トランスポート層を考慮した スナップショット・アルゴリズムの考察

Consideration on Snapshot Algorithm including Transport Layer

屋比久 友秀 † 河野 真治 †‡ 山城 潤 †

Tomohide YABIKU† Shinji KONO†‡ Jun YAMASHIRO†

† 琉球大学工学部情報工学科

†Dept. of Information Engineering, University of the Ryukyus

‡ 科学技術振興事業団さきがけ研究 21(機能と構成)

‡PRESTO, Japan Science and Technology Corporation

yabiku@cr.ie.u-ryukyu.ac.jp

本稿では、フロー制御や輻輳制御をユーザレベルから行う UDP ベースの通信ライブラリ”Suci” を利用したスナップショット・アルゴリズムについて考察する。様々な スナップショット・アルゴリズムが提案されているが、トランスポート層での通信の信頼性が保証されている事が前提であった。我々は スナップショットアルゴリズムを ユーザレベルでフロー制御する事で不用なトラフィックを抑える可能性がある事を示す。

1 はじめに

並列分散環境において PC クラスタなどの普及により、50 台以上の分散環境も比較的容易に利用する事ができるようになってきた。PC クラスタのような比較的密結合な分散環境では、MPI や PVM 等の並列ライブラリが利用されている。これらの多くはシステムレベルでフロー制御を行うタイプである。OSI 参照モデルでは、フロー制御はトランスポート層の役割であり、システムレベルに隠蔽されているので、アプリケーション層から制御する事はできない。最も広く使われているトランスポートプロトコルには、TCP がある。プログラマは TCP を使えば、面倒なフロー制御を意識する事なく、アプリケーションを作る事ができるようになる。その為、TCP は最も普及したトランスポートプロトコルとなった。しかし、インターネット等のような、通信の公平性を必要とするような環境では、そのようなプロトコルが有効ではあるが、PC クラスタなど比較的密な分散環境では無視できない幾つかの問題がある。

例えば、並列タブロー検証系 [1] がある。並列タブロー検証系は、タブロー法による時相論理式の展開を並列化するアプリケーションである。このアプリケーションでは比較的応答性が重要視されスループットはそれほど必要ない通信と比較的大きなメッセージがやりとりされスループットが重要になり応答性は重要ではない通信の 2 種類が必要になる。ところが、システムレベルのフロー制御では、通信の公平性を保つために同じように通信を行ってしまう。

また、並列タブロー検証系では、同時に多数のノードと通信する必要がある。通常 TCP の最大ウィンドウサイズは、TCP ヘッダのウィンドウサイズフィールドが $16bit$ であるため、$2^6 = 64KBytes$ である [2]。1000 台以上の完全結合型通信を TCP で実現しようとした場合、$64KBytes \times 1000 \times 2 =$ 約 $128MBytes$ もの送受信バッファがカーネル内部に生じる事になる。

このような問題が TCP にあるためシステムレベルのフロー制御では並列タブロー検証系のようなアプリケーションに対しては不充分である。我々はこれらの問題を解決するために、ユーザレベルにトランスポート層の仕組を実装し、アプリケーションプログラムからフロー制御を行えばよいと考え、ユーザレベルのフロー制御を備えた通信ライブラリ”Suci” を提案した [3][4]。本稿では、Suci(Simple User Communication Interface) を使った応用例として スナップショット・アルゴリズム [5] を取り上げ、ユーザレベルフロー制御を行う場合と TCP ベースの通信ライブラリとを比較し、ユーザレベルでフロー制御を行う場合の優位性を示す。

2 Suci の概要

Suci の特徴であるユーザレベルのフロー制御について述べ、Suci のプログラミングスタイルについて述べる。

2.1 ユーザレベルフロー制御

データ通信において、フロー制御は必須の機能である。たとえば、処理能力が高いコンピュータから処理能力が低いコンピュータへデータ転送を行う場合、送信能力限界のスループットで転送すると、受信側で受信しきれずに、パケットを取りこぼす場合がある。このようなことを防ぐために、送信側の送信データ量を調節することをフロー制御と呼ぶ。

