフローコントロールAPIの実装

UDPライブラリがライブラリレベルで自動で行うフローコントロールは、送信側がどんなタイミングで送信しても受信側のオーバランによるUDPデータグラムパケットのロスを起こさない、ということを達成するために行われる。UDPライブラリはアドレス毎に受信バッファを持っているが、UDPソケットバッファから受信バッファへのコピーが間にあわないことがことがオーバランの原因である。これを解決するには、UDPソケットの受信バッファ以上のデータグラムパケットの送信しなければよい。

UDPライブラリは、getsockopt()によって得たUDPソケットバッファの値を越えない値をUDPライブラリに登録されているアドレス毎にウインドウサイズを割り当て、通信相手先に通知する。

TCPとは違い、常に受け入れ可能なウィンドウサイズを送信側に通知せず、コネクション確立時やユーザプログラムのウィンドウサイズの変更要求によって変更されたときのみにアドレスに割り当てられた受信バッファの通知を行う。アプリケーションは通信相手先に割り当てられた受信バッファの容量を知ることができ、オーバフローしないタイミングで送信バッファををフラッシュすることが望まれる。

これを実装すれば受信バッファのオーバフローを防ぐ手段をアプリケーションは手にいれるが、多くのアドレスがUDPライブラリに登録される超分散的な環境では各アドレス毎に割り振られるウィンドウサイズは非常に小さくなってしまい、through-putが落ちる。しかし、全ての通信相手先に同様なthrough-putが必要とされることは少ない。

そこで、ユーザレベルからアドレス毎のウィンドウサイズの調節を行うユーザレベルフローコントロールを定義し、ユーザプログラムがプログラムの状態により通信相手先毎にフローコントロールを行う。

フローコントロールは通信相手先毎スループット優先とレスポンス優先の状態もたせることでおこなう。フローコントロールにもっと細かなパラメータを持たせることも可能だが、ライブラリレベルのフローコントロールとのすり合わせのため今回は2種類のみのAPIとした。

フローコントロールのprolog用のAPIを表2に示す。表中の+StreamはUDPライブラリが提供する通信終端である。フローコントロールはこの通信終端と通信相手先のIDのセットを指定して行う。アプリケーションはgetdestbufで送信先の受信バッファの総容量を知ることができ、その値をもとに送信バッファをフラッシュすることが期待される。

フローコントロールのAPIによってthrough-put優先にしたあとの通信は、相手先にバッファ、ウィンドウサイズを大きく確保するコントロールメッセージを送信し、このコントロールメッセージを受け取った通信相手先は、最大限の受信バッファを確保し、その確保したウィンドウサイズをコントロールメッセージとして返す。その値をもとに送信側はウィンドウサイズを変更する。

responseを優先した場合は、通信相手先にウィンドウサイズを小さくし、Acknowledgeの頻度をあげるようコントロールメッセージを送信する。

4 ユーザレベルのコネクションの管理

分散並列プログラミングでユーザプログラムレベルでのコネクション管理機構があればさらに効率よい負荷分散が行える可能性は述べた。しかし、コネクション管理をすべてユーザプログラムが行うとプログラミングが非常に繁雑になる。通常のコネクション管理はライブラリレベルで自動で行うほうがよい。従って、コネクション管理にはユーザレベルでプログラムが動的に行うコネクション管理と、ライブラリがAcknowledgeのtimeoutを検知し自動で行うライブラリレベルのコネクション管理が考えられる。この2つのコネクション管理機構を矛盾なく同時に実装するために図4のような状態遷移モデルを考えた。図中のExplicit Disconnect

ID の送信先の受信バッファサイズを得る	`datagram_getdestbuf(+Stream,ID,-Size)`
通信を throughput 優先にする	`datagram_through(+Stream,ID)`
通信をを response 優先にする	`datagram_rapid(+Stream,ID)`

表 2: Prolog 用フローコントロール API

とは、ユーザプログラムによる明示的な中断であり、Implicit Disconnect とは、ライブラリによる自動暗黙的な通信の中断である。ユーザプログラムによる明示的な通信の中断は、ユーザプログラムよって再開されるまで、回復しない。

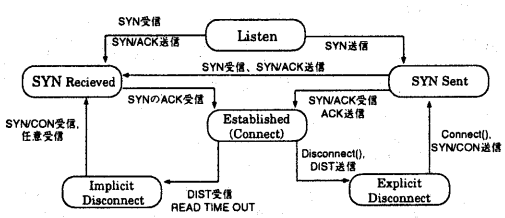

図 5: コネクションの状態遷移モデル

4.1 コネクション管理 API の実装

Explicit な disconnect の場合、通信相手先に Disconnect のコントロールパケットを送り、コントロールパケットを受け取った方は、Connect のコントロールパケット受け取るまで受信バッファを送信側に通知しない。

