ONC+ 開発ガイド

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

Part No: 816–3978–10
2002 年 5 月

Copyright 2002 Sun Microsystems, Inc.　4150 Network Circle, Santa Clara, CA 95054 U.S.A.　All rights reserved.

本製品およびそれに関連する文書は著作権法により保護されており、その使用、複製、頒布および逆コンパイルを制限するライセンスのもとにおいて頒布されます。サン・マイクロシステムズ株式会社の書面による事前の許可なく、本製品および関連する文書のいかなる部分も、いかなる方法によっても複製することが禁じられます。

本製品の一部は、カリフォルニア大学からライセンスされている Berkeley BSD システムに基づいていることがあります。UNIX は、X/Open Company, Ltd. が独占的にライセンスしている米国ならびに他の国における登録商標です。フォント技術を含む第三者のソフトウェアは、著作権により保護されており、提供者からライセンスを受けているものです。

Federal Acquisitions: Commercial Software–Government Users Subject to Standard License Terms and Conditions.

本製品に含まれる HG 明朝 L、HG-MincyoL-Sun、HG ゴシック B、および HG-GothicB-Sun は、株式会社リコーがリョービイマジクス株式会社からライセンス供与されたタイプフェースマスタをもとに作成されたものです。HG 平成明朝体 W3@X12 は、株式会社リコーが財団法人日本規格協会からライセンス供与されたタイプフェースマスタをもとに作成されたものです。フォントとして無断複製することは禁止されています。

Sun、Sun Microsystems、docs.sun.com、AnswerBook、AnswerBook2 --> は、米国およびその他の国における米国 Sun Microsystems, Inc. (以下、米国 Sun Microsystems 社とします) の商標もしくは登録商標です。

サンのロゴマークおよび Solaris は、米国 Sun Microsystems 社の登録商標です。

すべての SPARC 商標は、米国 SPARC International, Inc. のライセンスを受けて使用している同社の米国およびその他の国における商標または登録商標です。SPARC 商標が付いた製品は、米国 Sun Microsystems 社が開発したアーキテクチャに基づくものです。

OPENLOOK、OpenBoot、JLE は、サン・マイクロシステムズ株式会社の登録商標です。

Wnn は、京都大学、株式会社アステック、オムロン株式会社で共同開発されたソフトウェアです。

Wnn6 は、オムロン株式会社、オムロンソフトウェア株式会社で共同開発されたソフトウェアです。© Copyright OMRON Co., Ltd. 1995-2000. All Rights Reserved. © Copyright OMRON SOFTWARE Co.,Ltd. 1995-2002 All Rights Reserved.

「ATOK」は、株式会社ジャストシステムの登録商標です。

「ATOK Server/ATOK12」は、株式会社ジャストシステムの著作物であり、「ATOK Server/ATOK12」にかかる著作権その他の権利は、株式会社ジャストシステムおよび各権利者に帰属します。

本製品に含まれる郵便番号辞書 (7 桁/5 桁) は郵政事業庁が公開したデータを元に制作された物です (一部データの加工を行なっています)。

本製品に含まれるフェイスマーク辞書は、株式会社ビレッジセンターの許諾のもと、同社が発行する『インターネット・パソコン通信フェイスマークガイド '98』に添付のものを使用しています。© 1997 ビレッジセンター

Unicode は、Unicode, Inc. の商標です。

本書で参照されている製品やサービスに関しては、該当する会社または組織に直接お問い合わせください。

OPEN LOOK および Sun Graphical User Interface は、米国 Sun Microsystems 社が自社のユーザおよびライセンス実施権者向けに開発しました。米国 Sun Microsystems 社は、コンピュータ産業用のビジュアルまたはグラフィカル・ユーザインタフェースの概念の研究開発における米国 Xerox 社の先駆者としての成果を認めるものです。米国 Sun Microsystems 社は米国 Xerox 社から Xerox Graphical User Interface の非独占的ライセンスを取得しており、このライセンスは米国 Sun Microsystems 社のライセンス実施権者にも適用されます。

DtComboBox ウィジェットと DtSpinBox ウィジェットのプログラムおよびドキュメントは、Interleaf, Inc. から提供されたものです。(© 1993 Interleaf, Inc.)

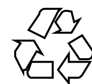
本書は、「現状のまま」をベースとして提供され、商品性、特定目的への適合性または第三者の権利の非侵害の黙示の保証を含みそれに限定されない、明示的であるか黙示的であるかを問わない、なんらの保証も行われないものとします。

本製品が、外国為替および外国貿易管理法 (外為法) に定められる戦略物資等 (貨物または役務) に該当する場合、本製品を輸出または日本国外へ持ち出す際には、サン・マイクロシステムズ株式会社の事前の書面による承諾を得ることのほか、外為法および関連法規に基づく輸出手続き、また場合によっては、米国商務省または米国所轄官庁の許可を得ることが必要です。

原典: *ONC+ Developer's Guide*

Part No: 816-1435-10

Revision A



020328@3689

目次

はじめに **21**

1 ONC+ 入門 **27**

ONC+ の概要 28

TI-RPC 28

XDR 28

NFS 28

NIS+ 29

2 TI-RPC 入門 **31**

TI-RPC の概要 31

TI-RPC の問題 32

パラメータの引き渡し 33

結合 33

トランスポートプロトコル 33

呼び出しセマンティクス 33

データ表現 33

プログラム、バージョン、手続き番号 34

インタフェースルーチンの概要 34

単純インタフェースのルーチン 34

標準インタフェースのルーチン 35

ネットワーク選択 37

トランスポート選択 38

名前からアドレスへの変換 39

アドレスルックアップサービス 40

アドレス登録 40
RPC 情報の取り出し 41

3 rpcgen プログラミングガイド **43**
rpcgen の概要 43
SunOS 5.9 ソフトウェア環境の機能 44
rpcgen チュートリアル 45
ローカル手続きをリモートプロシージャに変換 45
複雑なデータ構造の引き渡し 51
前処理命令 56
cpp 命令 57
コンパイル時に指定するフラグ 57
クライアント側およびサーバ側のテンプレートを生成するコンパイルオプション 58
C 形式モードでのコンパイル 59
マルチスレッド対応コードのコンパイル 62
自動マルチスレッド対応モードでのコンパイル 68
TI-RPC または TS-RPC のライブラリ選択 68
ANSI C に準拠したコードの生成 69
xdr_inline() カウント 69
rpcgen プログラミングテクニック 70
ネットワークタイプ / トランスポート選択 70
コマンド行の定義文 71
ブロードキャスト呼び出しへのサーバーからの応答 71
ポートモニターのサポート 72
タイムアウト値の変更 73
クライアントの認証 73
ディスパッチテーブル 74
rpcgen の 64 ビットの場合の考慮事項 75
rpcgen の IPv6 の場合の考慮事項 77
アプリケーションのデバッグ 77

4 **RPC** プログラマインタフェース **79**
単純インタフェース 79
クライアント側 80
サーバー側 82
ユーザーが作成する登録ルーチン 83

任意のデータ型の引き渡し 83
標準インタフェース 87
トップレベルのインタフェース 88
中間レベルのインタフェース 92
エキスパートレベルのインタフェース 94
ボトムレベルのインタフェース 99
サーバーのキャッシュ 100
下位レベルのデータ構造 100
下位レベルの Raw RPC を使用したプログラムテスト 103
接続型トランスポート 105
XDR によるメモリー割り当て 108

5 RPC プログラミングの高度なテクニック 111
サーバー側の poll( ) 111
ブロードキャスト RPC 113
バッチ処理 115
認証 118
AUTH_SYS タイプの認証 120
AUTH_DES タイプの認証 122
AUTH_KERB 認証形式 124
RPCSEC_GSS を使用した認証 125
RPCSEC_GSS API 126
RPCSEC_GSS ルーチン 127
コンテキストの作成 129
値の変更とコンテキストの破棄 130
主体名 131
サーバーで資格を受信する 133
コールバック 135
最大データサイズ 135
その他の関数 136
関連ファイル 136
ポートモニターの使用 138
inetd の使用 138
リスナーの使用 139
サーバーのバージョン 140
クライアントのバージョン 142
一時的な RPC プログラム番号の使用 143

6 TS-RPC から TI-RPC への移行について 145
アプリケーションの移行 145
移行の必要性 146
RPC の場合の IPv6 の考慮事項 146
特殊事項 147
TI-RPC と TS-RPC の相違点 147
関数の互換性のリスト 148
サービスの作成と廃棄 149
サービスの登録と登録削除 149
SunOS 互換性呼び出し 149
ブロードキャスト 149
アドレス管理 150
認証 150
その他の関数 150
旧バージョンとの比較 151

7 マルチスレッド **RPC** プログラミング **155**
マルチスレッドクライアントの概要 155
マルチスレッドサーバーの概要 157
サービストランスポートハンドルの共有 158
自動マルチスレッド対応モード 159
マルチスレッドユーザーモード 162
ユーザーモードでのライブラリリソースの解放 163

8 Sun RPC ライブラリの拡張 **167**
新しい機能 167
1 方向性メッセージング 168
`clnt_send()` 170
`oneway` 属性 170
非ブロッキング入出力 173
非ブロッキング入出力の使用 174
非ブロッキングとして構成したときの `clnt_call()` 177
クライアント接続クロージャーコールバック 178
クライアント接続クロージャーコールバックを使用した例 178
ユーザーファイル記述子コールバック 184
ファイル記述子の例 185

9 NIS+ プログラミングガイド 197
NIS+ の概要 197
NIS+ のドメイン 197
NIS+ とサーバー 198
NIS+ テーブル 198
NIS+ のセキュリティ 199
ネームサービススイッチ 199
NIS+ の管理コマンド 200
NIS+ の API 201
NIS+ サンプルプログラム 205
サポートされないマクロの使用 206
サンプルプログラムで使用する関数 206
プログラムのコンパイル 206

A XDR テクニカルノート 217
XDR の概要 217
データの標準形式 220
XDR ライブラリ 222
XDR ライブラリのプリミティブ 224
XDR ルーチンで必要なメモリー 224
整数フィルタ 226
浮動小数点フィルタ 227
列挙型フィルタ 227
データなしルーチン 228
合成データ型フィルタ 228
文字列 228
バイト配列 229
配列 230
隠されたデータ 233
固定長配列 233
識別型の共用体 234
ポインタ 236
フィルタ以外のプリミティブ 237
処理内容 238
ストリームへのアクセス 238
標準入出力ストリーム 238
メモリーストリーム 238

レコード (TCP/IP) ストリーム 239
XDR ストリームの作成 240
XDR オブジェクト 240
高度な XDR の機能 242
リンクリスト 242

B RPC プロトコルおよび言語の仕様 **247**
プロトコルの概要 247
RPC モデル 248
トランスポートとセマンティクス 248
結合と相互認識の独立性 249
プログラムと手続き番号 250
プログラム番号の割り当て 251
プログラム番号の登録 252
RPC プロトコルのその他の使用方法 252
RPC メッセージプロトコル 253
レコードマーク標準 256
認証プロトコル 256
AUTH_NONE 256
AUTH_SYS 257
AUTH_DES 認証 258
AUTH_DES 認証のベリファイア 259
ニックネームとクロック同期 260
DES 認証プロトコル (XDR 言語で記述) 260
AUTH_KERB 認証 263
RPC 言語の仕様 267
RPC 言語で記述されたサービスの例 267
RPCL 構文 268
RPCL 列挙型 269
RPCL 定数 270
RPCL 型の定義 270
RPCL 宣言 270
RPCL 単純宣言 270
RPCL 固定長配列宣言 271
RPCL 可変長配列宣言 271
RPCL ポインタ宣言 272
RPCL 構造体 272

RPCL 共用体 273
RPCL プログラム 273
RPCL 言語規則の例外 274
`rpcbind` プロトコル 276
`rpcbind` の操作 281

C XDR プロトコル仕様 **285**
XDR プロトコルの概要 285
グラフィックボックス表現 286
基本ブロックサイズ 286
XDR のデータ型宣言 287
符号付き整数 287
符号なし整数 288
列挙型 288
ブール型 289
hyper 整数と符号なし hyper 整数 289
浮動小数点 290
4 倍精度浮動小数点 291
固定長の隠されたデータ 292
可変長の隠されたデータ 293
カウント付きバイト文字列 294
固定長配列 295
可変長配列 295
構造体 296
識別型の共用体 297
void 298
定数 298
Typedef 298
オプションデータ 299
XDR 言語仕様 300
表記方法 300
字句解析 300
構文 302
XDR データ記述 303
RPC 言語リファレンス 304

D **RPC コーディング例** **305**
ディレクトリリストプログラムとその補助ルーチン (rpcgen) 305
時刻サーバープログラム (rpcgen) 309
2 つの数値の合計を求めるプログラム (rpcgen) 309
スプレイパケットプログラム (rpcgen) 310
メッセージ表示プログラムとそのリモートバージョン 311
バッチコードの例 314
バッチを使用しない例 316

E **portmap ユーティリティ** **319**
システム登録の概要 319
portmap プロトコル 320
portmap の操作 322
PMAPPROC_NULL 322
PMAPPROC_SET 322
PMAPPROC_UNSET 323
PMAPPROC_GETPORT 323
PMAPPROC_DUMP 323
PMAPPROC_CALLIT 323

F **SAF を使用したポートモニタープログラムの作成** **325**
SAF の概要 325
SAC の概要 326
ポートモニターの基本機能 326
ポート管理 327
アクティビティの監視 327
ポートモニターのその他の機能 327
ポートモニターの終了 328
SAF ファイル 329
ポートモニターの管理ファイル 329
サービスごとの構成ファイル 329
ポートモニターのプライベートファイル 329
SAC とポートモニターのインタフェース 330
メッセージ形式 330
メッセージクラス 332
ポートモニターの管理インタフェース 332
SAC の管理ファイル _sactab 332

ポートモニターの管理ファイル _pmtab 333
SAC 管理コマンド sacadm 335
ポートモニターの管理コマンド pmadm 336
モニタ固有の管理コマンド 336
ポートモニターとサービスのインタフェース 336
ポートモニターに必要な条件 337
重要なファイル 337
ポートモニターの実行すべきタスク 338
構成ファイルとスクリプト 339
構成スクリプトのインタプリタ: doconfig() 339
システムごとの構成ファイル 340
ポートモニターごとの構成ファイル 340
サービスごとの構成ファイル 340
構成スクリプト言語 340
構成スクリプトの印刷、インストール、置き換え 342
ポートモニターのサンプルプログラム 344
論理ダイアグラムとディレクトリ構造 349

用語集 **353**

索引 **355**

表目次

表 2–1 RPC ルーチン - 単純レベル 35
表 2–2 RPC ルーチン—トップレベル 35
表 2–3 RPC ルーチン—中間レベル 36
表 2–4 RPC ルーチン—エキスパートレベル 36
表 2–5 RPC ルーチン—ボトムレベル 37
表 2–6 nettype パラメータ 38
表 2–7 名前からアドレスへの変換ルーチン 39
表 3–1 rpcgen の前処理理命令 56
表 3–2 rpcgen コンパイル時に指定するフラグ 57
表 3–3 rpcgen テンプレート選択フラグ 58
表 3–4 RPC プログラミングテクニック 70
表 4–1 プリミティブタイプの等価関数 84
表 5–1 RPC が現在サポートしている認証タイプ 118
表 5–2 RPCSEC_GSS 関数 127
表 5–3 RPC inetdサービス 139
表 6–1 TI-RPC と TS-RPC の相違点 147
表 7–1 rpc_control ライブラリルーチン 159
表 9–1 NIS+ 名前空間管理コマンド 200
表 9–2 NIS+ の API 関数 201
表 9–3 NIS+ テーブルオブジェクト 208
表 B–1 RPC プログラム番号 251
表 B–2 RPC 言語の定義 269
表 C–1 XDR データ記述の例 304
表 F–1 サービスアクセスコントローラ _sactab ファイル 333
表 F–2 SVCTAG サービスエントリ 334
表 F–3 ポートモニターの主要ファイル 338

