RPC 拡張開発ガイド

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 816–3477–10
2002 年 3 月

原典: *RPC Extensions Developer's Guide*

Part No: 816-3576-10

Revision A

031119@6671

目次

はじめに

『*RPC* 拡張開発ガイド』は、Solaris™ 8 update 7 で Sun™ の RPC (リモートプロシージャコール) ライブラリに追加された機能について説明します。

対象読者

このマニュアルは、Solaris 開発環境でリモートプロシージャコールを使用するアプリケーション開発者を対象としており、以下に関する詳細情報が含まれます。

- 1 方向性メッセージング
- 非-ブロッキング入出力
- クライアント接続クロージャーコールバック
- ユーザーファイル記述子コールバック

このマニュアルでは、プログラミングの基本、C プログラミング言語での作業、および UNIX オペレーティングシステムでの作業に習熟している読者を想定しています。ネットワークプログラミングの経験は、あれば役立ちますが、このマニュアルの使用においては必須ではありません。

分散型サービスについての詳細は、『*ONC+* 開発ガイド』を参照してください。

内容の紹介

第 1 章 は、Solaris 8 update 7 で Sun の RPC ライブラリに追加された機能について説明します。

関連マニュアル

RPC を使用する開発の詳細、および分散型システムの概要については、『*ONC+* 開発ガイド』を参照してください。

Sun のマニュアルの注文方法

専門書を扱うインターネットの書店 Fatbrain.com から、米国 Sun Microsystems™, Inc. (以降、Sun とします) のマニュアルをご注文いただけます。

マニュアルのリストと注文方法については、http://www1.fatbrain.com/documentation/sun の Sun Documentation Center をご覧ください。

Sun のオンラインマニュアル

docs.sun.com™ では、Sun が提供しているオンラインマニュアルを参照することができます。マニュアルのタイトルや特定の主題などをキーワードとして、検索を行うこともできます。URL は、`http://docs.sun.com` です。

表記上の規則

このマニュアルでは、次のような字体や記号を特別な意味を持つものとして使用します。

表 **P–1** 表記上の規則

字体または記号	意味	例
AaBbCc123	コマンド名、ファイル名、ディレクトリ名、画面上のコンピュータ出力、コード例を示します。	.login ファイルを編集します。 ls -a を使用してすべてのファイルを表示します。 system%
AaBbCc123	ユーザーが入力する文字を、画面上のコンピュータ出力と区別して示します。	system% **su** password:
AaBbCc123	変数を示します。実際に使用する特定の名前または値で置き換えます。	ファイルを削除するには、rm *filename* と入力します。
『 』	参照する書名を示します。	『コードマネージャ・ユーザーズガイド』を参照してください。
「 」	参照する章、節、ボタンやメニュー名、強調する単語を示します。	第 5 章「衝突の回避」を参照してください。 この操作ができるのは、「スーパーユーザー」だけです。
\	枠で囲まれたコード例で、テキストがページ行幅を超える場合に、継続を示します。	sun% **grep '^#define ** **XV_VERSION_STRING'**

ただし AnswerBook2™ では、ユーザーが入力する文字と画面上のコンピュータ出力は区別して表示されません。

コード例は次のように表示されます。

- C シェル

  machine_name% **command y|n** [*filename*]

- C シェルのスーパーユーザー

  machine_name# **command y|n** [*filename*]

- Bourne シェルおよび Korn シェル

  $ **command y|n** [*filename*]

- Bourne シェルおよび Korn シェルのスーパーユーザー

  # **command y|n** [*filename*]

[ ] は省略可能な項目を示します。上記の例は、*filename* は省略してもよいことを示しています。

| は区切り文字 (セパレータ) です。この文字で分割されている引数のうち 1 つだけを指定します。

キーボードのキー名は英文で、頭文字を大文字で示します (例: `Shift` キーを押します)。ただし、キーボードによっては `Enter` キーが `Return` キーの動作をします。

ダッシュ (-) は 2 つのキーを同時に押すことを示します。たとえば、`Ctrl-D` は `Control` キーを押したまま `D` キーを押すことを意味します。

一般規則

- このマニュアルでは、英語環境での画面イメージを使っています。このため、実際に日本語環境で表示される画面イメージとこのマニュアルで使っている画面イメージが異なる場合があります。本文中で画面イメージを説明する場合には、日本語のメニュー、ボタン名などの項目名と英語の項目名が、適宜併記されています。
- このマニュアルでは、「IA」という用語は、Intel 32 ビットのプロセッサアーキテクチャを意味します。これには、Pentium、Pentium Pro、Pentium II、Pentium II Xeon、Celeron、Pentium III、Pentium III Xeon の各プロセッサ、および AMD、Cyrix が提供する互換マイクロプロセッサチップが含まれます。

第 1 章

Sun RPC ライブラリの拡張

Sun RPC ライブラリに新しい機能が追加され、標準の Solaris 9 製品に組み込まれます。

新規および変更されたマニュアルページで、Sun RPC ライブラリに追加された機能についての説明を参照することができます。

Sun RPC ライブラリへの追加は以下のとおりです :

新しい機能

Sun RPC ライブラリに追加された新規機能とそれらの利点は、次のとおりです:

- 1 方向性メッセージング - クライアントスレッドが処理継続前の待ち時間を減らすことができます。
- 非-ブロッキング入出力 - クライアントがリクエストをブロックされることなく送信できるようにします。
- クライアント接続クロージャーコールバック - サーバーがクライアントの接続切断を検知し、適切な対処を行えるようにします。
- ユーザーファイル記述子コールバック - 非-RPC 記述子を処理するために RPC サーバーを拡張します。

Sun RPC ライブラリの拡張

Sun RPC ライブラリの以前のバージョンでは、ほとんどのリクエストは 2 方向性メッセージングにより送られていました。2 方向性メッセージングでは、クライアントスレッドは処理を進める前にサーバーからの応答を得るまで待つ必要があります。クライアントスレッドが一定の時間内にサーバーからの応答を受け取らなかった場合、タイムアウトになります。このクライアントスレッドは、1 番目のリクエストが実行されるかタイムアウトになるまで、2 番目のリクエストを送ることができません。このメッセージングのメソッドを図 1–1 に図示します。