フロー制御には、2 つの目的がある。１つは、先程述べたように、受信側のバッファオーバーフローを防ぐことである。もう１つは、ネットワーク機器の能力を超えたパケットの送信を防ぎ、ネットワークの混雑を防ぐ事である。ネットワークが混雑しているときに大量の送出すると輻輳が発生し、パケットが失われる。輻輳が起きないようにパケットの送信量を調節し、輻輳が発生した場合にパケットの送信量を減らして輻輳状態から回復させることも、フロー制御の役割である。この機能は、とくに１つ目のフロー制御とは区別して輻輳制御と呼ばれる。

Suci の基本的なアイディアとして、フロー制御をシステムレベルで行うのではなく、ユーザレベルで行う事で、分散アルゴリズムにフロー制御を含める事ができると考えた。図 1 にシステムレベルとユーザレベルのフロー制御の概念図を示す。

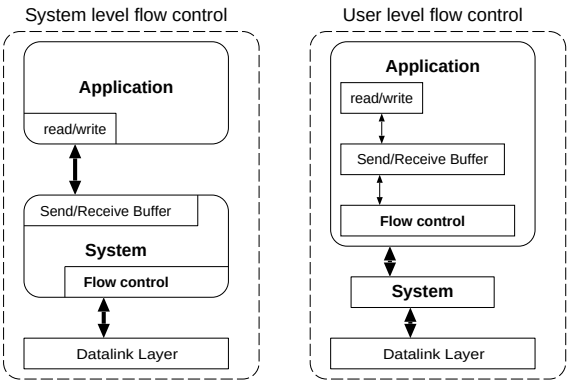

図 1: ユーザレベルフロー制御の概念図

システムレベルのフロー制御は OSI 参照モデルのトランスポート層で行っており、アプリケーション側からは隠蔽されていて制御できない仕組みになっている。これはプログラマが面倒なフロー制御を意識する事なくプログラミングができるが、より柔軟なフロー制御を行うような分散環境では問題となる。

図 1 の User level flow control の中で Suci が提供している部分は、アプリケーションレベルで使われている read/write の API、Send/Receive のバッファ管理、それと Flow Control 用の API を提供している。このような仕組みを持つ通信ライブラリを利用する事で、より柔軟なデータ通信を行う事ができる。

2.2 Suci のプログラミングスタイル

表 1 に Suci が提供してる主な API を示す。

図 2 に Suci を用いた分散プログラムの典型を示す。図 2 の datagram_destination() によって送信先を指定し、datagram_write によって出力を行う。また、必要ならば再送の処理を行う。再送処理は、datagram_queue_length によって再送 queue の状況をチェックし、datagram_ready() によって確認応答の処理を行いつつ datagram_ retransmission() を呼び出す。このとき、再送 queue の状態によって輻輳やバッファオーバフローをアプリケーションが検知し、再送の量を調節したり、再送タイミングを送らせることによってフロー制御を行う。

データ受信時には、まず datagram_ready() によって read ブロックを組み立て、ブロックが完成していれば、datagram_read() によってアプリケーションにデータを渡す。

このようにして Suci は、送受信タイミング、確認応答の処理のタイミングや再送処理のタイミングをアプリケーションが直接操作することにより、フロー制御を行うことができる。

また、データの送受信は非同期で行われるためメッセージの到着を待ちつつ、他の処理を実行できるようなプログラミングスタイルを採用している。

3 スナップショット・アルゴリズムへの応用

2 節で述べた Suci をスナップショット・アルゴリズムに適応した場合について述べる。スナップショット・アルゴリズムは、システム全体の状況を計算し記録するアルゴリズムである。スナップショット・アルゴリズムは分散環境においてアルゴリズムの終了、デッドロック、トークンの紛失などの定常特性の検出および判定に用いられている [5]。また、故障から