図目次

図 1–1 ONC+ 分散コンピューティングプラットフォーム 27

図 2–1 リモートプロシージャ呼び出しの動作 31

図 5–1 GSS-API と RPCSEC-GSS のセキュリティ階層 126

図 6–1 RCP アプリケーション 146

図 7–1 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) 156

図 7–2 マルチスレッド RPC サーバーのタイミングダイアグラム 158

図 8–1 1 方向性メッセージング 168

図 8–2 2 方向性メッセージング 169

図 8–3 非ブロッキングメッセージング 174

図 9–1 NIS+ のドメイン 197

図 9–2 NIS+ のテーブル 198

図 B–1 認証プロセスマップ 257

図 E–1 典型的な Portmap シーケンス (TCP/IP のみ) 320

図 F–1 SAF 論理のフレームワーク 349

図 F–2 SAF ディレクトリ構造 350

例目次

例 3–1 シングルコンピュータ用の printmsg.c 45
例 3–2 RPC バージョンの printmsg.c 47
例 3–3 printmsg.c を呼び出すクライアント側プログラム 48
例 3–4 RPC 言語で書かれたプロトコル記述ファイル (dir.x) 51
例 3–5 サーバー側の dir_proc.c の例 53
例 3–6 クライアント側のプログラム (rls.c) 54
例 3–7 時刻プロトコルを記述する rpcgen ソースプログラム 56
例 3–8 C 形式モードの add.x 59
例 3–9 デフォルトモードの add.x 59
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 60
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 61
例 3–12 C 形式モードのサーバー側プログラムの例 61
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 61
例 3–14 マルチスレッド対応プログラム : msg.x 62
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 62
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場合) 63
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 64
例 3–18 マルチスレッド対応のプログラム : add.x 65
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 65
例 3–20 マルチスレッド対応サーバー側プログラム : add.x 67
例 3–21 自動マルチスレッド対応モード : time.x 68
例 3–22 ANSI C に準拠した rpcgen サーバー側テンプレート 69
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 71
例 3–24 clnt_control ルーチン 73
例 3–25 AUTH_SYS クライアントの認証 73

例 3–26　スーパーユーザーだけが使用できる printmsg_1　74
例 3–27　ディスパッチテーブルの使用方法　75
例 4–1　rusers プログラム　80
例 4–2　単純インタフェースを使用する rusers プログラム　81
例 4–3　xdr_simple ルーチン　85
例 4–4　変換ルーチン xdr_varintarr　86
例 4–5　変換ルーチン xdr_vectorxdr_vector　86
例 4–6　変換ルーチン xdr_reference　86
例 4–7　ヘッダーファイル time_prot.h　88
例 4–8　時刻を返すサービス : クライアント側　88
例 4–9　時刻を返すサービス : サーバー側　90
例 4–10　時刻サービスのクライアント側プログラム　92
例 4–11　時刻サービスのサーバー側プログラム　93
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　95
例 4–13　下位レベル RPC を使用したサーバー側プログラム　97
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　99
例 4–15　ボトムレベル用のサーバー　100
例 4–16　クライアント側 RPC ハンドル (CLIENT 構造体)　100
例 4–17　クライアント側の認証ハンドル　101
例 4–18　サーバー側のトランスポートハンドル　101
例 4–19　Raw RPC を使用した簡単なプログラム　103
例 4–20　リモートコピー (両方向 XDR ルーチン)　105
例 4–21　リモートコピー : クライアント側ルーチン　106
例 4–22　リモートコピー : サーバー側ルーチン　107
例 5–1　svc_run() と poll()　112
例 5–2　RPC ブロードキャスト　113
例 5–3　ブロードキャストへの応答の収集　114
例 5–4　バッチ処理を使用しないクライアントプログラム　116
例 5–5　バッチ処理を使用するクライアントプログラム　116
例 5–6　バッチ処理を行うサーバー　117
例 5–7　AUTH_SYS タイプの資格 - 認証構造体　120
例 5–8　認証データをチェックするサーバープログラム　120
例 5–9　AUTH_DES タイプの認証を使用するサーバー　123
例 5–10　rpc_gss_seccreate()　129
例 5–11　rpc_gss_set_defaults ()　130
例 5–12　rpc_gss_set_svc_name()　131
例 5–13　rpc_gss_get_principal_name()　132
例 5–14　資格の入手　133

例 5–15　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　140
例 5–16　両バージョンを使用するサーバー　141
例 5–17　クライアント側での RPC バージョン選択　142
例 5–18　一時的な RPC プログラム - サーバー側　144
例 6–1　TS-RPC におけるクライアント作成　151
例 6–2　TI-RPC でのクライアント作成　151
例 6–3　TS-RPC におけるブロードキャスト　152
例 6–4　TI-RPC におけるブロードキャスト　153
例 7–1　マルチスレッド自動モードのサーバー　160
例 7–2　マルチスレッド自動モード : `time_prot.h` ヘッダーファイル　162
例 7–3　マルチスレッドユーザーモード : `rpc_test.h`　163
例 7–4　マルチスレッドユーザーモードでのクライアント　164
例 9–1　ディレクトリオブジェクトを作成する NIS+ ルーチン　207
例 9–2　グループオブジェクトを作成する NIS+ ルーチン　208
例 9–3　テーブルオブジェクトを作成する NIS+ ルーチン　208
例 9–4　テーブルにオブジェクトを追加する NIS+ ルーチン　209
例 9–5　`nis_list` を呼び出す NIS+ ルーチン　210
例 9–6　オブジェクトを一覧表示する NIS+ ルーチン　211
例 9–7　ディレクトリオブジェクトを削除する NIS+ ルーチン　213
例 9–8　オブジェクトをすべて削除する NIS+ ルーチン　213
例 9–9　NIS+ プログラムの実行　215
例 A–1　`writer` サンプルプログラム (初期状態)　218
例 A–2　`reader` サンプルプログラム (初期状態)　218
例 A–3　`writer` サンプルプログラム (XDR 修正バージョン)　219
例 A–4　`reader` サンプルプログラム (XDR 修正バージョン)　220
例 A–5　`xdr_sizeof` の例 1　224
例 A–6　`xdr_sizeof` の例 2　225
例 A–7　配列変換サンプルプログラム 1　231
例 A–8　配列変換サンプルプログラム 2　231
例 A–9　配列変換サンプルプログラム 3　232
例 A–10　xdr_netobj ルーチン　233
例 A–11　`xdr_vector` ルーチン　233
例 A–12　XDR 識別型の共用体サンプルプログラム　235
例 A–13　XDR ストリームインタフェースの例　240
例 A–14　リンクリスト　242
例 A–15　`xdr_pointer`　244
例 A–16　再帰呼び出しを行わない XDR 変換　244
例 B–1　RPC メッセージプロトコル　253

例 **B–2** AUTH_DES 認証プロトコル 260
例 **B–3** AUTH_DES 認証プロトコル 265
例 **B–4** RPC 言語を使用する ping サービス 267
例 **B–5** rpcbind プロトコル仕様 (RPC 言語で記述) 276
例 **C–1** XDR 仕様 301
例 **C–2** XDR ファイルデータ構造体 303
例 **D–1** rpcgen プログラム: dir.x 305
例 **D–2** リモート dir_proc.c 306
例 **D–3** rls.c クライアント 307
例 **D–4** rpcgen プログラム: time.x 309
例 **D–5** rpcgenプログラム: 2 つの数値の合計を求める 309
例 **D–6** rpcgen プログラム: spray.x 310
例 **D–7** printmesg.c 311
例 **D–8** printmesg.c のリモートバージョン 312
例 **D–9** rpcgen プログラム: msg.x 313
例 **D–10** mesg_proc.c 313
例 **D–11** バッチを使用するクライアントプログラムの例 314
例 **D–12** バッチを使用するサーバープログラムの例 315
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 316
例 **E–1** portmap プロトコル仕様 (RPC 言語で記述) 321
例 **F–1** ポートモニターのサンプル 344
例 **F–2** sac.h ヘッダーファイル 347

はじめに

このマニュアルでは、リモートプロシージャ呼び出し (RPC) と、米国 Sun Microsystems, Inc. (以下「Sun」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 9 は Sun 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 5.9 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、Sun 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。ネットワークプログラミングの経験は、あれば役立ちますが、このマニュアルの使用においては必須ではありません。

内容の紹介

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。

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。

第 5 章では、RPC の下位レベルインタフェースを使用するさまざまなプログラミング方法について説明します。

第 6 章では、TS-RPC アプリケーションを TI-RPC へ移行するための方法について説明します。

第 7 章では、マルチスレッド RPC プログラミングについて説明します。ここでは、マルチスレッドプログラミングの Solaris 実装についての基本的な事柄は説明していません。

第 8 章では、Sun RPC ライブラリに追加された新しい拡張機能について説明します。

第 9 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 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。 旧バージョンの SunOS 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 を使用する呼び出し側プログラムの作成方法を示します。

関連マニュアルおよびサイト

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 Version 3 Protocol Specification*』Sun Microsystems, 1993。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`
- *1094 NFS: Network File System Protocol Specification Version 2*
- *1813 NFS Version 3 Protocol Specification*
- *1831 RPC: Remote Procedure Call Protocol Specification Version 2*
- *1832 XDR: External Data Representation Standard*

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説明があります。

- Brent Callaghan.*NFS Illustrated*, Addision-Wesley Professional Computing Series. ISBN: 0201325705
- W. Richard Stevens. “Networking APIs: Sockets and XTI in *UNIX Network Programming Volume 1.* Englewood Cliffs, N.J. : Prentice Hall Software Series, 1990. UNIX ネットワークプログラミングについての説明とサンプルコードがあります。IPv4 と IPv6、ソケットと XTI、TCP と UDP、 raw ソケット、プログラミング技術、マルチキャスト、およびブロードキャストなどが取り上げられています。
- John Bloomer. *Power Programming with RPC* Sebastopol, Calif.: O’Reilly & Associates, Inc, 1992.

Sun のオンラインマニュアル

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

表記上の規則

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

表 P–1 表記上の規則

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

コード例は次のように表示されます。

- C シェル

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

- C シェルのスーパーユーザー

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

- Bourne シェルおよび Korn シェル

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

- Bourne シェルおよび Korn シェルのスーパーユーザー

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

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

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

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

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

一般規則

- このマニュアルでは、「IA」という用語は、Intel 32 ビットのプロセッサアーキテクチャを意味します。これには、Pentium、Pentium Pro、Pentium II、Pentium II Xeon、Celeron、Pentium III、Pentium III Xeon の各プロセッサ、および AMD、Cyrix が提供する互換マイクロプロセッサチップが含まれます。

第 1 章

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、それらが下位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リモート手続き呼び出し (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リモート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、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 V5 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 の仕様については、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
```

NIS+

NIS+ は Solaris 環境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。このためにシステムの稼働時間や性能に影響を与えることはありません。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。

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。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。

この章では、次のトピックについて説明します。

- 31 ページの「TI-RPC の概要」
- 32 ページの「TI-RPC の問題」
- 34 ページの「インタフェースルーチンの概要」
- 37 ページの「ネットワーク選択」
- 38 ページの「トランスポート選択」
- 40 ページの「アドレスルックアップサービス」

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。

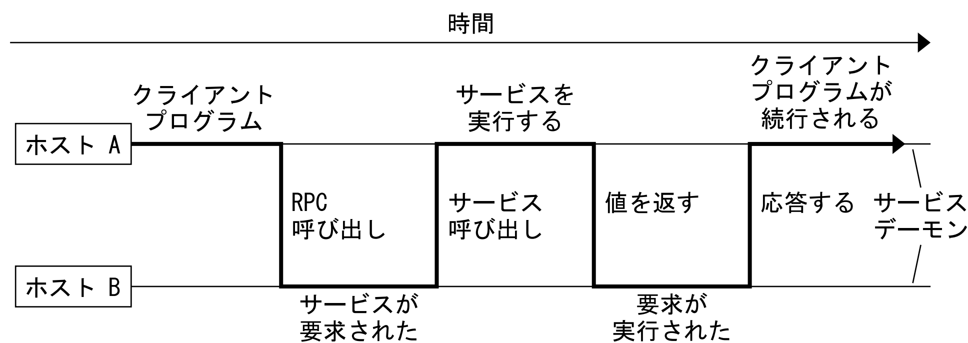


図 **2–1** リモートプロシージャ呼び出しの動作

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リモートプロシージャに渡され、呼び出し側はリモートプロシージャ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点に注意が必要です。

- 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、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 ファイルと ipnodes 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、リモートプロシージャの実行に関し、クライアントが仮定する内容を定義します。特に、その手続きが何回実行されたかを定義するのに使われます。これは、エラー条件を扱う場合に特に重要です。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。リモート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。XDR を使用することによって、TI-RPC では、各ホストのバイト順序や構造体の配置方法に影響されることなく、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 A および付録 C を参照してください。

プログラム、バージョン、手続き番号

リモートプロシージャは次の 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、関連するリモートプロシージャ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。各バージョンはリモート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。

各バージョンにはリモート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。

250 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リモートプロシージャ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。この説明とコード例は、79 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	機能
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 ()	指定したリモートホスト上の指定した手続きをリモート呼び出しする
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 つのレベルに分けられます。プログラマ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、88 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

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、92 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、94 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　(続き)

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、99 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、指定したリモート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、指定したリモート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できます。それは、可能であればアプリケーションによって使用されます。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使用しようとします。

/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、getnetconfig(3NSL) または 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	意味
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
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(3NSL) のマニュアルページ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、portmap ではなく rpcbind デーモンが使用されます。

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（3NSL）と rpcbind(3NSL) 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リモート呼び出しを実行する

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 自体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(3NSL) 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。

クライアントがリモート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、最後に呼び出したリモートプログラムのネットワークアドレスを保存されています。

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リモートプロシージャ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、267 ページの「RPC 言語の仕様」 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。rpcinfo では、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。また、rpcinfo は、登録を削除することもできます。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。

この章では、次のトピックについて説明します。

- 44 ページの「SunOS 5.9 ソフトウェア環境の機能」
- 45 ページの「rpcgen チュートリアル」
- 57 ページの「コンパイル時に指定するフラグ」
- 70 ページの「rpcgen プログラミングテクニック」

rpcgen の概要

rpcgen ツールは、リモートプログラムインタフェースモジュールを生成します。このツールは、RPC 言語で書かれたソースコードをコンパイルします。RPC 言語は構文も構造も C 言語に似ています。rpcgen ツールは C 言語ソースモジュールを生成するので、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 5.9 ソフトウェア環境の機能

この節では、現在の rpcgen コード生成プログラムで提供されている機能について説明します。

- SunOS でのテンプレート生成 - 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、58 ページの「クライアント側およびサーバ側のテンプレートを生成するコンパイルオプション」を参照してください。
- SunOS での 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、59 ページの「C 形式モードでのコンパイル」を参照してください。
- SunOS でのマルチスレッド対応コード - rpcgen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ます。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。詳細およびコード例については、62 ページの「マルチスレッド対応コードのコンパイル」を参照してください。
- SunOS での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、68 ページの「自動マルチスレッド対応モードでのコンパイル」を参照ください。
- SunOS でのライブラリの選択 - 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。68 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。
- SunOS での 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。69 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。

ローカル手続きをリモートプロシージャに変換

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例を次に示します。

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する */

#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)
```