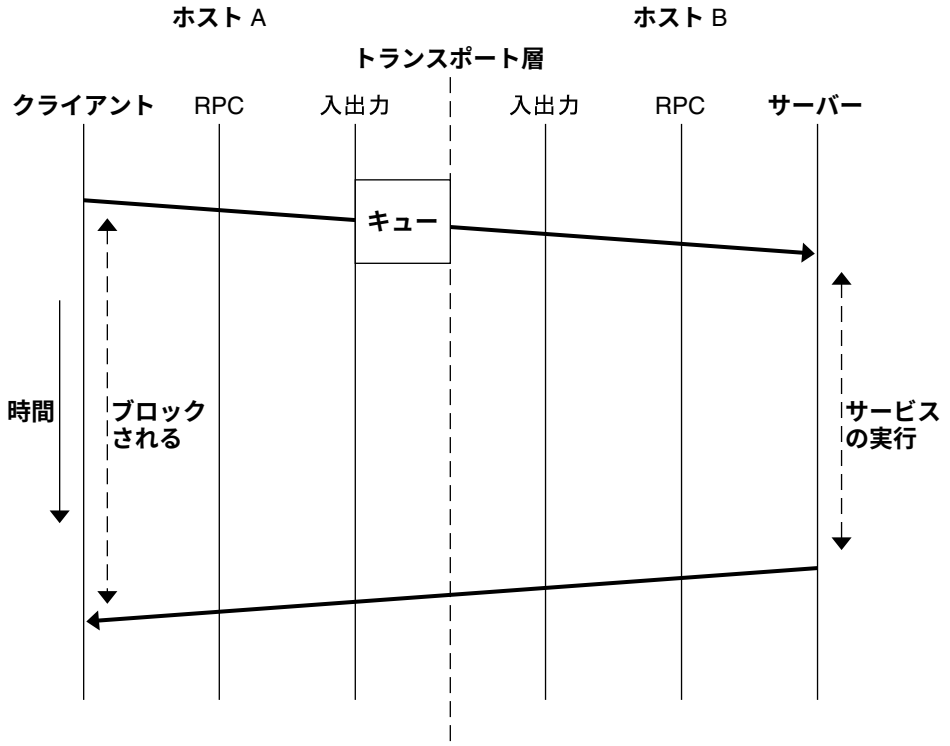

図 1–1 2 方向性メッセージング

注 – Sun RPC ライブラリの以前のバージョンには、バッチングと呼ばれるもう 1 つのメッセージング方法があります。この方法では、グループ中のリクエストが同時に処理可能の間、クライアントのリクエストはキューに保持されます。これは 1 方向性メッセージングの形態です。詳細は、『*ONC+* 開発ガイド』の「RPC プログラマインタフェース」を参照してください。

Sun RPC ライブラリの新しい機能は、ライブラリのパフォーマンスを次のように拡張します。

1. 1 方向性メッセージング - クライアントがサーバーにリクエストを送信する際、クライアントはサーバーからの応答を待つことなく、トランスポート層がその リクエストを受け入れる場合は処理を進めることができます。2 方向性メッセージングではなく 1 方向性メッセージングでメッセージを送信すると、処理時間を得ることができます。図 1–2 に、1 方向性メッセージングのメソッドを図示します。

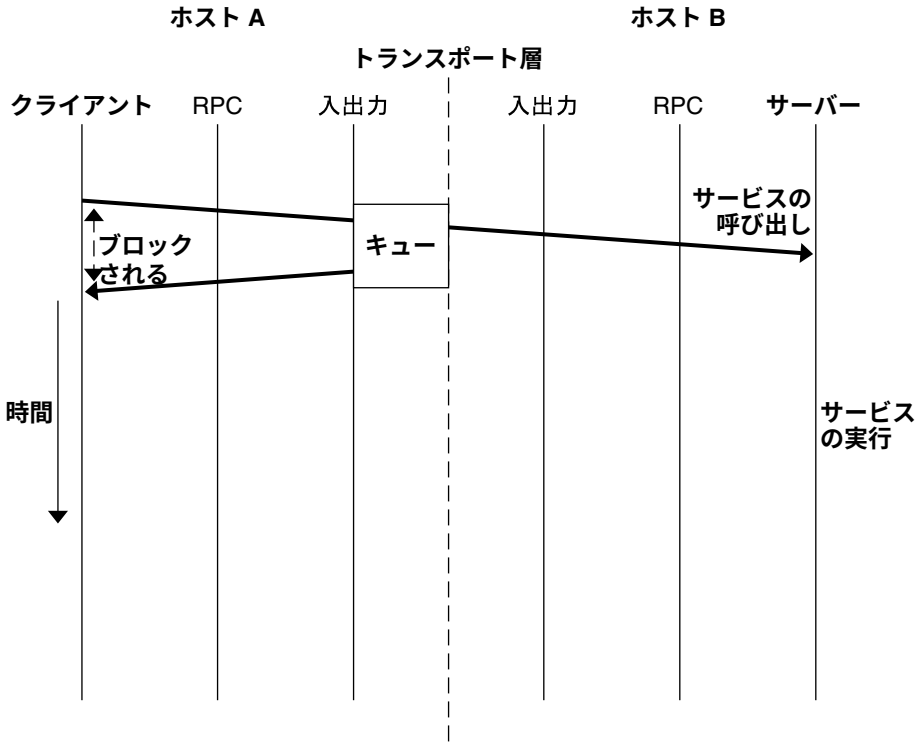

図 **1–2** 1 方向性メッセージング

2. 非-ブロッキング入出力 - クライアントとトランスポート層との間に追加バッファーが 1 つあります。図 1–3 は、入出力モードで非-ブロッキングを選択し、トランスポート層のキューが満杯であるケースを示しています。この状況では、リクエストはトランスポート層のキューに入ることができず、バッファー内に置かれます。クライアントは、そのリクエストがバッファーに受け入れられた場合は、トランスポート層がそのリクエストを受け入れるのを待つことなく処理を続けることができます。非-ブロッキング入出力を使用することにより、2 方向性メッセージングや 1 方向性メッセージングに較べてより多くの処理時間を得ることができます。クライアントは、処理の継続をブロックされることなくリクエストを続けて送信することができます 。

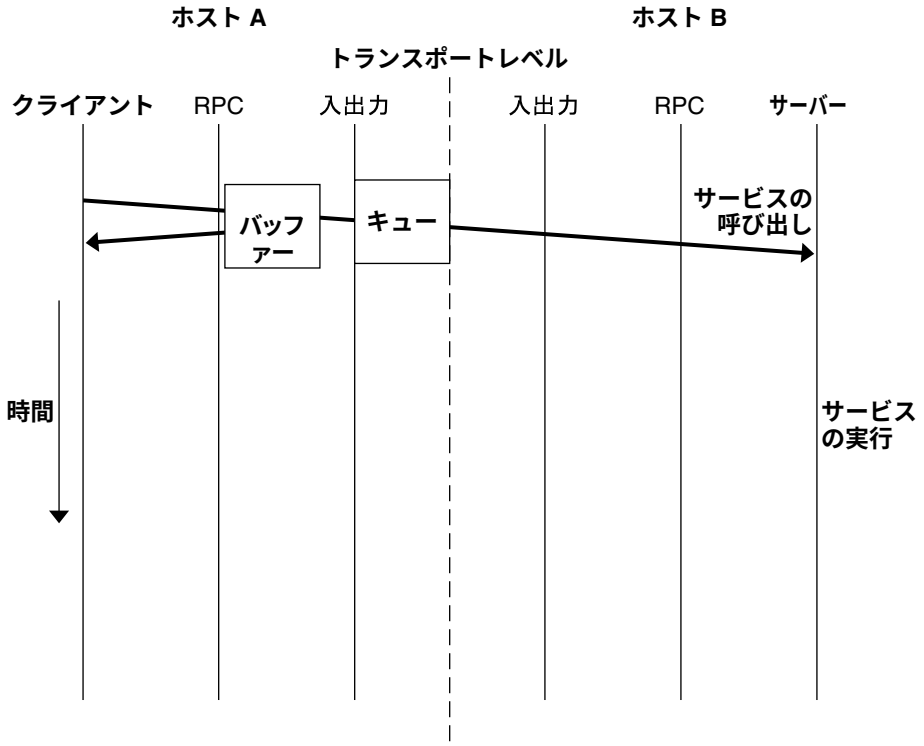

図 **1–3** 非-ブロッキングメッセージング

3. クライアント接続クロージャーコールバック - 接続型トランスポートにおいて、クライアントとの接続が切断されたことをサーバーが検知し、トランスポートエラーから回復するための必要なアクションを行うことができます。トランスポートエラーは、リクエ ストがサーバーに着信した時、またはサーバーがリクエストを待機していて接続が切断された時に起こります。
4. ユーザーファイル記述子コールバック - RPC サーバーが、RPC リクエスト同様非-RPC リクエストも受け入れることができるようにします。Sun RPC ライブラリの以前のバージョンでは、ユーザーが独自のサーバーループを記述するか、ソケット

