ONC+ 開発ガイド

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A. 650–960–1300

Part No: 805-5826–10
1998 年 11 月

原典: *ONC+ Developers Guide*

Part No: 805-4034-10

Revision A

Adobe PostScript

目次

はじめに

このマニュアルでは、遠隔手続き呼び出し (RPC) と、米国 SunSoft™, Inc. (以下、「サン・ソフト」とします) が開発した ONC+™ 分散サービスに含まれるネットワークネームサービスである NIS+ のためのプログラミングインタフェースについて説明します。

このマニュアルで説明するインタフェースは SunOS™ と Solaris™ とで共通であるため、ここでは SunOS と Solaris とは同じ意味で使用しています。Solaris 2.5 は サン・ソフト の分散コンピューティングオペレーティング環境です。これは、SunOS リリース 5.5 に ONC+ 技術、OpenWindows™、ToolTalk™、DeskSet™、OPEN LOOK、およびその他のユーティリティを統合したものです。

このマニュアルで説明するユーティリティ (オプションのユーティリティも含む)、ライブラリ関数は、すべて最新の Solaris 用のものです。Solaris は、サン・ソフトで開発したシステムソフトウェアです。旧バージョンの Solaris システムソフトウェア上でユーティリティやライブラリ関数をご使用になると、その動作が異なる場合があります。

対象読者

このマニュアルは、単独のコンピュータ用アプリケーションをネットワークに対応する分散アプリケーションに変換するユーザー、または分散アプリケーションの開発と実装を行うユーザーを対象にしています。

このマニュアルでは、基本的なプログラミングの能力と、C 言語および UNIX オペレーティングシステムの使用経験を前提としています。ネットワーク対応のプログラミングの経験があれば役に立ちますが、必須ではありません。

内容の紹介

Part 1-入門

第 1 章では、ONC+ 分散コンピューティングプラットフォームとサービスについて詳しく紹介します。

Part 2-遠隔手続き呼び出し (RPC)

第 2 章では TI-RPC を紹介します。

第 3 章では、`rpcgen` ツールを使用してクライアントとサーバーのスタブを作成する方法を説明します。

第 4 章では、プログラミング環境での RPC の使用方法について説明します。

Part 3-NIS+ アプリケーションプログラミングインタフェース

第 5 章では、NIS+ アプリケーションプログラミングインタフェースについて説明します。

付録

付録 A では、データのフォーマットと型変換のために XDR を使用する方法を説明します。

付録 B では、RPC 用プロトコルについて、構文と制限事項を説明します。

付録 C では、XDR プロトコルと言語について説明します。

付録 D では、このマニュアルで使用するサンプルプログラムの一部について、全リストを収録します。

付録 E では、portmap ユーティリティとその機能を説明します。付録 E は、SunOS の旧バージョンで作成したアプリケーションを現バージョンで使用するときに参照してください。

付録 F では、アプリケーション開発の参考として、SAF を使用するポートモニタプログラムの作成方法を示します。

関連マニュアル

次のオンライン AnswerBook™ 製品はネットワークプログラミングについて説明しています。

- 『*Solaris 2.6* リファレンスマニュアル *AnswerBook*』
- 『*Solaris 2.6* ソフトウェア開発 *Collection*』

サン・ソフト の NFS™ 分散コンピューティングファイルシステムについては、以下を参照してください。

- 『NFS: Network File System Protocol Specification』RFC 1094、27 ページ (Mar)、Sun Microsystems、1988 年。
- 『NFS: Network File System Version 3 Protocol Specification』 Sun Microsystems、1993 年。Postscript™ ファイルが anonymous ftp で入手できます。

  `ftp.uu.net:/networking/ip/nfs/NFS3.spec.ps.Z`

  `bcm.tmc.edu:/nfs/nfsv3.ps.Z`

  `gatekeeper.dec.com:/pub/standards/nfs/nfsv3.ps.Z`

以下の一般図書と記事には、ネットワークプログラミングについての詳しい説明があります。

- UNIX Network Programming, W. Richard Stevens (Prentice Hall Software Series, 1990)
- Power Programming with RPC, John Bloomer (O'Reilly & Associates, Inc, 1992)
- Networking Applications on UNIX System V Release 4, Michael Padovano (Prentice Hall, Inc., 1993)
- Distributed Computing:Implementation and Managment Strategies (Edited by Raman Khanna. Prentice Hall, 1993)

- Using Encryption for Authentication in Large Networks of Computers, R.M. Needham and M.D. Schroeder in *Communications of the ACM* (Vol. 21, No. 12, pages 993-999, 1978)
- Section E.2.1: Kerberos Authentication and Authorization System,S.P. Miller, B.C Neuman, J.I. Schiller and J.H. Saltzer *(Project Athena Technical Plan*, MIT Project Athena, December 1987)
- Kerberos: An Authentication Service for Open Network Systems, J.G. Steiner, B.C. Neuman, and J.I.Schiller (Usenix Conference Proceedings, Dallas, TX, pages 191-202, February, 1988)

マニュアルの注文方法

SunDocs™ プログラムでは、米国 Sun Microsystems™, Inc. (以降、Sun™ とします) の 250 冊以上のマニュアルを扱っています。このプログラムを利用して、マニュアルのセットまたは個々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、米国 SunExpress™, Inc. のインターネットホームページ `http://www.sun.com/sunexpress` にあるカタログセクションを参照してください。

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、または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は区別して示します。	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、または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越える場合、バックスラッシュは継続を示します。	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 シェルプロンプト

```
system% command y|n [filename]
```

- Bourne シェルおよび Korn シェルのプロンプト

```
system$ command y|n [filename]
```

- スーパーユーザーのプロンプト

```
system# command y|n [filename]
```

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

| 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。
- 「x86」という用語は、一般に Intel 8086 ファミリに属するマイクロプロセッサを意味します。これには、Pentium、Pentium Pro の各プロセッサ、および AMD と 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。このマニュアルでは、このプラットフォームのアーキテクチャ全体を指すときに「x86」という用語を使用し、製品名では「Intel 版」という表記で統一しています。

パート I 入門

Part 1 では ONC+ サービスについて紹介します。

- 第 1 章

第 1 章

ONC+ 入門

この章では、Sun のオープンシステム分散コンピューティング環境である ONC+ について簡単に紹介します。ONC+ は、異機種分散コンピューティング環境において分散アプリケーションを実装する開発者が利用できるサービスの中心に位置づけられるものです。ONC+ には、クライアント / サーバーネットワークを管理するツールも含まれています。

図 1–1 は ONC+ の上部に統合されているクライアント / サーバーアプリケーションと、それらが低レベルのネットワークプロトコルの上部に位置づけられている様子を示しています。

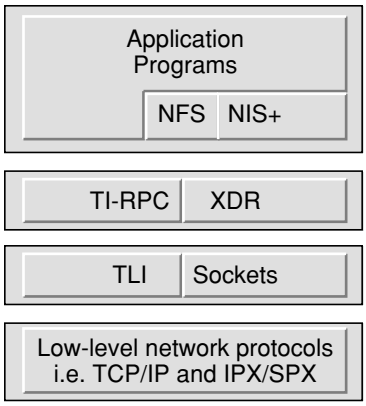


図 1–1　ONC+ 分散コンピューティングプラットフォーム

ONC+ の概要

ONC+ は、さまざまな技術、サービス、およびツールから構成されています。旧バージョンの ONC サービスとは下位互換性があり、相互に運用することができます。この節では主な構成要素について説明します。このマニュアルではプログラミング機能を必要とする技術について扱っています。

TI-RPC

トランスポート独立遠隔手続き呼び出し (TI-RPC) は、UNIX System V リリース 4 (SVR4) の一部として Sun と AT&T により開発されました。分散プログラムの 1 つのバイナリバージョンを複数のトランスポート上で実行することで、RPC アプリケーションをトランスポートに依存しないようにします。以前はトランスポートに固有な RPC であったため、コンパイル時にトランスポートが結合され、プログラムを再構築しないかぎりそのアプリケーションは他のトランスポートでは使用できませんでした。TI-RPC を使用すると、システム管理者がネットワーク構成ファイルを更新し、プログラムを再起動すれば、アプリケーションは新しいトランスポートを使用できます。バイナリアプリケーションを変更する必要はありません。

XDR

外部データ表現 (XDR: External Data Representation) はアーキテクチャに依存しないデータ表現方法です。データのバイト順序、データ型のサイズ、表現方法、異なるアーキテクチャ間のデータの並び方などの違いを解決します。XDR を使用するアプリケーションは、異機種ハードウェアシステム間でデータを交換できます。

NFS

NFS は Sun の分散コンピューティングファイルシステムであり、異機種ネットワーク上の遠隔ファイルシステムへの透過的なアクセスを提供します。NFS によりユーザーは PC、ワークステーション、メインフレーム、そしてスーパーコンピュータ間でファイルを共有できます。同じネットワークに接続されていれば、ファイルはユーザーのデスクトップ上にあるかのように表示されます。NFS 環境では Kerberos 認証、マルチスレッド、ネットワークロックマネージャ、自動マウンタなどの機能を利用できます。

NFS はプログラミング機能を持っていないため、このマニュアルでは説明していません。NFS V.3 の仕様については、anonymous ftp で入手できます。詳細は xviiページの「関連マニュアル 」を参照してください。

NIS+

NIS+ は Solaris 上において大規模な組織で使用できるネームサービスです。ホスト名、ネットワークアドレス、ユーザー名について、拡張性があり、また安全な基本情報を提供します。ネットワーク資源の追加、削除、再配置をサーバーで行うことによって、大規模なマルチベンダーのクライアント／サーバーネットワークを簡単に管理できるように設計されています。NIS+ のデータベース情報を変更すると、それはネットワーク全体の複製サーバーへ自動的にすぐに伝達されます。これによってシステムの稼働時間や性能に影響を与えません。NIS+ にはセキュリティ機能が組み込まれています。権利のないユーザーやプログラムは、ネームサービス情報を読み取ったり、変更したり、削除することはできません。

パート II　遠隔手続き呼び出し (RPC)

Part 2 では RPC について説明します。

- 第 2 章
- 第 3 章
- 第 4 章

第 2 章

TI-RPC 入門

この章では Sun RPC としても知られている TI-RPC について概要を説明します。RPC に初めて接するユーザーに役立つ情報を記載しています (用語の定義は、用語集を参照してください)。

TI-RPC の概要

TI-RPC はクライアントサーバーをベースにした分散型アプリケーションを構築するための強力な技術です。従来のローカルの手続き呼び出しの概念を拡張し、呼び出された手続きが呼び出す手続きと同じアドレス空間に存在する必要がないようにしています。2 つのプロセスが同じシステム上に存在することもあり、また、ネットワーク上で接続された異なるシステム上に存在する場合もあります。

RPC を使用すると、分散型アプリケーションを作成するプログラマはネットワークとの詳細なインタフェースを意識する必要がありません。 RPC はトランスポート層に依存しないため、データ通信の物理的および論理的な機構からアプリケーション

を切り離して作成することができ、したがって、アプリケーションはさまざまなトランスポートを使用できます。

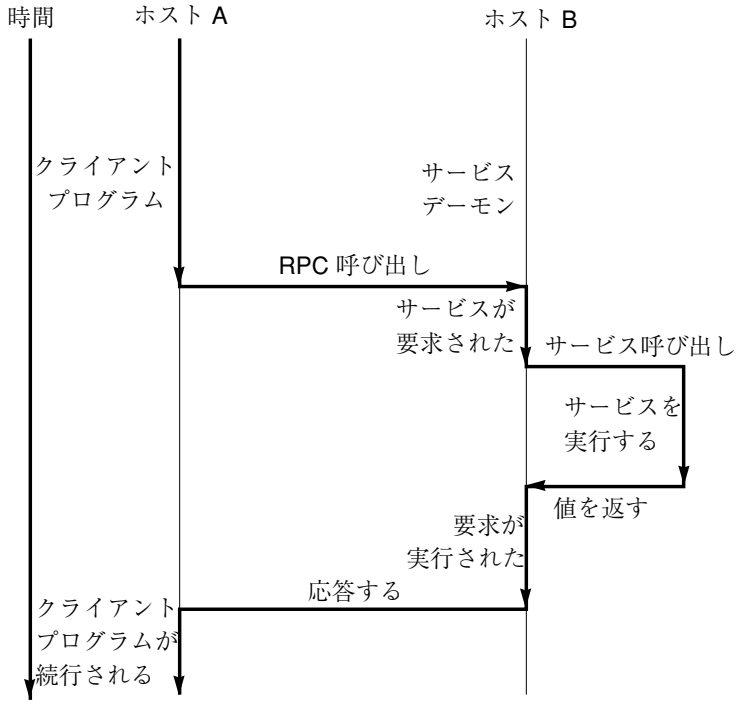


図 2–1　遠隔手続き呼び出しの動作

RPC は関数呼び出しに類似したものです。RPC を実行すると、呼び出し時の引数が遠隔手続きに渡され、呼び出し側は遠隔手続きからの応答を待ちます。

図 2–1 に、2 つのネットワーク上のシステム間でのRPC 呼び出し時に実行される動作のフローを示します。クライアントは、サーバーに要求を送信して応答を待つ手続き呼び出しを行います。応答が受信されるかまたはタイムアウトになるまで、スレッドの実行は停止されます。要求が届くと、サーバーは要求されたサービスを実行するディスパッチルーチンを呼び出し、その結果をクライアントに返します。RPC 呼び出しが終了すると、クライアントはプログラムの続きを実行します。

RPC はネットワークアプリケーションをサポートします。TI-RPC は TCP/IP のようなネットワーク機構上で実行されます。その他の標準の RPC としては、OSF DCE (Apollo の NCS システムをベースにしています)、Xerox Courier、Netwise があります。

TI-RPC の問題

特定の RPC を実装する場合には次の注意点があります。

- パラメータと結果が渡される方法
- 結合が行われる方法
- トランスポートプロトコルが使用される方法
- 呼び出しセマンティクス
- 使用されるデータ表現

パラメータの引き渡し

TI-RPC では 1 つのパラメータをクライアントからサーバーに渡すことができます。複数のパラメータが必要なときは、1 つの要素とみなされる 1 つの構造体に含めて渡されます。サーバーからクライアントに渡される情報は、関数の戻り値として渡されます。サーバーからクライアントにパラメータリストを通して情報を戻すことはできません。

結合

クライアントは、利用したいサービスの使用方法を知っていなければなりません。サーバーのホスト名を知ることと、実際のサーバーのプロセスに接続することが必要です。各ホストでは、`rpcbind` と呼ばれるサービスが RPC サービスを管理します。TI-RPC は `hosts`ファイル、NIS+、DNS などのホストネームサービスを使用してホストの位置を確認します。

トランスポートプロトコル

トランスポートプロトコルは、クライアントとサーバーとの間で呼び出しおよび返答メッセージがどのように送信されるかを指定します。TS-RPC はトランスポートプロトコルとして TCP と UDP を使用しますが、現在の TI-RPC バージョンはトランスポートに依存しません。つまり、TI-RPC は Solaris 2.*x* がサポートする任意のトランスポートプロトコルで動作します。

呼び出しセマンティクス

呼び出しセマンティクスは、遠隔手続きの実行に関し、特にその手続きが何回実行されたかについてクライアントが仮定することに関係があります。これはエラー条件を扱う場合に重要です。この場合、「1 回」、「多くても 1 回」、「少なくとも 1 回」の 3 つのセマンティクスがあります。ONC+ では「少なくとも 1 回」のセマンティクスを提供します。遠隔で呼び出される手順は一貫しています。つまり、たとえ数回にわたって呼び出されても同じ結果を返す必要があります。

データ表現

データ表現とは、プロセス間でパラメータと結果が渡されるときに使用されるフォーマットのことです。さまざまなシステムアーキテクチャ上で RPC が機能するためには、標準データ形式が必要です。TI-RPC では、標準データ形式として外部データ表現 (XDR: external Data Representation) を使用します。XDR はマシンに依存しないデータ形式と符号化のためのプロトコルです。TI-RPC では、XDR を使用することによって、各ホストのバイト順序や構造体の配置方法にわずらわされることなく、任意のデータ構造を扱うことができます。XDR の詳細については、付録 C および 付録 A を参照してください。

プログラム、バージョン、手続き番号

遠隔手続きは次の 3 つの要素によって一意に識別されます。

(プログラム番号、バージョン番号、手続き番号)

プログラム番号とは、関連する遠隔手続きがグループ化された 1 つのプログラムを示します。プログラム内の各手続きは固有の手続き番号を持っています。

プログラムは 1 つまたは複数のバージョンを持つ場合があります。各バージョンは遠隔で呼び出せる手続きの集まりです。バージョン番号を利用することにより、1 つの RPC プロトコルの複数のバージョンを同時に使用できます。

各バージョンには遠隔で呼び出せる多くの手続きが含まれます。各手続きは、手続き番号を持っています。

242ページの「プログラムと手続き番号」では、値の範囲と意味を示し、プログラム番号を RPC プログラムに割り当てる方法を説明しています。RPC サービス名とプ

ログラム番号との対応リストは、rpc ネットワークデータベースの /etc/rpc にあります。

インタフェースルーチンの概要

RPC が提供するサービスには、さまざまなレベルのアプリケーションインタフェースがあります。レベルごとに制御の度合いが異なるため、インタフェースのコーディング量との兼ね合いで適当なレベルを使用してください。この節では、制御の度合いとプログラムの複雑さの順に、各レベルで利用できるルーチンについて要約します。

単純インタフェースのルーチン

単純インタフェースは、使用するトランスポートタイプだけを指定して、他のマシン上のルーチンを遠隔手続き呼び出しにより実行します。ほとんどのアプリケーションで、このレベルのルーチンを使用します。説明とコード例は、68ページの「単純インタフェース」を参照してください。

表 2–1 RPC ルーチン - 単純レベル

ルーチン	説明
rpc_reg()	手続きを RPC プログラムとして、指定したタイプのトランスポートすべてに登録する。
rpc_call()	指定した遠隔ホスト上の、指定した手続きを遠隔呼び出しする。
rpc_broadcast()	指定したタイプのトランスポートすべてに呼び出しメッセージをブロードキャストする。

標準インタフェースのルーチン

標準インタフェースはトップレベル、中間レベル、エキスパートレベル、ボトムレベルの 4 つのレベルにわけられます。開発者はこれらのインタフェースを使用し

て、トランスポートの選択、エラーへの応答または要求の再送まで待つ時間の指定などのパラメータをかなり詳細に制御できます。

トップレベルのルーチン

トップレベルのインタフェースも簡単に使用できますが、RPC 呼び出しを行う前にクライアントハンドルを作成し、RPC 呼び出しを受ける前にサーバーハンドルを作成しなければなりません。アプリケーションをすべてのトランスポート上で実行したい場合は、このインタフェースを使用してください。このルーチンの使用方法とコード例は、78ページの「トップレベルのインタフェース」を参照してください。

表 2–2 RPC ルーチン - トップレベル

ルーチン	説明
`clnt_create()`	汎用のクライアント作成ルーチン。このルーチンは、サーバーの位置と、使用するトランスポートのタイプを指定して呼び出す。
`clnt_create_timed()`	`clnt_create()` に似ているが、クライアントの作成を試みる間、各トランスポートタイプに許される最長時間を指定できる。
`svc_create()`	指定したタイプのトランスポートすべてに対しサーバーハンドルを作成する。このルーチンは、使用するディスパッチ関数を `svc_create()` に指定して呼び出す。
`clnt_call()`	要求をサーバーに送信するための手続きをクライアント側から呼び出すルーチン。

中間レベルのインタフェース

RPC の中間レベルのインタフェースを使用すると、通信を詳細に制御できます。このような下位レベルのインタフェースを使用すると、プログラムは複雑になりますが、効率はよくなります。中間レベルでは特定のトランスポートを指定できます。このルーチンの使用方法とコード例は、82ページの「中間レベルのインタフェース」を参照してください。

表 2–3　RPC ルーチン - 中間レベル

ルーチン	説明
clnt_tp_create()	指定したトランスポートに対するクライアントハンドルを作成する。
clnt_tp_create_timed()	clnt_tp_create() に似ているが、許される最長時間を指定できる。
svc_tp_create()	指定したトランスポートに対するサーバーハンドルを作成する。
clnt_call()	要求をサーバーに送信するための手続きをクライアント側から呼び出す。

エキスパートレベルのインタフェース

エキスパートレベルには、トランスポートに関連するパラメータを指定するさまざまなルーチンがあります。このルーチンの使用方法とコード例は、85ページの「エキスパートレベルのインタフェース」を参照してください。

表 2–4　RPC ルーチン - エキスパートレベル

ルーチン	説明
clnt_tli_create()	指定したトランスポートに対するクライアントハンドルを作成する。
svc_tli_create()	指定したトランスポートに対するサーバーハンドルを作成する。
rpcb_set()	rpcbind デーモンを呼び出して、RPC サービスとネットワークアドレスとのマップを作成する。
rpcb_unset()	rpcb_set() で作成したマップを削除する。
rpcb_getaddr()	rpcbind デーモンを呼び出して、指定した RPC サービスのトランスポートアドレスを取り出す。
svc_reg()	指定したプログラム番号とバージョン番号のペアを、指定したディスパッチルーチンに関連付ける。

表 2–4　RPC ルーチン - エキスパートレベル　続く

ルーチン	説明
svc_unreg()	svc_reg() で設定した関連付けを解除する。
clnt_call()	要求をサーバーに送信するための手続きをクライアント側から呼び出す。

ボトムレベルのインタフェース

ボトムレベルには、トランスポートを完全に制御することができるルーチンがあります。これらのルーチンについては、90ページの「ボトムレベルのインタフェース」を参照してください。

表 2–5　RPC ルーチン - ボトムレベル

ルーチン	説明
clnt_dg_create()	非接続型トランスポートを使用して、指定した遠隔プログラムに対する RPC クライアントハンドルを作成する。
svc_dg_create()	非接続型トランスポートを使用して、RPC サーバーハンドルを作成する。
clnt_vc_create()	接続型トランスポートを使用して、指定した遠隔プログラムに対する RPC クライアントハンドルを作成する。
svc_vc_create()	接続型トランスポートを使用して、RPC サーバーハンドルを作成する。
clnt_call()	要求をサーバーに送信するための手続きをクライアント側から呼び出す。

ネットワーク選択

特定のトランスポートまたはトランスポートタイプで実行されるプログラムを書くことができます。あるいは、システムが選択するトランスポート、またはユーザー

が選択するトランスポート上で実行されるプログラムを書くこともできます。ネットワークの選択では、/etc/netconfig データベースと環境変数 NETPATH を使用します。これにより希望するトランスポートを指定したり、また可能であればアプリケーションがそれを使用できます。指定したトランスポートが不適当な場合、アプリケーションは自動的に適切な機能を持つ他のトランスポートを試みます。

/etc/netconfig には、ホストで使用できるトランスポートが記載されていて、タイプによって識別されます。NETPATH はオプションで、ユーザーはこれを使用してトランスポートを指定したり、/etc/netconfig にあるリストからトランスポートを選択したりできます。NETPATH を設定するとユーザーは、アプリケーションが利用できるトランスポートを試みる順序を指定できます。NETPATH を設定しないと、システムはデフォルトで /etc/netconfig に指定されているすべての選択可能なトランスポート (visible (可視) トランスポート) について、ファイルに現れる順番で選択を試みます。

ネットワーク選択についての詳細は、『*Transport Interfaces Programming Guide*』または getnetconfig(3N) と netconfig(4) のマニュアルページを参照してください。

RPC では、選択可能なトランスポートを次のタイプに分類します。

表 2–6 nettype パラメータ

値	説明
NULL	netpath と同じ。
visible	/etc/netconfig ファイルのエントリのうち、可視フラグ (v フラグ) の付いたトランスポートが使用される。
circuit_v	visible と同じ。ただし、接続型トランスポートに限定される。/etc/netconfig ファイルに記載された順に選択される。
datagram_v	visible と同じ。ただし、非接続型トランスポートに限定される。
circuit_n	接続型トランスポートが、NETPATH で設定された順に使用される。
datagram_n	非接続型トランスポートが、NETPATH で設定された順に使用される。

表 2–6　nettype パラメータ　続く

値	説明
udp	インターネット・ユーザーデータグラム・プロトコル (UDP) が指定される。
tcp	インターネット・トランスミッション・コントロール・プロトコル (TCP) が指定される。

トランスポート選択

RPC サービスは、接続型と非接続型の両方のトランスポートでサポートされています。トランスポート選択は、アプリケーションの性質で決まります。

アプリケーションが次のすべてに当てはまる場合は、非接続型トランスポートの方が適当です。

- 手続きの呼び出しによってサーバー内部の状態や関連データが変更されない。
- 引数と戻り値のサイズがトランスポートのパケットサイズより小さい。
- サーバーは非常に多くのクライアントを扱う必要がある。非接続型サーバーは各クライアントの状態データを保持する必要がないため、本質的に数多くのクライアントを処理することができる。これに対して、接続型サーバーではオープンしているクライアント接続すべての状態データを保持するため、処理できるクライアント数はホストの資源によって制限される。

アプリケーションが次のどれかに当てはまる場合は、接続型トランスポートの方が適当です。

- 非接続型トランスポートと比較して、アプリケーションで接続の確立により多くの手間をかけることができる。
- 手続きの呼び出しによってサーバー内部の状態や関連データが変更される可能性がある。
- 引数または戻り値のサイズがデータグラムパケットの最大サイズより大きい。

名前からアドレスへの変換

各トランスポートには固有の変換ルーチンがあり、汎用ネットワークアドレス (トランスポートアドレスを文字列で表現したもの) とローカルアドレスとの相互変換を行います。RPC システム内 (例えば、rpcbind とクライアントの間) では、汎用アドレスが使用されます。各トランスポートには、名前からアドレスへの変換ルーチンの入った実行時リンクライブラリがあります。表 2–7 に、主な変換ルーチンを示します。

上の各ルーチンについての詳細は、netdir(3N) のマニュアルページと『*Transport Interfaces Programming Guide*』を参照してください。どのルーチンの場合も、netconfig 構造体が名前からアドレスへの変換のコンテキストを提供していることに注意してください。

表 2–7 名前からアドレスへの変換ルーチン

netdir_getbyname()	ホストとサービスのペア（たとえば、server1、rpcbind) と netconfig 構造体から、netbuf アドレスのセットに変換する。netbuf は TLI 構造体で、実行時にトランスポート固有のアドレスが入る。
	netbuf アドレスと netconfig 構造体から、ホストとサービスのペアに変換する。
uaddr2taddr()	汎用アドレスと netconfig 構造体から、netbuf アドレスに変換する。
taddr2uaddr ()	netbuf アドレスと netconfig 構造体から、汎用アドレスに変換する。

アドレスルックアップサービス

トランスポートサービスにはアドレスルックアップサービスは含まれていません。トランスポートサービスはネットワーク上のメッセージ転送だけを行います。クライアントプログラムは、使用するサーバープログラムのアドレスを知る必要があります。旧バージョンの SunOS では、portmap デーモンがそのサービスを実行していました。現バージョンでは rpcbind デーモンを使用します。

RPC では、ネットワークアドレスの構造を考慮する必要がありません。RPC では、NULL で終わる ASCII 文字列で指定される汎用アドレスを使用するためです。RPC はトランスポート固有の変換ルーチンを使用して、汎用アドレスをローカルトランスポートアドレスに変換します。変換ルーチンについての詳細は、netdir(3N) と rpcbind(3N) のマニュアルページを参照してください。

rpcbind の機能を次に示します。

- アドレス登録を追加する。
- アドレス登録を削除する。
- 指定されたプログラム番号、バージョン番号、トランスポートで決まるアドレスを取り出す。
- 登録リスト全体を取り出す。
- クライアントのための遠隔呼び出しを実行する。
- 時刻を返す。

アドレス登録

rpcbind が RPC サービスをアドレスにマップするので、rpcbind 自身のアドレスはその利用者に知られていなければなりません。全トランスポートの名前からアドレスへの変換ルーチンが、トランスポートが使用する各タイプのためのアドレスを保有している必要があります。たとえば、インターネットドメインでは、TCPでも UDP でも rpcbind のポート番号は 111 です。rpcbind は、起動されるとホストがサポートしている全トランスポートに自分のアドレスを登録します。RPC サービスのうち rpcbind だけは、前もってアドレスが知られていなければなりません。

rpcbind は、ホストがサポートしている全トランスポートに RPC サービスのアドレスを登録して、クライアントがそれらを使用できるようにします。各サービスは、rpcbind デーモンでアドレスを登録し、クライアントから利用できるようになります。そこで、サービスのアドレスが rpcinfo(1M) と rpcbind(3N) のマニュアルページで指定されているライブラリルーチンを使用するプログラムとで利用可能になります。クライアントやサーバーからは RPC サービスのネットワークアドレスを知ることはできません。

クライアントプログラムとサーバープログラム、および、クライアントホストとサーバーホストとは通常別のものですが、同じであってもかまいません。サーバープログラムもまたクライアントプログラムになることができます。あるサーバーが別の rpcbind サーバーを呼び出す場合は、クライアントとして呼び出したことになります。

クライアントが遠隔プログラムのアドレスを調べるには、ホストの rpcbind デーモンに RPC メッセージを送信します。サービスがホスト上にあれば、デーモンは RPC 応答メッセージにアドレスを入れて返します。そこで、クライアントプログラムは RPC メッセージをサーバーのアドレスに送ることができます。(クライアントプログラムから rpcbind を頻繁に呼び出さなくて済むように、最後に呼び出した遠隔プログラムのネットワークアドレスを保存しておきます)。

rpcbind の RPCBPROC_CALLIT 手続きを使用すると、クライアントはサーバーのアドレスがわからなくても遠隔手続きを呼び出すことができます。クライアントは目的の手続きのプログラム番号、バージョン番号、手続き番号、引数を RPC 呼び出しメッセージで引き渡します。rpcbind は、アドレスマップから目的の手続きのアドレスを探し出し、RPC 呼び出しメッセージにクライアントから受け取った引数を入れて、その手続きに送信します。

目的の手続きから結果が返されると、RPCBPROC_CALLIT はクライアントプログラムにその結果を引き渡します。そのとき、目的の手続きの汎用アドレスも同時に渡されますので、次からはクライアントが直接その手続きを呼び出すことができます。

RPC ライブラリは rpcbind の全手続きのインタフェースを提供します。RPC ライブラリの手続きには、クライアントとサーバーのプログラムのために rpcbind を自動的に呼び出すものもあります。詳細については、付録 B を参照してください。

RPC 情報の取り出し

rpcinfo は、rpcbind で登録した RPC の最新情報を取り出すユーティリティです。rpcbind または portmap ユーティリティと共に rpcinfo を使用して、あるホストに登録されたすべての RPC サービスの汎用アドレスとトランスポートを知ることができます。指定したホスト上で指定したプログラムの特定バージョンを呼び出して、応答が返ったかどうかを調べることができます。詳細については、rpcinfo(1M) のマニュアルページを参照してください。

第 3 章

rpcgen プログラミングガイド

この章では、rpcgen ツールについて紹介します。コード例および使用可能なコンパイル時のフラグの使用方法を記載したチュートリアルです。この章で使用する用語の定義については、用語集を参照してください。

rpcgen の概要

rpcgen ツールは、RPC 言語で書かれたソースコードをコンパイルして、遠隔プログラムインタフェースモジュールを生成します。RPC 言語は構文も構造も C 言語に似ています。rpcgen は C 言語ソースモジュールを生成しますので、次に C コンパイラでコンパイルします。

デフォルトでは、rpcgen は次のコードを生成します。

- サーバーとクライアントに共通の定義の入ったヘッダーファイル
- ヘッダーファイルで定義されているデータ型を変換するための XDR ルーチン
- サーバー側のスタブプログラム
- クライアント側のスタブプログラム

オプションを指定すれば、rpcgen で次のことを行うことができます。

- さまざまなトランスポートの選択
- サーバーのタイムアウトの指定
- マルチスレッド対応サーバースタブの生成
- main プログラム以外のサーバー側スタブプログラムの生成
- C 形式で引数を受け渡す ANSI C 準拠のコードの生成
- 権限をチェックしてサービスルーチンを呼び出す RPC ディスパッチ

rpcgen を使用すると、下位レベルのルーチンを作成する手間が省けるのでアプリケーション開発時間を大幅に短縮できます。rpcgen の出力コードとユーザー作成コードとは、簡単にリンクできます。(rpcgen を使用しないで RPC プログラムを作成する方法については、第 4 章を参照してください)。

SunOS 5.X の機能

この節では、SunOS 4.*x* では提供されなかった機能で、SunOS 5.x rpcgen コード生成プログラムで追加されたものについて説明します。

テンプレートの生成

rpcgen では、クライアント側、サーバー側、および makefile の各テンプレートを生成することができます。オプションのリストについては、41ページの「クライアント側とサーバー側のテンプレート」を参照してください。

C 形式モード

rpcgen には、C 形式モードとデフォルトモードという 2 つのコンパイルモードがあります。C 形式モードでは、引数は構造体へのポインタではなく値で渡されます。また、C 形式モードでは複数の引数を渡すこともできます。デフォルトモードは旧バージョンと同じです。両方のモードのコード例については、42ページの「C 形式モード」を参照してください。

マルチスレッド対応コード

現バージョンでは、マルチスレッド環境で実行可能なマルチスレッド対応コードを生成することができるようになりました。デフォルトでは、rpcgen によって生成されたコードはマルチスレッド対応ではありません。詳細およびコード例については、45ページの「マルチスレッド対応のコード」を参照してください。

マルチスレッド自動モード

rpcgen では、マルチスレッド自動モードで実行するマルチスレッド対応サーバースタブを生成します。定義およびコーディング例については、52ページの「自動マルチスレッド対応モード」を参照ください。

ライブラリの選択

rpcgen では、TS-RPC ライブラリか TI-RPC ライブラリのどちらかを使用してコードを生成します。53ページの「TI-RPC または TS-RPC のライブラリ選択」を参照してください。

ANSI C 準拠のコード

rpcgen では、ANSI C に準拠したコードを生成します。また、ANSI C 準拠のコードは、SPARCompiler™ C++ 3.0 環境で使用することができます。53ページの「ANSI C に準拠したコードの生成」を参照してください。

rpcgen チュートリアル

rpcgen を使用すると、分散型アプリケーションを簡単に作成できます。サーバー側手続きは、手続き呼び出し規約に準拠した言語で記述します。サーバー側手続きは、rpcgen によって生成されたサーバースタブとリンクして、実行可能なサーバープログラムを形成します。クライアント側手続きも同様に記述およびリンクします。

この節では、rpcgen を使用した基本的なプログラミング例を示します。また、rpcgen(1) のマニュアルページを参照してください。

ローカル手続きを遠隔手続きに変換

単一のコンピュータ環境で実行されるアプリケーションを、ネットワーク上で実行する分散型アプリケーションに変更する場合を考えます。次の例で、システムコンソールにメッセージを表示するプログラムを分散型アプリケーションに変換する方法を、ステップ別に説明します。変換前のプログラムコード例 3–1 を次に示します。

コード例 3–1 シングルコンピュータ用の printmsg.c

```
/* printmsg.c: コンソールにメッセージを表示します。
 */
#include <stdio.h>
main(argc, argv)
 int argc;
 char *argv[];
{
 char *message;

 if (argc != 2) {
  fprintf(stderr, "usage: %s <message>\n",
     argv[0]);
  exit(1);
 }
 message = argv[1];
 if (!printmessage(message)) {
  fprintf(stderr,"%s: couldn't print your
     message\n",argv[0]);
 exit(1);
 }
 printf("Message Delivered!\n");
 exit(0);
}

/* コンソールにメッセージを表示する。
 * メッセージを表示できたかどうかを示すブール値を返す。*/

printmessage(msg)
 char *msg;
{
 FILE *f;

 f = fopen("/dev/console", "w");
 if (f == (FILE *)NULL) {
  return (0);
 }
 fprintf(f, "%s\n", msg);
 fclose(f);
 return(1);
}
```

このプログラムをシングルコンピュータ上で使用するときは、次のコマンドでコンパイルして実行できます。

```
$ cc printmsg.c -o printmsg
$ printmsg "Hello, there."
Message delivered!
$
```

printmessage() 関数を遠隔手続きに変換すると、ネットワーク上のどこからでも実行できるようになります。rpcgen を使用すると、簡単にこのような変換を実行できます。

最初に、手続きを呼び出すときのすべての引数と戻り値のデータ型を決定します。printmessage() の引数は文字列で、戻り値は整数です。このようなプロトコル仕様を RPC 言語で記述して、遠隔手続きとしての printmessage() を作成することができます。RPC 言語でこのプロトコル仕様を記述したソースコードは次のようになります。

```
/* msg.x: メッセージを表示する遠隔手続きのプロトコル */
 program MESSAGEPROG {
     version PRINTMESSAGEVERS {
         int PRINTMESSAGE(string) = 1;
   } = 1;
} = 0x20000001;
```

遠隔手続きは常に遠隔プログラムの中で宣言されます。上のコードでは、PRINTMESSAGE という手続きが 1 つだけ含まれた遠隔プログラムが 1 つ宣言されています。この例では、PRINTMESSAGE という手続きが、MESSAGEPROG という遠隔プログラム内の手続き 1、バージョン 1 として宣言されています。遠隔プログラムのプログラム番号は、0x20000001 です。(プログラム番号の指定方法については、付録 B を参照してください)。既存の手続きが変更されたり新規手続が追加されたりして、遠隔プログラムの機能が変更されると、バージョン番号が 1 つ増やされます。遠隔プログラムで複数のバージョンを定義することもできますし、1 つのバージョンで複数の手続きを定義することもできます。

プログラム名も手続き名も共に大文字で宣言していることに注意してください。

また、引数のデータ型を C 言語で書くときのように char * としないで stringとしていることにも注意してください。これは、C 言語で char * と指定すると、文

字型配列とも、単一の文字へのポインタとも解釈できて不明確なためです。RPC 言語では、NULL で終わる文字型配列は string 型で宣言します。

更に次の 2 つのプログラムを書く必要があります。

- 遠隔手続き自体
- 遠隔手続きを呼び出すクライアント側のメインプログラム

コード例 3–2 には、コード例 3–1 の手続きを PRINTMESSAGE という遠隔手続きに変更したものを示します。

コード例 **3–2** RPC バージョンの printmsg.c

```
/*
 * msg_proc.c: 遠隔手続きバージョンの printmessage
 */
#include <stdio.h>
#include "msg.h"    /* rpcgen が生成 */

int *
printmessage_1(msg, req)
 char **msg;
 struct svc_req *req;  /* 呼び出しの詳細 */
{
 static int result;   /* 必ず static で宣言 */
 FILE *f;

f = fopen("/dev/console", "w");
if (f == (FILE *)NULL) {
 result = 0;
 return (&result);
}
fprintf(f, "%s\n", *msg);
fclose(f);
result = 1;
return (&result);
}
```

遠隔手続き printmessage_1() と、ローカル手続き printmessage() の宣言は次の 4 つの点で異なることに注意してください。

1. 引数が文字へのポインタではなく、文字配列へのポインタになっています。–N オプションを使用しない遠隔手続きの場合は、引数自体が渡されるのではなく、常に引数へのポインタが渡されるからです。–N オプションを指定しなければ、遠隔手続きの呼び出しで引数が 1 つしか渡されません。複数の引数が必要な場合は、引数を struct 型にして渡す必要があります。

2. 引数が 2 つあります。第 2 引数には、関数呼び出しのときのコンテキスト、すなわち、プログラム、バージョン、手続きの番号、raw および canonical の認証、SVCXPRT 構造体へのポインタが入っています。 SVCXPRT 構造体にはトランスポート情報が入っています。呼び出された手続きが要求されたサービスを実行するときに、これらの情報が必要になる場合があります。
3. 戻り値は、整数そのものではなく整数へのポインタになっています。–N オプションを指定しない遠隔手続きの場合は、戻り値自体ではなく戻り値へのポインタが返されるためです。–M (マルチスレッド) オプション または –A (自動モード) オプションが使用されている限り、戻り値は static で宣言します。戻り値を遠隔手続きのローカル値にしてしまうと、遠隔手続きがリターンした後、サーバー側スタブプログラムからその値を参照することができなくなります。 –M および –A を使用している場合は、戻り値へのポインタは第 3 引数として手続きに渡されるため、戻り値手続きで宣言されません。
4. 手続き名に _1 が追加されています。一般に rpcgen が遠隔手続き呼び出しを生成するときは、次のように手続き名が決められます。プログラム定義で指定した手続き名 (この場合は PRINTMESSAGE) はすべて小文字に変換され、下線 (_) とバージョン番号 (この場合は 1) が追加されます。このように手続き名が決定されるので、同じ手続きの複数バージョンが使用可能になります。

コード例 3–3 には、この遠隔手続きを呼び出すクライアント側メインプログラムを示します。

コード例 3–3 printmsg.c を呼び出すクライアント側プログラム

```
/*
 * rprintmsg.c: printmsg.c の RPC 対応バージョン
 */
#include <stdio.h>
#include "msg.h"   /* rpcgen が生成 */

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

 if (argc != 3) {
  fprintf(stderr, "usage: %s host
     message\n", argv[0]);
  exit(1);
 }
```

(続く)

```
 server = argv[1];
 message = argv[2];
 /*
 * コマンドラインで指定したサーバーの
 * MESSAGEPROG の呼び出しで使用する
 * クライアント「ハンドル」を作成
 */
 clnt = clnt_create(server, MESSAGEPROG,
        PRINTMESSAGEVERS,
        "visible");
 if (clnt == (CLIENT *)NULL) {
  /*
   * サーバーとの接続確立に失敗したため、
   * エラーメッセージを表示して終了
   */
  clnt_pcreateerror(server);
  exit(1);
 }

 /*
  * サーバー上の遠隔手続き printmessage を呼び出す
  */
 result = printmessage_1(&message, clnt);
 if (result == (int *)NULL) {
  /*
   * サーバーの呼び出しでエラーが発生したため、
   * エラーメッセージを表示して終了
   */
  clnt_perror(clnt, server);
  exit(1);
 }
/*
 * 遠隔手続き呼び出しは正常終了
 */
 if (*result == 0) {
  /*
   * サーバーがメッセージの表示に失敗したため、
   * エラーメッセージを表示して終了
   */
  fprintf(stderr,
  "%s: could not print your message\n",argv[0]);
  exit(1);
 }

/*
 * サーバーのコンソールにメッセージが出力された
 */
 printf("Message delivered to %s\n",
    server);
 clnt_destroy( clnt );
 exit(0);
```

(続く)

```
}
```

この コード例 3–3 では、次の点に注意してください。

1. 最初に、RPC ライブラリルーチン clnt_create() を呼び出してクライアントハンドルを作成しています。クライアントハンドルは、遠隔手続きを呼び出すスタブルーチンに引き渡されます。(これ以外にもクライアントハンドルを作成する方法があります。詳細については、第 4 章を参照してください)。クライアントハンドルを使用する遠隔手続き呼び出しがすべて終了したら、clnt_destroy() を使用してそのクライアントハンドルを破棄し、システム資源を無駄に使用しないようにします。
2. clnt_create() の最後の引数に "visible" を指定して、/etc/netconfig で visible と指定したすべてのトランスポートを使用できるようにします。詳細については、/etc/netconfig ファイルと『*Transport Interfaces Programming Guide*』を参照してください。
3. 遠隔手続き printmessage_1() の呼び出しは、第 2 引数として挿入されたクライアントハンドルを除いて、msg_proc.c で宣言された通りに実行されています。戻り値も値ではなく、値へのポインタで返されています。
4. 遠隔手続き呼び出しのエラーには、RPC 自体のエラーと、遠隔手続きの実行中に発生したエラーの 2 種類があります。最初のエラーの場合は、遠隔手続き printmessage_1() の戻り値が NULL になります。2 つめのエラーの場合は、アプリケーションによってエラーの返し方が異なります。この例では、*result によってエラーがわかります。

これまでに示した各コードをコンパイルする方法を次に示します。

```
$ rpcgen msg.x
$ cc rprintmsg.c msg_clnt.c -o rprintmsg -lnsl
$ cc msg_proc.c msg_svc.c -o msg_server -lnsl
```

最初に rpcgen を実行してヘッダーファイル (msg.h)、クライアント側スタブプログラム (msg_clnt.c)、サーバー側スタブプログラム (msg_svc.c) を生成します。次の 2 つのコンパイルコマンドで、クライアント側プログラム rprintmsg とサー

バー側プログラム msg_server が作成されます。C のオブジェクトファイルは、ライブラリ libnsl とリンクする必要があります。ライブラリ libnsl には、RPC と XDR で必要な関数をはじめとするネットワーク関数がすべて含まれています。

この例では、アプリケーションが libnsl に含まれる基本型だけを使用しているので、XDR ルーチンは生成されません。

次に、rpcgen が入力ファイル msg.x から何を生成するかを説明します。

1. msg.h というヘッダーファイルを作成します。msg.h には、他のモジュールで使用できるように MESSAGEPROG、MESSAGEVERS、PRINTMESSAGE の #define 文が入っています。このヘッダーファイルは、クライアント側とサーバー側の両方のモジュールでインクルードする必要があります。
2. クライアント側スタブルーチンを msg_clnt.c というファイルに出力します。このファイルには、クライアントプログラム rprintmsg から呼び出されるルーチン printmessage_1() が 1 つだけ入っています。rpcgen への入力ファイルが FOO.x という名前ならば、クライアント側スタブルーチンは FOO_clnt.c というファイルに出力されます。
3. msg_svc.c の printmessage_1() を呼び出すサーバープログラムを msg_svc.c というファイルに出力します。サーバープログラムのファイル名は、クライアントプログラムのファイル名と同様の方法で決まります。rpcgen への入力ファイルが FOO.x という名前ならば、サーバープログラムは FOO_svc.c というファイルに出力されます。

サーバープログラムが作成されると、遠隔マシン上にインストールして実行することができます。(遠隔マシンが同じ機種の場合は、サーバープログラムをバイナリのままコピーすることができますが、機種が異なる場合は、サーバープログラムのソースファイルを遠隔マシンにコピーして再コンパイルしなければなりません)。遠隔マシンを remote、ローカルマシンを local とすると、遠隔システムのシェルから次のコマンドでサーバープログラムを起動することができます。

```
remote$ msg_server
```

rpcgen が生成したサーバープロセスは、常にバックグラウンドで実行されます。このとき、サーバープログラムにアンパサンド(&) を付けて起動する必要はありません。また、rpcgen が生成したサーバープロセスはコマンドラインからではなく、listen() や inetd() などのポートモニタから起動することもできます。

以降は、local マシン上のユーザーが次のようなコマンドを実行して、remote マシンのコンソールにメッセージを表示できます。

```
local$ rprintmsg remote "Hello, there."
```

rprintmsg を使用すると、サーバープログラム msg_server が起動されているどのシステムにでも (local システムも含む)、コンソールにメッセージを表示できます。

複雑なデータ構造の引き渡し

26ページの「ローカル手続きを遠隔手続きに変換」では、クライアント側とサーバー側のRPC コードの生成について説明しました。rpcgen を使用して、XDR ルーチンを生成することもできます (XDR ルーチンは、ローカルデータ形式と XDR 形式との相互変換を行います)。

遠隔ディレクトリを一覧表示する RPC サービスの全体を コード例 3–4 に示します。rpcgen を使用してスタブルーチンと XDR ルーチンの両方を生成します。

コード例 3–4 RPC 言語で書かれたプロトコル記述ファイル (dir.x)

```
/*
 * dir.x: 遠隔ディレクトリをリストするサービスのプロトコル
 *
 * rpcgen の機能を説明するためのサンプルプログラム
 */

const MAXNAMELEN = 255;      /* ディレクトリエントリの最大長 */
typedef string nametype<MAXNAMELEN>;  /* ディレクトリエントリ */
typedef struct namenode *namelist;   /* リスト形式でリンク */

/* ディレクトリリスト内のノード */
struct namenode {
 nametype name;       /* ディレクトリエントリ名 */
 namelist next;       /* 次のエントリ */
};

/*
 * READDIR の戻り値
 *
 * どこにでも移植できるアプリケーションにするためには、
 * この例のように UNIX の errno を返さないで、
 * エラーコードリストを設定して使用する方がよいでしょう。
 *
 * このプログラムでは、次の共用体を使用して、遠隔呼び出しが
 * 正常終了したか異常終了したかを区別します。
 */

union readdir_res switch (int errno) {
 case 0:
```

(続く)

```
  namelist list;  /* 正常終了: 戻り値はディレクトリリスト */
 default:
  void;      /* エラー発生: 戻り値なし */
};

/* ディレクトリをリストするプログラムの定義 */
program DIRPROG {
 version DIRVERS {
  readdir_res
  READDIR(nametype) = 1;
 } = 1;
} = 0x20000076;
```

上の例の readdir_res のように、RPC 言語のキーワード struct、union、enum を使用して型を再定義することができます。使用したキーワードは、後にその型の変数を宣言するときには指定しません。たとえば、共用体 foo を定義した場合、union foo ではなく foo で宣言します。

rpcgen でコンパイルすると、RPC の共用体は C 言語の構造体に変換されます。RPC の共用体は、キーワード union を使用して宣言してはいけません。

dir.x に対して rpcgen を実行すると、次の4 つのファイル、(1) ヘッダーファイル、(2) クライアント側のスタブルーチン、(3) サーバー側の骨組み、(4) XDR ルーチンの入った dir_xdr.c というファイルが生成されます。(4) のファイルに入っている XDR ルーチンは、宣言されたデータ型を、ホスト環境のデータ形式から XDR 形式に、またはその逆方向に変換します。

rpcgen では、.x ファイルで使用されている RPC 言語の各データ型に対して、データ型名の前にXDR ルーチンであることを示すヘッダー xdr_ が付いたルーチン (たとえば、xdr_int) が libnsl で提供されるものとみなします。.x ファイルにデータ型が定義されていると、rpcgen はそれに対するルーチンを生成します。msg.x のように、.x ソースファイルにデータ型が一切定義されていない場合は、_xdr.c ファイルは生成されません。

.x ソースファイルで、libnsl でサポートされていないデータ型を使用し、.x ファイルではそのデータ型を定義しないこともできます。その場合は、xdr_ ルーチンをユーザーが自分で作成することになります。こうして、ユーザー独自の xdr_ ルーチンを提供することができます。任意のデータ型を引き渡す方法についての詳

細は、第 4 章を参照してください。コード例 3–5 に、サーバー側の READDIR 手続きを示します。

コード例 **3–5**　サーバー側の dir_proc.c の例

```
/*
 * dir_proc.c: 遠隔手続き readdir
 */
#include <dirent.h>
#include "dir.h"                /* rpcgen が生成 */

extern int errno;
extern char *malloc();
extern char *strdup();

readdir_res *
readdir_1(dirname, req)
 nametype *dirname;
 struct svc_req *req;

{
 DIR *dirp;
 struct dirent *d;
 namelist nl;
 namelist *nlp;
 static readdir_res res; /* 必ず static で宣言 */

 /* ディレクトリのオープン */
 dirp = opendir(*dirname);
 if (dirp == (DIR *)NULL) {
  res.errno = errno;
  return (&res);
 }
 /* 直前の戻り値の領域解放 */
 xdr_free(xdr_readdir_res, &res);
/*
 * ディレクトリエントリをすべて取り出す。ここで割り当てたメモリーは、
 * 次に readdir_1 が呼び出されたときに xdr_free で解放する。
 */
 nlp = &res.readdir_res_u.list;
 while (d = readdir(dirp)) {
  nl = *nlp = (namenode *)
       malloc(sizeof(namenode));
  if (nl == (namenode *) NULL) {
   res.errno = EAGAIN;
   closedir(dirp);
   return(&res);
  }
  nl->name = strdup(d->d_name);
  nlp = &nl->next;
 }
 *nlp = (namelist)NULL;
 /* 結果を返す */
 res.errno = 0;
```

(続く)

```
 closedir(dirp);
 return (&res);
}
```

コード例 3–6 に、クライアント側のREADDIR 手続きを示します。

コード例 **3–6**　クライアント側のプログラム (rls.c)

```
/*
 * rls.c: クライアント側の遠隔ディレクトリリスト
 */

#include <stdio.h>
#include "dir.h"                          /* rpcgen が生成 */

extern int errno;

main(argc, argv)
 int argc;
 char *argv[];
{
 CLIENT *clnt;
 char *server;
 char *dir;
 readdir_res *result;
 namelist nl;

 if (argc != 3) {
  fprintf(stderr, "usage: %s host
     directory\n",argv[0]);
  exit(1);
 }
 server = argv[1];
 dir = argv[2];
/*
 * コマンドラインで指定したサーバーの MESSAGEPROG の呼び出しで使用する
 * クライアント「ハンドル」を作成
 */
 cl = clnt_create(server, DIRPROG,
        DIRVERS, "tcp");
 if (clnt == (CLIENT *)NULL) {
  clnt_pcreateerror(server);
  exit(1);
 }
 result = readdir_1(&dir, clnt);
 if (result == (readdir_res *)NULL) {
  clnt_perror(clnt, server);
```

(続く)

```
  exit(1);
 }
/*
 * 遠隔手続き呼び出しは正常終了
 */
 if (result->errno != 0) {
   /*
    * 遠隔システム上のエラー。エラーメッセージを表示して終了
    */
  errno = result->errno;
  perror(dir);
  exit(1);
 }
/*
 * ディレクトリリストの取り出しに成功。ディレクトリリストを表示。
 */
 for (nl = result->readdir_res_u.list;
        nl != NULL;
  nl = nl->next) {
  printf("%s\n", nl->name);
 }
 xdr_free(xdr_readdir_res, result);
 clnt_destroy(cl);
 exit(0);
}
```

この前のサンプルプログラムと同様に、システム名を local と remote とします。ファイルのコンパイルと実行は、次のコマンドで行います。

```
remote$ rpcgen dir.x
remote$ cc -c dir_xdr.c
remote$ cc rls.c dir_clnt.c dir_xdr.o -o rls -lnsl
remote$ cc dir_svc.c dir_proc.c dir_xdr.o -o dir_svc -lnsl
remote$ dir_svc
```

local システムに rls() をインストールすると、次のように remote システム上の/usr/share/lib の内容をリストできます。

```
local$ rls remote /usr/share/libascii
eqnchar
greek
kbd
marg8
tabclr
tabs
tabs4
local$
```

rpcgen が生成したクライアント側のコードは、RPC 呼び出しの戻り値のために割り当てたメモリーを解放しませんので、必要がなくなったら xdr_free() を呼び出してメモリーを解放してください。xdr_free() の呼び出しは free() ルーチンの呼び出しに似ていますが、XDR ルーチン戻り値のアドレスを引き渡す点が異なります。この例では、ディレクトリリストを表示した後で次のように xdr_free() を呼び出しています。

```
xdr_free(xdr_readdir_res,result);
```

注 - xdr_free() を使用して malloc() で割り当てたメモリーを解放します。xdr_free() を使用してメモリーを解放すると、メモリーリークを生じて失敗します。

前処理命令

rpcgen では、C 言語などの前処理命令をサポートしています。rpcgen の入力ファイルに入っている C 言語の前処理命令は、コンパイル前に処理されます。.x ソースファイルでは、標準 C のすべての前処理命令を使用できます。生成する出力ファイルのタイプによって、次の 5 つのシンボルが rpcgen によって定義されます。

rpcgen 入力ファイルのパーセント記号 (%) で始まる行はそのまま出力ファイルに書き出され、その行の内容には影響を及ぼしません。そのとき、意図した位置に出力されるとは限らないため注意が必要です。出力ファイルのどこに書き出されたか確認して、必要ならば編集し直してください。

表 3–1　rpcgen の前処理命令

シンボル	使用目的
RPC_HDR	ヘッダーファイルの出力
RPC_XDR	XDR ルーチンの出力
RPC_SVC	サーバー側スタブプログラムの出力
RPC_CLNT	クライアント側スタブプログラムの出力
RPC_TBL	インデックステーブルの出力

コード例 3–7 に、簡単な rpcgen の例を示します。rpcgen の前処理機能の使用方法に注意してください。

コード例 3–7　時刻プロトコルを記述する rpcgen ソースプログラム

```
/*
 * time.x: 遠隔時刻のプロトコル
 */
program TIMEPROG {
 version TIMEVERS {
   unsigned int TIMEGET() = 1;
 } = 1;
} = 0x20000044;

#ifdef RPC_SVC
%int *
%timeget_1()
%{
% static int thetime;
%
% thetime = time(0);
% return (&thetime);
%}
#endif
```

cpp 命令

rpcgen では、C 言語の前処理機能をサポートしています。rpcgen では、デフォルトで /usr/ccs/lib/cpp を C のプリプロセッサとして使用します。これを使用で

きないときは、/lib/cpp を使用します。これ以外の cpp を含むライブラリを使用するときは、rpcgen の –Y フラグで指定します。

たとえば、/usr/local/bin/cpp を使用するには、次のように rpcgen を起動します

```
rpcgen -Y /usr/local/bin test.x
```

コンパイル時に指定するフラグ

この節では、コンパイル時に使用可能な rpcgen オプションについて説明します。次の表に、この節で説明するオプションを要約します。

表 3–2 rpcgen コンパイル時に指定するフラグ

オプション	フラグ	コメント
テンプレート	–a, –Sc, –Ss, –Sm	表 3–3 を参照
C 形式	–N	新しい形式のモードを呼び出す
ANSI C	–C	–N オプションとともに使用
マルチスレッド対応コード	–M	マルチスレッド環境で使用
マルチスレッド自動モード	–A	このオプションを指定すると、–M も自動的に指定される
TS-RPC ライブラリ	–b	デフォルトは TI-RPC ライブラリ
xdr_inline カウント	–i	デフォルトはパックされた 5 つの要素。他の数字も指定できる

クライアント側とサーバー側のテンプレート

rpcgen で次のフラグを指定して、クライアント側とサーバー側のテンプレートを生成することができます。

表 3–3 rpcgen テンプレート選択フラグ

フラグ	機能
–a	すべてのテンプレートを生成
–Sc	クライアント側のテンプレートを生成
–Ss	サーバー側のテンプレートを生成
–Sm	makefile のテンプレートを生成

生成されたテンプレートファイルを参考にしてプログラムを書くか、テンプレートに抜けている部分を直接書き込んで使用します。rpcgen は、スタブプログラムのほかにこれらのテンプレートファイルを生成します。

ソースプログラム add.x から C 形式モードでサーバー側テンプレートを生成するときは、次のコマンドを実行します。

```
rpcgen -N -Ss -o add_server_template.c add.x
```

生成されたテンプレートファイルは add_server_template.c という名前になります。同じソースプログラム add.x から C 形式モードでクライアント側テンプレートを生成するときは、次のコマンド行を実行します。

```
rpcgen -N -Sc -o add_client_template.c add.x
```

生成されたテンプレートファイルは add_client_template.c という名前になります。同じソースプログラム add.x から makefile テンプレートを生成するときは、次のコマンド行を実行します。

```
rpcgen -N -Sm -o mkfile_template add.x
```

生成されたテンプレートファイルは mkfile_template という名前になります。このファイルを使用して、サーバー側とクライアント側のプログラムをコンパイルできます。次のように、–a フラグを指定した場合は、

```
rpcgen -N -a add.x
```

3 つのテンプレートファイルがすべて生成されます。クライアント側テンプレートは add_client.c、サーバー側テンプレートは add_server.c、makefile テンプレートは makefile.a という名前になります。このうち 1 つでも同名のファイルが存在していれば、rpcgen はエラーメッセージを表示して終了します。

注 - テンプレートファイルを生成する際には、次に rpcgen が実行された時に上書きされないように新しい名前を付けてください。

C 形式モード

–N フラグを指定して rpcgen を起動すると、C 形式モード (Newstyle モードとも呼ばれる) で処理が行われます。このモードでは、引数は値で渡され、複数の引数も構造体にせずに渡すことができます。この機能を使用して、RPC コードを、C 言語やその他の高級言語に近い形式で書くことができます。既存のプログラムや makefile との互換性を保つため、従来モード (標準モード) がデフォルトになっています。次の例では、–N フラグにより利用できる機能を示します。従来モードと C 形式モードの両方のソースモジュールを、コード例 3–8 と コード例 3–9 に示します。

コード例 **3–8**　C 形式モードの add.x

```
/*
 * このプログラムには、2 つの数値を加える手続きが入っています。
 * ここでは、C 形式モードによる引数の引き渡し方法を示します。
 * 関数 add() が 2 つの引数を取ることに注意してください。
 */
program ADDPROG {      /* プログラム番号 */
 version ADDVER {      /* バージョン番号 */
  int add(int, int) = 1;  /* 手続き */
 } = 1;
} = 0x20000199;
```

コード例 **3–9**　デフォルトモードの add.x

```
/*
 * このプログラムには、2 つの数値を加える手続きが入っています。
 * ここでは、デフォルトモードによる引数の引き渡し方法を示します。
 * デフォルトモードの場合、rpcgen は引数を 1 つしか処理しないことに
 * 注意してください。
 */
struct add_arg {
 int first;
 int second;
};
program ADDPROG {      /* プログラム番号 */
 version ADDVER {      /* バージョン番号 */
  int add (add_arg) = 1;  /* 手続き */
 } = 1;
} = 0x20000199;
```

コード例 3–10 からコード例 3–13 には、生成されるクライアント側テンプレートを示します。

コード例 **3–10**　C 形式モードのクライアント側スタブプログラムの例 : add.x

```
/*
 * C 形式のクライアント側メインルーチン。
 * 遠隔 RPC サーバー上の関数 add() を呼び出します。
 */
#include <stdio.h>
#include "add.h"

main(argc, argv)
int argc;
char *argv[];
{
 CLIENT *clnt;
 int *result,x,y;

 if(argc != 4) {
  printf("usage: %s host num1
      num2\n" argv[0]);
  exit(1);
 }
 /*
 * クライアントハンドルの作成 - サーバーに結合
 */
 clnt = clnt_create(argv[1], ADDPROG,
        ADDVER, "udp");
 if (clnt == NULL) {
  clnt_pcreateerror(argv[1]);
```

(続く)

```
  exit(1);
 }
 x = atoi(argv[2]);
 y = atoi(argv[3]);
/* 遠隔手続きの呼び出し: add_1() には、ポインタではなく、
 * 複数の引数が渡されていることに注意してください。
 */
 result = add_1(x, y, clnt);
 if (result == (int *) NULL) {
  clnt_perror(clnt, "call failed:");
  exit(1);
 } else {
  printf("Success: %d + %d = %d\n",
     x, y, *result);
 }
 exit(0);
}
```

コード例 3–11 に、デフォルトモードと C 形式モードとのコードの相違点を示します。

コード例 3–11　デフォルトモードのクライアント側スタブプログラムの例

```
 arg.first = atoi(argv[2]);
 arg.second = atoi(argv[3]);
/*
 * 遠隔手続きの呼び出し -- クライアント側スタブプログラムには、
 * 引数へのポインタを渡さなければならないことに注意してください。
 */
 result = add_1(&arg, clnt);
```

コード例 3–12 に、C 形式モードのサーバー側手続きを示します。

コード例 3–12　C 形式モードのサーバー側プログラムの例

```
#include "add.h"

int *
add_1(arg1, arg2, rqstp)
 int arg1;
 int arg2;
 struct svc_req *rqstp;
{
 static int result;

 result = arg1 + arg2;
 return(&result);
}
```

コード例 3–13 に、デフォルトモードのサーバー側手続きを示します。

コード例 3–13　デフォルトモードのサーバー側スタブプログラムの例

```
#include "add.h"
int *
add_1(argp, rqstp)
 add_arg *argp;
 struct svc_req *rqstp;
{
 static int result;

 result = argp->first + argp->second;
 return(&result);
}
```

マルチスレッド対応のコード

デフォルトでは、rpcgen で生成されるコードはマルチスレッド対応になりません。グローバル変数は保護されず、戻り値も静的変数で返されます。マルチスレッド環境で実行できるマルチスレッド対応コードを生成するには、–M フラグを指定します。–M フラグは、–N か –C のどちらか (または両方) のフラグと共に指定します。

この機能を使用したマルチスレッド対応プログラムの例を示します。コード例 3–14 に rpcgen のプロトコルファイル msg.x を示します。

コード例 **3–14**　マルチスレッド対応プログラム：msg.x

```
program MESSAGEPROG {
version PRINTMESSAGE {
        int PRINTMESSAGE(string) = 1;
        } = 1;
} = 0x4001;
```

文字列が遠隔手続きに渡され、遠隔手続きでは文字列を表示してから文字数をクライアントに返します。マルチスレッド対応のテンプレートを生成するには、次のコマンドを実行します。

```
% rpcgen -M msg.x
```

コード例 3–15 に、クライアント側のコードを示します。

コード例 **3–15**　マルチスレッド対応のクライアント側スタブプログラム

```
#include "msg.h"

void
messageprog_1(host)
 char *host;
{
 CLIENT *clnt;
 enum clnt_stat retval_1;
 int result_1;
 char * printmessage_1_arg;

 clnt = clnt_create(host, MESSAGEPROG,
         PRINTMESSAGE,
         "netpath");
 if (clnt == (CLIENT *) NULL) {
  clnt_pcreateerror(host);
  exit(1);
 }
 printmessage_1_arg =
       (char *) malloc(256);
 strcpy(printmessage_1_arg, "Hello World");

 retval_1 = printmessage_1(&printmessage_1_arg,
           &result_1,clnt);
 if (retval_1 != RPC_SUCCESS) {
```

(続く)

```
  clnt_perror(clnt, "call failed");
 }
 printf("result = %d\n", result_1);

 clnt_destroy(clnt);
}

main(argc, argv)
 int argc;
 char *argv[];
{
 char *host;

 if (argc < 2) {
  printf("usage:  %s server_host\n", argv[0]);
  exit(1);
 }
 host = argv[1];
 messageprog_1(host);
}
```

ここで、rpcgen が生成したコードには、引数も戻り値もポインタで渡さなければならないことに注意してください。これはプログラムを再入可能にするために必要です。スタブ関数の戻り値は、遠隔手続きの呼び出しが正常終了したかエラーが起こったかを示します。正常終了した場合は、RPC_SUCCESS が返されます。コード例 3–16 に示すマルチスレッド対応のクライアント側スタブプログラム (–M で生成) とマルチスレッド対応でないクライアント側スタブプログラムを比較してください。マルチスレッド未対応のクライアント側スタブプログラムは、静的変数を使用して戻り値を格納し、一度に 1 つしかスレッドを使用することができません。

コード例 3–16　クライアント側スタブプログラム (マルチスレッドに対応していない)

```
int *
printmessage_1(argp, clnt)
 char **argp;
 CLIENT *clnt;
{
 static int clnt_res;
 memset((char *)&clnt_res, 0,
        sizeof (clnt_res));
 if (clnt_call(clnt, PRINTMESSAGE,
  (xdrproc_t) xdr_wrapstring,
          (caddr_t) argp,
```

(続く)

```
  (xdrproc_t) xdr_int, (caddr_t)
          &clnt_res,
  TIMEOUT) != RPC_SUCCESS) {
  return (NULL);
 }
 return (&clnt_res);
}
```

コード例 3–17 に、サーバー側コードを示します。

注 - マルチスレッド対応モードを使用するサーバープログラムをコンパイルする場合は、スレッドライブラリをリンクしなければなりません。そのためには、コンパイルコマンドに –lthread オプションを指定します。

コード例 3–17 マルチスレッド対応サーバー側スタブプログラム

```
#include "msg.h"
#include <syslog.h>

bool_t
printmessage_1_svc(argp, result, rqstp)
 char **argp;
 int *result;
 struct svc_req *rqstp;
{
 int retval;

 if (*argp == NULL) {
  syslog(LOG_INFO, "argp is NULL\n");
  *result = 0;
 }
 else {
  syslog("argp is %s\n", *argp);
  *result = strlen (*argp);
 }
 retval = 1;
 return (retval);
}

int
messageprog_1_freeresult(transp, xdr_result, result)
 SVCXPRT *transp;
 xdrproc_t xdr_result;
```

(続く)

続き

```
 caddr_t result;
{
 /*
  * 必要に応じてメモリー解放のためのコードを挿入
  */
 (void) xdr_free(xdr_result, result);
}
```