例 **3–1** シングルコンピュータ用の printmsg.c　(続き)

```
    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() 関数をリモートプロシージャに変換すると、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。

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、リモートプロシージャとしての 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リモートプロシージャのプロトコル */
 program MESSAGEPROG {
     version PRINTMESSAGEVERS {
        int PRINTMESSAGE(string) = 1;
      } = 1;
} = 0x20000001;
```

リモートプロシージャは常にリモート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 つだけ含まれたリモートプログラムが 1 つ宣言されています。この例では、PRINTMESSAGE という手続きが、MESSAGEPROG というリモート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。リモート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。

リモート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。つまり、既存の手続きが変更されたり、新しい手続きが追加された場合などです。リモートプログラムで複数のバージョンを定義することも、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。

- リモートプロシージャ自体
- リモートプロシージャを呼び出すクライアント側のメインプログラム

例 3–2 には、例 3–1 の手続きを PRINTMESSAGE というリモートプロシージャに変更したものを示します。

例 **3–2** RPC バージョンの printmsg.c

```
/*
 * msg_proc.c: リモートプロシージャバージョンの "printmessage"
 */
#include <stdio.h>
#include "msg.h"                /* 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);}
```

リモートプロシージャ 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. printmessage_1() では、引数が文字へのポインタではなく、文字列配列へのポインタになっています。これは、-N オプションを使用しないリモートプロシージャの場合には常に当てはまります。その場合、常に引数自体ではなく、引数へのポインタが使われます。-N オプションを指定しなければ、リモートプロシージャ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. printmessage_1() には、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が入っています。つまりプログラム番号、バージョン、そして手続き番号、 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ります。

3. printmessage_1() では、戻り値は整数自体ではなく、整数へのポインタになります。これもまた、-N オプションを使用しないリモートプロシージャの場合には常に当てはまります。そこでは、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ない場合は、戻り値は static で宣言します。戻り値をリモートプロシージャのローカル値にしてしまうと、リモートプロシージャが戻り値を返した後、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 が使用されている場合は、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、戻り値は手続きで宣言されません。
4. 手続き名を見ると、_1 が追加されてprintmessage_1 () になっています。一般に rpcgen がリモートプロシージャ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となります。

リモートプロシージャを呼び出すクライアント側メインプログラムを次に示します。

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 が生成した msg.h */

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);
    }

    server = argv[1];
    message = argv[2];
    /*
 * コマンド行で指定したサーバの
 * 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,
```

例 **3–3** printmsg.c を呼び出すクライアント側プログラム　　*(続き)*

```
                            "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);
    }
    /*
     * サーバー上のリモートプロシージャ 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);
    }
/*
 *  リモートプロシージャ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);}
```

このコード例では、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、リモートプロシージャ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リモートプロシージャ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。

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。トランスポートについての詳細については、 /etc/netconfig ファイルと 『プログラミングインタフェース』 を参照してください。

リモートプロシージャ 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 で宣言されたとお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。

リモートプロシージャ呼び出しのエラーは、2 種類あります。RPC 自体のエラーと、リモートプロシージャの実行中に発生したエラーです。最初のエラーの場合は、リモートプロシージャ 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。

printmsg 全体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。それには、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。

rpcgen は、入力ファイル msg.x から次のファイルを生成します。

- 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。
- クライアント側スタブルーチン、msg_clnt.c を作成します。 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。
- msg_proc.c の printmessage_1() を呼び出すサーバープログラムである、msg_svc.c というファイルが作成されます。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、リモートマシン上にインストールして実行できます。リモート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。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リモートマシンにコピーして再コンパイルする必要があります。この例では、リモートマシンは remote、 ローカルマシーンは local とします。次のコマンドで、リモートシステムのシェルからサーバープログラムを起動します。

```
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行からではなく、listen() や inetd() などのポートモニター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

45 ページの「ローカル手続きをリモートプロシージャに変換」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)。

次に、RPC サービスのサンプルプログラム全体を示します。これは、リモートディレクトリを一覧表示するもので、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。

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: リモートディレクトリを一覧表示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 */
};
```

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

```
/*
 * 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
 *
 * このプログラムでは、次の共用体を使用して、リモート呼び出しが
 * 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:
        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 */
};

/* ディレクトリを一覧表示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 を使用して宣言しないでください。

dir.x に対して rpcgen を実行すると、次の 4 つのファイルが出力されます。

- ヘッダーファイル
- クライアント側のスタブルーチン
- サーバー側の骨組み
- XDR ルーチンが入った dir_xdr.c というファイル dir_xdr.c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。次に、サーバー側の READDIR 手続きを示します。

例 **3–5** サーバー側の dir_proc.c の例

```
/*
 * dir_proc.c: リモートプロシージャ 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
 */
    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;
```

例 **3–5** サーバー側の `dir_proc.c` の例　　(続き)

```
    /* 結果を返す */
    res.errno = 0;
    closedir(dirp);
    return (&res);
}
```

次に、クライアント側の READDIR 手続きを次に示します。

例 **3–6** クライアント側のプログラム (`rls.c`)

```
/*
 * rls.c: クライアント側のリモート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);
        exit(1);
    }
/*
```

例 **3–6** クライアント側のプログラム (rls.c)　　(続き)

```
 * リモートプロシージャ呼び出しは正常終了
 */
    if (result->errno != 0) {
   /*
    * リモート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 の内容を表示できます。

```
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ないので、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 ルーチンの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に、 xdr_free(xdr_readdir_res ,result); を呼び出しています。

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。

rpcgen で使用される前処理命令を、次の表で示します。

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

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

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: リモート時刻のプロトコル
 */
program TIMEPROG {
    version TIMEVERS {
```

例 **3–7** 時刻プロトコルを記述する rpcgen ソースプログラム　　(続き)

```
            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

表 3–2 rpcgen コンパイル時に指定するフラグ　(続き)

オプション	フラグ	コメント
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	-A 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

クライアント側およびサーバ側のテンプレートを生成するコンパイルオプション

rpcgen ツールで、クライアント側とサーバー側のテンプレートとして使われるサンプルコードを生成します。次の表で示されたオプションを指定して、必要なテンプレートを生成します。

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。

add.x ソースプログラムから C 形式モードでサーバー側テンプレートファイルを生成するときは、コマンド行で rpcgen -N -Ss -o add_server_template.c add.x を実行します。

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、rpcgen -N -Sc -o add_client_template.c add.x を実行します。

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には、rpcgen -N -Sm -o mkfile_template add.x を実行します。

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、クライアント側とサーバー側のプログラムをコンパイルできます。-a フラグを指定して、rpcgen -N -a add.x と実行した場合には、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 形式モードでのコンパイル

-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、例 3–8と 例 3–9 に示します。

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;
```

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 {
```

例 3–9 デフォルトモードの add.x　（続き）

```
    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;
```

次に、生成されるクライアント側テンプレートを示します。

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。
 * リモート 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]);
        exit(1);
    }
    x = atoi(argv[2]);
    y = atoi(argv[3]);
/* リモートプロシージャの呼び出し: 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",
```

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` (続き)

```
                    x, y, *result);
    }
    exit(0);
}
```

次に、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。

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]);
/*
 * リモートプロシージャの呼び出し -- 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);
```

次に、C 形式モードのサーバー側手続きを示します。

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);
}
```

次に、デフォルトモードのサーバー側手続きを示します。

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);
```

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 (続き)

```
}
```

マルチスレッド対応コードのコンパイル

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。次に、 rpcgen のプロトコルファイル msg.x を示します。

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;
```

文字列はリモートプロシージャに引き渡されます。リモートプロシージャは、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のスタブを作成するには、rpcgen -M msg.x を実行します。

次に、このプロトコルファイルを使用したクライアント側プログラムの例を示します。

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);
    }
```

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　(続き)

```
    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) {
        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);
}
```

プログラムを再入可能にするために、rpcgen が生成したコードには、引数も戻り値もポインタで渡す必要があります。スタブ関数の戻り値は、リモートプロシージャ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。例 3–15 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) と例 3–16 に示す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。

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場合)

```
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,
        (xdrproc_t) xdr_int, (caddr_t)
```

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場合)　　*(*続き*)*

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

次に、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。

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;
    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);
```

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　(続き)

```
}
```

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ないでください。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。

次に、- M フラグを–N と–C のフラグと共に指定する例として、add.x を示します。

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。このファイルに対して rpcgen を実行するには、 rpcgen -N -M -C add.x と入力します。 次に、マルチスレッド対応クライアントプログラムの例を示します。

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;
```

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　　*(続き)*

```
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];
    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;
```

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 （続き）

```
        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);
}
```

次に、サーバー側の手続きを示します。

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。

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でのコンパイル

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、第 7 章 を参照してください。さらに、159 ページの「自動マルチスレッド対応モード」 も参照してください。

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。rpcgen のプロトコルファイルであるtime.x のコードです。文字列はリモートプロシージャに引き渡されます。リモートプロシージャは、文字列を表示してクライアントの文字列長を返します。

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;
```

マルチスレッド対応のスタブを作成するには、rpcgen -A time.x コマンドを実行します。

-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。

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。

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.*x* 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 に準拠したコードも出力できます。それには -C フラグを指定します。ほとんどの場合、59 ページの「C 形式モードでのコンパイル」 にあるように -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を生成するには、コマンド行で rpcgen -N -C -Ss -o add_server_template.c add.x を実行します。

ここで、C++ 3.0 で記述されたサーバー上ではリモートプロシージャ名が接尾辞 _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。

例 **3–22** ANSI C に準拠した rpcgen 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 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(3NSL) の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できます。たとえば、rpcgen -i 3 test.x というコマンドでは、パックできる要素が 3 つ以上連続していれば、 rpcgen によってインラインコードが生成されます。 コマンド行 rpcgen -i 0 test.x の実行では、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。

表 **3–4** RPC プログラミングテクニック

テクニック	説明
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る
定義文	`rpcgen` コマンド行で、C 言語のプリプロセッサシンボルを定義できる
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ない
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する
ポートモニターのサポート	RPC サーバーの代わりにポートモニターで「受信待ち」することができる
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る
タイムアウト値の変更	クライアントのタイムアウト期間のデフォルト値を変更できる
認証	クライアントはサーバーに自分自身を証明できる。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る

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詳しくは、『プログラミングインタフェース』 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、rpcgen -s datagram_n prot.x を実行すると、NETPATH 環境変数で指定された、または、NETPATH が定義されていない場合には /etc/netconfig に記述された、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。コマンド行では、-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 – 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どお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。たとえば :

```
$ 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、リモートプロシージャの戻り値を 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。

NFS サーバーの場合だけ応答する手続きを示します。

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 */
```

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　　*(続き)*

```
        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 。

この例では、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 (`¬null`) は静的変数を指していなければなりません。

ポートモニターのサポート

`inetd` や `listen` のようなポートモニターは、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、ポートモニターは、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。この方法によりシステム資源を節約することができます。`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。詳細は、 138 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ー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、ポートモニター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ーの場合は、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 20 proto.x
```

```
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。

ポートモニターについての詳細は、付録 F を参照してください。

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、87 ページの「標準インタフェース」 を参照してください。また、rpc(3NSL) マニュアルページも参照してください。 タイムアウト値を変更する場合に、ネットワークを往復するのに必要な最短時間より長くなるようにする必要があります。次のプログラム例で、clnt_control () の使用方法を示します。

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も低いクライアント認証方法のうち、一般に使用される方法を示します。よりセキュリティレベルが高い認証方法の詳細については、118 ページの「認証」 を参照してください。

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。

次のプログラム例では、クライアントの認証データをチェックするサーバープログラムが記述されています。これは、45 ページの「rpcgen チュートリアル」 で説明した printmessage_1() を修正したものです。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。

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いときなどです。または、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
};
```

ここでの定義は、次のとおりです。

proc は、サービスルーチンへのポインタ

xdr_arg は、入力(引数) の xdr ルーチンへのポインタ

len_arg は、入力引数の長さ(バイト数)

xdr_res は、出力(結果) の xdr ルーチンへのポインタ

len_res は、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_nproc 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。

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

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、手続き、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、手続き、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。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、および手続きの 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。

Solaris 7 以降の rpcgen 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を考えます。この場合生成されるファイル foo_xdr.c ファイルでは、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。

```
struct foo {
        char      c;
        int       i1;
        int       i2;
        int       i3;
        long      l;
        short     s;
};

#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。

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、手続きおよびポートにはそれぞれ専用のタイプがあります。データ値が INT32_MIN と INT32_MAX の間にない場合、xdr_long() は失敗します。また、IXDR_GET_LONG() や 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ます。u_long 変数についても同様です。詳しくは、xdr_long(3NSL) マニュアルページを参照してください。

rpcgen の IPv6 の場合の考慮事項

IPv6 トランスポートをサポートするのは、TI-RPC だけです。現在または将来的に、IPv6 を使用してアプリケーションを実行する予定がある場合は、下位互換スイッチは使用する必要はありません。IPv4 と IPv6 のどちらを選択するかは、/etc/netconfig 内の関連エントリのそれぞれの順序によって決まります。

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(3NSL) 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 をリンクしないでください。

これまでに説明したサンプルプログラムの手続きを、コマンド cc rls.c dir_clnt.c dir_proc.c -o rls でリンクします。

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。詳細については、103 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、リモートプロシージャ呼び出し過程で起こるエラーです。例を示します。

- 手続きが実行できない
- リモートサーバーが応答しない
- リモートサーバーが引数を復号化できない

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() を使用してエラーの原因を表示するか、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。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(3NSL) 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。

この章では、次のトピックについて説明します。

- 79 ページの「単純インタフェース」
- 87 ページの「標準インタフェース」
- 103 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」
- 162 ページの「マルチスレッドユーザーモード」

注 – 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ます。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 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。rnusers()

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。次の例は、リモートホスト上のユーザー数を表示するプログラムです。RPC ライブラリルーチン rusers() を呼び出して、リモートホスト上のユーザー数を表示します。

例 **4–1** rusers プログラム

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

/*
 * rnusers() 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: rnusers\n");
        exit(1);
    }
    fprintf(stderr, "%d users on %s\n", num,
                        argv[1] );
    exit(0);
}
```

次のコマンドを使用して 例 4–1 のプログラムをコンパイルします。

```
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 */
);
```

関数 `rpc_call()` 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。リモートプロシージャに渡される引数は、`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、リモートプロシージャ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、構造体の中で処理されます。

次のプログラムは、単純インタフェースを使用するために例 4–1 を変更したものです。

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 <utmpx.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,
```

例 **4–2** 単純インタフェースを使用する rusers プログラム　*(続き)*

```
                    "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、データ型の表現も異なります。したがって、 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。また、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、理解しやすいものです。サーバーは、呼び出される手続きを登録するために rpc_reg() を呼び出します。次に、RPC ライブラリのリモートプロシージャディスパッチャである svc_run() を呼び出し、要求が来るのを待機します。

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      /* リモート関数の名前 */
      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リモートプロシージャが登録されたときに指定された XDR フィルタを使用して、リモートプロシージャの引数の復号化と、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。リモート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。 NULL は、送信する応答がないことを示します。

- 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 を使用するより、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。rpcgen は、汎用のコードジェネレータであるためです。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

リモートプロシージャへ渡すデータ型とリモートプロシージャ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、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。データは、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されます。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、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が作成した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。

XDR プリミティブタイプのルーチン

```
xdr_int()                xdr_double()
xdr_netobj()             xdr_u_short()
xdr_u_long()             xdr_wrapstring()
xdr_enum()               xdr_char()
xdr_long()               xdr_quadruple()
xdr_float()              xdr_u_char()
xdr_u_int()              xdr_void()
xdr_bool()               xdr_hyper()
xdr_short()              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()`、およびxdr_hyper() (および、それぞれの符号なしバージョン) には、次の表で示すように、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 **4–1** プリミティブタイプの等価関数