API にコンタクトをとる別個のスレッドを使用する場合にのみ、RPC コールおよび非-RPC ファイル記述子の両方をサーバーに受け入れさせることが可能でした。

ユーザーファイル記述子コールバックでは、RPC サーバーは、クライアントリクエストを `svc_run()` を使用して実行ループを介して処理します。詳細は、`svc_run(3NSL)` のマニュアルページを参照してください。サーバーはサーバー接続の開始時に実行ループに入り、その接続の終了時に実行ループを終了します。RPC コールを行うが、ユーザーファイル記述子コールバックが無い場合は、非-RPC リクエストの受け入れはブロックされます。

1 方向性メッセージング

1 方向性メッセージングでは、クライアントスレッドがメッセージを含むリクエストをサーバーに送信します。クライアントスレッドはサーバーからの応答を待つことなく、その要求がトランスポート層に受け入れられたら処理を進めることができます。そのリクエ ストはトランスポート層によって常にすぐサーバーに送信されるとは限りませんが、トランスポート層に送信されるまでキューで待機します。サーバーは、リクエスト内に含まれるメッセージを処理することによって、受け取ったリクエストを実行します。

トランスポート層がリクエストを受け取った後は、クライアントは転送の失敗を通知されることはなく、またサーバーからリクエストを受け取った旨を通知されることもありません。たとえば、サーバーが認証上の問題によりリクエストを拒否した場合は、クライアントはこの問題を通知されることはありません。トランスポート層がリクエストを受け入れなかった場合は、送信オペレーションにより直ちにエラーがクライアントに返されます。

注 – サーバーが正しく機能していることをチェックする必要がある場合は、2 方向性のリクエストをサーバーに送信してみます。このリクエストにより、サーバーが利用可能であり、クライアントから送信された 1 方向性リクエストをサーバーが受信中かを判定することができます。

---

1 方向性のメッセージングには、`clnt_send()` 関数が Sun RPC ライブラリに追加され、`oneway` 属性が RPC 文法に追加されています。

clnt_send()

Sun RPC ライブラリの以前のバージョンでは、リモートプロシージャコールの送信に `clnt_call()` 関数を使用していました。拡張された 1 方向性のメッセージングサービスでは、`clnt_send()` 関数が 1 方向性のリモートプロシージャコールを送信します。

クライアントが clnt_send() を呼び出す際は、クライアントはサーバーにリクエストを送信し、処理を続行します。リクエストがサーバーに到着すると、サーバーは着信リクエストを処理するためにディスパッチルーチンを呼び出します。

clnt_send() 関数は、clnt_call() と同様、サービスへアクセスするためにクライアントハンドルを使用します。詳細は、clnt_send(3NSL) および clnt_call(3NSL) のマニュアルページを参照してください。

clnt_create() に適切なバージョン番号を指定しない場合、clnt_call() は失敗します。clnt_send() は、サーバーがステータスを返さないため、同じ状況では失敗を報告しません。

oneway 属性

1 方向性メッセージングを使用するには、oneway キーワードをサーバー関数の XDR 定義に追加します。oneway キーワードを使用する場合は、rpcgen によって生成されるスタブは clnt_send() を使用します。ユーザーは次のいずれかを行うことができます:

- 『*ONC+* 開発ガイド』の「TI-RPC 入門」セクションに概説されている、単純インタフェースを使用します。単純インタフェースに使用されるスタブは clnt_send() を呼び出す必要があります。
- clnt_send(3NSL) のマニュアルページに記述されているように、clnt_send() 関数を直接呼び出します。

1 方向性メッセージングには、rpcgen コマンドのバージョン 1.1 を使用してください。

oneway キーワードを宣言する際は、次の構文を使用する RPC 言語の仕様に従ってください:

```
"oneway" function-ident "(" type-ident-list ")" "=" value;
```

RPC 言語の仕様についての詳細は、『*ONC+* 開発ガイド』の「RPC プロトコルおよび言語の仕様」のセクションを参照してください。

オペレーションに oneway 属性を宣言する際は、サーバーサイドには何の結果も作成されず、クライアントには何のメッセージも返されません。

oneway 属性の情報は、表 1–1 に示すように、『*ONC+* 開発ガイド』の「RPC 言語の仕様」セクションの「RPC 言語定義」の表に追加されていなければなりません。

表 **1–1** RPC 言語定義

用語	定義
プロシージャ (手続き)	`type-ident procedure-ident (type-ident) = value` `oneway procedure-ident (type-ident) = value`

例 **1–1** 単純なカウンターサービスを使用する、1 方向性コール

この例は、単純なカウンターサービスにおいて 1 方向性のプロシージャを使用する方法を例示しています。このカウンターサービスでは、ADD() 関数が、使用できる唯一の関数です。各リモートコールは整数を送信し、この整数は、サーバーによって管理されるグローバルカウンターに追加されます。このサービスを使用するためには、表 1–1 に記述されているように、RPC 言語定義に oneway 属性を宣言する必要があります。

この例では、-M、-N、および -C rpcgen オプションを使用してスタブを生成します。これらのオプションにより確実に、スタブがマルチスレッド-安全で、複数の入力パラメータを受け入れ、生成されたヘッダーが ANSI C++ に適合するようになります。スタブが変わらないので引数を渡すセマンティクスがよりクリアであり、アプリケーションへのスレッドの追加がより簡単になるため、クライアントおよびサーバーアプリケーションがシングルスレッドであっても、これらの rpcgen オプションを使用してください。

1. counter.x ファイルにサービスの記述を書きます。

```
   /* counter.x: リモートカウンタープロトコル */
program COUNTERPROG {
     version COUNTERVERS {
       oneway ADD(int) = 1;
     } = 1;
} = 0x20000001;
```

   このサービスは、プログラム番号（COUNTERPROG）0x200000001、バージョン番号（COUNTERVERS）1 を持ちます。

2. counter.x ファイルに rpcgen を呼び出します。

```
rpcgen -M -N -C counter.x
```

   これにより、クライアントおよびサーバーのスタブ counter.h、counter_clnt.c、counter_svc.c が生成されます。

3. server.c ファイルに示されているように、サーバーサイド用のサービスハンドラ、およびハンドラに割り当てられたメモリー領域を解放するために使用される counterprog_1_freeresult() 関数を記述します。サーバーがクライアントへ応答を送信する時に、RPC ライブラリがこの関数を呼び出します。

```
      #include <stdio.h>
#include "counter.h"

int counter = 0;

       bool_t
add_1_svc(int number, struct svc_req *rqstp)
{
      bool_t retval = TRUE;

       counter = counter + number;

       return retval;
}
      int
```