サーバー側のコードでは、静的変数を使用して戻り値を格納してはいけません。呼び出し側のルーチンから戻り値へのポインタが渡されますので、戻り値はそこに返します。正常終了の場合は 1 を返し、エラーが起こった場合は 0 を返します。

rpcgen が生成するコードには、手続きの呼び出しで割り当てたメモリーを解放するルーチンの呼び出しも含まれています。メモリーの不正使用を避けるため、サービスルーチンで割り当てたメモリーはすべてそのルーチンで解放する必要があります。上の例では、messageprog_1_freeresult() でメモリーの解放を行います。

通常は、xdr_free() を使用して割り当てたメモリーを解放します。(上の例では、メモリー割り当てを行なっていないので、メモリーの解放は実行されません)。

–M フラグを –N と –C のフラグと共に指定する例として、コード例 3–18 の add.x を見てみます。

コード例 3–18　マルチスレッド対応のプログラム：add.x

```
program ADDPROG {
 version ADDVER {
 int add(int, int) = 1;
  } = 1;
}= 199;
```

このプログラムでは、2 つの数値を加えてその結果をクライアントに返します。次のコマンドで、このファイルに対して rpcgen を実行します。

```
% rpcgen -N -M -C add.x
```

このプログラムを呼び出すコード例 3–19 は次のようになります。

コード例 3–19　マルチスレッド対応クライアント側プログラム：add.x

```
/*
 * このクライアント側メインルーチンでは複数のスレッドを起動します。
 * 各スレッドから同時にサーバールーチンを呼び出します。
 */

#include "add.h"

CLIENT *clnt;
#define NUMCLIENTS 5
struct argrec {
 int arg1;
 int arg2;
};

/*
 * 現在実行中のスレッド数をカウント
 */
int numrunning;
mutex_t numrun_lock;
cond_t condnum;

void
addprog(struct argrec *args)
{
 enum clnt_stat retval;
 int result;
 /* サーバールーチンの呼び出し */
 retval = add_1(args->arg1, args->arg2,
           &result, clnt);
 if (retval != RPC_SUCCESS) {
  clnt_perror(clnt, "call failed");
 } else
  printf("thread #%x call succeeded,
      result = %d\n", thr_getself(),
      result);
/*
 * 実行中のスレッド数をデクリメント
 */
 mutex_lock(&numrun_lock);
 numrunning--;
 cond_signal(&condnum);
 mutex_unlock(&numrun_lock);
 thr_exit(NULL);
}

main(int argc, char *argv[])
{
 char *host;
 struct argrec args[NUMCLIENTS];
```

(続く)

```
 int i;
 thread_t mt;
 int ret;

 if (argc < 2) {
  printf("usage:  %s server_host\n",
     argv[0]);
  exit(1);
 }
 host = argv[1];
 clnt = clnt_create(host, ADDPROG, ADDVER,
         "netpath");
 if (clnt == (CLIENT *) NULL) {
  clnt_pcreateerror(host);
  exit(1);
 };
 mutex_init(&numrun_lock, USYNC_THREAD, NULL);
 cond_init(&condnum, USYNC_THREAD, NULL);
 numrunning = 0;

 /* 個々のスレッドの起動 */
 for (i = 0; i < NUMCLIENTS; i++) {
  args[i].arg1 = i;
  args[i].arg2 = i + 1;
  ret = thr_create(NULL, NULL, addprog,
          (char *) &args[i],
           THR_NEW_LWP, &mt);
  if (ret == 0)
   numrunning++;
 }

 mutex_lock(&numrun_lock);
 /* 全スレッドの終了を待つ */
 while (numrunning != 0)
  cond_wait(&condnum, &numrun_lock);
 mutex_unlock(&numrun_lock);
 clnt_destroy(clnt);
}
```

サーバー側の手続きはコード例 3–20 のようになります。

注 - マルチスレッド対応モードを使用するサーバー側プログラムをコンパイルする場合は、スレッドライブラリにリンクしなければなりません。そのためには、コンパイルコマンドに `–lthread` オプションを指定します

コード例 **3–20**　マルチスレッド対応サーバー側プログラム：add.x

```
add_1_svc(int arg1, int arg2,
      int *result, struct svc_req *rqstp)
{
 bool_t retval;
 /* 結果の計算 */
 *result = arg1 + arg2;
 retval = 1;
 return (retval);
}

/*
 * サーバー手続きで割り当てたメモリーを解放するルーチン
 */
int
addprog_1_freeresult(SVCXPRT *transp,
         xdrproc_t xdr_result,
         caddr_t result)

{
 (void) xdr_free(xdr_result, result);
}
```

自動マルチスレッド対応モード

自動マルチスレッド対応モードにより、クライアントの要求を同時に処理するために Solaris スレッドが自動的に使用されます。–A オプションを指定して、RPC コードを自動マルチスレッド対応モードで生成します。また、–A を指定すると自動的に –M が指定されるため、–M を明示的に指定する必要はありません。生成されたコードはマルチスレッド対応でなければならないため、–M が (明示的ではなくても) 必要です。

マルチスレッド対応 RPC の詳細については 136ページの「マルチスレッド RPC プログラミング」、および 144ページの「マルチスレッド自動モード」 を参照してください。

次に、rpcgen によって生成される自動モードのプログラムの例を示します。コード例 3–21は、rpcgen のプロトコルファイルである time.x のコードです。文字列は遠隔手続きに引き渡されます。遠隔手続きは、文字列を表示してクライアントの文字列長を返します。マルチスレッド対応スタブを生成するには、次のコマンドを実行します。

コード例 **3–21**　自動マルチスレッド対応モード：time.x

```
 program TIMEPROG {
  version TIMEVERS {
   unsigned int TIMEGET(void) = 1;
   void TIMESET(unsigned) = 2;
  } = 1;
} = 0x20000044;
```

```
% rpcgen -A time.x
```

注 - –A オプションを使用すると、生成されたサーバー側のコードには、サーバーの自動マルチスレッド対応モードを使用するための命令が含まれます。

マルチスレッド対応モードを使用するサーバー側プログラムをコンパイルする場合は、スレッドライブラリにリンクしなければなりません。そのためには、コンパイルコマンドに –lthread オプションを指定します。

TI-RPC または TS-RPC のライブラリ選択

旧バージョンの rpcgen では、ソケット関数を使用してスタブプログラムを作成していました。SunOS 5.4 では、トランスポート独立の RPC ルーチン (TI-RPC) か、特定のトランスポート固有のソケットルーチン (TS-RPC) のどちらを使用するか選択できます。この機能は、旧バージョンとの互換性を保つために提供されています。デフォルトでは TI-RPC ルーチンが使用されます。TS-RPC ルーチンを使用したソースコードを生成するには、rpcgenで –b フラグを指定します。

ANSI C に準拠したコードの生成

rpcgen では、ANSI C に準拠したコードを出力するか、SPARCompiler C++3.0 に準拠したコードを選択するか指定できます。ANSI C に準拠したコードを生成するには、–C フラグを指定します。ほとんどの場合、42ページの「C 形式モード」指定フラグ –N も同時に指定します。

add.x のサーバー側テンプレート例は、次のコマンドで生成できます。

```
rpcgen -N -C -Ss -o add_server_template.c add.x
```

ここで、C++ 3.0 で記述されたサーバー上では遠隔手続き名が接尾辞 _svc で終わっていなければならないことに特に注意してください。次の例では、add.x に対して、コンパイルフラグ –C を指定してクライアント側の add_1 とサーバー側の add_1_svc が生成されています。

コード例 **3–22**　ANSI C に準拠したサーバー側テンプレート

```
/*
 * このファイルはテンプレートです。これを基にしてユーザー独自の関数を
 * 作成してください。
 */
#include <c_varieties.h>
#include "add.h"

int *
add_1_svc(int arg1, int arg2,
     struct svc_req *rqstp)
{
 static int result;
 /*
  * ここにサーバープログラムのコードを挿入
  */
 return(&result);
}
```

この出力ファイルは、構文も構造も ANSI C に準拠しています。–C フラグを指定して生成したヘッダーファイルは、ANSI C でも SPARCompiler C++ でも使用できます。

xdr_inline() カウント

rpcgen は、可能な限り xdr_inline() (xdr_admin(3) マニュアルページを参照) を使用して、より効率の良いコードを生成しようとします。構造体の中に xdr_inline() を使用できるような要素 (たとえば、integer、long、bool) があれば、構造体のその部分は xdr_inline() を使用してパックされます。デフォルトでは、パックされる要素が 5 つ以上連続していれば、インラインコードが生成されます。–i フラグを使用してインラインコードを生成する個数を変更することができます。たとえば、次のコマンド

```
rpcgen -i 3 test.x
```

では、パックできる要素が 3 つ以上連続していれば、インラインコードが生成されます。次のコマンド

```
rpcgen -i 0 test.x
```

では、インラインコードの生成が禁止されます。

ほとんどの場合、–i フラグを指定する必要はありません。このフラグの対象となるのは _xdr.c スタブプログラムだけです。

rpcgen プログラミングテクニック

この節では、RPC プログラミングと rpcgen の使用方法に関するさまざまなテクニックを示します。

- ネットワークタイプ

  rpcgen は、特定のトランスポートタイプに対応したサーバーコードを生成できます。

- 定義文

  C プリプロセッサシンボルを rpcgen のコマンドラインで定義できます。

- ブロードキャスト呼び出し

  サーバーはブロードキャスト呼び出しにエラー応答を送る必要はありません。

- アプリケーションのデバッグ

  通常の関数呼び出しとしてデバッグしてから、分散型アプリケーションに変更します。

- ポートモニタのサポート

  RPC サーバーの代わりにポートモニタで「受信待ち」することができます。

- ディスパッチテーブル

  プログラムからサーバーのディスパッチテーブルにアクセスできます。

- タイムアウト値の変更

クライアント側のデフォルトのタイムアウト値を変更できます。

- 認証

  クライアントはサーバーに自分自身を証明することができます。関連サーバーはクライアントの認証情報を調べることができます。

ネットワークタイプ / トランスポート選択

rpcgen の省略可能な引数には、使用したいネットワークのタイプや特定のネットワーク識別子を指定するためのものがあります。ネットワーク選択についての詳細は、『*Transport Interfaces Programming Guide*』を参照してください。

–s フラグを指定すると、指定したタイプのトランスポートからの要求に応答するサーバーが作成されます。たとえば、次のコマンド

```
rpcgen -s datagram_n prot.x
```

を実行すると、NETPATH 環境変数で指定した非接続型トランスポートすべてに応答するサーバーが標準出力に書き出されます。NETPATH 環境変数が定義されていない場合は、/etc/netconfig で指定した非接続型トランスポートすべてに応答するサーバーが標準出力に書き出されます。コマンドラインでは、–s フラグとネットワークタイプのペアを複数指定できます。

同様に、–n フラグを指定すると、1 つのネットワーク識別子で指定したトランスポートからの要求だけに応答するサーバーを作成することができます。

注意 - rpcgen で –n フラグを指定して作成したサーバーを使用するときは注意が必要です。ネットワーク識別子は各ホストに固有なため、作成されたサーバーは別のホストで予測通りに機能しないことがあります。

コマンド行の定義文

コマンド行で、C 言語のプリプロセッサシンボルを定義し、値を割り当てることができます。コマンド行の定義文は、たとえば、DEBUG シンボルが定義されているときの条件付きデバッグコードの生成に使用できます。

```
$ rpcgen -DDEBUG proto.x
```

ブロードキャスト呼び出しへのサーバーからの応答

手続きがブロードキャスト RPC を通して呼び出され、有効な応答を返せないときは、サーバーはクライアントに応答しないでください。その方がネットワークが混雑しません。サーバーが応答を返さないようにするには、遠隔手続きの戻り値を NULL にします。rpcgen が生成したサーバープログラムは、NULL を受け取った場合は応答しません。

コード例 3–23 に、NFS サーバーの場合だけ応答する手続きを示します。

コード例 3–23 ブロードキャスト呼び出しに対する NFS サーバーの応答

```
void *
reply_if_nfsserver()
{
 char notnull; /*
        *この場所のみで、そのアドレスが使用可能
        */

 if( access( "/etc/dfs/sharetab",
      F_OK ) < 0 ) {
  /* RPC の応答を禁止 */
  return( (void *) NULL );
 }
 /*
 * NULL 以外の値 notnull を指定したので、RPC は応答する
 */
 return( (void *) &notnull );
}
```

RPC ライブラリルーチンが応答するには、手続きが NULL 以外のポインタ値を返す必要があります

コード例 3–23 で手続き reply_if_nfsserver() が NULL 以外の値を返すように定義されているならば、戻り値 (&notnull) は静的変数を指していなければなりません。

ポートモニタのサポート

inetd や listen のようなポートモニタは、特定の RPC サービスに対するネットワークアドレスを監視することができます。特定のサービスに対する要求が到着すると、ポートモニタは、サーバープロセスを生成します。サービスを提供したら、サーバーは終了できます。この技法はシステム資源を節約するためのもので

す。rpcgen で生成するサーバー関数 main() は inetd で呼び出すことができます。その方法についての詳細は、129ページの「inetd の使用」を参照してください。

サーバープロセスがサービス要求に答えた後、続けて要求が来る場合に備えて一定時間待つことには意味があります。一定時間内に次の呼び出しが起こらなければ、サーバーは終了し、inetd のようなポートモニタがサーバーのための監視を続けます。サーバーが終了しないうちに次の要求が来れば、ポートモニタは新たなプロセスを生成することなく待ち状態のサーバーにその要求を送ります。

注 - listen() などのポートモニタの場合は、サーバーのための監視を行い、サービス要求が来れば必ず新たなプロセスを生成します。このようなモニタからサーバープロセスを起動する場合は、サーバープロセスはサービス提供後すぐに終了するようにしなければなりません。

---

rpcgen がデフォルトで生成したサービスは、サービス提供後 120 秒間待ってから終了します。待ち時間を変更するには、–K フラグを使用します。たとえば、次のコマンド

```
$ rpcgen -K 20 proto.x
```

では、サーバーは 20 秒待ってから終了します。サービス提供後すぐに終了させるには、次のように待ち時間に対して 0 を指定します。

```
$ rpcgen -K 0 proto.x.
```

ずっと待ち状態を続けて終了しないようにするには、–K -1 と指定します。

タイムアウト値の変更

クライアントプログラムはサーバーに要求を送った後、デフォルトで 25 秒間応答を待ちます。応答待ちのタイムアウト値は、clnt_control() ルーチンを使用して変更できます。clnt_control() ルーチンの使用方法についての詳細は、77ページの「標準インタフェース」および rpc(3N) のマニュアルページを参照してください。タイムアウト値を変更するときは、ネットワークを往復するのに必要な最低時間以上になるように注意します。コード例 3–24 にclnt_control () の使用方法を示します。

コード例 **3–24**　clnt_control ルーチン

```
struct timeval tv;
CLIENT *clnt;
clnt = clnt_create( "somehost", SOMEPROG,
         SOMEVERS, "visible" );

if (clnt == (CLIENT *)NULL)
 exit(1);
tv.tv_sec = 60; /*
        * タイムアウト値を 60 秒に変更
        */
tv.tv_usec = 0;
clnt_control(clnt, CLSET_TIMEOUT, &tv);
```

クライアントの認証

クライアント作成ルーチンにはクライアント認証機能はありません。クライアントによっては、サーバーに対して自分自身を証明する必要があります。

次の例では、セキュリティレベルが最低限のクライアント認証方法のうち、一般に使用できる方法を示します。よりセキュリティレベルが高い認証テクニックの詳細については、106ページの「認証」と115ページの「RPCSEC_GSS を使用した認証」を参照してください。

コード例 **3–25**　AUTH_SYS クライアントの認証

```
CLIENT *clnt;
clnt = clnt_create( "somehost", SOMEPROG,
        SOMEVERS, "visible" );
if (clnt != (CLIENT *)NULL) {
/* AUTH_SYS 形式の認証情報を設定 */
 clnt->cl_auth = authsys_createdefault();
}
```

一定のセキュリティレベルを保持しなければならないサーバーではクライアント認証情報が必要になります。クライアント認証情報は、第 2 引数でサーバーに渡されます。

コード例 3–26 に、クライアント認証情報をチェックするサーバープログラムを示します。これは、25ページの「rpcgen チュートリアル」で説明した

printmessage_1() を修正したもので、スーパーユーザーにだけコンソールへのメッセージの表示を許可します。

コード例 **3–26**　スーパーユーザーだけが使用できる printmsg_1

```
int *
printmessage_1(msg, req)
char **msg;
struct svc_req  *req;
{
static int result; /* 必ず static で宣言 */
FILE *f;
struct authsys_parms *aup;

aup = (struct authsys_parms *)req->rq_clntcred;
if (aup->aup_uid != 0) {
 result = 0;
 return (&result)
}

/* 元のコードと同じ */
}
```

ディスパッチテーブル

RPC パッケージで使用するディスパッチテーブルにプログラムからアクセスしたい場合があります。たとえば、サーバーディスパッチルーチンで権限を調べてからサービスルーチンを呼び出したり、クライアントライブラリで記憶管理や XDR データ変換の詳細を扱う場合です。

–T オプションを指定して rpcgen を起動すると、プロトコル記述ファイル proto.x で定義した各プログラムごとの RPC ディスパッチテーブルがファイル proto_tbl.i に出力されます。接尾辞.i は index を表します。–t オプションを指定して rpcgen を起動した場合は、ヘッダーファイルだけが生成されます。rpcgen を起動するときは、C 形式モード (–N オプション) と同時に –T または –t フラグを指定することはできません。

ディスパッチテーブルの各エントリは struct rpcgen_tableで、この構造体はヘッダーファイル proto.h で次のように定義されています。

```
struct rpcgen_table {
  char *(*proc)();
  xdrproc_t xdr_arg;
  unsigned len_arg;
  xdrproc_t xdr_res;
```

```
  xdrproc_t len_res
};
```

ここで

```
proc       サービスルーチンへのポインタ
 xdr_arg    入力 (引数) の xdr ルーチンへのポインタ
 len_arg    入力引数の長さ (バイト数)
xdr_res    出力 (結果) の xdr ルーチンへのポインタ
len_res     出力結果の長さ (バイト数)
```

サンプルプログラム dir.x のディスパッチテーブル dirprog_1_table は、手続き番号がインデックスになっています。変数 dirprog_1_table には、テーブル内のエントリ数が入っています。

ディスパッチテーブルから手続きを探し出すルーチン find_proc() を次に示します。

コード例 3–27　ディスパッチテーブルの使用方法

```
struct rpcgen_table *
find_proc(proc)
  rpcproc_t proc;
{
  if (proc >= dirprog_1_nproc)
       /* error */
  else
  return (&dirprog_1_table[proc]);
}
```

ディスパッチテーブル内の各エントリは対応するサービスルーチンへのポインタです。ところが、サービスルーチンは一般にクライアント側のコードでは定義されていません。未解決の外部参照を起こさないため、また、ディスパッチテーブルに対するソースファイルを 1 つだけ要求にするために RPCGEN_ACTION(proc_ver) で、rpcgen サービスルーチン の初期化を行います。

これを使用して、クライアント側とサーバー側に同一のディスパッチテーブルを持たせることができます。クライアント側プログラムをコンパイルするときは、次の define 文を使用します。

```
#define RPCGEN_ACTION(routine) 0
```

サーバー側プログラムを作成するときは、次の define 文を使用します。

```
#define RPCGEN_ACTION(routine)routine
```

rpcgen の 64 ビットの場合の考慮事項

コード例 3–27 では、proc は タイプ rpcproc_t として宣言されていることに注意してください。正式には、RPC のプログラム、バージョン、プロシージャ、およびポートは、タイプ u_long として宣言されていました。32 ビットマシン上では、u_long の量は 4 バイト (int として) で、64 ビットシステム上では u_long の量は 8 バイトになります。RPC のプログラム、バージョン、プロシージャ、およびポートを宣言できる場合には必ず、u_long と long の代わりに Solaris 7 で導入されたデータタイプ rpcprog_t、rpcvers_t、rpc_proc_t、rpcport_t を使用する必要があります。それは、これらの新しいタイプを使用すると、32 ビットシステムとの下位互換性があるからです。つまり、この新しいデータタイプによって、rpcgen を実行するシステムに関係なく、4 バイトの量が保証されます。プログラム、バージョン、およびプロシージャの u_long バージョンを使用する rpcgen プログラムを引き続き実行すると、32 ビットマシンと 64 ビットマシンでは、異なる結果になる場合があります。そのため、これらを該当する新しいデータタイプに置き換えることをお勧めします。実際、可能な限り long と u_long の使用は避けた方が賢明です (この後の注を参照)。

Solaris 7 を起動する場合、rpcgen によって作成されたソースファイルには XDR ルーチンが組み込まれていて、このソースファイルは、そのコードを 32 ビットマシンと 64 ビットマシン上のどちらで実行するかによって、異なるインラインマクロが使用されます。特に、IXDR_GETLONG() と IXDR_PUTLONG() の代わりに、IXDR_GET_INT32() と IXDR_PUT_INT32() マクロが使用されます。たとえば、rpcgen ソースファイル foo.x に以下のコードが組み込まれている場合、

```
struct foo {
        char      c;
        int       i1;
        int       i2;
        int       i3;
```

(続く)

```
        long      l;
        short     s;
};
```

結果のファイル foo_xdr.c ファイルでは、次のように適切なインラインマクロが使用されているかどうか確認されます。

```
#if defined(_LP64) || defined(_KERNEL)
        register int *buf;
#else
        register long *buf;
#endif

. . .

#if defined(_LP64) || defined(_KERNEL)
                        IXDR_PUT_INT32(buf, objp->i1);
                        IXDR_PUT_INT32(buf, objp->i2);
                        IXDR_PUT_INT32(buf, objp->i3);
                        IXDR_PUT_INT32(buf, objp->l);
                        IXDR_PUT_SHORT(buf, objp->s);
#else
                        IXDR_PUT_LONG(buf, objp->i1);
                        IXDR_PUT_LONG(buf, objp->i2);
                        IXDR_PUT_LONG(buf, objp->i3);
                        IXDR_PUT_LONG(buf, objp->l);
                        IXDR_PUT_SHORT(buf, objp->s);
#endif
```

このコードにより、 *buf* は int または long のどちらかになるように宣言されますが、これはマシンが 64 ビットであるか、または 32 ビットであるかによって決まるということに注意してください。

注 - 現在は、RPC を通じて転送されるデータタイプのサイズは、4 バイトの量 (32 ビット) に制限されています。8 バイトの long は、アプリケーションが 64 ビットのアーキテクチャを最大限に使用できるようにする場合に提供されます。ただし、プログラマは、int のために long や、x_putlong() などの long を使用する関数の使用は、可能な限り避ける必要があります。上述したように、RPC プログラム、バージョン、プロシージャおよびポートにはそれぞれ専用のタイプがあります。 それは、データ値が INT32_MIN と INT32_MAX 間にない場合、xdr_long() は失敗し、また、IXDR_GET_LONG() と IXDR_PUT_LONG() などのインラインマクロが使用されると、そのデータは切り捨てられるからです (u_long の場合も同様) 。xdr_long(3N) のマニュアルページも参照してください。

アプリケーションのデバッグ

作成したアプリケーションのテストとデバッグは、簡単に実行できます。最初は、クライアント側とサーバー側の手続きをリンクして全体をシングルプロセスとしてテストします。最初は、各手続きをそれぞれクライアント側とサーバー側のスケルトンとはリンクしません。クライアントを作成するRPC ライブラリルーチン (rpc_clnt_create(3N) のマニュアルページを参照) と認証ルーチンの呼び出し部分はコメントにします。この段階では、libnsl をリンクしてはいけません。

これまでに説明したサンプルプログラムの手続きを、次のコマンドでリンクします。

```
cc rls.c dir_clnt.c dir_proc.c -o rls
```

RPC と XDR の関数をコメントにすると、手続き呼び出しは通常のローカル関数呼び出しとなり、プログラムは dbxtool のようなローカルデバッガでデバッグ可能になります。プログラムが正しく機能することが確認されたら、クライアント側プログラムを rpcgen が生成したクライアント側のスケルトンとリンクし、サーバー側プログラムを rpcgen が生成したクライアント側のスケルトンとリンクします。

また、Raw PRC モードを使用して XDR ルーチンをテストすることもできます。その方法についての詳細は、95ページの「下位レベルの Raw RPC を使用したプログラムテスト」を参照してください。

RPC 呼び出しで発生するエラーには 2 種類あります。1 つは、遠隔手続き呼び出し過程で起こるエラーです。これには、(1) 手続きが実行できない、(2)遠隔サーバーが応答しない、(3) 遠隔サーバーが引数を復号化できない、などがあります。コード例

3–26 で考えると、result が NULL の場合は RPC エラーです。エラーの原因を調べるには、clnt_perror() を使用してエラー原因を表示するか、clnt_sperror() を使用してエラー文字列を取り出します。

もう 1 つのエラーは、サーバー自体のエラーです。コード例 3–26 で考えると、opendir() からエラーが返された場合です。このようなエラーの処理はアプリケーションによって異なるため、プログラマの責任で対応します。

–C オプションを指定した場合はサーバー側ルーチンに _svc という接尾辞が付くため、上の説明がそのまま当てはまらないことに注意してください。

第 4 章

RPC プログラマインタフェース

この章では、RPC との C インタフェースについて取り上げ、RPC を使用してネットワークアプリケーションを書く方法を説明します。RPC ライブラリにおけるルーチンの完全な仕様については、`rpc(3N)` のマニュアルページおよび関連するマニュアルページを参照してください。

マルチスレッド対応の RPC

この章で説明するクライアントおよびサーバーインタフェースは、特に注意書きがある場合 (raw モードなど) 以外は、マルチスレッド対応です。すなわち、RPC 関数を呼び出すアプリケーションはマルチスレッド環境で自由に実行することができます。

単純インタフェース

単純インタフェースでは、その他の RPC ルーチンは不要なため最も簡単に使用できるレベルです。しかし、利用できる通信メカニズムの制御は制限されます。このレベルでのプログラム開発は早く、rpcgen コンパイラによって直接サポートされます。大部分のアプリケーションに対しては、rpcgen が提供する機能で十分です。

RPC サービスの中には C の関数としては提供されていないものがありますが、それも RPC プログラムとして使用できます。単純インタフェースライブラリルーチンは、詳細な制御を必要としないプログラムでは RPC 機能を直接使用できます。rusers() のようなルーチンは、RPC サービスライブラリ librpcsvc にあります。コード例 4–1 は、RPC ライブラリルーチン rusers() を呼び出して、遠隔ホスト上のユーザー数を表示します。

コード例 4–1 rusers プログラム

```
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>
#include <stdio.h>

/*
 * rusers() サービスを
 * 呼び出すプログラム
 */

main(argc, argv)
 int argc;
 char **argv;
{
 int num;

 if (argc != 2) {
  fprintf(stderr, "usage: %s hostname\n",
      argv[0]);
  exit(1);
 }
 if ((num = rnusers(argv[1])) < 0) {
  fprintf(stderr, "error: rusers\n");
  exit(1);
 }
 fprintf(stderr, "%d users on %s\n", num,
      argv[1] );
 exit(0);
}
```

コード例 4–1 のプログラムを次のコマンドを使用してコンパイルします。

```
cc program .c -lrpcsvc -lnsl
```

クライアント側

単純インタフェースのクライアント側には、rpc_call() という関数が 1 つだけあります。次の 9 個のパラメータがあります。

```
int 0 or error code
rpc_call (
        char    *host  /* サーバーホストの名前 */
        rpcprog_t  prognum  /* サーバープログラム番号 */
        rpcvers_t  versnum  /* サーバーバージョン番号 */
        rpcproc_t  procnum  /* サーバー手続き番号 */
        xdrproc_t  inproc  /* 引数を符号化する XDR フィルタ */
        char    *in   /* 引数へのポインタ */
        xdr_proc_t  outproc  /* 結果を復号化するフィルタ */
        char    *out  /* 結果を格納するアドレス */
        char    *nettype /* トランスポートの選択 */
);
```

この関数は、host 上で、prognum、versum、procnum によって指定する手続きを呼び出します。遠隔手続きに渡される引数は、in パラメータによって指定され、inproc はこの引数を符号化するための XDR フィルタです。out パラメータは、遠隔手続きから戻される結果が置かれるアドレスです。outproc は、結果を復号化してこのアドレスに置く XDR フィルタです。

クライアントプログラムは、サーバーから応答を得るまで rpc_call() のところで停止します。サーバーが呼び出しを受け入れると、0 の値で RPC_SUCCESS を返します。呼び出しが失敗した場合は、0 以外の値が返されます。この値は、clnt_stat で指定される型に型変換されます。これは RPC インクルードファイルの中で定義される列挙型で、clnt_sperrno() 関数により解釈されます。この関数は、このエラーコードに対応する標準 RPC エラーメッセージへのポインタを返します。

この例では、/etc/netconfig に列挙されているすべての選択可能な可視トランスポートが試されます。試行回数を指定するには、下位レベルの RPC ライブラリを使用する必要があります。

複数の引数と複数の結果は、構造体に収集して扱われます。

コード例 4–1 を単純インタフェースを使用するために変更すると、コード例 4–2 のようになります。

コード例 **4–2**　単純インタフェースを使用する rusers プログラム

```
#include <stdio.h>
#include <utmp.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>

/*
 *RUSERSPROG RPC プログラムを呼び出すプログラム
 */

main(argc, argv)
 int argc;
 char **argv;
{
 unsigned int nusers;
 enum clnt_stat cs;

 if (argc != 2) {
     fprintf(stderr, "usage: rusers hostname\n");
     exit(1);
 }
 if( cs = rpc_call(argv[1], RUSERSPROG,
     RUSERSVERS, RUSERSPROC_NUM, xdr_void,
     (char *)0, xdr_u_int, (char *)&nusers,
                    "visible")  !=  RPC_SUCCESS )  {
            clnt_perrno(cs);
            exit(1);
            }
 fprintf(stderr, "%d users on %s\n", nusers,
      argv[1] );
 exit(0);
}
```

マシンが異なれば、データ型も異なる表現になるため、rpc_call() は RPC 引数の型と RPC 引数へのポインタを必要とします (サーバーから返される結果についても同様です)。RUSERSPROC_NUM の場合、戻り値は unsigned int 型であるため、rpc_call() の最初の戻りパラメータは xdr_u_int (unsigned int 用) で、2 番目は &nusers (unsigned int 型の値があるメモリーへのポインタ)です。RUSERSPROC_NUM には引数がないため、rpc_call() の XDR 符号化関数は xdr_void() で、その引数は NULL です。

サーバー側

単純インタフェースを使用するサーバープログラムは、大変理解しやすいものです。これは単に、呼び出される手続きを登録するため rpc_reg() を呼び出し、次に、RPC ライブラリの遠隔手続きディスパッチャである svc_run() を呼び出して、入ってくる要求を待ちます。

rpc_reg() には次の引数があります。

```
rpc_reg (
      rpcprog_t  prognum   /* サーバープログラム番号 */
      rpcvers_t  versnum   /* サーバーバージョン番号 */
      rpcproc_t  procnum   /* サーバー手続き番号 */
      char    *procname  /* 遠隔関数の名前 */
      xdrproc_t  inproc   /* 引数を符号化するフィルタ */
      xdrproc_t  outproc   /* 結果を復号化するフィルタ */
      char    *nettype   /* トランスポートの選択 */
);
```

svc_run() は RPC 呼び出しメッセージに応えてサービス手続きを起動します。rpc_reg() のディスパッチャは遠隔手続きが登録されたときに指定された XDR フィルタを使用して、遠隔手続きの引数の復号化と、結果の符号化を行います。サーバープログラムについての注意点をいくつか挙げます。

- ほとんどの RPC アプリケーションが、関数名の後に _1 を付ける命名規則に従っています。手続き名に *_n* 番号を付けることにより、サービスのバージョン番号 *n* を表します。
- 引数と結果はアドレスで渡されます。遠隔で呼び出される関数はすべてこうなります。関数の結果として NULL を渡すと、クライアントには応答が送信されません。送信する応答がないと仮定されます。
- 結果は固定のデータ領域に存在します。これは、その値が実際の手続きが終了したあとにアクセスされるからです。RPC 応答メッセージを作成する RPC ライブラリ関数は結果にアクセスして、その値をクライアントに戻します。
- 引数は 1 つだけ使用できます。データに複数の要素がある場合、構造体の中に入れると、1 つの引数として渡すことができます。
- 手続きは、指定するタイプのトランスポートごとに登録されます。タイプのパラメータが (char *)NULL の場合、手続きは NETPATH により指定されるすべてのトランスポートに登録されます。

ユーザーが作成する登録ルーチン

rpcgen は汎用のコードジェネレータであるため、ユーザーが自分で書いた方が、効率のよい短いコードにできる場合があります。そのような登録ルーチンの例を次に示します。次の例では、手続きを 1 つ登録してから、svc_run() に入ってサービス要求を待ちます。

```
#include <stdio.h>
 #include <rpc/rpc.h>
 #include <rpcsvc/rusers.h>
 void *rusers();

 main()
 {
  if(rpc_reg(RUSERSPROG, RUSERSVERS,
      RUSERSPROC_NUM, rusers,
      xdr_void, xdr_u_int,
      "visible") == -1) {
   fprintf(stderr, "Couldn't Register\n");
   exit(1);
  }
  svc_run();      /* この関数は値を戻さない */
  fprintf(stderr, "Error: svc_run
      returned!\n");
  exit(1);
 }
```

rpc_reg() は、異なるプログラム、バージョン、手続きを登録するごとに何度でも呼び出すことができます。

任意のデータ型の引き渡し

遠隔手続きへ渡すデータ型と遠隔手続きから受け取るデータ型は、事前に定義した型あるいはプログラマが定義する型の任意のものが可能です。RPC では、データを XDR (external data representation: 外部データ表現) 形式という標準データ形式に変換してからトランスポートに送信するため、個々のマシンに固有のバイト順序や構造体のデータレイアウトに関係なく、任意のデータ構造を扱うことができます。マシン固有のデータ形式から XDR 形式に変換することをシリアライズといい、反対に XDR 形式からマシン固有のデータ形式に変換することをデシリアライズといいます。

rpc_call() と rpc_reg() の引数で変換ルーチンを指定するときは、xdr_u_int() のような XDR プリミティブを指定することもできますし、引数とし

て渡された構造体全体を処理するようなユーザー作成の変換ルーチンを指定することもできます。引数の変換ルーチンは 2 つの引数を取ります。1 つは変換結果へのポインタで、もう 1 つは XDR ハンドルへのポインタです。

表 4–1 XDR プリミティブタイプのルーチン

XDR プリミティブ・ルーチン			
`xdr_int()`	`xdr_netobj()`	`xdr_u_long()`	`xdr_enum()`
`xdr_long()`	`xdr_float()`	`xdr_u_int()`	`xdr_bool()`
`xdr_short()`	`xdr_double()`	`xdr_u_short()`	`xdr_wrapstring()`
`xdr_char()`	`xdr_quadruple()`	`xdr_u_char()`	`xdr_void()`
`xdr_hyper()`	`xdr_u_hyper()`		

`int_types.h` 内にある固定幅の整数タイプに慣れている ANSI C プログラマにとって都合がよいように、ルーチン
`xdr_char()`、`xdr_short()`、`xdr_int()`、`xdr_hyper()` (および、それぞれの符号なしバージョン) には、表 4–2 で示すように、ANSI C を連想させる名前の付いた同等の関数があります。

表 4–2 プリミティブタイプの等価関数

関数	**Equivalent**
`xdr_char()`	`xdr_int8_t()`
`xdr_u_char()`	`xdr_u_int8_t()`
`xdr_short()`	`xdr_int16_t()`
`xdr_u_short()`	`xdr_u_int16_t()`
`xdr_int()`	`xdr_int32_t()`
`xdr_u_int()`	`xdr_u_int32_t()`

表 **4–2**　プリミティブタイプの等価関数　続く

関数	**Equivalent**
`xdr_hyper()`	`xdr_int64_t()`
`xdr_u_hyper()`	`xdr_u_int64_t()`

`xdr_wrapstring()` から呼び出す `xdr_string()` はプリミティブではなく、3 つ以上の引数を取ります。

ユーザーが作成する変換ルーチンの例を次に示します。手続きに渡す引数は次の構造体に入れます。

```
struct simple {
  int a;
  short b;
 } simple;
```

この構造体で渡された引数を変換する XDR ルーチン `xdr_simple()` は、コード例 4–3 に示すようになります。

コード例 **4–3**　`xdr_simple()` ルーチン

```
#include <rpc/rpc.h>
#include "simple.h"

bool_t
xdr_simple(xdrsp, simplep)
 XDR *xdrsp;
 struct simple *simplep;
{
 if (!xdr_int(xdrsp, &simplep->a))
  return (FALSE);
 if (!xdr_short(xdrsp, &simplep->b))
  return (FALSE);
 return (TRUE);
}
```

`rpcgen` でも、同じ機能を持つ変換ルーチンを自動生成できます。

XDR ルーチンは、データ変換に成功した場合はゼロ以外の値 (C では TRUE) を返し、失敗した場合はゼロを返します。XDR についての詳細は、付録 C を参照してください。

表 4–3 XDR ブロック構築ルーティン

基本のルーチン		
xdr_array()	xdr_bytes()	xdr_reference()
xdr_vector()	xdr_union()	xdr_pointer()
xdr_string()	xdr_opaque()	

たとえば、可変長の整数配列を送るときは、配列へのポインタと配列サイズを次のような構造体にパックします。

```
struct varintarr {
  int *data;
  int arrlnth;
 } arr;
```

この配列を変換するルーチン xdr_varintarr() は コード例 4–4 に示すようになります。

コード例 4–4 変換ルーチン xdr_varintarr()

```
bool_t
xdr_varintarr(xdrsp, arrp)
 XDR *xdrsp;
 struct varintarr *arrp;
{
 return(xdr_array(xdrsp, (caddr_t)&arrp->data,
  (u_int *)&arrp->arrlnth, MAXLEN,
  sizeof(int), xdr_int));
}
```

xdr_array() に渡す引数は、XDR ハンドル、配列へのポインタ、配列サイズへのポインタ、配列サイズの最大値、配列要素のサイズ、配列要素を変換する XDR

ルーチンへのポインタです。配列サイズが前もってわかっている場合は、コード例 4–5 のように xdr_vector() を使用します。

コード例 **4–5**　変換ルーチン xdr_vector ()

```
int intarr[SIZE];

bool_t
xdr_intarr(xdrsp, intarr)
 XDR *xdrsp;
 int intarr[];
{
 return (xdr_vector(xdrsp, intarr, SIZE,
    sizeof(int),
xdr_int));
}
```

XDR ルーチンでシリアライズすると、データが 4 バイトの倍数になるように変換されます。たとえば、文字配列を変換すると、各文字が 32 ビットを占有するようになります。xdr_bytes() は、文字をパックするルーチンで、xdr_array() の最初の 4 つの引数と同様の引数を取ります。

NULL で終わる文字列は xdr_string() で変換します。このルーチンは、長さの引数がない xdr_bytes() ルーチンのようなものです。文字列をシリアライズするときは strlen() で長さを取り出し、デシリアライズするときは NULL で終わる文字列を生成します。

コード例 4–6 では、組み込み関数 xdr_string() と xdr_reference() を呼び出して、文字列へのポインタと、前の例で示した構造体　simple へのポインタを変換します。

コード例 **4–6**　変換ルーチン xdr_reference ()

```
struct finalexample {
 char *string;
 struct simple *simplep;
} finalexample;

bool_t
xdr_finalexample(xdrsp, finalp)
```

(続く)

```
 XDR *xdrsp;
 struct finalexample *finalp;
{
 if (!xdr_string(xdrsp, &finalp->string,
   MAXSTRLEN))
  return (FALSE);
 if (!xdr_reference( xdrsp, &finalp->simplep,
   sizeof(struct simple), xdr_simple))
  return (FALSE);
 return (TRUE);

}
```

ここで、xdr_reference() の代わりに xdr_simple() を呼び出してもよいことに注意してください。

標準インタフェース

RPC パッケージの標準レベルへのインタフェースは、RPC 通信へのさらに詳細な制御を提供します。この制御を使用するプログラムはより複雑になります。下位レベルでの効果的なプログラミングには、コンピュータネットワークの構造に対するより深い知識が必要です。トップ、中間、エキスパート、ボトムレベルは、標準インタフェースの一部です。

この節では、RPC ライブラリの下位レベルを使用して RPC プログラムを詳細に制御する方法について説明します。たとえば、単純インタフェースレベルでは NETPATH を介してしか使用できなかったトランスポートプロトコルを自由に使用できます。これらのルーチンを使用するには、TLI に対する知識が必要です。

表 4–4 に示したルーチンにはトランスポートハンドルの指定が必要なため、単純インタフェースからは使用できません。たとえば、単純レベルでは、XDR ルーチンでシリアライズとデシリアライズを行うときに、メモリーの割り当てと解放を行うことはできません。

表 4–4　トランスポートハンドルの指定が必要な XDR ルーチン

単純インタフェースでは使用できないルーチン		
clnt_call()	clnt_destroy()	clnt_control()
clnt_perrno()	clnt_pcreateerror()	clnt_perror()
svc_destroy()		

トップレベルのインタフェース

トップレベルのルーチンを使用すると、アプリケーションで使用するトランスポートタイプを指定できますが、特定のトランスポートは指定できません。このレベルは、クライアントとサーバーの両方でアプリケーションが自分のトランスポートハンドルを作成する点で、単純インタフェースと異なります。

クライアント側

コード例 4–7 に示すようなヘッダーファイルがあるとします。

コード例 4–7　ヘッダーファイル time_prot.h

```
/* time_prot.h */
#include <rpc/rpc.h>
#include <rpc/types.h>

struct timev {
 int second;
 int minute;
 int hour;
};
typedef struct timev timev;
bool_t xdr_timev();

#define TIME_PROG 0x40000001
#define TIME_VERS 1
#define TIME_GET  1
```

コード例 4–8 に、クライアント側の、トップレベルのサービスルーチンを使用する簡単な日時表示プログラムを示します。このプログラムでは、時刻を返すサービスを呼び出します。トランスポートタイプはプログラムを起動するときの引数で指定します。

コード例 4–8 時刻を返すサービス：クライアント側

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

#define TOTAL (30)
/*
 * 時刻を返すサービスを呼び出すプログラム
 * 使用方法: calltime ホスト名
 */
main(argc, argv)
 int argc;
 char *argv[];
{
 struct timeval time_out;
 CLIENT *client;
 enum clnt_stat stat;
 struct timev timev;
 char *nettype;

 if (argc != 2 && argc != 3) {
  fprintf(stderr,''usage:%s host[nettype]\n''
     ,argv[0]);
  exit(1);
 }
 if (argc == 2)
  nettype = "netpath";  /* デフォルト */
 else
  nettype = argv[2];
 client = clnt_create(argv[1], TIME_PROG,
         TIME_VERS, nettype);
 if (client == (CLIENT *) NULL) {
  clnt_pcreateerror(``Couldn't create client'');
  exit(1);
 }
 time_out.tv_sec = TOTAL;
 time_out.tv_usec = 0;
 stat = clnt_call( client, TIME_GET,
     xdr_void, (caddr_t)NULL,
     xdr_timev, (caddr_t)&timev,
     time_out);
 if (stat != RPC_SUCCESS) {
  clnt_perror(client, "Call failed");
  exit(1);
 }
 fprintf(stderr,"%s: %02d:%02d:%02d GMT\n",
    nettype timev.hour, timev.minute,
    timev.second);
```

(続く)

```
 (void) clnt_destroy(client);
 exit(0);
}
```

プログラムを起動するときに nettype を指定しなかった場合は、代わりに「netpath」という文字列が使用されます。RPC ライブラリルーチンは、この文字列を見つけると、環境変数 NETPATH 値によって使用するトランスポートを決めます。

クライアントハンドルが作成できない場合は、clnt_pcreateerror() でエラー原因を表示するか、グローバル変数 rpc_createerr の値としてエラーステータスを取り出します。

クライアントハンドルが作成できたら、clnt_call() を使用して遠隔呼び出しを行います。clnt_call() の引数は、クライアントハンドル、遠隔手続き番号、入力引数に対する XDR フィルタ、引数へのポインタ、戻り値に対する XDR フィルタ、戻り値へのポインタ、呼び出しのタイムアウト値です。この例では、遠隔手続きに渡す引数はないので、XDR ルーチンとしては xdr_void() を指定しています。最後に clnt_destroy() を使用して使用済みメモリーを解放します。

上記の例でプログラマがクライアントハンドル作成に許される時間を 30 秒に設定したいとすると、次のコード例の一部のように、clnt_create() への呼び出しは clnt_create_timed() への呼び出しに替わります。

```
struct timeval timeout;
timeout.tv_sec = 30;  /* 30 秒 */
timeout.tv_usec = 0;

client = clnt_create_timed(argv[1],
    TIME_PROG, TIME_VERS, nettype,
    &timeout);
```

コード例 4–9 には、トップレベルのサービスルーチンを使用したサーバー側プログラムを示します。このプログラムでは、時刻を返すサービスを実行します。

コード例 **4–9**　時刻を返すサービス：サーバー側

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "time_prot.h"

static void time_prog();

main(argc,argv)
 int argc;
 char *argv[];
{
 int transpnum;
 char *nettype;

 if (argc > 2) {

  fprintf(stderr, "usage: %s [nettype]\n",
      argv[0]);
  exit(1);
 }
 if (argc == 2)
  nettype = argv[1];
 else
  nettype = "netpath";   /* デフォルト */
 transpnum =
svc_create(time_prog,TIME_PROG,TIME_VERS,nettype);
 if (transpnum == 0) {
  fprintf(stderr,''%s: cannot create %s service.\n'',
     argv[0], nettype);
  exit(1);
 }
 svc_run();
}

/*
 * サーバーのディスパッチ関数
 */
static void
time_prog(rqstp, transp)
 struct svc_req *rqstp;
 SVCXPRT *transp;
{
 struct timev rslt;
 time_t thetime;

 switch(rqstp->rq_proc) {
  case NULLPROC:
   svc_sendreply(transp, xdr_void, NULL);
   return;
  case TIME_GET:
   break;
  default:
   svcerr_noproc(transp);
   return;
  }
 thetime = time((time_t *) 0);
```

(続く)

```
  rslt.second = thetime % 60;
  thetime /= 60;
  rslt.minute = thetime % 60;
  thetime /= 60;
  rslt.hour = thetime % 24;
  if (!svc_sendreply( transp, xdr_timev, &rslt)) {
   svcerr_systemerr(transp);
   }
}
```

svc_create() は、サーバーハンドルを作成したトランスポートの個数を返します。サービス関数 time_prog() は、対応するプログラム番号とバージョン番号を指定したサービス要求がきたときに svc_run() に呼び出されます。サーバーは、svc_sendreply() を使用して戻り値をクライアントに返します。

rpcgen を使用してディスパッチ関数を生成する場合は、svc_sendreply() は手続きがリターンしてから呼び出されるため、戻り値 (この例では rslt) は実際の手続き内で static 宣言しなければなりません。この例では、svc_sendreply() はディスパッチ関数の中で呼び出されているので、rslt は static で宣言されていません。

この例の遠隔手続きには引数がありませんが、引数を渡す必要がある場合は次の 2 つの関数

```
svc_getargs( SVCXPRT_handle, XDR_filter, argument_pointer);
svc_freeargs( SVCXPRT_handle, XDR_filter argument_pointer );
```

を呼び出して、引数を取り出し、デシリアライズ (XDR 形式から復号化) し、解放します。

中間レベルのインタフェース

中間レベルのルーチンを使用するときは、使用するトランスポート自体をアプリケーションから直接選択します。

クライアント側

コード例 4–10 は、78ページの「トップレベルのインタフェース」の時刻サービスのクライアント側プログラムを、中間レベルの RPC で書いたものです。この例のプログラムを実行するときは、どのトランスポートで呼び出しを行うか、コマンド行で指定する必要があります。

コード例 4–10 時刻サービスのクライアント側プログラム

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <netconfig.h>  /* 構造体 netconfig を使用するため */
#include "time_prot.h"

#define TOTAL (30)

main(argc,argv)
 int argc;
 char *argv[];
{
 CLIENT *client;
 struct netconfig *nconf;
 char *netid;
 /* 以前のサンプルプログラムの宣言と同じ */

 if (argc != 3) {
  fprintf(stderr, "usage: %s host netid\n'',
     argv[0]);
  exit(1);
 }
 netid = argv[2];
 if ((nconf = getnetconfigent( netid)) ==

     (struct netconfig *) NULL) {
  fprintf(stderr, "Bad netid type: %s\n",
     netid);
  exit(1);
 }
 client = clnt_tp_create(argv[1], TIME_PROG,
          TIME_VERS, nconf);
 if (client == (CLIENT *) NULL) {
  clnt_pcreateerror("Could not create client");
  exit(1);
 }
 freenetconfigent(nconf);

 /* これ以降は以前のサンプルプログラムと同じ */
}
```

この例では、getnetconfigent(netid) を呼び出して netconfig 構造体を取り出しています 。詳細については、getnetconfig(3N) マニュアルページと『*Transport Interfaces Programming Guide*』を参照してください。このレベルの RPC を使用する場合は、プログラムで直接ネットワーク (トランスポート) を選択できます

上記の例でプログラマがクライアントハンドル作成に許される時間を 30 秒に設定したいとすると、次のコード例の一部のように、 clnt_tp_create() への呼び出しは clnt_tp_create_timed() への呼び出しに替わります。

```
struct timeval timeout;
timeout.tv_sec = 30; /* 30 秒 */
timeout.tv_usec = 0;

client = clnt_tp_create_timed(argv[1],
   TIME_PROG, TIME_VERS, nconf,
   &timeout);
```

サーバー側

これに対するサーバー側プログラムを コード例 4–11 に示します。サービスを起動するコマンド行では、どのトランスポート上でサービスを提供するかを指定する必要があります。

コード例 4–11 時刻サービスのサーバー側プログラム

```
/*
 * このプログラムは、サービスを呼び出したクライアントにグリニッチ標準時を
 * 返します。呼び出し方法: server netid
 */
#include <stdio.h>
#include <rpc/rpc.h>
#include <netconfig.h>     /* 構造体 netconfig を使用するため */
#include "time_prot.h"

static void time_prog();

main(argc, argv)
 int argc;
 char *argv[];
{
 SVCXPRT *transp;
 struct netconfig *nconf;

 if (argc != 2) {
  fprintf(stderr, "usage: %s netid\n",
```

(続く)

```
    argv[0]);
  exit(1);
 }
 if ((nconf = getnetconfigent( argv[1])) ==
     (struct netconfig *) NULL) {
  fprintf(stderr, "Could not find info on %s\n",
     argv[1]);
  exit(1);
 }
 transp = svc_tp_create(time_prog, TIME_PROG,
          TIME_VERS, nconf);
 if (transp == (SVCXPRT *) NULL) {
  fprintf(stderr, "%s: cannot create
      %s service\n", argv[0], argv[1]);
  exit(1)
 }
 freenetconfigent(nconf);
 svc_run();
}

 static
 void time_prog(rqstp, transp)
  struct svc_req *rqstp;
  SVCXPRT *transp;
{
/* トップレベルの RPC を使用したコードと同じ */
```

エキスパートレベルのインタフェース

エキスパートレベルのネットワーク選択は、中間レベルと同じです。中間レベルとの唯一の違いは、アプリケーションから CLIENT と SVCXPRT のハンドルをより詳細に制御できる点です。次の例では、clnt_tli_create() と svc_tli_create() の 2 つのルーチンを使用した制御方法を示します。TLI についての詳細は、『*Transport Interfaces Programming Guide*』を参照してください。

クライアント側

コード例 4–12 には、clnt_tli_create() を使用して UDP トランスポートに対するクライアントを作成するルーチン clntudp_create() を示します。このプログラムでは、指定したトランスポートファミリに基づいたネットワーク選択方法を示します。clnt_tli_create() には、クライアントハンドルの作成のほかに次の 3 つの機能があります。

- オープンした TLI ファイル記述子を渡します。結合されている場合と結合されていない場合があります。
- クライアントにサーバーアドレスを渡します。
- 送信バッファと受信バッファのサイズを指定します。

コード例 **4–12**　下位レベル RPC 使用に対するクライアント側プログラム

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <netconfig.h>
#include <netinet/in.h>
/*
 * 旧バージョンの RPC では、TCP/IP と UDP/IP だけがサポートされていました。
 * 現バージョンの clntudp_create() は TLI/STREAMS に基づいています。
 */
CLIENT *
clntudp_create(raddr, prog, vers, wait, sockp)
 struct sockaddr_in *raddr;  /* 遠隔アドレス */
 rpcprog_t prog;       /* プログラム番号 */
 rpcvers_t vers;       /* バージョン番号 */
 struct timeval wait;     /* 待ち時間 */
 int *sockp;         /* ファイル記述子 (fd) のポインタ */
{
 CLIENT *cl;         /* クライアントハンドル */
 int madefd = FALSE;      /* fd はオープンされているか */
 int fd = *sockp;       /* TLI の fd */
 struct t_bind *tbind;     /* 結合アドレス */
 struct netconfig *nconf;   /* netconfig 構造体 */
 void *handlep;

 if ((handlep = setnetconfig() ) == (void *) NULL) {
  /* ネットワーク設定開始でのエラー */
  rpc_createerr.cf_stat = RPC_UNKNOWNPROTO;
  return((CLIENT *) NULL);
 }
 /*
  * 非接続型で、プロトコルファミリが INET、名前が UDP の
  * トランスポートが見つかるまで探す。
  */
 while (nconf = getnetconfig( handlep)) {
  if ((nconf->nc_semantics == NC_TPI_CLTS) &&
       (strcmp( nconf->nc_protofmly, NC_INET ) == 0) &&
       (strcmp( nconf->nc_proto, NC_UDP ) == 0))
    break;
 }
 if (nconf == (struct netconfig *) NULL)
  rpc_createerr.cf_stat = RPC_UNKNOWNPROTO;
  goto err;
 }
 if (fd == RPC_ANYFD) {
  fd = t_open(nconf->nc_device, O_RDWR, &tinfo);
  if (fd == -1) {
   rpc_createerr.cf_stat = RPC_SYSTEMERROR;
```

(続く)

```
    goto err;
   }
  }
  if (raddr->sin_port == 0) { /* 未知の遠隔アドレス */
   u_short sport;
   /*
    * ユーザー作成のルーチン rpcb_getport() は rpcb_getaddr を呼び出して、
    * netbuf アドレスをホストのバイト順序に従ってポート番号に変換する
    */
   sport = rpcb_getport(raddr, prog, vers, nconf);
   if (sport == 0) {
    rpc_createerr.cf_stat = RPC_PROGUNAVAIL;
    goto err;
   }
   raddr->sin_port = htons(sport);
  }
  /* sockaddr_in を netbuf に変換 */
  tbind = (struct t_bind *) t_alloc(fd, T_BIND, T_ADDR);
  if (tbind == (struct t_bind *) NULL)
   rpc_createerr.cf_stat = RPC_SYSTEMERROR;
   goto err;
  }
  if (t_bind->addr.maxlen < sizeof( struct sockaddr_in))
   goto err;
  (void) memcpy( tbind->addr.buf, (char *)raddr,
                 sizeof(struct sockaddr_in));
  tbind->addr.len = sizeof(struct sockaddr_in);
  /* fd を結合 */
  if (t_bind( fd, NULL, NULL) == -1) {
   rpc_createerr.ct_stat = RPC_TLIERROR;
   goto err;
  }
  cl = clnt_tli_create(fd, nconf, &(tbind->addr), prog, vers,
                       tinfo.tsdu, tinfo.tsdu);
  /* netconfig ファイルを閉じる */
  (void) endnetconfig( handlep);
  (void) t_free((char *) tbind, T_BIND);
  if (cl) {
   *sockp = fd;
   if (madefd == TRUE) {
    /* fd はハンドルの破棄と同時に閉じる */
    (void)clnt_control(cl,CLSET_FD_CLOSE, (char *)NULL);
   }
   /* リトライ時間の設定 */
   (void) clnt_control( l, CLSET_RETRY_TIMEOUT,
                        (char *) &wait);
   return(cl);
  }
 err:
  if (madefd == TRUE)
   (void) t_close(fd);
  (void) endnetconfig(handlep);
```

(続く)

```
  return((CLIENT *) NULL);
}
```

ネットワーク (トランスポート) 選択に
は、setnetconfig()、getnetconfig()、endnetconfig() を使用します。

注 - endnetconfig() の呼び出しは、プログラムの終り近くの
clnt_tli_create() の呼び出しの後で行なっていることに注意してください。

---

clntudp_create() には、オープンしている TLI ファイル記述子を渡すことができます。ファイル記述子が渡されなかった場合 (fd == RPC_ANYFD)
は、t_open() に渡すデバイス名を UDP の netconfig 構造体から取り出して自分でオープンします。

遠隔アドレスがわからない場合 (raddr->sin_port == 0) は、遠隔の rpcbind デーモンを使って取り出します。

クライアントハンドルが作成されれば、clnt_control() を使用してさまざまな変更を加えることができます。RPC ライブラリはハンドルを破棄するときにファイル記述子を閉じ (fd をライブラリ内でオープンしたときは、clnt_destroy() の呼び出しにより閉じられる)、リトライのタイムアウト値を設定します。

サーバー側

コード例 4–13 には、これに対するサーバー側プログラム svcudp_create() を示します。サーバー側では svc_tli_create() を使用します。

svc_tli_create() は、アプリケーションで次のように詳細な制御を行う必要があるときに使用します。

- オープンされたファイル記述子をアプリケーションに渡します。
- ユーザーの結合アドレスを渡します。
- 送信バッファと受信バッファのサイズを指定します。引数 fd は、渡された時に結合されていないことがあります。その場合、fd は指定されたアドレスに結合され、そのアドレスがハンドルに格納されます。結合されたアドレスが NULL に

設定されていて、fd が最初から結合されていない場合、任意の最適なアドレスへ結合されます。

サービスを rpcbind により登録するには、rpcb_set() を使用します。

コード例 **4–13**　下位レベル RPC を使用したサーバー側プログラム

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <netconfig.h>
#include <netinet/in.h>

SVCXPRT *
svcudp_create(fd)
 register int fd;
{
 struct netconfig *nconf;
 SVCXPRT *svc;
 int madefd = FALSE;
 int port;
 void *handlep;
 struct  t_info tinfo;

 /* どのトランスポートも使用不可の場合 */
 if ((handlep = setnetconfig() ) == (void *) NULL) {
  nc_perror("server");
  return((SVCXPRT *) NULL);
 }
 /*
  * 非接続型で、プロトコルファミリが INET、名前が UDP の
  * トランスポートが見つかるまで探す。
  */
 while (nconf = getnetconfig( handlep)) {
  if ((nconf->nc_semantics == NC_TPI_CLTS) &&
      (strcmp( nconf->nc_protofmly, NC_INET) == 0 )&&
      (strcmp( nconf->nc_proto, NC_UDP) == 0 ))
   break;
 }
 if (nconf == (struct netconfig *) NULL) {
  endnetconfig(handlep);
  return((SVCXPRT *) NULL);
 }
 if (fd == RPC_ANYFD) {
  fd = t_open(nconf->nc_device, O_RDWR, &tinfo);
  if (fd == -1) {
   (void) endnetconfig();
   return((SVCXPRT *) NULL);
  }
  madefd = TRUE;
 } else
  t_getinfo(fd, &tinfo);
 svc = svc_tli_create(fd, nconf, (struct t_bind *) NULL,
                      tinfo.tsdu, tinfo.tsdu);
 (void) endnetconfig(handlep);
 if (svc == (SVCXPRT *) NULL) {
```

(続く)

```
  if (madefd)
    (void) t_close(fd);
  return((SVCXPRT *)NULL);
 }
 return (svc);
}
```

この例では、clntudp_create() と同じ方法でネットワーク選択を行なっています。svc_tli_create() で結合しているため、ファイル記述子は明示的にはトランスポートアドレスと結合されません。

svcudp_create() はオープンしている fd を使用できます。有効な fd が渡されなければ、選択された netconfig 構造体を使用してこのルーチン内でオープンします。

ボトムレベルのインタフェース

アプリケーションで RPC のボトムレベルインタフェースを使用すると、すべてのオプションを使用して通信を制御できます。clnt_tli_create() などのエキスパートレベルの RPC インタフェースは、ボトムレベルのルーチンを使用しています。ユーザーがこのレベルのルーチンを直接使用することはほとんどありません。

ボトムレベルのルーチンは内部データ構造を作成し、バッファを管理し、RPC ヘッダーを作成します。ボトムレベルルーチンの呼び出し側 (たとえば、clnt_tli_create()) では、クライアントハンドルの cl_netid と cl_tp の両フィールドを初期化する必要があります。作成したハンドルの cl_netid にはトランスポートのネットワーク ID (たとえば udp) を設定し、cl_tp にはトランスポートのデバイス名 (たとえば /dev/udp) を設定します。clnt_dg_create() と clnt_vc_create() のルーチンは、clnt_ops と cl_private のフィールドを設定します。

クライアント側

コード例 4–14 は、clnt_vc_create() と clnt_dg_create() の呼び出し方法を示します。

コード例 **4–14**　ボトムレベルのルーチンを使用したクライアント作成

```
/*
 * 使用する変数 :
 * cl: CLIENT *
 * tinfo: struct t_info (t_open() または t_getinfo() からの戻り値)
 * svcaddr: struct netbuf *
 */
 switch(tinfo.servtype) {
  case T_COTS:
  case T_COTS_ORD:
   cl = clnt_vc_create(fd, svcaddr,
    prog, vers, sendsz, recvsz);
   break;
  case T_CLTS:
   cl = clnt_dg_create(fd, svcaddr,
    prog, vers, sendsz, recvsz);
   break;
  default:
   goto err;
 }
```

これらのルーチンを使用するときは、ファイル記述子がオープンされて結合されている必要があります。svcaddr はサーバーのアドレスです。

サーバー側

サーバー側は コード例 4–15 のようになります。

コード例 **4–15**　ボトムレベル用のサーバー

```
/*
 * 使用する変数
 * xprt: SVCXPRT *
 */
switch(tinfo.servtype) {
 case T_COTS_ORD:
 case T_COTS:
  xprt = svc_vc_create(fd, sendsz, recvsz);

  break;
 case T_CLTS:
  xprt = svc_dg_create(fd, sendsz, recvsz);

  break;
 default:
  goto err;
```

(続く)

```
}
```

サーバーのキャッシュ

svc_dg_enablecache() はデータグラムトランスポートのキャッシュを開始します。キャッシュは、サーバープロシージャが「一度だけ」行われるバージョンにのみ、使用されるべきです。これは、キャッシュされたサーバープロシージャを何回も実行すると、異なる別の結果を生じるためです。

```
svc_dg_enablecache(xprt, cache_size)
    SVCXPRT *xprt;
    unsigned int cache_size;
```

この関数は、cache_size エントリを保持するために十分な大きさで、サービスのエンドポイント xprt に、重複要求キャッシュを割り当てます。サービスに、異なる戻り値を返す手続きが含まれる場合は、重複要求キャッシュが必要です。キャッシュをいったん有効にすると、後で無効にする方法はありません。

下位レベルのデータ構造

次のデータ構造は参考のために示しますが、変更される可能性があります。

最初に示すのは、クライアント側の RPC ハンドルで、<rpc/clnt.h> で定義されています。下位レベルの RPC を使用する場合は、コード例 4–16 に示したように接続ごとに 1 つのハンドルを作成して初期化する必要があります。

コード例 **4–16**　クライアント側 RPC ハンドル (CLIENT 構造体)

```
typedef struct {
 AUTH *cl_auth;         /* 認証情報 */
 struct clnt_ops {
  enum clnt_stat (*cl_call)();  /* 遠隔手続き呼び出し */
  void   (*cl_abort)();   /* 呼び出しの中止 */
  void   (*cl_geterr)();  /* 特定エラーコードの取得 */
  bool_t   (*cl_freeres)(); /* 戻り値の解放*/
  void   (*cl_destroy)();  /* この構造体の破棄 */
  bool_t   (*cl_control)(); /* RPC の ioctl() */
 } *cl_ops;
 caddrt_t   cl_private;    /* プライベートに使用 */
 char   *cl_netid;      /* ネットワークトークン */
 char   *cl_tp;       /* デバイス名 */
} CLIENT;
```

クライアント側ハンドルの第 1 フィールドは、 <rpc/auth.h> で定義された認証情報の構造体です。このフィールドはデフォルトで AUTH_NONE に設定されているため、コード例 4–17 に示すように、必要に応じてクライアント側プログラムで cl_auth を初期化する必要があります。

コード例 **4–17**　クライアント側の認証ハンドル

```
typedef struct {
 struct   opaque_auth  ah_cred;
 struct   opaque_auth  ah_verf;
 union   des_block    ah_key;
 struct auth_ops {
  void  (*ah_nextverf)();
  int  (*ah_marshal)();   /* nextverf とシリアライズ */
  int  (*ah_validate)();   /* 妥当性検査の確認 */
  int  (*ah_refresh)();   /* 資格のリフレッシュ */
  void  (*ah_destroy)();   /* この構造体の破棄 */
 } *ah_ops;
 caddr_t ah_private;
} AUTH;
```

AUTH 構造体の ah_cred には呼び出し側の資格が、ah_verf には資格を確認するためのデータが入っています。詳細については、106ページの「認証」 を参照してください。

コード例 4–18 には、サーバー側のトランスポートハンドルを示します。

コード例 **4–18** サーバー側のトランスポートハンドル

```
typedef struct {
 int   xp_fd;
#define xp_sock     xp_fd
 u_short xp_port;  /* 結合されたポート番号、旧形式 */
 struct xp_ops {
     bool_t    (*xp_recv)();  /* 要求の受信 */
    enum xprt_stat (*xp_stat)();   /* トランスポートステータスの取得 */
     bool_t    (*xp_getargs)(); /* 引数の取り出し */
     bool_t    (*xp_reply)();  /* 応答の送信 */
     bool_t    (*xp_freeargs)();  /* 引数に割り当てたメモリーの解放* */
     void    (*xp_destroy)();  /* この構造体の破棄 */
 } *xp_ops;
 int  xp_addrlen;   /* 遠隔アドレスの長さ、旧形式 */
 char  *xp_tp;    /* トランスポートプロバイダのデバイス名 */
 char  *xp_netid;   /* ネットワークトークン */
 struct netbuf  xp_ltaddr;  /* ローカルトランスポートアドレス */
 struct netbuf  xp_rtaddr;  /* 遠隔トランスポートアドレス */
 char    xp_raddr[16]; /* 遠隔アドレス、旧形式 */
 struct opaque_auth xp_verf; /* raw 応答の確認 */
 caddr_t    xp_p1;  /* プライベート: svc ops で使用 */
 caddr_t    xp_p2;  /* プライベート: svc ops で使用 */
 caddr_t    xp_p3;  /* プライベート: svc lib で使用 */
} SVCXPRT;
```

表 4–5 は、サーバー側のトランスポートハンドルに対応するフィールドを示します。

表 4–5 RPC サーバー側のトランスポートハンドル

xp_fd	ハンドルに結合したファイル記述子。複数のサーバーハンドルで 1 つのファイル記述子を共有できる
xp_netid	トランスポートのネットワーク ID (たとえば、udp)。ハンドルはこのトランスポート上に作成される。xp_tp は、このトランスポートに結合したデバイス名
xp_ltaddr	サーバー自身の結合アドレス
xp_rtaddr	RPC の呼び出し側アドレス (したがって、呼び出しのたびに変る)
xp_netid xp_tp xp_ltaddr	svc_tli_create() のようなエキスパートレベルのルーチンで初期化される

その他のフィールドは、ボトムレベルのサーバールーチン svc_dg_create() と svc_vc_create() で初期化されます。

接続型端点では、表 4–6 の各フィールドには、接続要求がサーバーに受け入れられるまで正しい値が入りません。

表 4–6 RPC 接続型端点

接続が確立するまでは無効なフィールド		
xp_fd	xp_ops()	xp_p1()
xp_p2	xp_verf()	xp_tp()
xp_ltaddr	xp_rtaddr()	xp_netid()

下位レベルの Raw RPC を使用したプログラムテスト

デバッグツールとして、ネットワーク機能をすべてバイパスする 2 つの擬似 RPC インタフェースがあります。ルーチン clnt_raw_create() と svc_raw_create() は、実際のトランスポートを使用しません。

注 - 製品システムで RAW モードは使用しないでください。RAW モードは、デバッグを行い易くするために使用します。RAW モードはマルチスレッド対応ではありません。

コード例 4–19 は、次の Makefile を使用してコンパイルとリンクが行われます。

