1H−1 UNIX上のトランザクション処理のための マルチスレッド環境

白木原敏雄、金井達徳
(株)東芝

1 はじめに

現在、我々は標準的なUNIXワークステーション上で、リモートプロシージャコール(RPC)をベースとするサーバ・クライアント型の分散処理プログラムに対して、トランザクション処理を用いて高信頼性を実現する環境の研究を行なっている[1]。トランザクション処理では短いジョブが多数発生するため、サーバはスループットの高いサービスを提供しなければならない。そのためには、マルチスレッドRPCサーバなどのように、同時に複数の処理を行なう機構が必要である。

そこで、我々は高スループットのサーバを実現するために、RPCをマルチスレッドで処理する環境(マルチスレッド環境)をSunOS上に実現した。SunOSでは、SunLWPライブラリというユーザレベルのスレッド機構が提供されている。

以下ではユーザレベルのスレッド機構を用いて、マルチスレッドでRPCの処理を行なう環境の実現方法について述べる。

2 マルチスレッド環境の構成

一般に、コネクション指向のTCPプロトコルのソケット[2]を用いたサーバ・クライアント型の通信において、クライアントおよびサーバは以下のような動作を行なう。

- クライアント

  クライアントはサーバのサービスを受けるために、サーバのコネクト待ちのソケットの情報を持っており、そのソケットに対してconnectシステムコールにより通信路を確立し、writeシステムコールによりメッセージを転送し、さらに、readシステムコールによりサーバからの応答を受けとる。一度コネクションが確立すると、それ以降のメッセージの転送はwrite/readシステムコールのみで行なう。

- サーバ

  acceptシステムコールにより、クライアントからのコネクト要求を受け、通信路が確立される。acceptシステムコールでは、新たにソケットが生成され、メッセージの転送はその新しいソケットで行なう。readシステムコールによりメッセージを受け、writeシステムコールにより応答を返す。

このような方法において、マルチスレッドサーバとしてRPCを処理するには、同時に複数の通信路を管理しなければならない。また、ユーザレベルのスレッド機構では、1つのスレッドがreadシステムコールまたはacceptシステムコールでブロックされた場合、プロセス全体がブロックされ、他のスレッドが動作できないという問題点がある。

我々のマルチスレッド環境は、上記の問題を解決するための通信管理部とRPCコンパイラの2つの部分から構成される。通信管理部は、複数のソケットを管理し、それらの中から、アクティブなソケットを調べ、アクティブなソケットのみをメッセージ処理を行なうスレッドに提供する。ここで、アクティブなソケットとはクライアントからのRPC要求により読みだし可能か、または通信路確立要求によりコネクト可能なソケットのことである。また、RPCコンパイラはユーザが定義したRPCインタフェースから、通信管理部の提供するインタフェースを使用するRPCスタブを生成することにより、RPCベースのプログラミング機能を提供している。

3 通信管理部

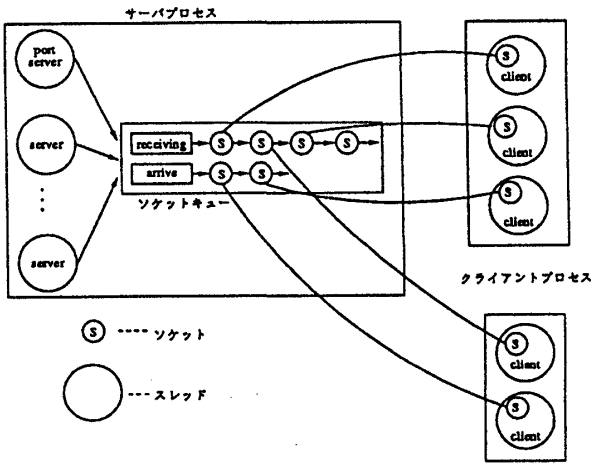

図1: 通信管理部の概要

通信管理部の概要を図1に示す。通信管理部は、ソケットがアクティブかを調べるスレッドport_serverとソケットのキューとして、receivingキューとarriveキューの2つを備えている。receivingキューには、コネクト待ちのソケットとメッセージ待ちのソケットが、またarriveキューにはアクティブなソケットがつながっている。キューのノードにはソケットの情報と、そのソケットがコネクト待ちかメッセージ待ちかを区別する情報が記録されている。

port_serverはアクティブなソケットのみをarriveキューにつなげておくために設けられている。そのた

Multi-Thread Environment for Transaction Processing under UNIX.
Toshio SHIRAKIHARA and Tatsunori KANAI.
Toshiba Corporation.