関数名	等価関数名
`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()`
`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。

```
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、次に示すようになります。

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。

XDR 構築基本ルーチンを、次に示します。

```
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()` を次に示します。

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、次のように xdr_vector() を使用します。

例 **4–5** 変換ルーチン xdr_vectorxdr_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す。これは、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。xdr_string() は、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。

次の例では、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 struct simple へのポインタを変換しています。

例 **4–6** 変換ルーチン xdr_reference

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

例 **4–6** 変換ルーチン xdr_reference　(続き)

```
bool_t
xdr_finalexample(xdrsp, finalp)
    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 (top-level interface) に関する知識が必要です。

次に示したルーチンでは、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。

```
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

次のようなヘッダファイルがあるとします。

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
```

次に、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。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。

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[];
```

例 **4–8** 時刻を返すサービス : クライアント側 *(続き)*

```
{
    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);
 (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します。または、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()` を使用してリモート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、リモートプロシージャ番号、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、リモートプロシージャ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);
```

次に、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。

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();
```

例 **4–9** 時刻を返すサービス : サーバー側　　*(続き)*

```
}

/*
 * 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);
 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()` は手続きが戻り値を返してから呼び出されます。そのため、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。

この例では、リモートプロシージャには引数がありません。引数を渡す必要がある場合は次の 2 つの関数を呼び出して、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。

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

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

次のプログラムは、88 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。

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 */
```

例 **4–10** 時刻サービスのクライアント側プログラム　（続き）

```
}
```

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(3NSL) マニュアルページと 『プログラミングインタフェース』 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

次に、対応するサーバー側プログラムを示します。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。

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",
                 argv[0]);
        exit(1);
```

例 **4–11** 時刻サービスのサーバー側プログラム　　*(続き)*

```
    }
 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、『プログラミングインタフェース』 を参照してください。

クライアント側

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の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。

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;        /* リモートアドレス */
    rpcprog_t prog;                    /* プログラム番号 */
    prcvers_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;
            goto err;
        }
 }
    if (raddr->sin_port == 0) { /* 未知のリモートアドレス */
     u_short sport;
        /*
         * ユーザー作成のルーチン rpcb_getport() は
```

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　　*(続き)*

```
                  * 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);
 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。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)、 clntudp_create() 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。

リモートアドレスがわからない場合 (raddr->sin_port == 0) は、リモートの 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ます。fd 自体をオープンしたときには、clnt_destroy () を呼び出して閉じます。 そして、リトライのタイムアウト値を設定します。

サーバー側

例 4–13 に、対応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。

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;
```

例 **4–13** 下位レベル RPC を使用したサーバー側プログラム　*(続き)*

```
	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) {
		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。有効なファイル記述子が渡されなければ、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 (エキスパートレベルのルーチンで言えば、`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

次に、`clnt_vc_create()` と `clnt_dg_create()` の呼び出し方法を示します。

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

次に、ボトムレベルのサーバーを作成する例を示します。

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;
}
```

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、サーバー手続きが「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手続き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。この実装は、変更される可能性があります。

最初に示す構造体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、次に示すように接続ごとに1 つのハンドルを作成して初期化する必要があります。

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 {
```

例 **4–16** クライアント側 RPC ハンドル (CLIENT 構造体)　　(続き)

```
        enum clnt_stat    (*cl_call)();  /* リモートプロシージャ呼び出し */
        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 に設定されています。次に示すように、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。

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、118 ページの「認証」 を参照してください

次に、サーバー側のトランスポートハンドルを示します。

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 */
```

例 **4–18** サーバー側のトランスポートハンドル　　*(続き)*

```
        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;    /* リモートアドレスの長さ、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;    /* リモートトランスポートアドレス */
 char                 xp_raddr[16];    /* リモートアドレス、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;
```

次の表では、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が示されています。

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、 次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。

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。

このプログラムは、次の Makefile を使用してコンパイルとリンクを行います。

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

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;
{
```

例 **4–19** Raw RPC を使用した簡単なプログラム　　*(続き)*

```
 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;
{
    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:
```

例 **4–19** Raw RPC を使用した簡単なプログラム　　*(続き)*

```
            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。
- `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

接続型トランスポート

例 4–20 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。

例 **4–20** リモートコピー (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>
```

例 **4–20** リモートコピー (両方向 XDR ルーチン)　　*(続き)*

```
#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);
 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);
     }
    }
}
```

例 4–21 と 例 4–22 には、例 4–20 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。

例 **4–21** リモートコピー : 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;
```

例 **4–21** リモートコピー : クライアント側ルーチン　　*(続き)*

```
{
    int xdr_rcp();

    if (argc != 2 7) {
        fprintf(stderr, "usage: %s servername\n", argv[0]);
        exit(1);
    }
    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);
}
```

次に、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。

例 **4–22** リモートコピー : 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"
```

例 **4–22** リモートコピー : サーバー側ルーチン　*(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。

第 **5** 章

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
- ポートモニターの使用 - ポートモニター inetd と listener のインタフェース
- 複数のプログラムのバージョン - 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、(デフォルトの) 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。プロセスには、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指定されている必要があります。その場合、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。

例 5–1 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 – __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。

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。 関数 __rpc_select_to_poll では、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。この関数の引数は次のとおりです。

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

関数 __rpc_dtbsize() 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(3NSL) および poll (2) のマニュアルページを参照してください。

例 **5–1** 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:
         /*
             * 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
```

例 5–1 svc_run() と poll() (続き)

```
            */
        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。ただし、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リモート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るので、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。Ethernet の MTU は 1500 バイトです。

次に、rpc_broadcast() の使用方法を示し、引数を説明します。

例 5–2 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>
```

例 5–2 RPC ブロードキャスト　　(続き)

```
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);
}
```

例 5–3 の関数では、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ます。したがって、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。

例 5–3 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;
{
```

例 **5–3** ブロードキャストへの応答の収集 *(続き)*

```
 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ない
- 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) で伝送される
- 呼び出し時に指定する、戻り値に対する XDR ルーチンは NULL である
- RPC 呼び出しのタイムアウト値はゼロである

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。このバッファリングにより、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。

次に、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。

例 **5–4** 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);
}
```

次に、バッチ処理を使用したクライアント側プログラムを示します。各文字列を送信した後に応答は待ちません。サーバーからの終了応答だけを待ちます。

例 **5–5** 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;
```

例 5–5 バッチ処理を使用するクライアントプログラム　(続き)

```
    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);
    /* Now flush the pipeline */
    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);
}
```

次に、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。

例 5–6 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);
```

例 5–6 バッチ処理を行うサーバー　(続き)

```
             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);
}
```

注 – 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、例 5–4、例 5–6 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

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オープンエンド型です。したがって、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。

次の表で、RPC が現在サポートしている認証タイプを示します。

表 5–1 RPC が現在サポートしている認証タイプ

メソッド	説明
AUTH_NONE	デフォルト。認証は実行されない

表 5–1 RPC が現在サポートしている認証タイプ　(続き)

メソッド	説明
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 を参照してください。
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 5 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場合を考えます。

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

この場合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、次に示す資格 - 認証構造体が渡されます。

例 **5–7** 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。

次に、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。

例 **5–8** 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))
```

例 **5–8** 認証データをチェックするサーバープログラム　　*(続き)*

```
         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;
        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;
    }
}
```

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。例のプロトコルでは、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() デーモンが実行されている必要があります。また、NIS または NIS+ ネームサービスも実行されている必要があります。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。例 5–8 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。例 5–8 のように、(char *)NULLchar *)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。次に、例 5–8 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示します。

例 **5–9** 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;
 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 V5 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、このあとで説明します。

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。具体的には、window を authkerb_seccreate() に引数として渡します。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 のマニュアルページを参照してください。

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。インスタンスは無視されます。

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。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。次の表記では、XDR が *object* を type とみなして使用されることを示しています。

```
xdr_type(object)
```

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、 および 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 が追加されており、RPC プログラマが利用可能です。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。 それは、完全性とプライバシです。

- 「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。

  米国の輸出の規約により、プライバシサービスはすべてのユーザーが利用できるわけではありません。

注 – 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 の機能を通じて参照できます。詳細は、『*GSS-API* のプログラミング』を参照してください。

RPCSEC_GSS API

RPCSEC_GSS API 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 の機能を最大に生かすことができます。RPCSEC_GSS は、次の図のように、GSS-API 階層の「最上部」に位置しています。

<table>
<tr><td colspan="3">アプリケーション</td></tr>
<tr><td colspan="3">RPCSEC_GSS</td></tr>
<tr><td colspan="3">GSS-API</td></tr>
<tr><td>Kerberos
V5</td><td>RSA
公開鍵</td><td>その他...</td></tr>
</table>

図 **5–1** 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)。

- セキュリティサービス — 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)。デフォルトは完全性です。この項目はメカニズムに依存しません。
- QOP — 保護の質。QOP により、プライバシまたは完全性サービスを実現するため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、RPCSEC_GSS によって提供される関数により、QOP およびメカニズムのリストを入手できます。136 ページの「その他の関数」 を参照してください。 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) を含むことができるようになりました。ただし、現在のところは RPCSEC_GSS がそれに該当する唯一のタイプということになります。

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、データを交換してこのコンテキストを破棄します。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(3NSL)` のマニュアルページを参照してください。

RPCSEC_GSS ルーチン

次の表は、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。各関数の詳細については、該当するマニュアルページを参照するか、RPCSEC_GSS データ構造のリストなどの概要が記載された、`rpcsec_gss(3NSL)` のマニュアルページを参照してください。

表 5–2 RPCSEC_GSS 関数

処理	関数	入力	出力
セキュリティコンテキストの作成	`rpc_gss_seccreate(3NSL)`	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

表 5–2 RPCSEC_GSS 関数　(続き)

処理	関数	入力	出力
コンテキストの QOP とサービスタイプの変更	`rpc_gss_set_defaults(3NSL)`	古い QOP とサービス	新しい QOP とサービス
セキュリティの変換前に、データの最大サイズを示す	`rpc_gss_max_data_length(3NSL)` (クライアント側)	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
セキュリティの変換前に、データの最大サイズを示す	`rpc_gss_svc_max_data_length(3NSL)` (サーバー側)	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
表示するサーバーの主体名を設定する	`rpc_gss_set_svc_name(3NSL)`	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
呼び出し側 (クライアント) の資格を得る	`rpc_gss_getcred(3NSL)`	`svc_req` 構造体へのポインタ	UNIX 資格、RPCSEC_GSS 資格、cookie
(ユーザーの作成した) コールバック関数を指定する	`rpc_gss_set_callback(3NSL)`	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
固有のパラメータから主体名の RPCSEC_GSS 構造体を作成する	`rpc_gss_get_principal_name(3NSL)`	メカニズム、ユーザー名、マシン名、ドメイン名	RPCSEC_GSS 主体名の構造体
RPCSEC_GSS ルーチンが失敗した場合にエラーコードを得る	`rpc_gss_get_error(3NSL)`		RPCSEC_GSS エラー番号、該当する場合には errno
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	`rpc_gss_get_mechanisms(3NSL)`		有効なメカニズムのリスト
有効な QOP 文字列を入手する	`rpc_gss_get_mech_info(3NSL)`	メカニズム	そのメカニズムの有効な QOP

表 5–2 RPCSEC_GSS 関数　(続き)

処理	関数	入力	出力
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	rpc_gss_get_versions(3NSL)		最大および最小のバージョン番号
メカニズムが導入されているかどうかをチェックする	rpc_gss_is_installed(3NSL)	メカニズム	インストールされている場合は TRUE
ASCII メカニズムを RPC オブジェクト識別子に変換する	rpc_gss_mech_to_oid(3NSL)	メカニズム (文字列で)	メカニズム (OID で)
ASCII QOP を整数に変換する	rpc_gss_qop_to_num(3NSL)	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() を使用して作成します。この関数では引数として次のものをとります。

- 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@acme.com)
- セッションのメカニズム (Kerberos V5 など)
- セキュリティサービスタイプ (プライバシなど)
- セッションの QOP
- 使用される場合は、ほとんど不透明（ opaque ）なまま使用される (つまり、プログラマが NULL 値を入れることができる) 2 つの GSS-API パラメータ

この関数は、AUTH 認証ハンドルを返します。次のプログラムでは、Kerberos v5 セキュリテ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。

例 5–10 rpc_gss_seccreate()

```
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");
```

例 **5–10** rpc_gss_seccreate()　(続き)

```
clnt->clnt_auth = rpc_gss_seccreate(clnt, service_name, mech,
                          rpc_gss_svc_integrity, NULL, NULL, NULL);

. . .
```

次の点に注意してください。

- 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() を用いて、使用できるメカニズムの表から入手します。
- QOP は、NULL として渡されます。これにより、QOP はこのメカニズムのデフォルトに設定されます。これ以外の場合は、このメカニズムを使用して、rpc_gss_get_mechanisms() を指定したプログラムで有効な値を入手できます。詳細については、マニュアルページの rpc_gss_get_mechanisms(3NSL) を参照してください。
- セキュリティサービスタイプ、rpc_gss_svc_integrity は、RPCSEC_GSS タイプの enum のひとつである rpc_gss_service_t です。rpc_gss_service_t のフォーマットは、次のようになります。

  ```
  typedef enum {
       rpc_gss_svc_default = 0,
       rpc_gss_svc_none = 1,
       rpc_gss_svc_integrity = 2,
       rpc_gss_svc_privacy = 3
  }  rpc_gss_service_t;
  ```

  デフォルトのセキュリティサービスは、完全性をマップするため、プログラマは指定された rpc_gss_svc_default を入手し、同じ結果を獲得することができます。

詳細については、rpc_gss_seccreate(3NSL) のマニュアルページを参照してください。

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 およびサービス値を変更する必要がある場合があります。たとえば、プログラムの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。

例 **5–11** rpc_gss_set_defaults ()

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

例 **5–11** rpc_gss_set_defaults () （続き）

```
. . .
```

この場合、セキュリティサービスはプライバシに設定されます。 129 ページの「コンテキストの作成」 を参照してください。 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。

QOP とサービスの変更に関する詳細は、 rpc_gss_set_defaults(3NSL) のマニュアルページを参照してください。

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@eng.acme.com のように指定します。

  クライアントがセキュリティコンテキストを作成する時には、この形式でサーバーの主体名を指定します。 129 ページの「コンテキストの作成」 を参照してください。 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、rpc_gss_set_svc_name () を使用します。この関数は、引数としてこのフォーマットの主体名をとります。
- サーバーが受信するクライアントの主体名は、rpc_gss_principal_t 構造の形式をとります。それは、使用するメカニズムによって決定される、特定の長さを持つ隠されたバイト列になります。この構造については、rpcsec_gss(3NSL) のマニュアルページを参照してください。

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。次のプログラムで示すように、 rpc_gss_set_svc_name() を使用してサーバー主体名の設定をします。

例 **5–12** rpc_gss_set_svc_name()

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

principal = "nfs@eng.acme.com";
mechanism = "kerberos_v5";
req_time = 10000;        /* 資格の有効時間 */

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

Kerberos は、*req_time* パラメータを無視します。他の認証システムでは、このパラメータを使用する場合があります。

詳細については、rpc_gss_set_svc_name(3NSL) のマニュアルページを参照してください。

クライアント主体名の作成

サーバーは、クライアントの主体名で稼動する必要があります。 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 資格が存在する場合にはそれを検出するために必要です。このような主体名は、rpc_gss_principal_t 構造体ポインタとして保存されます。 rpc_gss_principal_t についての詳細は、rpcsec_gss(3NSL) のマニュアルページを参照してください。 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、この形式で主体名を生成する必要があります。

次のプログラムで示すように、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 構造体ポインタとして主体名を生成します。

例 **5–13** rpc_gss_get_principal_name()

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

rpc_gss_get_principal_name() への引数は、次のとおりです。

- 「 principal 」には、設定された rpc_gss_principal_t 構造体へのポインタが入ります。
- 「mechanism 」は、使用され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。
- 「 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 マシン名などが入ります。
- 「 domain 」には、たとえば、DNS、NIS、または NIS+ドメイン名、あるいは Kerberos の領域が入ります。

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 にはユーザー名が必ず必要です。また、オプションの場合に限り、修飾されたノード名とドメイン名が必要です (Kerberos 用語では、ホスト名と領域名)。

詳細については、rpc_gss_get_principal_name(3NSL) のマニュアルページを参照してください。

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ばなりません。 例 5–14 で示すように、 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。このうち 1 つは、呼び出し側の UNIX 資格が組み込まれた rpc_gss_ucred_t 構造体 (存在する場合) へのポインタになります。