```
all: raw
CFLAGS += -g
raw: raw.o
cc -g -o raw raw.o -lnsl
```

コード例 **4–19**　Raw RPC を使用した簡単なプログラム

```
/*
 * 数値を 1 増加させる簡単なプログラム
 */

#include <stdio.h>
#include <rpc/rpc.h>
#include <rpc/raw.h>
#define prognum 0x40000001
#define versnum 1
#define INCR 1

struct timeval TIMEOUT = {0, 0};
static void server();

main (argc, argv)
 int argc;
 char **argv;
{
 CLIENT *cl;
 SVCXPRT *svc;
 int num = 0, ans;
 int flag;

 if (argc == 2)
  num = atoi(argv[1]);
  svc = svc_raw_create();
 if (svc == (SVCXPRT *) NULL) {
  fprintf(stderr, "Could not create server handle\n");
  exit(1);
 }
 flag = svc_reg( svc, prognum, versnum, server,
         (struct netconfig *) NULL );
    if (flag == 0) {
     fprintf(stderr, "Error: svc_reg failed.\n");
     exit(1);
 }
 cl = clnt_raw_create( prognum, versnum );
 if (cl == (CLIENT *) NULL) {
  clnt_pcreateerror("Error: clnt_raw_create");
  exit(1);
 }
 if (clnt_call(cl, INCR, xdr_int, (caddr_t) &num, xdr_int,
       (caddr_t) &ans, TIMEOUT)
   != RPC_SUCCESS) {
  clnt_perror(cl, "Error: client_call with raw");
  exit(1);
 }
 printf("Client: number returned %d\n", ans);
 exit(0);
}

static void
server(rqstp, transp)
 struct svc_req *rqstp;
 SVCXPRT *transp;
```

(続く)

```
{
 int num;

   fprintf(stderr, "Entering server procedure.\n");

 switch(rqstp->rq_proc) {
  case NULLPROC:
   if (svc_sendreply( transp, xdr_void,
    (caddr_t) NULL) == FALSE) {
    fprintf(stderr, "error in null proc\n");
    exit(1);
   }
   return;
  case INCR:
   break;
  default:
   svcerr_noproc(transp);
   return;
 }
 if (!svc_getargs( transp, xdr_int, &num)) {
  svcerr_decode(transp);
  return;
 }
    fprintf(stderr, "Server procedure: about to increment.\n");
 num++;
 if (svc_sendreply(transp, xdr_int, &num) == FALSE) {
  fprintf(stderr, "error in sending answer\n");
  exit (1);
 }
    fprintf(stderr, "Leaving server procedure.\n");
}
```

次の点に注意してください。

- サーバーはクライアントより先に作成しなければなりません。
- `svc_raw_create()` には引数がありません。
- サーバーは `rpcbind` デーモンに登録されません。`svc_reg()` の最後の引数は `(struct netconfig *) NULL` ですので、`rpcbind` デーモンに登録されないことがわかります。
- `svc_run()` が呼び出されません。
- RPC 呼び出しはすべて同一の制御スレッド内で行われます。
- サーバーディスパッチルーチンは通常の RPC サーバーの場合と同じです

RPC プログラミングの高度なテクニック

この節では、RPC の下位レベルインタフェースを使用するさまざまな開発テクニックを説明します。この章で説明する項目を次に示します。

- サーバー上の poll() - サーバー上で svc_run() が呼び出せない場合に、サーバーから直接ディスパッチャを呼び出す方法
- ブロードキャスト RPC - ブロードキャストの使用方法
- バッチ処理 - 一連の呼び出しをバッチ処理にして、パフォーマンスを向上させる方法
- 認証 - 今回のリリースで使用される認証方法
- ポートモニタの使用 - ポートモニタ inetd と listener のインタフェース
- サーバーのバージョン - 複数のプログラムバージョンのサービス方法

サーバー側の poll( )

この節で説明する内容は、(デフォルトで) シングルスレッドのモードで RPC を実行する場合にだけ適用されます。

RPC 要求をサービスしたり、その他のアクティビティを実行したりするプロセスでは、svc_run() を呼び出せない場合があります。他のアクティビティで定期的にデータ構造を更新する場合は、プロセスから svc_run() を呼び出す前に SIGALRM 信号をセットできます。そうすると、シグナルハンドラがデータ構造を処理してから svc_run() に制御を戻します。

プロセスから svc_run() をバイパスして直接ディスパッチャにアクセスするには、svc_getreqset() を呼び出します。待っているプログラムに結合したトランスポート端点のファイル記述子がわかれば、プロセスは自分で poll() を呼び出して、RPC ファイル記述子と自身の記述子の両方で要求を待つことができます。

コード例 4–20 には svc_run() を示します。svc_pollset は、_rpc_select_to_poll() の呼び出しを通して svc_fdset() から派生した pollfd 構造体の配列です。この配列は、RPC ライブラリルーチンのどれかが呼び出されるたびに変わる可能性があります。そのたびに記述子がオープンされ、クローズされるからです。poll() がいくつかの RPC ファイル記述子への RPC 要求の到着を確認すると、svc_getreq_poll() が呼び出されます。

注 - 関数 _rpc_dtbsize() と _rpc_select_to_poll() は、SVID の一部ではありませんが、libnsl ライブラリで使用できます。Solaris 以外でも実行できるように、これらの関数を作成するために、関数の仕様を説明します。

```
int __rpc_select_to_poll(int fdmax, fd_set *fdset,
                struct pollfd *pollset)
```

ビットフラグとして fd_set ポインタとチェックすべきビット数が指定されます。この関数内で、指定された pollfd 配列を RPC が使用するために初期化するようにします。RPC は、入力イベントだけをポーリングします。初期化された pollfd スロット数が返されます。

```
int __rpc_dtbsize()
```

この関数は、getrlimit() 関数を呼び出し、新しく作成された記述子にシステムが割り当てる最大値を決定します。結果は、効率化のためにキャッシュされます。

この節の SVID ルーチンについての詳細は、rpc_svc_calls(3N) および poll(2) のマニュアルページを参照してください。

コード例 4–20 svc_run( ) と poll( )

```
void
svc_run()
{
 int nfds;
 int dtbsize = __rpc_dtbsize();
 int i;
 struct pollfd svc_pollset[fd_setsize];

 for (;;) {
  /*
   * 要求待ちするサーバー fd があるかどうかをチェック
   */
  nfds = __rpc_select_to_poll(dtbsize, &svc_fdset,
                              svc_pollset);
  if (nfds == 0)
   break; /* 要求待ちの fd がないので終了 */

  switch (i = poll(svc_pollset, nfds, -1)) {
  case -1:
```

(続く)

```
    /*
     * エラーが起こった場合は、poll() ではなく、シグナルハンドラなど
     *  外部イベントによるものと考えて、無視して継続する
     */
   case 0:
    continue;
   default:
    svc_getreq_poll(svc_pollset, i);
   }
  }
}
```

ブロードキャスト RPC

RPC のブロードキャストが要求されると、メッセージはネットワーク上の rpcbind デーモンに送られます。要求されたサービスが登録されている rpcbind デーモンは、その要求をサーバーに送ります。ブロードキャスト RPC と通常の RPC 呼び出しとの主な相違点を次に示します。

- 通常の RPC では応答は 1 つですが、ブロードキャスト RPC には複数の応答があります (メッセージに応答するすべてのマシンから応答が返されます)。
- ブロードキャスト RPC は、UDP のようにブロードキャスト RPC をサポートする非接続型プロトコルでしか使用できません。
- ブロードキャスト RPC では、正常終了以外の応答は返されません。したがって、ブロードキャスタと遠隔のサービスでバージョンの不一致があれば、サービスからブロードキャスタには何も返されません。
- ブロードキャスト RPC では、rpcbind で登録されたデータグラムサービスだけがアクセス可能です。サービスアドレスはホストごとに異なりますので、rpc_broadcast() は、rpcbind のネットワークアドレスにメッセージを送信します。
- ブロードキャスト要求のサイズはローカルネットワークの最大伝送ユニット (MTU:maximum trasfer unit) により制限されます。イーサネットの MTU は 1500 バイトです。

コード例 4–21 では、rpc_broadcast() の使用方法を示し、引数を説明します。

コード例 **4–21**　RPC ブロードキャスト

```
/*
 * bcast.c: RPC ブロードキャストの使用例
 */

#include <stdio.h>
#include <rpc/rpc.h>

main(argc, argv)
 int argc;
 char *argv[];
{
 enum clnt_stat rpc_stat;
 rpcprog_t prognum;
 rpcvers_t vers;
 struct rpcent *re;

 if(argc != 3) {
  fprintf(stderr, "usage : %s RPC_PROG VERSION\n", argv[0]);
  exit(1);
 }
 if (isdigit( *argv[1]))
  prognum = atoi(argv[1]);
 else {
  re = getrpcbyname(argv[1]);
  if (! re) {
   fprintf(stderr, "Unknown RPC service %s\n", argv[1]);
   exit(1);
  }
  prognum = re->r_number;
 }
 vers = atoi(argv[2]);
 rpc_stat = rpc_broadcast(prognum, vers, NULLPROC, xdr_void,
            (char *)NULL, xdr_void, (char *)NULL, bcast_proc,
NULL);
 if ((rpc_stat != RPC_SUCCESS) && (rpc_stat != RPC_TIMEDOUT)) {
  fprintf(stderr, "broadcast failed: %s\n",
           clnt_sperrno(rpc_stat));
  exit(1);
 }
 exit(0);
}
```

コード例 4–22 の関数 `bcast_proc()` では、ブロードキャストに対する応答を収集します。通常は、最初の応答だけを取り出すか、応答をすべて収集します。`bcast_proc()` は、応答を返したサーバーの IP アドレスを表示します。この関数は `FALSE` を返して応答の収集を続け、RPC クライアントコードはタイムアウトになるまでブロードキャストを再送信し続けます。

コード例 **4–22**　ブロードキャストへの応答の収集

```
bool_t
bcast_proc(res, t_addr, nconf)
 void *res;          /* 応答なし */
 struct t_bind *t_addr;      /* 応答したアドレス */
 struct netconfig *nconf;
{
 register struct hostent *hp;
 char *naddr;

 naddr = taddr2naddr(nconf, &taddr->addr);
 if (naddr == (char *) NULL) {
  fprintf(stderr,"Responded: unknown\n");
 } else {
  fprintf(stderr,"Responded: %s\n", naddr);
  free(naddr);
 }
 return(FALSE);
}
```

TRUE が返されるとブロードキャストは終了し、rpc_broadcast() は正常終了します。FALSE が返された場合は、次の応答を待ちます。数秒間待ってから、要求が再びブロードキャストされます。応答が返されない場合は、rpc_broadcast() は RPC_TIMEDOUT を返します。

バッチ処理

RPC の設計方針では、クライアントは呼び出しメッセージを送信して、サーバーがそれに応答するのを待ちます。すなわち、サーバーが要求を処理する間、クライアントは停止していることになります。これは、クライアントが各メッセージへの応答を待つ必要がないときには非効率です。

RPC のバッチ処理を使用すると、クライアントは非同期に処理を進めることができます。RPC メッセージは呼び出しパイプラインに入れてサーバーに送られます。バッチ処理では次のことが必要になります。

- サーバーはどのような中間メッセージにも応答しません。
- 呼び出しパイプラインは、信頼性の高いトランスポート (たとえば、TCP) で伝送されなければなりません。
- 呼び出し時に指定する、戻り値に対する XDR ルーチンは NULL でなければなりません。

- RPC 呼び出しのタイムアウト値はゼロでなければなりません。

サーバーはそれぞれの呼び出しに対しては応答しないので、クライアントは、サーバーが前の呼び出しを処理している間に平行して次の呼び出しを送信できます。トランスポートは複数の呼び出しメッセージをバッファリングし、システムコール `write()` で一度にサーバーに送信します。このため、プロセス間通信のオーバヘッドが減少し、一連の呼び出しに要する総時間が短縮されます。クライアントは終了前に、パイプラインをフラッシュする呼び出しをバッチにしないで実行します。

コード例 4–23 には、バッチ処理を使用しないクライアント側プログラムを示します。文字配列 `buf` を走査して文字列を順に取り出し、1 つずつサーバーに送信します。

コード例 4–23 バッチ処理を使用しないクライアントプログラム

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "windows.h"

main(argc, argv)
 int argc;
 char **argv;
{
 struct timeval total_timeout;
 register CLIENT *client;
 enum clnt_stat clnt_stat;
 char buf[1000], *s = buf;

 if ((client = clnt_create( argv[1], WINDOWPROG, WINDOWVERS,
        "circuit_v")) == (CLIENT *) NULL) {
  clnt_pcreateerror("clnt_create");
  exit(1);
 }

 total_timeout.tv_sec = 20;
 total_timeout.tv_usec = 0;
 while (scanf( "%s", s ) != EOF) {
  if (clnt_call(client, RENDERSTRING, xdr_wrapstring, &s,
     xdr_void, (caddr_t) NULL, total_timeout) != RPC_SUCCESS) {
   clnt_perror(client, "rpc");
   exit(1);
  }
 }

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

コード例 4–24 には、このクライアントプログラムでバッチ処理を使用する場合を示します。各文字列の送信には応答を待たず、サーバーからの終了応答だけを待ちます

コード例 **4–24**　バッチ処理を使用するクライアントプログラム

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "windows.h"

main(argc, argv)
 int argc;
 char **argv;
{
 struct timeval total_timeout;
 register CLIENT *client;
 enum clnt_stat clnt_stat;
 char buf[1000], *s = buf;

 if ((client = clnt_create( argv[1], WINDOWPROG, WINDOWVERS,
         "circuit_v")) == (CLIENT *) NULL) {
  clnt_pcreateerror("clnt_create");
  exit(1);
 }
 timerclear(&total_timeout);
 while (scanf("%s", s) != EOF)
  clnt_call(client, RENDERSTRING_BATCHED, xdr_wrapstring,
             &s, xdr_void, (caddr_t) NULL, total_timeout);
 /* ここでパイプラインをフラッシュ*/
 total_timeout.tv_sec = 20;
 clnt_stat = clnt_call(client, NULLPROC, xdr_void,
          (caddr_t) NULL, xdr_void, (caddr_t) NULL,
total_timeout);
 if (clnt_stat != RPC_SUCCESS) {
  clnt_perror(client, "rpc");
  exit(1);
 }
 clnt_destroy(client);
 exit(0);
}
```

コード例 4–25 には、バッチ処理を使用した場合のサーバーのディスパッチ部分を示します。サーバーは、メッセージを送信しないので、クライアント側は、失敗に気付きません。

コード例 **4–25**　バッチ処理を行うサーバー

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "windows.h"

void
windowdispatch(rqstp, transp)
 struct svc_req *rqstp;
 SVCXPRT *transp;
{
 char    *s = NULL;

 switch(rqstp->rq_proc) {
  case NULLPROC:
   if (!svc_sendreply( transp, xdr_void, NULL))
    fprintf(stderr, "can't reply to RPC call\n");
   return;
  case RENDERSTRING:
   if (!svc_getargs( transp, xdr_wrapstring, &s)) {
    fprintf(stderr, "can't decode arguments\n");
    /* 呼び出し側にエラーを通知 */
    svcerr_decode(transp);
    break;
   }
   /* 文字列 s を処理するコード */
   if (!svc_sendreply( transp, xdr_void, (caddr_t) NULL))
    fprintf( stderr, "can't reply to RPC call\n");
   break;
  case RENDERSTRING_BATCHED:
   if (!svc_getargs(transp, xdr_wrapstring, &s)) {
    fprintf(stderr, "can't decode arguments\n");
    /* プロトコルエラーのため何も返さない */
    break;
   }
   /* 文字列 s を処理するコード。ただし応答はしない。 */
   break;
  default:
   svcerr_noproc(transp);
   return;
 }
 /* 引数の復号化で割り当てた文字列を解放 */
 svc_freeargs(transp, xdr_wrapstring, &s);
}
```

バッチ処理のパフォーマンス

バッチ処理によるパフォーマンスの向上を調べるために、コード例 4–23、コード例 4–25 で 25144 行のファイルを処理しました。このサービスは、ファイルの各行を引き渡すだけの簡単なサービスです。バッチ処理を使用した方が、使用しない場合の 4 倍の速さで終了しました。

認証

この章でこれまでに示した例では、呼び出し側は自分自身の ID をサーバーに示さず、サーバーも呼び出し側の ID を要求しませんでした。ネットワークサービスによっては、ネットワークファイルシステムのように、呼び出し側の ID が要求される場合があります。『*Solaris* のシステム管理』を参照して、この節で説明したいずれかの認証の方法を実行してください。

RPC のクライアントとサーバーを作成するときにさまざまなトランスポートを指定できるように、RPC クライアントにもさまざまなタイプの認証メカニズムを採用できます。RPC の認証サブシステムは端点が開かれているので、認証はさまざな使用法がサポートされます。認証プロトコルは、付録 B で詳細に定義されています。

RPC が現在サポートしている認証タイプを 表 4–7 に示します。

表 4–7 RPC が現在サポートしている認証タイプ

AUTH_NONE	デフォルト。認証は実行されない
AUTH_SYS	UNIX オペレーティングシステムのプロセスアクセス権を基にした認証タイプ
AUTH_SHORT	サーバーによっては効率向上のため AUTH_SYS の代わりに AUTH_SHORT を使用できる。AUTH_SYS 認証を使用するクライアントプログラムは、サーバーからの AUTH_SHORT 応答ベリファイアを受信できる。詳細は、付録 B を参照
AUTH_DES	DES 暗号化技法を基にした認証タイプ
AUTH_KERB	DES フレームワークを基にした Version 4 Kerberos 認証形式

呼び出し側が次の方法で RPC クライアントハンドルを新規作成すると、

```
clnt = clnt_create(host, prognum, versnum, nettype);
```

対応するクライアント作成ルーチンが次のように認証ハンドルを設定します。

```
clnt->cl_auth = authnone_create();
```

新たな認証インスタンスを作成するときは、auth_destroy(clnt->cl_auth) を使用して現在のインスタンスを破棄します。この操作はメモリーの節約のために必要です。

サーバー側では、RPC パッケージがサービスディスパッチルーチンに、任意の認証スタイルが結合されている要求を渡します。サービスディスパッチルーチンに渡された要求ハンドルには、rq_cred という構造体が入っています。その構成は、認証資格タイプを示すフィールドを除いて、ユーザーから隠されています。

```
/*
 * 認証データ
 */
struct opaque_auth {
   enum_t    oa_flavor;  /* 資格スタイル */
   caddr_t   oa_base;   /* より詳細な認証データのアドレス */
   u_int     oa_length;  /* 最大 MAX_AUTH_BYTES まで */
};
```

RPC パッケージでは、サービスディスパッチルーチンに対して次のことを保証しています。

- svc_req 構造内の rq_cred フィールドは完全に設定済みです。したがって、rq_cred.oa_flavor を調べて認証タイプを得ることができます。得られた認証タイプが RPC でサポートされていない場合は、rq_cred のその他のフィールドも調べることができます。
- サービス手続きに引き渡される rq_clntcred フィールドには NULL が入っているか、サポートされている認証資格タイプの設定済み構造体へのポインタが入っています。AUTH_NONE タイプには認証データはありません。rq_clntcred は、authsys_parms、short_hand_verf、 authkerb_cred、authdes_cred の各構造体へのポインタにだけキャストできます。

AUTH_SYS タイプの認証

クライアント側で AUTH_SYS (旧バージョンでは AUTH_UNIX) タイプの認証を使用するには、RPC クライアントハンドルの作成後に clnt-->cl_auth を次のように設定します。

```
clnt->cl_auth = authsys_create_default();
```

以降は、この clnt を使用した RPC 呼び出しでは、clnt とともに以下に示す資格 - 認証構造体コード例 4–26 が渡されます。

コード例 **4–26**　AUTH_SYS タイプの資格 - 認証構造体

```
/*
 * AUTH_SYS タイプの資格
 */
struct authsys_parms {
 u_long aup_time;     /* 資格作成時刻 */
 char *aup_machname;     /* クライアント側のホスト名 */
 uid_t aup_uid;       /* クライアント側の実効 uid */
 gid_t aup_gid;       /* クライアント側の現在のグループ ID */
 u_int aup_len;       /* aup_gids の配列の長さ */
 gid_t *aup_gids;     /* ユーザーが所属するグループの配列 */
};
```

rpc.broadcast では、デフォルトで AUTH_SYS タイプの認証になります。

コード例 4–27 には、手続きを使用し、ネットワーク上のユーザー数を返すサーバープログラムである RUSERPROC_1() を示します。認証の例として AUTH_SYS タイプの資格をチェックし、呼び出し側の uid が 16 の場合は要求に応じないようにしてあります。

コード例 **4–27**　認証データをチェックするサーバープログラム

```
nuser(rqstp, transp)
 struct svc_req *rqstp;
 SVCXPRT *transp;
{
 struct authsys_parms *sys_cred;
 uid_t uid;
 unsigned int nusers;

 /* NULLPROC の場合は認証データなし */
 if (rqstp->rq_proc == NULLPROC) {
  if (!svc_sendreply( transp, xdr_void, (caddr_t) NULL))
   fprintf(stderr, "can't reply to RPC call\n");
  return;
 }

 /* ここで uid を取得 */
 switch(rqstp->rq_cred.oa_flavor) {
  case AUTH_SYS:
   sys_cred = (struct authsys_parms *) rqstp->rq_clntcred;
   uid = sys_cred->aup_uid;
   break;
```

(続く)

```
  default:
   svcerr_weakauth(transp);
   return;
 }
 switch(rqstp->rq_proc) {
  case RUSERSPROC_1:
   /* 呼び出し側が、この手続きの呼び出し資格を持っているかどうか確認 */
   if (uid == 16) {
    svcerr_systemerr(transp);

    return;
   }
   /*
    * ユーザー数を求めて変数 nusers に設定するコード
    */
   if (!svc_sendreply( transp, xdr_u_int, &nusers))
    fprintf(stderr, "can't reply to RPC call\n");
   return;
  default:
   svcerr_noproc(transp);
   return;
 }
}
```

このプログラムでは次の点に注意してください。

- NULLPROC (手続き番号はゼロ) に結合した認証パラメータは、通常はチェックされません。
- 認証パラメータのタイプが弱すぎる場合、サーバーは svcerr_weakauth() を呼び出します。サーバーが要求する認証タイプのリストを取り出す方法はありません。
- サービスプロトコルでは、アクセスが拒否された場合のステータスを返さなければなりません。コード例 4–27 のプロトコルでは、その代わりにサービスプリミティブ svcerr_systemerr() を呼び出しています。

最後の点で重要なのは、RPC の認証パッケージとサービスの関係です。RPC は認証を処理しますが、個々のサービスへのアクセス制御は行いません。サービス自体でアクセス制御の方針を決め、それがプロトコル内で戻り値として反映されるようにしなければなりません。

AUTH_DES タイプの認証

AUTH_SYS タイプより厳しいセキュリティレベルが要求されるプログラムでは、AUTH_DES タイプの認証を使用します。AUTH_SYS タイプは AUTH_DES タイプに簡単に変更できます。たとえば、authsys_create_default() を使用する代わりに、プログラムから authsys_create() を呼び出し、RPC 認証ハンドルを変更して目的のユーザー ID とホスト名を設定することができます。

AUTH_DES タイプの認証を使用するには、サーバー側とクライアント側の両方のホストで、keyserv() デーモンと NIS または NIS+ ネームサービスが実行されている必要があります。また、両方のホスト上のユーザーに対してネットワーク管理者が割り当てた公開鍵 / 秘密鍵ペアが、publickey() のデータベースに入っていなければなりません。ユーザーは keylogin() のコマンドを実行して自分の秘密鍵を暗号化しておく必要があります。通常は、ログインパスワードと Secure RPC パスワードが同一の場合には、これを login() で行います。

AUTH_DES タイプの認証を使用するには、クライアントが認証ハンドルを正しく設定しなければなりません。その例を次に示します。

```
cl->cl_auth = authdes_seccreate(servername, 60, server,
              (char *)NULL);
```

最初の引数は、サーバープロセスのネットワーク名か、サーバープロセスの所有者のネット名です。サーバープロセスは通常 root プロセスで、次の関数呼び出しでネット名を得ることができます。

```
char servername[MAXNETNAMELEN];
host2netname(servername, server, (char *)NULL);
```

servername は受信文字列へのポインタで、server はサーバープロセスが実行されているホスト名です。サーバープロセスがスーパーユーザー以外のユーザーから起動されている場合は、次のように user2netname() を呼び出します。

```
char servername[MAXNETNAMELEN];
user2netname(servername, serveruid(), (char *)NULL);
```

serveruid() はサーバープロセスのユーザー id です。どちらの関数も最後の引数は、サーバーを含むドメイン名です。NULL を指定すると、ローカルドメイン名が使用されます。

authdes_seccreate() の第 2 引数では、このクライアントの資格の存在時間 (ウィンドウとも呼ばれる) を指定します。この例では 60 秒が指定されているので、この資格はクライアント側が RPC 呼び出しを行なってから、60 秒間で失われます。プログラムから再びこの資格を使用しようとしても、サーバー側の RPC サブシステムは、資格がすでに失われていることを知って、資格を失ったクライアントからの要求に答えません。また資格の存在時間中に別のプログラムがその資格を再使用しようとしても拒否されます。サーバー側の RPC サブシステムが最近作成された資格を保存していて、重複して使用できないようにするためです。

authdes_seccreate() の第 3 引数は、クロックを同期させる timehost 名です。AUTH_DES タイプの認証を使用するには、サーバーとクライアントの時間が一致していなければなりません。この例では、サーバーに同期させています。(char *)NULL と指定すると同期しません。この指定は、クライアントとサーバーがすでに同期していることが確実な場合にだけ行なってください。

authdes_seccreate() の第 4 引数は、タイムスタンプとデータとを暗号化するための DES 暗号化キーへのポインタです。この例のように (char *)NULL と指定した場合は、ランダムキーが選択されます。このキーは、認証ハンドルの ah_key フィールドに入っています。

サーバー側はクライアント側より簡単です。コード例 4–27 のサーバーを AUTH_DES タイプの認証を使用するように変更したものを、コード例 4–28 に示します。

コード例 4–28 AUTH_DES タイプの認証を使用するサーバー

```
#include <rpc/rpc.h>
 ...
 ...
nuser(rqstp, transp)
 struct svc_req *rqstp;
 SVCXPRT *transp;
{
 struct authdes_cred *des_cred;
 uid_t uid;
 gid_t gid;
 int gidlen;
```

(続く)

```
gid_t gidlist[10];

/* NULLPROC の場合は認証データなし */
if (rqstp->rq_proc == NULLPROC) {
 /* 元のプログラムと同じ */
}
/* ここで uid を取得 */
switch(rqstp->rq_cred.oa_flavor) {
 case AUTH_DES:
  des_cred = (struct authdes_cred *) rqstp->rq_clntcred;
  if (! netname2user( des_cred->adc_fullname.name, &uid,
                      &gid, &gidlen, gidlist)) {
   fprintf(stderr, "unknown user: %s\n",
            des_cred->adc_fullname.name);
   svcerr_systemerr(transp);
   return;
  }
  break;
 default:
  svcerr_weakauth(transp);
  return;
}
/* 以降は元のプログラムと同じ */
```

netname2user() ルーチンは、ネットワーク名 (またはユーザーの *netname*) をローカルシステム ID に変換することに注意してください。このルーチンはグループ ID も返します (この例では使用していません)。

AUTH_KERB 認証形式

SunOS 5.*x* は、klogin 以外の Kerberos 4.0 の大部分のクライアント側機能をサポートします。AUTH_KERB は AUTH_DES と概念的に同じです。主要な違いは、DES がネットワーク名と暗号化された DES セッションキーを引き渡すのに対し、Kerberos は、暗号化されたサービスチケットを引き渡すことです。実装状態と相互運用性に影響を及ぼすその他の要因については、続けて説明します。

詳細は、kerberos(3N) マニュアルページと MIT Project Athena implementation of Kerberos の Steiner-Neuman-Shiller 報告書[1]を参照してください。MIT 文書には、athena-dist.mit.edu 上の FTP ディレクトリ、/pub/kerberos/doc また

1. Steiner, Jennifer G., Neuman, Clifford, and Schiller, Jeffrey J. "Kerberos: An Authentication Service for Open Network Systems." *USENIX Conference Proceedings*, USENIX Association, カリフォルニア州, バークレー, June 1988.

は、ドキュメント URL、ftp://athena-dist.mit.edu/pub/kerberos/doc を使用して Mosaic でアクセスできます。

時刻の同期化

Kerberos はその資格が有効である時間ウィンドウの概念を使用します。クライアントまたはサーバーのクロックを制限しません。クライアントは、サーバーに指定されたウィンドウの時間を調整することによって、自身とサーバー間のずれを決定し、サーバーに指定されたウィンドウ時間を調整することによって、この違いを補う必要があります。具体的には、window を authkerb_seccreate() に引き数として渡します。この場合、ウィンドウは変わりません。timehost が authkerb_seccreate() の引き数として指定されると、クライアント側は timehost から時刻を取得して、時刻の差異によってタイムスタンプを変更します。時刻を同期化するには、さまざまな方法が使用できます。詳細は、kerberos_rpc(3N) マニュアルページを参照してください。

周知の名前

Kerberos ユーザーは、一次名、インスタンス、領域によって識別されます。RPC 認証コードは、領域とインスタンスを無視しますが、Kerberos ライブラリコードは無視しません。ユーザー名は、クライアントとサーバー間で同じであると仮定します。これによって、サーバーは一次名をユーザー ID 情報に変換することができます。周知の名前として 2 つの書式が使用されます (領域は省略されます)。

- root.*host* は、クライアント側 *host* の特権を与えられたユーザーを表します。
- *user*.ignored は、ユーザー名が *user* であるユーザーを表します。インスタンは無視されます。

暗号化

Kerberos は、完全資格名 (チケットとウィンドウを含むもの) の送信時に暗号文ブロックチェイン (CBC: Cipher Block Chaining) モード、それ以外の場合は、電子コードブック (ECB: Electronic Code Book) モードを使用します。CBC と ECB は、DES 暗号化のための 2 つの方法です。詳細は、des_crypt(3) マニュアルページを参照してください。セッションキーは、CBC モードに対する初期入力ベクトルとして使用されます。表記は次のようになります。

```
xdr_type(object)
```

これは、XDR が object を type とみなして使用されることを示します。次のコードセクションの長さ (資格またはベリファイアのバイト数) を、4 バイト単位に丸めらたサイズで表されます。完全資格名およびベリファイアは、次のようになります。

```
xdr_long(timestamp.seconds)
xdr_long(timestamp.useconds)
xdr_long(window)
xdr_long(window - 1)
```

セッションキーに等しい入力ベクトルを持つ CBC で暗号化を行うと、出力結果は次のような 2 つの DES 暗号化ブロックになります。

```
CB0
CB1.low
CB1.high
```

資格は、次のようになります。

```
xdr_long(AUTH_KERB)
xdr_long(length)
xdr_enum(AKN_FULLNAME)
xdr_bytes(ticket)
xdr_opaque(CB1.high)
```

ベリファイアは、次のようになります。

```
xdr_long(AUTH_KERB)
xdr_long(length)
xdr_opaque(CB0)
xdr_opaque(CB1.low)
```

ニックネーム交換によって、次のように生成されます。

```
xdr_long(timestamp.seconds)
xdr_long(timestamp.useconds)
```

ニックネームは、ECB によって暗号化され、ECB0 と資格を得ます。

```
xdr_long(AUTH_KERB)
xdr_long(length)
xdr_enum(AKN_NICKNAME)
xdr_opaque(akc_nickname)
```

ベリファイアは、次のようになります。

```
xdr_long(AUTH_KERB)
xdr_long(length)
xdr_opaque(ECB0)
xdr_opaque(0)
```

RPCSEC_GSS を使用した認証

上述の認証タイプ (AUTH_SYS、AUTH_DES、AUTH_KERB) は、1 つの決まった見方で同じように扱うことができます。このため、新しいネットワーキング階層、Generic Security Standard API (汎用セキュリティ規格 API)、すなわち GSS-API が追加されています。GSS-API のフレームワークでは、認証に加え次の 2 つの「サービス」が提供されています。

- 「完全性」

  一貫性サービスでは、GSS-API は下位層のメカニズムを使用してプログラム間で交換されるメッセージを認証します。暗号化チェックサムによって、以下が確立されます。

  - データの発信側から受信側への識別情報 (ID)
  - 受信側から発信側への識別情報 (ID) (相互の認証が要求された場合)

- 伝送されたデータそのものの認証

- 「プライバシ」

  プライバシサービスには、完全性サービスが含まれています。これに加えて、伝送データも「暗号化」され傍受者から保護されます。

また、GSS-API を使用すると、それがサポートする kerberos V 5、RSA 公開鍵、Diffie-Hellman 公開鍵、CSM や将来サポートされるメカニズムをアプリケーションから使用することができます。[2]

RPCSEC_GSS セキュリティタイプを使用すると、ONC RPC アプリケーションは GSS-API の機能の利点を利用することができます。RPCSEC_GSS は、次のように GSS-API 階層の「最上部」に位置します。

2. 現在、GSS-API はまだ発表されていません。ただし、特定の GSS-API 機能は RPCSEC_GSS の機能 (この機能は「不透明な」(内部の見えない) 形で扱うことができます) を通じて参照できます。プログラマはこれらの値に直接かかわる必要はありません。

図 4–1　GSS-API と RPCSEC-GSS のセキュリティ階層

RPCSEC-GSS のプログラミングインタフェースを使用する場合は、ONC RPC アプリケーションは以下の項目を指定できます。

メカニズム　セキュリティのパラダイム。各種セキュリティメカニズムでは、1 つまたは複数レベルのデータ保護と同時に、それぞれ異なる種類のデータ保護を提供します。この場合、GSS-API によってサポートされる任意のセキュリティメカニズムを指定します (Kerberos v5、RSA 公開鍵など)。

サービス　プライバシまたは完全性のいずれかを指定します (あるいはどちらも指定しない)。この項目はメカニズムに依存しません。

QOP

QOP (Quality of Protection) は、保護の質を意味します。QOP により、プライバシまたは完全性サービスとともに使用する暗号化アルゴリズムのタイプが指定されます。各セキュリティメカニズムには、それに関連する 1 つまたは複数の QOP があります。

アプリケーションは、構成ファイル (`/etc/gss/qop` と `/etc/gss/mech`) またはネームサービスから、有効な QOP とメカニズムを入手できます。開発者は、メカニズムと QOP をハードコード化し、使用するアプリケーション内に埋め込むことは避けてください。そうすれば、新しい、または異なるメカニズムおよび QOP を使用するためにアプリケーションを修正する必要はありません。

注 - これまでは、「セキュリティタイプ」と「認証タイプ」は同じものを表していました。RPCSEC_GSS の導入によって、「タイプ」は現在、多少異なる意味を持ちます。タイプには、認証とともにサービス (一貫性またはプライバシ) を含むことができますが、現在は RPCSEC_GSS が、これを実行できる唯一のタイプです。

RPCSEC_GSS を使用すると、ONC RPC アプリケーションは、他のタイプを使用して行う場合と同様に、ピアにセキュリティコンテキストを確立し、データを交換し、そしてこのコンテキストを破壊します。一度コンテキストが確立されると、アプリケーションは、送信したデータユニットごとに QOP およびサービスを変更できます。

RPCSEC_GSS データタイプを含む RPCSEC_GSS の詳細については、`rpcsec_gss(3N)` のマニュアルページを参照してください。

RPCSEC_GSS ルーチン

表 4–8 は、RPCSEC_GSS コマンドを要約したものです。この表では、各関数の個別の説明ではなく、RPCSEC_GSS 関数の全般的な概要を示しています。詳細については、各関数のマニュアルページを参照してください。

表 4–8 RPCSEC_GSS Functions

関数	目的	入力	出力	注
rpc_gss_seccreate()	セキュリティコンテキストの作成	クライアントのハンドル、主体名、メカニズム、QOP、サービスタイプ	AUTH ハンドル	
rpc_gss_set_defaults()	コンテキストの QOP とサービスタイプの変更	古い QOP とサービス	新しい QOP とサービス	クライアント側
rpc_gss_max_data_length()	セキュリティの変換前に、データの最大サイズを示す	伝送できる最大データサイズ	変換前の最大データサイズ	クライアント側
rpc_gss_svc_max_data_length()	セキュリティの変換前に、データの最大サイズを示す	伝送できる最大データサイズ	変換前の最大データサイズ	サーバー側
rpc_gss_set_svc_name()	表示するサーバーの主体名を設定する	主体名 , RPC プログラム、バージョン番号	正常に完了した場合は TRUE	サーバー側
rpc_gss_getcred()	呼び出し元 (クライアント) の資格を得る	svc_req 構造へのポインタ	UNIX 資格、RPCSEC_GSS 資格、cookie	
rpc_gss_set_callback()	(ユーザーの作成した) コールバック関数を指定する	コールバック関数へのポインタ	正常に完了した場合は TRUE	
rpc_gss_get_principal_name()	固有のパラメータから主体名の RPCSEC_GSS 構造を作成する	メカニズム、ユーザー名、マシン名、ドメイン名	RPCSEC_GSS 主体名の構造	

表 4–8 RPCSEC_GSS Functions 続く

関数	目的	入力	出力	注
rpc_gss_get_error()	RPCSEC_GSS ルーチンが失敗した場合にエラーコードを得る		RPCSEC_GSS エラー番号、該当する場合には errno	
rpc_gss_get_mechanisms()	インストールされているメカニズムの文字列を入手する		有効なメカニズムのリスト	
rpc_gss_get_mech_info()	有効な QOP 文字列を入手する	メカニズム	そのメカニズムの有効な QOP	
rpc_gss_get_versions()	サポートされている RPCSEC_GSS の最大および最小のバージョン番号を得る		最大および最小のバージョン番号	
rpc_gss_is_installed()	メカニズムが導入されているかどうかをチェックする	メカニズム	インストールされている場合は TRUE	
rpc_gss_mech_to_oid()	ASCII メカニズムを RPC オブジェクト識別子に変換する	メカニズム (文字列で)	メカニズム (OID で)	
rpc_gss_qop_to_num()	ASCII QOP を整数に変換する	QOP (文字列で)	QOP (整数で)	

コンテキストの作成

コンテキストは、rpc_gss_seccreate() 呼び出しを使用して作成します。この関数では引数として、セッションの、クライアントハンドル (たとえば

clnt_create() によって戻されたもの) , サーバーの主体名 (nfs@machine.eng.company.com)、メカニズム、サービスタイプ、および QOP と、使用される場合はほとんどの場合不透明なまま使用される 2 つの GSS-API パラメータをとります。この関数で、AUTH 認証ハンドルを返します。以下のコードフラグメントは、Kerberos v5 セキュリティメカニズムと完全性サービスを使用したコンテキストを作成する場合、rpc_gss_seccreate() がどのように使用されるかを示しています。

```
CLIENT *clnt;      /* クライアントハンドル */
char server_host[] = "foo";
char service_name[] = "nfs@machine.eng.company.com";
char mech[] = "kerberosv5";

clnt = clnt_create(server_host, SERVER_PROG, SERV_VERS, "netpath");
clnt->clnt_auth = rpc_gss_seccreate(clnt, service_name, mech, \
               rpc_gss_svc_integrity, NULL, NULL, NULL);

. . .
```

この例では、注意する点が数カ所あります。まず、メカニズムは明示的に宣言してありますが (読みやすくするために)、通常は、rpc_gss_get_mechanisms() 用いて、使用できるメカニズムの表から入手します。QOP にも同様のことが言えます。 ここでは、最初の NULL を渡しています。これは、QOP をこのメカニズムのデフォルトに設定するものです。次に、サービスタイプは、RPCSEC_GSS タイプの enum のひとつである rpc_gss_service_t です。最後に、末尾の 2 つの NULL 引数は、不透明な GSS-API オプション用で、プログラマは安全に NULL を値として渡すことができます。

値の変更とコンテキストの破棄

コンテキストが設定されると、アプリケーションは伝送される個々のデータユニットの QOP およびービス値を変更できます。たとえば、パスワードは暗号化したいがログイン名は暗号化したくない場合。これは、次のように rpc_gss_set_defaults() を使用すると実行できます。

```
rpc_gss_set_defaults(clnt->clnt_auth, rpc_gss_svc_privacy, qop);
. . .
```

ここで、 qop は新しい QOP の名前を表わす文字列へのポインタです。

コンテキストは、通常どおり、auth_destroy() を使用して破棄します。

主体名

セキュリティコンテキストを確立し、保持するには、次の 2 つのタイプの主体名が必要です。

- 「サーバー」主体名

  サーバーの主体名は、通常、「service@host」の形式の NULL で終わる ASCII 文字列で指定します。たとえば、 nfs@dalkey.eng.company.com のように指定します。

  クライアントが rpc_gss_seccreate() でセキュリティコンテキストを作成する時に、この方法でサーバーの主体名を指定します。同様にサーバーは、表示する主体名を設定する必要がある場合は、引数としてこのフォーマットの主体名をとる rpc_gss_set_svc_name() を使用します。

- 「クライアント」主体名

  サーバーが受信するクライアントの主体名は、rpc_gss_principal_t 構造の形式 (使用するメカニズムによって決定される、不透明で長さを暗示したバイト列) をとります。

サーバー主体名の設定

サーバーは、起動時に、そのサーバーを表わす主体名を指定する必要があります。1 つのサーバーが複数の主体として機能する場合もあります。サーバー主体名の設定には、rpc_gss_set_svc_name() を使用します。

```
char *principal, *mechanism;
u_int req_time;

principal = "nfs@engineering.company.com";
mechanism = "kerberosv5";
req_time = 10000;  /* 資格の有効時間 */

rpc_gss_set_svc_name(principal, mechanism, req_time, SERV_PROG, SERV_VERS);
```

クライアント主体名の作成

サーバーは、クライアントの主体名で稼動できなければなりません。たとえば、クライアントの主体名をアクセス制御リストと比較するため、またはクライアントの UNIX 資格を検出するため (このような資格が存在する場合) に必要です。サーバーが、受信した主体名を既知のエンティティの名前と比較する必要がある場合、サーバーは、既知のエンティティ用に `rpc_gss_principal_t` 主体名を生成できなければなりません。 `rpc_gss_get_principal_name()` 呼び出しでは、ネットワーク上で個人を識別するパラメータをいくつか入力し、`rpc_gss_principal_t` 構造ポインタの形式で主体名を設定します。

```
rpc_gss_principal_t *principal;

rpc_gss_get_principal_name(principal, mechanism, name, node, domain);
. . .
```

最初に設定される引数は、`rpc_gss_principal_t` です。次の引数は、使用するセキュリティメカニズムです。生成される主体名はメカニズムに依存します。最後の 3 つの引数は、ネットワーク外の個人を識別するために使用されるパラメータです。「name」には、`joeh` または `nfs` などの、個人名またはサービス名が入ります。「node」には UNIX マシン名、「domain」には DNS、NIS、または NIS+ ドメイン名が入ります。各セキュリティメカニズムには、別々の識別パラメータが必要です。たとえば、Kerberos v5 には、ユーザー名を必ず指定します。それに加えて修飾されたノード名とドメイン名を指定することができます。 (Kerberos 用語では、ホスト名と領域名)。

主体名の解放

主体名は、free() ライブラリコールを使用して解放します。

サーバーで資格を受信する

サーバーは、クライアントの資格を獲得できなければなりません。rpc_gss_getcred() 関数を使用すると、サーバーは UNIX 資格または RPCSEC_GSS 資格のいずれか (またはこの両方) を検索できます。これは、この関数が正常に終了した場合に設定された 2 つの引数によって実行されます。最初の引数は、呼び出し元の UNIX 資格 (uid とgid) が組み込まれた rpc_gss_ucred_t 構造 (存在する場合) へのポインタになります。2 番めの引数は rpc_gss_raw_cred_t 構造へのポインタです。

```
typedef struct {
  u_int version;               /* RPCSEC_GSS プログラムバージョン */
  char         *mechanism;
  char         *qop;
  rpc_gss_principal_t  *client_principal;  /* クライアント主体名 */
  char         *svc_principal;      /* サーバー主体名 */
  rpc_gss_service_t   service;        /* プライバシ、完全性 enum */
} rpc_gss_rawcred_t;
```

(ここでは、上記のように rpc_gss_gss_principal_t 構造が表示されます。) rpc_gss_rawcred_t にはクライアントとサーバーの両方の主体名が組み込まれているため、rpc_gss_getcred() は両方の名前を戻します。

コード例 4–29 は 1 つのサーバー側のディスパッチプロシージャの例です。これにより、サーバーは呼び出し元の資格を入手します。このプロシージャでは、呼び出し元の UNIX 資格を入手してから、次に rpc_gss_rcred_t 引数内で検出された、メカニズム、QOP、サービスタイプを使用してユーザーの識別情報 (ID) を確認します。

コード例 4–29 資格の入手

```
static void server_prog(struct svc_req *rqstp, SVCXPRT *xprt)
{
  rpc_gss_ucred_t *ucred;
  rpc_gss_rawcred_t *rcred;

  if (rqst->rq_proq == NULLPROC) {
```

```
    svc_sendreply(xprt, xdr_void, NULL);
    return;
  }
  /*
   * 他の全ての要求を認証する */
   */

  switch (rqstp->rq_cred.oa_flavor) {
  case RPCSEC_GSS:
   /*
    * 資格情報を取得する
    */
   rpc_gss_getcred(rqstp, &rcred, &ucred, NULL);
   /*
   * 認証ファイルを参照してセキュリティパラメータを
   * 使用することでユーザーにアクセスが許可されている
   * ことを確認する
   */
   if (!authenticate_user(ucred->uid, rcred->mechanism,
    rcred->qop, rcred->service)) {
    svcerr_weakauth(xprt);
    return;
   }
   break;  /* ユーザーに許可する */
  default:
   svcerr_weakauth(xprt);
   return;
  } /* スイッチの終り */

  switch (rqstp->rq_proq) {
  case SERV_PROC1:
   . . .
  }

  /* 通常の要求処理 ; 応答を送る ... */

  return;

}
```

Cookies

コード例 4–29 では、 `rpc_gss_getcred()`への最後の引数は、ユーザー定義の cookie です。このコンテキストの作成時にサーバーによってどのような値が指定されていても、このユーザー定義の値が戻されます。この cookie は 4 バイトの値で、そのアプリケーションに適したあらゆる方法で使用されます。RPC はこれを解釈しません。たとえば、 cookie は、コンテキストの起動元を示す構造へのポインタまたはインデックスになることができます。また、各要求ごとにこの値を計算する代わ

りに、サーバーがコンテキスト作成時にこの値を計算します。このため、要求の処理時間が削減されます。

コールバック

これ以外に cookie が使用される場所は、コールバックです。サーバーは、rpc_gss_set_callback() 関数を使用することにより、(ユーザー定義の) コールバックを指定してコンテキストが最初に使用された時を認知できます。コールバックは、コンテキストが指定されたプログラムとバージョン用に確立されたあとに、そのコンテキストがデータ交換に最初に使用された時に呼び出されます。

ユーザー定義のコールバックルーチンは、以下のような形式になります。

bool_t **callback**(struct svc_req **req*, gss_cred_id_t *deleg*, gss_ctx_id_t *gss_context*, rpc_gss_lock_t **lock*, void ***cookie*);

2 番めと 3 番めの引数は、GSS-API データタイプで、現在はまだ公開されていません。そのため、コールバック関数はこれらを無視します。簡単に説明すると、プログラムが GSS-API オペレーションをこのコンテキスト上で実行する必要がある場合、すなわち受信条件のテストをする場合、*deleg* は代表されるピアの識別情報になり、一方 *gss_context* は GSS-API コンテキストへのポインタになります。*cookie* 引数については、すでに説明しました。

lock 引数は、以下のように rpc_gss_lock_t 構造へのポインタです。

```
typedef struct {
  bool_t       locked;
  rpc_gss_rawcred_t  *raw_cred;
} rpc_gss_lock_t;
```

このパラメータを使用すると、サーバーはセッションに対し強制的に特定の QOP とサービスを実行できます。QOP とサービスは、rpc_gss_rawcred_t 構造内で検出できます。サーバーはこれらの値は変更できません。ユーザー定義のコールバックが呼び出されると、*locked* パラメータは FALSE に設定されます。サーバーは、コンテキストをロックするために、これを TRUE に設定する必要があります。コンテキストがロックされると、QOP とサービスの値と一致する要求だけが受理されます。

最大データサイズ

rpc_gss_max_data_length() と rpc_gss_svc_max_data_length() の 2 つの関数は、1 つのデータが、セキュリティ測度によって変換され「ワイヤを通じて」送信される前に、そのデータの大きさを判別する場合に便利です。つまり、暗号化などのセキュリティ変換により、通常、伝送される 1 つのデータのサイズは変更されます (通常は、大きくなる)。データが使用できるサイズ以上に大きくならないように、これら 2 つの関数 (前者はクライアント側バージョンで、後者はサーバー側バージョン) により、指定されたトランスポートの変換前の最大サイズが戻されます。

その他の関数

関数の中には、導入されたセキュリティシステムに関する情報を入手する場合に使用できるものもあります。rpc_gss_get_mechanisms() は、導入されたセキュリティメカニズムのリストを戻します。一方、rpc_gss_is_installed() は、指定したメカニズムがインストールされているかどうかを検査します。rpc_gss_get_mech_info() は、指定されたメカニズムの有効な QOP を戻します。これらの関数を使用することによって、プログラマは、アプリケーション内のセキュリティパラメータのハードコード化を避けることができます。
RPCSEC_GSS 関数については、表 4–8 と rpcsec_gss(3N) マニュアルページを参照してください。

関連ファイル

gsscred テーブル

サーバーが要求に関連するクライアントの資格を検索すると、サーバーはクライアントの主体名 (rpc_gss_principal_t 構造ポインタの形式)、またはクライアントのローカル UNIX 資格 (UID) のいずれかを入手できます。NFS 要求などのサービスでは、アクセス検査に必要なローカル UNIX 資格が必要ですが、他の資格は必要ありません。つまり、これらのサービスでは、たとえば主体名は、rpc_gss_principal_t 構造として直接、独自のアクセス制御リスト内に格納できるからです。

注 - クライアントのネットワーク資格 (その主体名) とローカル UNIX 資格間の対応は自動的に行われません。これは、ローカルのセキュリティ管理者が明示的に設定する必要があります。

---

gsscred ファイルには、クライアントの UNIX 資格とネットワーク資格の両方が入っています。後者は、rpc_gss_principal_t 構造の Hex-ASCII 表示です。これには、XFN を通じてアクセスするため、このテーブルは、ファイル、NIS、NIS+、あるいは XFN によってサポートされる将来のネームサービス上に導入できます。XFN 階層では、このテーブルは *this_org_unit*/*service*/gsscred として表示されます。 gsscred テーブルは、gsscred ユーティリティとともに保持されます。このユーティリティを使用すると、管理者はユーザーやメカニズムの追加および削除が行えます。

/etc/gss/qop と /etc/gss/mech

便宜上、RPCSEC_GSS では、メカニズムと保護の質 (QOP) パラメータを表示するためにリテラルの文字列を使用します。ただし、基本的なメカニズム自体では、メカニズムをオブジェクト識別子として、QOP は 32 ビット整数として表示する必要があります。また、各メカニズムごとに、そのメカニズムのサービスを実現する共有ライブラリを指定する必要があります。

/etc/gss/mech ファイルには、システム上に導入されたすべてのメカニズムに関する情報、メカニズム名 (ASCII 形式)、メカニズムの ODI、サービスを実現する共有ライブラリとカーネルモジュールが格納されます。次に例を示します。

```
kerberosv5   1.2.840.113554.1.2.2   mech_krb5.so   kmech_krb5
```

/etc/gss/qop ファイルには、導入されたすべてのメカニズム用に、各メカニズムがサポートするすべての QOP が、ASCII 文字列とそれに対応する 32 ビット整数の両方で格納されます。

/etc/gss/mech と /etc/gss/qop は、両方とも指定されたシステムにセキュリティメカニズムが最初に導入されたときに作成されます。

カーネル内 RPC ルーチンは、通常、文字列にない値を使用してメカニズムと QOP を表すため、アプリケーションは、これらのカーネル内ルーチンを利用したい場合には、rpc_gss_mech_to_oid() と rpc_gss_qop_to_num() 関数を使用してこれらのパラメータと同等の文字列にない値を入手します。

ポートモニタの使用

RPC サーバーは、inetd や listen のようなポートモニタから起動できます。ポートモニタは、要求が来ているかどうか監視し、要求が来ればそれに応じてサーバー

を生成します。生成されたサーバープロセスには、要求を受信したファイル記述子 0 が渡されます。inetd の場合、サーバーは処理を終えるとすぐに終了するか、次の要求がくる場合に備えて指定された時間だけ待ってから終了します。

listen の場合は常に新たなプロセスが生成されるため、サーバーは応答を返したらすぐに終了しなければなりません。次に示す関数呼び出しでは、ポートモニタから起動されるサービスで使用する SVCXPRT ハンドルが作成されます。

```
transp = svc_tli_create(0, nconf, (struct t_bind *)NULL, 0, 0)
```

ここで、nconf は要求を受信したトランスポートの netconfig 構造体です。

サービスはポートモニタによりすでに rpcbind で登録されているので、登録する必要はありません。ただし、サービス手続きは次のように svc_reg() を呼び出して登録しなければなりません。

```
svc_reg(transp, PROGNUM, VERSNUM, dispatch,(struct netconfig *)NULL)
```

ここでは netconfig 構造体として NULL を渡し、svc_reg() が rpcbind を呼び出してサービスを登録しないようにしています。

注 - rpcgen が生成したサーバー側スタブプログラムを調べて、これらのルーチンの呼び出し順序を確認してください。

接続型トランスポートの場合は、次のルーチンにより下位レベルインタフェースが提供されます。

```
transp = svc_fd_create(0, recvsize, sendsize);
```

最初の引数ではファイル記述子 0 を指定します。recvsize と sendsize には、適当なバッファサイズを指定できます。どちらの引数も 0 とすると、システムのデフォルト値が使用されます。自分で監視を行わないアプリケーションサーバーの場合は、svc_fd_create() を使用します。

inetd の使用

/etc/inet/inetd.conf のエントリ形式は、ソケットサービス、TLI サービス、RPC サービスによってそれぞれ異なります。RPC サービスの場合の inetd.conf のエントリ形式は次のようになります。

```
rpc_prog/vers endpoint_type rpc/proto flags user pathname args
```

各エントリの内容を次に示します。

表 4–9 RPC `inetd` サービス

rpc_prog/vers	RPC プログラム名に / とバージョン番号 (またはバージョン番号の範囲) を付けたもの
endpoint_type	`dgram` (非接続型ソケット)、`stream` (接続型ソケット)、`tli` (TLI 端点) のどれか
proto	サポートされているトランスポートすべてを意味する *、nettype、netid のどれか。または、nettype と netid をコンマで区切ったリスト
flags	`wait` または `nowait` のどちらか
user	有効な `passwd` データベースに存在しているユーザー
pathname	サーバーデーモンへのフルパス名
args	デーモンの呼び出し時に渡される引数

エントリの例を次に示します。

```
rquotad/1 tli rpc/udp wait root /usr/lib/nfs/rquotad rquotad
```

`inetd` についての詳細は、`inetd.conf(4)` のマニュアルページを参照してください。

リスナの使用

次に示すように `pmadm` を使用して RPC サービスを追加します。

```
pmadm -a -p pm_tag -s  svctag -i id -v vers  \
 -m 'nlsadmin -c command -D -R prog:vers'
```

引数の –a はサービスの追加を意味します。–p *pm_tag* ではサービスへのアクセスを提供するポートモニタに結合したタグを指定します。–s *svctag* はサーバーの ID コードです。–i *id* はサービス *svctag* に割り当てられた /etc/passwd 内のユーザー名です。–v *vers* はポートモニタのデータベースファイルのバージョン番号です。–m ではサービスを呼び出す nlsadmin コマンドを指定します。nlsadmin コマンドには引数を渡すことができます。たとえば、rusersd という名前の遠隔プログラムサーバーのバージョン 1 を追加する場合は、pmadm コマンドは次のようになります。

```
# pmadm -a -p tcp -s rusers -i root -v 4 \
   -m 'nlsadmin -c /usr/sbin/rpc.ruserd -D -R 100002:1'
```

このコマンドでは、root パーミッションが指定され、listener データベースファイルのバージョン 4 でインストールされ、TCP トランスポート上で使用可能になります。pmadm の引数やオプションは複雑であるため、RPC サービスはコマンドスクリプトでもメニューシステムでも追加できます。メニューシステムを使用するには、sysadm ports と入力して、port_services オプションを選択します。

サービスを追加した場合は、その後リスナを再初期化してサービスを利用可能にしなければなりません。そのためには、次のようにリスナを一度止めてから再起動します (このとき rpcbind が実行されていなければならないことに注意してください)。

```
# sacadm -k -p pmtag
# sacadm -s -p pmtag
```

リスナプロセスの設定などについての詳細は、listen(1M)、pmadm(1M)、sacadm(1M)、sysadm(1M) のマニュアルページと『*TCP/IP* とデータ通信』を参照してください。

サーバーのバージョン

一般に、プログラム PROG の最初のバージョンは PROGVERS_ORIG とし、最新バージョンは PROGVERS と命名します。プログラムのバージョン番号は続き番号で割り

当てなければなりません。バージョン番号に飛ばされた番号があると、検索したときに定義済みのバージョン番号を探し出せないようなことが起こります。

プログラムのバージョン番号は、プログラムの所有者以外は決して変更しないでください。自分が所有していないプログラムのバージョン番号を追加したりすると、そのプログラムの所有者がバージョン番号をインクリメントするときに重大な問題が起こります。バージョン番号の登録やご質問はご購入先へお問い合わせ下さい。

ruser プログラムの新バージョンが、int ではなく unsigned short を返すように変更されたとします。新バージョンの名前を RUSERSVERS_SHORT とすると、新旧の 2 つのバージョンをサポートするサーバーは二重登録することになります。次のように、どちらの登録でも同じサーバーハンドルを使用します。

コード例 4–30 同一ルーチンの 2 つのバージョンのためのサーバーハンドル

```
if (!svc_reg(transp, RUSERSPROG, RUSERSVERS_ORIG,
      nuser, nconf))
{
 fprintf(stderr, "can't register RUSER service\n");
 exit(1);
}
if (!svc_reg(transp, RUSERSPROG, RUSERSVERS_SHORT, nuser,
      nconf)) {
 fprintf(stderr, "can't register RUSER service\n");
 exit(1);
}
```

次のように、1 つの手続きで両バージョンを実行できます。

コード例 4–31 両バージョンを使用するサーバー

```
void
nuser(rqstp, transp)
 struct svc_req *rqstp;
 SVCXPRT *transp;
{
 unsigned int nusers;
 unsigned short nusers2;
 switch(rqstp->rq_proc) {
  case NULLPROC:
   if (!svc_sendreply( transp, xdr_void, 0))
    fprintf(stderr, "can't reply to RPC call\n");
   return;
  case RUSERSPROC_NUM:
   /*
```

(続く)

```
    * ユーザー数を求めて変数 nusers に設定するコード
    */
  switch(rqstp->rq_vers) {
   case RUSERSVERS_ORIG:
    if (! svc_sendreply( transp, xdr_u_int, &nusers))
     fprintf(stderr, "can't reply to RPC call\n");
    break;
   case RUSERSVERS_SHORT:
    nusers2 = nusers;
    if (! svc_sendreply( transp, xdr_u_short, &nusers2))
     fprintf(stderr, "can't reply to RPC call\n");
    break;
  }
  default:
   svcerr_noproc(transp);
   return;
 }
 return;
}
```

クライアントのバージョン

異なるホストでは RPC サーバーの異なるバージョンが実行されている可能性があるので、クライアントはさまざまなバージョンに対応できるようにしなければなりません。たとえば、あるサーバーでは旧バージョン RUSERSPROG(RUSERSVERS_ORIG) が実行されており、別のサーバーでは最新バージョン RUSERSPROG(RUSERSVERS_SHORT) が実行されているとします。

サーバーのバージョンがクライアント作成ルーチン clnt_call() で指定したバージョン番号と一致しない場合は、clnt_call() から RPCPROGVERSMISMATCH というエラーが返されます。サーバーがサポートしているバージョン番号を取り出して、正しいバージョン番号をもつクライアントハンドルを作成することもできます。そのためには、コード例 4–32 のルーチンを使用するか、clnt_create_vers() を使用します。詳細については、rpc(3N) のマニュアルページを参照してください。

コード例 4–32 クライアント側での RPC バージョン選択

```
main()
{
 enum clnt_stat status;
 u_short num_s;
 u_int num_l;
 struct rpc_err rpcerr;
```

```
int maxvers, minvers;
CLIENT *clnt;

clnt = clnt_create("remote", RUSERSPROG, RUSERSVERS_SHORT,
                "datagram_v");
if (clnt == (CLIENT *) NULL) {
 clnt_pcreateerror("unable to create client handle");
 exit(1);
}
to.tv_sec = 10;       /* タイムアウト値を設定 */
to.tv_usec = 0;

status = clnt_call(clnt, RUSERSPROC_NUM, xdr_void,
                  (caddr_t) NULL, xdr_u_short,
                 (caddr_t)&num_s, to);
if (status == RPC_SUCCESS) {   /* 最新バージョン番号が見つかった場合 */
 printf("num = %d\n", num_s);
 exit(0);
}
if (status != RPC_PROGVERSMISMATCH) {  /* その他のエラー */
 clnt_perror(clnt, "rusers");
 exit(1);
}
/* 指定したバージョンがサポートされていない場合 */
clnt_geterr(clnt, &rpcerr);
maxvers = rpcerr.re_vers.high;  /* サポートされている最新バージョン */
minvers = rpcerr.re_vers.low;  /* サポートされている最も古いバージョン */
if (RUSERSVERS_SHORT < minvers || RUSERSVERS_SHORT > maxvers)
{
                          /* サポート範囲内にない場合 */
 clnt_perror(clnt, "version mismatch");
 exit(1);
}
(void) clnt_control(clnt, CLSET_VERSION, RUSERSVERS_ORIG);
status = clnt_call(clnt, RUSERSPROC_NUM, xdr_void,
   (caddr_t) NULL, xdr_u_int, (caddr_t)&num_l, to);
if (status == RPC_SUCCESS)
                 /* 識別できるバージョン番号が見つかった場合 */
 printf("num = %d\n", num_l);
else {
 clnt_perror(clnt, "rusers");
 exit(1);
}
}
```

一時的な RPC プログラム番号の使用

場合によっては、動的に生成される RPC プログラム番号をアプリケーションが使用すると便利なことがあります。たとえば、コールバック手続きを実装する場合などで

す。コールバックでは、クライアントプログラムは通常、動的に生成される、つまり一時的な RPC プログラム番号を使用して RPC サービスを登録し、これを要求とともにサーバーに渡します。次にサーバーは一時的な RPC プログラム番号を使用してクライアントプログラムをコールバックし、結果を返します。クライアントの要求を処理するのにかなりの時間がかかり、クライアントが停止できない (シングルスレッドであると仮定して) 場合などには、この機構が必要になります。このような場合、サーバーはクライアントの要求を認識し、あとで結果とともにコールバックを行います。コールバックを使用する別の例としては、サーバーから定期的なレポートを生成する場合があります。クライアントは RPC 呼び出しを行い、報告を開始します。そしてサーバーはクライアントプログラムが提供する一時的な RPC プログラム番号を使用して、定期的にレポートとともにクライアントをコールバックします。

動的に生成される一時的な RPC 番号は、0x40000000 から 0x5fffffff の範囲です。次に示すルーチンは指定されるトランスポートタイプ用に、一時的な RPC プログラムに基づいてサービスを作成します。サービスハンドルと一時的な rpc プログラム番号が返されます。呼び出し側はサービスディスパッチルーチン、バージョンタイプ、トランスポートタイプを提供します。

コード例 4–33 一時的な RPC プログラム - サーバー側

```
SVCXPRT *
register_transient_prog(dispatch, program, version, netid)
 void (*dispatch)(); /* サービスディスパッチルーチン */
 rpcproc_t  *program;    /* 一時的な RPC 番号が返される */
 rpcvers_t version;     /* プログラムバージョン */
 char *netid;        /* トランスポート id */
{
 SVCXPRT  *transp;
 struct netconfig *nconf;
 rpcprog_t prognum;
 if ((nconf = getnetconfigent(netid)) == (struct netconfig
*)NULL)
  return ((SVCXPRT *)NULL);
 if ((transp = svc_tli_create(RPC_ANYFD, nconf,
    (struct t_bind *)NULL, 0, 0)) == (SVCXPRT *)NULL) {
  freenetconfigent(nconf);
  return ((SVCXPRT *)NULL);
 }
 prognum = 0x40000000;
 while (prognum < 0x60000000 && svc_reg(transp, prognum,
version,
         dispatch, nconf) == 0) {
  prognum++;
 }
 freenetconfigent(nconf);
 if (prognum >= 0x60000000) {
  svc_destroy(transp);
```

(続く)

```
  return ((SVCXPRT *)NULL);
 }
 *program = prognum;
 return (transp);
}
```

マルチスレッド RPC プログラミング

このマニュアルには、Solaris でのマルチスレッドプログラミングについては説明していません。次の項目については、『マルチスレッドのプログラミング』を参照してください。

- スレッドの作成
- スケジューリング
- 同期
- シグナル
- プロセスリソース
- 軽量プロセス (lwp)
- 並列性
- データロックの技術

TI-RPC は、Solaris 2.4 以降のマルチスレッド RPC サーバーをサポートします。マルチスレッドサーバーと シングルスレッドのサーバーの違いは、マルチスレッドサーバーがスレッドの技術を使用して複数のクライアント要求を同時に処理することです。マルチスレッドサーバーの方が、高度なパフォーマンスと可用性を備えています。

このリリースで新規に使用可能なインタフェースについては、まず、142ページの「マルチスレッドサーバーの概要」から読んでください。

マルチスレッドクライアントの概要

マルチスレッド対応のクライアントプログラムでは、RPC 要求が出されるたびにスレッドを 1 つ作成することができます。複数スレッドが同一のクライアントハンドルを共有する場合は、RPC 要求発行できるのは一度に 1 つのスレッドだけです。その他のすべてのスレッドは、未処理の要求が終了するまで待たなければなりません。これに対して、複数スレッドがそれぞれ固有のクライアントハンドルを使用して RPC 要求を出す場合には、複数の要求が同時に処理されます。図 4–2 は、異なるクライアントハンドルを使用するクライアント側の 2 つのスレッドから成るマルチスレッド対応クライアント環境でのタイミングの例を示したものです。

コード例 4–34 は、クライアント側でマルチスレッド `rstat` プログラムを実行する場合を示します。クライアントプログラムは各ホストに対してスレッドを作成します。スレッドはそれぞれ、固有のクライアントハンドルを作成し、指定のホストにさまざまな RPC 呼び出しを行なっています。クライアント側の各スレッドは異なるハンドルを使用して RPC 呼び出しを行うため、RPC 呼び出しは同時に実行されます。

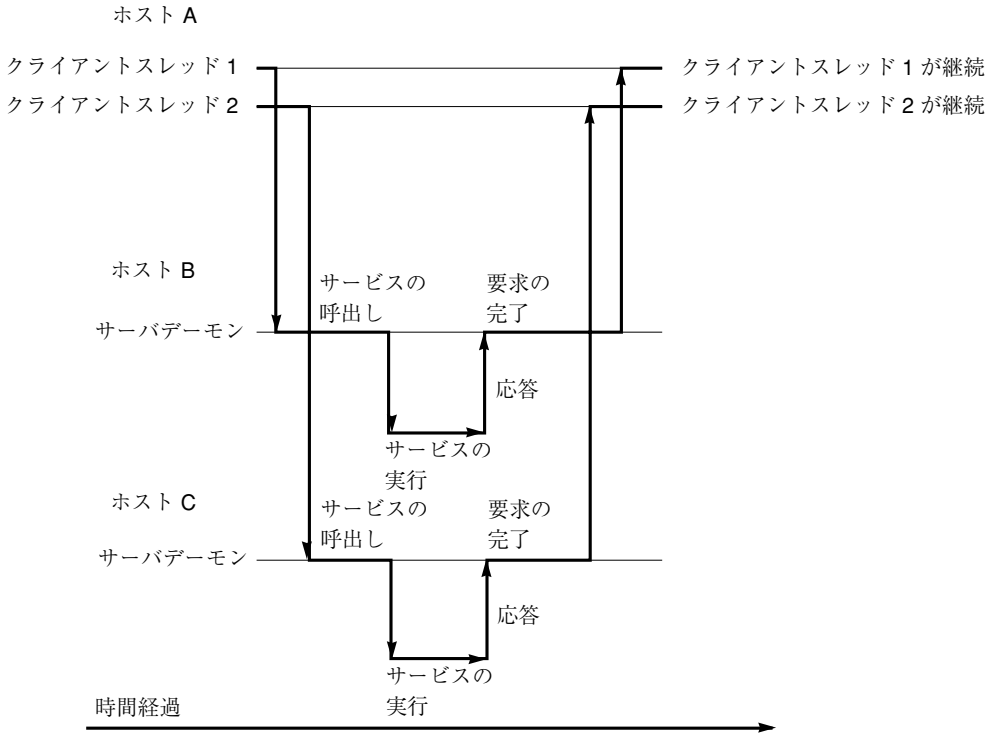


図 4–2　異なるクライアントハンドルを使用する 2 つのクライアントスレッド (リアルタイム)

注 - RPC マルチスレッド対応アプリケーションを作成する場合は常に、スレッドライブラリをリンクしなければなりません。コンパイルコマンドで `-lthread` を指定して、スレッドライブラリを最後にリンクするようにしなければなりません。

次のように入力して コード例 4–34 のプログラムを作成します。

$