例 1–1 単純なカウンターサービスを使用する、1 方向性コール　(続き)

```
counterprog_1_freeresult(SVCXPRT *transp, xdrproc_t xdr_result, caddr_t
                         result)
{
       (void) xdr_free(xdr_result, result);

        /*
           * 必要であれば、ここに解放のための追加コードを挿入する
           */

        return TRUE;
}
```

サービスハンドラ、counter_svc.c スタブをコンパイルおよびリンクして、サーバーを構築します。このスタブには、TI-RPC の初期化および処理に関する情報が含まれています。

4. クライアントアプリケーション client.c を記述します。

```
      #include <stdio.h>
#include "counter.h"

main(int argc, char *argv[])
{
      CLIENT *clnt;
      enum clnt_stat result;
      char *server;
      int number;

       if(argc !=3) {
           fprintf(stderr, "usage: %s server_name number\n", argv[0];
              exit(1);
           }
      server = argv[1];
      number = atoi(argv[2]);

       /*
           * クライアントハンドルを作成する
           */
      clnt = clnt_create(server, COUNTERPROG, COUNTERVERS, "tcp");

       if(clnt == (CLIENT *)NULL) {
         /*
          * 接続を確立できなかった
          */
               clnt_pcreateerror(server);
           exit(1);
      }

       result = add_1(number, clnt);
      if (result !=RPC_SUCCESS) {
           clnt_perror(clnt, "call failed");
      }
```

例 1–1 単純なカウンターサービスを使用する、1 方向性コール　(続き)

```
        clnt_destroy(clnt);
       exit(0);
}
```

add_1() クライアント関数は、リモート関数用に生成された counter_clnt.c スタブです。

クライアントを構築するには、クライアントの main および counter_clnt.c をコンパイルおよびリンクします。

5. 構築したサーバーを起動します。

```
% ./server
```

6. 別のシェルでサービスを呼び出します。

```
% ./client servername 23
```

23 は、グローバルカウンターに追加される番号です。

非-ブロッキング入出力

非-ブロッキング入出力は、接続型プロトコルでの 1 方向性メッセージングにおいて、リクエストがトランスポート層に受け入れられるのを待つ間に、クライアントがブロックされるのを防ぎます。

接続型のプロトコルでは、ネットワークプロトコルのキューに入れられるデータの量に上限があります。この上限は、使用されるトランスポートプロトコルにより異なります。クライアントが送信しているリクエストがこのデータ上限に達すると、このクライアントは、そのリクエストがキュー内に入るまで、処理をブロックされます。ユーザーは、このメッセージがキューに追加されるまでどのくらいの時間待つのかを判定することはできません。

非-ブロッキング入出力では、トランスポートのキューが満杯の場合にクライアントとトランスポート層との間で利用可能な追加バッファーがあります。トランスポートのキューに受け入れられなかったリクエストをこのバッファー内に格納できるため、クライアントはブロックされません。リクエストをバッファー内に入れるとすぐに、クライアントは処理を続けることができます。クライアントは、リクエストがキューに入れられるまで待つことはなく、またバッファーがリクエストを受け付けた後でリクエストのステータス情報を受けとることもありません。

非-ブロッキング入出力の使用

非-ブロッキング入出力を使用するには、clnt_control() 関数の CLSET_IO_MODE *rpciomode_t** オプションで RPC_CL_NONBLOCKING 引数を指定して、クライアントハンドルを構成します。詳細は、clnt_control(3NSL) のマニュアルページを参照してください。

トランスポートのキューが満杯の場合には、バッファーが使用されます。次の 2 つの基準が満たされるまで、バッファーが使用され続けます。

- バッファーが空になる。
- キューがリクエストをすぐに受け入れられる。

その後、トランスポートのキューが満杯になるまで、リクエストは直接トランスポートのキューに送られます。バッファーのデフォルトのサイズは、16 K バイトです。

バッファーは自動的に空にされるのではないことに留意してください。バッファーにデータが含まれる場合には、ユーザーがバッファーをフラッシュする必要があります。

CLSET_IO_MODE で RPC_CL_NONBLOCKING 引数を選択している場合は、フラッシュモードを選択することができます。CLSET_FLUSH_MODE に RPC_CLBESTEFFORT_FLUSH または RPC_CL_BLOCKING_FLUSH 引数のいずれかを指定できます。また、clnt_call() などの同期コールを送信することにより、バッファーを空にすることもできます。詳細は、clnt_control(3NSL) のマニュアルページを参照してください。

バッファーが満杯の場合は、RPC_CANTSTORE エラーがクライアントに返され、そのリクエストは送られません。クライアントは、後でそのメッセージを再送信する必要があります。CLGET_CONNMAXREC および CLSET_CONNMAXREC コマンドを使用することにより、バッファーのサイズを確認したり、変更したりすることができます。バッファー内に格納されている、すべての保留状態のリクエストのサイズを確認する場合は、CLGET_CURRENT_REC_SIZE コマンドを使用します。これらのコマンドについての詳細は、clnt_control(3NSL) のマニュアルページを参照してください。

サーバーは、リクエストが受け付けられたかどうかや処理されたかどうかの確認は行いません。ユーザーは、リクエストがバッファーに入った後で clnt_control() を使用すると、リクエストのステータス情報を入手することができます。

例 **1–2** 非-ブロッキング入出力で単純なカウンターを使用する

例 1–1 に示されている client.c ファイルは、非-ブロッキング入出力モードの使用法を例示するために、変更されています。この新しい client_nonblo.c ファイルでは、RPC_CL_NONBLOCKING 引数の使用により入出力モードが非-ブロッキングに指定されており、RPC_CL_BLOCKING_FLUSH の使用によりフラッシュモードがブロッキングに選択されています。入出力モードおよびフラッシュモードは、CLSET_IO_MODE で呼び出されます。エラーが発生すると、RPC_CANT_STORE がクライアントに返され、プログラムによりバッファーのフラッシュが試みられます。フラッシュの別のメソッドを選択するには、clnt_control(3NSL) のマニュアルページを参照してください。

例 **1–2** 非-ブロッキング入出力で単純なカウンターを使用する　*(続き)*

```
#include <stdio.h>
#include "counter.h"

main(int argc, char *argv[])
{
    CLIENT* clnt;
    enum clnt_stat result;
    char *server;
    int number;
    bool_t bres;
        /*
         * 使用する入出力モードとフラッシュメソッドを選択する。
         * この例では、非-ブロッキング入出力モードと
         * ブロッキングフラッシュが選択されている。
         */
    int mode = RPC_CL_NONBLOCKING;
    int flushMode = RPC_CL_BLOCKING_FLUSH;

    if (argc != 3) {
              fprintf(stderr, "usage: %s server_name number\n", argv[0]);
              exit(1);
    }
    server = argv[1];
    number = atoi(argv[2]);

    clnt= clnt_create(server, COUNTERPROG, COUNTERVERS, "tcp");
    if (clnt == (CLIENT*) NULL) {
              clnt_pcreateerror(server);
              exit(1);
    }

        /*
      * clnt_control を使用して入出力モードを設定する。
      * この例では、非-ブロッキング入出力モードが
      * 選択されている。
      */
    bres = clnt_control(clnt, CLSET_IO_MODE, (char*)&mode);
    if (bres)
        /*
          * フラッシュモードをブロッキングに設定する
          */
        bres = clnt_control(clnt, CLSET_FLUSH_MODE, (char*)&flushMode);

    if (!bres) {
        clnt_perror(clnt, "clnt_control");
        exit(1);
    }
         /*
       * RPC サービスを呼び出す。
       */
    result = add_1(number, clnt);

    switch (result) {
```