```
typedef struct {
    uid_t   uid;          /* ユーザー ID */
    gid_t   gid;          /* グループ ID */
    short   gidlen;
    git_t   *gidlist;     /* グループのリスト */
} rpc_gss_ucred_t;
```

もう 1 つの引数は、次のような、rpc_gss_raw_cred_t 構造体へのポインタです。

```
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;
```

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() は両方の名前を返します。 rpc_gss_principal_t 構造体の解説と作成方法については、132 ページの「クライアント主体名の作成」 を参照してください。

次のプログラムは、単純なサーバー側のディスパッチ手続きの例です。サーバーは、呼び出し側の資格を入手しています。この手続きでは、呼び出し側の 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。

例 **5–14** 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;
```

例 5–14 資格の入手　(続き)

```
        }
        /*
         * 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);
            /*
            * 設定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;

}
```

詳細については、`rpc_gss_getcred(3NSL)` のマニュアルページを参照してください。

Cookies

例 5–14 では、`rpc_gss_getcred()` への最後の引数 (ここでは `NULL` になっている) 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体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 *deleg* と *gss_context* は、GSS-API データタイプで、現在はまだ公開されていません。詳細については、『*GSS-API* のプログラミング』 を参照してください。 *deleg* は委譲されたピアを識別する情報になり、一方 *gss_context* は GSS-API コンテキストへのポインタになります。プログラムが GSS-API 処理をこのコンテキスト上で実行する必要がある場合、つまり受信条件のテストをする場合に、このポインタが必要となります。 *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` 構造体へのポインタです。

```
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。例 5–14 に記載したように、QOP とサービスは、`rpc_gss_rawcred_t` 構造体内で検出できます。サーバーは、サービスと QOP の値を変更する必要はありません。ユーザー定義のコールバックが呼び出されると、*locked* は `FALSE` に設定されます。サーバーが、*locked* を `TRUE` に設定すると、QOP とサービスの値が、`rpc_gss_rawcred_t` 構造体内の値と一致する要求だけが受理されます。

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

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、通常は大きくなります。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。

詳細については、rpc_gss_max_data_length(3NSL) のマニュアルページを参照してください。

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(3NSL)() 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。
- rpc_gss_is_installed(3NSL)() 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(3NSL)() 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 5–2 および rpcsec_gss(3NSL) マニュアルページを参照してください)。

関連ファイル

RPCSEC_GSS は各種のファイルを使用して情報を保存します。

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 構造体ポインタの形式)、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。これらのサービスでは、たとえば主体名は、 rpc_gss_principal_t 構造体として直接、独自のアクセス制御リスト内に格納できます。

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（たとえば、Kerberos V5) 資格の両方が入っています。ネットワーク資格は、rpc_gss_principal_t 構造体が 16 進 ASCII 表現されています。gsscred ファイルは、XFN を通じてアクセスされます。したがって、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実装可能となります。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。システム管理者は、ユーザーやマシンを追加したり削除したりできる gsscred ユーティリティを利用して、gsscred テーブルの保守管理を実行できます。

/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が実装された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が保存されています。ASCII 形式によるメカニズム名、そのメカニズムの OID、このメカニズムによって提供されるサービスが実装された共有ライブラリ名、さらに、オプションとして、サービスが実装されたカーネルモジュール名です。次に例を示します。

```
kerberos_v5   1.2.840.113554.1.2.2    gl/mech_krb5.so gl_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。

多くのカーネル内 RPC ルーチンでは、文字列ではない値によって、メカニズムと QOP が表現されています。したがって、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、文字列ではない値で表現された、これらのパラメータを入手します。

ポートモニターの使用

RPC サーバーは、inetd や listen のようなポートモニターから起動できます。ポートモニターは、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。 付録 F も参照してください。

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、ポートモニター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ーによりすでに 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、次のように svc_reg() を呼び出してサービス手続きを登録する必要があります。

```
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。

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。

表 **5–3** RPC inetdサービス

サービス	説明
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

次に、エントリの例を示します。

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

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

リスナーの使用

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`
```

次に、コマンド行引数を示します。

-a　サービスの追加

-p *pm_tag*　サービスへのアクセスを提供するポートモニターに結合したタグの指定

-s *svctag*　サーバーの ID コード

-i *id*　サービス *svctag* に割り当てられた /etc/passwd 内のユーザー名

-v *ver*　ポートモニターのデータベースファイルのバージョン番号

-m　サービスを呼び出す nlsadmin コマンドを指定。nlsadmin コマンドには引数を渡すことができます。たとえば、rusersd という名前のリモート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、その後リスナー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、次のようにリスナーを一度止めてから再起動します。このとき rpcbind が実行されている必要があります。

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

リスナープロセスの設定などについての詳細は、 listen(1M)、pmadm(1M)、および sacadm(1M) のマニュアルページを参照してください。 また、『*Solaris* のシステム管理 *(IP* サービス*)*』の「TCP/IP プロトコルがデータ通信を行う方法」も参照してください。

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。

バージョン番号を変更できるのは、自分が所有しているファイルだけです。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追加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。

例 **5–15** 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);
```

例 **5–15** 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 (続き)

```
}
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。

例 **5–16** 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:
            /*
             * 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、次に示す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(3NSL) のマニュアルページを参照してください。

例 5–17 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 */
```

例 **5–17** クライアント側での RPC バージョン選択　　(続き)

```
    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 サービスを登録します。そして、その番号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場合 (シングルスレッドであると思われるとき) 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。

例 **5–18** 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);
        return ((SVCXPRT *)NULL);
    }
    *program = prognum;
 return (transp);
}
```

第 6 章

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 6–1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、『プログラミングインタフェース』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。

RPC の場合の IPv6 の考慮事項

IPv6 は、IPv4 の後継バージョンで、今日のインターネットテクノロジーにおいて、最も一般的に使用される階層 2 のプロトコルです。また、IPv6 は IP の次世代 (IPng) とも呼ばれています。詳細については、『*Solaris* のシステム管理 *(IP* サービス*)*』を参照してください。

ユーザーは、IPv4 と IPv6 の両方を使用できます。COTS (コネクション型のトランスポートサービス) を使用する場合、アプリケーションにより、使用する「スタック」が選択されます。この場合、TCP または CLTS (コネクションレス型のトランスポートサービス) を選択できます。

次の図では、IPv4 または IPv6 プロトコルスタックを経由して実行される、典型的な RPC アプリケーションを示しています。

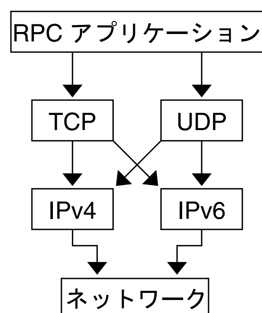


図 **6–1** RCP アプリケーション

IPv6 がサポートされるのは、TI-RPC アプリケーションだけです。現在、TS-RPC では、IPv6 をサポートしていません。TI-RPC におけるトランスポートの選択は、`NETPATH` 環境変数、または `/etc/netconfig` のいずれかによって制御されます。

IPv4 または IPv6 の代わりに、TCP または UDP を選択するかどうかは、`/etc/netconfig` 内の該当エントリの順序によって決まります。`/etc/netconfig` には、IPv6 に関連する新しいエントリが 2 つあります。また、デフォルトでは、これらのエントリは、ファイルの最初の 2 つのエントリです。まず、TI-RPC が IPv6 を試行します。失敗すると、IPv4 を試行します。この場合、RPC アプリケーション自体に変更が無いこと (つまり、トランスポートに関する情報がまったく無く、トップレベルのインタフェースを使用して書かれたものであること) が必要条件です。

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、次の表に示します。TI-RPC と TS-RPC の比較については、151 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 **6–1** 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。

表 6–1 TI-RPC と TS-RPC の相違点　(続き)

項目	TI-RPC	TS-RPC
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分けして示します。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のために残されているものです。将来のバージョンではなくなる可能性もあります。

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
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 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 リリースで使用できます。

```
*callrpc
clnt_call
*svc_getcaller - IP ベースの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

`clnt_broadcast` 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()` は `portmap` サービスにだけブロードキャストできます。`rpcbind` サービスは、サポートしていません。

rpc_broadcast 関数は、 portmap と rpcbind の両方にブロードキャストでき、現在の SunOS リリースで使用できます。

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。ただし、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

例 6–1 と 例 6–2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。
- リモートホストの RPC 結合プロセスと通信してサービスアドレスを得ます。
- リモート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。

例 **6–1** 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);
```

次の例で示すように、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。

例 **6–2** 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) {
```

例 **6–2** TI-RPC でのクライアント作成 *(続き)*

```
        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")
                            == 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);
```

例 6–3 と例 6–4 では、ブロードキャスト部分が旧バージョンと SunOS 5.*x* とでどう違うかを示します。SunOS 4.*x* の `clnt_broadcast()` は SunOS 5.*x* の `rpc_broadcast()` とほぼ同じです。大きく異なるのは、`collectnames()` 関数です。`collectnames()` 関数は、重複アドレスを削除ブロードキャストに応答したホスト名を表示します。

例 **6–3** 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++)
```

例 6–3 TS-RPC におけるブロードキャスト　　(続き)

```
        if (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 */
    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);
}
```

次に、TI-RPC におけるブロードキャストを示します。

例 6–4 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);
     } else
            printf("unknown");
    }
}
```

例 **6–4** TI-RPC におけるブロードキャスト　　(続き)

```
netbufeq(a, b)
    struct netbuf *a, *b;
{
    return(a->len == b->len && !memcmp( a->buf, b->buf, a->len));
}
```

第 7 章

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、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。

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 要求を発行できるのは一度に 1 つのスレッドだけです。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 4–1 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。

次の図は、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。

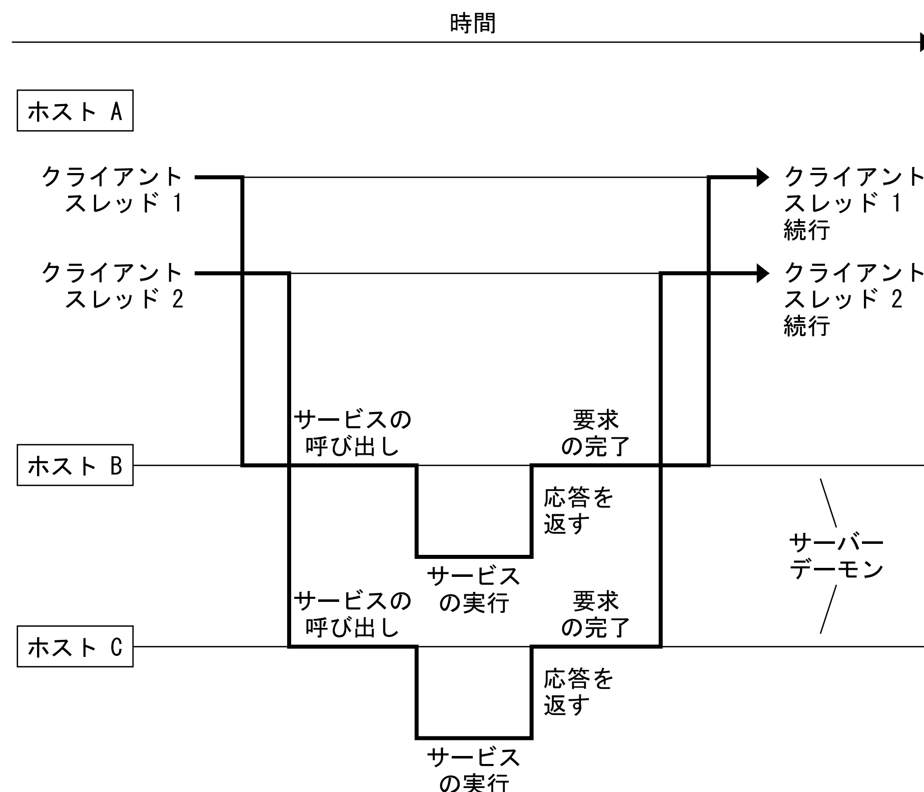


図 **7–1** 異なるクライアントハンドルを使用する 2 つのクライアントスレッド (リアルタイム)

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。スレッドライブラリは、コマンド行で最後にリンクする必要があります。そのためには、コンパイルコマンドで -lthread を指定します。

---

cc rstat.c -lnsl -lthread のように入力して、プログラムを作成します。

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 秒かかるとします。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。

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高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が提供されます。TI-RPC では、サーバーのマルチスレッドの 2 つのモードがサポートされます。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です。

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内でリンクしなければなりません。スレッドライブラリは、コマンド行で最後にリンクする必要があります。そのためには、コンパイルコマンドで `-lthread` オプションを指定します。

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 アプリケーション対応ではありません。

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(3THR)、 rwlock(3THR)、および mutex(3THR) のマニュアルページを参照してください。

次の図は、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。

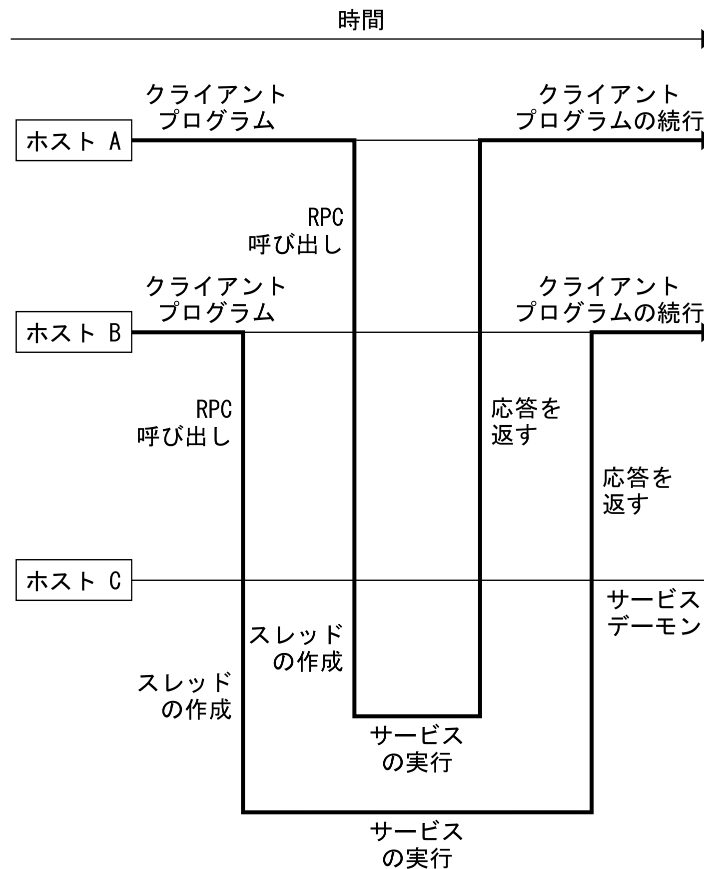


図 7–2 マルチスレッド RPC サーバーのタイミングダイアグラム

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(3NSL) のマニュアルページを参照してください。

自動マルチスレッド対応モード

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手続き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。次の表は、自動モード用に定義された rpc_control() 操作を示します。詳細は rpc_control(3NSL) マニュアルページを参照してください。