```
cc rstat.c -lnsl -lthread
```

コード例 **4–34**　マルチスレッド `rstat` に対するクライアント

```
/* @(#)rstat.c 2.3 93/11/30 4.0 RPCSRC */
/*
 * w コマンドと同様の形式で、遠隔ホストのステータスを表示する簡単な
```

```
 * プログラム
 */

#include <thread.h>  /* スレッドインタフェースの定義 */
#include <synch.h> /* 相互排他的ロックの定義 */
#include <stdio.h>
#include <sys/param.h>
#include <rpc/rpc.h>
#include <rpcsvc/rstat.h>
#include <errno.h>

mutex_t tty;     /* printf のための tty の制御 */
cond_t cv_finish;
int count = 0;

main(argc, argv)
int argc;
char **argv;
{
 int i;
 thread_t tid;
 void *do_rstat();

 if (argc < 2) {
  fprintf(stderr, ''usage: %s \''host\'' [...]\n'', argv[0]);
  exit(1);
 }

 mutex_lock(&tty);

 for (i = 1; i < argc; i++) {
  if (thr_create(NULL, 0, do_rstat, argv[i], 0, &tid) < 0) {
   fprintf(stderr, ''thr_create failed: %d\n'', i);
   exit(1);
  } else
   fprintf(stderr, ''tid: %d\n'', tid);
 }

 while (count < argc-1) {
  printf(''argc = %d, count = %d\n'', argc-1, count);
  cond_wait(&cv_finish, &tty);

 }

 exit(0);
}

bool_t rstatproc_stats();

void *
do_rstat(host)
char *host;
{
```

(続く)

```
CLIENT *rstat_clnt;
statstime host_stat;
bool_t rval;
struct tm *tmp_time;
struct tm host_time;
struct tm host_uptime;
char days_buf[16];
char hours_buf[16];

mutex_lock(&tty);
printf(``%s: starting\n'', host);
mutex_unlock(&tty);

/* rstat クライアントハンドル */
rstat_clnt = clnt_create(host, RSTATPROG, RSTATVERS_TIME,
      ``udp'');
if (rstat_clnt == NULL) {
 mutex_lock(&tty);    /* ty の制御権を取得 */
 clnt_pcreateerror(host);
 count++;
 cond_signal(&cv_finish);
 mutex_unlock(&tty); /* tty の制御権を解放 */

 thr_exit(0);

}

rval = rstatproc_stats(NULL, &host_stat, rstat_clnt);
if (!rval) {
 mutex_lock(&tty); /* tty の制御権を取得 */
 clnt_perror(rstat_clnt, host);
 count++;
 cond_signal(&cv_finish);
 mutex_unlock(&tty); /* tty の制御権を解放 */

 thr_exit(0);

}

tmp_time = localtime_r(&host_stat.curtime.tv_sec,
                       &host_time);

host_stat.curtime.tv_sec = host_stat.boottime.tv_sec;

tmp_time = gmtime_r(&host_stat.curtime.tv_sec,
                    &host_uptime);

if (host_uptime.tm_yday != 0)
 sprintf(days_buf, ``%d day%s, ``, host_uptime.tm_yday,
          (host_uptime.tm_yday > 1) ? ``s'' : ````);
else
 days_buf[0] = '\0';
```

(続く)

```
 if (host_uptime.tm_hour != 0)
  sprintf(hours_buf, ``%2d:%02d,'',
   host_uptime.tm_hour, host_uptime.tm_min);

 else if (host_uptime.tm_min != 0)
  sprintf(hours_buf, ``%2d mins,'', host_uptime.tm_min);
 else

  hours_buf[0] = '\0';

 mutex_lock(&tty); /* tty の制御権を取得 */
 printf(``%s: ``, host);
 printf(`` %2d:%02d%cm up %s%s load average: %.2f %.2f %.2f\n'',
  (host_time.tm_hour > 12)  ? host_time.tm_hour - 12

  : host_time.tm_hour,
  host_time.tm_min,
  (host_time.tm_hour >= 12) ? 'p'
  : 'a',
  days_buf,
  hours_buf,
  (double)host_stat.avenrun[0]/FSCALE,
  (double)host_stat.avenrun[1]/FSCALE,
  (double)host_stat.avenrun[2]/FSCALE);
 count++;
 cond_signal(&cv_finish);
 mutex_unlock(&tty); /* tty の制御権を解放 */
 clnt_destroy(rstat_clnt);

 sleep(10);
 thr_exit(0);
}

/* クライアント側の MT rstat プログラムの実行 */
/* clnt_control() を使用してデフォルトのタイムアウトを変更可能 */
static struct timeval TIMEOUT = { 25, 0 };

bool_t
rstatproc_stats(argp, clnt_resp, clnt)
 void *argp;
 statstime *clnt_resp;
 CLIENT *clnt;
{

 memset((char *)clnt_resp, 0, sizeof (statstime));
 if (clnt_call(clnt, RSTATPROC_STATS,
  (xdrproc_t) xdr_void, (caddr_t) argp,
  (xdrproc_t) xdr_statstime, (caddr_t) clnt_resp,
  TIMEOUT) != RPC_SUCCESS) {
  return (FALSE);

 }
```

(続く)

```
 return (TRUE);
}
```

マルチスレッドサーバーの概要

Solaris 2.4 より前のバージョンでは、RPC サーバーはシングルスレッドでした。つまり、クライアント側から要求が来るごとに処理していました。たとえば、2 つの要求を同時に受け取り、最初の処理に 30 秒、次の処理に 1 秒かかるとすると、2 つめの要求を出したクライアントは最初の処理が完了するまで待たなければなりません。これは、各 CPU が異なる要求を同時に処理するマルチプロセッササーバー環境を利用できず、他の要求がサーバーによって処理することができるのに 1 つの要求の I/O の完了を待っている状態が生じ、望ましいものではありません。

Solaris 2.4 以降 の RPC ライブラリでは、サービス開発者がエンドユーザーにより良いパフォーマンスを提供するマルチスレッドサーバーを作成できる機能を追加しました。サーバーのマルチスレッドの 2 つのモード、自動マルチスレッドモードとユーザー・マルチスレッド・モードは、TI-RPC でサポートされます。

自動モードでは、サーバーは、クライアント要求を受信するごとに新規スレッドを自動的に作成します。このスレッドは要求を処理し、応答してから終了します。ユーザーモードでは、サービス開発者が、入って来るクライアント要求を同時に処理するスレッドを作成、管理します。自動モードはユーザーモードより使用はしやすいのですが、ユーザーモードの方が特別な要件を必要とするサービス開発者に対して柔軟性があります。

注 - RPC マルチスレッド対応アプリケーションを作成する場合は常に、スレッドライブラリをリンクしなければなりません。コンパイルコマンドで `-lthread` を指定して、スレッドライブラリを最後にリンクするようにしなければなりません。

サーバー側のマルチスレッドをサポートする呼び出しでは、`rpc_control()` と `svc_done()` がサポートされています。これらの呼び出しによってサーバー側でマルチスレッド処理が行えるようになりました。`rpc_control()` 呼び出しがマルチスレッドモードを設定するために、自動モードとユーザーモードの両方で使用されます。サーバーが自動モードを使用する場合には、`svc_done()` を呼び出す必要はあ

りません。ユーザーモードの場合には、サーバーが要求処理からのリソースを再要求できるようにするため、svc_done() は各クライアント要求が処理されてから呼び出されなければなりません。さらにマルチスレッド RPC サーバーは、svc_run() をマルチスレッド対応で呼び出さなければなりません。svc_getreqpoll() と svc_getreqset() は、MT アプリケーションでは安全ではありません。

注 - サーバープログラムが新規インタフェース呼び出しを行わない場合には、デフォルトのモードのシングルスレッドモードのままです。

サーバーが使用しているモードに関係なく、RPC サーバー手続きはマルチスレッド対応にしなければなりません。通常これは、すべての静的変数とグロール変数が mutex ロックで保護される必要がある、ということです。相互排他と他の同期 API は、synch.h で定義されます。さまざまな同期インタフェースのリストは、condition(3T)、rwlock(3T)、mutex(3T) を参照してください。

図 4–3 は、マルチスレッドモードのどちらかで実行されるサーバーの実行タイミングを示します。

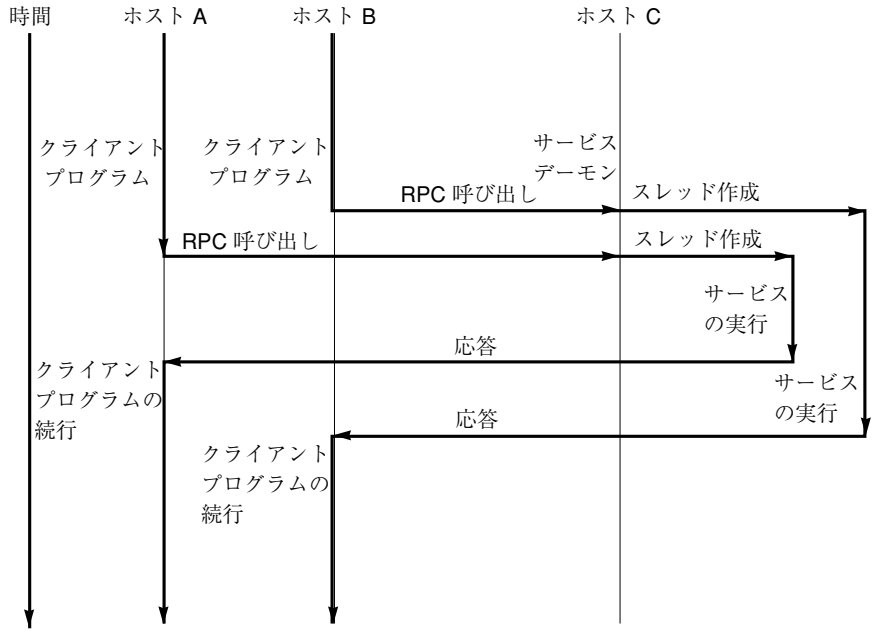


図 4–3　マルチスレッド RPC サーバーのタイミング図

サービス・トランスポート・ハンドルの共有

サービス・トランスポート・ハンドル、SVCXPRT には、引き数を復号化するための領域と結果をコード化するための領域である 1 つのデータ領域があります。したがって、デフォルトでは、シングルスレッドモードであり、この構造は、これらの操作を行う関数を呼び出すスレッド間では自由に共有することはできません。ただし、サーバーが、マルチスレッド自動モードまたはユーザーモードにある場合には、この構造のコピーは、同時要求処理を可能にするために、サービスディスパッチ用のプログラムに引き渡されます。これらの状況では、ルーチンのマルチスレッド対応ではない一部のルーチンがマルチスレッド対応となります。特別に注意書きがない場合には、サーバーインタフェースは通常、マルチスレッド対応です。サーバー側のインタフェースについての詳細は、rpc_svc_calls(3N) マニュアルページを参照してください。

マルチスレッド自動モード

マルチスレッド自動モードでは、RPC ライブラリはスレッドを作成し、管理することができます。サービス開発者が新規インタフェース呼び出し、rpc_control() を呼び出し、svc_run() を呼び出す前にサーバーをマルチスレッド自動モードにします。このモードでは、プログラマはサービスプロシージャがマルチスレッド対応であることを確認するだけで十分です。

rpc_control() の使用によって、アプリケーションでグローバル RPC 属性を設定できます。現在はサービス側の操作しかサポートしていません。 表 4–10 は、自動モード用に定義された rpc_control() 操作を示します。追加の情報については、rpc control(3N) マニュアルページを参照してください。

表 4–10 rpc_control() ライブラリルーチン

RPC_SVC_MTMODE_SET()	マルチスレッドモードの設定
RPC_SVC_MTMODE_GET()	マルチスレッドの取得
RPC_SVC_THRMAX_SET()	最大スレッド数の設定
RPC_SVC_THRMAX_GET()	最大スレッド数の取得
RPC_SVC_THRTOTAL_GET ()	現在アクティブなスレッドの合計数

表 4–10　rpc_control() ライブラリルーチン　続く

RPC_SVC_THRCREATES_GET()	RPC ライブラリ作成のスレッドの累積数
RPC_SVC_THRERRORS_GET()	RPC ライブラリ内の thr_create エラー数

注 - 表 4–10 の get 演算は、RPC_SVC_MTMODE_GET() 以外はすべて、自動マルチスレッドモードにだけ適用されます。マルチスレッド・ユーザー・モードまたはデフォルトのシングル・スレッド・モードで使用する場合には、演算の結果は定義されません。

デフォルトでは、RPC ライブラリが一度に作成できるスレッドの最大数は 16 です。サーバーが 16 以上のクライアント要求を同時に処理する必要がある場合には、スレッドの最大数を指定して設定する必要があります。このパラメータは、サーバーによっていつでも設定することができ、これによってサーバー開発者はサーバーによって使用されるスレッドリソースの上限を設定できます。コード例 4–35 は、マルチスレッド自動モードに作成された RPC プログラムの例です。この例では、スレッドの最大数は 20 に設定されています。

マルチスレッドのパフォーマンスは、関数 svc_getargs() が、NULLPROCS 以外のプロシージャによって呼び出されるごとに、引き数 (この場合には xdr_void()) がない場合でも改善されていきます。これはマルチスレッド自動モードとマルチスレッドユーザーモード両方の場合においてです。詳細は、rpc_svc_calls(3N) マニュアルページを参照してください。

コード例 4–35 は、マルチスレッド自動モードでのサーバーを示したものです。

注 - RPC マルチスレッド対応アプリケーションを作成する場合は常に、スレッドライブラリ内でリンクしなければなりません。コンパイルコマンドで –lthread を指定して、スレッドライブラリを最後にリンクするようにしなければなりません。

次のように入力してコード例 4–35 のプログラムを作成します。

```
$ cc time_svc.c -lnsl -lthread
```

コード例 4–35　マルチスレッド自動モードのサーバー

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <synch.h>
#include <thread.h>
```

```
#include "time_prot.h"

void time_prog();

main(argc, argv)
int argc;
char *argv[];
{
int transpnum;
char *nettype;
int mode = RPC_SVC_MT_AUTO;
int max = 20;       /* スレッド最大数を 20 に設定 */

if (argc > 2) {
 fprintf(stderr, "usage: %s [nettype]\n", argv[0]);
 exit(1);
}

if (argc == 2)
 nettype = argv[1];
else
 nettype = "netpath";

if (!rpc_control(RPC_SVC_MTMODE_SET, &mode)) {
 printf("RPC_SVC_MTMODE_SET: failed\n");
 exit(1);
}
if (!rpc_control(RPC_SVC_THRMAX_SET, &max)) {
 printf("RPC_SVC_THRMAX_SET: failed\n");
 exit(1);
}
transpnum = svc_create( time_prog, TIME_PROG, TIME_VERS,
  nettype);

if (transpnum == 0) {
 fprintf(stderr, "%s: cannot create %s service.\n",
 argv[0], nettype);
 exit(1);
}
svc_run();
}

/*
 * サーバーのディスパッチプログラムです。RPC サーバーライブラリは、
 * サーバーのディスパッチャルーチン time_prog () を実行するスレッドを
 * 作成します。RPC ライブラリがスレッドを廃棄した後に行われます。
 */

static void
time_prog(rqstp, transp)
 struct svc_req *rqstp;
 SVCXPRT *transp;
{
```

(続く)

```
 switch (rqstp->rq_proc) {
  case NULLPROC:
   svc_sendreply(transp, xdr_void, NULL);
   return;
  case TIME_GET:
   dotime(transp);
   break;
  default:
   svcerr_noproc(transp);
   return;
 }
}
dotime(transp)
SVCXPRT *transp;
{

 struct timev rslt;
 time_t thetime;

 thetime = time((time_t *)0);
 rslt.second = thetime % 60;
 thetime /= 60;
 rslt.minute = thetime % 60;
 thetime /= 60;
 rslt.hour = thetime % 24;
 if (!svc_sendreply(transp, xdr_timev,(caddr_t) &rslt)) {
  svcerr_systemerr(transp);
 }
}
```

コード例 4–36 は、サーバーの time_prot.h ヘッダーファイルを示します。

コード例 4–36 マルチスレッド自動モード : time_prot.h ヘッダーファイル

```
include <rpc/types.h>

struct timev {
 int second;
 int minute;
 int hour;
};

typedef struct timev timev;
bool_t xdr_timev();

#define TIME_PROG 0x40000001
```

(続く)

```
#define TIME_VERS 1
#define TIME_GET 1
```

マルチスレッド・ユーザー・モード

マルチスレッド・ユーザー・モードでは、RPC ライブラリはスレッドを作成しません。このモードは、基本的には、シングルスレッド、またはデフォルトのモードのように作動します。唯一の違いは、データ構造のコピー (サービス・ディスパッチ・ルーチンへのトランスポートサービスなど) をマルチスレッド対応に引き渡す点です。

RPC サーバー開発者は、スレッドライブラリ全体のスレッドの作成と管理に対する責任を持ちます。ディスパッチルーチンでは、サービス開発者は、プロシージャの実行を新規作成のまたは既存のスレッドに割り当てることができます。`thr_create()` API は、さまざまな属性を持つスレッドを作成するために使用されます。すべてのスレッドのライブラリインタフェースは、`thread.h` で定義されます。詳細は、`pthread_create(3T)` マニュアルページを参照してください。

このモードは、サーバー開発者に幅広い柔軟性を提供しています。スレッドは、サービス要件に応じたスタックサイズを持ちます。スレッドは限定されます。異なるプロシージャは、異なる特長を持つスレッドによって実行されます。サービス開発者は、サービスの一部をシングルスレッドで実行できます。また、特定のスレッドに固有のシグナル処理を行うこともできます。

自動モードの場合と同じように、`rpc_control()` ライブラリは、ユーザーモードに切り換える場合に使用されます。表 4–10 に示した `rpc_control()` 演算 (`RPC_SVC_MTMODE_GET()` 以外) は、マルチスレッド自動モードにだけ適用されます。マルチスレッド・ユーザー・モードまたはシングルスレッドのデフォルトモードで使用すると、演算の結果が定義できません。

ユーザーモードでのライブラリリソースの解放

マルチスレッド・ユーザー・モードでは、サービスプロシージャは、戻しの前に `svc_done()` を呼び出さなければなりません。`svc_done()` は、クライアント要求が指定のサービス・トランスポート・ハンドルに向けたサービスに割り当てたリ

ソースを解放しなければなりません。この機能は、クライアント要求がサービスされた後、あるいは応答の送信を妨げたエラーまたは異常な状態の後に呼び出されます。svc_done() が呼び出された後に、サービス・トランスポート・ハンドルは、サービスプロシージャによって参照されるべきではありません。コード例 4–37 は、マルチスレッド・ユーザー・モードでのサーバーを示します。

注 - svc_done() は、マルチスレッド・ユーザー・モード内でだけ呼び出すことができます。詳細は、rpc_svc_calls(3N) マニュアルページを参照してください。

コード例 4–37 マルチスレッド・ユーザー・モード : rpc_test.h

```
#define SVC2_PROG 0x30000002
#define SVC2_VERS 1
#define SVC2_PROC_ADD 1)
#define SVC2_PROC_MULT 2

struct intpair {
 u_short a;
 u_short b;
};

typedef struct intpair intpair;

struct svc2_add_args {
 int argument;
 SVCXPRT *transp;
};

struct svc2_mult_args {
 intpair mult_argument;
 SVCXPRT *transp;
};

extern bool_t xdr_intpair();

#define NTHREADS_CONST 500
```

コード例 4–38 は、マルチスレッド・ユーザー・モードでのクライアントです。

コード例 4–38 マルチスレッド・ユーザー・モードでのクライアント

```
#define  _REENTRANT
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/uio.h>
#include <netconfig.h>
```

```
#include <netdb.h>
#include <rpc/nettype.h>
#include <thread.h>
#include "rpc_test.h"
void *doclient();
int NTHREADS;
struct thread_info {
 thread_t client_id;
 int client_status;
};
struct thread_info save_thread[NTHREADS_CONST];
main(argc, argv)
 int argc;
 char *argv[];
{
 int index, ret;
 int thread_status;
 thread_t departedid, client_id;
 char *hosts;
 if (argc < 3) {
  printf("Usage: do_operation [n] host\n");
  printf("\twhere n is the number of threads\n");
  exit(1);
 } else
  if (argc == 3) {
   NTHREADS = NTHREADS_CONST;
   hosts = argv[1];  /* live_host */
  } else {
   NTHREADS = atoi(argv[1]);
   hosts = argv[2];
  }
 for (index = 0; index < NTHREADS; index++){
  if (ret = thr_create(NULL, NULL, doclient,
  (void *)  hosts, THR_BOUND, &client_id)){
   printf("thr_create failed: return value %d", ret);
   printf(" for %dth thread\n", index);
   exit(1);
  }
  save_thread[index].client_id = client_id;
 }
 for (index = 0; index < NTHREADS; index++){
  if (thr_join(save_thread[index].client_id, &departedid,
  (void *)
  &thread_status)){
   printf("thr_join failed for thread %d\n",
   save_thread[index].client_id);
   exit(1);
  }
  save_thread[index].client_status = thread_status;
 }
}
 void *doclient(host)
 char *host;
```

(続く)

```
{
 struct timeval tout;
 enum clnt_stat test;
 int result = 0;
 u_short mult_result = 0;
 int add_arg;
 int EXP_RSLT;
 intpair pair;
 CLIENT *clnt;
 if ((clnt = clnt_create(host, SVC2_PROG, SVC2_VERS, "udp"
==NULL) {
  clnt_pcreateerror("clnt_create error: ");
  thr_exit((void *) -1);
 }
 tout.tv_sec = 25;
 tout.tv_usec = 0;
 memset((char *) &result, 0, sizeof (result));
 memset((char *) &mult_result, 0, sizeof (mult_result));
 if (thr_self() % 2){
  EXP_RSLT = thr_self() + 1;
  add_arg = thr_self();
  test = clnt_call(clnt, SVC2_PROC_ADD, (xdrproc_t) xdr_int,
  (caddr_t) &add_arg, (xdrproc_t) xdr_int, (caddr_t) &result,
  tout);
 } else {
  pair.a = (u_short) thr_self();
  pair.b = (u_short) 1;
  EXP_RSLT = pair.a * pair.b;
  test = clnt_call(clnt, SVC2_PROC_MULT, (xdrproc_t)
  xdr_intpair,
  (caddr_t) &pair, (xdrproc_t) xdr_u_short,
  (caddr_t) &mult_result, tout);
  result = mult_result;
 }
 if (test != RPC_SUCCESS) {
  printf("THREAD: %d clnt_call hav
  thr_exit((void *) -1);
 };
 thr_exit((void *) 0);
}
```

コード例 4–39 は、マルチスレッド・ユーザー・モードのサーバー側を示します。マルチスレッドパフォーマンスは、関数 `svc_getargs()` が `NULLPROC` 以外の各プロシージャに呼び出される場合は、引き数 (この場合には `xdr_void`) がなくても改善されます。これは、マルチスレッド自動モードモードとマルチスレッドユーザーモードにおいてです。詳細は、`rpc_svc_calls(3N)` マニュアルページを参照してください。

注 - RPC マルチスレッド対応アプリケーションを作成する場合は常に、スレッドライブラリ内でリンクしなければなりません。コンパイルコマンドで `-lthread` を指定して、スレッドライブラリを最後にリンクするようにしなければなりません。

コード例 4–39 マルチスレッド・ユーザー・モードのサーバー側

```
#define  _REENTRANT
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/uio.h>
#include <signal.h>
#include <thread.h>
#include "operations.h"

SVCXPRT *xprt;
void add_mult_prog();
void *svc2_add_worker();
void *svc2_mult_worker();
main(argc, argv)
 int argc;
 char **argv;
{
 int transpnum;
 char *nettype;
 int mode = RPC_SVC_MT_USER;
       if(rpc_control(RPC_SVC_MTMODE_SET,&mode) == FALSE){
                printf(" rpc_control is failed to set AUTO mode\n");
                exit(0);
        }
 if (argc > 2) {
  fprintf(stderr, "usage: %s [nettype]\n", argv[0]);
  exit(1);
 }
 if (argc == 2)
  nettype = argv[1];
 else
  nettype = "netpath";
 transpnum = svc_create(add_mult_prog, SVC2_PROG,
  SVC2_VERS, nettype);
 if (transpnum == 0) {
  fprintf(stderr, "%s: cannot create %s service.\n", argv[0],
   nettype);
  exit(1);
 }
 svc_run();
}
void add_mult_prog (rqstp, transp)
 struct svc_req *rqstp;
 SVCXPRT *transp;
{
```

(続く)

```
int argument;
u_short mult_arg();
intpair mult_argument;
bool_t (*xdr_argument)();
struct svc2_mult_args *sw_mult_data;
struct svc2_add_args *sw_add_data;
int ret;
thread_t worker_id;
switch (rqstp->rq_proc){
 case NULLPROC:
  svc_sendreply(transp, xdr_void, (char *) 0);
  svc_done(transp);
  break;
 case SVC2_PROC_ADD:
  xdr_argument = xdr_int;
  (void) memset((char *) &argument, 0, sizeof (argument));
  if (!svc_getargs(transp, xdr_argument,
  (char *) &argument)){
   printf("problem with getargs\n");
   svcerr_decode(transp);
   exit(1);
  }
  sw_add_data = (struct svc2_add_args *)
  malloc(sizeof (struct svc2_add_args));
  sw_add_data->transp = transp;
  sw_add_data->argument = argument;
  if (ret = thr_create(NULL, THR_MIN_STACK + 16 * 1024,
  svc2_add_worker, (void *) sw_add_data, THR_DETACHED,
   printf("SERVER: thr_create failed:");
   printf(" return value %d", ret);
   printf(" for add thread\n");
   exit(1);
  }
  break;
 case SVC2_PROC_MULT:
  xdr_argument = xdr_intpair;
  (void) memset((char *) &mult_argument, 0,
  sizeof (mult_argument));
  if (!svc_getargs(transp, xdr_argument,
  (char *) &mult_argument)){
   printf("problem with getargs\n");
   svcerr_decode(transp);
   exit(1);
  }
  sw_mult_data = (struct svc2_mult_args *)
  malloc(sizeof (struct svc2_mult_args));
  sw_mult_data->transp = transp;
  sw_mult_data->mult_argument.a = mult_argument.a;
  sw_mult_data->mult_argument.b = mult_argument.b;
  if (ret = thr_create(NULL, THR_MIN_STACK + 16 * 1024,
  svc2_mult_worker, (void *) sw_mult_data, THR_DETACHED,
  &worker_id)){
   printf("SERVER: thr_create failed:");
```

(続く)

```
    printf("return value %d", ret);
    printf("for multiply thread\n");
    exit(1);

   break;
  default:
   svcerr_noproc(transp);
   svc_done(transp);
   break;
 }
}

u_short mult_arg();
int add_one();
void *svc2_add_worker(add_arg)
struct svc2_add_args *add_arg;
{ int *result;
 bool_t (*xdr_result)();
 xdr_result = xdr_int;
 result = *malloc(sizeof (int));
 *result = add_one(add_arg->argument);
 if (!svc_sendreply(add_arg->transp, xdr_result,
 (caddr_t) result)){
  printf("sendreply failed\n");
  svcerr_systemerr(add_arg->transp);
  svc_done(add_arg->transp);
  thr_exit((void *) -1);
 }
 svc_done(add_arg->transp);
 thr_exit((void *) 0);
}
void *svc2_mult_worker(m_arg)
struct svc2_mult_args *m_arg;
{
 u_short *result;
 bool_t (*xdr_result)();
 xdr_result = xdr_u_short;
 result = (u_short *) malloc(sizeof (u_short));
 *result = mult_arg(&m_arg->mult_argument);
 if (!svc_sendreply(m_arg->transp, xdr_result,
 (caddr_t) result)){
  printf("sendreply failed\n");
  svcerr_systemerr(m_arg->transp);
  svc_done(m_arg->transp);
  thr_exit((void *) -1);
 }
 svc_done(m_arg->transp);
 thr_exit((void *) 0);
}
u_short mult_arg(pair)
 intpair *pair;
{
 u_short result;
```

(続く)

```
 result = pair->a * pair->b;
 return (result);}
int add_one(arg)
 int arg;
{
 return (++arg);
}
```

接続型トランスポート

コード例 4–40 に示すサンプルプログラムは、あるホストのファイルを別のホストにコピーするプログラムです。RPC の send() 呼び出しで標準入力から読み込まれたデータがサーバ ーreceive() に送信され、サーバーの標準出力に書き出されます。また、このプログラムでは、1 つの XDR ルーチンでシリアライズとデシリアライズの両方を実行する方法も示します。ここでは、接続型トランスポートを使用します。

コード例 4–40　遠隔コピー (両方向 XDR ルーチン)

```
/*
 * XDR ルーチン:
 *       復号化時にネットワークを読み取り fp に書き込む
 *       符号化時に fp を読み取りネットワークに書き込む
 */
#include <stdio.h>
#include <rpc/rpc.h>

bool_t
xdr_rcp(xdrs, fp)
 XDR *xdrs;
 FILE *fp;
{
 unsigned long size;
 char buf[BUFSIZ], *p;

 if (xdrs->x_op == XDR_FREE)                    /* 解放するものなし */
  return(TRUE);
```

(続く)

```
 while (TRUE) {
  if (xdrs->x_op == XDR_ENCODE) {
   if ((size = fread( buf, sizeof( char ), BUFSIZ, fp))
       == 0 && ferror(fp)) {
    fprintf(stderr, "can't fread\n");
    return(FALSE);
   } else
    return(TRUE);
  }
  p = buf;
  if (! xdr_bytes( xdrs, &p, &size, BUFSIZ))
   return(0);
  if (size == 0)
   return(1);
  if (xdrs->x_op == XDR_DECODE) {
   if (fwrite( buf, sizeof(char), size, fp) != size) {
    fprintf(stderr, "can't fwrite\n");
    return(FALSE);
   } else
    return(TRUE);
  }
 }
}
```

コード例 4–41 と コード例 4–42には、コード例 4–40 に示した xdr_rcp()ルーチンだけでシリアライズとデシリアライズを行うプログラムを示します。

コード例 4–41　遠隔コピー：クライアント側ルーチン

```
/* 送信側のルーチン */
#include <stdio.h>
#include <netdb.h>
#include <rpc/rpc.h>
#include <sys/socket.h>
#include <sys/time.h>
#include "rcp.h"

main(argc, argv)
 int argc;
 char **argv;
{
 int xdr_rcp();

 if (argc != 2 7) {
  fprintf(stderr, "usage: %s servername\n", argv[0]);
  exit(1);
```

(続く)

```
 }
 if( callcots( argv[1], RCPPROG, RCPPROC, RCPVERS, xdr_rcp,
stdin,
    xdr_void, 0 ) != 0 )
  exit(1);
 exit(0);
}

callcots(host, prognum, procnum, versnum, inproc, in, outproc,
out)
 char *host, *in, *out;
 xdrproc_t inproc, outproc;
{
 enum clnt_stat clnt_stat;
 register CLIENT *client;
 struct timeval total_timeout;

 if ((client = clnt_create( host, prognum, versnum,
"circuit_v")
    == (CLIENT *) NULL)) {
  clnt_pcreateerror("clnt_create");
  return(-1);
 }
 total_timeout.tv_sec = 20;
 total_timeout.tv_usec = 0;
 clnt_stat = clnt_call(client, procnum, inproc, in, outproc,
out,
                       total_timeout);
 clnt_destroy(client);
 if (clnt_stat != RPC_SUCCESS)
  clnt_perror("callcots");
 return((int)clnt_stat);
}
```

コード例 4–42 では、受信側のルーチンを定義します。サーバー側では、xdr_rcp() がすべての処理を自動的に実行することに注意してください。

コード例 4–42　遠隔コピー：サーバー側ルーチン

```
/*
 * 受信側ルーチン
 */
#include <stdio.h>
#include <rpc/rpc.h
#include "rcp.h"
```

(続く)

続き

```
main()
{
 void  rcp_service();
 if (svc_create(rpc_service,RCPPROG,RCPVERS,"circuit_v") == 0) {
  fprintf(stderr, "svc_create: errpr\n");
  exit(1);
 }
 svc_run();                     /* この関数は戻らない */
 fprintf(stderr, "svc_run should never return\n");
}

void
rcp_service(rqstp, transp)
 register struct svc_req *rqstp;
 register SVCXPRT *transp;
{
 switch(rqstp->rq_proc) {
  case NULLPROC:
   if (svc_sendreply(transp, xdr_void, (caddr_t) NULL) == FALSE)
    fprintf(stderr, "err: rcp_service");
   return;
  case RCPPROC:
   if (!svc_getargs( transp, xdr_rcp, stdout)) {
    svcerr_decode(transp);
    return();
   }
   if(!svc_sendreply(transp, xdr_void, (caddr_t) NULL)) {
    fprintf(stderr, "can't reply\n");
    return();
   }
   return();
  default:
   svcerr_noproc(transp);
   return();
 }

}
```

XDR によるメモリー割り当て

XDR ルーチンは通常、データのシリアライズとデシリアライズに使用します。XDR ルーチンは、多くの場合、メモリーを自動的に割り当て、そのメモリーを解放しま

す。一般に、配列や構造体へのポインタの代わりに NULL ポインタを渡されると、XDR ルーチンはデシリアライズを行うときに自分でメモリーを割り当てるようにします。次の例の xdr_chararr1() では、長さが SIZE の固定長配列を処理するようになっており、必要に応じてメモリーを割り当てることができません。

```
xdr_chararr1(xdrsp, chararr)
   XDR *xdrsp;
   char chararr[];
{
   char *p;
   int len;

   p = chararr;
   len = SIZE;
   return (xdr_bytes(xdrsp, &p, &len, SIZE));
}
```

chararr にすでに領域が確保されている場合は、サーバー側から次のように呼び出すことができます。

```
char chararr[SIZE];

svc_getargs(transp, xdr_chararr1, chararr);
```

XDR ルーチンや RPC ルーチンにデータを引き渡すための構造体は、基底アドレスが、アーキテクチャで決められた境界になるようなメモリーの割り当てにならなければなりません。XDR ルーチンでメモリーを割り当てるときも、次の点に注意して割り当てます。

- 呼び出し側が要求した場合にメモリー割り当てを行う
- 割り当てたメモリーへのポインタを返す

次の例では、第 2 引数が NULL ポインタの場合、デシリアライズされたデータを入れるためのメモリーが割り当てられます。

```
xdr_chararr2(xdrsp, chararrp)
   XDR *xdrsp;
   char **chararrp;
{

  int len;

  len = SIZE;
  return (xdr_bytes(xdrsp, charrarrp, &len, SIZE));
}
```

これに対する RPC 呼び出しを次に示します。

```
char *arrptr;
arrptr = NULL;
svc_getargs(transp, xdr_chararr2, &arrptr);
/*
 * ここで戻り値を使用
 */
svc_freeargs(transp, xdr_chararr2, &arrptr);
```

文字配列は、使用後に `svc_freeargs()` を使用して解放します。`svc_freeargs()` は、第 2 引数に `NULL` ポインタを渡された場合は何もしません。

これまでに説明したことをまとめると、次のようになります。

- 通常、XDR ルーチンではシリアライズ、デシリアライズ、メモリー解放を行います。
- `svc_getargs()` は、XDR ルーチンを呼び出してデシリアライズを行います。
- `svc_freeargs()` は、XDR ルーチンを呼び出してメモリーの解放を行います。

TS-RPC から TI-RPC への移行について

トランスポート独立の RPC ルーチン (TI-RPC ルーチン) を使用すると、アプリケーション開発者はトランスポート層へのアクセスレベルを自由に選択できます。最上位レベルのルーチンは、トランスポートが完全に抽象化されて、本当の意味でトランスポート独立になっています。下位レベルのルーチンを使用すると、旧バージョンと同じように個々のトランスポートに依存したアクセスレベルになります。

この節では、トランスポート特定 RPC (TS-RPC) アプリケーションを TI-RPC へ移行するための非公式ガイドになっています。表 4–11 では、いくつかのルーチンを選んで相違点を示します。ソケットとトランスポート層インタフェース (TLI) の移行の問題点についての詳細は、『*Transport Interfaces Programming Guide*』を参照してください。

アプリケーションの移行

TCP または UDP に基づくアプリケーションはバイナリ互換モードで実行できます。すべてのソースファイルをコンパイルし直したり、リンクし直したりできるのは、一部のアプリケーションだけです。RPC 呼び出しだけを使用し、ソケット、TCP、UDP に固有の機能を使用していないアプリケーションがこれに当たります。

ソケットセマンティクスに依存していたり、TCP や UDP の固有の機能を使用しているアプリケーションでは、コードの変更や追加が必要な場合があります。ホストアドレス形式を使用したり、バークレイ UNIX の特権ポートを使用するアプリケーションがこれに当たります。

ライブラリ内部や個々のソケット仕様に依存していたり、特定のトランスポートアドレスに依存するアプリケーションは、移行の手間も大きく、本質的な変更が必要な場合もあります。

移行の必要性

TI-RPC へ移行することの利点を次に示します。

- アプリケーションがトランスポート独立になるため、より多くのトランスポート上で実行できます。
- アプリケーションの効率を改善する新規インタフェースが使用できます。

- バイナリレベルの互換性の影響は、ネイティブモードより少なくなります。
- 旧インタフェースは、将来のバージョンで使用できなくなる可能性があります。

特殊事項

libnsl ライブラリ

ネットワーク関数は libc から外されました。コンパイル時には libnsl を明示的に指定して、ネットワーク・サービス・ルーチンをリンクする必要があります。

旧インタフェース

旧インタフェースの多くは libnsl ライブラリでもサポートされていますが、TCP と UDP でしか使用できません。それ以外の新たなトランスポートを利用するには、新インタフェースを使用する必要があります。

名前 - アドレス変換機能

トランスポート独立にするには、アドレスを直接使用できません。すなわち、アプリケーションでアドレス変換を行う必要があります。

TI-RPC と TS-RPC の相違点

トランスポート独立型の RPC とトランスポート特定の RPC との主な相違点を表 4–11 に示します。TI-RPC と TS-RPC の比較については、167ページの「旧バージョンとの比較」 のサンプルプログラムを参照してください。

表 4–11 TI-RPC と TS-RPC の相違点

項目	**TI-RPC**	**TS- RPC**
デフォルトのトランスポート選択	TI-RPC では TLI インタフェースを使用する。	TS-RPC ではソケットインタフェースを使用する。
RPC アドレス結合	TI-RPC ではサービスの結合に `rpcbind()` を使用する。`rpcbind()` はアドレスを汎用アドレス形式で扱う。	TS-RPC ではサービスの結合に `portmap` を使用する。
トランスポート情報	トランスポート情報はローカルファイル `/etc/netconfig` に保存する。`netconfig` で指定したトランスポートはすべてアクセス可能になる。	トランスポートは TCP と UDP だけをサポートする。
ループバックトランスポート	`rpcbind` サービスではサーバー登録に安全なループバックトランスポートが必要。	TS-RPC サービスではループバックトランスポートは不要。
ホスト名の解決	TI-RPC のホスト名の解決順序は、`/etc/netconfig` で指定した動的ライブラリのエントリ順で決定される。	ホスト名の解決はネームサービスが 実行する。順序は `hosts` データベースの状態で設定される。
ファイル記述子	ファイル記述子は TLI 端点とみなす。	ファイル記述子はソケットとみなす。
`rpcgen`	TI-RPC の `rpcgen` では、複数引数、値渡し、サンプルクライアントとサンプルサーバーファイルのサポートを追加。	SunOS 4.1 と直前のリリースは TI-RPC `rpcgen` に対して一覧表示された特徴はサポートしない。
ライブラリ	TI-RPC では、アプリケーションが `libnsl` ライブラリにリンクしていることを必要とする。	TS-RPC の機能はすべて `libc` で提供される。
マルチスレッドのサポート	マルチスレッド RPC クライアントとサーバーがサポートされる。	マルチスレッド RPC はサポートされない。

関数の互換性のリスト

この節では、RPC ライブラリ関数を機能別にグループ化して示します。各グループ内では、旧バージョンと同じ関数、機能が追加された関数、旧バージョンにはなかった新規関数に分けて示します。

注 - アスタリスクの付いた関数は、新バージョンへの移行期間はサポートされていますが、Solaris の将来のバージョンではなくなる可能性があります。

クライアントハンドルの作成

次の関数は、旧バージョンと同じで、現在の SunOS で使用できます。

```
clnt_destroy
clnt_pcreateerror
*clntraw_create
clnt_spcreateerror
*clnttcp_create
*clntudp_bufcreate
*clntudp_create
clnt_control
clnt_create
clnt_create_timed
clnt_create_vers
clnt_dg_create
clnt_raw_create
clnt_tli_create
clnt_tp_create
clnt_tp_create_timed
clnt_vc_create
```

サービスの作成と廃棄

次の関数は、旧バージョンと同じで、現在の SunOS で使用できます。

```
svc_destroy
svcfd_create
*svc_raw_create
*svc_tp_create
*svcudp_create
*svc_udp_bufcreate
svc_create
svc_dg_create
```

(続く)

続き

```
svc_fd_create
svc_raw_create
svc_tli_create
svc_tp_create
svc_vc_create
```

サービスの登録と登録削除

次の関数は、旧バージョンと同じで、現在の SunOS で使用できます。

```
*registerrpc
*svc_register
*svc_unregister
xprt_register
xprt_unregister
rpc_reg
svc_reg
svc_unreg
```

SunOS 4.x との互換性呼び出し

次の関数は、旧バージョンと同じで、現在の SunOS で使用できます。

```
*callrpc
clnt_call
*svc_getcaller - IP に基づくトランスポートでのみ使用可
rpc_call
svc_getrpccaller
```

ブロードキャスト

次の関数の機能は旧バージョンと同じです。旧バージョンとの互換性を保つためにだけサポートされています。

```
*clnt_broadcast
```

clnt_broadcast() は portmap サービスにだけブロードキャストできます。

portmap と rpcbind の両方にブロードキャストできる次の関数が現在の SunOS で使用できます。

```
rpc_broadcast
```

アドレス管理

TI-RPC ライブラリ関数は、portmap と rpcbind の両方で使用できますが、それぞれサービスが異なるため、次のように 2 組の関数が提供されています。

次の関数は portmap と共に使用します。

```
pmap_set
pmap_unset
pmap_getport
pmap_getmaps
pmap_rmtcall
```

次の関数は rpcbind と共に使用します。

```
rpcb_set
rpcb_unset
rpcb_getaddr
rpcb_getmaps
rpcb_rmtcall
```

認証

次の関数の機能は旧バージョンと同じです。旧バージョンとの互換性を保つためにだけサポートされています。

```
authdes_create
authunix_create
authunix_create_default
authdes_seccreate
authsys_create
authsys_create_default
```

その他の関数

現バージョンの rpcbind ではタイムサービス (主として、安全な RPC のためにクライアント側とサーバー側の時間を同期させるときに使用) が提供されており、rpcb_gettime() 関数で利用できます。pmap_getport() と rpcb_getaddr() は、登録サービスのポート番号を取り出すときに使用します。サーバーでバージョンが 2、3、4 の rcpbind が実行されている場合には、rpcb_getaddr() を使用します。pmap_getport() はバージョン 2 が実行されている場合しか使用できません。

旧バージョンとの比較

コード例 4–43 と コード例 4–44 では、クライアント作成部分が TS-RPC と TI-RPC とでどう違うかを示します。どちらのプログラムも次のことを実行します。

- UDP 記述子を作成します。
- 遠隔ホストの RPC 結合プロセスと通信してサービスアドレスを得ます。
- 遠隔サービスのアドレスを記述子に結合します。
- クライアントハンドルを作成してタイムアウト値を設定します。

コード例 **4–43**　TS-RPC におけるクライアント作成

```
 struct hostent *h;
 struct sockaddr_in sin;
 int sock = RPC_ANYSOCK;
 u_short port;
 struct timeval wait;

 if ((h = gethostbyname( "host" )) == (struct hostent *) NULL)
{
  syslog(LOG_ERR, "gethostbyname failed");
  exit(1);
 }
 sin.sin_addr.s_addr = *(u_int *) hp->h_addr;
 if ((port = pmap_getport(&sin, PROGRAM, VERSION, "udp")) == 0) {
  syslog (LOG_ERR, "pmap_getport failed");
  exit(1);
 } else
  sin.sin_port = htons(port);
 wait.tv_sec = 25;
 wait.tv_usec = 0;
 clntudp_create(&sin, PROGRAM, VERSION, wait, &sock);
```

TI-RPC では、UDP トランスポートは `netid` *udp* を持つものとみなします。`netid` はよく知られた名前でなくてもかまいません。

コード例 **4–44**　TI-RPC でのクライアント作成

```
 struct netconfig *nconf;
 struct netconfig *getnetconfigent();
 struct t_bind *tbind;
 struct timeval wait;

 nconf = getnetconfigent("udp");
 if (nconf == (struct netconfig *) NULL) {
  syslog(LOG_ERR, "getnetconfigent for udp failed");
  exit(1);
 }
 fd = t_open(nconf->nc_device, O_RDWR, (struct t_info *)NULL);
 if (fd == -1) {
  syslog(LOG_ERR, "t_open failed");
  exit(1);
 }
 tbind = (struct t_bind *) t_alloc(fd, T_BIND, T_ADDR);
 if (tbind == (struct t_bind *) NULL) {
  syslog(LOG_ERR, "t_bind failed");
  exit(1);
 }
 if (rpcb_getaddr( PROGRAM, VERSION, nconf, &tbind->addr, "host")
```

(続く)

続き

```
       == FALSE) {
 syslog(LOG_ERR, "rpcb_getaddr failed");
 exit(1);
}
cl = clnt_tli_create(fd, nconf, &tbind->addr, PROGRAM, VERSION,
                     0, 0);
(void) t_free((char *) tbind, T_BIND);
if (cl == (CLIENT *) NULL) {
 syslog(LOG_ERR, "clnt_tli_create failed");
 exit(1);
}
wait.tv_sec = 25;
wait.tv_usec = 0;
clnt_control(cl, CLSET_TIMEOUT, (char *) &wait);
```

コード例 4–45 とコード例 4–46 では、ブロードキャスト部分が旧バージョンと SunOS 5.3 とでどう違うかを示します。SunOS 4.*x* の `clnt_broadcast()` は SunOS 5.3 の `rpc_broadcast()` とほぼ同じです。大きく異なるのは、`collectnames()` 関数で重複アドレスを削除し、ブロードキャストに応答したホスト名を表示する点です。

コード例 4–45 TS-RPC におけるブロードキャスト

```
statstime sw;
extern int collectnames();

clnt_broadcast(RSTATPROG, RSTATVERS_TIME, RSTATPROC_STATS,
     xdr_void, NULL, xdr_statstime, &sw, collectnames);
 ...
collectnames(resultsp, raddrp)
 char *resultsp;
 struct sockaddr_in *raddrp;
{
 u_int addr;
 struct entry *entryp, *lim;
 struct hostent *hp;
 extern int curentry;

 /* 重複アドレスはカット */
 addr = raddrp->sin_addr.s_addr;
 lim = entry + curentry;
 for (entryp = entry; entryp < lim; entryp++)
  if (addr == entryp->addr)
```

(続く)

```
   return (0);
 ...
 /* ホスト名がわかればホスト名、わからなければアドレスを表示 */
 hp = gethostbyaddr(&raddrp->sin_addr.s_addr, sizeof(u_int),
     AF_INET);
 if( hp == (struct hostent *) NULL)
  printf("0x%x", addr);
 else
  printf("%s", hp->h_name);
}
```

コード例 4–46 は、TI-RPC におけるブロードキャストを示します。

コード例 **4–46**　TI-RPC におけるブロードキャスト

```
statstime sw;
extern int collectnames();

rpc_broadcast(RSTATPROG, RSTATVERS_TIME, RSTATPROC_STATS,
     xdr_void, NULL, xdr_statstime, &sw, collectnames, (char *)
0);
 ...
collectnames(resultsp, taddr, nconf)
 char *resultsp;
 struct t_bind *taddr;
 struct netconfig *nconf;
{
 struct entry *entryp, *lim;
 struct nd_hostservlist *hs;
 extern int curentry;
 extern int netbufeq();

 /* 重複アドレスはカット */
 lim = entry + curentry;
 for (entryp = entry; entryp < lim; entryp++)
  if (netbufeq( &taddr->addr, entryp->addr))
   return (0);
 ...
 /* ホスト名がわかればホスト名、わからなければアドレスを表示 */
 if (netdir_getbyaddr( nconf, &hs, &taddr->addr ) == ND_OK)
  printf("%s", hs->h_hostservs->h_host);
 else {
  char *uaddr = taddr2uaddr(nconf, &taddr->addr);
  if (uaddr) {
   printf("%s\n", uaddr);
   (void) free(uaddr);
```

(続く)

```
  } else
   printf("unknown");
 }
}
netbufeq(a, b)
 struct netbuf *a, *b;
{
 return(a->len == b->len && !memcmp( a->buf, b->buf, a->len));
}
```

パートIII NIS+

Part 3 では NIS+ のための API について説明します。

- 第 5 章

第 5 章

NIS+ プログラミングガイド

この章は、NIS+ アプリケーションのプログラミングインタフェースの基礎について説明し、詳しいサンプルプログラムを示します。NIS+ API は、Solaris を使用するネットワーク用のアプリケーションを構築するプログラマを対象としています。NIS+ API には、アプリケーションをサポートするための基本的な機能が用意されています。

NIS+ ネットワークネームサービスが対象とするクライアント-サーバーネットワークは、数個のサーバーが 10 個のクライアントをサポートする程度の単純なローカルエリアネットワークから、全世界のさまざまなサイトに位置する 20 ～ 100 個の専用サーバーが 10,000 個ものマルチベンダークライアントをサポートし、さまざまな公衆ネットワークで連結された大規模なものまであります。

NIS+ の概要

ドメイン

NIS+ は、階層ドメインをサポートします。図 5–1 に、簡単な例を示します。

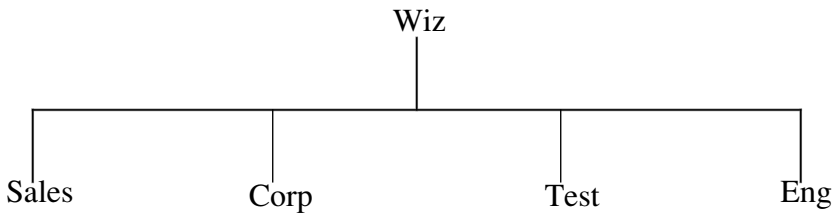


図 5–1　NIS+ のドメイン

NIS+ の各ドメインは、組織の各部分のワークステーション、ユーザー、ネットワークサービスを記述するデータの集合です。NIS+ の各ドメインは、他のドメインとは独立に管理できます。この機能のおかげで、NIS+ は、小さなものから大きなものまでさまざまな規模のネットワークで使用できます。

サーバー

各ドメインはいくつかのサーバーでサポートされています。このうち中心となるサーバーをマスタサーバー、バックアップサーバーを複製サーバーといいます。マスタサーバーと複製サーバーの両方で NIS+ プログラムを実行します。オリジナルテーブルはマスタサーバーが保管し、複製サーバーはそのコピーを保存します。

NIS+ では、段階的に複製を更新します。最初にマスタサーバーを変更すると、次にその変更が自動的に複製サーバーに伝えられ、名前空間全体で有効となります。

テーブル

NIS+ では、情報がマップやゾーンファイルではなくテーブルに保存されます。NIS+ には 図 5–2 で示されている 16 種類の定義済みテーブル (システムテーブル) があります。

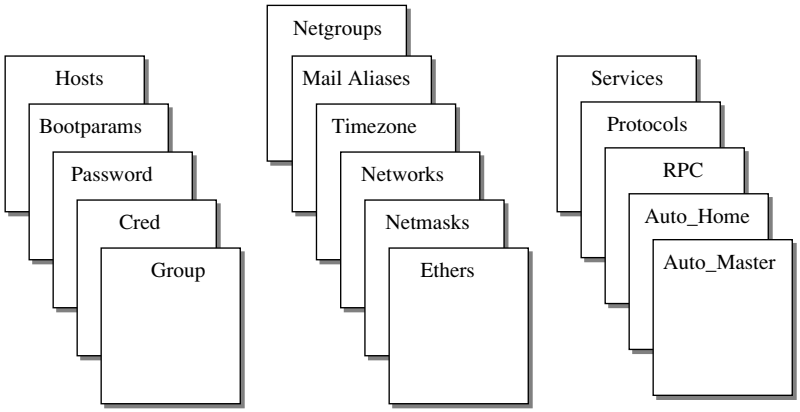


図 5–2　NIS+ のテーブル

各テーブルにはそれぞれ違う種類の情報が保存されます。たとえば、`Hosts` テーブルには、ホスト名とインターネットアドレスのペアが保存され、`Password` テーブルにはネットワークユーザーに関する情報が保存されます。

NIS+ テーブルは、次の 2 つの点で NIS マップから大きく改善されています。1 つは、NIS+ テーブルでは第 1 カラム (「キー」ともいう) だけでなく任意のカラムにアクセスできる点です。このため、NIS の `hosts.byname` マップや `hosts.byaddr` マップのような重複マップが不要になりました。もう 1 つは、NIS+ テーブルの情報には、テーブルレベル、エントリレベル、カラムレベルの 3 つのレベルでアクセスできる点です。

NIS+ のセキュリティ

NIS+ のセキュリティモデルには、許可と認証の 2 つの機能があります。まず、名前空間上の各オブジェクトごとに、どのような主体にどの種類の操作を許すかを指定します。これを許可といいます。名前空間へのアクセスが要求されると、NIS+ はその要求を出した主体を確認します。NIS+ は、要求の発信元を認識すると、その特定の主体に対する特定の操作をオブジェクトが認可しているか確かめます。NIS+ はこのように、認証情報とオブジェクトごとの許可情報に基づいて、アクセス要求を許すか拒否するかを決定します。

ネームサービススイッチ

NIS+ はネームサービススイッチとともに使用できます。ネームサービススイッチ (単に「スイッチ」ともいう) を使用すると、Solaris 2.*x* を使用するワークステーショ

ンで、複数のネットワーク情報サービスから情報を得ることができます。特に、ローカルファイル、/etc ファイル、NIS マップ、DNS ゾーンファイル、NIS+ テーブルから情報が得られます。ネームサービススイッチを使用すると、単に情報源を選択するだけでなく、ワークステーションで情報の種類別に異なる情報源を使用できます。ネームサービスの設定は、/etc/nsswitch.conf ファイルで行います。

NIS+ の管理コマンド

NIS+ では、名前空間を管理するのに必要なコマンドがすべて提供されています。

表 5–1 にその要約を示します。

表 5–1 NIS+ のネーム空間管理コマンド

コマンド	説明
nischgrp	NIS+ オブジェクトのグループ所有者を変更する。
nischmod	オブジェクトのアクセス権を変更する。
nischown	NIS+ オブジェクトの所有者を変更する。
nisgrpadm	NIS+ グループの作成と破棄、グループメンバーリストの表示を行う。また、グループにメンバーを追加または削除したり、グループメンバーかどうかのテストを行う。
niscat	NIS+ テーブルの内容を表示する。
nisgrep	NIS+ テーブルのエントリを検索する。
nisls	NIS+ ディレクトリの内容をリストする。
nismatch	NIS+ テーブルのエントリを検索する。
nisaddent	/etc ファイル、または、NIS マップの情報を NIS+ テーブルに追加する。
nistbladm	NIS+ テーブルの作成や削除を行う。また、NIS+ テーブルのエントリを追加、変更、削除する。
nisaddcred	NIS+ 主体の資格を作成し、それを Cred テーブルに保存する。
nispasswd	NIS+ の Passwd テーブルのパスワード情報を変更する。

表 5–1　NIS+ のネーム空間管理コマンド　続く

コマンド	説明
nisupdkeys	NIS+ オブジェクトに保存されている公開鍵を更新する。
nisinit	NIS+ のクライアントまたはサーバーを初期化する。
nismkdir	NIS+ ディレクトリを作成し、そのマスタサーバーと複製サーバーを指定する。
nisrmdir	ネーム空間から、NIS+ ディレクトリとその複製を削除する。
nissetup	org_dir と groups_dir の 2 つのディレクトリを作成し、NIS+ ドメインに対する全種類の NIS+ テーブル (空のテーブル) を作成する。
rpc.nisd	NIS+ のサーバープロセス。
nis_cachemgr	NIS+ クライアントの NIS+ キャッシュマネージャを起動する。
nischttl	NIS+ オブジェクトの生存時間を変更する。
nisdefaults	NIS+ オブジェクトのデフォルト値 (ドメイン名、グループ名、ワークステーション名、NIS+ 主体名、アクセス権、ディレクトリの検索パス、生存時間) をリストする。
nisln	2 つの NIS+ オブジェクト間のシンボリックリンクを作成する。
nisrm	ディレクトリ以外の NIS+ オブジェクトを名前空間から削除する。
nisshowcache	NIS+ キャッシュマネージャが管理している NIS+ 共有キャッシュの内容をリストする。

NIS+ の API

NIS+ の アプリケーションプログラマーズインタフェース (API) は関数の集合です。アプリケーションでは、これらの関数を呼び出して NIS+ オブジェクトにアクセスしたり変更したりできます。NIS+ の API には 54 個の関数があり、それらは次の 9 つのグループに分類されます。

- オブジェクト操作に使用する関数 (nis_names)

- テーブルアクセスに使用する関数 (nis_tables)
- ローカル名を取り出す関数 (nis_local_names)
- グループ操作に使用する関数 (nis_groups)
- サーバー関連の関数 (nis_server)
- データベースアクセスに使用する関数 (nis_db)
- エラーメッセージを表示する関数 (nis_error)
- トランザクションのログを取る関数 (nis_admin)
- その他の関数 (nis_subr)

表 5–2 では、これらの関数をグループごとにまとめて簡単に説明します。グループ名は、NIS+ のマニュアルページのグループ名と同じです。

表 5–2 NIS+ の API 関数

関数	説明
nis_names()	オブジェクトの検索と操作
nis_lookup()	NIS+ オブジェクトのコピーを返す。リンクをたどることができる。エントリオブジェクトの検索はできないが、リンクがエントリオブジェクトを指している場合は、それを返すことができる。
nis_add()	NIS+ オブジェクトを名前空間に追加する。
nis_remove()	NIS+ オブジェクトを名前空間から削除する。
nis_modify()	名前空間の NIS+ オブジェクトを変更する。
nis_tables()	テーブルの検索と更新を行う。
nis_list()	NIS+ 名前空間内のテーブルを検索して、検索条件に一致するエントリオブジェクトを返す。テーブル間のリンクと検索パスをたどることができる。
nis_add_entry()	NIS+ テーブルにエントリオブジェクトを追加する。既存オブジェクトがあれば操作を中止するか上書きするかを指定できる。操作が正常終了した場合は、追加したオブジェクトのコピーが返される。
nis_freeresult()	nis_result 構造体に割り当てられたメモリーをすべて解放する。

表 5–2　NIS+ の API 関数　続く

関数	説明
nis_remove_entry()	NIS+ テーブルからエントリオブジェクトを削除する。削除するオブジェクトは、検索条件で指定するか、キャッシュされたオブジェクトコピーへのポインタで指定できる。検索条件で指定する場合は、その条件に一致するオブジェクトをすべて削除できる。したがって、検索条件を適切に指定すれば、テーブル内の全エントリを削除することもできる。操作が正常終了した場合は、削除したオブジェクトのコピーが返される。
nis_modify_entry()	NIS+ テーブルのエントリオブジェクトを変更する。変更するオブジェクトは、検索条件で指定するか、キャッシュされたオブジェクトコピーへのポインタで指定する。
nis_first_entry()	NIS+ テーブルの最初のエントリオブジェクトのコピーを返す
nis_next_entry()	NIS+ テーブルの「次」のエントリオブジェクトのコピーを返す。この関数への呼び出しと呼び出しの間に、テーブルの更新やエントリの追加、変更が行われる可能性があるので、返されたエントリの順序が実際のテーブル内のエントリ順序と一致しない場合がある。
nis_local_names()	現在のプロセスのデフォルト名を取り出す。
nis_local_directory()	ワークステーションの NIS+ ドメイン名を返す。
nis_local_host()	修飾子付きのワークステーション名を返す。修飾子付きのワークステーション名は、*host-name*、*domain-name* という形式になる。
nis_local_group()	現在の NIS+ グループ名を返す。現在の NIS+ グループは、環境変数 NIS_GROUP で指定される。
nis_local_principal()	NIS+ 主体名 (その主体の UID は呼び出し側プロセスに結合) を返す。
nis_getnames()	特定の名前に対する可能な展開形のリストを返す。
nis_freenames()	nis_getnames()が生成したリストが使用するメモリーを解放する。

表 5–2　NIS+ の API 関数　続く

関数	説明
nis_groups()	グループ操作と許可。
nis_ismember()	ある主体がグループのメンバーかどうか調べる。
nis_addmember()	グループにメンバーを追加する。メンバーは主体、グループ、ドメインのどれか。
nis_removemember()	グループからメンバーを削除する。
nis_creategroup()	グループオブジェクトを作成する。
nis_destroygroup()	グループオブジェクトを削除する。
nis_verifygroup()	グループオブジェクトが存在しているかどうか調べる。
nis_print_group_entry()	グループオブジェクトのメンバーになっている主体を表示する。
nis_server()	NIS+ アプリケーションのための種々のサービス。
nis_mkdir()	特定ホストの特定ディレクトリに対するサービスをサポートするためのデータベースを作成する。
nis_rmdir()	ホストからディレクトリを削除する。
nis_servstate()	NIS+ サーバーの状態変数の設定と読み込みを行い、内部キャッシュをフラッシュする。
nis_stats()	サーバーのパフォーマンスに関する統計情報を取り出す。
nis_getservlist()	特定のドメインをサポートするサーバーのリストを返す。
nis_freeservlist()	nis_getservlist() が返したサーバーリストが使用するメモリーを解放する。
nis_freetags()	nis_servstate() と nis_stats() の戻り値が使用するメモリーを解放する。

表 5–2　NIS+ の API 関数　続く

関数	説明
nis_db()	NIS+ サーバーとデータベースの間のインタフェース。NIS+ クライアントでは使用不可。
db_first_entry()	指定したテーブルの最初のエントリのコピーを返す。
db_next_entry()	指定したエントリの次のエントリのコピーを返す。
db_reset_next_entry()	最初または次のエントリシーケンスを終了する。
db_list_entries()	指定した属性に一致するエントリのコピーを返す。
db_remove_entry()	指定した属性に一致するエントリをすべて削除する。
db_add_entry()	指定した属性に一致するテーブルエントリを、指定したオブジェクトのコピーで置き換える。または、指定したオブジェクトをテーブルに追加する。
db_checkpoint()	テーブルの内容を再編成して、テーブルのアクセス効率を改善する。
db_standby()	データベースマネージャに資源の解放を勧める。
nis_error()	NIS+ のステータス値に対応するメッセージ文字列を取り出す関数。
nis_sperrno()	メッセージ文字列定数へのポインタを返す。
nis_perror()	メッセージ文字列定数を標準出力に表示する。
nis_lerror()	メッセージ文字列定数を syslog に送信する。
nis_sperror()	strdup() 使用するかまたはでコピーするために静的領域に割り当てられた文字列へのポインタを返す。
nis_admin()	トランザクションのログを取る関数。サーバーで使用。

表 5–2　NIS+ の API 関数　続く

関数	説明
nis_ping()	ディレクトリのマスタサーバーが、ディレクトリのタイムスタンプを作成するのに使用する。この関数を実行すると、複製も強制的に更新される。
nis_checkpoint()	ログデータを強制的にディスク上のテーブルに保存する。
nis_subr()	NIS+ のネームとオブジェクトの操作のための補助関数。
nis_leaf_of()	NIS+ ネームの最初のラベルを返す。返されたネームには、末尾のピリオドは付いていない。
nis_name_of()	ネームからドメイン関連のラベルをすべて削除して、固有のオブジェクト部分だけを返す。この関数に渡されたネームは、ローカルドメイン、または、その子ドメインでなければならない。それ以外の場合は NULL が返される。
nis_domain_of()	オブジェクトが入っているドメイン名を返す。返されたドメイン名はピリオドで終わっている。
nis_dir_cmp()	2 つの NIS+ 名を比較する。大文字小文字の違いは無視して比較し、ネームが同じか、派生関係にあるか、無関係かを返す。
nis_clone_object()	NIS+ オブジェクトの完全な複製を作成する。
nis_destroy_object()	nis_clone_object() で作成したオブジェクトを破棄する。
nis_print_object()	NIS+ オブジェクト構造体の内容を標準出力に表示する。

NIS+ サンプルプログラム

サンプルプログラムでは次のタスクを実行します。

- ローカル主体とローカルドメインを決定します。

- ローカルディレクトリオブジェクトのルックアップを行います。
- ローカルドメインの下に foo というディレクトリを作成します。
- ドメイン foo の下に groups_dir と org_dir というディレクトリを作成します。
- グループオブジェクト admins.foo を作成します。
- admins グループにローカル主体を追加します。
- org_dir.foo の下にテーブルを作成します。
- org_dir.foo テーブルにエントリを 2 つ追加します。
- admins グループの新たなメンバーリストを取り出して表示します。
- コールバックを使用して、ドメイン foo の下の名前空間をリストします。
- コールバックを使用して、作成したテーブルの内容をリストします。
- 次のものを削除して、作成した全オブジェクトを削除します。
  - admins グループのローカル主体
  - admins グループ
  - はじめにテーブル内のエントリ、次にテーブル自体
  - groups_dir と org_dir のディレクトリオブジェクト
  - foo ディレクトリオブジェクト

サンプルプログラムは典型的なアプリケーションとはいえません。通常は、ディレクトリとテーブルの作成と削除はコマンド行インタフェースで行い、アプリケーションでは NIS+ エントリオブジェクトの操作だけを行います。

サポートされないマクロの使用

サンプルプログラムでは <rpcsvc/nis.h> ファイルで定義されているマクロを使用しています。ここで定義されているマクロは、サポートが保証されている正式な API ではなく、将来変更されたりなくなったりする可能性があります。ここではサンプルプログラムで使用方法を説明するために使用していますが、実際のプログラムで使用するときはユーザー自身の責任で使用してください。サンプルプログラムで使用しているマクロを次に示します。

- NIS_RES_OBJECT
- ENTRY_VAL

- DEFAULT_RIGHTS

サンプルプログラムで使用する関数

サンプルプログラムでは、次の NIS+ API 関数を C 言語で使用する方法を示します。

nis_add()、nis_add_entry()、nis_addmember()

nis_creategroup()、nis_destroygroup()、nis_domain_of()

nis_freeresult()、nis_leaf_of()、nis_list()

nis_local_directory()、nis_local_principal()

nis_lookup()、nis_mkdir()、nis_perror()、nis_remove()

nis_remove_entry()、nis_removemember()