例 **1–2** 非-ブロッキング入出力で単純なカウンターを使用する　(続き)

```
    case RPC_SUCCESS:
        fprintf(stdout,"Success\n");
        break;
        /*
      * RPC_CANTSTORE は、バッファーがリクエストを格納できない場合に、
         * クライアントに返される新しい値。
         */
    case RPC_CANTSTORE:
        fprintf(stdout,"RPC_CANTSTORE error. Flushing ... \n");
        /*
         * バッファーは、ブロッキングフラッシュを使用してフラッシュされる
         */
              bres = clnt_control(clnt, CLFLUSH, NULL);
        if (!bres) {
                    clnt_perror(clnt, "clnt_control");
              }
        break;
    default:
              clnt_perror(clnt, "call failed");
        break;
    }

    /* フラッシュする */
    bres = clnt_control(clnt, CLFLUSH, NULL);
    if (!bres) {
              clnt_perror(clnt, "clnt_control");
    }

    clnt_destroy(clnt);
    exit(0);
}
```

非-ブロッキングとして構成したときの clnt_call()

1 方向性メッセージングには、clnt_send() 関数を使用します。クライアントがリクエストをサーバーに送信する際、応答を待機しないので、タイムアウトは適用されません。

2 方向性メッセージングには、clnt_call() を使用します。サーバーが応答を送信するかエラーのステータスメッセージを送信するか、またはクライアントサイドでタイムアウトが発生するまで、クライアントはブロックされたままになります。

非-ブロッキング機能では、2 方向性と 1 方向性のコールを共に送信することができます。RPC_CL_NONBLOCKING 入出力モードを使用し、非-ブロッキングとして構成したクライアントサイドで clnt_call() を使用すると、次のような動作の変更があります。2 方向性のリクエストがバッファーに送られると、バッファー内にすでに入って

いる 1 方向性のすべてのリクエストが、その 2 方向性のリクエストが処理される前にトランスポート層を介して送られます。バッファーを空にするための時間は、2 方向性コールのタイムアウトにはカウントされません。詳細は、clnt_control(3NSL) のマニュアルページを参照してください。

クライアント接続クロージャーコールバック

クライアント接続クロージャーコールバックでは、サーバーが、クライアントの接続が切断されていることを検知し、クライアント接続が切断されたことによるエラーから回復できるようにします。

接続クロージャーコールバックは、接続上でリクエストが現在全く処理されていない時に呼び出されます。リクエストが処理される時にクライアント接続が切断されると、サーバーはそのリクエストを処理しますが、応答がクライアントに送られないことがあります。接続クロージャーコールバックは、すべての待機中のリクエストが完了した時に呼び出されます。

接続の切断が起こると、トランスポート層がクライアントへエラーメッセージを送信します。たとえば次のように、svc_control() を使用してハンドラがサービスに添付されます:

```
svc_control(service, SVCSET_RECVERRHANDLER, handler);
```

svc_control() の引数は、次のとおりです:

1. サービスまたはこのサービスのインスタンス。引数がサービスの場合は、そのサービスへのすべての新規の接続はエラーハンドラを継承します。引数がサービスのインスタンスの場合は、この接続だけがエラーハンドラを取得します。
2. エラーハンドラのコールバック。このコールバック関数のプロトタイプは次のようになります:

```
void handler(const SVCXPRT *svc, const boot_t IsAConnection);
```

詳細は svc_control(3NSL) のマニュアルページを参照してください。

注 – XDR 非整列化エラーについては、サーバーがリクエストを非整列化できない場合、メッセージは破棄されエラーが直接クライアントへ返されます。

例 **1–3** クライアント接続クロージャーコールバックを使用した例

この例では、メッセージログサーバーを実装しています。クライアントは、ログ (実体はテキストファイル) を開いたり、メッセージログを保存したり、ログを閉じたりするのにこのサーバーを使用することができます。

例 **1–3** クライアント接続クロージャーコールバックを使用した例　(続き)

log.x ファイルは、ログプログラムのインタフェースを記述します。

```
enum log_severity { LOG_EMERG=0, LOG_ALERT=1, LOG_CRIT=2, LOG_ERR=3,
            LOG_WARNING=4, LOG_NOTICE=5, LOG_INFO=6 };

program LOG {
              version LOG_VERS1 {
                          int OPENLOG(string ident) = 1;

                          int CLOSELOG(int logID) = 2;

                          oneway WRITELOG(int logID, log_severity severity,
                  string message) = 3;
                } = 1;
} = 0x20001971;
```

2 つのプロシージャ (OPENLOG および CLOSELOG) は、logID で指定されたログをそれぞれ開いたり閉じたりします。WRITELOG() プロシージャ (この例では oneway として宣言されている) は、開かれたログにメッセージを記録します。ログメッセージは、重要度属性およびテキストメッセージを含みます。

これは、ログサーバーの Makefile です。この Makefile を使用して、log.x ファイルを呼び出します。

```
RPCGEN = rpcgen

CLIENT = logClient
CLIENT_SRC = logClient.c log_clnt.c log_xdr.c
CLIENT_OBJ = $(CLIENT_SRC:.c=.o)

SERVER = logServer
SERVER_SRC = logServer.c log_svc.c log_xdr.c
SERVER_OBJ = $(SERVER_SRC:.c=.o)

RPCGEN_FILES = log_clnt.c log_svc.c log_xdr.c log.h

CFLAGS += -I.

RPCGEN_FLAGS    = -N -C
LIBS = -lsocket -lnsl

all: log.h ./$(CLIENT) ./$(SERVER)


$(CLIENT): log.h $(CLIENT_OBJ)
              cc -o $(CLIENT) $(LIBS) $(CLIENT_OBJ)

$(SERVER): log.h $(SERVER_OBJ)
              cc -o $(SERVER)  $(LIBS) $(SERVER_OBJ)

$(RPCGEN_FILES): log.x
              $(RPCGEN) $(RPCGEN_FLAGS) log.x
```

例 **1–3** クライアント接続クロージャーコールバックを使用した例　*(続き)*

```
clean:
              rm -f $(CLIENT_OBJ) $(SERVER_OBJ) $(RPCGEN_FILES)
```

logServer.c は、ログサーバーの実装を示します。ログサーバーは、ログメッセージを保存するためにファイルを開くため、openlog_1_svc() にクロージャー接続コールバックを登録します。クライアントプログラムが closelog() プロシージャを呼び出すことを忘れた (または呼び出す前にクラッシュした) 場合でも、ファイル記述子が閉じられるようにこのコールバックが使用されます。この例は、RPC サーバー内のクライアントに関連付けられたリソースを解放するのに接続クロージャーコールバック機能を使用する方法を例示しています。

```
#include "log.h"
#include <stdio.h>
#include <string.h>

#define NR_LOGS 3

typedef struct {
    SVCXPRT* handle;
    FILE* filp;
    char* ident;
} logreg_t;


static logreg_t logreg[NR_LOGS];
static char* severityname[] = {"Emergency", "Alert", "Critical", "Error",
                               "Warning", "Notice", "Information"};

    static void
close_handler(const SVCXPRT* handle, const bool_t);


    static int
get_slot(SVCXPRT* handle)
{
    int i;

    for (i = 0; i < NR_LOGS; ++i) {
        if (handle == logreg[i].handle) return i;
    }
    return -1;
}

    static FILE*
_openlog(char* logname)
/*
 * ログファイルを開く
 */
{
    FILE* filp = fopen(logname, "a");
    time_t t;
```