Disconnect している通信相手先への write は常に失敗する。Disconnect する以前に成功したが Acknowledge が返らないうちに disconnect した write data はバッファにためこまれ、reconnect したときに send される。ユーザプログラムから、アドレス毎のコネクションの状態は参照することができる。

	UDP Library	UDP	TCP
ヘッダサイズ	20Byte	8Byte	24Byte
信頼性	ある	ない	ある
流量制御	ユーザレベル	ない	カーネルが行う
コネクション	ユーザレベルで管理	ない	カーネルが管理

表 4: UDP Library と TCP の比較

4.2 実装

5 考察と今後の課題

5.1 比較

今回ユーザレベルフローコントロールとコネクション管理の API を実装した UDP ライブラリは、以下のような特徴をもっている。

- ユーザプログラムが動的にフローを制御できるため、ユーザプログラムはデータの性質や負荷に応じた通信を行える。
- ユーザレベルでのコネクション管理が行える。
- 完全結合型の通信を UDP ソケット一つで提供する。

今回実装した UDP ライブラリと UDP、TCP との比較を表 4 に示す。IP の TOS フィールドを用いたでの QoS の確保 [4] は研究されている。また、IPV6 ではセグメントヘッダにパケットの優先順位とフローラベルのフィールドがあり、アプリケー

通信を一時中断する	datagram_disconnect([+Stream],+Id).
通信を再開する	datagram_reconnect([+Stream],+ID).
状態を参照する	datagram_state([+Stream],+ID).

表 3: Prolog 用通信の一時中断と再開の API

ション毎に適当な値がフロー情報としてが設定できる [5]。しかし、これらの機能を利用するためにはルータによるサポートが必要であり、現在広くつかわれてはいない。

UDP ライブラリを用いたユーザレベルフローコントロールは、ルータのサポートを必要としない。アプリケーション間のフローコントロールに関する合意をライブラリレベルで自動化する。

5.2 課題

現在、sliding window は実装されているが、複数パケットに対する Acknowlege のの一括処理は実装されていない。これはパケット数を減らし帯域を有効に使うことになるので実装する必要がある。

また、全2重通信にはパケットヘッダに Acknowledge フィールドを加えた方が、データパケットと Acknowledge を一つのパケット送れるため、パケット数が減り効率的である。しかし、この実装によりパケットヘッダによるオーバヘッドは増える。アプリケーションのもとめる通信品質に応じてパケットヘッダを動的に返る仕組みが必要かもしれない。

今回ユーザレベルのフローコントロールは、今回 response 優先と throughput 優先の 2 種類のみしか選択できないようにしている。フローコントロールにはもっと細かなパラメータを指定することも可能であるし、どこまでアプリケーションが行いどこからをライブラリが行うかを検討する必要がある。現在、再送のタイミングや送信パケットのフラッシュはユーザプログラムが行うことが期待されるが、この操作は、ユーザプログラムの自由度が高い半面、繁雑なプログラムになるといえる。これらの処理はライブラリが自動で行ってもよい。しかし、その場合ユーザプログラムの自由度は下がる。

アプリケーションが共有にもつ通信への要望を取り出しそれをどのようにフローコントロールの API として実現していくか、分散環境でのユーザプログラムにおける適切なフローコントロール API を検討を今後の課題としてあげておく。

6 まとめ

本稿では、並列プログラミングにおけるユーザレベルフローコントロールを提案し、UDP ライブラリにユーザレベルのコネクション管理とフローコントロール用の API を付加した。

参考文献

[1] 河野真治, 神里健司: UDP を使った分散計算機環境とその応用, 情報処理学会システムソフトウェアとオペレーティングシステム研究会 (OS) , May, 2000.

[2] 河野真治, 池村正之: 状態集合の分割による時相論理の検証の並列化, 電気学会・電子情報通信学会合同講演会, December, 1998.

[3] 揚 挺, 神里健司, 謝花蔵, 河野真治: Datagram を使った分散ライブラリの評価, 沖縄ワークショップ 2000, September.

[4] The Design and Implementation of the ALTQ Traffic Management System Kenjiro Cho ,Keio University,Graduate School of Media and Governance,doctoral dissertation, in January 2001

[5] Philip Miller 著、苅田幸雄監訳 "マスタリング TCP/IP 応用編" オーム社, 1998.

[6] W.Richard Stevens 著、篠田陽一訳 "UNIX ネットワークプログラミング第 2 版" ピアソン・エデュケーション,1999.