プログラムのコンパイル

コード例 5–1 が示すサンプルプログラムは、これを実行する NIS+ 主体がローカルドメインにディレクトリオブジェクトを作成することを許可されているものとして書かれています。このプログラムをコンパイルするコマンドを次に示します。

```
yourhost% cc -o example.c example -lnsl
```

このプログラムを実行するコマンドを次に示します。

```
yourhost% example [dir]
```

ここで、dir には NIS+ ディレクトリを指定します。サンプルプログラムは、すべての NIS+ オブジェクトをそのディレクトリに作成します。引数 dir を指定しないと、ローカルドメインの親ディレクトリにオブジェクトが作成されます。nis_lookup() の呼び出しでは、ディレクトリを指定する文字列に空白とローカルドメイン名が追加されることに注意してください。引数で指定するのは、作成した NIS+ オブジェクトを入れるための NIS+ ディレクトリ名です。このプログラムを実行する主体は、そのディレクトリ内でのオブジェクト作成許可を持っている必要があります。

コード例 **5–1**　NIS+ のプログラム例 example.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <rpcsvc/nis.h>

#define   MAX_MESG_SIZE 512
#define   BUFFER_SIZE 64
#define   TABLE_TYPE "test_data"

main(argc, argv)
 int  argc;
 char  *argv[];
{
 char  *saved_grp, *saved_name, *saved_owner;
 char  dir_name[NIS_MAXNAMELEN];
 char  local_domain[NIS_MAXNAMELEN];
 char  local_princip [NIS_MAXNAMELEN];
 char  org_dir_name [NIS_MAXNAMELEN];
 char  grp_name [NIS_MAXNAMELEN];
 char  grp_dir_name [NIS_MAXNAMELEN];

 char  table_name [NIS_MAXNAMELEN];
 nis_object    *dirobj, entdata;
 nis_result    *pres;
 u_int  saved_num_servers;
 int  err;

 if (argc == 2)
  sprintf (local_domain, "%s.", argv[1]);
 else
  strcpy (local_domain, "");

 strcat (local_domain, (char *) nis_local_directory());

 strcpy (local_princip, (char *) nis_local_principal());

 /*
  * 次の 2 つの目的で、ディレクトリオブジェクトからローカルドメインを探す。
  * 1.nis_object のテンプレートを得る。
  * 2.返されたディレクトリオブジェクトに含まれる情報を再利用する。
  * static で宣言した nis_object があり、それをほとんどそのままで
  * 使用できる場合は、nis_object 構造体を初期化するより、それを変更して
  * 使用するほうが簡単である。
  */

 pres = nis_lookup (local_domain, 0);
 if (pres->status != NIS_SUCCESS) {
  nis_perror (pres->status, "unable to lookup local
directory");
  exit (1);
 }

 /*
  * 親ディレクトリオブジェクトのほとんどのフィールドを再利用 - 変更する
```

(続く)

```
 * フィールドへのポインタを保存しておき、元のオブジェクトを解放しても
 * ポインタ参照が残らないようにする。
 */
dirobj = NIS_RES_OBJECT (pres);
saved_name = dirobj->DI_data.do_name;
saved_owner = dirobj--->zo_owner;
saved_grp = dirobj---->zo_group;

/*
 * 新たな名前、グループ、所有者、および、foo ドメインへの新たな
 * アクセス権を設定する。

 */
sprintf (dir_name, "%s.%s", "foo", local_domain);
sprintf (grp_name, "%s.%s", "admins", dir_name);
dirobj->DI_data.do_name = dir_name;
dirobj->zo_group = grp_name;
dirobj->zo_owner = local_princip;

/*
 * NIS+ のアクセス権は u_long 型に保存される。最上位バイトは使用不可で、
 * 次の 8 バイトは所有者に、続いて、グループ、その他の順に保存される。
 * この例では、ディレクトリへのアクセス権は "----rmcdrmcd----" の
 * パターンに従って与える。
 */
dirobj->zo_access = ((NIS_READ_ACC + NIS_MODIFY_ACC
     + NIS_CREATE_ACC + NIS_DESTROY_ACC) << 16)
     | ((NIS_READ_ACC + NIS_MODIFY_ACC
     + NIS_CREATE_ACC + NIS_DESTROY_ACC) << 8);


/*
 * 親ディレクトリオブジェクトが持っていたサーバー数を保存しておき、
 * 後に nis_freeresult() を呼び出す前にこの値を元に戻して、
 * メモリーリークを避ける。
 */
saved_num_servers = dirobj-
>DI_data.do_servers.do_servers_len;

/* このディレクトリは 1 つのサーバーだけで管理する。 */
dirobj->DI_data.do_servers.do_servers_len = 1;

dir_create (dir_name, dirobj);

/* foo の下に groups_dir と org_dir という 2 つのディレクトリを作成する。 */
sprintf (grp_dir_name, "groups_dir.%s", dir_name);
dirobj->DI_data.do_name = grp_dir_name;
dir_create (grp_dir_name, dirobj);

sprintf (org_dir_name, "org_dir.%s", dir_name);

dirobj->DI_data.do_name = org_dir_name;
dir_create (org_dir_name, dirobj);
```

(続く)

```
 grp_create (grp_name);

 printf ("\nAdding principal %s to group %s ... \n",
    local_princip, grp_name);
 err = nis_addmember (local_princip, grp_name);

 if (err != NIS_SUCCESS) {
   nis_perror (err,
   "unable to add local principal to group.");
  exit (1);
 }

 sprintf (table_name, "test_table.org_dir.%s", dir_name);
 tbl_create (dirobj, table_name);

 /*
  * 作成したばかりのテーブルに、NIS+ エントリオブジェクトを作成する。
  */

 stuff_table (table_name);

 /* 作成したオブジェクトを表示する。 */
 list_objs(dir_name, table_name, grp_name);

 /* 作成したものすべてのクリーンアップ。 */
 cleanup (local_princip, grp_name, table_name, dir_name,
dirobj);

 /*
  * オリジナルの pres 構造体から保存しておいたポインタを元に戻し、
  * 関連メモリーを解放してもメモリーリークが起こらないようにする
  */
 dirobj->DI_data.do_name = saved_name;
 dirobj->zo_group = saved_grp;
 dirobj->zo_owner = saved_owner;
 dirobj->DI_data.do_servers.do_servers_len =
 saved_num_servers;
 (void) nis_freeresult (pres);
}
```

コード例 5–2 が示すルーチンは main() から呼び出され、ディレクトリオブジェクトを作成します。

コード例 **5–2**　ディレクトリオブジェクトを作成する NIS+ ルーチン

```
void
dir_create (dir_name, dirobj)
 nis_name   dir_name;
 nis_object   *dirobj;
{
 nis_result  *cres;
 nis_error   err;

 printf ("\n Adding Directory %s to namespace ... \n",
  dir_name);
 cres = nis_add (dir_name, dirobj);

 if (cres->status != NIS_SUCCESS) {
  nis_perror (cres->status, "unable to add directory foo.");
  exit (1);
 }

 (void) nis_freeresult (cres);

 /*
  * 注: nis_mkdir を実行して、作成するディレクトリの内容を保存する
  * テーブルを作成する必要がある。
  */

 err = nis_mkdir (dir_name,

      dirobj->DI_data.do_servers.do_servers_val);
 if (err != NIS_SUCCESS) {
  (void) nis_remove (dir_name, 0);

  nis_perror (err,
    "unable to create table for  directory object foo.");
  exit (1);
 }
}
```

次のルーチンは `main()` から呼び出され、グループオブジェクトを作成します。`nis_creategroup()` はグループオブジェクトだけを対象としているので、グループ名には `groups_dir` というリテラルは必要ありません。

コード例 **5–3**　グループオブジェクトを作成する NIS+ ルーチン

```
void
grp_create (grp_name)
 nis_name   grp_name;
{
 nis_error   err;

 printf ("\n Adding %s group to namespace ... \n", grp_name);
```

```
 err = nis_creategroup (grp_name, 0);
 if (err != NIS_SUCCESS) {
  nis_perror (err, "unable to create group.");
  exit (1);
 }
}
```

コード例 5–3 が示すルーチンは main() から呼び出され、テーブルオブジェクトを表 5–3 のようなレイアウトで作成します。

表 5–3 NIS+ テーブルオブジェクト

	カラム **1**	カラム **2**
ネーム	id	name
属性	検索可能、テキスト	検索可能、テキスト
アクセス権	----rmcdr---r---	----rmcdr---r---

定数 TA_SEARCHABLE を指定すると、サービスカラムを検索可能とします。TEXT カラム (デフォルト属性) だけが検索可能です。TA_CASE を指定すると、サービスは、カラムの値を検索するときに大文字小文字を区別しません。

コード例 5–4 テーブルオブジェクトを作成する NIS+ ルーチン

```
#define   TABLE_MAXCOLS 2
#define   TABLE_COLSEP ':'
#define   TABLE_PATH 0

void
tbl_create (dirobj, table_name)
 nis_object  *dirobj;     /* フィールドによっては必要 */
 nis_name   table_name;
{
 nis_result    *cres;
 static nis_object     tblobj;
 static table_col     tbl_cols[TABLE_MAXCOLS] = {
```

(続く)

```
  {"Id", TA_SEARCHABLE | TA_CASE, DEFAULT_RIGHTS},
  {"Name", TA_SEARCHABLE | TA_CASE, DEFAULT_RIGHTS}
 };

 tblobj.zo_owner = dirobj->zo_owner;
 tblobj.zo_group = dirobj->zo_group;
 tblobj.zo_access = DEFAULT_RIGHTS;    /* nis.h で定義されたマクロ */
 tblobj.zo_data.zo_type = TABLE_OBJ;   /* nis.h で定義された列挙型 */
 tblobj.TA_data.ta_type = TABLE_TYPE;
 tblobj.TA_data.ta_maxcol = TABLE_MAXCOLS;
 tblobj.TA_data.ta_sep = TABLE_COLSEP;
 tblobj.TA_data.ta_path = TABLE_PATH;
 tblobj.TA_data.ta_cols.ta_cols_len =
 tblobj.TA_data.ta_maxcol;          /* 常にこの指定 ! */
 tblobj.TA_data.ta_cols.ta_cols_val = tbl_cols;

/*
 * 完全指定のテーブル名、すなわち、org_dir というリテラルが埋め込まれた
 * テーブル名を使用する。nis_add はあらゆるタイプの NIS+ オブジェクトを
 * 操作するからであり、テーブルが作成済みならばそのフルパス名も必要である。
 */
 printf ("\n Creating table %s ... \n", table_name);
 cres = nis_add (table_name, &tblobj);

 if (cres->status != NIS_SUCCESS) {
  nis_perror (cres->status, "unable to add table.");
  exit (1);
 }
 (void) nis_freeresult (cres);
}
```

コード例 5–5 が示すルーチンは main() から呼び出され、テーブルオブジェクトにエントリオブジェクトを追加します。2 つのエントリがテーブルオブジェクトに追加されます。どちらのエントリのカラム幅も、文字列のターミネータとしての NULL 文字を含む値で設定することに注意してください。

コード例 5–5 テーブルにオブジェクトを追加する NIS+ ルーチン

```
#define   MAXENTRIES 2
void
stuff_table(table_name)
 nis_name  table_name;
{
 int   i;
 nis_object   entdata;
```

(続く)

```
nis_result   *cres;
static entry_col ent_col_data[MAXENTRIES][TABLE_MAXCOLS] = {
  {0, 2, "1", 0, 5, "John"},
  {0, 2, "2", 0, 5, "Mary"}
};

printf ("\n Adding entries to table ... \n");

/*
 * テーブルに追加するエントリは、テーブルと同じ所有者、グループ所有者、
 * アクセス権を持たなければならないので、はじめにテーブルオブジェクトを
 * 調べる。
 */
cres = nis_lookup (table_name, 0);

if (cres->status != NIS_SUCCESS) {

 nis_perror (cres->status, "Unable to lookup table");
 exit(1);
}
entdata.zo_owner = NIS_RES_OBJECT (cres)->zo_owner;
entdata.zo_group = NIS_RES_OBJECT (cres)->zo_group;
entdata.zo_access = NIS_RES_OBJECT (cres)->zo_access;

/* cres を解放して再利用できるようにする。 */
(void) nis_freeresult (cres);

entdata.zo_data.zo_type = ENTRY_OBJ; /* nis.h で定義された列挙型 */
entdata.EN_data.en_type = TABLE_TYPE;
entdata.EN_data.en_cols.en_cols_len = TABLE_MAXCOLS;
for (i = 0; i < MAXENTRIES; ++i) {
 entdata.EN_data.en_cols.en_cols_val = &ent_col_data[i][0];
 cres = nis_add_entry (table_name, &entdata, 0);


 if (cres->status != NIS_SUCCESS) {
  nis_perror (cres->status, "unable to add entry.");
  exit (1);
 }
 (void) nis_freeresult (cres);
}
}
```

コード例 5–6 が示すルーチンは nis_list() の呼び出しで使用する印刷関数です。list_objs() が nis_list() を呼び出すときに、引数の 1 つとして print_info() へのポインタを渡します。サービスが print_info() を呼び出す

たびに、エントリオブジェクトの内容が印刷されます。戻り値は、ライブラリに対して、テーブルの次のエントリを呼び出すように指示します。

コード例 **5–6**　nis_list() を呼び出す NIS+ ルーチン

```
int
print_info (name, entry, cbdata)
 nis_name   name;   /* 未使用 */
 nis_object   *entry;   /* NIS+ エントリオブジェクト */
 void   *cbdata;   /* 未使用 */
{
 static u_int    firsttime = 1;
 entry_col    *tmp;  /* ソースを読みやすくするためだけに使用 */
 u_int    i, terminal;

 if (firsttime) {
  printf ("\tId.\t\t\tName\n");
  printf ("\t---\t\t\t----\n");
  firsttime = 0;
 }
 for (i = 0; i < entry->EN_data.en_cols.en_cols_len; ++i) {
  tmp = &entry->EN_data.en_cols.en_cols_val[i];
  terminal = tmp->ec_value.ec_value_len;
  tmp->ec_value.ec_value_val[terminal] = '\0';

 }

 /*
  * ENTRY_VAL は、指定したエントリの特定カラムの値を返すマクロ
  */
 printf("\t%s\t\t\t%s\n", ENTRY_VAL (entry, 0),
        ENTRY_VAL (entry, 1));
 return (0); /* さらに呼出しを行う */
}
```

コード例 5–7 が示すルーチンは main() から呼び出され、グループ、テーブル、ディレクトリオブジェクトの内容をリストします。次のルーチンで、コールバックの使用方法を示します。グループのメンバーを取り出して表示します。グループのメンバーリストは、オブジェクトの中に保存されていないので、nis_list() ではなく、nis_lookup() で調べます。nis_lookup() は、グループだけでなくあらゆるタイプの NIS+ オブジェクトを扱うため、グループ名は groups_dir を付けた形式で指定しなければなりません。

コード例 **5–7**　オブジェクトをリストする NIS+ ルーチン

```
void
list_objs(dir_name, table_name, grp_name)
         nis_name   dir_name, table_name, grp_name;
{
 group_obj   *tmp;  /* ソースを読みやすくするためだけに使用 */
 u_int   i;
 char   grp_obj_name [NIS_MAXNAMELEN];
 nis_result   *cres;
 char   index_name [BUFFER_SIZE];

 sprintf (grp_obj_name, "%s.groups_dir.%s",
    nis_leaf_of (grp_name), nis_domain_of (grp_name));
 printf ("\nGroup %s membership is: \n", grp_name);

 cres = nis_lookup(grp_obj_name, 0);

 if (cres->status != NIS_SUCCESS) {
  nis_perror (cres->status, "Unable to lookup group object.");

  exit(1);
 }

 tmp = &(NIS_RES_OBJECT(cres)->GR_data);
 for (i = 0; i < tmp->gr_members.gr_members_len; ++i)
  printf ("\t %s\n", tmp->gr_members.gr_members_val[i]);
 (void) nis_freeresult (cres);

 /*
  * ドメイン foo の内容を、コールバックを使用しないで表示する。
  */
 printf ("\nContents of Directory %s are: \n", dir_name);
 cres = nis_list (dir_name, 0, 0, 0);
 if (cres->status != NIS_SUCCESS) {
  nis_perror (cres->status,"Unable to list Contents of Directory foo.");
  exit(1);
 }
 for (i = 0; i < NIS_RES_NUMOBJ(cres); ++i)
  printf("\t%s\n", NIS_RES_OBJECT(cres)[i].zo_name);

 (void) nis_freeresult (cres);

 /*
  * 作成したテーブルの内容を、nis_list()でコールバックを使用してリスト
   * する。
  */
 printf ("\n Contents of Table %s are: \n", table_name);
 cres = nis_list (table_name, 0, print_info, 0);
 if(cres->status != NIS_CBRESULTS && cres->status !=
NIS_NOTFOUND){
  nis_perror (cres->status,
   "Listing entries using callback failed");
  exit(1);
 }
 (void) nis_freeresult (cres);
```

(続く)

続き

```
 /*
  * 作成したテーブルの 1 エントリだけをリストする。エントリの取り出しには、
  * インデックスの付いた名前を使用する。
  */

 printf("\n Entry corresponding to id 1 is:\n");

 /*
  * 通常、カラム名はテーブルオブジェクトから抽出できるので、
  * 始めにそれを取り出す。
  */
 sprintf(index_name, "[Id=1],%s", table_name);
 cres = nis_list (index_name, 0, print_info, 0);
 if(cres->status != NIS_CBRESULTS && cres->status !=
NIS_NOTFOUND){
  nis_perror (cres->status,
  "Listing entry using indexed names and callback failed");
  exit(1);
 }
 (void) nis_freeresult (cres);
}
```

コード例 5–8 のルーチンは cleanup() から呼び出され、名前空間からディレクトリオブジェクトを削除します。また、このディレクトリを管理しているサーバーにも、名前空間から削除したことを通知します。ポインタ cres が指している戻り値の構造体が入ったメモリーは、戻り値が使用済みになってから解放しなければならないことに注意してください。

コード例 **5–8**　ディレクトリオブジェクトを削除する NIS+ ルーチン

```
void
dir_remove(dir_name, srv_list, numservers)
 nis_name   dir_name;
 nis_server   *srv_list;
 u_int   numservers;
{
 nis_result   *cres;
 nis_error   err;
 u_int   i;

 printf ("\nRemoving %s directory object from namespace ...
 \n",dir_name);
```

(続く)

```
 cres = nis_remove (dir_name, 0);
 if (cres->status != NIS_SUCCESS) {
  nis_perror (cres->status, "unable to remove directory");
  exit (1);
 }
 (void) nis_freeresult (cres);

 for (i = 0; i < numservers; ++i) {
  err = nis_rmdir (dir_name, &srv_list[i]);

  if (err != NIS_SUCCESS) {
   nis_perror (err,
   "unable to remove server from directory");
   exit (1);
  }
 }
}
```

コード例 5–9 のルーチンは main() から呼び出され、このサンプルプログラムで作成したオブジェクトをすべて削除します。nis_remove_entry() を呼び出すときのフラグ REM_MULTIPLE の使用法に注意してください。テーブルを削除するときは、その前にテーブル内の全エントリを削除しておく必要があります。

コード例 **5–9**　オブジェクトをすべて削除する NIS+ ルーチン

```
void
cleanup(local_princip, grp_name, table_name, dir_name, dirobj)
 nis_name   local_princip, grp_name, table_name, dir_name;
 nis_object   *dirobj;
{
 char  grp_dir_name [NIS_MAXNAMELEN];
 char  org_dir_name [NIS_MAXNAMELEN];
 nis_error   err;
 nis_result   *cres;

 sprintf(grp_dir_name, "%s.%s", "groups_dir", dir_name);
 sprintf(org_dir_name, "%s.%s", "org_dir", dir_name);

 printf("\n\n\nStarting to Clean up ... \n");
 printf("\n\nRemoving principal %s from group %s \n",
      local_princip, grp_name);
 err = nis_removemember (local_princip, grp_name);
```

(続く)

```
 if (err != NIS_SUCCESS) {
  nis_perror (err,
  "unable to delete local principal from group.");
  exit (1);
 }

 /*
  * admins グループを削除する。グループ削除の API は、グループだけを
  * 対象にしているので、グループ名に group_dir は不要。
  * グループ名には、自動的にリテラル group_dir が埋め込まれる。
  */
 printf("\nRemoving %s group from namespace ... \n",
 grp_name);
 err = nis_destroygroup (grp_name);
 if (err != NIS_SUCCESS) {
  nis_perror (err, "unable to delete group.");
  exit (1);
 }

 printf("\n Deleting all entries from table %s ... \n",
table_name);

 cres = nis_remove_entry(table_name, 0, REM_MULTIPLE);
 switch (cres->status) {
  case NIS_SUCCESS:
  case NIS_NOTFOUND:
   break;
  default:
   nis_perror(cres->status, "Could not delete entries from
          table");
   exit(1);
 }
 (void) nis_freeresult (cres);

 printf("\n Deleting table %s itself ... \n", table_name);
 cres = nis_remove(table_name, 0);

 if (cres->status != NIS_SUCCESS) {
  nis_perror(cres->status, "Could not delete table.");
 exit(1);
 }
 (void) nis_freeresult (cres);

 /* groups_dir、org_dir、foo の各ディレクトリオブジェクトを削除する。*/

 dir_remove (grp_dir_name,
      dirobj->DI_data.do_servers.do_servers_val,
      dirobj->DI_data.do_servers.do_servers_len);
 dir_remove (org_dir_name,
      dirobj->DI_data.do_servers.do_servers_val,
      dirobj->DI_data.do_servers.do_servers_len);
 dir_remove (dir_name,
```

(続く)

```
      dirobj->DI_data.do_servers.do_servers_val,
      dirobj->DI_data.do_servers.do_servers_len);
}
```

このプログラムを実行すると、画面は 図 5–3 のように表示されます。

```
myhost% domainname  sun.com
 myhost% ./sample  Adding Directory foo.sun.com. to namespace ...
 Adding Directory groups_dir.foo.sun.com. to namespace ...
 Adding Directory org_dir.foo.sun.com. to namespace ...
 Adding admins.foo.sun.com. group to namespace ...
 Adding principal myhost.sun.com. to group admins.foo.sun.com. ...
 Creating table test_table.org_dir.foo.sun.com. ...
 Adding entries to table ...
 Group admins.foo.sun.com. membership is:
         myhost.sun.com.
 Contents of Directory foo.sun.com. are:
         groups_dir
         org_dir
 Contents of Table test_table.org_dir.foo.sun.com. are:
         Id.                     Name
         ---                     ----
         1                       John
         2                       Mary

Entry corresponding to id 1 is:
         1                       John

Starting to Clean up ...

Removing principal myhost.sun.com. from group admins.foo.sun.com.
Removing admins.foo.sun.com. group from namespace ...
Deleting all entries from table test_table.org_dir.foo.sun.com. ...
Deleting table test_table.org_dir.foo.sun.com. itself ...
Removing groups_dir.foo.sun.com. directory object from namespace ...
Removing org_dir.foo.sun.com. directory object from namespace ...
Removing foo.sun.com. directory object from namespace ...
myhost%
```

図 5–3　NIS+ プログラムの実行

デバッグのために、次の一連のコマンドを入力して、これと同じ操作を実行することもできます。最初のコマンド

```
% niscat -o domainname .
```

を実行すると、マスターサーバー名が表示されます。以下に示すコマンドを入力するときは、master のところを、表示されたマスターサーバー名に置き換えて入力します。

```
% nismkdir -m master foo.domainname.
```

```
# 指定されたマスターでサブディレクトリ org_dir.foo を作成します。
% nismkdir -m master org_dir.foo.domainname.
# 指定されたマスターでサブディレクトリ groups_dir.foo を作成します。
% nismkdir -m master groups_dir.foo.domainname.
# グループ admins を作成します。
% nisgrpadm -c admins.foo.domainname.
```

```
# 自分自身をこのグループのメンバーとして追加します。
% nisgrpadm -a admins.foo.domainname. `nisdefaults -p`
```

```
# Id と Name の 2 つのカラムで test_table を作成します。
% nistbladm -c test_data id=SI Name=SI \ test_table.org_dir.foo.domainname.
```

```
# そのテーブルにエントリを 1 つ追加します。
% nistbladm -a id=1 Name=John test_table.org_dir.foo.domainname.
# そのテーブルに別のエントリを追加します。
% nistbladm -a id=2 Name=Mary test_table.org_dir.foo.domainname.
```

```
# グループ admins のメンバーをリストする。
% nisgrpadm -l admins.foo.domainname.
# ディレクトリ foo の内容をリストする。
% nisls foo.domainname.
# test_table の内容とヘッダーをリストする。
% niscat -h test_table.org_dir.foo.domainname.
```

```
# test_table から id が 1 のエントリを取り出す。
% nismatch id=1 test_table.org_dir.foo.domainname.
```

```
# 作成したものをすべて削除する。
# 初めに、グループ admins から自分自身を削除する。
% nisgrpadm -r admins.foo.domainname. `nisdefaults -p`
# グループ admins を削除する。
% nisgrpadm -d admins.foo.domainname.
# test_table からすべてのエントリを削除する。
% nistbladm -r ``[],test_table.org_dir.foo.domainname.''
# test_table を削除する。
% nistbladm -d test_table.org_dir.foo.domainname.
# 作成した 3 つのディレクトリをすべて削除する。

% nisrmdir groups_dir.foo.domainname.
% nisrmdir org_dir.foo.domainname.
% nisrmdir foo.domainname.
```

付録 **A**

XDR テクニカルノート

付録 A は、サン・ソフトが XDR (External Data Representation: 外部データ表現) 標準規約に準拠して開発したライブラリルーチンセットについてのテクニカルノートです。XDR ライブラリルーチンを使用すると、C プログラマは任意のデータ構造をマシンに依存しない形式で記述できます。

XDR の概要

遠隔手続き呼び出しのときにデータは XDR 標準形式で伝送されますので、XDR は米国サンソフトの RPC パッケージの重要な基本概念となっています。複数の異なるタイプのマシンからアクセスされる (読み書きされる) データの転送には、XDR ライブラリルーチンを使用する必要があります。

XDR は異なる言語、異なるオペレーティングシステム、異なるマシンアーキテクチャの間で機能します。ほとんどのユーザー (特に RPC ユーザー) は、「整数フィルタ」、「浮動小数点フィルタ」、「列挙型フィルタ」の節を読むだけで十分でしょう。RPC と XDR を別のマシン上に実装したいプログラマは、このテクニカルノートやプロトコルの仕様に関心を持たれるかも知れません。

RPC 呼び出しを行わない場合でも、`rpcgen` を使用して XDR ルーチンを書くことができます。

XDR ルーチンを使用する C プログラムはファイル `<rpc/xdr.h>` をインクルードしなければなりません。`<rpc/xdr.h>` には、XDR システムとの必要なインタフェースがすべて入っています。XDR ルーチンはすべてライブラリ `libnsl.a` に入っていますので、コンパイルは次のコマンドで実行します。

```
example% cc program.c
```

移植性を保つためには、各環境でさまざまな基準を守ってプログラミングしなければなりません。プログラミング方法の微妙な違いを見つけるのも必ずしも簡単とはいえませんが、それらが広範囲に渡って関連し合っています。以下に示すサンプルプログラム コード例 A–1 と コード例 A–2 (テキスト行の読み込みと書き出しのプログラム) で考えてみましょう。

コード例 **A–1**　writer サンプルプログラム (初期状態)

```
#include <stdio.h>

main()          /* writer.c */
{  int i;

for (i = 0; i < 8; i++) {
   if (fwrite((char *) &i, sizeof(i), 1, stdout) != 1) {
    fprintf(stderr, "failed!\n");
    exit(1);
   }
  }
  exit(0);
}
```

コード例 **A–2**　reader サンプルプログラム (初期状態)

```
#include <stdio.h>

main()       /* reader.c */
{
  int i, j;

 for (j = 0; j < 8; j++) {
   if (fread((char *) &i, sizeof(i), 1, stdin) != 1) {
    fprintf(stderr, "failed!\n");
    exit(1);
   }
   printf("%ld ", i);
  }
  printf("\n");
  exit(0);
}
```

この 2 つのプログラムは移植可能に見えます。なぜならば、(a) lint チェックをパスし、(b) 2 つの異なるハードウェアアーキテクチャ、SPARC と VAX 上で実行しても同じように動作するからです。

writer プログラムの出力を reader プログラムにパイプした場合も、SPARC と VAX の両方で同じ結果が得られます。

```
sun% writer | reader
0 1 2 3 4 5 6 7
sun%
vax% writer | reader
0 1 2 3 4 5 6 7
vax%
```

ローカルエリアネットワークと 4.2BSD の出現に伴って、「ネットワークパイプ」の概念が導入されました。すなわち、あるマシン上のプロセスで生成したデータを、別のマシン上のプロセスが使用するという概念です。writer と reader のプログラムでも、ネットワークパイプを使用できます。最初に SPARC 上でデータを生成し、そのデータを VAX 上で使用したときの実行結果を次に示します。

```
sun% writer | rsh vax reader
0 16777216 33554432 50331648 67108864 83886080 100663296
117440512
sun%
```

writer を VAX で実行し、reader を SPARC で実行した場合も同じ結果になります。この原因は、VAX と SPARC の int 型整数は、ワードサイズは同じでもデータのバイト順序が異なるためです。16777216 は 2^{24} です。4 バイトを逆順にすると、24 番目のビットに 1 が置かれることに注意してください。

複数の異なるタイプのマシンでデータを共有するときは、データの可搬性についての注意が必要です。read() と write() 呼び出しを XDR ライブラリルーチンの呼び出しで xdr_int() に置き換えることにより、データが可搬なプログラムになります。xdr_int() は、int 型の整数の外部用の標準的な表現を扱うことができるフィルタです。writerプログラムの改良版をコード例 A–3 に示します。

コード例 **A–3**　writer サンプルプログラム (XDR 修正バージョン)

```
#include <stdio.h>
#include <rpc/rpc.h> /* XDR は RPC のサブライブラリ */

main()       /* writer.c */
{
  XDR xdrs;
  int i;

xdrstdio_create(&xdrs, stdout, XDR_ENCODE);
  for (i = 0; i < 8; i++) {
   if (!xdr_int(&xdrs, &i)) {
    fprintf(stderr, "failed!\n");
    exit(1);
   }
  }
  exit(0);
}
```

コード例 A–4 のサンプルプログラムは、reader プログラムを修正したものです。

コード例 **A–4**　reader サンプルプログラム (XDR 修正バージョン)

```
#include <stdio.h>
#include <rpc/rpc.h> /* xdr は rpc のサブライブラリ */

main()                /* reader.c */
{
  XDR xdrs;
  int i, j;

xdrstdio_create(&xdrs, stdin, XDR_DECODE);
  for (j = 0; j < 8; j++) {
   if (!xdr_int(&xdrs, &i)) {
    fprintf(stderr, "failed!\n");
    exit(1);
   }
   printf("%ld ", i);
  }
  printf("\n");
  exit(0);
}
```

変更したプログラムを SPARC 上で実行した結果、VAX 上で実行した結果、および SPARC から VAX にパイプした結果を次に示します。

```
sun% writer | reader
0 1 2 3 4 5 6 7
sun%
vax% writer | reader
0 1 2 3 4 5 6 7
vax%
sun%  writer | rsh vax reader
0 1 2 3 4 5 6 7
sun%
```

注 - 整数データの移植問題は、データの移植問題のごく簡単な一例にすぎません。どのデータ構造体にも移植性の問題があり、特にデータ境界とポインタが大きな問題になります。ワード境界の違いにより、マシンごとに構造体のサイズが異なる可能性があります。ポインタは使用するには便利ですが、ポインタが定義されたマシンの外部では何の意味も持ちません。

データの標準形式

XDR のアプローチは、データを 1 つの標準形式に統一するというものです。すなわち、XDR では、1 つのバイト順序、1 つの浮動小数点形式 (IEEE)、というように標準形式が定義されています。どのマシンで実行するプログラムでも XDR を使用すれば、データがローカルデータ形式から XDR 標準データ形式に変換されるので、移植可能なデータを作成できます。同様に、どのマシンで実行するプログラムでも XDR を使用すれば、データが XDR 標準データ形式からローカルデータ形式に変換されるので、移植可能なデータを読み込むことができます。1 つの標準形式を使用することにより、移植可能データを作成し送信するプログラムと、移植可能データを受信し使用するプログラムとが完全に切り離されます。新たなマシンや新たな言語が加わっても、移植データを作成し使用する既存のプログラムグループには何の影響もありません。新たなマシンは、標準データ形式と自身のローカルデータ形式との変換方法を「学ぶ」ことにより、既存のプログラムグループとの通信が可能になります。新たなマシンでは、他のマシンのローカルデータ形式を学ぶ必要がありません。これと同様に、他のマシン上で実行中の既存のプログラムグループでも、新たなマシンのローカルデータ形式を学ぶ必要がありません。新たなマシンが生成

した移植可能なデータは、既存プログラムが既に理解している標準データ形式に従っているので、そのまま読み込むことができるからです。

XDR の標準化アプローチには、さまざまな先例があります。たとえば、TCP/IP、UDP/IP、XNS、Ethernet、および、ISO の OSI 参照モデルの第 5 層より下のプロトコルはすべて標準プロトコルです。標準化アプローチの利点は単純であることです。XDR の場合も、一連の変換ルーチンを一度書いてしまえばそれをずっと使用できます。標準化アプローチには欠点もありますが、実際に使用するデータ伝送アプリケーションでは大きな問題ではありません。たとえば、バイト順序がリトルエンディアンの 2 つのマシン間で XDR 標準を使用して整数データを伝送するとします。送信側のマシンでは、整数データのバイト順をリトルエンディアンからビッグエンディアン(XDR のバイト順序) に変換し、受信側のマシンではその反対の変換を行います。この 2 つのマシンのバイト順序は同じなので、本来このような変換は不要ですが、重要なのは必要かどうかよりも、別の方法と比較したときのコストです。

標準データ形式とローカルデータ形式との変換に要する時間は重要ではありません。特に、分散型アプリケーションの場合は大した問題にはなりません。データ構造体を伝送する準備に要する時間のほとんどは、データ変換ではなく、データ構造体の各要素を取り出すのにかかります。たとえば、ツリー構造を伝送するには、葉の部分をすべてたどって、リーフレコード内の各要素をバッファにコピーして境界を合わせます。葉の部分を格納した記憶領域はその後解放しなければなりません。同様にツリー構造を受信するには、それぞれの葉の部分に対して記憶領域を割り当て、データをバッファからその記憶領域に移動して正しく境界を合わせ、葉と葉をリンクするポインタを設定します。どのマシンでも、標準データ形式との変換のあるなしに関係なく、データ構造体の走査とコピーのコストがかかります。分散型アプリケーションではこのような通信オーバヘッド、すなわち、送信側のプロトコル層を下ってネットワークを通り受信側のプロトコル層を上るのに時間がかかるため、標準データ形式とのデータ変換のオーバヘッドは相対的に小さくなります。

XDR ライブラリ

XDR ライブラリを使用すると、移植の問題が解決するばかりでなく、C の任意のデータ構造を一貫した、文書化された明確な方法で入出力することができます。したがって、ネットワーク上の複数のマシンでデータを共有する場合以外でもXDR ライブラリを使用する意味があります。

XDR ライブラリには、いくつか例を挙げてみるだけでも、文字列 (NULL で終わるバイト配列)、構造体、共用体、配列に対するフィルタルーチンがあります。ユー

ザーはさまざまなより基本的なルーチンを使用して、独自のXDR ルーチンを作成し、任意のデータ構造体 (配列の要素、共用体のアーム、他の構造体からポイントされるオブジェクト) を記述できます。構造体自体にも、任意の要素の配列や他の構造体へのポインタを持たせることができます。

2 つのサンプルプログラムをより詳しく見てください。XDR ストリーム作成ルーチンファミリがあります。ファミリの各メンバーはビットストリームの扱いが異なります。この例では、標準入出力ルーチンを使用してデータを操作しているので、xdrstdio_create() を使用します。XDR ストリーム作成ルーチンに渡す引数は、ルーチンの機能によって異なります。サンプルプログラムでは、初期化する XDR 構造体へのポインタ、入出力を行う FILE へのポインタ、処理内容の 3 つを引数として渡しています。処理内容は、writer プログラムではシリアライズするので XDR_ENCODE、reader プログラムではデシリアライズするので XDR_DECODE を指定します。

注 - RPC ユーザーは XDR ストリームを作成する必要はありません。RPC システムでストリームが作成され、ユーザーに引き渡されます。

---

xdr_int() プリミティブは、ほとんどの XDR ライブラリプリミティブと、すべてのクライアント作成の XDR ルーチンに共通の特徴を持っています。第 1 に、このルーチンはエラーが起こると FALSE(0) を返し、正常終了すると TRUE(1) を返します。第 2 に、各データ型 xxx ごとに対応する XDR ルーチンがあります。XDR ルーチンの形式を次に示します。

```
xdr_xxx(xdrs, xp)
   XDR *xdrs;
   xxx *xp;
{
}
```

このサンプルプログラムの場合は、xxx は int ですので、対応する XDR プリミティブは xdr_int() になります。クライアントも同様にして、任意の構造体 xxxを定義し、対応するルーチン xdr_int() を作成して、個々のフィールドごとにデータ型に一致する XDR ルーチンを呼び出します。どのデータ型の場合も、最初のパラメータ xdrs は隠されたハンドルとして、そのままプリミティブに渡します。

XDR ルーチンは両方向の変換を行います。すなわち、データをシリアライズするときもデシリアライズするときも同じルーチンを呼び出します。この機能は、移植可能データのソフトウェアエンジニアリングには不可欠な機能です。どちらの変換操

作にも同一ルーチンを呼び出すという概念は、シリアライズしたデータはデシリアライズが可能であることをほぼ保証するものです。ネットワークデータを生成するときも使用するときも同じルーチンが使用できます。これは、ルーチンには必ず、オブジェクト自体ではなくオブジェクトへのポインタを渡すことにより実現されます。デシリアライズの場合だけは、オブジェクト自体が変更されます。この機能は、簡単なサンプルプログラムではわかりませんが、より複雑なデータ構造体をマシン間で引き渡すときに、この機能の価値が明らかになります。必要な場合は、XDR の処理内容を取り出すこともできます。詳細は、228ページの「処理内容」の節を参照してください。

もう少し複雑な例を以下に示します。ある人の総資産と負債のデータをプロセス間で交換するとします。このデータは、次のような独自のデータ型が必要なほど重要であるとします。

```
struct gnumbers {
   int g_assets;
   int g_liabilities;
};
```

このデータ型に対応する XDR ルーチンでは、この構造体を次のように記述します。

```
bool_t                        /* 正常終了では TRUE、異常終了では FALSE */
xdr_gnumbers(xdrs, gp)
   XDR *xdrs;
   struct gnumbers *gp;
{
   if (xdr_int(xdrs, &gp->g_assets) &&
         xdr_int(xdrs, &gp->g_liabilities))
      return(TRUE);
   return(FALSE);
}
```

引数 xdrs は値を調べたり変更したりすることなく、そのままサブコンポーネントルーチンに渡していることに注意してください。XDR ルーチンを呼び出すたびにその戻り値を調べ、異常終了している場合はすぐに FALSE を返してリターンしなければなりません。

この例では、bool_t 型が TRUE (1) と FALSE (0) のどちらかの値だけをとる整数として宣言されていることがわかります。このマニュアルでは次の定義を使用します。

```
#define bool_t int
#define TRUE 1
#define FALSE 0
```

これらの取り決めを使用すると、xdr_gnumbers() は次のように書き換えられます。

```
xdr_gnumbers(xdrs, gp)
   XDR *xdrs;
   struct gnumbers *gp;
{
   return(xdr_int(xdrs, &gp->g_assets) &&
            xdr_int(xdrs, &gp->g_liabilities));
}
```

このマニュアルでは、両方のコーディング形式を使用します。

XDR ライブラリのプリミティブ

この節では、XDR プリミティブの概要を述べます。始めにメモリー割り当てと基本データ型を説明し、次に合成データ型を説明します。最後に、XDR ユーティリティについて説明します。XDR のプリミティブとユーティリティのインタフェースはインクルードファイルで定義されています。<rpc/xdr.h> は <rpc/rpc.h> から自動的にインクルードされます。

XDR ルーチンで必要なメモリー

XDR ルーチンを使用するときは、前もってメモリーを割り当てておかなければならない (または、必要なメモリーサイズを決定しておかなければならない) 場合があります。XDR 変換ルーチンで使用するメモリーの割り当てや割り当て解除が必要なときのために、xdr_sizeof() というルーチンが提供されています。xdr_sizeof() は、XDR フィルタルーチン (func()) がデータ (data()) の符号化や復号化で使用するバイト数を返します。xdr_sizeof() が返す値には、RPC ヘッダーやレコードマークは含まれていませんので、必要なメモリーサイズを正確に求めるには、そ

れらのバイト数も加えなければなりません。エラーが起こった場合、xdr_sizeof() はゼロを返します。

```
xdr_sizeof(xdrproc_t func, void *data)
```

xdr_sizeof() は、RPC 環境以外で XDR を使用するアプリケーションでメモリーを割り当てるとき、トランスポートプロトコルを選択するとき、下位レベルの RPC を使用してクライアント作成関数やサーバー作成関数を実行するときに特に便利です。

コード例 A–5 と コード例 A–6で、xdr_sizeof()の 2 通りの使用方法を説明します。

コード例 **A–5** xdr_sizeof() の使用例 1

```
#include <rpc/rpc.h>

/*
 * この関数への入力引数は、CLIENT ハンドル、XDR 関数、XDR 変換を行う
 * データへのポインタです。XDR 変換を行うデータが、CLIENT ハンドルに
 * 結合しているトランスポートで送信可能な大きさの場合は TRUE、
 * 大き過ぎて送信不可能な場合は FALSE を返します。
 */
bool_t
cansend(cl, xdrfunc, xdrdata)
 CLIENT *cl;
 xdrproc_t xdrfunc;
 void   *xdrdata;
{
 int fd;
 struct t_info tinfo;

 if (clnt_control(cl, CLGET_FD, &fd) == -1) {
  /* ハンドルの clnt_control() エラー */
  return (FALSE);
 }

 if (t_getinfo(fd, &tinfo) == -1) {
  /* ハンドルの t_getinfo() エラー */
  return (FALSE);
 } else {
  if (tinfo.servtype == T_CLTS) {
   /*
    * 現在使用しているのは非接続型トランスポートです。
    * xdr_sizeof() を使用して、メモリー要求がこのトランスポートでは
    * 大き過ぎないか調べます。
    */
   switch(tinfo.tsdu) {
    case 0:                         /* TSDU の概念なし */
    case -2:                          /* 通常データの送信不可能 */
     return (FALSE);
```

(続く)

```
     break;
    case -1:                          /* TSDU サイズの制限なし */
     return (TRUE);
     break;
    default:
     if (tinfo.tsdu < xdr_sizeof(xdrfunc, xdrdata))
      return (FALSE);
     else
      return (TRUE);
   }
  } else
   return (TRUE);
 }
}
```

コード例 A–6 は、xdr_sizeof() の使用例 2 になります。

コード例 A–6 xdr_sizeof() の使用例 2

```
#include <sys/statvfs.h>
#include <sys/sysmacros.h>

/*
 * この関数への入力引数は、ファイル名、XDR 関数、XDR 変換を行うデータへの
 * ポインタです。この関数は、ファイルが置かれているファイルシステムに、
 * データを XDR 変換するのに必要な空間が残っていれば TRUE を返します。
 * ファイルシステムに関して statvfs(2) で得られる情報はブロック数単位
 * なので、xdr_sizeof() の戻り値もバイト数からディスクブロック数に
 * 変換しなければならないことに注意してください。
 */
bool_t
canwrite(file, xdrfunc, xdrdata)
 char      *file;
 xdrproc_t xdrfunc;
 void     *xdrdata;
{
 struct statvfs s;

 if (statvfs(file, &s) == -1) {
  /* ハンドルの statvfs() エラー */
  return (FALSE);
 }

 if (s.f_bavail >= btod(xdr_sizeof(xdrfunc, xdrdata)))
  return (TRUE);
 else
  return (FALSE);
```

(続く)

```
}
```

整数フィルタ

XDR ライブラリでは、整数を対応する外部表現に変換するプリミティブが提供されます。プリミティブが変換の対象とする整数は、次の組み合わせで表されます。

```
[signed, unsigned] * [short, int, long]
```

具体的には次の 8 つのプリミティブが提供されています。

```
bool_t xdr_char(xdrs, op)
   XDR *xdrs;
   char *cp;
bool_t xdr_u_char(xdrs, ucp)
   XDR *xdrs;
   unsigned char *ucp;
bool_t xdr_int(xdrs, ip)
   XDR *xdrs;
   int *ip;
bool_t xdr_u_int(xdrs, up)
   XDR *xdrs;
   unsigned *up;
bool_t xdr_long(xdrs, lip)
   XDR *xdrs;
   long *lip;
bool_t xdr_u_long(xdrs, lup)
   XDR *xdrs;
   u_long *lup;
bool_t xdr_short(xdrs, sip)
   XDR *xdrs;
   short *sip;
bool_t xdr_u_short(xdrs, sup)
   XDR *xdrs;
   u_short *sup;
```

最初の引数 xdrs は、XDR のストリームハンドルです。第 2 引数は、ストリームへ渡すデータのアドレス、または、ストリームからデータを受け取るアドレスです。どのルーチンも、変換に成功すれば TRUE、失敗すれば FALSE を返します。

浮動小数点フィルタ

XDR ライブラリでは、C の浮動小数点型データのプリミティブも提供されます。

```
bool_t xdr_float(xdrs, fp)
   XDR *xdrs;
   float *fp;
bool_t xdr_double(xdrs, dp)
   XDR *xdrs;
   double *dp;
```

最初の引数 xdrs は、XDR のストリームハンドルです。第 2 引数は、ストリームへ渡す浮動小数点データのアドレスまたはストリームから浮動小数点データを受け取るアドレスです。どちらのルーチンも、変換に成功すれば TRUE、失敗すれば FALSE を返します。

注 - 数値の表現形式は、浮動小数点に関する IEEE 標準規約に従っているため、IEEE 準拠の表現形式からマシン固有の表現形式に復号化したり、その反対方向に変換したりすると、エラーが起こる場合があります。

列挙型フィルタ

XDR ライブラリでは、一般の列挙型に対するプリミティブを提供しています。このプリミティブでは、C の enum 型のマシン内部表現が C の整数と同じであるとみなしています。ブール型は enum 型の重要な一例です。ブール値の外部表現は常に TRUE (1) と FALSE (0) です。

```
#define bool_t int
#define FALSE  0
#define TRUE   1
#define enum_t int
bool_t xdr_enum(xdrs, ep)
   XDR *xdrs;
   enum_t *ep;
bool_t xdr_bool(xdrs, bp)
   XDR *xdrs;
   bool_t *bp;
```

第 2 引数 ep と bp は、ストリーム xdrs へ渡すデータのアドレス、または、ストリーム xdrs からデータを受け取るアドレスです。

データなしルーチン

ときには、データが一切渡されず要求されていなくても、XDR ルーチンを RPC システムに提供しなければならない場合があります。ライブラリでは、そのためのルーチンを提供しています。

```
bool_t xdr_void(); /* 常に TRUE を返す */
```

合成データ型フィルタ

合成データ型、または、複合データ型を変換するプリミティブは、これまでに説明したプリミティブより多くの引数を必要とし、より複雑な機能を実行します。この節では、文字列、配列、共用体、構造体へのポインタに対するプリミティブを説明します。

合成データ型のプリミティブでは、メモリー管理を使用する場合があります。多くの場合、XDR_DECODEを指定してデータをデシリアライズすると、メモリーが割り当てられます。そのため、XDR パッケージではメモリー割り当てを解除する方法を提供しなければなりません。メモリー割り当ての解除は、XDR_FREE という XDR 処理で行います。XDRの 処理内容には、XDR_ENCODE、XDR_DECODE、XDR_FREE の 3 つがあります。

文字列

C 言語では、文字列が、NULL コードで終わるバイトシーケンスと定義されています。NULL コードは、文字列の長さを求めるときは計算に入れません。ところが、文字列を引き渡したり操作したりするときは、文字へのポインタが使用されます。そのため、XDRライブラリでは文字列を文字シーケンスではなく、char * と定義しています。文字列の外部表現は、内部表現とは大きく異なります。

文字列は内部では文字へのポインタで表現されますが、外部では ASCII 文字シーケンスで表現されます。この 2 つの表現形式の間の変換は、ルーチン xdr_string() で実行します。

```
bool_t xdr_string(xdrs, sp, maxlength)
   XDR *xdrs;
   char **sp;
```

```
    u_int maxlength;
```

最初の引数 xdrs は、XDR のストリームハンドルです。第 2 引数 sp は文字列へのポインタ (データ型は char **) です。第 3 引数 maxlength は、符号化または復号化の対象とする最大バイト数です。通常、この値はプロトコルで決まります。たとえば、あるプロトコル仕様では、ファイル名は最大 255 文字までとされています。文字数が maxlength の値を超えていれば FALSE、超えていなければ TRUE が返されます。

xdr_string() の機能は、この節でこれまでに説明した他の変換ルーチンと同様です。処理内容が XDR_ENCODE の場合は最も簡単です。引数 sp はある長さの文字列を指しています。この文字列の長さが maxlength を超えていなければ、この文字列がシリアライズされます。

デシリアライズの場合はもう少し複雑です。最初に、ストリームから取り込む文字列の長さを決定します。文字列の長さは maxlength を超えてはいけません。次に sp をデレファレンスします。その値が NULL の場合は、適切なサイズの文字列を割り当てて、*sp がその文字列を指すように設定します。*sp の元々の値が NULL でない場合は、デシリアライズしたデータを入れるターゲットエリアが既に割り当てられており、maxlength 以下の長さの文字列をそこに格納できるものとみなします。どちらの場合も、文字列が復号化されてターゲットエリアに保存されます。次に、文字列の最後に NULL コードが付加されます。

XDR_FREE 処理の場合は、sp をデレファレンスして文字列を取り出します。その文字列が NULL 文字列でなければ、領域を解放して *sp を NULL に設定します。この処理を実行するときは、xdr_string() は引数 maxlength を無視します。

空の文字列 ("") を XDR 変換することはできますが、NULL 文字列を XDR 変換はできないことに注意してください。

バイト配列

文字列よりも可変長バイト配列を使用する方が便利な場合があります。バイト配列は、次の 3 つの点で文字列と異なっています。(1) 配列の長さ(バイトカウント) を符号なし整数として明示的に保持している。(2) バイトシーケンスが NULL コードで終

了しない。(3) データの外部表現と内部表現が一致する。バイト配列の内部表現と外部表現との変換には、プリミティブ xdr_bytes() を使用します。

```
bool_t xdr_bytes(xdrs, bpp, lp, maxlength)
   XDR *xdrs;
   char **bpp;
   u_int *lp;
   u_int maxlength;
```

このルーチンの第 1、第 2、第 4 引数はそれぞれ、xdr_string() の第 1、第 2、第 3 引数と同じです。シリアライズの場合は、lp をデレファレンスしてバイトシーケンスの長さを得ます。デシリアライズの場合は、*lp にバイトシーケンスの長さが設定されます。

配列

XDR ライブラリパッケージでは、任意の要素で構成される配列を処理するプリミティブが提供されています。xdr_bytes() ルーチンは一般の配列のサブセットを処理します。すなわち、xdr_bytes() ルーチンでは、配列要素のサイズは 1 に決まっており、各要素の外部記述も組み込まれています。一般の配列に対するプリミティブ xdr_array() の引数は、xdr_bytes() の引数より 2 つ多く、配列要素のサイズと、各要素を変換する XDR ルーチンとが渡されます。渡されたXDR ルーチンは、配列の各要素の符号化または復号化のときに呼び出されます。

```
bool_t
xdr_array(xdrs, ap, lp, maxlength, elementsize, xdr_element)
   XDR *xdrs;
   char **ap;
   u_int *lp;
   u_int maxlength;
   u_int elementsize;
   bool_t (*xdr_element)();
```

引数 ap は、配列へのポインタのアドレスです。配列をデシリアライズするときに *ap が NULL の場合は、適切なサイズの配列が割り当てられ、*ap はその配列を指すように設定されます。配列をシリアライズするときは、配列の要素数を *lp から取り出します。配列をデシリアライズするときは、*lp には配列の長さが設定されます。引数 maxlength は、配列に入れることができる最大要素数で

す。elementsiz は、配列の各要素のサイズ (バイト数) です。C の sizeof() 関数を使用してこの値を調べることができます。xdr_element() ルーチンは、配列の各要素のシリアライズ、デシリアライズ、解放を行うときに呼び出されます。

このほかの合成データ型の説明の前に、3 つのサンプルプログラムを説明します。

配列変換サンプルプログラム 1

ネットワークに接続したマシンのユーザーは、次の 3 つの項目によって識別できます。(a) マシン名。たとえば、krypton。(b) ユーザーの UID。これについては、geteuid(2) のマニュアルページを参照してください。(c) ユーザーが所属するグループ番号。これについては、getgroups(2) のマニュアルページを参照してください。これらの識別情報を持つ構造体と、それに対する XDR ルーチンはコード例 A–7 のようにコーディングできます。

コード例 **A–7**　配列変換のサンプルプログラム 1

```
struct netuser {
  char  *nu_machinename;
  int   nu_uid;
  u_int nu_glen;
  int   *nu_gids;
 };
#define NLEN 255        /* マシン名は 255 文字以下 */
#define NGRPS 20        /* ユーザーが所属するグループ数は 20 以下 */

bool_t
xdr_netuser(xdrs, nup)
  XDR *xdrs;
  struct netuser *nup;
{
  return(xdr_string(xdrs, &nup->nu_machinename, NLEN) &&
       xdr_int(xdrs, &nup->nu_uid) &&
       xdr_array(xdrs, &nup->nu_gids, &nup->nu_glen, NGRPS,
                 sizeof (int), xdr_int));
}
```

配列変換サンプルプログラム 2

ネットワークユーザーのパーティは、netuser 構造体の配列で表すことができます。構造体の宣言と、それに対する XDR ルーチンは コード例 A–8 のようになります。

コード例 **A–8**　配列変換のサンプルプログラム 2

```
struct party {

  u_int p_len;
  struct netuser *p_nusers;
};
#define PLEN 500 /* パーティに所属するユーザー数の上限 */
bool_t
xdr_party(xdrs, pp)
  XDR *xdrs;
  struct party *pp;
{
  return(xdr_array(xdrs, &pp->p_nusers, &pp->p_len, PLEN,
   sizeof (struct netuser), xdr_netuser));
}
```

配列変換サンプルプログラム 3

main に対するよく知られた引数 argc と argv を持つ構造体を作成し、その構造体の配列にコマンドヒストリを保存することができます。構造体の宣言と、その XDR ルーチンは コード例 A–9 のようになります。

コード例 **A–9**　配列変換のサンプルプログラム 3

```
struct cmd {
  u_int c_argc;
  char **c_argv;
};
#define ALEN 1000           /* argc は 1000 以下 */
 #define NARGC 100          /* 各コマンドの args は 100 以下 */

struct history {
  u_int h_len;
  struct cmd *h_cmds;
};
#define NCMDS 75            /* ヒストリは 75 コマンドまで */
bool_t
xdr_wrapstring(xdrs, sp)

  XDR *xdrs;
  char **sp;
{
  return(xdr_string(xdrs, sp, ALEN));
 }

bool_t
xdr_cmd(xdrs, cp)
```

(続く)

```
  XDR *xdrs;
  struct cmd *cp;
{
  return(xdr_array(xdrs, &cp->c_argv, &cp->c_argc, NARGC,
          sizeof (char *), xdr_wrapstring));
}
bool_t
xdr_history(xdrs, hp)

  XDR *xdrs;
  struct history *hp;
{
  return(xdr_array(xdrs, &hp->h_cmds, &hp->h_len, NCMDS,
          sizeof (struct cmd), xdr_cmd));
}
```

このプログラムで最もむずかしいのは、xdr_string() を呼び出すためのルーチン xdr_wrapstring() が必要な点です。xdr_array() が配列要素記述ルーチンを呼び出すときは引数が 2 つしか渡されないため、xdr_string() の第 3 引数を提供するルーチン xdr_wrapstring() が必要になります。

これまでの説明で XDR ライブラリの再帰的性質が明らかになりました。その他の合成データ型の変換も見てみましょう。

隠されたデータ

プロトコルによっては、サーバーからクライアントにハンドルが渡され、クライアントは後からハンドルをサーバーに送り返します。クライアントではハンドルの内容を調べることはなく、受け取ったものをそのまま送り返します。すなわち、ハンドルは隠されたデータ (内容が隠されたデータ) です。固定長の隠されたデータを記述するには、xdr_opaque() プリミティブを使用します。

```
bool_t
xdr_opaque(xdrs, p, len)
   XDR *xdrs;
   char *p;
   u_int len;
```

引数 p はデータのアドレスです。len は隠れたオブジェクトに入っているバイト数です。隠されたデータの定義からすると、実際に隠れたオブジェクトに入っているデータはマシン間で移植不可能です。

SunOS/SVR4 には、隠されたデータの操作用にもう 1 つのルーチンが提供されています。そのルーチン xdr_netobj() は xdr_opaque() と同様にカウント付きの隠されたデータを送信します。 コード例 A–10 に、xdr_netobj() の構文を示します。

コード例 **A–10**　xdr_netobj() のサンプルプログラム

```
struct netobj {
 u_int   n_len;
 char    *n_bytes;
};
typedef struct netobj netobj;

bool_t
xdr_netobj(xdrs, np)
 XDR *xdrs;
 struct netobj *np;
```

xdr_netobj() はフィルタプリミティブで、可変長の隠されたデータとその外部表現との変換を行います。引数 np は netobj() 構造体のアドレスです。netobj() 構造体には、隠されたデータの長さと隠されたデータへのポインタが入っています。長さは、MAX_NETOBJ_SZ バイトを超えてはいけません。変換に成功すれば TRUE、失敗すれば FALSE が返されます。

固定長配列

xdr_vector() ライブラリでは、コード例 A–11 サンプルプログラムで示すように、固定長配列に対するプリミティブ xdr_vector()が提供されています。

コード例 **A–11**　xdr_vector() のサンプルプログラム

```
#define NLEN 255 /* マシン名は 255 文字以下 */
#define NGRPS 20 /* ユーザーは正確に 20 のグループに所属 */

struct netuser {
  char *nu_machinename;
  int nu_uid;
  int nu_gids[NGRPS];
```

```
};

bool_t
xdr_netuser(xdrs, nup)
  XDR *xdrs;
  struct netuser *nup;
{
  int i;

 if (!xdr_string(xdrs, &nup->nu_machinename, NLEN))
   return(FALSE);
  if (!xdr_int(xdrs, &nup->nu_uid))
   return(FALSE);
  if (!xdr_vector(xdrs, nup->nu_gids, NGRPS, sizeof(int),
       xdr_int))
   return(FALSE);
  return(TRUE);
}
```

識別型の共用体

XDR ライブラリは識別型の共用体もサポートしています。識別型の共用体は、C の共用体に、共用体の「アーム」を選択する enum_t 型の値が付加されたものです。

```
struct xdr_discrim {
   enum_t value;
   bool_t (*proc)();
};

bool_t
 xdr_union(xdrs, dscmp, unp, arms, defaultarm)
   XDR *xdrs;
   enum_t *dscmp;
   char *unp;
   struct xdr_discrim *arms;
   bool_t (*defaultarm)(); /* may equal NULL */
```

このルーチンでは、最初に *dscmp にある要素識別子を変換します。要素識別子は常に enum_t 型です。次に、*unp にある共用体が変換されます。引数 arms は xdr_discrim 構造体配列へのポインタです。各構造体には、[value,proc] のペアが入っています。共用体の要素識別子が対応する value と一致すれば、それに

対する proc() が呼び出されて共用体が変換されます。xdr_discrim 構造体配列の終わりは、ルーチンの値が NULL(0) なので判別できます。arms 配列の中に要素識別子に一致するものがない場合は、defaultarm() 手続きが NULL でなければそれが呼び出されます。NULL の場合は FALSE を返します。

識別型の共用体サンプルプログラム

共用体のデータ型は、整数、文字へのポインタ (文字列)、gnumbers 構造体のどれかで、共用体と現在の型は構造体で宣言されています。宣言は次のようになります。

```
enum utype {INTEGER=1, STRING=2, GNUMBERS=3};
struct u_tag {
   enum utype utype; /* 共用体の要素識別子 */
   union {
      int ival;
      char *pval;
      struct gnumbers gn;
   } uval;
};
```

コード例 A–12 では、合成データと XDR 手続きで、識別型共用体のシリアライズ (またはデシリアライズ) を行います。

コード例 **A–12**　XDR 識別型の共用体サンプルプログラム

```
struct xdr_discrim u_tag_arms[4] = {
  {INTEGER, xdr_int},
  {GNUMBERS, xdr_gnumbers}
  {STRING, xdr_wrapstring},
  {__dontcare__, NULL}
  /* アームの最後は常に NULL の xdr_proc */
 }

bool_t
xdr_u_tag(xdrs, utp)
  XDR *xdrs;
  struct u_tag *utp;
{
  return(xdr_union(xdrs, &utp->utype, &utp->uval,
        u_tag_arms, NULL));
}
```

xdr_gnumbers() については「XDR ライブラリ」の節で説明し、xdr_wrapstring() はサンプルプログラム C に示しました。この例では、xdr_union()のデフォルトの arm 引数 (最後の引数) には NULL を渡しています。したがって、共用体の要素識別子の値は、u_tag_arms 配列にリストした値のどれかでなければなりません。また、この例から、arm 配列の要素はソートされていなくてもよいことがわかります。

要素識別子は、この例では連続した値を取っていますが、連続していなくてもかまいません。要素識別子の型の各要素に、明示的に整数値を割り当てておくと、要素識別子の外部表現のドキュメントにもなり、異なる C コンパイラでも要素識別子が同じ値で出力されることが保証されます。

練習

この節の他のプリミティブを使用して xdr_union() を作成してください。

ポインタ

C では、構造体の中に、別の構造体へのポインタを入れることがよくあります。プリミティブ xdr_reference() を使用すると、そのように参照される構造体を簡単にシリアライズ (またはデシリアライズ) できます。

```
bool_t
xdr_reference(xdrs, pp, size, proc)
   XDR *xdrs;
   char **pp;
   u_int ssize;
   bool_t (*proc)();
```

引数 pp は構造体へのポインタのアドレスです。引数 ssize は構造体のサイズ (バイト数) です。C の sizeof() 関数を使用してこの値を調べることができます。引数 proc() は構造体を記述する XDR ルーチンです。データを復号化するとき、*pp が NULL の場合は記憶領域が割り当てられます。

プリミティブ xdr_struct() では、構造体内の構造体を記述する必要はありません。ポインタだけで十分です。

練習

xdr_array() を使用して xdr_reference() を作成してください。

警告 - xdr_reference() と xdr_array() のデータ外部表現は互換ではありません。

ポインタのサンプルプログラム

人の名前、および、その人の総資産と負債の入った gnumbers 構造体へのポインタとで構成される次のような構造体があるとします。

```
struct pgn {
   char *name;
   struct gnumbers *gnp;
};
```

これに対する XDR ルーチンは次のようになります。

```
bool_t
xdr_pgn(xdrs, pp)
   XDR *xdrs;
   struct pgn *pp;
{
   return(xdr_string(xdrs, &pp->name, NLEN) &&
      xdr_reference(xdrs, &pp->gnp, sizeof(struct gnumbers),
                    xdr_gnumbers));
}
```

ポインタセマンティクス

C のプログラマは多くのアプリケーションで、ポインタの値に 2 つの意味を持たせています。一番多い例としては、ポインタ値が NULL (ゼロ) の場合はデータが不要なことを意味する方法ですが、アプリケーションごとにさまざまな解釈ができます。実際に C プログラマは、ポインタ値の解釈をオーバロードすることにより、効率よく識別型共用体を実現しています。たとえば、サンプルプログラム E では、gnp のポインタ値を NULL にすれば、その人の総資産と負債が未知であると解釈できます。すなわち、ポインタ値には 2 つの意味があります。データがあるか

どうかを示し、もしあるとしたらメモリーのどこにあるかを示します。リンクリストは、アプリケーション固有のポインタ解釈の極端な例です。

プリミティブ xdr_reference() はシリアライズの際に、値が NULL のポインタに特別な意味を持たせることができません。そのため、データをシリアライズするときに、xdr_reference() に値が NULL のポインタのアドレスを渡すと、多くの場合メモリーフォルトを引き起こし、UNIX システムではコアダンプが起こります。

xdr_pointer() では NULL ポインタも正しく処理できます。

フィルタ以外のプリミティブ

XDR ストリームの操作は、次に示すプリミティブで行います。

```
u_int xdr_getpos(xdrs)
   XDR *xdrs;

bool_t xdr_setpos(xdrs, pos)
   XDR *xdrs;
   u_int pos;

xdr_destroy(xdrs)
   XDR *xdrs;
```

ルーチン xdr_getpos()はデータストリーム内の現在位置を符号なし整数で返します。

警告 : XDR ストリームの中には、dr_getpos()の返す値が意味を持たないものがあります。その場合は、-1 が返されます (ただし、-1 も正当な値です)。

ルーチンxdr_setpos() はストリーム位置を posに設定します。

警告 : XDR ストリームの中には、xdr_setpos() で位置設定ができないものもあります。その場合は、FALSE が返されます。指定した位置が適用範囲外の場合もエラーとなります。設定位置の適用範囲はストリームごとに異なります。

XDR ストリームを廃棄するには、プリミティブ xdr_destroy() を使用します。このルーチンを呼び出した後で、ストリームを使用した場合の動作は未定です。

処理内容

処理内容 (XDR_ENCODE、XDR_DECODE、XDR_FREE のどれか) を利用して XDR ルーチンを最適化したい場合があります。XDR の処理内容は、常に xdrs->x_op に入っています。233ページの「リンクリスト」の節には、xdrs->x_op フィールドを活用するサンプルプログラムが示されています。

ストリームへのアクセス

XDR ストリームは、適切な作成ルーチンを呼び出すことで得られます。作成ルーチンへの引数でストリームのさまざまな特性が決まります。現在、データのシリアライズとデシリアライズに使用できるストリームには、標準入出力 FILE ストリーム、レコードストリーム、UNIX ファイル、メモリーがあります。

標準入出力ストリーム

XDR ストリームは、xdrstdio_create() を使用して標準入出力とのインタフェースをとることができます。

```
#include <stdio.h>
#include <rpc/rpc.h> /* XDR は RPC のサブセット */

void
xdrstdio_create(xdrs, fp, xdr_op)
   XDR *xdrs;
   FILE *fp;
   enum xdr_op x_op;
```

xdrstdio_create() は、xdrs が指す XDR ストリームを初期化します。XDR ストリームは、標準入出力ライブラリとのインタフェースが可能です。引数 fp はオープンしているファイル、x_op は XDR の処理内容です。

メモリーストリーム

メモリーストリームを作成すると、メモリーの特定領域とのデータストリームが使用できます。

```
#include <rpc/rpc.h>

void
xdrmem_create(xdrs, addr, len, x_op)
   XDR *xdrs;
   char *addr;
   u_int len;
   enum xdr_op x_op;
```

xdrmem_create() ルーチンは、ローカルメモリー内の XDR ストリームを初期化します。引数 addr は使用するメモリを指します。引数 len はメモリーの大きさ (バイト数) です。引数 xdrs と x_op は、xdrstdio_create() の同名の引数と同じです。現在、RPC ではデータグラムの実現に xdrstdio_create() を使用しています。TLI ルーチン t_sndndata() を呼び出す前に、完全な呼び出しメッセージ (または応答メッセージ) がメモリーに構築されます。

レコード (TCP/IP) ストリーム

レコードストリームは、レコードマーク標準の上に構築される XDR ストリームです。レコードマーク標準は、UNIX ファイル、または、4.2BSD 接続インタフェースの上に構築されます。