例 **1–3** クライアント接続クロージャーコールバックを使用した例　(続き)

```
    if (NULL == filp) return NULL;

    time(&t);
    fprintf(filp, "Log opened at %s\n", ctime(&t));

    return filp;
}

    static void
_closelog(FILE* filp)
{
    time_t t;

    time(&t);
    fprintf(filp, "Log close at %s\n", ctime(&t));
        /*
      * ログファイルを閉じる
      */
    fclose(filp);
}

    int*
openlog_1_svc(char* ident, struct svc_req* req)
{
    int slot = get_slot(NULL);
    FILE* filp;
    static int res;
    time_t t;

    if (-1 != slot) {
        FILE* filp = _openlog(ident);
        if (NULL != filp) {
            logreg[slot].filp = filp;
            logreg[slot].handle = req->rq_xprt;
            logreg[slot].ident = strdup(ident);

        /*
      * クライアントが clnt_destroy を呼び出すか、
      * クライアントの接続が切断されて clnt_destroy が自動的に呼び出されると、
      * サーバーは close_handler コールバックを実行する
      */
            if (!svc_control(req->rq_xprt, SVCSET_RECVERRHANDLER,
                                        (void*)close_handler)) {
                        puts("Server: Cannot register a connection closure
                    callback");
                        exit(1);
                    }
        }

    }
    res = slot;
    return &res;
```

例 **1–3** クライアント接続クロージャーコールバックを使用した例　*(続き)*

```
}

    int*
closelog_1_svc(int logid, struct svc_req* req)
{
    static int res;

    if ((logid>= NR_LOGS) || (logreg[logid].handle != req->rq_xprt)) {
        res = -1;
        return &res;
    }
    logreg[logid].handle = NULL;
    _closelog(logreg[logid].filp);
    res = 0;
    return &res;
}

/*
 * メッセージをログへ書き込むようリクエストがあると、
 * write_log_1_svc が呼び出される
 */
    void*
writelog_1_svc(int logid, log_severity severity, char* message,
               struct svc_req* req)
{
    if ((logid>= NR_LOGS) || (logreg[logid].handle != req->rq_xprt)) {
        return NULL;
    }
        /*
         * メッセージをファイルへ書き込む
         */
    fprintf(logreg[logid].filp, "%s (%s): %s\n",
            logreg[logid].ident, severityname[severity], message);
    return NULL;
}

    static void
close_handler(const SVCXPRT* handle, const bool_t dummy)
{
    int i;

        /*
         * クライアントの接続が切断されると、closelog でログが閉じられる
         */
    for (i = 0; i < NR_LOGS; ++i) {
        if (handle == logreg[i].handle) {
            logreg[i].handle = NULL;
            _closelog(logreg[i].filp);
        }
    }
}
```

logClient.c ファイルは、ログサーバーを使用するクライアントを示しています。

例 **1–3** クライアント接続クロージャーコールバックを使用した例　(続き)

```
#include "log.h"
#include <stdio.h>

#define MSG_SIZE 128

    void
usage()
{
    puts("Usage: logClient <logserver_addr>");
    exit(2);
}

    void
runClient(CLIENT* clnt)
{
    char msg[MSG_SIZE];
    int logID;
    int* result;

         /*
       * クライアントがログを開く
       */
    result = openlog_1("client", clnt);
    if (NULL == result) {
        clnt_perror(clnt, "openlog");
        return;
    }
    logID = *result;
    if (-1 == logID) {
        puts("Cannot open the log.");
        return;
    }

    while(1) {
        struct rpc_err e;

             /*
               * クライアントがメッセージをログに書き込む
               */
        puts("Enter a message in the log (\".\" to quit):");
        fgets(msg, MSG_SIZE, stdin);
        /*
                * 末尾の CR を削除する
                */
        msg[strlen(msg)-1] = 0;

        if (!strcmp(msg, ".")) break;

        if (writelog_1(logID, LOG_INFO, msg, clnt) == NULL) {
            clnt_perror(clnt, "writelog");
            return;
        }
    }
```

例 1–3 クライアント接続クロージャーコールバックを使用した例　(続き)

```
        /*
         * クライアントがログを閉じる
         */
    result = closelog_1(logID, clnt);
    if (NULL == result) {
        clnt_perror(clnt, "closelog");
        return;
    }
    logID = *result;
    if (-1 == logID) {
        puts("Cannot close the log.");
        return;
    }
}

    int
main(int argc, char* argv[])
{
    char* serv_addr;
    CLIENT* clnt;

    if (argc != 2) usage();

    serv_addr = argv[1];

    clnt = clnt_create(serv_addr, LOG, LOG_VERS1, "tcp");

    if (NULL == clnt) {
        clnt_pcreateerror("Cannot connect to log server");
        exit(1);
    }
    runClient(clnt);

    clnt_destroy(clnt);
}
```

ユーザーファイル記述子コールバック

ユーザーファイル記述子コールバックでは、ユーザーが、コールバックにファイル記述子を登録して 1 つまたは複数のイベントタイプを指定できるようになります。これにより、RPC サーバーを使用して、Sun RPC ライブラリ用に書かれていないファイル記述子を扱えるようになります。

ユーザーファイル記述子コールバックを実装するために、2 つの新しい関数 `svc_add_input`(3NSL) と `svc_remove_input`(3NSL) が Sun RPC ライブラリに追加されました。これらの関数は、ファイル記述子とともに、コールバックの宣言または削除を行います。

この新しいコールバック機能を使用する際は、ユーザーは次を行う必要があります。

1. 次の構文でユーザーコードを記述して、callback() 関数を作成します:

   ```
   typedef void (*svc_callback_t) (svc_input_id_t id, int fd, \
   unsigned int revents, void* cookie);
   ```
   callback() 関数に渡される 4 つのパラメータは、次のとおりです:

   - *id* - 各コールバックに識別子を提供する。この識別子は、コールバックを削除するのに使用できます。
   - *fd* - ユーザーのコールバックが待機する対象のファイル記述子。
   - *revents* - 発生したイベントを表す、符号無しの整数。このイベントセットは、コールバックが登録された時に指定されたリストのサブセットです。
   - *cookie* - コールバックが登録された時に指定された cookie。この cookie は、サーバーがコールバック時に必要とする特定のデータを指定することができます。
2. サーバーが識別する必要のある、ファイル記述子、および読み取りや書き込みなどの関連イベントを登録するために、svc_add_input() を呼び出します。

   ```
   svc_input_id_t svc_add_input (int fd, unsigned int revents, \
   svc_callback_t callback, void* cookie);
   ```
   指定可能なイベントのリストについては、poll(2) を参照してください。
3. ファイル記述子を指定します。このファイル記述子は、ソケットやファイルなどのエンティティにすることができます。