表 1: Suci の主な API

Stream Open	`datagram(addrdb,myaddr,myport,myid)`
Choice of transmitting point	`datagram_destination(distid,sock)`
Transmission	`datagram_write(sock,buf,len)`
Reception	`datagram_read(sock,buf,len)`
Checking the packet	`datagram_ready(sock,sec,msec)`
Waiting	`datagram_wait(sock,sec,msec,size,id)`
Checking the transmission queue	`datagram_queue_length(sock,id)`
Retransmission	`datagram_retransmission(sock,destid)`

```
while(1){
  // 送信先を指定
  datagram_destination(ID, sock);
  datagram_write(sock, buff, size);
  // Acknowledge がとれるまで、メッセージを再送信
  while(datagram_queue_length(sock,ID)) {
    datagram_ready(sock, 0, usec);
    datagram_retransmission(sock, ID);
  }

  if((datagram_ready(sock, sec, usec) > 0){
    // read block があれば受信データを読込む
    datagram_read(sock, recvbuf, size);
  }
}
```

図 2: Suci を使った典型的なメッセージ転送プログラム例

復帰するためのチェックポイント、ロールバックアルゴリズム [6][7] や分散プログラムのデバック [7][8] にも利用されている。ネットワーク形状の無矛盾性を考慮した スナップショット・アルゴリズム [9] も提案されている。

3.1 スナップショット・アルゴリズム

K. M. Chandy と L. Lamport の スナップショット・アルゴリズムについて簡単に説明する。Chandy らの スナップショット・アルゴリズムは, 標識 (*marker*) メッセージを利用して、プロセスやチャンネルの状態を記録するタイミングをプロセスに知らせる。起動プロセスを除く全てのプロセスはアルゴリズムの実行を開始すると、現在の状態を記録し、全ての隣接プロセスに対して標識を送信する。隣接するプロセス p_i とプロセス p_j を接続するチャンネルについては、p_i から p_j へ伝送中のメッセージは、p_i が状態を記録する前に送信され, p_j が状態を記録した後に受信される。全てのチャンネルは FIFO であるので、p_i が標識を送信する前に送信されたメッセージは必ず p_j において標識よりも先に受信される図 3。

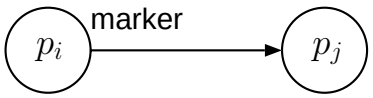

図 3: 標識の役割

従って、p_j は p_i から p_j へ伝送中のメッセージを記録することができる。このように, Chandy らのアルゴリズムでは、伝送中のメッセージを必ずその受信側が記録している。以下に標識の役割についてまとめる。

- 標識は各プロセスに対してできるだけ早く、スナップショット・アルゴリズムの実行が開始された事を知らせる。
- 標識は各プロセスに対して、伝送中のメッセージを記録するタイミングを知らせる。

我々は、この標識を送受信する場合に、ユーザレベルフロー制御 を使う事で幾つかメリットが考えられる。それについて以下で考察する。

3.2 スナップショット・アルゴリズム上のフロー制御

3.2 節でスナップショット・アルゴリズムは標識を使って各プロセスの状態を記録するタイミングをとっている事を述べたが、この標識を送受信する時に、受信側で標識を受け取れなかった場合、システムレベルのフロー制御では受信側が受信するまで標識を再送信する。しかし、Suci を利用したユーザレベルのフロー制御では、より新しいプロセスの状態を記録し、