表 **7–1** rpc_control ライブラリルーチン

ルーチン	説明
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
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 7–1 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と、演算の結果が定義されません。

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できます。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。例 7–1 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` 以外の手続きによって呼び出されるごとに、引数 (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の両方においてです。詳細は、`rpc_svc_calls(3NSL)` の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。スレッドライブラリは、コマンド行で最後にリンクする必要があります。そのためには、コンパイルコマンドで `-lthread` オプションを指定します。

次の例は、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。このプログラムをコンパイルするには、`cc time_svc.c -lnsl -lthread` を実行します。

例 7–1 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";
```

例 7–1 マルチスレッド自動モードのサーバー　(続き)

```
    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;
{

       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;
```

例 7–1 マルチスレッド自動モードのサーバー　　*(続き)*

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

次に、サーバーの time_prot.h ヘッダーファイルを示します。

例 7–2 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

        #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、手続き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(3THR) の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。異なる手続きは、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 7–1 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、サービス手続きは、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、サービス手続きによって参照されるべきではありません。次の例は、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(3NSL) のマニュアルページを参照してください。

例 **7–3** 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;
```

例 **7–3** マルチスレッドユーザーモード : `rpc_test.h` (続き)

```
        SVCXPRT *transp;
    };

    extern bool_t xdr_intpair();

    #define NTHREADS_CONST 500
```

次の例は、マルチスレッドユーザーモードでのクライアントです。

例 **7–4** 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;
{
    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,
```

例 **7–4** マルチスレッドユーザーモードでのクライアント　　*(続き)*

```
        (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);
}
```

引数がない場合であっても、関数 `svc_getargs()` が `NULLPROC` 以外の各手続きに呼び出されるようにすれば、マルチスレッドのパフォーマンスが改善されます。この例では、*xdr_void* を使用できます。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のどちらにも当てはまります。詳細は、`rpc_svc_calls` (3NSL) の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。スレッドライブラリは、コマンド行で最後にリンクする必要があります。コンパイルコマンドに `-lthread` オプションを指定します。

第 8 章

Sun RPC ライブラリの拡張

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

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

以下のセクションで説明される、Sun RPC ライブラリへの機能機能は、以下のとおりです。

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

新しい機能

次に、Sun RPC ライブラリに追加された新機能を示します。

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

1 方向性メッセージング

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

次の図で、1 方向性メッセージングを示します。

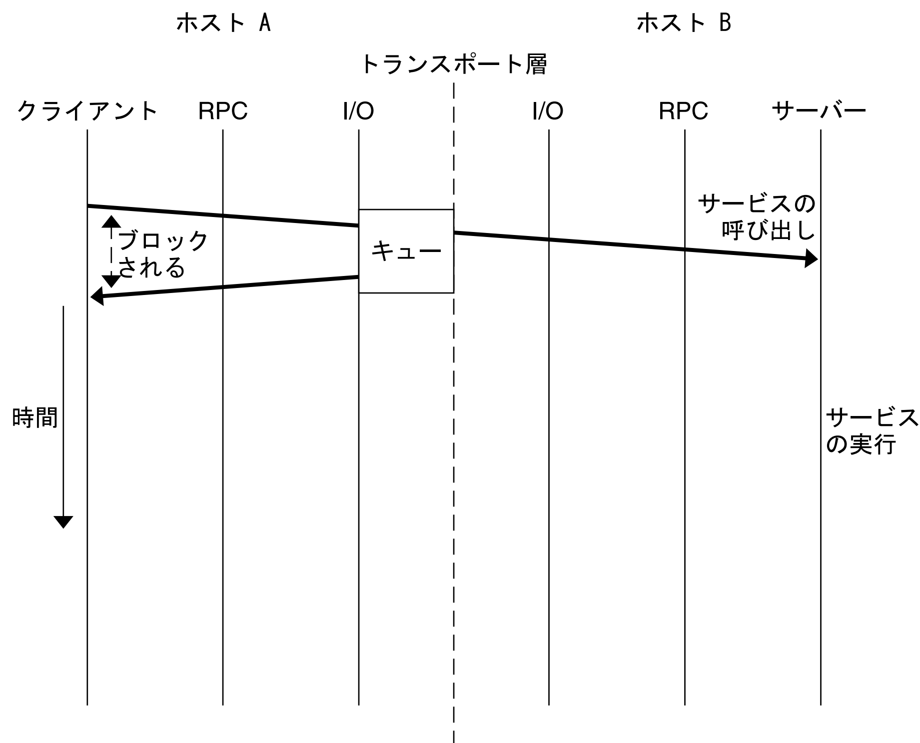


図 **8–1** 1 方向性メッセージング

Sun RPC ライブラリの以前のバージョンでは、ほとんどの要求は 2 方向性メッセージングにより送られていました。2 方向性メッセージングでは、クライアントスレッドは処理を進める前にサーバーからの応答を得るまで待つ必要があります。クライアン

トスレッドが一定の時間内にサーバーからの応答を受け取らなかった場合、タイムアウトになります。このクライアントスレッドは、1 番目の要求が実行されるかタイムアウトになるまで、2 番目の要求を送ることができません。 次の図に、メッセージングのメソッドを示します。

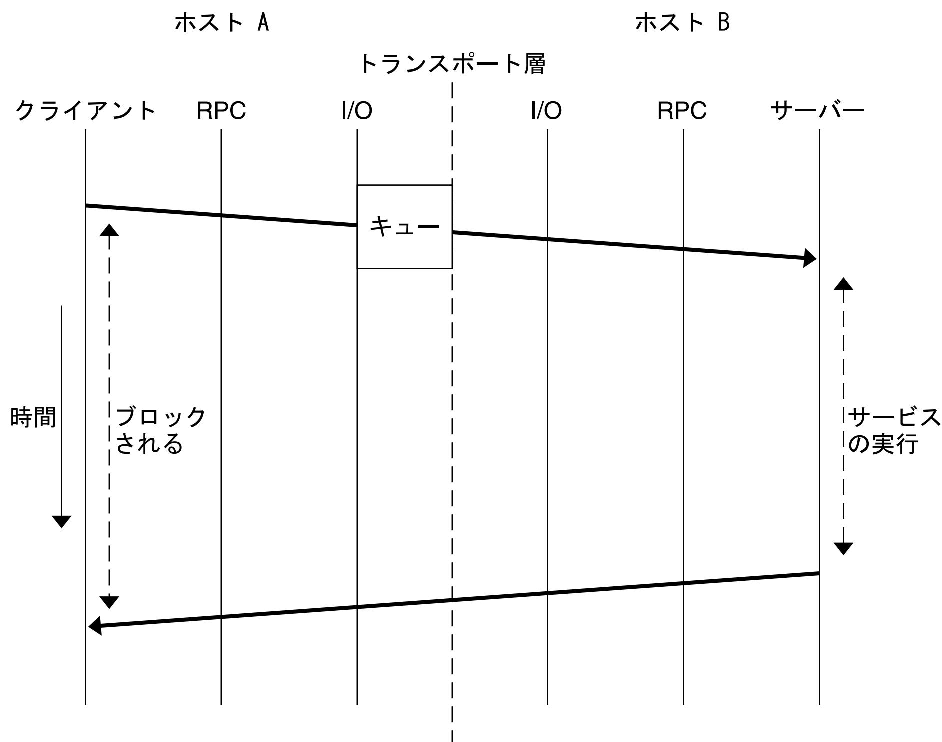


図 **8–2** 2 方向性メッセージング

Sun RPC ライブラリの以前のバージョンには、バッチングと呼ばれるもう 1 つのメッセージング方法があります。この方法では、グループ中の要求が同時に処理可能の間、クライアントの要求はキューに保持されます。これは 1 方向性メッセージングの形態です。詳細は、第 4 章 を参照してください。

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

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

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

clnt_send()

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

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

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

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

oneway 属性

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

- 第 2 章 に概説されている、simplified interface を使用します。単純インタフェースに使用されるスタブは clnt_send() を呼び出す必要があります。
- clnt_send(3NSL) のマニュアルページに記述されているように、clnt_send() 関数を直接呼び出します。

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

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

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

RPC 言語の仕様についての詳細は、付録 B を参照してください。

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

次に示す oneway 属性の情報を、267 ページの「RPC 言語の仕様」 の 「RPC 言語定義」の表に追加する必要があります。

```
type-ident procedure-ident (type-ident) = value
oneway procedure-ident (type-ident) = value
```

単純なカウンターサービスを使用する、1 方向性コール

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

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

1. 最初に、 counter.x にサービスを記述します。

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

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

2. counter.x ファイルに rpcgen を呼び出します。

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

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

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

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

   int counter = 0;
   ```

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

       counter = counter + number;

       return retval;
}
      int
counterprog_1_freeresult(SVCXPRT *transp, xdrproc_t xdr_result, caddr_t
                         result)
{
      (void) xdr_free(xdr_result, result);

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

       return TRUE;
}
```

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

4. 次に、クライアントアプリケーションである client.c を記述します。

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

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

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

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

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

```
        exit(1);
    }

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

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

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

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

5. ./server と入力して、構築したサーバーを起動します。
6. 最後に、次のように別のシェルからサービスを起動します。./client servername23

   23 は、グローバルカウンターに追加される番号です。

非ブロッキング入出力

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

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

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

非ブロッキング入出力を使用することにより、2 方向性メッセージングや 1 方向性メッセージングに比べてより多くの処理時間を得ることができます。クライアントは、処理の継続をブロックされることなく要求を続けて送信することができます 。

次の図は、入出力モードで非ブロッキングを選択し、 トランスポート層のキューが満杯であるケースを示しています。

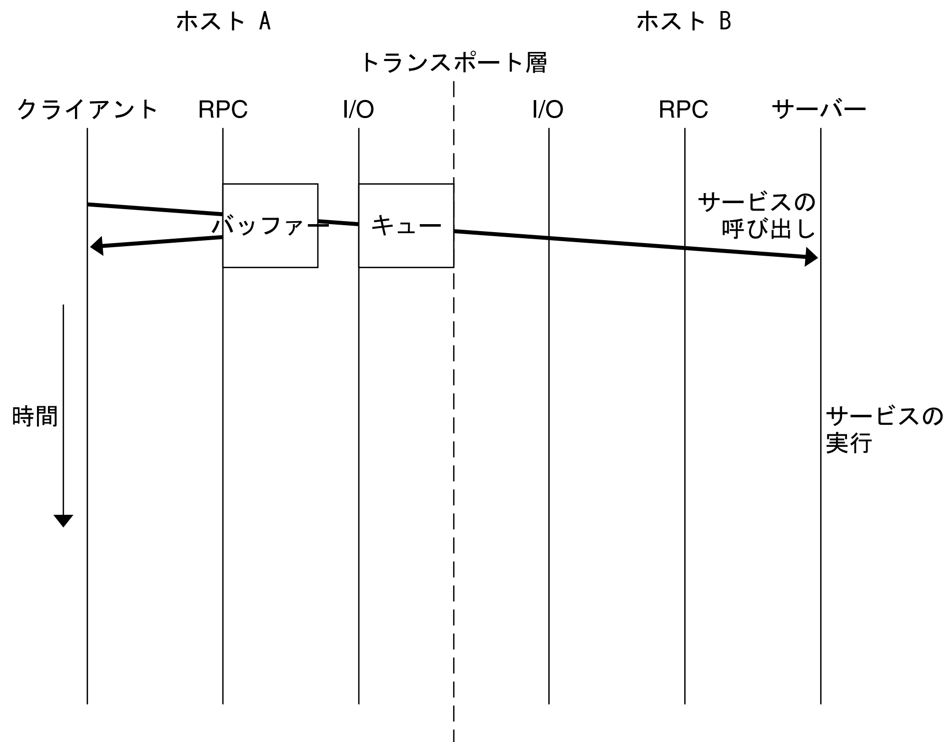


図 **8–3** 非ブロッキングメッセージング

非ブロッキング入出力の使用

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

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

- バッファーが空になる。
- キューが要求をすぐに受け入れられる。

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

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

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

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

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

非ブロッキング入出力で単純なカウンターを使用する

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

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

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

```
     * 使用する入出力モードとフラッシュメソッドを選択する。
     * この例では、非ブロッキング入出力モードと
     * ブロッキングフラッシュが選択されている。
     */
int mode = RPC_CL_NONBLOCKING;
int flushMode = RPC_CL_BLOCKING_FLUSH;

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

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

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

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

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

```
        if (!bres) {
                clnt_perror(clnt, "clnt_control");
            }
        break;
    default:
            clnt_perror(clnt, "call failed");
        break;
    }

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

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

非ブロッキングとして構成したときの clnt_call ()

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

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

非ブロッキング機能では、2 方向性と 1 方向性のコールを共に送信することができます。RPC_CL_NONBLOCKING 入出力モードを使用し、非ブロッキングとして構成したクライアントサイドで clnt_call() を使用すると、次のような動作の変更があります。2 方向性の要求がバッファーに送られると、バッファー内にすでに入っている 1 方向性のすべての要求が、その 2 方向性の要求が処理される前にトランスポート層を介して送られます。バッファーを空にするための時間は、2 方向性コールのタイムアウトにはカウントされません。詳細は、clnt_control(3NSL) のマニュアルページを参照してください。

クライアント接続クロージャーコールバック

クライアント接続クロージャーコールバックでは、サーバーが、クライアントの接続が切断されていることを検知します。サーバーは、クライアント接続が切断されたことによるエラーから回復できるようになります。トランスポートエラーは、要求がサーバーに着信した時、またはサーバーが要求を待機していて接続が切断された時に起こります。

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

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

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

`svc_control()` の引数は、次のとおりです:

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

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

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

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

クライアント接続クロージャーコールバックを使用した例

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

`log.x` ファイルは、ログプログラムのインタフェースを記述します。

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

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

                          int CLOSELOG(int logID) = 2;

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

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

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

```
RPCGEN = rpcgen

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

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

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

CFLAGS += -I.

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

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


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

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

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

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

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

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

#define NR_LOGS 3

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


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

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


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

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

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

    if (NULL == filp) return NULL;

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

```
    return filp;
}

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

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

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

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

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

    }
    res = slot;
    return &res;
}

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

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

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

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

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

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

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

#define MSG_SIZE 128

    void
usage()
{
```

```
    puts("Usage: logClient <logserver_addr>");
    exit(2);
}

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

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

    while(1) {
        struct rpc_err e;

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

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

        if (writelog_1(logID, LOG_INFO, msg, clnt) == NULL) {
            clnt_perror(clnt, "writelog");
            return;
        }
    }
         /*
          * クライアントがログを閉じる
          */
    result = closelog_1(logID, clnt);
    if (NULL == result) {
        clnt_perror(clnt, "closelog");
        return;
    }
    logID = *result;
    if (-1 == logID) {
```

```
        puts("Cannot close the log.");
        return;
    }
}

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

    if (argc != 2) usage();

    serv_addr = argv[1];

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

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

    clnt_destroy(clnt);
}
```

ユーザーファイル記述子コールバック

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

Sun RPC ライブラリの以前のバージョンでは、ユーザーが独自のサーバーループを記述するか、ソケット API にコンタクトをとる別個のスレッドを使用する場合にのみ、RPC コールおよび非 RPC ファイル記述子の両方をサーバーに受け入れさせることが可能でした。

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

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

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