指定されたイベントのいずれかが発生すると、標準のサーバーループが svc_run() を介してユーザーコードを呼び出し、ユーザーのコールバックがファイル記述子 (ソケットまたはファイル) 上で必要な操作を実行します。

特定のコールバックが必要でなくなった場合は、そのコールバックを削除するために、対応する識別子を指定して svc_remove_input() を呼び出します。

例 **1–4** RPC サーバーにユーザーファイル記述子を登録する方法

この例は、RPC サーバーにユーザーファイル記述子を登録する方法、およびユーザーが定義したコールバックを提供する方法を示します。この例では、サーバーとクライアント両方での日時を監視できます。

1. この例の Makefile を次に示します。

```
   RPCGEN = rpcgen

CLIENT = todClient
CLIENT_SRC = todClient.c timeofday_clnt.c
CLIENT_OBJ = $(CLIENT_SRC:.c=.o)

SERVER = todServer
SERVER_SRC = todServer.c timeofday_svc.c
SERVER_OBJ = $(SERVER_SRC:.c=.o)

RPCGEN_FILES = timeofday_clnt.c timeofday_svc.c  timeofday.h
```

例 1–4 RPC サーバーにユーザーファイル記述子を登録する方法　(続き)

```
CFLAGS += -I.

RPCGEN_FLAGS = -N -C
LIBS = -lsocket -lnsl

all: ./$(CLIENT) ./$(SERVER)


$(CLIENT): timeofday.h $(CLIENT_OBJ)
              cc -o $(CLIENT) $(LIBS) $(CLIENT_OBJ)

$(SERVER): timeofday.h $(SERVER_OBJ)
              cc -o $(SERVER)  $(LIBS) $(SERVER_OBJ)

timeofday_clnt.c: timeofday.x
                $(RPCGEN) -l $(RPCGEN_FLAGS)
 timeofday.x> timeofday_clnt.c

timeofday_svc.c: timeofday.x
              $(RPCGEN) -m $(RPCGEN_FLAGS) timeofday.x> timeofday_svc.c

timeofday.h: timeofday.x
              $(RPCGEN) -h $(RPCGEN_FLAGS) timeofday.x> timeofday.h


clean:
              rm -f $(CLIENT_OBJ) $(SERVER_OBJ) $(RPCGEN_FILES)
```

2. timeofday.x ファイルは、この例の中でサーバーによって提供される RPC サービスを定義します。この例のサービスは、gettimeofday() および settimeofday() です。

```
program TIMEOFDAY {
              version VERS1 {
                           int SENDTIMEOFDAY(string tod) = 1;

                           string GETTIMEOFDAY() = 2;
              } = 1;
} = 0x20000090;
```

3. userfdServer.h ファイルは、この例におけるソケットで送られるメッセージの構造を定義します。

```
   #include "timeofday.h"
#define PORT_NUMBER 1971

/*
 * 接続用のデータを保存するのに使用される構造
     *    (user fds test)
 */
typedef struct {
    /*
 * このリンクのコールバック登録の ID
 */
```

例 1–4 RPC サーバーにユーザーファイル記述子を登録する方法　(続き)

```
    svc_input_id_t in_id;
    svc_input_id_t out_id;

    /*
          * この接続から読み取られるデータ
          */
    char in_data[128];

    /*
          * この接続に書き込まれるデータ
          */
    char out_data[128];
    char* out_ptr;

} Link;

    void
socket_read_callback(svc_input_id_t id, int fd, unsigned int events,
                                    void* cookie);
    void
socket_write_callback(svc_input_id_t id, int fd, unsigned int events,
                        void* cookie);
    void
socket_new_connection(svc_input_id_t id, int fd, unsigned int events,
                        void* cookie);

    void
timeofday_1(struct svc_req *rqstp, register SVCXPRT *transp);
```

4. todClient.c ファイルは、クライアントで日時がどのように設定されるかを示します。このファイルでは、RPC はソケットとともにでも、ソケットなしでも使用されます。

```
   #include "timeofday.h"

#include <stdio.h>
#include <netdb.h>
#define PORT_NUMBER 1971

    void
runClient();
    void
runSocketClient();


char* serv_addr;

    void
usage()
{
    puts("Usage: todClient [-socket] <server_addr>");
    exit(2);
}
```

例 1–4 RPC サーバーにユーザーファイル記述子を登録する方法　(続き)

```
    int
main(int argc, char* argv[])
{
    CLIENT* clnt;
    int sockClient;

    if ((argc != 2) && (argc != 3))
        usage();

    sockClient = (strcmp(argv[1], "-socket") == 0);

        /*
         * ソケットの使用を選択する (sockClient)。
         * ソケットが利用できない場合は、
         * ソケットなしで RPC を使用する (runClient)。
         */
    if (sockClient && (argc != 3))
        usage();

    serv_addr = argv[sockClient? 2:1];

    if (sockClient) {
        runSocketClient();
    } else {
        runClient();
    }

    return 0;
}
/*
     * ソケットなしで RPC を使用する
 */
    void
runClient()
{
    CLIENT* clnt;
    char* pts;
    char** serverTime;

    time_t now;

    clnt = clnt_create(serv_addr, TIMEOFDAY, VERS1, "tcp");
    if (NULL == clnt) {
        clnt_pcreateerror("Cannot connect to log server");
        exit(1);
    }

    time(&now);
    pts = ctime(&now);

    printf("Send local time to server\n");
```

例 1–4 RPC サーバーにユーザーファイル記述子を登録する方法　(続き)

```
    /*
          * 現地の時刻を設定し、この時刻をサーバーへ送信する。
          */
    sendtimeofday_1(pts, clnt);

    /*
          * サーバーに現在の時刻を尋ねる。
          */
    serverTime = gettimeofday_1(clnt);

    printf("Time received from server: %s\n", *serverTime);

    clnt_destroy(clnt);
}

/*
 * ソケットとともに RPC を使用する
 */
    void
runSocketClient()
/*
 * ソケットを作成する
 */
{
    int s = socket(PF_INET, SOCK_STREAM, 0);
    struct sockaddr_in sin;
    char* pts;
    char buffer[80];
    int len;

    time_t now;
    struct hostent* hent;
    unsigned long serverAddr;

    if (-1 == s) {
        perror("cannot allocate socket.");
        return;
    }

    hent = gethostbyname(serv_addr);
    if (NULL == hent) {
        if ((int)(serverAddr = inet_addr(serv_addr)) == -1) {
            puts("Bad server address");
            return;
        }
    } else {
        memcpy(&serverAddr, hent->h_addr_list[0], sizeof(serverAddr));
    }

    sin.sin_port = htons(PORT_NUMBER);
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = serverAddr;
```

例 1–4 RPC サーバーにユーザーファイル記述子を登録する方法　(続き)

```
        /*
         * ソケットを接続する
         */
    if (-1 == connect(s, (struct sockaddr*)(&sin),
                      sizeof(struct sockaddr_in))) {
        perror("cannot connect the socket.");
        return;
    }

    time(&now);
    pts = ctime(&now);

        /*
         * ソケット上にメッセージを記述する。
         * メッセージはクライアントの現在時刻。
         */
    puts("Send the local time to the server.");
    if (-1 == write(s, pts, strlen(pts)+1)) {
        perror("Cannot write the socket");
        return;
    }

        /*
         * ソケット上のメッセージを読み取る。
         * メッセージはサーバーの現在時刻。
         */
    puts("Get the local time from the server.");
    len = read(s, buffer, sizeof(buffer));

    if (len == -1) {
        perror("Cannot read the socket");
        return;
    }
    puts(buffer);

    puts("Close the socket.");
    close(s);
}
```