その後で標識を送信する事が可能になる。すなわち、より最新の スナップショット を取る事ができる。

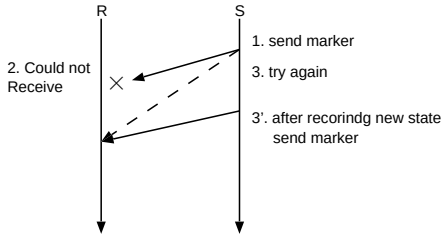

図 4: スナップショットの再送

図 4 では、はじめに S(Sender) から R(Receiver) に標識を送信するが、R が何らかの原因で標識を受信できない場合、R の計算がより進んだ状態を記録した後で、標識を送信したい場合がある。このような事をシステムレベルのフロー制御で実現することは難しいが、Suci を用いたユーザレベルのフロー制御では、それが可能になる。

3.3 リング形状 スナップショットにおける Acknowledge の削減

リング形状の スナップショット では、標識をリングに沿って送受信を行う。図 5 の点線が標識の送受信を表している。この場合、個々のノード間で Ack や Nack の送受信を行いメッセージ到達を保証している。Suci の場合は、図 5 の太線で一方向に標識を送信し、さらに Acknowledge の制御などもユーザレベルから行う事が可能である為、個々のノード間で Ack の送信を行わない事もできる。この場合、メッセージの到達保証は、スナップショット・アルゴリズムを起動したノードに標識が戻ってくる事で保証される。即ち、Ack,Nack を制御する事で余分なメッセージを抑える事ができる。

4 まとめと今後の課題

本稿では、ユーザレベルフロー制御の有効性を示し、その例としてスナップショット・アルゴリズムを取り上げた。スナップショット・アルゴリズムでは、ユーザレベルからフロー制御を行う事でシステムレベルフロー制御では不可能だった スナップショット・アルゴリズムのフロー制御が可能になる。さらに、リング形状の スナップショット では Acknowledge を使わない通信も可能になる為、通信量を抑える事ができる事を示した。

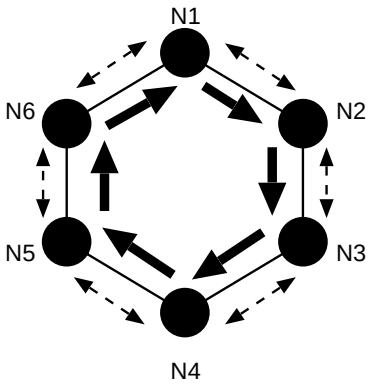

図 5: リング形状 スナップショット

今後の課題としては、Suci の上位層に スナップショット・アルゴリズムを実装し、他の通信ライブラリと比較、Suci ライブラリの有効性を検証する必要がある。

参考文献

[1] 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と並列化, 電気学会・電子情報通信学会合同後援会,1998.

[2] Postel, J. (ed.): Transmission Control Protocol, DARPA Internet Program Protocol Specification, RFC 793,1981.

[3] 河野 真治, 神里 健司: UDP を使った分散環境とその応用, 日本ソフトウェア科学会大会論文集,2000.

[4] 屋比久 友秀, 河野 真治: 並列分散ライブラリ Suci の実装と評価, 情報処理学会第 90 回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,2002.

[5] K. Chandy and L. Lamport: Distributed snapshots: Determining global states of distributed systems. *ACM Transaction on Computer Systems, Vol. 3, No. 1, pp63-75,1985.*

[6] 青柳, 真鍋: 分散デバッガのためのチェックポイント・ロールバックアルゴリズム, 日本ソフトウェア科学会 ソフトウェア研究会 (関西) 資料, SW-92-10-4,1992.

[7] 増澤, 都倉: 分散デバッガのための Casual Distributed Breakpoint を求めるアルゴリズム, 電子情報通信学会秋期大会,SD-1-8,1992.

[8] 守屋 宣, 櫟 粛之: インターネットエージェントのための動的スナップショットアルゴリズム, 電子情報通信学会 技術研究報告, Vol. 101, No.44, pp17-24,2001.

[9] 佐藤 泰郎, 井上 美智子, 増澤 利光, 藤原 秀雄: 分散移動システムにおけるスナップショット・アルゴリズム, 情報処理学会 研究報告 アルゴリズム, No.47-4,1995.