```
#include <rpc/rpc.h>      /* XDR は RPC のサブセット */

xdrrec_create(xdrs, sendsize, recvsize, iohandle, readproc,
              writeproc)
   XDR *xdrs;
   u_int sendsize, recvsize;
   char *iohandle;
   int (*readproc)(), (*writeproc)();
```

xdrrec_create() は、任意の長さの双方向レコードシーケンスが可能な XDR ストリームインタフェースを提供します。レコードの内容は、XDR 形式のデータと考えられます。レコードストリームは、主に RPC と TCP 接続とのインタフェースとして使用されますが、通常の UNIX ファイルとのデータ入出力にも使用できます。

引数 xdrs は、これまでに説明した同名の引数と同じです。レコードストリームでは、標準入出力ストリームと同様のデータバッファリングを行います。引数

sendsize と recvsize には、それぞれ送信バッファと受信バッファのサイズ (バイト数) を指定します。この値がゼロ (0) の場合は、あらかじめ指定されたデフォルトのバッファサイズが使用されます。バッファにデータを読み込んだり、データをフラッシュしたりするときは、ルーチン readproc() または writeproc() を呼び出します。この 2 つのルーチンの使用方法と機能は、UNIXのシステムコール read() と write() に似ていますが、第1 引数には隠されたデータ iohandle を渡します。その後の 2 つの引数 (および nbytes) と、戻り値 (バイトカウント) はシステムルーチンと同じです。次の xxx() を readproc() または writeproc() とすると、その形式は次のようになります。

```
/* 実際に伝送したバイト数を返す。エラーのときの戻り値は -1 */
int
xxx(iohandle, buf, len)
   char *iohandle;
   char *buf;
   int nbytes;
```

XDR ストリームには、バイトストリームのレコードを区切る方法が提供されています。XDR ストリームを作成するのに必要な抽象データ型については、231ページの「XDR ストリームの作成」を参照してください。XDRストリームレコードを区切るのに使用する RPC プロトコルについては、249ページの「レコードマーク標準」を参照してください。

レコードストリームに特有なプリミティブは以下のとおりです。

```
bool_t
xdrrec_endofrecord(xdrs, flushnow)
   XDR *xdrs;
   bool_t flushnow;

bool_t
xdrrec_skiprecord(xdrs)
   XDR *xdrs;

bool_t
xdrrec_eof(xdrs)
   XDR *xdrs;
```

xdrrec_endofrecord() ルーチンを呼び出すと、現在出力しているデータにレコードマークが付けられます。引数 flushnow に TRUE を指定すると、ストリーム

の writeproc() が呼び出されます。TRUE を指定しないと、出力バッファがいっぱいになったときに writeproc() が呼び出されます。

xdrrec_skiprecord() ルーチンを呼び出すと、入力ストリーム内の現在位置が、現在レコードの境界まで移動し、ストリーム内の次のレコードの始めに位置します。

ストリームの入力バッファにデータがなくなると、xdrrec_eof() ルーチンから TRUE が返されます。ストリームの元のファイル記述子にもデータが残っていないという意味ではありません。

XDR ストリームの作成

この節では、新たな XDR ストリームインスタンスの作成に必要な抽象データ型を示します。

XDR オブジェクト

コード例 A–13 の構造体は、XDR ストリームとのインタフェースを定義します。

コード例 A–13 XDR ストリームインタフェースの例

```
enum xdr_op {XDR_ENCODE=0, XDR_DECODE=1, XDR_FREE=2};

typedef struct {
 enum xdr_op x_op;
 struct xdr_ops {
   bool_t (*x_getlong)();        /* ストリームから long 型データを入力 */
   bool_t (*x_putlong)();        /* ストリームに long 型データを出力 */
   bool_t (*x_getbytes)();       /* ストリームから複数バイトを入力 */
   bool_t (*x_putbytes)();       /* ストリームに複数バイトを出力 */
   u_int (*x_getpostn)();        /* ストリームのオフセットを返す */
   bool_t (*x_setpostn)();       /* オフセットの再設定 */
   caddr_t (*x_inline)();        /* バッファリングされたデータへのポインタ */
   VOID (*x_destroy)();          /* プライベートエリアの解放 */
  bool_t (*x_control)();    /* クライアント情報の検索、変更 */
  bool_t (*x_getint32)();       /* ストリームから int を取得 */
   bool_t (*x_putint32)();       /* ストリームに int を出力 */
  } *x_ops;
  caddr_t x_public;                /* ユーザーデータ */
  caddr_t x_private;               /* プライベートデータへのポインタa */
  caddr_t x_base;                  /* 位置情報のためのプライベートデータ */
```

(続く)

```
  int   x_handy;                    /* その他のプライベートワード */
} XDR;
```

x_op フィールドは、現在ストリームに対して実行している処理内容を示します。このフィールドは XDR プリミティブにとっては重要なフィールドですが、ストリームの実現に対しては影響ありません。すなわち、ストリームは、この値に依存した方法で作成してはいけません。x_private、x_base、x_handy の各フィールドは、特定のストリームを実現するためのプライベートデータです。x_public は XDR クライアントのためのフィールドですので、XDR ストリームを実現するために使用したり、XDR プリミティブで使用したりしてはいけません。x_getpostn()、x_setpostn()、x_destroy() はストリームへのアクセス操作に使用するマクロです。x_inline() ルーチンには、XDR * と符号なし整数 (バイトカウント) の 2 つの引数を渡します。このルーチンからは、ストリームの内部バッファセグメントへのポインタが返されます。呼び出し側ではそのバッファセグメントを自由に使用できます。ストリーム側からみると、そのバッファセグメントに入っているデータは失われます。このルーチンは、要求されたサイズのバッファセグメントを返せない場合は NULL を返します。x_inline() ルーチンは、バッファをサイクリックに使用するためのもので、このバッファの使用は移植可能ではありません。この機能の使用はお勧めできません。

ストリームに対するバイトシーケンスの単純な入出力には、x_getbytes() と x_putbytes() を使用します。入出力に成功すれば TRUE、失敗すれば FALSE が返されます。この 2 つのルーチンの引数は同じです。xxx を関数名に合わせて置き換えます。

```
bool_t
xxxbytes(xdrs, buf, bytecount)
   XDR *xdrs;
   char *buf;
   u_int bytecount;
```

x_getint32() と x_putint32() のオペレーションにより、データストリームから int 数値を受け取ったり、データストリームへ数値を出力したりできます。この 2 つのルーチンは、数値のマシン固有表現から標準外部表現への変換を担当します。

変換には、UNIX のプリミティブ htonl() と ntohl() を利用できます。上位レベルのXDR 対応ソフトウェアでは、符号付きと符号なしの整数のビット数は同じで、非負の整数のビットパターンは符号なしの整数のビットパターンと同じであるとみなします。入出力に成功すれば TRUE 、失敗すれば FALSE が返されます。

関数 x_getint() と x_putint() では、これらのオペレーションを使用します。この 2 つのルーチンの引数は同じです。

```
bool_t
xxxint(xdrs, ip)
   XDR *xdrs;
   int32_t *ip;
```

これらのオペレーションの long バージョン (x_getlong() と x_putlong()) でも、 x_getint32() と x_putint32() を呼び出します。この場合、プログラムが 64 ビットマシンまたは 32 ビットマシンのどちらで実行されていても、処理される量は必ず 4 バイトになります。

XDR ストリームを新規開発するときは、作成ルーチンを使用して、新たな操作ルーチンを持つ XDR 構造体を作成し、クライアントに提供しなければなりません。

高度な機能

この節では、データ構造体を引き渡す方法を説明します。そのような構造体の 1 つに、リンクリスト (長さは任意) があります。これまでの簡単なサンプルプログラムとは違い、ここでは XDR C ライブラリルーチンと、XDR データ記述言語の両方を使用します。XDR データ記述言語詳細は、付録 C を参照してください。

リンクリスト

226ページの「ポインタのサンプルプログラム」の節では、各個人の総資産と負債に関する C のデータ構造体とそれに対する XDR ルーチンのサンプルプログラムを示しました。コード例 A–14 では、リンクリストを使用して、ポインタのサンプルプログラムと同じものを作成します。

コード例 **A–14**　リンクリストのサンプルプログラム

```
struct gnumbers {
  int g_assets;
  int g_liabilities;
 };

bool_t
xdr_gnumbers(xdrs, gp)
  XDR *xdrs;
  struct gnumbers *gp;
{
  return(xdr_int(xdrs, &(gp->g_assets) &&
        xdr_int(xdrs, &(gp->g_liabilities)));
}
```

同じ情報を持つリンクリストを作成します。構造体は次のようになります。

```
struct gnumbers_node {
   struct gnumbers gn_numbers;
   struct gnumbers_node *gn_next;
};
typedef struct gnumbers_node *gnumbers_list;
```

リンクリストのヘッドは単に構造体の短縮形というわけではなく、データオブジェクトと考えられます。同様に、gn_next フィールドも、オブジェクトの終わりかどうかを示すのに使用されます。残念ながら、オブジェクトが次につながる場合も、gn_next フィールドに続きのアドレスが入ります。オブジェクトをシリアライズするときは、リンクアドレスの情報は役に立ちません。

このリンクリストを XDR データ記述言語で示すと、gnumbers_list の再帰宣言となります。

```
struct gnumbers {
   int g_assets;
   int g_liabilities;
};
struct gnumbers_node {
   gnumbers gn_numbers;
   gnumbers_node *gn_next;
};
```

この記述では、次のデータがあるかどうかがブール値でわかります。ブール値が FALSE の場合は、それが構造体の最終データフィールドとなります。ブール値が TRUE の場合は、その後に gnumbers 構造体と gnumbers_list とが (再帰的に) 続きます。C の宣言では明示的に宣言されたブール値がなく (ただし、gn_next フィールドが暗黙にその情報を持ちます)、XDR データ記述では明示的に宣言されたポインタがないことに注意してください。

gnumbers_list に対する XDR ルーチンを作成するためのヒントは、上のXDR 記述から簡単に得られます。上の XDR 共用体を実現するためのプリミティブ xdr_pointer()の使用方法に注意してください。

コード例 **A–15**　xdr_pointer() のサンプルプログラム

```
bool_t
xdr_gnumbers_node(xdrs, gn)
  XDR *xdrs;
  gnumbers_node *gn;
{
 return(xdr_gnumbers(xdrs, &gn->gn_numbers) &&
        xdr_gnumbers_list(xdrs, &gn->gn_next));
}

bool_t
xdr_gnumbers_list(xdrs, gnp)
  XDR *xdrs;
  gnumbers_list *gnp;
{
 return(xdr_pointer(xdrs, gnp, sizeof(struct gnumbers_node),
                    xdr_gnumbers_node));
 xdr_pointer}
```

リンクリストをこれらのルーチンで XDR を使用すると、C のスタックがリスト内のノードの数だけ増大するという問題が起こります。これは再帰呼び出しが原因です。コード例 A–16 のルーチンは、互いに再帰呼び出しを行う上の 2 つのルーチンを 1 つにまとめて、再帰呼び出しが起こらないようにしたものです。

コード例 **A–16**　再帰呼び出しを行わない XDR 変換

```
bool_t
xdr_gnumbers_list(xdrs, gnp)
  XDR *xdrs;
  gnumbers_list *gnp;
{
  bool_t more_data;
```

```
  gnumbers_list *nextp;

 for(;;) {
   more_data = (*gnp != NULL);
   if (!xdr_bool(xdrs, &more_data))
    return(FALSE);
   if (! more_data)
    break;
   if (xdrs->x_op == XDR_FREE)
    nextp = &(*gnp)->gn_next;
   if (!xdr_reference(xdrs, gnp,
   sizeof(struct gnumbers_node), xdr_gnumbers))
   return(FALSE);
   gnp = (xdrs->x_op == XDR_FREE) ? nextp : &(*gnp)->gn_next;
  }
  *gnp = NULL;
  return(TRUE);
}
```

最初に行うのは、次に続くデータがあるかどうかを調べ、このブール値をシリアライズできるようにします。XDR_DECODE の場合はこの文は不要なことに注意してください。なぜなら、次の文で more_data をデシリアライズするまでブール値はわからないからです。

次の文で XDR 共用体の more_data フィールドに XDR を作成します。本当に次のデータがないことがわかれば、最終ポインタを NULL に設定してリストの終了を示し、処理は終了したので TRUE を返します。XDR_DECODE の場合だけ、ポインタを NULL に設定する意味があることに注意してください。XDR_ENCODE と XDR_FREE の場合は既に NULL になっているからです。

処理内容が XDR_FREE の場合は次に、リストの次のポインタアドレスを nextp に保存します。このアドレスはここで保存しておかなければなりません。なぜならば、次の文で gnp をデレファレンスして、リスト内の次の項目のアドレスを取り出しますが、その文を実行すると、gnp が指している記憶領域は解放されてその中のデータが失われるためです。XDR_DECODE の場合は、次の文を実行するまで gnp に値が設定されないため、処理内容によってアドレス保存を行うかどうかを決めています。

次にプリミティブ xdr_reference() を使用して、このノードのデータを XDR 変換します。xdr_reference() はこれまでに使用した xdr_pointer() に似ていますが、次のデータがあるかどうかを示すブール値を送信しない点が異なります。ここでは、その情報を既にXDR 変換してしまっているので、xdr_pointer() は使用

しません。ここで、リストの要素と異なる型の XDR ルーチンを渡していることに注意してください。ルーチン xdr_gnumbers() が渡されますが、実際にリストに入っている各要素の型は gnumbers_node() です。xdr_gnumbers_node() を渡さないのは、このルーチンが再帰呼び出しを行うからです。代わりに渡された xdr_gnumbers() は、非再帰部分をすべて XDR 変換します。このような方法がうまくいくのは、各要素の最初の項目が gn_numbers()フィールドの場合だけです。最初のフィールドなので xdr_reference() に渡される両者のアドレスが同じだからです。

最後に gnp の値を更新して、リスト内の次の項目を指すようにします。処理内容が XDR_FREE の場合は保存しておいた値を設定し、それ以外の場合は gnpをデレファレンスして正しい値を取り出します。再帰呼び出しを使用する方法よりむずかしくなりますが、この方が数多くの手続き呼び出しのオーバヘッドがなくなってパフォーマンスも向上します。もっとも、ほとんどのリストはそれほど大きくはないので (項目数がせいぜい 100 以下)、その場合は再帰呼び出しを行なっても問題ありません。

付録 **B**

RPC プロトコルおよび言語の仕様

付録 B では、RPC パッケージで使用しているメッセージプロトコルの仕様を説明します。メッセージプロトコルは XDR 言語を使用して示します。XDR についての詳細は、 付録 C を参照してください。

プロトコルの概要

RPC プロトコルには、以下の機能があります。

- 呼び出される手続きの仕様が一意です。
- 要求メッセージに対する応答メッセージを提供します。
- 呼び出し側からサーバーへ、およびサーバーから呼び出し側への認証を提供します。さらに、RPC パッケージには、以下の現象を検出する機能があります。
  - RPC プロトコルの不一致
  - 遠隔プログラムプロトコルのバージョンの不一致
  - プロトコルのエラー (手続きのパラメータの仕様が間違っているなど)
  - 遠隔認証が失敗した原因

ネットワーク・ファイル・サービスが 2 つのプログラムで構成されていると考えてください。1 つのプログラムは、ファイルシステムへのアクセス制御やロックなど高レベルのアプリケーションを扱います。もう 1 つのプログラムは、低いレベルの入出力ファイルを扱い、「読込み」や「書込み」などの手続があります。ネットワークファイルサービスのクライアントマシンは、クライアントマシンのユーザーのために 2 つのプログラムに関連する手続きを呼び出します。クライアントサーバー・モデルでは、遠隔手続き呼び出しは、サービスを呼び出す場合に使用します。

RPC モデル

RPC モデルは、ローカル手続き呼び出しモデルに似ています。ローカル手続き呼び出しでは、呼び出し側が手続きへの引数を特定の記憶領域に書き込みます。次に呼び出し側が制御を手続きに渡しますが、最終的には再び制御が呼び出し側に戻ります。その時点で手続きからの戻り値は特定の記憶領域から取り出され、呼び出し側は処理を続行します。

RPC もこれと同様に実行されますが、1 つの制御スレッドが論理的に 2 つのプロセスにまたがって実行されます。2 つのプロセスとは、呼び出し側のプロセスとサーバープロセスです。概念的には、呼び出し側のプロセスがサーバープロセスに呼び出しメッセージを送り、その後応答メッセージが返されるまで待ちます (ブロックします)。呼び出しメッセージには、さまざまなデータとともに手続きへの引数が含まれています。応答メッセージには、さまざまなデータとともに手続きからの戻り値が含まれています。応答メッセージを受信すると、その中から戻り値が取り出され、呼び出し側の処理が再開します。

サーバー側では、プロセスは呼び出しメッセージが到着するまで休止しています。呼び出しメッセージが到着すると、サーバープロセスはその中から手続きへの引数を取り出し、戻り値を計算し、応答メッセージを送信して、次の呼び出しメッセージが来るのを待ちます。

この説明では、2 つのプロセスのうち同時にはどちらか一方だけがアクティブになることに注意してください。ただし、これ以外の呼び出し方法も可能です。RPCプロトコルでは、同時実行モデルも使用できます。たとえば、RPC 呼び出しを非同期モードで実行すれば、クライアントはサーバーからの応答を待つ間も作業を続けることができます。また、サーバーでは、到着した要求を処理するために新たなタスクを生成して、別の要求の受信も続けることができます。

トランスポートとセマンティクス

RPC プロトコルとトランスポートプロトコルとは互いに独立しています。すなわち、RPC では、メッセージがプロセス間で実際にどのように伝送されるかについては関知しません。RPC プロトコルで対象にしているのは、メッセージの仕様と解釈方法だけです。

RPC では、トランスポートの信頼性を保証していません。ですから、RPC で使用されるトランスポートプロトコルの型についての情報をアプリケーションに指定する必要があります。もし、RPC サービスが TCP のような信頼性の高いトランスポートを使用しているとわかっていれば、必要な作業はほとんどトランスポートで実行されています。反対に、RPC が UDP のような信頼性の低いトランスポート上で実行されている場合は、サービスの方で再転送やタイムアウトに対する処理を行わなければなりません。RPC ではそのようなサービスを提供しません。

RPC はトランスポート独立であるため、RPC プロトコルは、遠隔手続きやその実行に特定のセマンティクスを結び付けることができません。セマンティクスは、使用しているトランスポートプロトコルから推測されます。ただし、明示的に指定されなければなりません。たとえば、RPC が信頼性の低いトランスポート上で実行されている場合を考えてみます。アプリケーションが短時間のタイムアウト後に RPC メッセージを再転送する場合、応答が返されなければ手続きが 0 回以上実行されたことしか推測できません。応答が返された場合は、手続きが少なくとも一度は実行されたことが推測できます。

サーバーでは、一度だけ実行というセマンティクスをある程度実現するため、以前にクライアントから受け取った要求を記憶しておいて、同じ要求を再受信しないようにする場合があります。その場合サーバーは、RPC要求に必ず含まれているトランザクション ID を利用します。トランザクション ID は、主として RPC クライアントが、応答と要求との対応を調べるために使用します。クライアントアプリケーションでは、要求を再送信するときに以前のトランザクション ID を再使用することができます。サーバーアプリケーションでもこのことを確認していれば、要求を受信したときはトランザクション ID を記憶しておいて、同じ ID を持つ要求は再受信しないことができます。サーバーでは、以前と同じ要求かどうか調べるため以外の目的でトランザクション ID を使用することはできません。

反対に TCP のような信頼性の高いトランスポートを使用している場合、応答メッセージが返されればアプリケーションは手続きが一度だけ実行されたと推測できます。ところが、応答メッセージが返されないからといって、遠隔手続きが一度も実行されなかったと推測することはできません。TCP のような接続型プロトコルを使用する場合も、サーバーのクラッシュに対応するために、アプリケーションでタイムアウトと再接続確立の操作が必要なことに注意してください。

結合と相互認識の独立性

クライアントとサービスの結合は、遠隔手続き呼び出しの仕様の一部ではありません。結合という重要で不可欠な機能は、より上位レベルのソフトウェアで行います。そのソフトウェアでも RPC を使用することがあります。271ページの「rpcbind プロトコル」の節を参照してください。

そのようなソフトウェアを開発する場合は、RPC プロトコルをネットワーク間のジャンプ - サブルーチン命令 (JSR 命令) と考えます。ローダ (バインダ) は、JSR 命令を実行可能にするために、ローダ自身も JSR 命令を使用します。同様に、ネットワークは RPC を実行可能にするためにネットワーク自身も RPC を使用します。

RPC プロトコルには、サービスに対してクライアントが自分自身を証明するため、またはその反対方向の証明のためのフィールドが用意されています。セキュリティやアクセス制御の機能は、メッセージ認証の上に成り立っており、何種類かの認証プロトコルをサポートできます。どのプロトコルを使用するかは、RPC ヘッダーの 1 フィールドで指定します。認証プロトコルについての詳細は、249ページの「レコードマーク標準」の節を参照してください。

プログラムと手続き番号

RPC 呼び出しメッセージには、呼び出される手続きを一意に識別する次の 3 つの符号なしフィールドがあります。

- 遠隔プログラム番号
- 遠隔プログラムのバージョン番
- 遠隔手続き番号

プログラム番号は、245ページの「プログラム番号の登録」にあるように、中央の 1 人の管理者が決定します。

プログラムを最初に作成したときは、バージョン番号は通常 1 になります。プロトコルは次第に改善されて、安定し、よりよいプロトコルになるため、呼び出し側のプロセスでは、呼び出しメッセージのバージョンフィールドを使用してどのプロトコルバージョンを使用するかを指定できます。バージョン番号を使用することにより、これまで使用していたプロトコルと新規プロトコルとが同じサーバープロセスで「使用可能」になります。

手続き番号では、どの手続きを呼び出すかを指定します。手続き番号は、各プログラムのプロトコル仕様に記されています。たとえば、ファイルサービスのプロトコル仕様には、手続き番号 5 は `read` で、手続き番号 12 は `write` というように記されています。

遠隔プログラムのプロトコルがバージョンが変わるたびに変更されるように、RPC メッセージプロトコルも変わることがあります。したがって、呼び出しメッセージには RPC バージョン番号も入っています。ここで説明する RPC のバージョン番号は常に 2 です。

要求メッセージに対する応答メッセージには、次に示すエラー条件を識別できるような情報が入っています。

- RPC の遠隔プログラム側がプロトコルバージョン 2 を使用していない。サポートしている RPC バージョン番号の最大値と最小値が返される
- 遠隔システム上で指定した遠隔プログラムが使用できない
- 遠隔プログラムは要求されているバージョン番号をサポートしていない。サポートしている遠隔プログラムバージョン番号の最大値と最小値が返される
- 要求されている手続き番号が存在しない。これは、呼び出し側のプロトコルエラーかプログラミングエラーであることが多い
- サーバー側から見ると、遠隔手続きへの引数に誤りがある。このエラーもまた、クライアントとサービスの間のプロトコルの不一致による場合が多い

RPC プロトコルの一部として、呼び出し側からサービスへの認証、および、その反対方向の認証が提供されています。呼び出しメッセージには、認証証明とベリファイアという 2 つの認証フィールドがあります。応答メッセージには、応答ベリファイアという認証フィールドがあります。RPC プロトコル仕様では、この 3 つのフィールドはすべて次のような隠されたデータ型で定義されています。

```
enum auth_flavor {
   AUTH_NONE = 0,
   AUTH_SYS = 1,
   AUTH_SHORT = 2,
   AUTH_DES = 3,
  AUTH_KERB = 4
   /* その他のタイプも定義可能 */
  };
  struct opaque_auth {
   enum         auth_flavor;        /* 認証のタイプ */
   caddr_t      oa_base;          /* その他の認証データのアドレス */
   u_int        oa_length;        /* データ長は MAX_AUTH_BYTES 以下 */
  };
```

`opaque_auth` 構造体には、列挙型 `auth_flavor` に続いて、RPC プロトコルには隠された認証データが入ります。

認証フィールドに入っているデータの解釈とセマンティクスは、個々の独立した認証プロトコル仕様で定義します。さまざまな認証プロトコルについては、249ページの「レコードマーク標準」の節を参照してください。

認証パラメータが拒絶された場合は、応答メッセージの中に拒絶理由が返されます。

プログラム番号の割り当て

0x20000000 のグループのプログラム番号は 表 B–1 に示すように分散されます。

表 B–1 RPC プログラム番号

プログラム番号	説明
00000000 - 1fffffff	米国サンマイクロシステムズ社で定義
20000000 - 3fffffff	ユーザーが定義
40000000 - 5fffffff	一時的 (カスタマ作成アプリケーションのために予約)
60000000 - 7fffffff	予約
80000000 - 9fffffff	予約
a0000000 - bfffffff	予約
c0000000 - dfffffff	予約
e0000000 - ffffffff	予約

最初のグループの番号は全カスタマで一致している必要があり、サン・マイクロシステムズ社で管理しています。一般に使用できるアプリケーションをカスタマが開発した場合は、そのアプリケーションに最初のグループの番号を割り当てなければなりません。

第 2 グループの番号は特定のカスタマアプリケーションのために予約されています。この範囲の番号は、主に新規プログラムのデバッグで使用します。

第 3 グループは、動的にプログラム番号を生成するアプリケーションのために予約されています。

最後のグループは将来のために予約されており、使用してはいけません。

プログラム番号の登録

プロトコル仕様を登録するには、email で rpc@sun.comに送信するか、次の住所に送ってください。

```
RPC Administrator
Sun Microsystems, Inc
901 San Antonio Blvd.
Palo Alto,CA 94303
U.S.A.
```

その際には rpcgen で生成した、プロトコルを記述する「.x」ファイルも同封してください。一意に識別できるプログラム番号を返送します。

標準 RPC サービスの RPC プログラム番号とプロトコル仕様は、/usr/include/rpcsvc のインクルードファイルに入っています。ただし、これらのサービスは登録されているサービスのほんの一部分にすぎません。

RPC プロトコルのその他の使用方法

本来、RPC プロトコルは遠隔手続き呼び出しを目的に作成されています。すなわち、各呼び出しメッセージがそれぞれ 1 つの応答メッセージに一致します。ところが、プロトコル自体はメッセージ引き渡しプロトコルですので、RPC 以外のプロトコルで対応できます。RPC パッケージでサポートされている RPC 以外のプロトコルとしては、バッチとブロードキャストがあります。

バッチ

バッチを使用すると、クライアントは任意の大きさの呼び出しメッセージシーケンスをサーバーに送信できます。一般にバッチでは、トランスポートとして TCP のような信頼性の高いバイトストリームプロトコルを使用します。バッチを使用すると、クライアントはサーバーからの応答を待たず、サーバーもバッチ要求に対しては応答しません。バッチ呼び出しシーケンスを終了するには、通常、非バッチの RPC 呼び出しを行なってパイプラインをフラッシュします。このときは肯定応答が返されます。詳細については、102ページの「バッチ処理」の節を参照してください。

ブロードキャスト RPC

ブロードキャスト RPC では、クライアントがブロードキャストパケットをネットワークに送信し、それに対する数多くの応答を待ちます。ブロードキャスト RPC では、トランスポートに UDP のような非接続型のパケットベースプロトコルを使用します。ブロードキャストプロトコルをサポートするサーバーは、要求を正しく処理できたときだけ応答を返し、エラーが起これば応答は返しません。ブロードキャスト RPC では rpcbind サービスを使用してそのセマンティクスを達成します。詳細については、 100ページの「ブロードキャスト RPC」、および 271ページの「rpcbind プロトコル」 の節を参照してください。

RPC メッセージプロトコル

この節では、RPC メッセージプロトコルを、XDR データ記述言語を使用して説明します。メッセージは、コード例 B–1 で示すようにトップダウン形式で定義します。

コード例 **B–1**　RPC メッセージプロトコル

```
enum msg_type {
  CALL = 0,
  REPLY = 1
 };

/*
 * 呼び出しメッセージに対する応答には、2 つの形式があります。メッセージが
 * 受け入れられた場合と拒絶された場合のどちらかです。
 */
 enum reply_stat {
  MSG_ACCEPTED = 0,
  MSG_DENIED = 1
 };

/*
 * 呼び出しメッセージが受け入れられた場合、遠隔手続きを呼び出したときの
 * ステータスが次のように示されます。
 */
enum accept_stat {
  SUCCESS = 0,       /* RPC が正常に実行された */
  PROG_UNAVAIL = 1,  /* 遠隔サービスにエクスポートされたプログラムがない */
  PROG_MISMATCH = 2, /* 遠隔サービスがそのバージョン番号をサポートしていない */
  PROC_UNAVAIL = 3,  /* プログラムがその手続きをサポートしていない */
  GARBAGE_ARGS = 4   /* 手続きが引数を復号化できない */
};

/*
 * 呼び出しメッセージが拒絶された原因
 */
```

(続く)

```
enum reject_stat {
  RPC_MISMATCH = 0,  /* RPC のバージョン番号が 2 でない */
  AUTH_ERROR = 1     /* 遠隔サービスで呼び出し側の認証エラー */
};
/*
 * 認証が失敗した原因
 */
enum auth_stat {
  AUTH_BADCRED = 1,        /* 認証エラーの原因 */
  AUTH_REJECTEDCRED = 2,   /* クライアントは新規セッションが必要 */
  AUTH_BADVERF = 3,        /* ベリファイアのエラー */
  AUTH_REJECTEDVERF = 4,   /* ベリファイアの失効または再使用 */
  AUTH_TOOWEAK = 5         /* セキュリティによる拒絶 */
};

/*
 * RPC メッセージ:
 * どのメッセージもトランザクション ID xid と
 * それに続く識別型共用体 (アームは 2 つ) で始まります。
 * 共用体の要素識別子は msg_type で、2 つのメッセージタイプのうち
 * どちらのタイプのメッセージかを示します。REPLY メッセージの xid は、
 * 対応する CALL メッセージの xid に一致します。注意: xid フィールドは、
 * クライアント側で応答メッセージがどの呼び出しメッセージに対応するかを
 * 調べるか、サーバー側で再送信かどうかを調べるためにだけ使用できます。
 * サービス側では xid をなんらかのシーケンス番号として使用することはできません。
 */
struct rpc_msg {
  unsigned int xid;
  union switch (msg_type mtype) {
   case CALL:
    call_body cbody;
   case REPLY:
    reply_body rbody;
  } body;
};

/*
 * RPC 要求呼び出しの本体:
 * RPC プロトコル仕様のバージョン 2 では、rpcvers は 2 でなければ
 * なりません。prog、vers、proc の各フィールドにはそれぞれ、
 * 遠隔プログラム、そのバージョン番号、遠隔プログラムに入っている
 * 呼び出し対象の手続きを指定します。これらのフィールドに続いて
 * 2 つの認証パラメータ cred (認証を証明するもの) と verf (認証を検証する
 * もの : 認証ベリファイア) があります。この 2 つの認証パラメータの後には、
 * 遠隔手続きへの引数が入りますが、それらは特定プログラムの
 * プロトコルで指定されます。
*/
struct call_body {
  unsigned int rpcvers; /* この値は 2 でなければならない */
  unsigned int prog;
  unsigned int vers;
  unsigned int proc;
  opaque_auth cred;
```

(続く)

```
  opaque_auth verf;
  /* ここからは手続きに固有の引数 */
 };

/*
 * RPC 要求への応答の本体:
 * 呼び出しメッセージは受け入れられたか拒絶されたかのどちらか
 */
union reply_body switch (reply_stat stat) {
  case MSG_ACCEPTED:
   accepted_reply areply;
  case MSG_DENIED:
   rejected_reply rreply;
} reply;

/*
 * RPC 要求がサーバーに受け入れられた場合の応答: 要求が受け入れられた場合も
 * エラーはあり得ます。最初のフィールドはサーバーが呼び出し側に自分自身を
 * 証明する認証ベリファイアです。次のフィールドは共用体で、
 * 要素識別子は列挙型 accept_stat です。この共用体の SUCCESS アームは
 * プロトコルによって異なります。
 * PROG_UNAVAIL、PROC_UNAVAIL、GARBAGE_ARGP の
 * アームは void です。PROG_MISMATCH アームにはサーバーが
 * サポートしている遠隔プログラムのバージョン番号の
 * 最大値と最小値が入ります。
 */
struct accepted_reply {
  opaque_auth verf;
  union switch (accept_stat stat) {
   case SUCCESS:
    opaque results[0];
    /* ここからは手続き固有の戻り値 */
   case PROG_MISMATCH:
    struct {
     unsigned int low;
     unsigned int high;
    } mismatch_info;
   default:
    /*
     * PROG_UNAVAIL、PROC_UNAVAIL、GARBAGE_ARGS
       * の場合は void
     */
    void;
  } reply_data;
 };

/*
 * RPC 要求がサーバーに拒絶された場合の応答:
 * 要求が拒絶されるのには 2 つの原因があります。互換性のあるバージョンの
 * RPC プロトコルがサーバーで実行されていない場合 (RPC_MISMATCH) と、
 * サーバーが呼び出し側の認証を拒否した場合 (AUTH_ERROR) です。
 * RPC バージョンの不一致の場合は、サーバーがサポートしている RPC バージョンの
 * 最大値と最小値が返されます。認証拒否の場合は、
```

(続く)

```
 * 異常終了ステータスが返されます。
 */
union rejected_reply switch (reject_stat stat) {
  case RPC_MISMATCH:
   struct {
    unsigned int low;
    unsigned int high;
   } mismatch_info;
  case AUTH_ERROR:
   auth_stat stat;
};
```

レコードマーク標準

RPC メッセージが TCP のようなバイトストリーム型のトランスポートに渡されるとき、ユーザープロトコルエラーを検出し、できれば回復するために各メッセージの区切りを知る必要があります。これをレコードマーク (RM) といいます。1 つの RPC メッセージは 1 つの RM レコードに収められます。

レコードはいくつかのレコードフラグメントで構成されます。レコードフラグメントには、4 バイトのヘッダーに続いて 0〜 (2**31) - 1 バイトのフラグメントデータが入っています。データには符号なしバイナリ数値が符号化され、バイト順序は XDR 整数と同様にネットワークのバイト順序に従います。

ヘッダーには次の 2 つの値が符号化されています。

- レコード内の最終フラグメントかどうかを示すブール値 (ビット値が 1 の場合は最終フラグメント)
- フラグメントデータの長さ (バイト数) を示す 31 ビットの符号なしバイナリ値。最終フラグメントを示すブール値はヘッダーの最上位ビットに入り、データ長は下位 31 ビットに入る。このレコード仕様は、XDR 標準形式には含まれない

認証プロトコル

認証パラメータは内容が隠されていますが、以降の RPC プロトコルで自由に解釈できます。この節では、既に定義されているタイプの認証について説明します。別の

サイトでは自由に新たな認証タイプを作成し、プログラム番号割り当て規則と同様の認証タイプ番号割り当て規則に従って、認証タイプ番号を割り当てることができます。認証タイプ番号はサン・ソフトで保守・管理しています。認証番号の割り当てを希望されるユーザーは、RPC プログラム番号を割り当てるときと同様に、245 ページの「プログラム番号の登録」に示すように、サン・マイクロシステムズ社の RPC 管理者に連絡してください。

AUTH_NONE

呼び出し側は自身を証明せず、また、サーバー側も呼び出し側が誰でもかまわないという呼び出しもあります。その場合は RPC メッセージの認証証明、ベリファイア、応答ベリファイアの `flavor` (`opaque_auth` 共用体の要素識別子) は `AUTH_NONE` にします。`AUTH_NONE` タイプの認証を使用するときは、body フィールドの長さをゼロにします。

AUTH_SYS

これは、`AUTH_UNIX` として知られる以前に説明した認証 *flavor* と同じです。遠隔手続きを呼び出す側では、従来の UNIX のプロセス許可認証を使用して自分自身を証明する場合があります。そのような RPC 呼び出しメッセージでは、`opaque_auth` の *flavor* は `AUTH_UNIX`となります。`body` には、次に示す構造体が符号化されます。

```
struct auth_sysparms {
   unsigned int  stamp;
   string machinename<255>;
   uid_t uid;
   gid_t gid;
   gid_t gids<10>;
 };
```

- *stamp* は、呼び出し側のマシンで生成できる任意の ID
- *machinename* は、呼び出し側のマシン名
- *uid* 呼び出し側の実効ユーザー ID
- *gid* は、呼び出し側の実効グループ
- *gids* は、呼び出し側がメンバーであるグループの可変長配列

認証証明に伴うベリファイアの *flavor* は `AUTH_NONE` でなければなりません。

AUTH_SHORT タイプのベリファイア

AUTH_SYSタイプの認証を使用するときは、サーバーからの応答メッセージに入っている応答ベリファイアの *flavor* は AUTH_NONE か AUTH_SHORT のどちらかです。

AUTH_SHORT の場合、応答ベリファイアの文字列には short_hand_verf 構造体が符号化されています。この隠された構造体を、元の AUTH_SYS 認証証明の代わりにサーバーに渡すことができます。

サーバー側では、隠された short_hand_verf 構造体 (AUTH_SHORT タイプの応答ベリファイアによって返される) を呼び出し側の元の認証証明にマップするキャッシュを保存します。呼び出し側は、新たな認証証明を使用してネットワークの帯域幅とサーバーの CPU サイクルを保存できます。

サーバー側では、隠された short_hand_verf 構造体をいつでもフラッシュできます。そうすると、遠隔手続き呼び出しメッセージは認証エラーにより拒絶されます。エラー原因は AUTH_REJECTEDCRED になります。この場合、呼び出し側では AUTH_SYS タイプの元の認証証明を試すこともできます。 図 B–1 を参照してください。

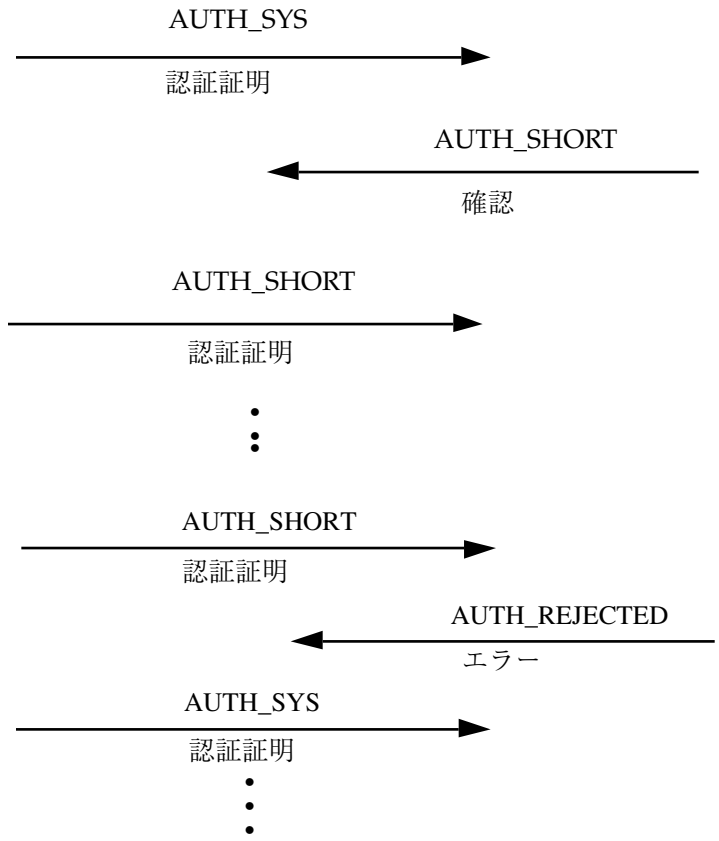


図 B–1　認証過程のマップ

AUTH_DES タイプの認証

AUTH_SYS タイプの認証には次のような問題があります。

1. 異なるオペレーティングシステムのマシンが同じネットワークに接続している場合に、呼び出し側の ID が一意に決まるとは限りません。
2. ベリファイアがないため簡単に認証証明をごまかすことができます。AUTH_DES タイプの認証はこの 2 つの問題を解決するための方法です。

最初の問題を解決するには、呼び出し側をオペレーティングシステム固有の整数ではなく単純文字列で呼びます。この文字列のことを、呼び出し側の *netname* (ネットワーク名) といいます。サーバーでは、呼び出し側の識別のためにだけ呼び出し側

の名前を使用します。したがって、名前ドメイン内では呼び出し側を一意に識別できるようなネットワーク名を設定しなければなりません。

遠隔サーバーを呼び出す各ユーザーに一意のネットワーク名を生成するのは、それぞれのオペレーティングシステムで実現されている AUTH_DES 認証機能の責任です。オペレーティングシステムには既に、システムのローカルユーザーを識別します。通常はこれを単純に拡張してネットワーク名とします。たとえば、ユーザー ID を 515 とすると、「UNIX.515@sun.com」いうネットワーク名を割り当てることができます。このネットワーク名は、確実に一意の名前にするために 3 つの要素で構成されています。後ろから見ると、インターネットには sun.com という名前ドメインは 1 つしかありません。その名前ドメイン内では、ID が 515 の UNIX ユーザーは 1 人しかいません。ただし、同一の名前空間にある別のオペレーティングシステム (たとえば VMS) 上のユーザーが偶然同じユーザー ID を持つことはあります。そのような 2 人のユーザーを区別するために、オペレーティングシステム名を追加します。そうすると、一方のユーザーは「UNIX.515@sun.com」となり、他方のユーザーは「VMS.515@sun.com」となります。

最初のフィールドは実際にはオペレーティングシステム名とは別の命名方法で指定します。現在、その命名方法とオペレーティングシステム名とがほぼ 1 対 1 に対応しているだけです。命名方法の標準が確立すれば、最初のフィールドにはオペレーティングシステム名ではなくその標準規約に従った名前を入れます。

AUTH_DES 認証のベリファイア

AUTH_SYS 認証とは違って、AUTH_DES 認証にはベリファイアがあり、サーバーはクライアントの認証証明が正しいかどうかを確認できます。また、その反対方向の認証確認もできます。ベリファイアの主な内容は暗号化されたタイムスタンプです。サーバーは暗号化されたタイムスタンプを解読し、もしそれが実際の時刻に近ければ、クライアントが正しく暗号化したものと考えられます。クライアントが正しくタイムスタンプを暗号化するには、RPC セッションの会話鍵を知っていなければなりません。会話鍵を知っているクライアントならば、本当のクライアントのはずです。

会話鍵は、クライアントが生成して最初の RPC 呼び出しでサーバーに通知する DES [5] 鍵です。会話鍵は、最初のトランザクションで公開鍵方式で暗号化されます。AUTH_DES 認証で使用する公開鍵方式は、192 ビット鍵を使用する Diffie-Hellman [3] 暗号化手法です。この暗号化方式については後に詳しく説明します。

この認証方法が正しく機能するためには、クライアントとサーバーで時刻が一致していなければなりません。ネットワークの時刻同期が保証できないときは、会話を始める前にクライアントの方でサーバーと時刻を合わせることができます。rpcbind の提供する手続き RPCBPROC_GETTIME を使用すれば、現在時刻を取り出すことができます。

サーバーはクライアントのタイムスタンプが正当なものかどうか判定します。2 番目以降のすべてのトランザクションに対して、サーバーは次の 2 つの項目をチェックします。

- タイムスタンプが、同じクライアントからの以前のものより大きい値になっている
- タイムスタンプが失効していない。サーバーの時刻が、クライアントのタイムスタンプにクライアントウィンドウと呼ばれる値を加えた時刻より後ならば、タイムスタンプは失効している。ウィンドウの値は、最初のトランザクションのときにクライアントが暗号化してサーバーに引き渡す。ウィンドウとは、認証証明の存在時間と考えることができる

最初のトランザクションでは、サーバーはタイムスタンプが失効していないことを確認します。さらに、クライアントは最初のトランザクションで、ウィンドウベリファイアと呼ばれるウィンドウの値より 1 少ない値を暗号化して送信します。そうしないと、認証証明がサーバーに棄却されてしまいます。

クライアントはサーバーから返されたベリファイアが正当なものかどうか調べなければなりません。サーバーは、クライアントから受信したタイムスタンプから 1 秒少ない値を暗号化してクライアントに送り返します。クライアントはそれ以外の値を受け取った場合は、それを拒絶します。

ニックネームとクロック同期

最初のトランザクションの後で、サーバーの AUTH_DES 認証サブシステムからクライアントへのベリファイアの中に整数値のニックネームが返されます。クライアントは以降のトランザクションで、ネットワーク名、暗号化された DES 鍵、ウィンドウを毎回渡す代わりに、ニックネームを使用できます。ニックネームは、サーバーが各クライアントのネットワーク名、復号化された DES 鍵、ウィンドウを保存しているテーブルのインデックスのようなものですが、クライアントからは隠されたデータとしてしか使用できません。

クライアントとサーバーのクロックは最初は同期していても、やがて同期が壊れることがあります。その場合、クライアント側の RPC サブシステムは RPC_AUTHERROR を受け取りますので、その時点で再び同期を取る必要があります。

クライアントとサーバーの時刻が同期していてもクライアントが RPC_AUTHERROR を受け取ることがあります。その原因は、サーバーのニックネームテーブルの大きさには制限があり、足りなくなるとエントリが失われることがあるためです。その場合、クライアントは元の認証証明を再送信しなければなりません。サーバーはそれに新たなニックネームを割り当てます。もしもサーバーがクラッシュすると、ニックネームテーブル全体が失われ、全クライアントが元の認証証明を再送信しなければなりません。

DES 認証プロトコル (XDR 言語で記述)

コード例 **B–2**　AUTH_DES 認証プロトコル

```
/*
 * 認証証明には 2 種類あります。1 つは完全なネットワーク名を使用する方法で、

 * もう 1 つはサーバーがクライアントに割り当てたニックネーム (符号なし整数) を
 * 使用する方法です。クライアントからサーバーへの初めてのトランザクションでは
 * 完全なネットワーク名を使用しなければなりません。それに対して、サーバーは
 * クライアントにニックネームを返します。クライアントはそれ以降、サーバーへの
 * トランザクションでニックネームを使用できます。必ずニックネームを
 * 使用しなければならないわけではありませんが、パフォーマンスから考えても、
 * ニックネームを使用する方がよいでしょう。
 */
enum authdes_namekind {
  ADN_FULLNAME = 0,
  ADN_NICKNAME = 1
 };

/*
 * 暗号化した DES データのための 64 ビットブロック
 */
 typedef opaque des_block[8];

/*
 * ユーザーのネットワーク名の最大長
 */
const MAXNETNAMELEN = 255;

/*
 * クライアントのネットワーク名、暗号化された会話鍵、
       * ウィンドウの含まれたフルネーム。
 * ウィンドウは認証証明の存在時間を示します。
 * ベリファイアに示されたタイムスタンプにウィンドウを加えた時刻が
 * 過ぎている場合は、サーバーは要求を無効として許可しません。要求を
```

(続く)

```
 * 再送信しないために、最初のトランザクション以外は、
 * タイムスタンプが以前の値より大きくないと、サーバーはそれを受け入れません。
 * 最初のトランザクションの場合は、サーバーはウィンドウベリファイアが
 * ウィンドウより 1 小さい値であることを確認します。
 */
struct authdes_fullname {
  string name<MAXNETNAMELEN>; /* クライアント名 */
  des_block key;              /* PK で暗号化された会話鍵 */
  unsigned int window;        /* 暗号化されたウィンドウ */
};                             /* 注: PK は公開鍵の略 */

/*
 * 認証証明はフルネームかニックネーム
   */
unionauthdes_credswitch(authdes_namekindadc_namekind){
  case ADN_FULLNAME:
   authdes_fullname adc_fullname;
  case ADN_NICKNAME:
   unsigned int adc_nickname;
};

/*
 * タイムスタンプには、1970 年 1 月 1 日の午前 0 時からの秒数を符号化
 */
struct timestamp {
  unsigned int seconds;      /* 秒数 */
  unsigned int useconds;     /* マイクロ秒 */
};

/*
 * ベリファイア: クライアント側
   */
struct authdes_verf_clnt {
  timestamp adv_timestamp;   /* 暗号化されたタイムスタンプ */
  unsigned int adv_winverf;  /* 暗号化されたウィンドウベリファイア */
};

/*
 * ベリファイア: サーバー側
 * サーバーは、クライアントから渡されたタイムスタンプから 1 秒少ない値を
 * 暗号化して返します。また、クライアントが以降のトランザクションで使用できる
 * ニックネームを暗号化せずに渡します。
 */
struct authdes_verf_svr {
  timestamp adv_timeverf;    /* 暗号化されたベリファイア */
  unsigned int adv_nickname; /* クライアントの新しいニックネーム */
};
```

Diffie-Hellman の暗号化手法

この暗号化手法では、2 つの定数 PROOT と HEXMODULUSを使用します。DES 認証プロトコルでは、この 2 つの定数として次の値を使用します。

```
const PROOT = 3;
const HEXMODULUS = /* 16 進 */
 "d4a0ba0250b6fd2ec626e7efd637df76c716e22d0944b88b";
```

この暗号化手法は次の例で説明するとわかりやすいでしょう。ここに A と B という 2 人の人が互いに暗号化したメッセージを送信するとします。Aと B はそれぞれランダムに秘密鍵を生成し、この鍵は誰にも教えません。秘密鍵をそれぞれ SK(A) と SK(B) とします。また、2 人は公開ディレクトリにそれぞれ公開鍵を示します。公開鍵は次のように計算されます。

```
PK(A) = (PROOT ** SK(A)) mod HEXMODULUS
PK(B) = (PROOT ** SK(B)) mod HEXMODULUS
```

** という記号はべき乗を表します。

ここで A と B は、互いに秘密鍵を知らせ合うことなく 2 人の間の共通鍵 CK(A,B) を求めることができます。

A は次のように計算します。

```
CK(A, B) = (PK(B) ** SK(A)) mod HEXMODULUS
```

B は次のように計算します。

```
CK(A, B) = (PK(A) ** SK(B)) mod HEXMODULUS
```

上の 2 つの式から次の等式が得られます。

```
(PK(B)**SK(A))  mod HEXMODULUS = (PK(A)**SK(B)) mod HEXMODULUS
```

ここで、mod HEXMODULUS という部分を両辺から取り除いてモジュロ計算を省略し、プロセスを簡単にします。

```
PK(B) ** SK(A) = PK(A) ** SK(B)
```

次に、PK(B) を先に B が計算した値で置き換えます。PK(A) も同様に置き換えます。

```
((PROOT ** SK(B)) ** SK(A) = (PROOT ** SK(A)) ** SK(B)
```

この式は次のように書き換えられます。

```
PROOT ** (SK(A) * SK(B)) = PROOT ** (SK(A) * SK(B))
```

共通鍵 CK(A,B) は、プロトコルで使用されるタイムスタンプの暗号化には使用しません。共通鍵は会話鍵の暗号化にだけ使用し、タイムスタンプの暗号化には会話鍵を使用します。これは、共通鍵を使用する回数をできるだけ少なくして、共通鍵が破られないようにするためです。会話時間は比較的短いため、会話鍵の方が破られる心配がずっと少ないからです。

会話鍵は、56 ビットの DES 鍵を使用して暗号化します。共通鍵は 192 ビットですので、共通鍵から次のようにして 56 ビットを選択し、ビット数を減らします。共通鍵から中央の 8 バイトを選択し、各バイトの下位ビットにパリティを加えます。こうして、8 ビットのパリティの付いた 56 ビット鍵が生成されます。

AUTH_KERB 認証プロトコル

AUTH_KERB の SunOS 5.*x* 実装に使用されたカーネルは、Kerberos のコードをオペレーティングシステムのカーネルへコンパイルしないで、kerbd という代理の RPC デーモンを使用します。このデーモンは、以下の 3 つの手続きをエクスポートします。詳細については、kerbd(1M) マニュアルページを参照してください。

1. クライアントによって提供された認証プロトコルを検査するために、サーバー側の RPC が使用する KGETKCRED
2. 主体名、インスタンス、領域が指定されると、暗号化されたチケットおよび DES セッション鍵を返す KSETKCRED
3. UNIX 固有の KGETUCRED。KGETUCRED は、主な名前がサーバーにもわかるユーザー名にマップされると想定し、ユーザーの ID、グループ ID、およびグループリストを返す

Kerberos の内容を的確に説明するには、現在 Kerberos を実装しているサービスであるネットワークファイルシステム (NFS) を例として使用するのが良いでしょう。サーバー s の NFS サービスは、nfs.s という周知の主体名を持つとします。クライアント c の特権ユーザーは、root という一次名と、インスタンス c をもっているとします。AUTH_DES の場合とは異なり、ユーザーのチケット発行用のチケットの期限が切れた場合は、kinit() を再び呼び出さなければならないことに注意してください。Kerberos マウントの NFS サービスは、新しいチケット発行用のチケットを獲得するまで成功しません。

NFS マウント例

この節全体を通して、AUTH_KERB を使用した NFS マウント要求について説明します。マウント要求は、SunOS のルートで実行されるので、ユーザーの識別情報は、root.c. になります。

クライアント c は、マウントするディレクトリのファイルハンドルを獲得するために、サーバー s に MOUNTPROC_MOUNT 要求を実行します。クライアントのマウントプログラムは、ファイルハンドル、mountflavor、時間同期アドレス、サーバーの既知の主体名である nfs.s を、クライアントのカーネルに渡して、NFS マウントシステムコールを実行します。次に、クライアントのカーネルが時間同期ホストでサーバーに接続し、クライアント/ サーバー間の時間差を取得します。

クライアントのカーネルは、次の RPC 呼び出しを行います。(1) チケットおよびセッションキーを獲得するためにのローカルの kerbd への KSETKCRED 呼び出し。(2) フルネームの資格およびベリファイアを使用した、サーバーの NFS サービスへの NFSPROC_GETATTR 呼び出し。サーバーは、呼び出しを受信し、ローカルの kerbd へ KGETKCRED 呼び出しを行ってクライアントのチケットを検査します。

サーバーの kerbd と Kerberos ライブラリは、チケットの暗号を解除し、主体名および DES セッション鍵を他のデータの中に返します。サーバーは、チケットがまだ有効であることをチェックし、セッション鍵を使用して資格、ベリファイアの DES の暗号化された部分を複号化し、ベリファイアが有効であることを検査します。

この時に返される可能性のある Kerberos 認証エラーは、下記のとおりです。

- ベリファイアが無効な場合 (資格にある暗号が解除された win と、ベリファイアでの win +1 は、一致しない)、またはタイムスタンプがウィンドウの範囲外の場合は、AUTH_BADCRED が返される
- やり直しが検出された場合は、AUTH_REJECTEDCRED が返される
- ベリファイアが誤伝送された場合は、AUTH_BADVERF が返される

エラーを受信しない場合、サーバーはクライアントの識別情報をキャッシュに書き込み、NFS 回答に返されるニックネーム (小さい整数) を割り当てります。その時サーバーは、クライアントがサーバーと同じ領域かどうかをチェックします。クライアントがサーバーと同じ領域の場合、サーバーは、KGETUCRED をローカルの kerbd に呼び出して、主体名を UNIX の資格に変換します。変換できない場合、ユーザーは匿名であるとマークされます。サーバーは、ファイルシステムのエクスポート情報に対するこれらの資格を検査します。次の 3 つのケースを考えてください。

1. KGETUCRED 呼び出しが失敗し、匿名の要求が受け入れられた場合、匿名のユーザーに UNIX 資格が割り当てられます。

2. KGETUCRED 呼び出しが失敗し、匿名の要求が受け入れられない場合、NFS 呼び出しは失敗し、AUTH_TOOWEAK が返されます。
3. KGETUCRED 呼び出しが成功する場合は、資格が割り当てられ、その後にルートのアクセス権のチェックも含む、正常な保護検査が行われます。

次に、サーバーが、ニックネームおよびサーバーのベリファイアを組み込んで NFS 回答を送信します。クライアントは回答を受信し、ベリファイアの複号化と妥当性検査を行い、今後の呼び出しのためにニックネームを格納します。クライアントがサーバーに 2 番目の NFS 呼び出しを行うと、先にサーバーに書込まれた呼び出しが繰り返されます。クライアントのカーネルが、以前に記述されたニックネーム資格およびベリファイアを使用して、サーバーの NFS サービスに NFSPROC_STATVFS 呼び出しを行います。サーバーは呼び出しを受信し、ニックネームの妥当性検査を行います。これが範囲外であれば、エラー AUTH_BADCRED を返します。サーバーは、獲得したばかりのセッション鍵を使用して、ベリファイアの DES の暗号化された部分を複号化し、ベリファイアの妥当性検査を行います。

この時に返される可能性のある Kerberos 認証エラーは、次のとおりです。

- タイムスタンプが無効で、やり直しが検出されるか、タイムスタンプがウィンドウの範囲外の場合は、AUTH_REJECTEDVERF を返します。
- サービスチケットの期限が切れると、AUTH_TIMEEXPIRE を返します。

エラーが受信されない場合、サーバーは、ニックネームを使用して、呼び出し側の UNIX 資格を検出します。それから、サーバーはファイルシステムのエクスポート情報に対するこれらの資格を検査し、ニックネームおよびサーバーのベリファイアを組み込んだ NFS 回答を送信します。クライアントは回答を受信し、ベリファイアの複号化および妥当性検査を行い、これからの呼び出しのためにニックネームを格納します。最後に、クライアントのNFS マウントシステムコールが返り、要求が終了します。

KERB 認証プロトコル (XDR 言語で記述)

コード例 B–3 (AUTH_KERB) は、コード例 B–2 で示された AUTH_DES と似ています。両者の違いに注意してください。

コード例 **B–3**　Kerb 認証プロトコル

```
#define AUTH_KERB 4
/*
 * 資格には 2 種類あります。1 つはクライアントが (前もって暗号化された)
```

(続く)

```
 * Kerberos チケット送信する資格で、もう 1 つはクライアントがサーバーに指定され
 * た「ニックネーム」(符号なしの整数のみ) を使用する資格です。クライアントは、
 * サーバーへの初めてのトランザクションでは、フルネームを使用しなければなりませ
 * ん。それに対してサーバーはクライアントにニックネームを返します。クライアントは
 * それ以降、サーバーへのトランザクションでニックネームを使用できます。
 * (チケットの期限が切れるまで)。  必ずニックネームを使用しなけれ
 * ばならないわけではありませんが、パフォーマンスから考えても、ニック
 * ネームを使用する方がよいでしょう。
 */
enum authkerb_namekind {
  AKN_FULLNAME = 0,
  AKN_NICKNAME = 1
 };


/*
 * フルネームには、暗号化されたサービスチケットと有効期限が含まれます。

 * 実際、この有効期限は、資格の有効期限と同じです。ベリファイアの
 * タイムスタンプに示されている時間に有効期限を加えた時間がすでに経過すると、
 * サーバーは要求を満了にして、もはや承諾しません。要求がやり直されないようにす
 * るため、サーバーは、はじめのトランザクション以外の場合に、タイムスタンプが以
 * 前のタイムスタンプより大きくないかどうかをチェックする必要があります。はじ
 * めのトランザクションでは、サーバーはウィンドウベリファイアがウィンドウより小
 * さいことを検査します。
 */
struct authkerb_fullname {
  KTEXT_ST ticket;               /* Kerberos サービスチケット */
  unsigned long window;         /* 暗号化されたウィンドウ */
};
/*
 * 資格はフルネームかニックネーム
 */
union authkerb_credswitch(authkerb_namekind akc_namekind){
  case AKN_FULLNAME:
   authkerb_fullname akc_fullname;
  case AKN_NICKNAME:
   unsigned long akc_nickname;
};

/*
 * タイムスタンプには、1970 年 1 月 1 日の午前 0 時からの秒数を符号化

 */
struct timestamp {
  unsigned long seconds;       /* 秒 */
  unsigned long useconds;      /* マイクロ秒 */
};
/*
 * ベリファイア:クライアント側

 */
```

(続く)

続き

```
struct authkerb_verf_clnt {
  timestamp akv_timestamp;   /* 暗号化されたタイムスタンプ */
  unsigned long akv_winverf;  /* 暗号化されたウィンドウベリファイア */
};

/*
ベリファイア:サーバー側
 * クライアントによりサーバーは、タイムスタンプ(暗号化)が与えられた。
 * また、サーバーは、クライアントがニックネームを今後の(暗号化されて
 * いない)トランザクションで使用するように指定します。
 */
struct authkerb_verf_svr {
  timestamp akv_timeverf;    /* 暗号化されたベリファイア */
  unsigned long akv_nickname; /* クライアントの新しいニックネーム */
};
```

RPC 言語の仕様

XDR データ型を形式言語で記述する必要があるのと同様に、これらの XDR データ型に対して作用する手続きも、形式言語で記述する必要があります。XDR 言語の拡張版である RPC 言語は、XDR 言語を形式言語で記述するための言語です。次に、RPC 言語についての例を示します。

RPC 言語で記述されたサービスの例

コード例 B–4 は、単純な `ping` プログラムの仕様を示します。

コード例 **B–4**　RPC 言語を使用した `ping` サービス

```
/*
 * 単純な ping プログラム
 */
program PING_PROG {
  version PING_VERS_PINGBACK {
   void
```

```
    PINGPROC_NULL(void) = 0;
    /*
    * 呼び出し側の ping は、往復時間をミリ秒で返します。
    * オペレーションがタイムアウトの場合は、-1 を返します。
    */
   int
    PINGPROC_PINGBACK(void) = 1;
    /* void - 上記は呼び出しへの引き数 */
   } = 2;
/*
 * オリジナルのバージョン
 */
  version PING_VERS_ORIG {
   void
   PINGPROC_NULL(void) = 0;
  } = 1;
} = 200000;
const PING_VERS = 2; /* 最新バージョン */
```

記述された最初のバージョンは、2 つの手続き、PINGPROC_NULL、および PINGPROC_PINGBACK が組み込まれた PING_VERS_PINGBACKです。

PINGPROC_NULL は引数を必要とせず、結果も返しませんが、クライアントとサーバー間の往復時間を計算するときになどに便利です。規則によると、RPCプログラムの手続き 0 はすべて同じセマンティクスを持つことになっているので、認証は必要ありません。

2 番目の手続きは、オペレーションにかかった合計時間を (マイクロ秒で) 返します。

次のバージョンである PING_VERS_ORIG は、プロトコルのオリジナルのバージョンで、PINGPROC_PINGBACK 手続きは含まれません。PING_VERS_ORIG は、古いクライアントプログラムと互換性を持たせる場合に便利ですが、このプログラムが完成すると、プロトコルから完全に削除されることがあります。

RPCL 構文

RPC 言語 (RPCL) は C に似ています。この節では、例を示しながら RPC 言語の構文を説明します。また、出力ヘッダーファイルで、RPC 型定義および XDR 型定義を C 型定義にコンパイルする方法についても説明します。

RPC 言語ファイルは次の一連の定義から構成されています。

```
definition-list:
   definition;
   definition; definition-list
```

定義には、6 つの型があります。

```
definition:
      enum-definition
   const-definition
   typedef-definition
   struct-definition
   union-definition
   program-definition
```

定義は宣言と同じではありません。1 つまたは一連のデータ要素の型定義以外の定義によっては領域を割り当てることはできません。これは、変数は定義するだけでは十分でなく、宣言もする必要があることを意味しています。

RPC 言語は、表 B–2 で追加された定義以外は、XDR 言語と同じです。

表 B–2 RPC 言語定義

用語	定義
プログラム定義	`program program-ident {version-list} = value`
バージョンリスト	`version;`
	`version; version-list`
バージョン	`version version-ident {procedure-list} = value`
手続きリスト	`procedure;`
	`procedure; procedure-list`
手続き	`type-ident procedure-ident (type-ident) = value`

- `program`、`version` のキーワードが追加されますが、識別子としては使用できない
- バージョン名およびバージョン番号は、プログラム定義範囲内で一度しか指定できない
- 手続き名および手続き番号は、バージョン定義内で一度しか指定できまない

- プログラム識別子は、定数および型識別子と同じ名前空間にある
- 符号なしの定数だけは、プログラム、バージョン、および手続きに割り当てられる

列挙法

RPC/XDR 列挙法の構文は、C 列挙法と同じです。

```
enum-definition:
   "enum" enum-ident "{"
   enum-value-list
  "}"
enum-value-list:
  enum-value
  enum-value "," enum-value-list
enum-value:
  enum-value-ident
  enum-value-ident "=" value
```

次に、コンパイルされる XDR enum および C enum の例を示します。

```
enum colortype {                enum colortype {
  RED = 0,                        RED = 0,
  GREEN = 1,        -->           GREEN = 1,
  BLUE = 2                        BLUE = 2,
};                                };
                                  typedef enum colortype colortype;
```

定数

シンボリック定数は、整数の定数が使用されればどこでも使用できます。たとえば配列サイズ仕様で使用すると、次のようになります。

```
   const-definition:
   const const-ident = integer
```

次の例では定数 DOZEN を 12 に定義します。

```
const DOZEN = 12; --> #define DOZEN 12
```

型定義

XDR typedef の構文は、C typedef と同じです。

```
typedef-definition:
      typedef declaration
```

この例では、最大 255 文字のファイル名の文字列を宣言するために使用する fname_type を定義します。

```
typedef string fname_type<255>; --> typedef char *fname_type;
```

宣言

XDR には、4 種類の宣言があります。これらの宣言は、struct または typedef の中に記述する必要があります。単独では使用できません。

```
declaration:
      simple-declaration
   fixed-array-declaration
   variable-array-declaration
   pointer-declaration
```

単純宣言

単純宣言は、C 単純宣言に似ています。

```
simple-declaration:
     type-ident variable-ident
```

次に例を示します。

```
   colortype color; --> colortype color;
```

固定長配列宣言

固定長配列宣言は、C 配列宣言に似ています。

```
fixed-array-declaration:
      type-ident variable-ident [value]
```

次に例を示します。

```
colortype palette[8]; --> colortype palette[8];
```

変数宣言を型宣言と混同するプログラマがよくいます。rpcgen は、変数宣言をサポートしないことに注意してください。この例は、コンパイルされないプログラムです。

```
int data[10];
program P {
   version V {
      int PROC(data) = 1;
  } = 1;
} = 0x200000;
```

上記の例は、変数宣言なのでコンパイルされません。

```
int data[10]
```

代わりに以下を使用します。

```
typedef int data[10];
```

または

```
struct data {int dummy [10]};
```

可変長配列宣言

可変長配列宣言は、C 構文とはまったく異なります。XDR 言語は、構文を使用しないで山括弧でくくります。

```
variable-array-declaration:
      type-ident variable-ident <value>
      type-ident variable-ident < >
```

最大サイズは山括弧内で指定します。配列のサイズにこだわらない場合は、サイズを省略することができます。

```
int heights<12>; /* 最高 12 項目 */
int widths<>; /* 項目数に制限なし */
```

可変長配列が C 構文とまったく異なるので、これらの宣言はコンパイルされて struct 宣言になります。たとえば、heights 宣言はコンパイルされて struct になります。

```
struct {
   u_int heights_len;                /* # 配列の項目番号 */
  int *heights_val;                 /* 配列へのポインタ */
} heights;
```

配列の項目の番号は、_len 構成要素に、配列へのポインタは _val 構成要素に格納されます。各構成要素名のはじめの部分は、宣言された XDR 変数名 (heights) と同じです。

ポインタ宣言

XDR には、C とまったく同じポインタ宣言が作成されます。アドレスポインタはネットワーク上で送信されませんが。その代わり、XDR ポインタは、リストおよびツリーなどの再帰的なデータ型を送信するのに便利です。この型は、XDR 言語では「ポインタ」ではなく、「オプション・データ」と呼ばれます。

```
ポインタ宣言:
  type-ident *variable-ident
```

次に例を示します。

```
listitem *next; --> listitem *next;
```

構造体

RPC/XDR `struct` は C `struct` とほぼ同様に宣言されます。RPC/XDR `struct` の宣言は次のようになります。

```
struct-definition:
  struct struct-ident "{"
      declaration-list
  "}"


declaration-list：
     declaration ";"
    declaration ";" declaration-list
```

次の左の部分は二次元の座標の XDR 構造体の例で、右の部分はそれを C 言語にコンパイルした構造体です。

```
struct coord {                  struct coord {
  int x;             -->            int x;
  int y;                            int y;
};                                };
                                  typedef struct coord coord;
```

出力は、出力の末端部で追加された `typedef` 以外は入力と同じです。これによって、項目を宣言する時に、`struct coord` のかわりに `coord` を使用することができます。

共用体

XDR 共用体は、識別された共用体なので、C の共用体とは似ていません。どちらかというと、Pascal 変数レコードに似ています。