めに、receving キューにつながっているソケットがアクティブかを, select システムコールを使って調べ、アクティブなものを receiving キューからはずし、arrive キューにつなぐという動作を行なっている。

各サーバスレッドは arrive キューを調べ、ソケットがつながっていれば、それを arrive キューからはずすことによりソケットを獲得し、そのソケットに対して、クライアントからの要求に応じて以下の処理を行なう。

- コネクト要求

  ソケットがコネクト待ちの場合には、クライアントからのコネクト要求であるため、accept システムコールを実行し通信路を確立する。その際、メッセージの転送を行なうソケットが生成されるため、それを receiving キューにつなぐ。

- メッセージ

  ソケットがメッセージ待ちの場合には、クライアントがメッセージを転送してきた場合であるため、read システムコールを実行し、読み込んだメッセージに対する処理を行ない、クライアントへの応答を返す。

これらの処理が終了した後は、再び獲得したソケットへのメッセージを受けるために、receiving キューにつなぐことによりソケットを解放する。

以上のような機構により、サーバスレッドは常にアクティブなソケットに対して処理を行なうため、プロセス全体がブロックされることはない。

4 RPC コンパイラ

RPC コンパイラは、ユーザが定義したインタフェースを基に、サーバ用とクライアント用の RPC スタブを生成する。これにより、ユーザは通信管理部のインタフェースを意識することなく、プロシージャコールと同様のプログラミングにより、RPC ベースプログラミングを行なうことができる。

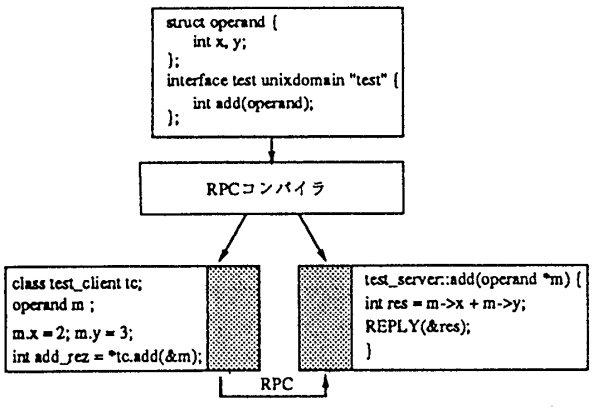

図 2: PRC コンパイラの概要

RPC コンパイラの概要を図2に示す。ユーザは RPC で交換されるメッセージのタイプと RPC のサービス名およびネットワーク情報を定義する。このインタフェース定義を RPC コンパイラにかけると、クライアント用のスタブとサーバ用のスタブは C++のクラスとして生成される。クライアントプログラム中での RPC は、そのクラスのオブジェクトのメンバ関数を実行することにより行う。サーバプログラム中でのサービスの処理内容は、そのクラスのメンバ関数として記述する。

RPC スタブでは、通信管理部のインタフェース上に RPC を実現するために、以下の機能を提供している。

- XDR によるデータ変換

  XDR は SunOS でサポートされているライブラリで、マシンに依存しない形式で任意のデータ構造を記述するためのものである。スタブによる通信は、データを XDR で変換して行なうため、異機種間での通信も可能になる。

- ソケット資源の管理機構 (クライアントスタブ)

  クライアントスタブでは、使用していないソケットのプールを管理し、通信の開始時には、そのプールからソケットを得る。これにより、クライアント側でのソケットの有効利用をはかっている。

- 通信エラー処理 (サーバスタブ)

  プロセス間の通信は TCP プロトコルで行なっているおり、クライアントプロセスが何らかの原因でアボートした場合、コネクションが切れるため、サーバはそのアボートを検出できる。サーバクラスのオブジェクトを生成する時に、エラー処理を行なうべき関数のアドレスを指定しておくと、クライアントプロセスがアボートした場合、その関数が呼ばれ、その中でクライアントに関する資源の解放等を行なうことができる。

5 おわりに

本論文で述べたマルチスレッド環境により、ユーザレベルのスレッド機構で RPC の処理を行なうマルチスレッド RPC サーバが実現できた。また、RPC コンパイラにより、RPC ベースプログラミングが容易になり、生産性が向上した。本マルチスレッド環境は、研究中の高信頼分散処理環境 [1] の開発のベースとして使用している。

一方、本マルチスレッド環境では信頼性を重視して TCP プロトコルを採用したが、そのことに起因した以下の問題点がある。(1) プロセスおよびスレッドの数が増加した場合、多数のソケットが必要になる。ソケットの数は OS で制限されており、それ以上のソケットが必要になる数のサーバは動作しない。(2) TCP プロトコルは UDP プロトコルに比べて速度が遅い。

今後は、UDP プロトコルへの移行を実現し、より高速なマルチスレッド環境の構築を行なっていく予定である。

参考文献

[1] 金井, 白木原: 階層トランザクション機構による UNIX 上の高信頼分散処理環境, 情報処理学会第 84 回計算機アーキテクチャ研究会 92-8, 1992.

[2] W.R.Stevens:UNIX Network Programming, Prentice Hall, 1990.