5. todServer.c ファイルは、サーバーサイドからの timeofday サービスの使用法を示します。

```
   #include "timeofday.h"
#include "userfdServer.h"
#include <stdio.h>
#include <errno.h>
#define PORT_NUMBER 1971

int listenSocket;

/*
 * RPC サーバーの実装
 */
```

例 1–4 RPC サーバーにユーザーファイル記述子を登録する方法　(続き)

```
    int*
sendtimeofday_1_svc(char* time, struct svc_req* req)
{
    static int result = 0;

    printf("Server: Receive local time from client  %s\n", time);
    return &result;
}

    char **
gettimeofday_1_svc(struct svc_req* req)
{
    static char buff[80];
    char* pts;
    time_t now;
    static char* result = &(buff[0]);

    time(&now);
    strcpy(result, ctime(&now));

    return &result;
}

/*
     * ソケットサーバーの実装
 */

    int
create_connection_socket()
{
    struct sockaddr_in sin;
    int size = sizeof(struct sockaddr_in);
    unsigned int port;

         /*
          * ソケットを作成する
          */
    listenSocket = socket(PF_INET, SOCK_STREAM, 0);

    if (-1 == listenSocket) {
        perror("cannot allocate socket.");
        return -1;
    }

    sin.sin_port = htons(PORT_NUMBER);
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = INADDR_ANY;

    if (bind(listenSocket, (struct sockaddr*)&sin, sizeof(sin)) == -1) {
        perror("cannot bind the socket.");
        close(listenSocket);
        return -1;
```

例 **1–4** RPC サーバーにユーザーファイル記述子を登録する方法　*(続き)*

```
    }
         /*
          * サーバーは、クライアント接続が
          * 作成されるのを待機する
          */
    if (listen(listenSocket, 1)) {
        perror("cannot listen.");
        close(listenSocket);
        listenSocket = -1;
        return -1;
    }

         /*
          * svc_add_input は、待機しているソケット上に、
          * 読み取りコールバック socket_new_connection を登録する。
          * 新しい接続が保留状態の時に、
          * このコールバックが呼び出される。*/
    if (svc_add_input(listenSocket, POLLIN,
                      socket_new_connection, (void*) NULL) == -1) {
        puts("Cannot register callback");
        close(listenSocket);
        listenSocket = -1;
        return -1;
    }

    return 0;
}

/*
 * socket_new_connection コールバック関数を定義する
 */
    void
socket_new_connection(svc_input_id_t id, int fd,
                      unsigned int events, void* cookie)
{
    Link* lnk;
    int connSocket;

         /*
          * ソケット上で接続が保留状態にある時に、
          * サーバーが呼び出される。この接続を今受け付ける。
          * コールは非-ブロッキング。
          * このコールを扱うためにソケットを作成する。
          */
    connSocket = accept(listenSocket, NULL, NULL);
    if (-1 == connSocket) {
        perror("Server: Error: Cannot accept a connection.");
        return;
    }

    lnk = (Link*)malloc(sizeof(Link));
    lnk->in_data[0] = 0;
```

例 **1–4** RPC サーバーにユーザーファイル記述子を登録する方法　*(続き)*

```
            /*
             * 新規のコールバック socket_read_callback が作成された。
             */
        lnk->in_id = svc_add_input(connSocket, POLLIN,
                        socket_read_callback, (void*)lnk);
    }
    /*
     * 新規のコールバック socket_read_callback が定義される
     */
        void
    socket_read_callback(svc_input_id_t id, int fd, unsigned int events,
                         void* cookie)
    {
        char buffer[128];
        int len;
        Link* lnk = (Link*)cookie;

        /*
              * メッセージを読み取る。この読み取りコールはブロックは行わない。
              */
        len = read(fd, buffer, sizeof(buffer));

        if (len> 0) {
              /*
                    * データを取得した。このソケット接続に
                    * 関連付けられたバッファー内にそのデータをコピーする。
                    */
            strncat (lnk->in_data, buffer, len);

            /*
                    * 完全なデータを受信したかどうかをテストする。
             * 完全なデータでない場合は、これは部分的な読み取りである。
                    */
            if (buffer[len-1] == 0) {
                char* pts;
                time_t now;

                /*
                         * 受信した日時を出力する。
                         */
                printf("Server: Got time of day from the client: \n %s",
                       lnk->in_data);

                /*
                         * 応答データをセットアップする
                         * (サーバーの現在の日時)。
                         */
                time(&now);
                pts = ctime(&now);

                strcpy(lnk->out_data, pts);
                lnk->out_ptr = &(lnk->out_data[0]);
```

例 1–4 RPC サーバーにユーザーファイル記述子を登録する方法　(続き)

```
                    /*
                     * 応答の書き込み時にブロックを行わない
                     * 書き込みコールバック (socket_write_callback) を登録す
る。
                     * ソケットへの書き込みアクセス権を保持している場合は、
                     * POLLOUT を使用することができる。
                     */
            lnk->out_id = svc_add_input(fd, POLLOUT,
                                        socket_write_callback, (void*)
lnk);

        }
    } else if (len == 0) {
    /*
     * 相手側でソケットがクローズされた。ソケットをクローズする。
     */
    close(fd);
    } else {
        /*
                * ソケットが相手側によりクローズされているか ?
                */
        if (errno != ECONNRESET) {
            /*
                     * クローズされていない場合は、これはエラーである。
                     */
            perror("Server: error in reading the socket");
            printf("%d\n", errno);
        }
        close(fd);
    }
}

/*
 * socket_write_callback を定義する。
 * ソケットへの書き込みアクセス権を保持している場合は、
 * このコールバックが呼び出される。
 */
    void
socket_write_callback(svc_input_id_t id, int fd, unsigned int events,
                      void* cookie)
{
    Link* lnk = (Link*)cookie;

    /*
          * 書き込む残りのデータ長を計算する。
          */
    int len = strlen(lnk->out_ptr)+1;

/*
 * 時間をクライアントへ送信する
 */
    if (write(fd, lnk->out_ptr, len) == len) {
    /*
```

例 **1–4** RPC サーバーにユーザーファイル記述子を登録する方法　*(続き)*

```
            * すべてのデータが送られた。
            */

          /*
                * 2 つのコールバックの登録を解除する。
            * この登録解除は、ファイル記述子がクローズされる時に
           * この登録が自動的に削除されるため、
                * ここに例示されている。
                */
          svc_remove_input(lnk->in_id);
          svc_remove_input(lnk->out_id);

          /*
                  * ソケットをクローズする。
                  */
          close(fd);
      }
  }

      void
  main()
  {
      int res;

           /*
            * timeofday サービスおよびソケットを作成する
            */
      res = create_connection_socket();
      if (-1 == res) {
          puts("server: unable to create the connection socket.\n");
          exit(-1);
      }

      res = svc_create(timeofday_1, TIMEOFDAY, VERS1, "tcp");
      if (-1 == res) {
          puts("server: unable to create RPC service.\n");
          exit(-1);
      }

           /*
            ユーザーファイル記述子をポーリングする。
            */
      svc_run();
  }
```

索引