```
union-definition:
 "union" union-ident "switch" "("simple declaration")" "{"
   case-list
  "}"
case-list:
  "case" value ":" declaration ";"
  "case" value ":" declaration ";" case-list
  "default" ":" declaration ";"
```

以下は、「読み取りデータ」操作の結果として返された型の例です。エラーがない場合は、データのブロックが返されます。エラーがある場合は、何も返されません。

```
union read_result switch (int errno) {
  case 0:
   opaque data[1024];
  default:
   void;
  };
```

C 言語にコンパイルされると次のようになります。

```
struct read_result {
  int errno;
  union {
   char data[1024];
  } read_result_u;
};
typedef struct read_result read_result;
```

出力 struct の共用体構成要素の名前は、接尾語 _u を除いて型名前と同じです。

プログラム

RPC プログラムは、次の構文を使用して宣言します

```
program-definition:
  "program" program-ident "{"
   version-list
  "}" "=" value;
version-list:
  version ";"
  version ";" version-list
version:
  "version" version-ident "{"
   procedure-list
  "}" "=" value;
procedure-list:
  procedure ";"
  procedure ";" procedure-list
procedure:
  type-ident procedure-ident "(" type-ident ")" "=" value;
```

–N オプションが指定されると、rpcgen は次の構文も認識できます。

```
手続き:
  type-ident procedure-ident "(" type-ident-list ")" "=" value;
type-ident-list:
  type-ident
  type-ident "," type-ident-list
```

次に例を示します。

```
/*
 * time.x: 時間を取得、または設定します。
 * 時間は、1970 年 1 月 1 日 0:00 から経過した秒数で表されます。
 */
program TIMEPROG {
   version TIMEVERS {
      unsigned int TIMEGET(void) = 1;
   void TIMESET(unsigned) = 2;
  } = 1;
} = 0x20000044;
```

void という引数の型は、引数が引き渡されないことを意味しています。

このファイルは、コンパイルされると、出力ヘッダーファイル内で以下の #define 文になります。

```
#define TIMEPROG 0x20000044
#define TIMEVERS 1
#define TIMEGET 1
#define TIMESET 2
```

RPC 言語規則の例外

RPC 言語規則には、例外があります。

C 形式 モード

この節では、rpcgen の C 形式モードの機能について説明します。これらの機能は、void 引数の引き渡しに関してかかわっています。値が void の場合、引数が引き渡される必要はありません。

ブール値

C には組み込み型のブール型はありません。ただし、RPC ライブラリは、TRUE または FALSE のうちいずれかの bool_t と呼ばれるブール値を使用します。XDR 言語で型 bool として宣言されたパラメータは、コンパイルされると、出力ヘッダーファイルで bool_t になります。

次に例を示します。

```
bool married; --> bool_t married;
```

文字列

C 言語は組み込み型の文字列型ではありませんが、代わりに null で終了する char * 規則を使用します。C では、文字列は通常 null で終了する単一配列であるとみなされます。

XDR 言語では、string キーワードを使用して文字列が宣言されて、出力ヘッダーファイルで char * 型にコンパイルされます。山括弧でくくられた最大サイズは、文字列で使用できる最大文字数を指定します (NULL文字をカウントしません)。 任意の文字列のサイズを表す場合は、最大サイズを省略することができます。

次に例を示します。

```
string name<32>;   --> char *name;
string longname<>; --> char *longname;
```

注 - NULL 文字列は引き渡されません。ただし、0 長の文字列 (つまりターミネータだけ、または NULL バイト) は引き渡されます。

隠されたデータ

隠されたデータは、未入力のデータ、つまり任意のバイトのシーケンスを記述するために、XDR で使用されます。隠されたデータは、固定長または可変長配列として宣言できます。次に例を示します。

```
opaque diskblock[512]; --> char diskblock[512];
opaque filedata<1024>; --> struct {
                             u_int filedata_len;
                             char *filedata_val;
                           } filedata;
```

Voids

void 宣言では、変数を指定できません。宣言は、void だけで、Void 宣言が使用されるのは、共用体およびプログラム定義 (遠隔手続きの引数または結果として、引数が引き渡されなかったなどで使用される) の 2 箇所だけです。

rpcbind プロトコル

rpcbind は RPC のプログラム番号とバージョン番号を汎用アドレスにマップし、遠隔プログラムの動的結合を可能にします。

rpcbind はそれをサポートしているトランスポートのよく知られたアドレスに結合しています。他のプログラムは、動的に割り当てられたアドレスを rpcbind で登録します。rpcbind は、それらのアドレスを一般に使用できるようにします。汎用アドレスとは、トランスポートに依存したアドレスで、文字列で表現されています。汎用アドレスは、各トランスポートのアドレス管理者が定義します。

rpcbind はブロードキャスト RPC にも利用できます。RPC プログラムは異なるマシン上で異なるアドレスを持っているため、これらすべてのプログラムに直接ブロードキャストする方法はありません。ところが、rpcbind のアドレスはわかっているため、クライアントが特定のプログラムにブロードキャストするには、送信先マシン上の rpcbind プロセスにメッセージを送ります。rpcbind はブロードキャストメッセージを取り出し、クライアントが指定したローカルサービスを呼び出します。rpcbind はローカルサービスから応答を受け取ると、それをクライアントに渡します。

コード例 **B–5**　rpcbind プロトコル仕様 (RPC 言語で記述)

```
/*
 * rpcb_prot.x
 * RPCBIND プロトコルを RPC 言語で記述
 */
/*
 * (プログラム、バージョン、ネットワーク ID) を汎用アドレスにマップ
 */
struct rpcb {
 rpcproc-t r_prog;                /* プログラム番号 */
 rpcvers-t r_vers;               /* バージョン番号 */
 string r_netid<>;               /* ネットワーク ID */
 string r_addr<>;                /* 汎用アドレス */
 string r_owner<>;               /* サービスの所有者 */ };
/* マップリスト */
struct rpcblist {
 rpcb rpcb_map;
 struct rpcblist *rpcb_next;
};

/* 遠隔呼び出しの引数 */
struct rpcb_rmtcallargs {
 rpcprog_t prog;             /* プログラム番号 */
 rpcvers_t vers;             /* バージョン番号 */
 rpcproc_t proc;             /* 手続き番号 */
 opaque args<>;                  /* 引数 */
};

/* 遠隔呼び出しの戻り値 */
struct rpcb_rmtcallres {
 string addr<>;                  /* 遠隔汎用アドレス */
 opaque results<>;               /* 結果 */
};
```

(続く)

```
/*
 * rpcb_entry には、特定トランスポート上のサービスアドレスと、
 * それに関連した netconfig 情報がマージされて入っています。
 * RPCBPROC_GETADDRLIST は、rpcb_entry のリストを返します。
 * r_nc_* フィールドで使用する値については、netconfig.h を参照してください。
 */
struct rpcb_entry {
 string          r_maddr<>;      /* マージされたサービスアドレス */
 string          r_nc_netid<>;   /* netid フィールド */
 unsigned int   r_nc_semantics; /* トランスポートのセマンティクス */
 string          r_nc_protofmly<>; /* プロトコルファミリ */
 string          r_nc_proto<>;   /* プロトコル名 */
};

/* サービスがサポートしているアドレスリスト */
struct rpcb_entry_list {
 rpcb_entry rpcb_entry_map;
 struct rpcb_entry_list *rpcb_entry_next;
};

typedef rpcb_entry_list *rpcb_entry_list_ptr;

/* rpcbind 統計情報 */
const rpcb_highproc_2 = RPCBPROC_CALLIT;
const rpcb_highproc_3 = RPCBPROC_TADDR2UADDR;
const rpcb_highproc_4 = RPCBPROC_GETSTAT;
const RPCBSTAT_HIGHPROC = 13;  /* rpcbind V4 の手続き数 + 1 */
const RPCBVERS_STAT = 3;  /* rpcbind V2,V3,V4 だけのために提供 */
const RPCBVERS_4_STAT = 2;
const RPCBVERS_3_STAT = 1;
const RPCBVERS_2_STAT = 0;

/* getport と getaddr の全状態のリンクリスト */
struct rpcbs_addrlist {
 rpcprog_t prog;
 rpcvers_t vers;
 int success;
 int failure;
 string netid<>;
 struct rpcbs_addrlist *next;
};

/* rmtcall の全状態のリンクリスト */
struct rpcbs_rmtcalllist {
 rpcprog_t prog;
 rpcvers_t vers;
 rpcproc_t proc;
 int success;
 int failure;
 int indirect;   /* callit か indirect かを示す */
 string netid<>;
 struct rpcbs_rmtcalllist *next;
```

(続く)

```
};

typedef int rpcbs_proc[RPCBSTAT_HIGHPROC];
typedef rpcbs_addrlist *rpcbs_addrlist_ptr;
typedef rpcbs_rmtcalllist *rpcbs_rmtcalllist_ptr;

struct rpcb_stat {
 rpcbs_proc              info;
 int                     setinfo;
 int                     unsetinfo;
 rpcbs_addrlist_ptr      addrinfo;
 rpcbs_rmtcalllist_ptr   rmtinfo;
};

/*
 * 監視される rpcbind の各バージョンに対して rpcb_stat 構造体が
 * 1 つずつ返されます。
 */
typedef rpcb_stat rpcb_stat_byvers[RPCBVERS_STAT];
/* rpcbind 手続き */
program RPCBPROG {
 version RPCBVERS {
  void
  RPCBPROC_NULL(void) = 0;

  /*
   * この手続きは、[r_prog, r_vers, r_addr, r_owner, r_netid] の
   * 組み合わせを登録します。rpcbind サーバーは、セキュリティ上の理由
   * から、この手続きへの要求をループバックトランスポートだけで受け入れ
   * ます。 正常終了では TRUE、異常終了では FALSE が返されます。
   */
  bool
  RPCBPROC_SET(rpcb) = 1;

  /*
   *この手続きは、[r_prog, r_vers, r_owner, r_netid] の

   * 組み合わせの登録を解除します。
   * vers がゼロの場合は、全バージョンを登録解除します。
   * この手続きへの要求は、ループバックトランスポートだけで受け入れます。
   * 正常終了では TRUE、異常終了では FALSE が返されます。
   */
  bool
  RPCBPROC_UNSET(rpcb) = 2;

  /*
   * この手続きは、[r_prog, r_vers, r_netid] の組み合わせが登録さ
   * れている汎用アドレスを返します。r_addr が指定されていれば、
   * 汎用アドレスが r_addr にマージされて返されます。r_owner は無視
   * します。 異常終了の場合は、FALSE が返されます。
   */
  string
  RPCBPROC_GETADDR(rpcb) = 3;
```

(続く)

```
/* この手続きは、全マップのリストを返します。 */

rpcblist
RPCBPROC_DUMP(void) = 4;

/*
 * この手続きは、遠隔マシン上の手続きを呼び出します。
 * 手続きが登録されていない場合は何もしません。
 * すなわち、エラー情報も返しません。
 */
rpcb_rmtcallres
RPCBPROC_CALLIT(rpcb_rmtcallargs) = 5;

/*
 * この手続きは、rpcbind サーバーシステムの時刻を返します。
 */
unsigned int
RPCBPROC_GETTIME(void) = 6;

struct netbuf
RPCBPROC_UADDR2TADDR(string) = 7;

string
RPCBPROC_TADDR2UADDR(struct netbuf) = 8;

} = 3;
version RPCBVERS4 {
bool
RPCBPROC_SET(rpcb) = 1;

bool
RPCBPROC_UNSET(rpcb) = 2;

string
RPCBPROC_GETADDR(rpcb) = 3;

rpcblist_ptr
RPCBPROC_DUMP(void) = 4;

/*
 * 注: RPCBROC_BCAST と CALLIT の機能は同じです。
 * 新たな名前を付けた目的は、ブロードキャスト RPC にはこの手続きを
 * 使用し、間接呼び出しには RPCBPROC_INDIRECT を
 * 使用することを示すためです。
 */
rpcb_rmtcallres
RPCBPROC_BCAST(rpcb_rmtcallargs) = RPCBPROC_CALLIT;

unsigned int
RPCBPROC_GETTIME(void) = 6;

struct netbuf
```

(続く)

```
  RPCBPROC_UADDR2TADDR(string) = 7;

  string
  RPCBPROC_TADDR2UADDR(struct netbuf) = 8;

/*
  * この手続きは、RPCBPROC_GETADDR と同じですが、指定された
 * バージョン番号がない場合はアドレスを返さない点が異なります。
 */
  string
  RPCBPROC_GETVERSADDR(rpcb) = 9;

  /*
   * この手続きは、遠隔マシン上の手続きを呼び出します。
   * 手続きが登録されていない場合は、エラー情報を
   * 返します。
   */
  rpcb_rmtcallres
  RPCBPROC_INDIRECT(rpcb_rmtcallargs) = 10;

  /*
   * この手続きは、RPCBPROC_GETADDR と同じですが、(prog, vers) の
   * 組み合わせで登録されているアドレスのリストを返す点が異なります。
   */
  rpcb_entry_list_ptr
  RPCBPROC_GETADDRLIST(rpcb) = 11;

  /*
   * この手続きは、rpcbind サーバーのアクティビティに関する統計情報を返します。
   */
  rpcb_stat_byvers
  RPCBPROC_GETSTAT(void) = 12;
 } = 4;
} = 100000;
```

rpcbind の操作

rpcbind にアクセスするには、使用するトランスポートごとに割り当てられているアドレスを使用します。たとえば TCP/IP と UDP/IP の場合は、ポート番号 111 が割り当てられています。各トランスポートには、このようによく知られているアドレスがあります。以下には、rpcbind がサポートしている各手続きを説明します。

RPCBPROC_NULL

この手続きは何もしない手続きです。習慣的にどのプログラムでも、手続き 0 は引数も戻り値もない手続きとします。

RPCBPROC_SET

マシン上でプログラムが初めて使用可能になるときは、そのマシンで実行されている rpcbind に自分自身を登録します。登録時にプログラムが渡すのは、プログラム番号 *prog*、バージョン番号 *vers*、ネットワーク ID *netid*、サービス要求を待つ汎用アドレス *uaddr* です。

この手続きは、プログラムのマッピングに成功すれば TRUE、失敗すれば FALSE のブール値を返します。指定された (*prog*、 *vers*、 *netid*) の組み合わせで既にマップされたものがあれば、新たなマップは行いません。

netid と *uaddr* はどちらも NULL にしてはいけません。また、*netid* には、呼び出しを行うマシン上のネットワーク ID が正しく指定されていなければなりません。

RPCBPROC_UNSET

プログラムが使用できなくなった場合は、同一マシン上の rpcbind で自分自身を登録解除する必要があります。

この手続きの引数と戻り値は、RPCBPROC_SET と同じです。(*prog*、*vers*、*netid*) の組み合わせと uaddr のマッピングが削除されます。

netid が NULL の場合は、(*prog*、*vers*、*) 組み合わせとそれに対応する汎用アドレスのマッピングがすべて削除されます。サービスの登録解除は、サービスの所有者かスーパーユーザーだけが実行できます。

RPCBPROC_GETADDR

プログラム番号 *prog*、バージョン番号 *vers*、ネットワークID *netid* を指定してこの手続きを呼び出すと、そのプログラムが呼び出し要求を待っている汎用アドレスが返されます。

引数の netid フィールドは無視され、要求が到着するトランスポートの netid から取り出します。

RPCBPROC_DUMP

この手続きは、rpcbind データベースの全エントリのリストを返します。

この手続きには引数がなく、戻り値は、プログラム、バージョン、ネットワーク ID、汎用アドレス、のリストです。この手続きを呼び出すときは、データグラムトランスポートではなくストリームトランスポートを使用します。これは、大量のデータが返されるのを回避するためです。

RPCBPROC_CALLIT

この手続きを使用すると、汎用アドレスがわからなくても同一マシン上にある遠隔手続きを呼び出すことができます。この手続きの目的は、rpcbind の汎用アドレスを通して任意の遠隔プログラムにブロードキャストできるようにすることです。

パラメータ prog、vers、proc、args_ptr にはそれぞれプログラム番号、バージョン番号、手続き番号、遠隔手続きへの引数を指定します

注 - この手続きは正常終了の場合は応答しますが、異常終了の場合は一切応答しません。

この手続きからは、遠隔プログラムの汎用アドレスと、遠隔手続きからの戻り値が返されます。

RPCBPROC_GETTIME

この手続きは、自分のマシンのローカル時刻を、1970 年 1 月 1 日午前 0 時からの秒数で返します。

RPCBPROC_UADDR2TADDR

この手続きは、汎用アドレスをトランスポート (netbuf) アドレスに変換します。この手続きは、uaddr2taddr() (netdir(3N) マニュアルページを参照) と同じ機能を持ちます。名前 - アドレス変換のライブラリモジュールとリンクできないプロセスだけが、この手続きを使用します。

RPCBPROC_TADDR2UADDR

この手続きは、トランスポート (netbuf) アドレスを汎用アドレスに変換します。この手続きは、taddr2uaddr() (netdir(3N) マニュアルページを参照) と同じ機能を持ちます。名前 - アドレス変換のライブラリモジュールとリンクできないプロセスだけが、この手続きを使用します。

rpcbind のバージョン 4

rpcbind のバージョン 4 では、これまでに説明した手続きのほかに、次に示す手続きが追加されています。

RPCBPROC_BCAST

この手続きは、バージョン 3 の RPCBPROC_CALLIT 手続きと同じです。新たな名前を付けたのは、この手続きはブロードキャスト RPC だけに使用することを示すためです。これに対して、次のテキストで定義する RPCBPROC_INDIRECT は、間接 RPC 呼び出しだけに使用します。

RPCBPROC_GETVERSADDR

この手続きは、RPCBPROC_GETADDR に似ています。異なる点は、rpcb 構造体の r_vers フィールドで目的のバージョンを指定できることです。そのバージョンが登録されていない場合、アドレスは返されません。

RPCBPROC_INDIRECT

この手続きは、RPCBPROC_CALLIT に似ていますが、エラーが起こった場合 (たとえば、呼び出すプログラムがシステムに登録されていない場合) にエラー情報を返す点が異なります。この手続きはブロードキャスト RPC には使用できません。間接 RPC 呼び出しだけに使用します。

RPCBPROC_GETADDRLIST

この手続きは、指定された rpcb エントリのアドレスリストを返します。クライアントはそのリストを使用して、サーバーと通信するための代替トランスポートを調べることができます。

RPCBPROC_GETSTAT

この手続きは、rpcbind サーバーのアクティビティに関する統計情報を返します。統計情報には、サーバーが受信した要求の種類と回数が示されます。

注 - RPCBPROC_SET と RPCBPROC_UNSET 以外の手続きはすべて、rpcbind が実行されているマシンとは別のマシン上のクライアントから呼び出すことができます。rpcbind は、RPCPROC_SET と RPCBPROC_UNSET の要求だけはループバックトランスポートからでないと受け入れません。

参考文献

この付録、リファレンスで説明した技術内容やアーキテクチャに関する参考文献には、次のものがあります。

1. 『Implementing Remote Procedure Calls』 Birrel, Andrew D.& Nelson, Bruce Jay 著、XEROX 社 CSL-83-7、1983 年 10 月
2. 『VMTP: Versatile Message Transaction Protocol, Preliminary Version 0.3』 Cheriton, D. 著、Stanford 大学、1987 年 1 月
3. 『New Direction in Cryptography, IEEE Transactions on Information Theory 』 Diffie & Hellman 著、Transactions on Information Theory IT-22、1976 年 11 月
4. 『Time Server, RFC 738』Harrenstien, K. 著、Information Sciences Institute、1977 年 10 月
5. 『Data Encryption Standard』 National Bureau of Standards、Federal Information Processing Standards Publication Publication 46、1977 年 1 月
6. 『Transmission Control Protocol - DARPA Internet Program Protocol Specification, RFC 793』Postel, J. 著、Information Sciences Institute、1981 年 9 月
7. 『User Datagram Protocol, RFC 768』Postel, J. 著、Information Sciences Institute、1980 年 8 月

付録 **C**

XDR プロトコル仕様

この付録では、XDR プロトコル言語の仕様について説明します。

XDR プロトコルの概要

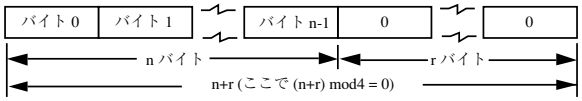
外部データ表現 (external data representation: XDR) は、データの記述と符号化の標準規約です。XDR プロトコルは、異なるコンピュータアーキテクチャ間のデータ伝送に利用できます。これまで、Sun™ Workstation™、VAX 、IBM PC、Cray など種々のマシン間のデータ通信に使用されてきました。XDR は、ISO の参照モデルのプレゼンテーション層 (第 6 層) に対応するもので、X.409 「ISO 抽象構文表記」におおむね従っています。XDR と X.409 との一番大きな違いは、XDR が暗黙的データ型を使用するのに対して、X.409 は明示的データ型を使用する点です。

XDR では、言語を使用してデータ形式を記述しますが、この言語はプログラミング言語ではないので、データの記述のためにだけしか使用できません。XDR 言語を使用すると、複雑なデータ形式も簡潔に表現できます。XDR 言語は C 言語に似ています。RPC や NFS のようなプロトコルでは、XDR でデータ形式を記述しています。

XDR 標準規約では、バイト (オクテット) は移植可能な 8 ビットデータとみなしています。

グラフィックボックス表現

この章では、データの説明や比較のときに、グラフィックボックス表現を使用します。ほとんどの場合、各ボックスが 1 バイトを表します。各バイトは `0` ～ `n-1` で番号付けされます。データは、バイト `m` がバイト `m+1` の直前に位置するという関係が保たれるバイトストリームで読み書きされます。データ項目はすべて 4 バイト (32 ビット) の倍数で表現されます。`n` バイトのデータの後には、0 ～ 3 個の余分なゼロバイト `r` が付加されて、全体のバイト数が 4 の倍数になるように調整されます。ボックス間にある省略記号は、1 バイト以上の追加が必要が、またはまったく必要でないことを表します。次にその例を示します。



基本ブロックサイズ

XDR のブロックサイズの選択はさまざまな条件の兼ね合いで決まります。2 のような小さな値を選択して符号化データを小さくすると、そのようなデータ境界を使用しないマシンでは整合の問題が起こります。8 のような大きな値にすると、事実上すべてのマシンでデータ整合が可能になりますが、符号化データが大きくなり過ぎます。妥協案として 4 が選ばれました。4 は適度な大きさでほとんどのアーキテクチャに対応できますが、8 バイト境界の Cray などのマシンは例外です。

ただし Cray コンピュータで標準 XDR が使用できないわけではありません。各データ項目のオーバヘッドが、4 バイト (32 ビット) アーキテクチャのマシンより大きくなるという意味です。4 という値は符号化データを妥当なサイズに押さえる意味でも適当な値です。

どのマシンでも同じデータは同じ値に符号化されなければ、符号化データを比較したりチェックサムを取ったりできません。したがって、可変長データの最後は、ゼロデータでパディングしなければなりません。

XDR のデータ型宣言

以降の各節は次の部分に分れています。

- XDR 標準規約で定義されているデータ型を説明します。
- XDR 言語でどのようにデータ型を宣言するかを示します。
- 符号化方法を図示します。

XDR 言語で使用できる各データ型の宣言方法を示します。大小記号による括弧 (< と >) は可変長のデータシーケンスを示し、角括弧 [と])は固定長のデータシーケンスを示します。n、m、r は整数を表します。XDR 言語の詳細仕様については、296ページの「XDR 言語仕様」の節を参照してください。

いくつかのデータ型については、具体的なデータ記述例も示します。より詳しいデータ記述例については、299ページの「XDR データ記述 」を参照してください。

符号付き整数

説明

XDR の符号付き整数は、[-2147483648, 2147483647] の範囲の整数が符号化された 32 ビットデータです。整数は 2 の補数で表されます。最上位バイト (MSB) と最下位バイト (LSB) はそれぞれバイト 0 とバイト 3 です。

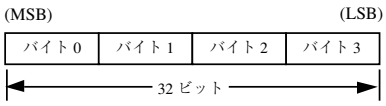
宣言

整数は次のように宣言します。

```
int identifier;
```

符号化

整数



符号なし整数

説明

XDR の符号なし整数は、[0, 4294967295] の範囲の正の整数が符号化された 32 ビットデータです。整数は符号なしの 2 進数で表されます。最上位バイト (MSB) と最下位バイト (LSB) はそれぞれバイト 0 とバイト 3 です。

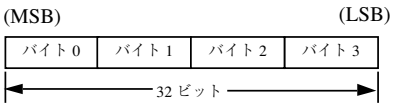
宣言

符号なし整数は次のように宣言します。

```
unsigned int identifier;
```

符号化

符号なし整数



列挙型

説明

列挙型のデータ表現方法は符号付き整数と同じです。列挙型は、整数のサブセットを記述するのに便利です。

宣言

列挙型は次のように宣言します。

```
enum {name-identifier = constant , ... } identifier;
```

たとえば、列挙型を使用して赤、黄、青の 3 色を次のように表すことができます。

```
enum {RED = 2, YELLOW = 3, BLUE = 5} colors;
```

列挙型に、enum 宣言で指定されていない整数を代入しようとするとエラーになります。

符号化

283ページの「符号付き整数」を参照してください。

ブール型

説明

ブール型は、標準規約の明示型に対応するための型で、よく使用される重要なデータ型です。ブール型には、整数の 0 と 1 を使用します。

宣言

ブール型は次のように宣言します。

```
bool identifier;
```

これは、次の宣言と同じです。

```
enum {FALSE = 0, TRUE = 1} identifier;
```

符号化

283ページの「符号付き整数」を参照してください。

hyper 整数と符号なし hyper 整数

説明

標準規約では 64 ビット (8 バイト) の整数として hyper int と unsigned hyper int を定義しています。その表現方法は明らかに、上で説明した integer と unsigned integer を拡張したものです。hyper 整数は 2 の補数で表されます。最上位バイト (MSB) と最下位バイト (LSB) はそれぞれバイト 0 とバイト 7 です。

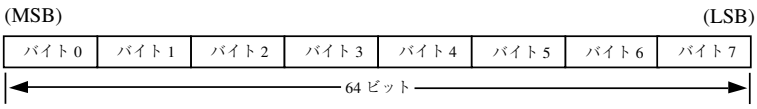
宣言

Hyper 整数は次のように宣言します。

```
hyper int identifier;
unsigned hyper int identifier;
```

符号化

Hyper 整数



浮動小数点

説明

標準規約では、浮動小数点型 `float` (32 ビット、すなわち 4 バイト) を定義しています。符号化方法は、正規化された単精度浮動小数点に関するIEEE 標準規約 [1] に従います。単精度浮動小数点は次の 3 つのフィールドで記述されます。

S: 符号を表す 1 ビットのフィールドです。0 が正、1 が負を表します。

E: 数値の指数部 (基数は 2) を表します。このフィールドは 8 ビットです。指数部の値を 127 だけバイアスした値が入っています。

F: 数値の仮数部 (基数は 2) を表します。このフィールドは 23 ビットです。

したがって、単精度浮動小数点型の値は次のように記述されます。

```
(-1)**S * 2**(E-Bias) * 1.F
```

宣言

単精度浮動小数点データは次のように宣言します。

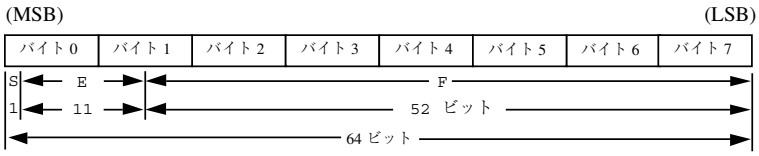
```
float identifier;
```

倍精度浮動小数点データは次にように宣言します。

```
double identifier;
```

符号化

倍精度浮動小数点



整数の最上位バイトと最下位バイトがバイト 0 とバイト 3 であるのと同様に、倍精度浮動小数点型の値の最上位ビットと最下位ビットはビット 0 とビット 63 になります。S、E、F の各フィールドの開始ビット (最上位ビット) のオフセットはそれぞれ 0、1、12 になります。

これらのオフセットは論理的ビット位置を示すもので、物理的位置を示すものではありません (物理的位置は媒体によって異なります)。

符号付きゼロ、符号付き無限大 (オーバフロー)、正規化されない値 (アンダフロー) の符号化については、IEEE 標準規約 [1] を参照してください。IEEE 標準規約によると、NaN (not a number) は、システムごとに異なるため外部表現では使用できません。

4 倍精度浮動小数点

説明

標準規約では、4 倍精度浮動小数点型 quadruple (128 ビット、すなわち 16 バイト) を定義しています。符号化方法は、正規化された 4 倍精度浮動小数点に関する IEEE 標準規約 [1] に従います。標準規約では、4 倍精度浮動小数点は次の 3 つのフィールドに符号化されます。

S: 符号を表す 1 ビットのフィールドです。0 が正、1 が負を表します。

E: 数値の指数部 (基数は 2) を表します。このフィールドは 15 ビットです。指数部の値を 16383 だけバイアスした値が入っています。

F: 数値の仮数部 (基数は 2) を表します。このフィールドは 111 ビットです。

したがって、4 倍精度浮動小数点型の値は次のように記述されます。

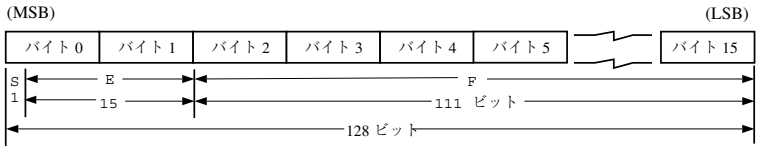
```
(-1)**S * 2**(E-Bias) * 1.F
```

宣言

```
quadruple identifier;
```

符号化

4 倍精度浮動小数点



整数の最上位バイトと最下位バイトがバイト 0 とバイト 3 であるのと同様に、4 倍精度浮動小数点型の値の最上位ビットと最下位ビットはビット0 とビット 127 になります。S、E、F の各フィールドの開始ビット (最上位ビット) のオフセットはそれぞれ 0、1、16 になります。これらのオフセットは論理的ビット位置を示すもので、物理的位置を示すものではありません。物理的位置は媒体によって異なります。

符号付きゼロ、符号付き無限大 (オーバフロー)、正規化されない値 (アンダフロー) の符号化については、IEEE 標準規約 [1] を参照してください。IEEE標準規約によると、NaN (not a number) は、システムごとに異なるため外部表現では使用できません。

固定長の隠されたデータ

説明

内容を解釈しない固定長データをマシン間で受け渡さなければならない場合があります。このデータを隠されたデータといいます。

宣言

隠されたデータは次のように宣言します。

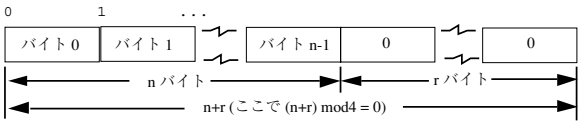
```
opaque identifier[n];
```

ここで定数 n は、隠されたデータを入れるのに必要な (固定) バイト数です。n バイトの隠されたデータの後には、0 ～ 3 個の余分なゼロバイト r が付加されて、隠されたオブジェクト全体のバイト数が 4 の倍数になるように調整されます。

符号化

n バイトの隠されたデータの後には、0 ～ 3 個の余分なゼロバイト r が付加されて、隠されたオブジェクト全体のバイト数が 4 の倍数になるように調整されます。

固定長の隠されたデータ



可変長の隠されたデータ

説明

標準規約では、可変長 (カウント付き) の隠されたデータも次のように定義されています。n バイトの任意のバイトシーケンス(バイト番号は 0 ～ n-1) がカウントされて下に示すように符号なし整数 n に符号化され、その後に n バイトのバイトシーケンスが続きます。

シーケンス内のバイト b は必ずバイト b+1 の直前に位置し、シーケンス内のバイト 0 はバイトシーケンスの長さ (カウント) の次に位置しています。n バイトのデータの後には、0～ 3 個の余分なゼロバイト r が付加されて、全体のバイト数が 4 の倍数になるように調整されます。

宣言

可変長の隠されたデータは次のように宣言します。

```
opaque identifier<m>;
```

または

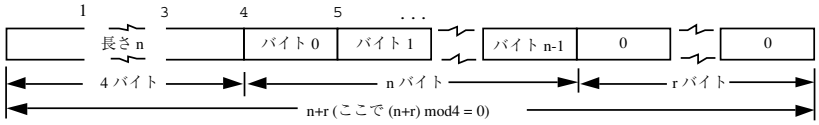
```
opaque identifier<>;;
```

定数 m は、シーケンスに含まれるバイト数の上限を示します。2 番目の宣言のように m を指定しないと、最大バイト数は (2**32) - 1 となります。たとえば、ファイル伝送プロトコルで最大データ伝送サイズを 8192 バイトとするには、次のように宣言します。

```
opaque filedata<8192>;
```

符号化

可変長の隠されたデータ



指定した最大バイト数以上の長さを符号化するとエラーになります。

カウント付きバイト文字列

説明

標準規約では、n バイトの ASCII 文字列 (バイト番号は 0〜 n-1) を次のように定義します。バイト数が符号なし整数 n に符号化されたものに、n バイトの文字列が続きます。文字列のバイト b は必ずバイト b+1 の直前に位置し、文字列のバイト 0 は文字列の長さの次に位置しています。n バイトのデータの後には、0 〜 3 個の余分なゼロバイト r が付加されて、全体のバイト数が 4 の倍数になるように調整されます。

宣言

カウント付きバイト文字列は次のように宣言します。

```
string object<m>;
```

または

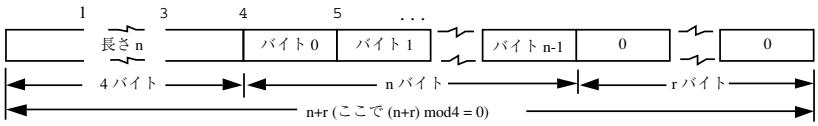
```
string object<>;
```

定数 m は、文字列に含まれるバイト数の上限を示します。2 番目の宣言のように、m を指定しないと、最大バイト数は (2**32) - 1 となります。定数 m は、通常プロトコル仕様で決められています。たとえば、ファイル伝送プロトコルでファイル名を最大 255 バイトとするには、次のように宣言します。

```
string filename<255>;
```

符号化

文字列



指定した最大バイト数以上の長さを符号化するとエラーになります。

固定長配列

固定長配列の要素番号は `0` ～ `n-1` で、個々の配列要素が `0` ～ `n-1` の番号順に符号化されます。各配列要素のバイト数は 4 の倍数になっています。全要素が同一のデータ型であっても、要素のサイズが異なることがあります。たとえば、文字列の固定長配列の場合、要素のデータ型はすべて `string` 型ですが、個々の要素の長さは異なります。

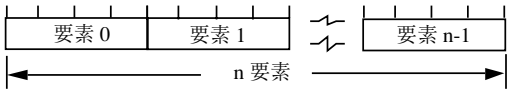
宣言

各要素のデータ型がすべて同じである固定長配列は、次のように宣言します。

`type-name` *identifier*`[n];`

符号化

固定長配列



可変長配列

説明

可変長配列をカウント付きで符号化することによって、固定長の要素と同じように符号化できます。要素カウント `n` (符号なし整数) に続けて、要素番号 `0` ～ `n-1` の順に各要素が符号化されます。

宣言

可変長配列は次のように宣言します。

```
type-name identifier<m>;
```

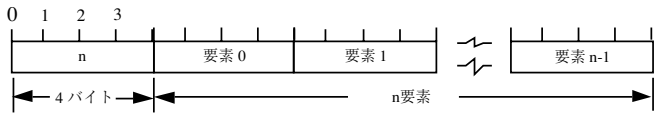
または

```
type-name identifier<>;
```

定数 m は、配列に含まれる要素数の上限を示します。m を指定しないと、最大要素数は (2**32) - 1 とみなされます。

符号化

可変長配列



仕様で決められた最大要素数より大きな長さを符号化するとエラーになります。

構造体

説明

構造体の構成要素は、構造体の宣言で並べた順に符号化されます。各構成要素のサイズはそれぞれ異なる可能性がありますが、各々が 4 の倍数に調整されます。

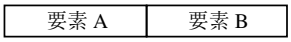
宣言

構造体は次のように宣言します。

```
struct {
    component-declaration-A;
    component-declaration-B;
    ...
} identifier;
```

符号化

構造体



識別型の共用体

説明

識別型の共用体には、要素識別子に続いて、あらかじめ配置された一連のデータ型から要素識別子の値に応じて選択されたものが入ります。要素識別子のデータ型は、int、unsigned int、bool などの列挙型、のいずれかです。共用体の構成要素の型をアームといい、符号化を暗黙に示す要素識別子に続けて記述されます。

宣言

識別型の共用体は次のように宣言します。

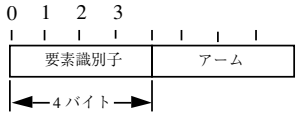
```
union switch (discriminant-declaration) {
   case discriminant-value-A:
    arm-declaration-A;
   case discriminant-value-B:
    arm-declaration-B;
   ...
   default:
    default-declaration;
} identifier;
```

キーワード case の後には、要素識別子として指定できる値を書きます。デフォルトアームは省略できますが、その場合は要素識別子として定義されていない値を持つものを正しく符号化できません。各アームのサイズは、それぞれ 4 の倍数になります。

識別型の共用体は、要素識別子に続けて、それに対応するアームが符号化されます。

符号化

識別型の共用体



Void

説明

XDR の void 型は 0 バイトのデータです。void は、入力データまたは出力データを持たない操作を記述するときに使用します。また、共用体で、アームによってデータを持つものと持たないときがある場合にも使用できます。

宣言

void の宣言は次のように簡単です。

```
void;
```

定数

説明

const 型は定数を表すシンボル名を定義するのに使用しますが、データを宣言するものではありません。シンボル定数は、通常の定数を使用できるところならどこでも使用できます。

次の例では、12 を表すシンボル定数 DOZEN を定義します。

```
const DOZEN = 12;
```

宣言

定数は次のように宣言します。

```
const name-identifier = n;
```

Typedef

typedef はデータを宣言するものではなく、新たな識別子でデータを宣言できるようにするためのものです。typedef の構文を次に示します。

```
typedef declaration;
```

typedef の宣言部分の変数名が、新たな型名になります。次の例では、既存の型 egg とシンボル定数 DOZEN を使用して、eggbox という新たな型を定義しています。

```
typedef egg eggbox[DOZEN];
```

新たな型名で宣言した変数は、typedef で変数として見た場合の型と同じ型を持ちます。したがって、次の 2 つの宣言は同じ型の変数 fresheggs: を宣言しています。

```
eggbox fresheggs;
 egg fresheggs[DOZEN];
```

typedef に struct、enum、union の定義が含まれるときは、同じ型を定義するのに、別のより望ましい構文が使用できます。一般に、typedef は次の形式で指定します。

```
typedef <<struct, union, or enum definition>> identifier;
```

この形式から typedef を取り去り、最後の識別子を struct、enum、union のキーワードの後に置くこともできます。bool 型を宣言する 2 つの方法を次に示します。

```
typedef enum {/* typedef を使用 */
   FALSE = 0,
   TRUE = 1
} bool;
enum bool {/* 望ましい方法 */
   FALSE = 0,
   TRUE = 1
};
```

最初の構文では宣言の最後まで見ないと新しい型名がわからないので、後の構文の方が望ましい方法です。

オプションデータ

オプションデータは共用体の一種ですが、次のような特殊な構文を持ち、非常によく使用されます。次のように宣言されます。

```
type-name *identifier;
```

これは次のように宣言した共用体と同じです。

```
union switch (bool opted) {
  case TRUE:
  type-name element;
  case FALSE:
  void;
} identifier;
```

また、これは次のような可変長配列の宣言とも同じです。ブール型 opted は、配列の長さと解釈できるからです。

```
type-name identifier<1>;
```

オプションデータは再帰的データ構造体、たとえば、リンクリストやツリーの宣言に便利です。

XDR 言語仕様

表記方法

この節では、XDR 言語を修正バッカス-ナウア記法で記述します。その表記規則を次に簡単に説明します。

1. 特殊文字として、|、(、)、[、]、* を使用する。
2. 終端記号は、引用符 (") で囲んだ文字列または文字とする。
3. 非終端記号は、非特殊文字からなる文字列で、イタリックで表示する。
4. 代替項目は縦棒 (|) で区切って並べる。
5. 省略可能な項目は角括弧で囲む。
6. 項目をグループ化するときは、括弧で囲む。
7. 項目の後に * が付いている場合は、その項目の 0 回以上の繰り返しを表す。

たとえば、次のパターンを考えてみます。

```
"a " "very" (", " " very")* [" cold " "and"] " rainy "
       ("day" | "night")
```

このパターンには、次の文字列を始めとして無数の文字列が一致します。

```
a very rainy day
   a very, very rainy day
   a very cold and rainy day
   a very, very, very cold and rainy night
```

字句解析ノート

1. コメントは、/* と */ で囲む。
2. 空白は項目と項目の区切りに使用し、意味を持たない。
3. 識別子は英字で始まり、英字、数字、下線 (_) を含むことができる。識別子では、大文字と小文字を区別する。
4. 任意の桁数の 10 進数を定数といい、始めに負符号 (-)を付けることができる。

コード例 **C–1**　XDR 仕様

```
Syntax Information
declaration:
 type-specifier identifier
 | type-specifier identifier "[" value "]"
 | type-specifier identifier "<" [ value ] ">"
 | "opaque" identifier "[" value "]"
 | "opaque" identifier "<" [ value ] ">"
 | "string" identifier "<" [ value ] ">"
 | type-specifier "*" identifier
 | "void"

value:
 constant
 | identifier

type-specifier:
  [ "unsigned" ] "int"
 | [ "unsigned" ] "hyper"
 | "float"
 | "double"
 | "quadruple"
 | "bool"
 | enum-type-spec
 | struct-type-spec
 | union-type-spec
 | identifier

enum-type-spec:
 "enum" enum-body

enum-body:
 "{"
 ( identifier "=" value )
 ( "," identifier "=" value )*
 "}"

struct-type-spec:
 "struct" struct-body

struct-body:
 "{"
 ( declaration ";" )
 ( declaration ";" )*
 "}"

 union-type-spec:
 "union" union-body

union-body:
 "switch" "(" declaration ")" "{"
 ( "case" value ":" declaration ";" )
 ( "case" value ":" declaration ";" )*
 [ "default" ":" declaration ";" ]
 "}"
```

(続く)

```
constant-def:
 "const" identifier "=" constant ";"

type-def:
 "typedef" declaration ";"
 | "enum" identifier enum-body ";"
 | "struct" identifier struct-body ";"
 | "union" identifier union-body ";"

definition:
 type-def
 | constant-def

specification:
 definition *
```

構文ノート

1. 次に示すものはキーワードとして予約されており、識別子として使用できない。

表 C–1 XDR キーワード

bool	const	enum	int	string	typedef	void
cas	default	float	opaque	struct	union	
cha	double	hyper	quadruple	switch	unsigned	

1. 配列のサイズ指定に使用できるのは、符号なし定数だけである。識別子で指定するときは、その識別子をそれまでに const 定義を使用して符号なし定数として宣言しておく必要がある。
2. 指定範囲内の識別子の定数と型は、同じ名前空間内にあり、この範囲内で一意に宣言されていなくてはなりません。
3. 同様に、構造体と共用体の宣言の有効範囲内では、変数名が一意的に決まらなければならない。構造体と共用体の宣言がネストしている場合は、新たな有効範囲ができる。

4. 共用体の要素識別子は、整数を表す型でなければならない。すなわち、int、unsigned int、bool、enum、または、このどれかの型を typedef で定義したものでなければならない。case で指定する値は、要素識別子の型に応じた値でなければならない。また、union 宣言の有効範囲内で case の値を 2 回以上指定してはならない。

XDR データ記述

ファイルのデータ構造を XDR で記述した簡単な例を次に示します。このデータは、マシン間のファイル転送に使用することができます。

コード例 **C–2** XDR ファイルデータ構造体

```
const MAXUSERNAME = 32;/* ユーザー名の最大長 */
const MAXFILELEN = 65535;  /* ファイルの最大長 */
const MAXNAMELEN = 255;    /* ファイル名の最大長 */

/* ファイルのタイプ */
enum filekind {
  TEXT = 0, /* ASCII データ */
  DATA = 1, /* raw データ */
  EXEC = 2  /* 実行可能形式 */
 };

/* ファイルタイプごとのファイル情報 */
union filetype switch (filekind kind) {
  case TEXT:
   void;                              /* その他の情報なし */
  case DATA:
   string creator<MAXNAMELEN>;        /* データ作成者 */
  case EXEC:
   string interpretor<MAXNAMELEN>; /*プログラムインタプリタ*/
};

/* ファイル全体 */
struct file {
  string filename<MAXNAMELEN>;         /* ファイル名 */
  filetype type;                       /* ファイル情報 */
  string owner<MAXUSERNAME>;           /* ファイルの所有者 */
  opaque data<MAXFILELEN>;             /* ファイルデータ*/
};
```

john というユーザーが、(quit) というデータだけが入った自分の LISP プログラム sillyprog を XDR 形式で保存するとします。このファイルは次のように符号化されます。

表 C–2　XDR データ記述の例

オフセット	バイト (16 進)	ASCII	説明
0	00 00 00 09	....	ファイル名の長さ = 9
4	73 69 6c 6c	sill	ファイル名の文字列
8	79 70 72 6f	ypro	ファイル名の文字列 (続き)
12	67 00 00 00	g...	ファイル名の文字列 (続き) と 3 バイトのヌルパディング
16	00 00 00 02	....	ファイルタイプは EXEC = 2
20	00 00 00 04	....	インタプリタ名の長さ = 4
24	6c 69 73 70	lisp	インタプリタの文字列
28	00 00 00 04	....	所有者名の長さ = 4
32	6a 6f 68 6e	john	所有者名
36	00 00 00 06	....	ファイルデータの長さ = 6
40	28 71 75 69	(qu	ファイルデータ
44	74 29 00 00	t)..	ファイルデータ (続き) と 2 バイトのヌルパディング

RPC 言語リファレンス

RPC 言語は XDR 言語を拡張したものです。唯一の拡張は program 型と version 型の追加です。

XDR 言語への RPC 拡張の説明については、 付録 B を参照してください。

RPC 言語は C 言語と同様です。この節では、実際の例を示しながら RPC 言語の構文について説明します。また、RPC 型と XDR 型の定義が、出力ッダーファイル内で、どのように C 型の定義にコンパイルされるかについても説明します。

RPC 言語ファイルは以下の定義から構成されています。

```
definition-list:
  definition;
  definition; definition-list
```

また、RPC 言語ファイルは 6 つの型の定義を認めています。

```
definition:
  enum-definition
  const-definition
  typedef-definition
  struct-definition
  union-definition
  program-definition
```

定義は宣言と同じではありません。1 つまたは一連のデータ要素の型定義以外の定義によっては領域を割り当てることはできません。これは、変数は定義するだけでは十分でなく、宣言もする必要があることを意味しています。

列挙型

RPC/XDR 列挙型の構文は、C 列挙型と同様です。

```
enum-definition:
   "enum" enum-ident "{"
   enum-value-list
   "}"

enum-value-list:
   enum-value
   enum-value "," enum-value-list

enum-value:
   enum-value-ident
   enum-value-ident "=" value
```

XDR enum のコードと、それを C にコンパイルした結果の例を以下に示します。

```
enum colortype {                enum colortype {
  RED = 0,                        RED = 0,
  GREEN = 1,        -->           GREEN = 1,
  BLUE = 2                        BLUE = 2,
};                              };
                                typedef enum colortype colortype;
```

定数

XDR シンボル定数は、整数定数が使用される場合は常に使用できます。たとえば、配列サイズを以下のように指定します。

```
const-definition:
  const const-ident = integer
```

次の例では、定数 DOZEN が 12 になるように定義します。

```
const DOZEN = 12; --> #define DOZEN 12
```

型定義

XDR typedef の構文は、C typedef と同様です。

```
typedef-definition:
   typedef declaration
```

次の例では、最大の長さが 255 文字のファイル名文字列を宣言するために使用される fname_type を定義しています。

```
typedef string fname_type<255>; --> typedef char *fname_type;
```

宣言

XDR には、4 種類の宣言があります。これらの宣言は、struct または typedef の一部でなければならず、単独では使用できません。

```
declaration:
  simple-declaration
  fixed-array-declaration
  variable-array-declaration
  pointer-declaration
```

単純な宣言

単純な宣言は単純な C の宣言とほとんど同じです。

```
simple-declaration:
  type-ident variable-ident
```

例 :

```
colortype color; --> colortype color;
```

固定長配列宣言

固定長配列宣言は C の配列宣言とほとんど同じです。

```
fixed-array-declaration:
  type-ident variable-ident [value]
```

例 :

```
colortype palette[8]; --> colortype palette[8];
```

多くのプログラマは、変数の宣言を型の宣言と混同します。rpcgen は変数の宣言をサポートしないことに注意することが重要です。以下に、コンパイルできないプログラムの例を示します。

```
int data[10];
program P {
   version V {
      int PROC(data) = 1;
  } = 1;
} = 0x200000;
```

上記の例は、変数の宣言があるためにコンパイルできません。

```
int data[10]
```

int data[10] の代わりに、以下を使用してください。

```
typedef int data[10];
```

または

```
struct data {int dummy [10]};
```

可変長配列宣言

C 言語には、可変長配列宣言の明示的な構文がありません。XDR 言語には、山括弧を使用する構文があります。

```
variable-array-declaration:
  type-ident variable-ident <value>
  type-ident variable-ident < >
```

最大のサイズは山括弧の間に指定されます。サイズを省略すると、配列が任意のサイズを持つということが示されます。

```
int heights<12>; /* 最大で 12 項目 */
int widths<>; /* 任意の数の項目 */
```

C では可変長配列の明示的な構文がないため、これらの宣言は struct 宣言にコンパイルされます。たとえば、heights 宣言は次の struct 宣言にコンパイルされます。

```
struct {
   u_int heights_len;     /* 配列の項目の数 */
  int *heights_val;      /* 配列へのポインタ */
} heights;
```

配列の項目の数は _len 構成要素に格納され、配列へのポインタは _val 構成要素に格納されます。各構成要素名の最初の部分は、宣言された XDR 変数 (heights) の名称と同様です。

ポインタ宣言

ポインタ宣言は、C で行われる場合とまったく同様に XDR でも行われます。アドレスポインタは、実際にはネットワーク上に送信されないのに対して、XDR ポインタは、リストおよびツリーなどの再帰的なデータ型を送信するのに有効です。XDR 言語では、型は「ポインタ」ではなく、「オプショナルデータ」と呼ばれます。

```
pointer-declaration:
  type-ident *variable-ident
```

例：

```
listitem *next; --> listitem *next;
```

構造体

RPC/XDR `struct` はその C の構造体とほとんど同じように宣言されます。宣言は以下のように行われます。

```
struct-definition:
   struct struct-ident "{"
      declaration-list
  "}"

declaration-list:
   declaration ";"
  declaration ";" declaration-list
```

次に、二次元の座標である XDR 構造体と、その構造体がコンパイルされて生成される C 構造体の例を示します。

```
struct coord {                  struct coord {
   int x;             -->           int x;
  int y;                          int y;
};                                };
                                  typedef struct coord coord;
```

出力は、最後に追加された `typedef` を除いて入力と同一です。これによって、項目を宣言する際に、`struct coord` の代わりに `coord` を使用することができます。

共用体

XDR 共用体は識別された共用体であり、C 共用体には似ていません。XDR 共用体は Pascal 変形レコードに、より似ています。

```
union-definition:

"union" union-ident "switch" "("simple declaration")" "{"
       case-list
```

```
   "}"

case-list:
   "case" value ":" declaration ";"
  "case" value ":" declaration ";" case-list
  "default" ":" declaration ";"
```

次に、「データの読み取り」操作の結果として返される型の例を示します。エラーがない場合はデータのブロックを返し、エラーが発生した場合は何も返しません。

```
union read_result switch (int errno) {
  case 0:
      opaque data[1024];
  default:
   void;
  };
```

上記のコードは以下のようにコンパイルされます。

```
struct read_result {
  int errno;
  union {
      char data[1024];
  } read_result_u;
};
typedef struct read_result read_result;
```

出力 struct の共用体構成要素が、後に付いている _u を除いて型の名称と同じ名称であることに注意してください。

プログラム

RPC プログラムは、次の構文を使用して宣言します。

```
program-definition:
  "program" program-ident "{"
   version-list
  "}" "=" value;
version-list:
  version ";"
  version ";" version-list
version:
  "version" version-ident "{"
   procedure-list
  "}" "=" value;
procedure-list:
  procedure ";"
  procedure ";" procedure-list
procedure:
   type-ident procedure-ident "(" type-ident ")" "=" value;
```

–N オプションを指定すると、rpcgen は次の構文も認識します。

```
procedure:
  type-ident procedure-ident "(" type-ident-list ")" "=" value;
type-ident-list:
  type-ident
  type-ident "," type-ident-list
```

たとえば以下のようになります。

```
/*
 * time.x: 時間を取得するか、または設定してください。
 * 時間は、1970 年 1 月 1 日 0:00 から経過した秒数で表されます。
 */
program TIMEPROG {
   version TIMEVERS {
      unsigned int TIMEGET(void) = 1;
   void TIMESET(unsigned) = 2;
  } = 1;
} = 0x20000044;
```

void 引き数型は、引数が渡されないことを意味していることに注意してください。

このファイルは、出力ヘッダーファイルの、以下の #define 文にコンパイルされます。

```
#define TIMEPROG 0x20000044
#define TIMEVERS 1
#define TIMEGET 1
#define TIMESET 2
```

特殊な場合

RPC 言語規則には、いくつかの例外があります。

C 形式モード

新しい機能の節で、rpcgen の C 形式モードの機能について説明しました。これらの機能は、void 引数に関連しています。引数の値が void である場合は、引数は渡される必要はありません。

ブール型

C には組み込みのブール型はありません。しかし、RPC ライブラリは、TRUE または FALSE の、bool_t と呼ばれるブール型を使用します。XDR 言語でブール型として宣言されたパラメータは、出力ヘッダーファイルの bool_t にコンパイルされます。

例 :

```
bool married; --> bool_t married;
```

文字列

C 言語には組み込みの文字列型がありませんが、その代わりに、NULL で終了する char * 規則を使用します。C では、文字列は通常 NULL で終了する一次元の配列とみなされます。

XDR 語では、文字列は string キーワードを使用して宣言され、出力ッダーファイルの char * 型にコンパイルされます。山括弧の間で指定される最大サイズによって、文字列内で使用できる文字の最大数が指定されます (NULL 文字はカウントされない)。最大サイズを省略すると、任意の長さの文字列を指定できます。

例 :

```
string name<32>;   --> char *name;
string longname<>; --> char *longname;
```

注 - NULL 文字列を渡すことはできません。ただし、長さが 0 の文字列 (すなわち、ターミネータまたは NULL バイト) を渡すことはできます。

隠されたデータ

XDR では、隠されたデータは、任意のバイトのシーケンスなどの、型のないデータを記述するために使用されます。隠されたデータは、固定長または可変長の配列として宣言することもできます。たとえば、以下のようになります。

```
opaque diskblock[512]; --> char diskblock[512];
opaque filedata<1024>; --> struct {
                               u_int filedata_len;
                               char *filedata_val;
                           } filedata;
```

Voids

void 宣言では、変数は指定されません。void 宣言には、void 以外何も記述しません。void 宣言は、共用体定義とプログラム定義の 2 つの部分で行うことができます (たとえば、引数を渡さない遠隔手続きの引き数または結果として)。

付録 **D**

RPC サンプルプログラム

この章では、このマニュアルの rpcgen と RPC の章で使用したサンプルプログラムの完全なリストを示します。特に但し書き (擬似プログラムであるというような但し書き) がない限り、このまま書かれているとおりにコンパイルして実行できます。これらのサンプルプログラムはマニュアルの説明を補うために示すものです。サン・ソフトは実行結果に対してどのような責任も負いません。

ディレクトリリストプログラムとその補助ルーチン (rpcgen)

コード例 **D–1**　rpcgen プログラム : dir.x

```
/*
 *  dir.x: 遠隔ディレクトリリストの
 * プロトコル
 *
 * このソースモジュールは rpcgen が生成したソースモジュールです。
 * これを使用して rpcgen ツールの機能を説明します。
 *
 * このプログラムは、rpcgen の -h -T のスイッチを使用して
 * ヘッダーファイル (.h) と付随するデータ構造体も同時に生成して
 * コンパイルします。
 *
 */
const MAXNAMELEN = 255; /* ディレクトリエントリの最大長 */

typedef string nametype<MAXNAMELEN>; /* ディレクトリエントリ */
typedef struct namenode *namelist;   /* リスト内のリンク */
```

(続く)

```
/*
 * ディレクトリリスト内のノード struct namenode
 */
struct namenode {
  nametype name;        /* ディレクトリエントリ名 */
  namelist next;        /* 次のエントリ */
 };

/*
 * READDIR 操作の結果:
 * 完全に移植可能なアプリケーションにするには、ここで行なっているように
 * UNIX の errno を使用せずに、取り決めに従ったエラーコードリストを
 * 使用します。この例では、遠隔呼び出しが成功したか失敗したかを
 * 識別するのに共用体を使用しています。
 */
union readdir_res switch (int errno) {
 case 0:
  namelist list; /* エラーなし: ディレクトリリストを返す */
 default:
  void;       /* エラー発生: 戻り値なし */
};

/*
 * ディレクトリリストプログラムの定義
 */
program DIRPROG {
  version DIRVERS {
   readdir_res
   READDIR(nametype) = 1;
  } = 1;
 } = 0x20000076;
```

コード例 **D–2**　遠隔 dir_proc.c

```
/*
 * dir_proc.c: 遠隔手続き readdir
 */
#include <rpc/rpc.h>      /* 必ず必要 */
#include <dirent.h>
#include "dir.h"          /*  rpcgen が生成 */

extern int errno;
extern char *malloc();
extern char *strdup();
```

(続く)

```
/* 使用する引数 */
readdir_res *
readdir_1(dirname,req)
  nametype *dirname;
  struct svc_req *req;
{
  DIR *dirp;
  struct dirent *d;
  namelist nl;
  namelist *nlp;
  static readdir_res res; /* 必ず static で宣言 */

 /*
  * ディレクトリのオープン
  */
 dirp = opendir(*dirname);
 if (dirp == (DIR *)NULL) {
   res.errno = errno;
   return (&res);
  }
 /*
  * 以前の結果を解放
  */
 xdr_free(xdr_readdir_res, &res);
 /*
  * ディレクトリエントリの収集。ここで割り当てられたメモリーは、
  * 次に readdir_1 が呼び出されたときに xdr_free によって解放されます。
  */

 nlp = &res.readdir_res_u.list;
  while (d = readdir(dirp)) {
   nl = *nlp = (namenode *) malloc(sizeof(namenode));
   if (nl == (namenode *) NULL) {
    res.errno = EAGAIN;
    closedir(dirp);
    return(&res);
   }
   nl->name = strdup(d->d_name);
   nlp = &nl->next;
  }
  *nlp = (namelist)NULL;
  /* 結果を返す */
  res.errno = 0;
  closedir(dirp);
  return (&res);
}
```

コード例 **D–3**　rls.c クライアント

```
/*
 * rls.c: 遠隔ディレクトリリスト (クライアント側)
 */

#include <stdio.h>
#include <rpc/rpc.h> /* 必ず必要 */
#include "dir.h"     /* rpcgen が生成 */

extern int errno;

main(argc, argv)
  int argc;
  char *argv[];
{
  CLIENT *cl;
  char *server;
  char *dir;
  readdir_res *result;
  namelist nl;

 if (argc != 3) {
  fprintf(stderr, "usage: %s host directory\n",
  argv[0]);
  exit(1);
 }
 server = argv[1];
  dir = argv[2];
/*
 * コマンド行で指定したサーバー上の MESSAGEPROG の呼び出しに使用する
 * クライアント「ハンドル」を作成します。
 */
 cl = clnt_create(server, DIRPROG, DIRVERS, "visible");
  if (cl == (CLIENT *)NULL) {
   clnt_pcreateerror(server);
   exit(1);
  }

 result = readdir_1(&dir, cl);
  if (result == (readdir_res *)NULL) {
   clnt_perror(cl, server);
   exit(1);
  }

/* 遠隔手続きの呼び出しに成功 */

 if (result->errno != 0) {
 /*
  * 遠隔システムでエラーが発生。エラーメッセージを表示して終了。
  */
 }
 if (result->errno < sys_nerr)
   fprintf (stderr, "%s : %s\n", dir,
   sys_enlist[result->errno]);
   errno = result->errno;
   perror(dir);
```

(続く)

```
   exit(1);
  }

/* ディレクトリリストの取り出しに成功。ディレクトリリストを表示 */
 for(nl = result->readdir_res_u.list; nl != NULL; nl = nl-
>next) {
  printf("%s\n", nl->name);
 }
exit(0);
```

時刻サーバープログラム (rpcgen)

コード例 **D–4**　rpcgen プログラム：time.x

```
/*
 * time.x: 遠隔時刻プロトコル
 */
program TIMEPROG {
  version TIMEVERS {
    unsigned int TIMEGET(void) = 1;
  } = 1;
} = 0x20000044;

#ifdef RPC_SVC
%int *
%timeget_1()
%{
% static int thetime;
%
% thetime = time(0);
% return (&thetime);
%}
#endif
```

2 つの数値の合計を求めるプログラム (rpcgen)

コード例 **D–5**　rpcgen プログラム : 2 つの数値の合計を求める

```
/* 新たな rpcgen 機能を説明するために、このプログラムには
 * 2 つの数値を加える手続きが入っています。ここでは add() が 2 つの引数を取ること
 * に注意してください。
 */
program ADDPROG {    /* プログラム番号 */
 version ADDVER {  /* バージョン番号 */
  int add ( int, int )   /* 手続き */
   = 1;
 } = 1;
} = 199;
```

スプレイパケットプログラム (rpcgen)

このツールの使用方法については、spray(1M) のマニュアルページの注を参照してください。

コード例 **D–6**　rpcgen プログラム : spray.x

```
/*
 * 著作権所有 (c) 1987 年、1991 年  サンマイクロシステムズ社
 */

/* spray.x から */

#ifdef RPC_HDR
#pragma ident "@(#)spray.h  1.2  91/09/17 SMI"
#endif

/*
 * サーバーにパケットをスプレイします。
 * ネットワークインタフェースのもろさのテストに使用します。
 */

const SPRAYMAX = 8845; /* スプレイ可能な最大量 */

/*
```

(続く)

```
 * 1970 年 1 月 1 日 0:00 からの GMT
 */
struct spraytimeval {
 unsigned int sec;
 unsigned int usec;
};

/*
 * スプレイ統計情報
 */
struct spraycumul {
 unsigned int counter;
 spraytimeval clock;
};

/*
 * スプレイデータ
 */
typedef opaque sprayarr<SPRAYMAX>;

program SPRAYPROG {
 version SPRAYVERS {
  /*
   * データを捨てて、カウンタをインクリメントします。この呼び出しは
   * リターンしないため、クライアントは必ずタイムアウトになります。
   */
  void
  SPRAYPROC_SPRAY(sprayarr) = 1;

  /*
   * カウンタ値と、最後にクリアしたときからの経過時間を取り出します。
   */
  spraycumul
  SPRAYPROC_GET(void) = 2;

  /*
   * カウンタをクリアし、経過時間をリセットします。
   */
  void
  SPRAYPROC_CLEAR(void) = 3;
 } = 1;
} = 100012;
```

メッセージ表示プログラムとその遠隔バージョン

コード例 **D–7** printmesg.c

```
/* printmsg.c: コンソールにメッセージを表示 */
#include <stdio.h>

main(argc, argv)
  int argc;
  char *argv[];
{
 char *message;

 if (argc != 2) {
  fprintf(stderr, "usage: %s <message>\n", argv[0]);
  exit(1);
 }
 message = argv[1];
 if( !printmessage(message) ) {
  fprintf(stderr, "%s: couldn't print your message\n",
          argv[0]);
  exit(1);
 }
 printf("Message Delivered!\n");
 exit(0);
}

/* コンソールにメッセージを表示します。 */

/*
 * メッセージが実際に表示されたかどうかを示すブール値を返します。
 */
printmessage(msg)
 char *msg;
{
 FILE *f;

 if = fopen("/dev/console","w");
  if (f == (FILE *)NULL)
   return (0);
 fprintf(f,"%sen'', msg);
 fclose(f);
 return (1);
}
```

コード例 **D–8**　printmesg.c の遠隔バージョン

```
/*  * rprintmsg.c: printmsg.c の遠隔バージョン  */
#include <stdio.h>
#include <rpc/rpc.h> /* 必ず必要 */
#include "msg.h''     /* msg.h は rpcgen が生成 */

main(argc, argv)
 int argc;
 char *argv[];
{
 CLIENT *cl;
 int *result;
 char *server;
 char *message;
 extern int sys_nerr;
 extern char *sys_errlist[];

 if (argc != 3) {
  fprintf(stderr,"usage: %s host messagen", argv[0]);
  exit(1);
 }
 /*
  * コマンド行で指定した引数の値を保存します。
  */
 server = argv[1];
 message = argv[2];
/*
 * コマンド行で指定したサーバー上の MESSAGEPROG の呼び出しに使用する
 * クライアント「ハンドル」を作成します。
 */
 cl = clnt_create(server, MESSAGEPROG, PRINTMESSAGEVERS,
                  "visible");
 if (cl == (CLIENT *)NULL) {
  /*
   * サーバーとの接続の確立に失敗。
   * エラーメッセージを表示して終了します。
   */
  clnt_pcreateerror(server);
  exit(1);
 }
 /* サーバー上の遠隔手続き printmessage を呼び出します。 */
 result = printmessage_1(&message, cl);
 if (result == (int *)NULL) {
 /*
 * サーバーの呼び出しでエラーが発生。
  * エラーメッセージを表示して終了します。
  */
  clnt_perror(cl, server);
  exit(1);
 }
 /* 遠隔手続きの呼び出しに成功。 */
 if (*result == 0) {
  /*
   * サーバーはメッセージの表示に失敗。
   * エラーメッセージを表示して終了します。
   */
```

(続く)

```
  fprintf(stderr,"%s"
 }
/* サーバーのコンソールにメッセージが出力されました。 */
 printf("Message delivered to %s!\n", server);
 exit(0);
}
```

コード例 **D–9**　rpcgen プログラム：msg.x

```
/* msg.x: 遠隔メッセージ印刷プロトコル */
program MESSAGEPROG {
  version MESSAGEVERS {
   int PRINTMESSAGE(string) = 1;
  } = 1;
} = 0x20000001;
```

コード例 **D–10**　mesg_proc.c

```
/*
 *  msg_proc.c: 遠隔手続き printmessage
 */

#include <stdio.h>
#include <rpc/rpc.h>  /* 必ず必要 */
#include "msg.h''  /* msg.h は rpcgen が生成 */

/*
 * printmessage の遠隔バージョン
 */
/* 使用する引数 */
int printmessage_1(msg, req)
 char **msg;
 struct svc_req *req;
{
 static int result; /* 必ず static で宣言 */
 FILE *f;

 f = fopen("/dev/console", "w");
 if (f == (FILE *)NULL) {
  result = 0;
```

(続く)

続き

```
  return (&result);
 }
 fprintf(f, "%sen", *msg);
  fclose(f);
  result = 1;
  return (&result);
}
```

バッチコードの例

コード例 **D–11**　バッチを使用するクライアントプログラムの例

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "windows.h"

main(argc, argv)
 int             argc;
 char          **argv;
{
 struct timeval  total_timeout;
 register CLIENT *client;
 enum clnt_stat  clnt_stat;
 char            buf[1000], *s = buf;

 if ((client = clnt_create(argv[1], WINDOWPROG, WINDOWVERS,
              "CIRCUIT_V")) == (CLIENT *) NULL) {
  clnt_pcreateerror("clnt_create");
  exit(1);
 }

 timerclear(&total_timeout);
 while (scanf("%s", s) != EOF) {
  clnt_call(client, RENDERSTRING_BATCHED, xdr_wrapstring,
    &s,xdr_void, (caddr_t) NULL, total_timeout);
 }

 /* ここでパイプラインをフラッシュ */
 total_timeout.tv_sec = 20;
 clnt_stat = clnt_call(client, NULLPROC, xdr_void,
```