```
typedef void (*svc_callback_t) (svc_input_id_t id, int fd, \ unsigned
                int revents, void* cookie);
```

callback() 関数に渡される 4 つのパラメータは、次のとおりです:

id 各コールバックに識別子を提供する。この識別子は、コールバックを削除するのに使用できます。

fd ユーザーのコールバックが待機する対象のファイル記述子。

revents 発生したイベントを表す、符号無しの整数。このイベントセットは、コールバックが登録された時に指定されたリストのサブセットです。

cookie コールバックが登録された時に指定された cookie。この cookie は、サーバーがコールバック時に必要とする特定のデータを指定することができます。

- サーバーが識別する必要のある、ファイル記述子、および読み取りや書き込みなどの関連イベントを登録するために、svc_add_input() を呼び出します。

```
svc_input_id_t svc_add_input (int fd, unsigned int revents, \
svc_callback_t callback, void* cookie);
```

指定可能なイベントのリストについては、poll(2) を参照してください。

- ファイル記述子を指定します。このファイル記述子は、ソケットやファイルなどのエンティティにすることができます。

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

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

ファイル記述子の例

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

このプログラムの makefile を次に示します。

```
   RPCGEN = rpcgen

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

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

RPCGEN_FILES = timeofday_clnt.c timeofday_svc.c  timeofday.h

CFLAGS += -I.
```

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

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


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

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

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

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

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


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

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

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

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

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

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

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

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

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

} Link;

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

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

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

```
   #include "timeofday.h"

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

    void
runClient();
    void
runSocketClient();


char* serv_addr;

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

    int
main(int argc, char* argv[])
```

```
{
    CLIENT* clnt;
    int sockClient;

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

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

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

    serv_addr = argv[sockClient? 2:1];

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

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

    time_t now;

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

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

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

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

```
    /*
          * サーバーに現在時刻を要求する。
          */
    serverTime = gettimeofday_1(clnt);

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

    clnt_destroy(clnt);
}

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

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

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

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

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

         /*
          * ソケットを接続する。
          */
    if (-1 == connect(s, (struct sockaddr*)(&sin),
                    sizeof(struct sockaddr_in))) {
```

```
        perror("cannot connect the socket.");
        return;
    }

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

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

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

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

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

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

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

int listenSocket;

/*
 * RPC サーバーの実装
 */

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

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

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

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

    return &result;
}

/*
     *ソケットサーバーの実装
 */

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

         /*
          * ソケットの作成
          */
    listenSocket = socket(PF_INET, SOCK_STREAM, 0);

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

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

    if (bind(listenSocket, (struct sockaddr*)&sin, sizeof(sin)) == -1) {
        perror("cannot bind the socket.");
        close(listenSocket);
        return -1;
    }
         /*
          * サーバーはクライアントの接続
          * が作成されるのを待機する。
          */
    if (listen(listenSocket, 1)) {
        perror("cannot listen.");
```

```
        close(listenSocket);
        listenSocket = -1;
        return -1;
    }

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

    return 0;
}

/*
 * socket_new_connection コールバック関数の定義。
 */
    void
socket_new_connection(svc_input_id_t id, int fd,
                      unsigned int events, void* cookie)
{
    Link* lnk;
    int connSocket;

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

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

         /*
          * 新規のコールバック socket_read_callback が作成された。
          */
    lnk->in_id = svc_add_input(connSocket, POLLIN,
                   socket_read_callback, (void*)lnk);
}
/*
 * 新規のコールバックsocket_read_callback が定義される
```

```
 */
    void
socket_read_callback(svc_input_id_t id, int fd, unsigned int events,
                     void* cookie)
{
    char buffer[128];
    int len;
    Link* lnk = (Link*)cookie;

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

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

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

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

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

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

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

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

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

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

/*
 * 時間をクライアントへ送信する
 */
    if (write(fd, lnk->out_ptr, len) == len) {
    /*
          * すべてのデータが送られた。
          */

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

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

    void
main()
{
    int res;

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

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

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

第 9 章

NIS+ プログラミングガイド

この章では NIS+ アプリケーションプログラミングインタフェースの基本原理を説明し、詳細なサンプルプログラムを提示します。 NIS+ API は、Solaris 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。

この章の内容は、次のとおりです。

- 197 ページの「NIS+ の概要」
- 201 ページの「NIS+ の API」
- 205 ページの「NIS+ サンプルプログラム」

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

このセクションでは、NIS+ ネットワークネームサービスのさまざまな面について説明します。

NIS+ のドメイン

NIS+ は次の図のような階層型ドメインをサポートします。

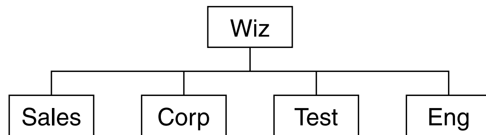


図 **9–1** NIS+ のドメイン

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。この独立性により NIS+ は、小規模から大規模まで、広い範囲のネットワークで使用できます。

NIS+ とサーバー

各ドメインは複数のサーバーによってサポートされます。このうち中心となるサーバーをマスターサーバー、バックアップサーバーを複製サーバーといいます。マスター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ーサーバーが保管し、複製サーバーはそのコピーを保存します。

NIS+ は、複製サーバーの増分更新を行います。最初にマスター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達され、名前空間全体で有効となります。

NIS+ 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+ には次の図のような、16 種類の事前定義されたシステムテーブルがあります。

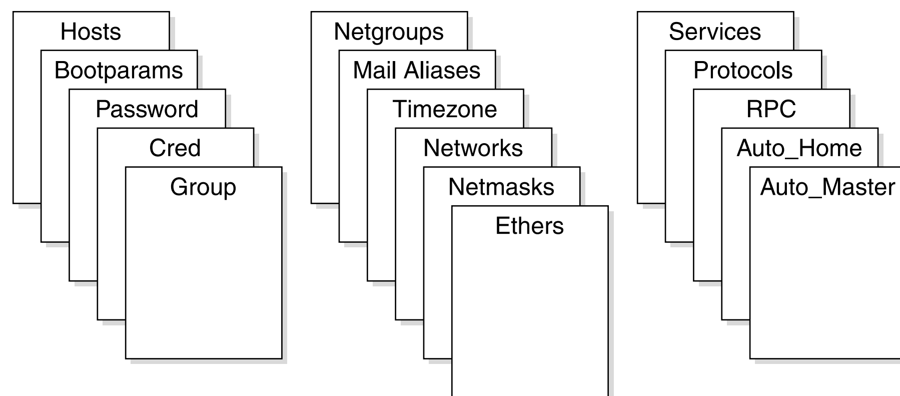


図 **9–2** NIS+ のテーブル

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。このアクセスにより、NIS では必要であった `hosts.byname` や `hosts.byaddr` などの複製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、承認と認証の 2 つの機能があります。承認については、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承認するかを指定します。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の認証を試みます。NIS+ は、要求の発信元を特定後、その特定の主体に対する特定の操作をオブジェクトが承認しているか判定します。NIS+ はこのように、認証情報とオブジェクトごとの承認情報に基づいて、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呼ばれる別の機能と連係して動作します。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 ベースの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。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、単にソース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ソースを使用できます。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+ は次の表に一覧表示した名前空間を管理するため、全種類のコマンドを提供します。

表 **9–1** NIS+ 名前空間管理コマンド

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+ グループの作成と削除、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+ パスワードテーブルに保管されているパスワード情報を変更する。
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、そのマスターサーバーと複製サーバーを指定する
nisrmdir	名前空間から、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。

表 9–1 NIS+ 名前空間管理コマンド　(続き)

コマンド	説明
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期間の値を変更する
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、ディレクトリ検索パス、生存期間) を一覧表示する
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によって管理される NIS+ 共有キャッシュの内容を一覧表示する

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。

- オブジェクト操作関数 (nis_names)
- テーブルアクセス関数 (nis_tables)
- ローカル名関数 (nis_local_names)
- グループ操作関数 (nis_groups)
- サーバー関連関数 (nis_server)
- データベースアクセス関数 (nis_db)
- エラーメッセージ表示関数 (nis_error)
- トランザクションログ関数 (nis_admin)
- その他の関数 (nis_subr)

各グループの関数について次の表で簡単に説明します。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 9–2 NIS+ の API 関数

機能	説明
nis_names()	オブジェクトの検出と操作を行う

表 9–2 NIS+ の API 関数　(続き)

機能	説明
`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、削除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
`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、キャッシュに書き込ま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。この関数へのコールとコールの間に、テーブルの更新やエントリの追加、削除、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

表 **9–2** NIS+ の API 関数　　(続き)

機能	説明
`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()`	完全指定のワークステーション名を返す完全指定の名前は *host-name domain-name* の形式を取る
`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 ()`	呼び出し側プロセスに関連付けられた UID を持つ NIS + 主体名を返す
`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
`nis_groups()`	グループ操作と承認
`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一覧表示する
`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取りを行い、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

表 9–2 NIS+ の API 関数　(続き)

機能	説明
nis_freetags()	nis_servstate() とnis_stats() の戻り値に割り当てられていたメモリーを解放する
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() を使用するかまたは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
nis_ping	ディレクトリのマスターサーバーが、ディレクトリのタイムスタンプの作成に使用する。この関数を実行すると、ディレクトリの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+ の名前とオブジェクトの操作のための補助関数
nis_leaf_of()	NIS+ の名前の最初のラベルを返す。返された名前には、末尾のピリオドは付いていない

表 9–2 NIS+ の API 関数 (続き)

機能	説明
nis_name_of()	名前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名前は、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、名前が同じか、派生関係にあるか、無関係かを返す
nis_clone_object()	NIS+ オブジェクトの正確な複製を作成する
nis_destroy_object()	nis_clone_object() で作成したオブジェクトを削除する
nis_print_object()	NIS+ オブジェクト構造体の内容を stdout へ出力する

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 の下の名前空間を一覧表示します。
- コールバックを使用して、作成したテーブルの内容を一覧表示します。
- 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

このアプリケーションを動作させる NIS+ 主体はローカルドメイン内にディレクトリオブジェクトの作成権を持ちます。次のように入力してプログラムを編集します。

```
% cc -o example.c example -lnsl
```

次のように入力して呼び出します。

```
% 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権を持っている必要があります。

次のサンプルコードは、ディレクトリオブジェクトを作成するため `main()` が呼び出すルーチンを示しています。

例 **9–1** 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。

例 **9–2** 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);
    }
}
```

main () は次の例に示すルーチンを呼び出し、テーブルオブジェクトを次の表のようなレイアウトで作成します。

表 **9–3** NIS+ テーブルオブジェクト

	カラム **1**	カラム **2**
名前	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、サービスでカラムが検索可能となります。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。

例 **9–3** 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] = {
        {"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;
```

例 **9–3** テーブルオブジェクトを作成する NIS+ ルーチン　　*(続き)*

```
    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);
}
```

main() は次の例のようなルーチンを呼び出し、テーブルオブジェクトへ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。

例 **9–4** 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;
    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。
     */
```

例 **9–4** テーブルにオブジェクトを追加する NIS+ ルーチン　　*(続き)*

```
    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);
    }
}
```

次の例に示すルーチンは `nis_list()` の呼び出しで使用する出力関数です。 `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出力されます。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。

例 **9–5** `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;
    }
```

例 **9–5** nis_list を呼び出す NIS+ ルーチン　　*(続き)*

```
    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 */
}
```

main () は次の例に示すルーチンを呼び出し、グループ、テーブル、ディレクトリオブジェクトの内容を一覧表示します。 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() コールではなく、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、グループは groups_dir を付けた形式で指定しなければなりません。

例 **9–6** オブジェクトを一覧表示する 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);

    /*
```

例 9–6 オブジェクトを一覧表示する NIS+ ルーチン　(続き)

```
	 * 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。
	 * of nis_list().
	 */
	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);

	/*

	 * 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);
}
```

cleanup() は次の例に示すルーチンを呼び出し、名前空間からディレクトリオブジェクトを削除します。このルーチンはまた、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。

例 **9–7** 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);
    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);
        }
    }
}
```

main() が呼び出す次のルーチンは、このサンプルプログラムで作成したすべてのオブジェクトを削除します。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。必ずテーブルからすべてのエントリを削除してから、テーブル自体を削除します。

例 **9–8** 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);

    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);
```

例 **9–8** オブジェクトをすべて削除する NIS+ ルーチン　　*(続き)*

```
    /* 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, dirobj-
>DI_data.do_servers.do_servers_val,
                          dirobj->DI_data.do_servers.do_servers_len);
}
```

プログラムを実行すると、画面上に次のように表示されます。

例 **9–9** NIS+ プログラムの実行

```
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%
```

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。最初のコマンドはマスターサーバーの名前を表示します。

変数 *master* と表示されている部分には、マスターサーバー名を入力してください。

```
% niscat -o domainname .

% 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、Sun が XDR (External Data Representation: 外部データ表現) 標準規格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

XDR は Sun のリモートプロシージャコール (RPC) パッケージの中枢です。 RPC 用データはこの標準規格を使って転送されます。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するプログラマは、このテクニカルノートやプロトコル仕様を読んでください。

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> をインクルードする必要があります。<rpc/xdr.h> には、必要な XDR システムとのインタフェースがすべて入っています。ライブラリ libnsl.a にはすべての XDR ルーチンが含まれているため、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。小さなプログラム上の変更の影響は外見上明らかにならない場合もありますが、しばしば、大きな影響を持ちます。次の 2 つのサンプルプログラム (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。

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);
}
```

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);
}
```

次の理由から、2 つのプログラムは移植可能と考えられます。

- `lint` チェックにパスしている
- どのハードウェアアーキテクチャ上でローカルに実行した場合も同じ動作を示す

`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 Intel マシン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%
intel% writer | reader
0 1 2 3 4 5 6 7
```

```
intel%
```

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 を使用してネットワークパイプを構築できます。次は SPARC システム上で writer がデータを生成し、そのデータを Intel アーキテクチャ上で reader が使用した場合です。

```
sun% writer | rsh intel reader
0 16777216 33554432 50331648 67108864 83886080 100663296
117440512
sun%
```

Intel 上で writer 、SPARC 上で reader を実行した場合も同一の結果となります。この原因は、Intel 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。

注 – 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 が置かれます。

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、データの移植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、データを移植可能にすることができます。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。次のコーディング例では、writer の改訂バージョンを示します。

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);
}
```

次のコーディング例は reader の改訂バージョンです。

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 システム上で実行した結果、Intel 上で実行した結果、および SPARC から Intel 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%
intel% writer | reader
0 1 2 3 4 5 6 7
intel%
sun% writer | rsh intel reader
0 1 2 3 4 5 6 7
sun%
```

注 – どのデータ構造体にも移植性の問題が生じる場合があり、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。新たなマシンは標準データ形式とローカルデータ形式の変換方法を「学習」します。 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。

標準データ形式によるアプローチは、XDR 以前より一般に行われていた方法です。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。

標準化アプローチの欠点は、同一のバイト順序を持つ 2 つのマシン間でデータを転送する際に、本来ならば必要ない XDR 標準形式への変換、また、XDR 標準形式からの変換が必要となる点です。 たとえば、バイト順序が Intel 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 Intel から 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。

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。同じように、ツリー構造を受信する場合も、各リーフへ記憶領域を割り当て、バッファからリーフへデータを移動し、正しく境界を合わせ、ポインタを構築してリーフを正しく接続する、という作業が必要になります。

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に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、移植の問題を解決し、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。このため、ネットワーク上のマシン間でデータを共有しない場合でも XDR ライブラリは有用です。

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。

例 A–3 および例 A–4 を注意して見てください。XDR ストリーム作成ルーチンファミリの各メンバー間で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。これは両方の操作に同じルーチンを呼び出すというものです。 この方法によりシリアライズされたデータのデシリアライズがほぼ保証されます。

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。このコンセプトは、常に、オブジェクト自体ではなく、オブジェクトのアドレスを渡すことにより実装されます。デシリアライズの場合のみ、オブジェクトは変更されます。 この機能は、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、238 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。このパラメータは単にサブコンポーネントルーチンに渡されているだけです。 サブルーチンが異常終了した場合は、ルーチンは各 XDR 呼び出しの戻り値を調べ、すぐに停止し、`FALSE` を返さなくてはなりません。

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
```