(続く)

```
    (caddr_t) NULL, xdr_void, (caddr_t) NULL,
total_timeout);
 if (clnt_stat != RPC_SUCCESS) {
  clnt_perror(client, "rpc");
  exit(1);
 }
 clnt_destroy(client);
 exit(0);
}
```

コード例 **D–12**　バッチを使用するサーバープログラムの例

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "windows.h"

void            windowdispatch();
main()
{
 int num;

 num = svc_create(windowdispatch, WINDOWPROG, WINDOWVERS,
       "CIRCUIT_V");
 if (num == 0) {
  fprintf(stderr, "can't create an RPC server\n");
  exit(1);
 }
 svc_run();                          /* 戻らない */
 fprintf(stderr, "should never reach this point\n");
}

void
windowdispatch(rqstp, transp)
 struct svc_req *rqstp;
 SVCXPRT        *transp;
{
 char           *s = NULL;

 switch (rqstp->rq_proc) {
  case NULLPROC:
   if (!svc_sendreply(transp, xdr_void, 0))
    fprintf(stderr, "can't reply to RPC call\n");
   return;
  case RENDERSTRING:
   if (!svc_getargs(transp, xdr_wrapstring, &s)) {
    fprintf(stderr, "can't decode arguments\n");
```

(続く)

```
    /* 呼び出し側にエラーを通知 */
    svcerr_decode(transp);
    break;
   }
   /* 文字列 s を処理するコード */
   if (!svc_sendreply(transp, xdr_void, (caddr_t) NULL))
    fprintf(stderr, "can't reply to RPC call\n");
   break;
  case RENDERSTRING_BATCHED:
   if (!svc_getargs(transp, xdr_wrapstring, &s)) {
    fprintf(stderr, "can't decode arguments\n");
    /* プロトコルエラーのため何も返さない */
    break;
   }
   /* 文字列 s を処理するコード。ただし応答はしない。 */
   break;
  default:
   svcerr_noproc(transp);
   return;
 }
 /* 引数の復号化で割り当てた文字列を解放 */
 svc_freeargs(transp, xdr_wrapstring, &s);
}
```

バッチを使用しない例

次のプログラムは参考のためだけに示します。コード例 D–11 で示したバッチを使用するクライアントプログラムの例を、バッチを使用しないように書き直したものです。

コード例 D–13　バッチを使用しないクライアントプログラムの例

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "windows.h"

main(argc, argv)
 int             argc;
 char          **argv;
```

(続く)

```
{
 struct timeval  total_timeout;
 register CLIENT *client;
 enum clnt_stat  clnt_stat;
 char            buf[1000], *s = buf;

 if ((client = clnt_create(argv[1], WINDOWPROG, WINDOWVERS,
             "CIRCUIT_V")) == (CLIENT *) NULL) {
  clnt_pcreateerror("clnt_create");
  exit(1);
 }
 total_timeout.tv_sec = 20;
 total_timeout.tv_usec = 0;
 while (scanf("%s", s) != EOF) {
  if(clnt_call(client, RENDERSTRING, xdr_wrapstring, &s,
      xdr_void, (caddr_t) NULL, total_timeout) != RPC_SUCCESS) {
   clnt_perror(client, "rpc");
   exit(1);
  }
 }
 clnt_destroy(client);
 exit(0);
}
```

付録 **E**

portmap ユーティリティ

SunOS の旧バージョンのユーティリティである portmap ユーティリティの代わりに rpcbind ユーティリティを使用します。この章は、portmap ユーティリティの話題からポートとネットワークアドレスの解決方法の変遷を理解していただくためのものです。

SunOS 4.*x* の RPC ベースのサービスでは、システム登録サービスとしては portmap を使用していました。portmap は、ポート (論理通信チャネル) と、そこに登録されたサービスとの対応テーブルを管理します。portmap は、サーバーがサポートしている RPC プログラムに対する TCP/IP または UDP/IP のポート番号をクライアントが調べるための標準的な方法を提供します。

システム登録の概要

クライアントプログラムがネットワーク上の分散サービスを利用するには、サーバープログラムのネットワークアドレスを知る必要があります。ネットワークトランスポート (プロトコル) サービスはこのような機能を提供しません。ネットワーク上のプロセス間メッセージを伝送するだけです。つまり、メッセージはトランスポート固有のネットワークアドレスに送信されます。ネットワークアドレスとは論理通信チャネルのことで、プロセスは特定のネットワークアドレスを監視することにより、ネットワークからのメッセージを受信します。

プロセスがどのようにネットワークアドレスを監視するかは、オペレーティングシステムごとに異なりますが、どのオペレーティングシステムでも、メッセージの到着に同期してプロセスがアクティビティを起こすような機能があります。メッセー

ジは受信側プロセスに対してネットワーク上に送信されるのではなく、特定のネットワークアドレスに対して送信され、そこを監視していた受信側プロセスがそのメッセージを取り出します。ネットワークアドレスが重要なのは、受信側のオペレーティングシステムの方針に依存しない方法で、メッセージの宛先を指定できるからです。TI-RPC はトランスポート独立ですので、ネットワークアドレスの実際の構造については関知しません。その代わりに、TI-RPC は汎用アドレスを使用します。汎用アドレスは NULL で終わる文字列で指定します。汎用アドレスは、各トランスポートプロバイダに固有のルーチンにより、ローカルトランスポートアドレスに変換されます。

rpcbind プロトコルでは、サーバーがサポートする任意の遠隔プログラムのネットワークアドレスをクライアントから調べるための標準の方法を提供します。rpcbind はどのトランスポートにも実現できますので、全クライアント、全サーバー、全ネットワークに適用できるような 1 つの問題解決方法を提供します。

portmap プロトコル

portmap プログラムは、RPC プログラムとバージョン番号を、トランスポート固有のポート番号にマップします。portmap プログラムは、遠隔プログラムの動的結合を可能にします。

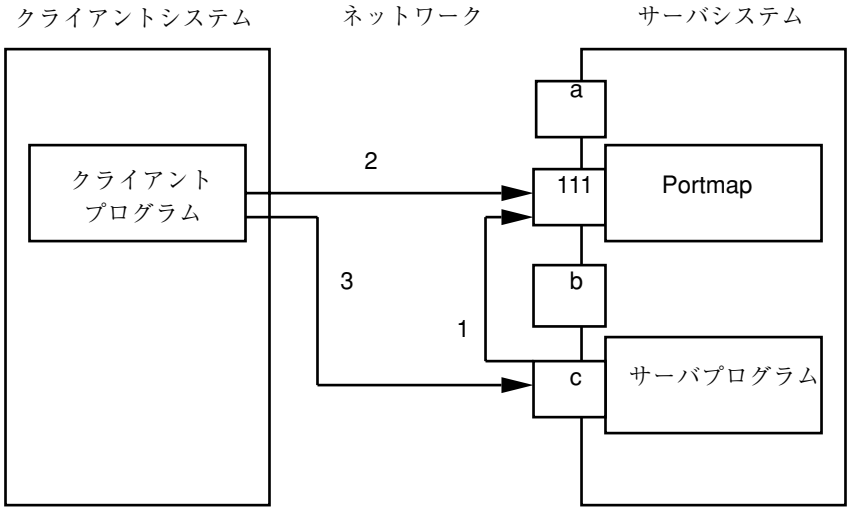


図 E–1　典型的な Portmap シーケンス (TCP/IP トランスポートのみ)

図 E–1 は、次の処理を示しています。

1. サーバーが portmap に登録する。
2. クライアントは、サーバーのポートを portmap から得る。
3. クライアントが、サーバーを呼び出す。

予約ポート番号は少ないのに対して、遠隔プログラム数は非常に多くなる可能性があります。したがって、よく知られたポートでポートマッパを実行しておけば、その他の遠隔プログラムのポート番号はポートマッパに問い合わせることによって得られます。図 E–1 では a、111、b、c はポート番号を表しています。111 はポートマッパに割り当てられたポート番号です。

ポートマッパはブロードキャスト RPC にも役立ちます。特定の RPC プログラムは、マシンが異なると通常別のポート番号に結合されますので、直接これらすべてのプログラムにブロードキャストする方法はありません。これに対して、ポートマッパは固定のポート番号を持っています。そこで、特定のプログラムにブロードキャストするには、クライアントはブロードキャストアドレスにあるポートマッパにメッセージを送信します。各ポートマッパは、ブロードキャストを受けて、クライアントが指定しているローカルサービスを呼び出します。portmap はローカルサービスからの応答を取り出すと、それをクライアントに送信します。portmap プロトコル仕様は、コード例 E–1 に示します。

コード例 E–1 portmap プロトコル仕様 (RPC 言語で記述)

```
const PMAP_PORT = 111;        /* ポートマッパのポート番号 */
 /*
  * (プログラム、バージョン、プロトコル) とポート番号のマッピング
  */
 struct pmap {
  rpcprog_t prog;
  rpcvers_t vers;
  rpcprot_t prot;
  rpcport_t port;
 };
 /*
  * prot フィールドに指定できる値
  */
 const IPPROTO_TCP = 6; /* TCP/IP のプロトコル番号 */
 const IPPROTO_UDP = 17; /* UDP/IP のプロトコル番号 */
 /*
  * マッピングリスト
  */
 struct pmaplist {
  pmap map;
  pmaplist *next;
 };
```

(続く)

```
/*
 * callit への引数
 */
struct call_args {
  rpcprog_t prog;
rpcvers_t vers;
rpcproc_t proc;
 opaque args<>;
};
/*
* callit からの戻り値
*/
struct call_result {
 rpcport_t port;
 opaque res<>;
};
/*
* ポートマッパの手続き
*/
program PMAP_PROG {
 version PMAP_VERS {
  void
  PMAPPROC_NULL(void) = 0;
  bool
  PMAPPROC_SET(pmap) = 1;
  bool
  PMAPPROC_UNSET(pmap) = 2;
  unsigned int
  PMAPPROC_GETPORT(pmap) = 3;
  pmaplist
  PMAPPROC_DUMP(void) = 4;
  call_result
  PMAPPROC_CALLIT(call_args) = 5;
 } = 2;
} = 100000;
```

portmap の操作

portmap が現在サポートしているプロトコルは 2 つ (TCP/IP と UDP/IP) です。portmap にアクセスするには、どちらかのプロトコルで割り当てられたポート番号 111 (SUNRPC (5)) と通信します。ポートマッパの各手続きを次に説明します。

PMAPPROC_NULL

この手続きは何もしない手続きです。習慣的に、どのプロトコルでも手続き 0 は引数も戻り値もない手続きにします。

PMAPPROC_SET

プログラムは、プログラム番号 *prog*、バージョン番号 *vers*、トランスポートプロトコル番号 *prot*、サービス要求を受信するポート *port* を引き渡します。この手続きは、指定したポートに既にマッピングが存在していて結合されていれば、マッピングの設定を拒絶します。マッピングが存在していてポートが未結合の場合は、ポートを登録解除し、要求されたマッピングを設定します。マッピングが正しく設定されれば TRUE、設定されなければ FALSE を返します。rpc_soc(3N) のマニュアルページの pmap_set() 関数も参照してください。

PMAPPROC_UNSET

プログラムが使用不可能になれば、同一マシン上のポートマッパプログラムで自身の登録を解除しなければなりません。そのとき渡す引数と戻り値は、PMAPPROC_SET と同じです。引数のうち、プロトコルとポート番号のフィールドは無視されます。rpc_soc(3N) マニュアルページの pmap_unset() 関数も参照してください。

PMAPPROC_GETPORT

プログラム番号 *prog*、バージョン番号 *vers*、トランスポートプロトコル番号 *prot*、をこの手続きに引き渡すと、そのプログラムが呼び出し要求を待っているポート番号が返されます。ポート番号 0 が返された場合は、そのプログラムが登録されていないことを示します。引数の port フィールドは無視されます。rpc_soc(3N) マニュアルページの pmap_getport() 関数も参照してください。

PMAPPROC_DUMP

この手続きは、ポートマッパのデータベースの全エントリをリストします。この手続きへの引数はなく、返されるのは、プログラム、バージョン、プロトコル、ポート番号のリストです。rpc_soc(3N) マニュアルページの pmap_getmaps() 関数も参照してください。

PMAPPROC_CALLIT

この手続きを使用すると、呼び出し側がポートマッパと同一マシン上の遠隔手続きのポート番号を知らなくても、その遠隔手続きを呼び出すことができます。この手続きは、よく知られているポートマッパのポートを通して、任意の遠隔プログラムへのブロードキャストを可能にするためのものです。引数の *prog*、*vers*、*proc*、*args* はそれぞれプログラム番号、バージョン番号、手続き番号、遠隔手続きへの引数です。`rpc_soc(3N)` マニュアルページの `pmap_rmtcall()` 関数も参照してください。

この手続きは正常終了したときだけ応答を返し、異常終了したときは応答を返しません。また、遠隔プログラムのポート番号と、遠隔手続きからの戻り値を返します。

ポートマッパが遠隔プログラムとの通信に使用できるのは `UDP/IP` だけです。

参考文献

1. 『Implementing Remote Procedure Calls』 Birrell, Andrew D.、Nelson, B ruce Jay 共著、XEROX 社 CSL-83-7、1983 年 10 月
2. 『VMTP: Versatile Message Transaction Protocol, Preliminary Version 0.3』 Cheriton, D.著、Stanford 大学、1987 年 1 月
3. 『Transmission Control Protocol - DARPA Internet Program Protocol Spec ification, RFC 793』 Postel, J. 著、Information Sciences Institute、1981 年 9 月
4. 『User Datagram Protocol, RFC 768』 Postel, J. 著、Information Sciences Institute、1980 年 8 月
5. 『Assigned Numbers, RFC 923』Reynolds, J. & Postel, J.著、Information Sciences Institute、1984 年 10 月

付録 **F**

SAF を使用したポートモニタプログラムの作成

この付録は、ポートモニタがサービスアクセス機能 (SAF : Service Access Facility) とサービスアクセスコントローラ (SAC) の下で走る時に実行する必要のある各機能について簡単に説明します。

SAF の概要

SAF を使用すると、サービスアクセスの手続きが汎用化され、ローカルシステムへのログインアクセスも、ローカルサービスへのネットワークアクセスも同じ方法で管理できます。SAF を使用すると、システムがサービスにアクセスするときに、ttymon を始めとする各種のポートモニタ、リスナ、ユーザーアプリケーションのために特別に作成したポートモニタを使用できます。

ポートモニタがどのようにアクセスポートの監視と管理を行うかは、SAF から決まるのではなく個々のポートモニタで決まります。したがって、ユーザーは独自のポートモニタを開発してインストールすれば、システムを希望に応じて拡張できます。このようにユーザーが自由に拡張できるという点は、SAF の大きな特徴の一つです。

SAF については、サービスは要求に応じて起動するプロセスにすぎず、サービスの提供する機能に制限はありません。

SAF は、制御プロセスであるサービスアクセスコントローラ (SAC) と、サポートディレクトリ構造の 2 つのレベルに対応する 2 つの管理レベルとで構成されます。上位の管理レベルはポートモニタを管理し、下位レベルはサービスを管理します。

管理レベルでみると、SAF は次の要素で構成されています

- SAC
- システムごとの構成スクリプト
- SAC 管理ファイル
- SAC 管理コマンド sacadm
- ポートモニタ
- ポートモニタごとの構成スクリプト (オプション)
- ポートモニタごとの管理ファイル
- 管理コマンド pmadm
- サービスごとの構成スクリプト (オプション)

SAC の概要

SAC は SAF の制御プロセスです。SAC は /etc/inittab のエントリに入れて init() で起動します。SAC の機能は、ポートモニタを、システム管理者が指定した状態に保持することです。

管理コマンド sacadm を使用して、SAC にポートモニタの状態を変更させることができます。また、sacadm を使用して、ポートモニタを SAC の管理下に入れたり、管理から外したり、SAC が管理しているポートモニタの情報をリストしたりできます。

SAC の管理ファイルには、SAC の管理下の各ポートモニタを一意的に識別できるタグと、各ポートモニタを起動するコマンドのパス名が入っています。

SAC が実行する主な機能を簡単に示します。

- 環境のカスタマイズ
- 適切なポートモニタの起動
- 管理下のポートモニタをポーリングして、必要に応じて回復手続きを実行

ポートモニタの基本機能

ポートモニタは、マシン上の同タイプの入力ポートセットを監視するプロセスです。ポートモニタの主目的は、外部からのサービス要求を検出し、それを適切にディスパッチすることです。

ポートとは、外部に開かれたシステム上のアクセスポイントです。ポートには具体的には、ネットワーク上のアドレス (TSAP や PSAP)、コードで接続された端末ライン、電話線入力などがあります。何をポートとみなすかは、ポートモニタ自体が定義します。

ポートモニタはいくつかの基本機能を実行します。機能によっては SAF のために必要なものもありますし、ポートモニタ自身が必要に応じて設計した機能もあります。

ポートモニタの主な機能を 2 つ示します。

- ポートの管理
- ポートに発生するアクティビティの監視

ポート管理

ポートモニタの第 1 の機能はポートの管理です。ポートを実際にどう管理するかの詳細は、ポートモニタの開発者が定義します。1 つのポートモニタで 1 つのポートを管理するとは限りません。同時に複数のポートを管理することもできます。

注 - ポート管理の例として、電話回線の接続における回線速度の設定、適切なネットワークアドレスとの結合、サービス終了後のポートの再初期化、プロンプトの出力などがあります。

アクティビティの監視

ポートモニタの第 2 の機能は、アクティビティ指示に対応する責任があるポートを監視することです。検出されるアクティビティには、次の 2 つのタイプがあります。

1. アクティビティの 1 つは、ポートモニタがモニタ固有のアクションを取るための指示です。ポートモニタ固有アクティビティの例として、ブレークキーが押されると回線速度を循環させるという指示があります。すべてのポートモニタが、同じ指示を認識して同じように反応するわけではありません。ポートモニタがどのような指示を検出するかは、ポートモニタの開発者が定義します。
2. もう 1 つは外部からのサービス要求です。ポートモニタはサービス要求を受信すると、受信したポートからどのサービスが要求されているかを判定できなければなりません。同じサービスを複数のポートで提供できることに注意してください。

ポートモニタのその他の機能

この項では、ポートモニタのその他の機能について簡単に説明します。

システムへのアクセス制限

ポートモニタは現在実行中のサービスに影響することなく、システムへのアクセスを制限できなければなりません。そのために、ポートモニタは有効と無効の 2 つの内部状態を保持しています。ポートモニタの起動時の状態は、sac で設定した環境変数 ISTATE の値で決まります。335ページの「SAC とポートモニタのインタフェース」 の項を参照してください。

ポートモニタの有効と無効を切り換えると、その管理下にあるすべてのポートが影響を受けます。ポートモニタが 1 つのポートだけを管理している場合は、そのポートだけが影響を受けます。複数ポートがそのポートモニタの管理下にあるときは、それらのポートすべてが影響を受けます。

ポートモニタの有効と無効の切り換えは動的な操作です。この操作で、ポートモニタの内部状態が変わります。ただし、次にポートモニタを起動するときには、この状態は失われます。

これに対して、個々のポートの有効と無効の切り換えは静的な操作です。この操作によって管理ファイルが変更されます。次にポートモニタを起動しても、ポートの状態は残ります。これに対して、個々のポートの有効と無効の切り換えは静的な操作です。この操作によって管理ファイルが変更されます。次にポートモニタを起動しても、ポートの状態は残ります。

utmp エントリの作成

ポートモニタは、起動するサービスの utmp エントリを作成する責任があります。そのとき、エントリのタイプフィールドは USER_PROCESS に設定します。ただし、pmadm でサービスを追加したときに –fu を指定した場合だけです。utmp エントリは、次にサービスによって変更されます。サービスが終了したときは、utmp エントリは DEAD_PROCESS に設定されなければなりません。

ポートモニタのプロセス ID とファイルのロック

ポートモニタは、起動したときに自分のプロセス ID をカレントディレクトリの _pid というファイルに書き込み、そのファイルにアドバイザリロックを設定します。

サービス環境の変更: doconfig() の実行

ポートモニタは、ポートモニタの管理ファイル _pmtab に指定されているサービスを起動する前に、サービスごとの構成スクリプトがあればライブラリ関数 doconfig() を呼び出してそれを実行します。セキュリティ上の理由だけではなく、サービスごとの構成スクリプトで制限コマンドの実行が指示される場合があるので、ポートモニタはルートアクセス権で起動されます。サービスの呼び出し方法の詳細は、ポートモニタの開発者が定義します。

ポートモニタの終了

ポートモニタはシグナル SIGTERM を受け取ると、手続きを守って自分自身を終了させます。ポートモニタの終了シーケンスを次に示します。

1. ポートモニタは停止状態に入ります。これ以降はサービス要求を受け取りません。
2. ポートモニタの状態を有効に切り換える指示はすべて無視します。
3. ポートモニタは、管理下の全ポートの制御を明け渡します。ポートモニタの以前のインスタンスが停止状態のときは、新たなインスタンスが正常に起動できなければなりません。

4. プロセス ID ファイルのアドバイザリロックを解除します。アドバイザリロックが解除されると、プロセス ID ファイルの内容は未定となり、ポートモニタを新たに起動できるようになります。

SAF ファイル

この項では、SAF によって使用されるファイルについて簡単に説明します。

ポートモニタの管理ファイル

ポートモニタのカレントディレクトリには、_pmtab という管理ファイルがあります。_pmtab は、pmadm コマンドとポートモニタ固有の管理コマンドとを使用して管理します。ポートモニタ固有の管理コマンドについては次に説明します。

注 - ポートモニタ listen に固有の管理コマンドは nlsadmin() です。ポートモニタ ttymon に固有の管理コマンドは ttyadm() です。ユーザーがポートモニタを作成するときは、これらのコマンドと同様の機能を持つ、ポートモニタ固有の管理コマンドを提供する必要があります。

サービスごとの構成ファイル

ポートモニタのカレントディレクトリには、サービスごとの構成スクリプトも含まれています (サービスごとの構成スクリプトはオプションのためでない場合もあります)。サービスごとの構成スクリプトの名前は、_pmtab ファイルで設定したサービスタグと同じです。

ポートモニタのプライベートファイル

ポートモニタは、ディレクトリ /var/saf/tag (tag はポートモニタの名前) にプライベートファイルを作成できます。プライベートファイルの例として、ログファイルや一時ファイルがあります。

SAC とポートモニタのインタフェース

sac は、起動するポートモニタごとに次の 2 つの環境変数を作成します。

1. PMTAG
2. ISTATE

   sac は、この変数を一意的に識別できるポートモニタタグに設定します。ポートモニタは sac メッセージに応答するときに、このタグを使用して自分自身を示します。ISTATE は、ポートモニタの起動時の内部状態を指定するのに使用します。ISTATE は、ポートモニタが許可状態でスタートするときに enable に、禁止状態でスタートするときに disable に設定されます。sac はポートモニタの正常なポーリングを定期的に実行します。

sac とポートモニタは FIFO を使用して通信します。ポートモニタは、sac からのメッセージを受信するためにカレントディレクトリに _pmpipe をオープンし、sac に応答メッセージを返すために ../_sacpipe をオープンします。

メッセージ形式

この節では、sac からポートモニタに送るメッセージ (sac メッセージ) と、ポートモニタから sac に送るメッセージ (ポートモニタメッセージ) について説明します。メッセージは、FIFO を通して C の構造体形式で送信されます。コード例 F–2 を参照してください。

sac メッセージ

sac から送られるメッセージの形式は、次の構造体 sacmsg で定義されています

```
struct sacmsg {
  int sc_size; /* オプションデータ部分のサイズ */
  char sc_type; /* メッセージのタイプ */
};
```

sac からポートモニタに送られるメッセージには 4 つのタイプがあります。メッセージがどのタイプなのかは、sacmsg 構造体の sc_type フィールドに次のどれかの値を設定して示します。

SC_STATUS　　ステータス要求

SC_ENABLE	有効設定メッセージ
SC_DISABLE	無効設定メッセージ
SC_READDB	ポートモニタの _pmtab ファイルを読み込むように指示するメッセージ

sc_size はメッセージのオプションデータ部分のサイズを示します。337ページの「メッセージクラス」 を参照してください。sc_size では、sc_size は常に 0 です。

ポートモニタは、sac からのメッセージには必ず応答しなければなりません。

ポートモニタメッセージ

ポートモニタから sac に送られるメッセージの形式は、次の構造体 pmmsg で定義されています。

```
struct pmmsg {
  char pm_type;                 /* メッセージのタイプ */
  unchar pm_state;              /* ポートモニタの現在の状態 */
  char pm_maxclass;             /* このポートモニタが解釈できる
                                   最大メッセージクラス */
  char pm_tag[PMTAGSIZE + 1];   /* ポートモニタのタグ */
  int pm_size;                  /* オプションデータ部分のサイズ */
};
```

ポートモニタから sac に送られるメッセージには、2 つのタイプがあります。メッセージがどのタイプなのかは、pmmsg 構造体の pm_type フィールドに次のどちらかの値を設定して示します。

PM_STATUS	状態情報
PM_UNKNOWN	否定応答

どちらのタイプのメッセージの場合も、pm_tag フィールドにはポートモニタのタグが、pm_state フィールドにはポートモニタの現在の状態が設定されます。pm_state フィールドに設定できる状態は次の 4 つです。

PM_STARTING	起動
PM_ENABLED	有効

PM_DISABLED	無効
PM_STOPPING	停止

現在の状態は、sac からの最後のメッセージによる状態変更を反映しています。

通常の応答メッセージは状態メッセージです。否定応答メッセージを返すのは、sac から受信したメッセージが理解できなかったときだけです。

pm_size はメッセージのオプションデータ部分のサイズを示します。pm_maxclass はメッセージクラスを指定するのに使用します。この 2 つのフィールドは、337ページの「メッセージクラス」 の項で説明します。Solaris では常に pm_maxclass は 1、sc_size は 0 です。

ポートモニタから先にメッセージを送ることはありません。ポートモニタは、受信したメッセージに応答するだけです。

メッセージクラス

メッセージクラスの概念は、SAF の拡張性に含まれています。これまでに説明したメッセージはすべてクラス 1 のメッセージです。どのメッセージにも可変長データは含まれておらず、必要な情報はすべてメッセージヘッダーに入っていました。

新たなメッセージをプロトコルに加えると、それによって新たなメッセージクラス (たとえば、クラス 2) が定義されます。sac からポートモニタに送られる最初のメッセージは常にクラス 1 のメッセージです。どのポートモニタもクラス 1 のメッセージは理解できますから、sac が送る最初のメッセージは必ず理解されます。ポートモニタは、それに対する sac への応答の中で、ポートモニタが理解できる最大メッセージクラス番号を pm_maxclass フィールドに設定します。sac は、ポートモニタに対して、pm_maxclass の値より大きいクラスのメッセージは送りません。ポートモニタが理解できるクラスより上のクラスのメッセージを必要とする要求は、失敗に終わります。Solaris では、pm_maxclass は常に 1 です。

注 - どのポートモニタも、pm_maxclass の値と等しいクラスかそれより小さいクラスのメッセージを受信できます。したがって、pm_maxclass フィールドが 3 の場合、このポートモニタはクラス 1、2、3 のメッセージを理解できます。ポートモニタはメッセージを生成することはありません。受信したメッセージに応答するだけです。ポートモニタからの応答は、元のメッセージと同じクラスでなければなりません。

sac の側からだけメッセージが生成されるので、sac が生成できるクラスより上のクラスのメッセージをポートモニタが処理できる場合も、このプロトコルは正しく機能します。

pm_size (pmmsg 構造体の要素) と sc_size (sacmsg 構造体の要素) は、メッセージの中のオプションデータ部分のサイズを示します。オプションデータ部分の形式は未定義です。オプションデータ部分の形式定義は、メッセージのタイプによって決まっています。Solaris では、sc_size と pm_size は常に 0 です。

ポートモニタの管理インタフェース

この項では、SAC で使用できる管理ファイルについて説明します。

SAC の管理ファイル _sactab

サービスアクセスコントローラの管理ファイルには、SAC の管理下にある全ポートモニタの情報が入っています。このファイルは配布システムに入っています。最初、このファイルには SAC のバージョン番号の入ったコメント行が 1 行入っているだけです。システムにポートモニタを追加するときは、SAC の管理ファイルにエントリを作成します。エントリを追加するには、管理コマンド sacadm に –a オプションを付けて実行します。SAC の管理ファイルからエントリを削除するときも、sacadm を使用します。

SAC の管理ファイルの各エントリには、表 F–1 で示されている次の情報が含まれています。

表 F–1 SAC_sactab ファイル

フィールド	説明
PMTAG	個々のポートモニタを一意的に識別できるタグ。ポートモニタの命名はシステム管理者が行う。SAC は、管理目的でポートモニタを識別するときはこのタグを使用する。PMTAG には、14 文字までの英数字が入る。
PMTYPE	ポートモニタのタイプ。各ポートモニタには、それを一意に識別できるタグのほかにタイプ指示子がある。タイプ指示子は、同一エンティティを別々に起動したことによるポートモニタグループを示す。有効なポートモニタタイプの例として ttymon と listen が挙げられる。タイプ指示子を使用すると、関連ポートモニタからなるグループの管理が楽になる。タイプ指示子がないと、システム管理者は各ポートモニタタグがどのタイプのポートモニタなのか判断できない。PMTYPE には、14 文字までの英数字が入る。
FLGS	現在定義されているフラグは次の 2 つがある。 –d 起動時にポートモニタを有効にしない。 –x ポートモニタを起動しない。 フラグを指定しないと、デフォルトのアクションが実行される。デフォルトでは、ポートモニタは起動され有効に設定される。
RCNT	ポートモニタが何回エラーを起こしたらエラー停止状態になるかを示す。SAC は、エラー停止状態になったポートモニタは再起動しない。エントリ作成時にこの回数を指定しないと、このフィールドは 0 になる。再起動回数が 0 ということは、ポートモニタが一度でもエラーストップすると、以後再起動されないことを意味する。
COMMAND	ポートモニタを起動するコマンドの文字列。文字列の最初の要素 (コマンド自身) は、フルパスで指定しなければならない。

ポートモニタの管理ファイル _pmtab

各ポートモニタは、それぞれ 2 つの専用ディレクトリを持ちます。カレントディレクトリには、SAF で決められたファイル (_pmtab、_pid) と、サービスごとの構成スクリプト (作成した場合のみ) とがあります。ディレクトリ /var/saf/pmtag (pmtag はポートモニタのタグ) には、ポートモニタのプライベートファイルを入れることができます。

各ポートモニタは自分の管理ファイルを持っています。管理ファイルのサービスエントリの追加、削除、変更を行うときは、pmadm コマンドを使用します。pmadm で管理ファイルを変更するたびに、対応するポートモニタはそのファイルを読み直さなければなりません。ポートモニタの管理ファイルの各エントリでは、個々のポートの扱いと、そのポートで呼び出されるサービスとを定義します。

どのタイプのポートモニタでも省略できないフィールドがあります。エントリには、サービスを一意的に識別するためのサービスタグと、サービスが起動されたときに割り当てられる ID (たとえば、root) が必ず入っていなければなりません。

注 - サービスタグとポートモニタタグを組み合わせると、サービスインスタンスが一意的に定義されます。別のポートモニタの下のサービスを識別するのに、同じサービスタグを使用することができます。レコードにはポートモニタに固有のデータも入っています。たとえば、ポートモニタ ttymon の場合、ttymon にだけ必要なプロンプト文字列が入っています。ポートモニタのタイプごとに、ポートモニタの固有データを引数とし、そのデータを適当な形式で管理ファイルの中に書き出すコマンドが提供されていなければなりません。ttymon に対しては ttyadm コマンドが、listen に対しては nlsadmin コマンドがこの処理を行います。ユーザーがポートモニタを作成するときは、このような管理コマンドも同時に作成する必要があります。

ポートモニタ管理ファイルの各サービスエントリには、以下にリストする情報が次の形式で入っていなければなりません。

```
svctag:flgs:id:reserved:reserved:reserved: pmspecific# comment
```

SVCTAG はサービスを一意的に識別するタグです。タグは、サービスが提供されるポートモニタの中でのみ一意になるようにします。別のポートモニタのサービス (同一サービスまたは別のサービス) には、同じタグが付けられていてもかまいません。サービスを一意的に識別するには、ポートモニタタグとサービスタグの両方が必要です。

SVCTAG には、14 文字までの英数字が入ります。サービスエントリは、表 F–2 で定義されています。

表 F–2 SVCTAG サービスエントリ

サービスエントリ	説明
FLGS	このフィールドには、現在次のフラグを入れることができる。 –x このポートは有効にしない。デフォルトでは有効になる。 –u このサービスの utmp エントリを作成する。デフォルトではサービスの utmp エントリは作成されない。
ID	サービスを起動するときの ID。ID は、/etc/passwd に入っているログイン名と同じ形式を持つ。
PMSPECIFIC	ポートモニタ固有情報の例として、アドレス、実行するプロセスの名前、接続を渡す STREAMS パイプの名前が挙げられる。この情報はポートモニタのタイプごとに異なる。
COMMENT	サービスエントリに関するコメントを書く。

注 - サービス呼び出し方法からみて、utmp を作成するのが適当でない場合は、ポートモニタが –u フラグを無視することに注意してください。サービスによっては、utmp エントリが作成されていないと正しく起動できないものもあります (たとえば、login サービス)。

各ポートモニタの管理ファイルには、次の形式の特殊なコメントが入っていなければなりません。

```
# VERSION=value
```

ここで、value はポートモニタのバージョン番号を表す整数です。バージョン番号により、ポートモニタの管理ファイルの形式がわかります。このコメント行は、ポートモニタをシステムに追加したときに自動的に作成されます。これだけが 1 行となってサービスエントリの前に入ります。

SAC 管理コマンド sacadm

sacadm は、SAF 階層の上位レベル (すなわち、ポートモニタ管理) に対する管理コマンドです (sacadm(1M) のマニュアルページを参照)。SAF では、sacadm コマンドを使用して SAC の管理ファイルを変更することによりポートモニタを管理します。sacadm は、次にのような機能があります。

- 指定されたポートモニタ情報を SAC 管理ファイルから取り出して印刷する
- ポートモニタの追加と削除を行う
- ポートモニタの有効・無効を切り換える
- ポートモニタの起動と停止を行う
- システムごとの構成スクリプトのインストールまたは置き換えを行う
- ポートモニタごとの構成スクリプトのインストールまたは置き換えを行う
- 管理ファイルを読み直すよう SAC に指示する

ポートモニタの管理コマンド pmadm

pmadm は、SAF 階層の下位レベル (すなわち、サービス管理) に対する管理コマンドです (pmadm(1M) のマニュアルページを参照)。複数のポートで同じサービスを提供することは可能ですが、1 つのポートには 1 つのサービスしか結合できません。pmadm の機能を次に示します。

- 指定されたサービスステータス情報をポートモニタの管理ファイルから取り出して印刷する
- サービスの追加と削除を行う
- サービスの有効と無効を切り換える
- サービスの有効と無効を切り換える

サービスインスタンスを一意的に識別するには、pmadm コマンドで、サービス (–s) と、サービスが提供されるポートモニタ (–p または –t) の両方を指定する必要があります。

モニタ固有の管理コマンド

前の節では、_pmtab ファイル内の 2 つの情報、すなわち、ポートモニタのバージョン番号と、_pmtab ファイルのサービスエントリに入っているポートモニタ情報とを説明しました。新たなポートモニタを追加するときは、_pmtab ファイルを正しく初期化するためにバージョン番号がわかっていなければなりません。新たなサービスを追加するときは、_pmtab エントリのポートモニタ情報が正しくフォーマットされなければなりません。 各ポートモニタには、この 2 つの処理を行うための管理コマンドが必要です。ポートモニタの設計者は、そのような管理コマンドとその入力オプションも定義しなければなりません。

管理コマンドは、そのような入力オプションを指定して呼び出されると、サービスエントリのポートモニタ固有部分に必要な情報を、ポートモニタの _pmtab ファイルに入るようにフォーマットし、それを標準出力に書き出さなければなりません。バージョン番号を調べるときは、コマンドに –V オプションを付けて実行します。その場合管理コマンドは、ポートモニタの現在のバージョン番号を標準出力に書き出さなければなりません。

どちらの処理を実行している場合も、管理コマンドに何らかのエラーが起これば、標準出力には何も表示されません。

ポートモニタとサービスのインタフェース

ポートモニタとサービスのインタフェースは、サービスの側から決まります。サービスを呼び出す方法の例として、2 つの方法を説明します。

サービスの新規呼び出し

インタフェースの 1 つに、要求ごとに新たにサービスを起動する方法があります。この場合、まずポートモニタに子プロセスの `fork()` を要求します。子プロセスは、`exec()` を実行することで、指定されたサービスになります。ポートモニタは、`exec()` が起こる前に、ポートモニタ固有のアクションを実行します。ただし、サービスごとの構成スクリプトがあれば、必ずそれを読み込んで実行しなければなりません。そのためには、ライブラリルーチン `doconfig()` を呼び出します。

実行中のサービスの呼び出し

現在実行中のサービスを呼び出すためのもう 1 つのインタフェースがあります。このインタフェースを使用するサービスは、ストリームパイプの一端をオープンにしておき、そこを通して接続を受信できるようにしておかなければなりません。

ポートモニタに必要な条件

ポートモニタを開発するには、いくつかの一般的な条件が満たされていなければなりません。この節では、そのような条件を簡単に説明します。ポートモニタだけでなく、管理コマンドも同時に作成する必要があります。

起動時の環境

ポートモニタが起動されるときは、次のような初期実行環境が整っていなければなりません。

- オープンしているファイル記述子を持たない
- プロセスグループリーダにはならない
- /etc/utmp に、タイプが LOGIN_PROCESS のエントリを持つ
- 環境変数 ISTATE が「enabled」か「disabled」のどちらかに設定されており、ポートモニタの正しい初期状態を示す
- 環境変数 PMTAG が、ポートモニタに割り当てられたタグに設定されている
- ポートモニタの管理ファイルの入ったディレクトリがカレントディレクトリになっている
- ポートモニタは、/var/saf/tag ディレクトリにプライベートファイルを作成できる (ここで、tag はポートモニタのタグ)
- ポートモニタは、ユーザー ID が 0 (root) で実行される

重要なファイル

カレントディレクトリには、次に示すポートモニタの主要ファイルが存在します。

表 F–3 ポートモニタの主要ファイル

ファイル	説明
_config	ポートモニタの構成スクリプト。ポートモニタの構成スクリプトは SAC が実行する。SAC は init() により起動される。そのためには、/etc/inittab に sac を呼び出すためのエントリを入れておく
_pid	ポートモニタが自分のプロセス ID を書き込むファイル
_pmtab	ポートモニタの管理ファイル。このファイルには、ポートモニタの管理下にあるポートとサービスの情報が入っている
_pmpipe	ポートモニタが sac からのメッセージを受信するための FIFO

表 F–3　ポートモニタの主要ファイル　続く

ファイル	説明
svctag	svctag というタグを持つサービスの構成スクリプト
../_sacpipe	ポートモニタが sac へのメッセージを送信するための FIFO

ポートモニタの実行すべきタスク

ポートモニタは、ポートモニタ固有機能のほかに次のタスクを実行する責任があります。

- 自分のプロセス ID をファイル _pid に書き込み、そのファイルにアドバイザリロックをかける
- シグナル SIGTERM を受信すると、手順に従って終了する
- sac とのメッセージ交換プロトコルに従う

ポートモニタはサービスを起動するときに次のタスクを実行しなければなりません。

- 要求されたサービスに対する _pmtab エントリに「–u」フラグがあれば、utmp エントリを作成する

注 - サービスの起動方法から見て utmp エントリを作成する意味がなければ、ポートモニタは「–u」フラグを無視します。これとは反対に、utmp エントリが作成されていないと正しく起動できないサービスもあります。

- 要求されたサービスに対して、サービスごとの構成スクリプトがあれば、ライブラリルーチン doconfig() を呼び出してそれを解釈する。

構成ファイルとスクリプト

構成スクリプトのインタプリタ: doconfig()

libnsl.so に定義されているライブラリルーチン doconfig() は、ファイル /etc/saf/pmtag/_sysconfig (システムごとの構成スクリプト)、/etc/saf/_sysconfig (ポートモニタごとの構成スクリプ

ト)、/etc/saf/pmtag/svctag (サービスごとの構成スクリプト) に入っている構成スクリプトを解釈します。doconfig() の構文を次に示します。

```
# include <sac.h>
int doconfig (int fd, char *script, long rflag);
```

script は構成スクリプト名です。fd は、ストリーム操作オペレーションが適用されるストリームのファイル記述子です。rflag は、script を解釈するモードを指定するビットマスクです。rflag に指定できる値は、NORUN か NOASSIGN、またはこの 2 つの OR を取った値です。rflag がゼロの場合は、構成スクリプトのすべてのコマンドが解釈されます。rflag の NOASSIGN ビットがオンになっていると、assign コマンドは解釈できず doconfig() はエラー終了します。rflag の NORUN ビットがオンになっていると、run と runwait のコマンドは解釈できず doconfig() はエラー終了します。

スクリプトのどれかのコマンドでエラーが起こった場合、doconfig() はそこでスクリプトの解釈を終了し正の整数を返します。この値は、エラーが起こった行番号を表します。システムエラーが起こった場合は、-1 を返します。

スクリプトでエラーが起こった場合は、スクリプトが実行環境を設定していたプロセスは起動されません。

次の例では、サービスごとの構成スクリプトを解釈するのに doconfig() を使用しています。

```
. . .
  if ((i = doconfig (fd, svctag, 0)) != 0){
  error ("doconfig failed online %d of script %s",i,svctag);
}
```

システムごとの構成ファイル

システムごとの構成ファイル /etc/saf/_sysconfig は空の状態で配布されます。このファイルはシステム上の全サービスの環境をカスタマイズするもので、この章と doconfig(3N) のマニュアルページで説明するインタプリタ言語で書かれたコマンドスクリプトが入っています。SAC は、起動されると doconfig() 関数を呼び出してシステムごとの構成スクリプトを翻訳します。SAC が起動されるのは、システムがマルチユーザーモードに入ったときです。

ポートモニタごとの構成ファイル

ポートモニタごとの構成ファイル (/etc/saf/pmtag/_config) はオプションの構成ファイルです。ユーザーはこのファイルを使用して、ポートモニタの環境と、

ポートモニタが管理している特定のポートで提供されるサービスの環境とをカスタマイズできます。ポートモニタごとの構成スクリプトも、システムごとの構成スクリプトと同じ言語で書かれます。

ポートモニタごとの構成スクリプトは、ポートモニタが起動されたときに翻訳されて実行されます。ポートモニタが SAC に起動されるのは、SAC が起動されて自分自身の構成スクリプト (/etc/saf/_sysconfig) を実行した後です。

ポートモニタごとの構成スクリプトがあると、システムごとの構成スクリプトで提供されるデフォルトの構成スクリプトではなく、ポートモニタごとのスクリプトが実行されます。

サービスごとの構成ファイル

ユーザーは、サービスごとの構成ファイルを使用して特定のサービスの環境をカスタマイズできます。たとえば、一般ユーザーには許されていない特殊な特権を必要とするサービスがあるとします。doconfig(3N) のマニュアルページに記述されている言語を使用すると、特定のポートモニタで提供される特定のサービスにこのような特殊な特権を与えたり奪ったりするスクリプトを書くことができます。

サービスごとの構成スクリプトを作成しておくと、上位レベルの構成スクリプトで提供されるデフォルトのスクリプトの代わりに、そのスクリプトが実行されます。たとえば、サービスごとの構成スクリプトで、デフォルトとは異なる STREAMS モジュールセットを指定することができます。

構成スクリプト言語

構成スクリプトを記述する言語は、一連のコマンドで構成されており、各コマンドはそれぞれ個別に翻訳されます。5 つのキーワード
assign、push、pop、runwait、run が定義されています。コメント文字は # です。空白行は無視されます。スクリプトの各行は 1024 文字を超えてはいけません。

assign *variable=value*

assign コマンドは環境変数を定義するのに使用します。*variable* は環境変数名で、*value* は環境変数に割り当てられる値です。*value* は文字列定数でなければなりません。パラメータ置換は使用できません。*value* は引用符で囲むことができます。引用符で囲むときの規則は、シェルで環境変数を定義するときの規則と同じです。

新たな環境変数を割り当てるための空間が足りない場合、指定構文に誤りがあるときは assign コマンドでエラーが起こります。

```
push module1[,
 module2, module3, ...]
```

push コマンドは、STREAMS モジュールを fd で指定されたストリームにプッシュします。doconfig(3N) マニュアルページを参照してください。module1 は最初にプッシュされるモジュール名、module2 は 次にプッシュ されるモジュール名です。(module3 以下も同様)。指定したどれかのモジュールがプッシュできなかったとき、このコマンドはエラーとなります。その場合、同じ行で指定している残りのモジュールは無視され、既にプッシュされたモジュールはポップされます。

```
pop [module]
```

pop コマンドは、指定したストリームから STREAMS モジュールをポップするのに使用します。引数なしで pop コマンドを実行すると、ストリームの一番上のモジュールがポップされます。引数を指定して pop コマンドを実行すると、指定したモジュールがストリームの一番上に来るまで、モジュールが 1 つずつポップされます。指定したモジュールがストリームにない場合は、ストリームはもとの状態のままで、コマンドはエラー終了します。モジュール名の代わりに特殊キーワード ALL を指定すると、ストリームから全モジュールがポップされます。ただし、一番上のドライバより上のモジュールだけが対象になることに注意してください。

runwait *command*

runwait コマンドは、コマンドを実行してその終了を待ちます。*command* には、実行するコマンドのパス名を指定します。*command* は /bin/sh –c を付けて実行されます。シェルスクリプトもこのようにして構成スクリプトから実行できます。指定した *command* が見つからないか実行できなかったとき、または、*command* は存在していてもステータスが 0 以外のときは、runwait コマンドがエラー終了します。

run *command*

run コマンドは runwait コマンドと同じですが、run コマンドでは *command* の終了を待たない点が違います。*command* には、実行するコマンドのパス名を指定します。run コマンドがエラー終了するのは、*command* を実行する子プロセスを作成できなかったときだけです。

構文上は区別がつきませんが、run と runwait で実行されるコマンドのいくつかはインタプリタの組み込みコマンドです。インタプリタの組み込みコマンドが使用

されるのは、プロセスのコンテキストの中でプロセスの状態を変える必要があるときです。doconfig() の組み込みコマンドは、シェルの特殊コマンドと同様、実行するための別のプロセスを生成しません。sh(1) マニュアルページを参照してください。組み込みコマンドの初期セットを次に示します。

```
cd ulimit umask
```

構成スクリプトの印刷、インストール、置き換え

この節では、SAC とポートモニタの管理コマンドを使用して、3 種類の構成スクリプトをインストールする方法を説明します。システムごとの構成スクリプトとポートモニタごとの構成スクリプトは sacadm コマンドで管理します。サービスごとの構成スクリプトは pmadm コマンドで管理します。

システムごとの構成スクリプト

```
sacadm –G  [ –z  script ]
```

システムごとの構成スクリプトを印刷するか置き換えるときは、–G オプションを使用します。–G オプションだけを指定すると、システムごとの構成スクリプトが印刷されます。–G オプションと –z オプションを組み合わせると、/etc/saf/_sysconfig が script に指定されたファイルの内容で置き換えられます。–G オプションは –z オプション以外のオプションと組み合わせて使用することはできません。

システムごとの構成スクリプトの例

次に示すシステムごとの構成ファイル (_sysconfig) では、時間帯を示す変数 TZ を設定しています。

```
assign TZ=EST5EDT # set TZ
runwait echo SAC is starting > /dev/console
```

ポートモニタごとの構成スクリプト

```
sacadm -g -p pmtag [ -z script ]
```

–g オプションは、ポートモニタごとの構成スクリプトの印刷、インストール、置き換えを行うときに使用します。–g オプションには –p オプションが必要です。–g オプションに –p オプションだけを組み合わせて指定すると、pmtag に指定したポー

トモニタのポートモニタごとの構成スクリプトが印刷されます。–g オプションに –p オプションと –z オプションを組み合わせると、script に指定したファイルがポートモニタ (pmtag) に対するポートモニタごとの構成スクリプトとしてインストールされます。あるいは、/etc/saf/pmtag/_config が既に存在している場合は、そのファイルの内容が script に指定したファイルで置き換わります。–g オプションをこれ以外のオプションと組み合わせて使用することはできません。

ポートモニタごとの構成スクリプトの例

次に示す _config ファイルの例では、/usr/bin/daemon を、STREAMS マルチプレクサを構築するデーモンプロセスを起動するためのコマンドとします。この構成スクリプトをインストールすると、ポートモニタが必要とするコマンドを、ポートモニタの起動の直前に実行できます。

```
# build a STREAMS multiplexor
run /usr/bin/daemon
runwait echo $PMTAG is starting > /dev/console
```

サービスごとの構成スクリプト

```
pmadm -g -p pmtag -s  svctag [ -z script ]
 pmadm -g -s svctag -t type -z script
```

サービスごとの構成スクリプトは、サービスが呼び出される前に、ポートモニタによって解釈されて実行されます。

注 - SAC は自分自身の構成ファイル (_sysconfig) と、ポートモニタの構成ファイルの両方を解釈して実行します。サービスごとの構成ファイルだけは、ポートモニタが実行します。

–g オプションは、サービスごとの構成スクリプトの印刷、インストール、置き換えを行うときに使用します。–g オプションに –p オプションと –s オプションを組み合わせると、ポートモニタ (*pmtag*) で提供されるサービス (*svctag*) のサービスごとの構成スクリプトが印刷されます。–g オプションに –p オプション、–s オプション、–z オプションを組み合わせると、指定したファイル (*script*) に入っているスクリプトが、ポートモニタ (*pmtag*) で提供されるサービス (*svctag*) のサービスごとの構成スクリプトとしてインストールされます。–g オプションに –s オプション、–t オプション、–z オプションを組み合わせると、指定したファイル (*script*) が、ポートモニタタイプ (*type*) で提供されるサービス (*svctag*) のサービスごとの構成スクリプトとしてインストールされます。–g オプションを、上で述べた以外のオプションと組み合わせて使用することはできません。

サービスごとの構成スクリプトの例

次に示すサービスごとの構成スクリプトでは 2 つのことを実行します。1 つは、プロセスの ulimit を 4096 に設定することにより、プロセスが作成するファイルサイズの上限を設定しています。もう 1 つは、umask を 077 に設定することにより、プロセスが作成するファイルに適用される保護マスクを指定しています。

```
runwait ulimit 4096
runwait umask 077
```

ポートモニタのサンプルプログラム

コード例 F–1 に示すのは、SAC からのメッセージに応答する以外は何もしないポートモニタのサンプルプログラムです。

コード例 F–1 ポートモニタのサンプル

```
# include <stdlib.h>
# include <stdio.h>
# include <unistd.h>
# include <fcntl.h>
# include <signal.h>
# include <sac.h>

char Scratch[BUFSIZ]; /* スクラッチバッファ */
char Tag[PMTAGSIZE + 1]; /* ポートモニタのタグ */
FILE *Fp; /* ログファイルへのファイルポインタ */
FILE *Tfp; /* pid ファイルへのファイルポインタ */
char State; /* ポートモニタの現在の状態 */

main(argc, argv)
 int argc;
 char *argv[];
{
 char *istate;
 strcpy(Tag, getenv("PMTAG"));
/*
 * ポートモニタのプライベートディレクトリにログファイルをオープンします。
 */
 sprintf(Scratch, "/var/saf/%s/log", Tag);
 Fp = fopen(Scratch, "a+");
 if (Fp == (FILE *)NULL)
  exit(1);
 log(Fp, "starting");
/*
 * 初期状態 (enabled または disabled) を取り出して、
```

(続く)

```
 * State をそれに合わせて設定します。
 */
 istate = getenv("ISTATE");
 sprintf(Scratch, "ISTATE is %s", istate);
 log(Fp, Scratch);
 if (!strcmp(istate, "enabled"))
  State = PM_ENABLED;
 else if (!strcmp(istate, "disabled"))
  State = PM_DISABLED;
 else {
  log(Fp, "invalid initial state");
  exit(1);
 }
 sprintf(Scratch, "PMTAG is %s", Tag);
 log(Fp, Scratch);
/*
 * pid ファイルを設定してロックし、ポートモニタがアクティブなことを示します。
 */
 Tfp = fopen("_pid", "w");
 if (Tfp == (FILE *)NULL) {
  log(Fp, "couldn't open pid file");
  exit(1);
 }
 if (lockf(fileno(Tfp), F_TEST, 0) < 0) {
  log(Fp, "pid file already locked");
  exit(1);
 }
 fprintf(Tfp, "%d", getpid());
 fflush(Tfp);
 log(Fp, "locking file");
 if (lockf(fileno(Tfp), F_LOCK, 0) < 0) {
  log(Fp, "lock failed");
  exit(1);
 }
/*
 * sac からのポーリングメッセージを処理します。...この関数はリターンしません。
 */
 handlepoll();
 pause();
 fclose(Tfp);
 fclose(Fp);
}

handlepoll()
{
 int pfd; /* 入力パイプのファイル記述子 */
 int sfd; /* 出力パイプのファイル記述子 */
 struct sacmsg sacmsg; /* 入力メッセージ */
 struct pmmsg pmmsg; /* 出力メッセージ */
/*
 * sac からのメッセージを受信するためのパイプをオープンします。
 */
 pfd = open("_pmpipe", O_RDONLY|O_NONBLOCK);
```

(続く)

```
 if (pfd < 0) {
  log(Fp, "_pmpipe open failed");
  exit(1);
 }
/*
 * sac へメッセージを送信するためのパイプをオープンします。
 */
 sfd = open("../_sacpipe", O_WRONLY);
 if (sfd < 0) {
  log(Fp, "_sacpipe open failed");
  exit(1);
 }
/*
 * 応答メッセージの構築を開始します。ここでは、クラス 1 のメッセージだけを
 * サポートします。
 */
 strcpy(pmmsg.pm_tag, Tag);
 pmmsg.pm_size = 0;
 pmmsg.pm_maxclass = 1;
/*
 * sac からのメッセージへの応答を続けます。
 */
  for (;;) {
   if (read(pfd, &sacmsg, sizeof(sacmsg)) != sizeof(sacmsg)) {
    log(Fp, "_pmpipe read failed");
    exit(1);
   }
/*
 * メッセージのタイプを判定して、正しい応答を返します。
 */
   switch (sacmsg.sc_type) {
    case SC_STATUS:
     log(Fp, "Got SC_STATUS message");
     pmmsg.pm_type = PM_STATUS;
     pmmsg.pm_state = State;
     break;
    case SC_ENABLE:
     /* 内部状態が変わっていることに注意 */
     log(Fp, "Got SC_ENABLE message");
     pmmsg.pm_type = PM_STATUS;
     State = PM_ENABLED;
     pmmsg.pm_state = State;
     break;
    case SC_DISABLE:
     /*内部状態が変わっていることに注意 */
     log(Fp, "Got SC_DISABLE message");
     pmmsg.pm_type = PM_STATUS;
     State = PM_DISABLED;
     pmmsg.pm_state = State;
     break;
    case SC_READDB:
    /*
     * 完全なポートモニタの場合は、ここで _pmtab を読み込んで
```

(続く)

```
     * 必要なアクションを起こします。
     */
     log(Fp, "Got SC_READDB message");
     pmmsg.pm_type = PM_STATUS;
     pmmsg.pm_state = State;
     break;
    default:
     sprintf(Scratch, "Got unknown message <%d>",
     sacmsg.sc_type);
     log(Fp, Scratch);
     pmmsg.pm_type = PM_UNKNOWN;
     pmmsg.pm_state = State;
     break;
   }
/*
 * ポーリングに対する応答を返し、
 * 現在の状態を知らせます。
 */
   if (write(sfd, &pmmsg, sizeof(pmmsg)) != sizeof(pmmsg))
    log(Fp, "sanity response failed");
  }
}
/*
 * 一般的なログ機能
 */
log(fp, msg)
 FILE *fp;
 char *msg;
{
 fprintf(fp, "%d; %s\n", getpid(), msg);
 fflush(fp);
}
```

コード例 F–2 はヘッダーファイル sac.h を示しています。

コード例 **F–2**　ヘッダーファイル sac.h

```
/* utmp id のバイト数 */
# define IDLEN 4
/* utmp ids のワイルド文字 */
# define SC_WILDC 0xff
/* ポートモニタタグの最大長 (バイト数) */
# define PMTAGSIZE 14
/*
 * doconfig() のための rflag 値
 */
```

(続く)

```
/* assign コマンドは許可しない */
# define NOASSIGN 0x1
/* run と runwait のコマンドは許可しない */
# define NORUN 0x2
/*
 * sac へのメッセージ (ヘッダーのみ)。このヘッダー形式はずっと固定されます。
 * サイズフィールド (pm_size) には、ヘッダーに続くオプションデータ部のサイズを
 * 指定します。オプションデータ部の形式は、メッセージタイプ (pm_type) によって
 * 厳密に定義されます。
 */
struct pmmsg {
 char pm_type;                /* メッセージタイプ */
 unchar pm_state;             /* ポートモニタの現在の状態 */
 char pm_maxclass;            /* ポートモニタが理解できる最大の
                        メッセージクラス */
 char pm_tag[PMTAGSIZE + 1]; /* ポートモニタのタグ */
 int pm_size;                 /* オプションデータ部のサイズ */
};
/*
 * pm_type の値
 */
# define PM_STATUS 1 /* status response */
# define PM_UNKNOWN 2 /* unknown message was received */
/*
 * pm_state の値
 */
/*
 * クラス 1 の応答
 */
# define PM_STARTING 1   /* ポートモニタは起動状態 */
# define PM_ENABLED 2    /* ポートモニタは有効状態 */
# define PM_DISABLED 3   /* ポートモニタは無効状態 */
# define PM_STOPPING 4   /* ポートモニタは停止状態 */
/*
 * ポートモニタへのメッセージ
 */
struct sacmsg {
 int sc_size;          /* オプションデータ部のサイズ */
 char sc_type;         /* メッセージタイプ */
};
/*
 * sc_type の値
 * 次に定義するコマンドは SAC がポートモニタに送信するコマンドです。
 * 拡張性を持たせるため、コマンドは各クラスに分類されます。
 * 後に定義されるクラスは、それまでに定義されたクラスのコマンドに、
 * そのクラスの新たなコマンドが追加されたスーパーセットです。
 * どのクラスもヘッダーは同じです。新たなコマンドは、
 * ヘッダーにオプションデータ部が追加される形で定義されます。
 * オプションデータ部の形式は、コマンドで自動的に決まります。
 * 重要な注: SAC から最初に送信されるメッセージは常にクラス 1 の
 * メッセージです。これに対して、ポートモニタは応答メッセージで
 * 自分が理解できる最大のクラスを知らせます。
 * もう 1 つ注意しなければならないのは、ポートモニタは必ず
```

(続く)

```
 * 受信したメッセージと同じクラスで応答しなければならないことです。
 * (すなわち、クラス 1 のコマンドには必ずクラス 1 で応答します。)
 */
/*
 * クラス 1 のコマンド (現在は、クラス 1 のコマンドしかありません)
 */
# define SC_STATUS 1     /* ステータス要求 */
# define SC_ENABLE 2     /* 有効に設定 */
# define SC_DISABLE 3    /* 無効に設定 */
# define SC_READDB 4     /* pmtab の読み込みを要求 */
/*
 * Saferrno のエラー番号。Saferrno は pmadm と sacadm の両方で共有する
 * ことに注意してください。
 */
# define E_BADARGS 1    /* 引数またはコマンド行が不正 */
# define E_NOPRIV 2     /* ユーザーは操作特権を持っていない */
# define E_SAFERR 3     /* 一般的な SAF エラー */
# define E_SYSERR 4     /* システムエラー */
# define E_NOEXIST 5    /* 指定が無効 */
# define E_DUP 6        /* エントリが既に存在している */
# define E_PMRUN 7      /* ポートモニタが実行中 */
# define E_PMNOTRUN 8   /* ポートモニタが実行されていない */
# define E_RECOVER 9    /* 修復中 */
```

論理ダイアグラムとディレクトリ構造

図 F–2 は、SAF の論理ダイアグラムを示します。ひとつの SAF が、システムごとに複数のポートモニタをどのように生成するのかを示しています。これは、いくつかのモニタが並行して作動し、各種の異なるプロトコルを同時に起動させていることを意味します。

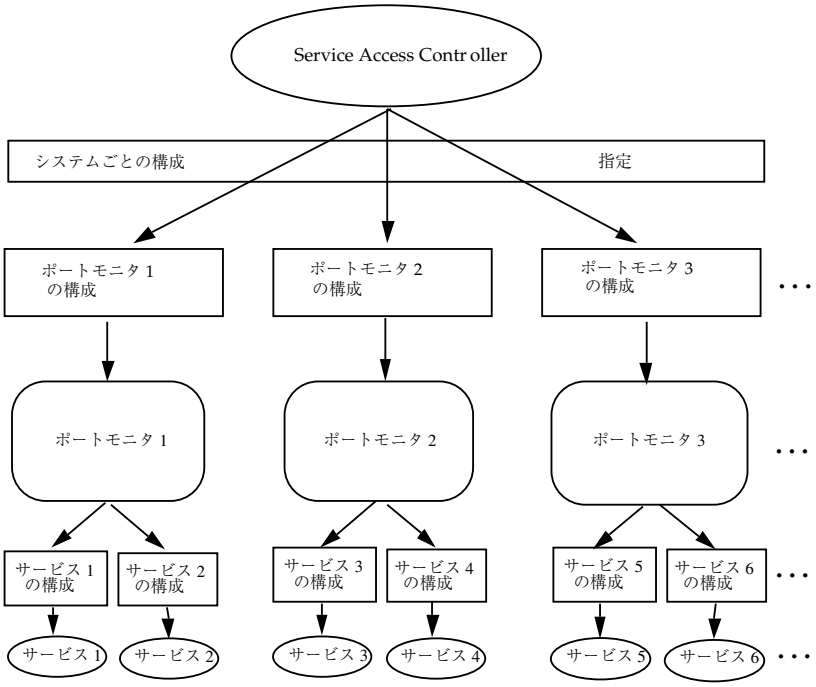


図 F–1　SAF の論理フレームワーク

`/etc/saf/_sysconfig` は、論理ダイアグラムに対応するディレクトリ構造を示します。358ページの「`/etc/saf/_sysconfig`」 の後に、ディレクトリ構造内のファイルおよびディレクトリの説明があります。

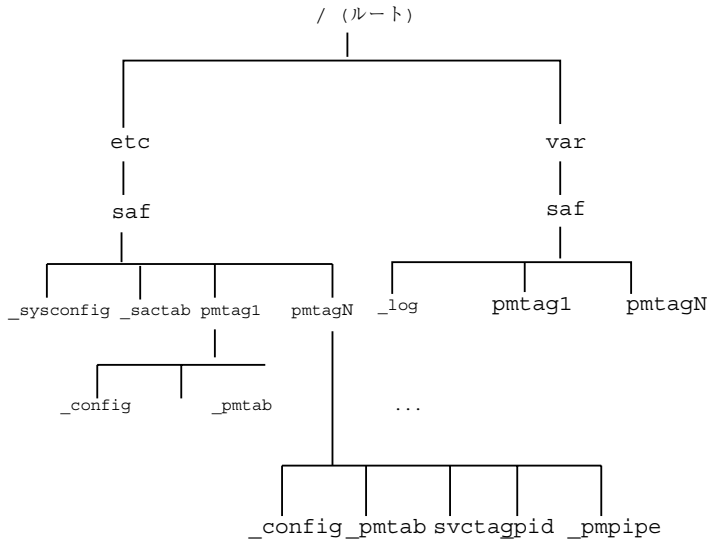


図 F–2　SAF のディレクトリ構造

/etc/saf/_sysconfig

システムごとの構成スクリプト。

/etc/saf/_sactab

SAC の管理ファイル。ここには、SAC の管理下にあるポートモニタの情報が入っています。

/etc/saf/pmtag

ポートモニタ pmtag のホームディレクトリ。

/etc/saf/pmtag/_config

ポートモニタ pmtag の、ポートモニタごとの構成スクリプト。

/etc/saf/pmtag/_pmtab

ポートモニタ pmtag の管理ファイル。ここには、pmtag の管理下にあるサービス情報が入っています。

/etc/saf/pmtag/svctag

ポートモニタ pmtag の管理下にあるサービス svctag の、サービスごとの構成スクリプトが入っているファイル。

/etc/saf/pmtag/_pid

ポートモニタがカレントディレクトリで プロセス ID を書き込んだり、アドバイザリロックを設定したりするファイル。

/etc/saf/pmtag/_pmpipe

ポートモニタが、sac および ../_sacpipe からのメッセージを受信したり、sac に応答メッセージを返したりするファイル。

/var/saf/_log

SAC のログファイル。

/var/saf/pmtag

ポートモニタ pmtag が作成するファイル (たとえば、ログファイル) を入れるディレクトリ。

用語集

RPC プログラミング用語

次に挙げる用語は、本書を通じて使用される RPC の概念について定義します。

クライアント	遠隔プログラムまたは遠隔システムのサービスを使用するプログラムまたはシステム
クライアントハンドル	特定サーバーの RPC プログラムへのクライアントの結合を表すクライアントプロセスのデータ構造体
接続型トランスポート	「ストリームトランスポート」を参照
非接続トランスポート	「データグラムトランスポート」を参照
データグラムトランスポート	バッファサイズによって制限されるデータの伝送形式の一種。接続型トランスポートよりオーバヘッドは少ないが信頼性は低いとされる
デシリアライズ	データを XDR 書式からマシン特定の表現に変換すること
ハンドル	サービスライブラリによって使用される抽象概念。ファイル、またはソケットなどのファイルに似たオブジェクトを指す
ホスト	ネットワークに接続されたコンピュータ (メインフレーム、小型、サーバー、ワークステーション、またはパーソナルコンピュータ)
マルチスレッドホット	ライブラリまたは呼び出しによって自動的にスレッドが作成される場合、インタフェースはマルチスレッドホットである、という

マルチスレッド対応	インタフェースをマルチスレッド環境で呼び出すことができる場合、インタフェースはマルチスレッド対応である、という。マルチスレッド対応のインタフェースは複数のスレッドに対して同時に起動される
ネットワーククライアント	通常クライアント。サービスに対する遠隔手続き呼び出しを行うプロセス
ネットワークサーバー	通常サーバー。ネットワークサービスを行うプロセス。遠隔サービスプログラムの複数のバージョンをサポートして変更プロトコルの使用によって互換性を持たせるようにする
ネットワークサービス	1 つ以上の遠隔サービスプログラムの集合
ping	`ping` サービスは遠隔システムのアクティビティを検査するために使用される
遠隔プログラム	1 つ以上の遠隔手続きを実現するプログラム
RPC 言語 **(RPCL)**	`rpcgen` コンパイラによって変換される C 類似プログラミング言語。RPCL は XDR 言語のスーパーセット
RPC ライブラリ	コンパイル時にリンカエディタに指定される `libnsl`。RPC パッケージともいう
RPC プロトコル	RPC パッケージの基底となるメッセージ引き渡しプロトコル
RPC/XDR	「RPC 言語 (RPCL)」 を参照
シリアライズ	マシン言語から XDR 書式へのデータ変換
サーバー	クライアントに遠隔サービスを提供するプロセス
ストリームトランスポート	無制限のデータサイズのバイトストリーム配信をサポートする、信頼性があるとされるデータの伝送形式の一種
トランスポート層	開放型相互接続 (OSI) 参照モデルの 4 番目の層
トランスポートハンドル	トランスポートのデータ構造体を指す RPC ライブラリによって使用される抽象概念

TI-RPC	トランスポート独立な RPC。SunOS 5.*x* でサポートされる RPC バージョン
TS-RPC	トランスポート特定の RPC。SunOS 4.*x* でサポートされる RPC バージョン。TS-RPC も SunOS 5.*x* でサポートされる
一般アドレス	トランスポートアドレスのマシン独立な表現
仮想回路トランスポート	「ストリームトランスポート」を参照
XDR 言語	データ記述言語およびデータ表現プロトコル

索引

数字

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

X

あ

さ

し

な

に

ね

は

ひ

ま

め