これらの定義を使用し、`xdr_gnumbers ()` を次のように書き換えることができます。

```
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場合や、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 変換ルーチン用のメモリーの割り当てと割り当て解除を制御する必要がある場合は、 `xdr_sizeof()` ルーチンを使用します。このルーチンは XDR フィルタ関数の 1 つ (`func()`) を使用して、データの符号化と復号化に必要なバイト数を返します。`xdr_sizeof()` 関数が返す値には、RPC ヘッダーやレコードマーカーは含まれません。必要なメモリーサイズを正確に計算するにはこれらのバイト数も追加し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)
```

RPC 環境外で 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や、通信プロトコルを選択するとき、下位レベルの RPC でクライアント作成関数とサーバー作成関数を実行するときに、`xdr_sizeof()` を使用してください。

次の 2 つのコーディング例は `xdr_sizeof ()` の 2 つの使用例を示しています。

例 **A–5** `xdr_sizeof` の例 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、
```

例 **A–5** xdr_sizeof の例 1　（続き）

```
 * 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);
                    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);
    }
}
```

次は xdr_sizeof() の 2 つめの使用例です。

例 **A–6** xdr_sizeof の例 2

```
#include <sys/statvfs.h>

#include <sys/sysmacros.h>
```

例 **A–6** xdr_sizeof の例 2　(続き)

```
/*
* 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);
}
```

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、暗号化したりすると、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 パラメータ *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

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。XDR 処理 XDR_FREE がこの方法を提供します。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。

外部では、文字列は一連の ASCII 文字で表現されます。内部では、文字列は文字ポインタで表現されます。 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。たとえば、 XDR_ENCODE 処理では、引数 *sp* は一定の長さの文字列を指します。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 を超えることはできません。次に sp を間接参照します。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 を間接参照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 変換はできません。

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。

- 配列の長さ、バイト数、符号なし整数内のバイト数。
- バイトシーケンスが NULL で終わらない。
- バイトの外部表現と内部表現が同じ。

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 を間接参照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。

他の合成データ型を定義する前に、3 つの例を示します。

配列変換サンプルプログラム 1

ネットワーク上のマシンのユーザーは次に基づいて特定できます。

- マシン名
- ユーザーの UID。これについては、マニュアルページ getuid(2) を参照してください。
- ユーザーが所属するグループ番号。マニュアルページ getgroups(2) を参照してください。

この情報を持つ構造体とそれに関連付けられた XDR ルーチンは次のコーディング例に示す方法でコード化できます。

例 **A–7** 配列変換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グループを netuser 構造体の配列で表すことができます。次のコーディング例では宣言およびそれに関連付けられた XDR ルーチンを示します。

例 **A–8** 配列変換サンプルプログラム 2

```
struct party {

     u_int p_len;
     struct netuser *p_nusers;
};
#define PLEN 500 /* グループ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* を組み合わせて構造体を作成します。これらの構造体の配列でコマンドヒストリを作成できます。構造体の宣言とその XDR ルーチンは次の例のようになります。

例 **A–9** 配列変換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)

     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));
}
```

このプログラムの複雑な点は、`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 パラメータを提供するルーチン `xdr_wrapstring()` が必要になります。`xdr_wrapstring()` が `xdr_string()` へ第 3 パラメータを提供します。

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。合成データ型についてさらに説明します。

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* は隠されたオブジェクトのアドレスを、 *len* は隠されたオブジェクト内のバイト数を示します。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 システムには、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。次のコーディング例は xdr_netobj() の構文を示します。

例 **A–10** xdr_netobj ルーチン

```
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 バイトを超えることはできません。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 ライブラリは、次のサンプルプログラムのように、固定長配列用のプリミティブ xdr_vector() を提供します。

例 **A–11** xdr_vector ルーチン

```
#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)(); /* NULL と同じ */
```

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;
};
```

次のサンプルプログラムでは、XDR プロシージャを構築し、識別型の共用体をデシリアライズします。

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}
     /* arm 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));
}
```

ルーチン xdr_gnumbers() は 222 ページの「XDR ライブラリ」 内で説明しました。 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 配列に表示された値のどれかになります。例 A–12 もまたアームの配列内要素をソートする必要がないことを示しています。

要素識別子は、この例では連続した値をとっていますが、連続しない値でも構いません。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として明記でき、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。

ポインタ

C では構造体内に別の構造体へのポインタをおくと便利な場合があります。 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、ポインタだけで十分であるため、構造体内で構造体を記述する必要はありません。

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、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、ある場合は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

この節で説明するプリミティブを使って XDR ストリームを操作することができます。

```
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の位置を記述する符号なし整数を返します。

注意 – XDR ストリームの中には、xdr_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。 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不定です。

処理内容

XDR_ENCODE、 XDR_DECODE、または XDR_FREE 処理内容を利用して、XDR ルーチンを最適化したいと考える場合もあるでしょう。XDR の処理内容は、常に xdrs->x_op に入っています。242 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() ルーチン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 は使用するメモリーを指します。パラメータ 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 xdrmem_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、4.2 BSD 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、上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、240 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、256 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次のとおりです。

```
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。指定しない場合は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データがなくなった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

次のサンプルプログラムの構造体は、XDR ストリームへのインタフェースを定義します。

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 {
```

例 **A–13** XDR ストリームインタフェースの例　　*(続き)*

```
    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 */
     caddr_t x_base;            /* 位置情報のためのプライベートデータ */
     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ないでください。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なので、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できません。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。ストリームは、そのバッファセグメント内のバイトは消費されたと解釈します。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。

注意 – x_inline() ルーチンは、バッファを反復して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。この機能は使用しないでください。

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* は同じ文字列で置換します)。

```
bool_t
xxxbytes(xdrs, buf, bytecount)
   XDR *xdrs;
```

```
   char *buf;
   u_int bytecount;
```

x_getint32() と x_putint32() の演算により、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。これらのルーチンはマシン表現形式と外部 (標準) 表現形式の間で整数を変換します。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、負にならない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、これらの演算を使用します。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;
```

これらの演算の 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、プログラムがどのようなマシン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 XDR の機能

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、任意の長さのリンクリストがあります。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。この言語の詳細は 付録 C を参照してください。

リンクリスト

236 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。次のサンプルプログラムでは、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。

例 **A–14** リンクリスト

```
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ダーはデータオブジェクトと考えてください。 便宜のため付けられた構造体の単なる短縮形ではありません。 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、ブール値は次のデータがあるかどうかを示します。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 宣言内には明示的にブール値が宣言されていませんが、gn_next フィールドは黙示的に情報を示している点に注意してください、XDR データ記述では明示的にポインタが宣言されません。

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 についての記述から得られます。上の XDR 共用体を実現するためのプリミティブ xdr_pointer() の使用方法を参照してください。

例 **A–15** xdr_pointer

```
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}
```

これらのルーチンを持つリスト上で XDR を使用すると、リスト内のノード数に比例して C スタックが大きくなります。これは再帰呼び出しが原因です。次の例では、互いに再帰呼び出しを行う 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ています。

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。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、リスト中の次のポインタアドレスを nextp の値として設定します。この値を設定するのは、リスト内の次の項目のアドレスを検出するために、 *gnp* を間接参照する必要があるためです。次の文以降では、*gnp* がポイントする記憶領域は解放され無効となります。しかし、次の文までは、XDR_DECODE 処理内容内に *gnp* の値が設定されていないため、すべての処理内容にはこの値を設定することはできません。

次にプリミティブ xdr_reference() を使用して、このノードのデータ上で XDR を使用します。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。この情報に対しすでに XDR を使用しているため、xdr_pointer() の代わりに xdr _reference () を使用することができます。

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 を間接参照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を使用する場合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く、項目数がせいぜい 100 以下のため、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。

この付録の内容は、次のとおりです。

- 247 ページの「プロトコルの概要」
- 250 ページの「プログラムと手続き番号」
- 256 ページの「認証プロトコル」
- 267 ページの「RPC 言語の仕様」

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
  - リモート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、たとえば手続きのパラメータの仕様が間違っているなど
  - リモート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、下位レベルのファイル入出力を処理し、読み書きなどの手続きを持ちます。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上のユーザーのために、この 2 つのサービスプログラムに関連付けられた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、リモートプロシージャコールは、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、応答メッセージを待ちます。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パラメータが含まれています。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、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 コール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、着信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。このため、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、サービスに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、リモートプロシージャやその実行に特定の意味論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応答なしの短時間のタイムアウトの後に RPC メッセージを再送した場合、その手続きは 0 回以上実行されたと推論されます。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 を使用します。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。

一方、TCP のような信頼性の高いトランスポートを使用する場合、アプリケーションは、応答メッセージが返されれば、手続きは正確に 1 回実行されたと推論できます。応答メッセージが受け取られないと、アプリケーションがリモートプロシージャは実行されなかったと推論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、リモートプロシージャ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。ソフトウェアは RPC 自体を使用できます。 276 ページの「rpcbind プロトコル」 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。ローダーは、JSR 命令を実行可能にするために、ローダー自身も 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、256 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。

- リモートプログラム番号
- リモートプログラムのバージョン番
- リモートプロシージャ番号

プログラム番号は、252 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。新規プロトコルは通常、品質と安定性がより高い、成熟したプロトコルへと進化します。このため、コールメッセージのバージョンフィールドで、呼び出し側が使用するプロトコルのバージョンを指定します。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 を読み取り、手続き番号 12 を書き込みなどと記されます。

リモート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 のリモート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
- リモートシステム上で指定したリモートプログラムが使用できない
- リモート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。サポートしているリモート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
- サーバー側がリモートプロシージャへのパラメータに誤りがあると解釈す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、資格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、256 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

次の表のように、プログラム番号は 0x20000000 のグループへ分散されます。

表 **B–1** RPC プログラム番号

プログラム番号	説明
00000000 から 1fffffff	ホストが定義
20000000 から 3fffffff	ユーザーが定義
40000000 から 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 から 7fffffff	予約
80000000 から 9fffffff	予約
a0000000 から bfffffff	予約
c0000000 から dfffffff	予約
e0000000 から 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、Sun 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いるので、使用しないでください。

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, 901 San Antonio Road, Palo Alto, CA 94043

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 プロトコルはリモートプロシージャ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なので、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コール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、115 ページの「バッチ処理」 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、113 ページの「ブロードキャスト RPC」 と 276 ページの「rpcbind プロトコル」 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、次の例のように、トップダウン方式で定義されます。

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、
 * リモートプロシージャ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,
              /* リモート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, /* リモート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
 */
enum reject_stat {
     RPC_MISMATCH = 0,  /* RPC のバージョン番号が 2 でない */
     AUTH_ERROR = 1     /* リモート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
 */
```

例 B–1 RPC メッセージプロトコル　(続き)

```
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、
 * リモートプログラム、そのバージョン番号、リモート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 (認証資格) と verf (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、
 * リモートプロシージャ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;
     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
```

例 **B–1** RPC メッセージプロトコル　　*(続き)*

```
 */
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
 * サポートしているリモート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、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;
```

例 **B–1** RPC メッセージプロトコル　　*(続き)*

```
        } mismatch_info;
    case AUTH_ERROR:
        auth_stat stat;
};
```

レコードマーク標準

RPC メッセージを TCP のようなバイトストリームトランスポートの最上位に渡す場合は、メッセージごとに区切り、検出やプロトコルエラーからの回復を可能にします。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` から `(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。

- そのフラグメントがレコード内の末尾のフラグメントであるかどうかを指定するブール値。 ビット値が 1 の場合はそのフラグメントは末尾のフラグメント
- 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 標準形式ではありません。

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 Sun で保守、管理しています。RPC プログラム番号などの認証番号の割り当てや登録を行うには、Sun の RPC 管理者までご連絡ください。

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。この場合、RPC メッセージの資格、ベリファイア、応答ベリファイアの *flavor* 値は `AUTH_NONE` にします。flavor 値は `opaque_auth` 共用体の要素識別子です。`AUTH_NONE` 認証 flavor を使用する場合、その body の長さは 0 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。リモートプロシージャを呼び出す側では、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_SYS 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	呼び出し側のマシンで生成できる任意の ID
machinename	呼び出し側マシンの名前
uid	呼び出し側の実効ユーザー ID
gid	呼び出し側の実行グループ ID
gids	呼び出し側がメンバーであるグループの可変長配列

資格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 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 資格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 構造体を呼び出し側の元の資格にマップするキャッシュを保存します。AUTH_SHORT タイプの応答ベリファイアがこれらの構造体を返します。呼び出し側は、新たな資格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。フラッシュが発生した場合、リモートプロシージャコールメッセージは認証エラーにより拒否されます。エラー原因は AUTH_REJECTEDCRED になります。この場合、次の図のように呼び出し側では、AUTH_SYS タイプの元の資格を試すこともできます。

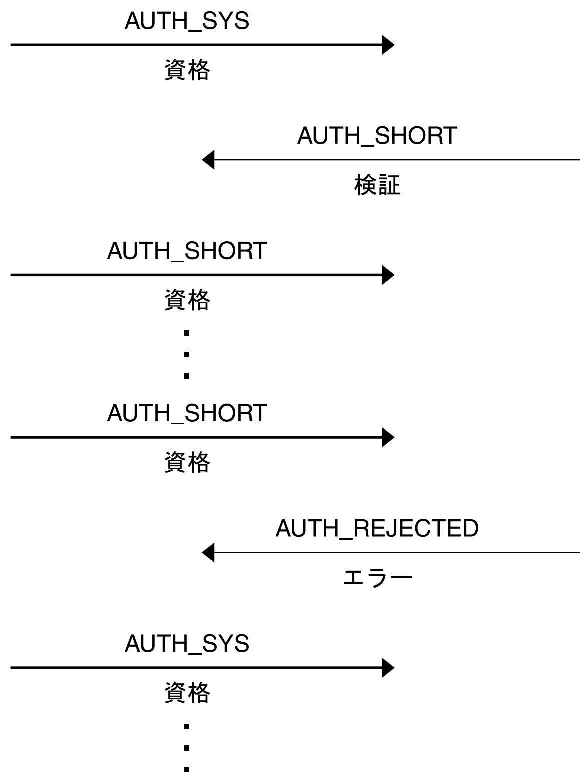


図 **B–1** 認証プロセスマップ

AUTH_DES 認証

AUTH_SYS 認証を使用した場合には、次の状況が発生することがあります。

- 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らない
- ベリファイアが存在せず、容易に資格を偽造可能

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指定します。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。

リモート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では既に、システムのローカルユーザーを識別しています。通常この機構を簡単にネットワークへ拡張できます。

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。末尾から見ていきます。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。世界共通の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格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資格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 262 ページの「Diffie-Hellman の暗号化手法」 で詳しく説明します。

この検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。このウィンドウは資格の寿命とみなすことができる。

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。さらに検査を行うため、クライアントは最初のトランザクション内へウィンドウベリファイアと呼ばれる暗号化した値を送ります。このベリファイアはウィンドウの値より 1 だけ小さい値に、それ以外の場合はサーバーは資格を拒否します。

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。クライアントへこれ以外の値が返された場合、ベリファイアは拒否されます。

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、サーバーの AUTH_DES 認証サブシステムは、クライアントへのベリファイアの中に整数のニックネームを返します。以降のトランザクションにおいて、クライアントはネットワーク名、暗号化された DES 鍵、およびウィンドウを毎回渡す代わりに、このニックネームを使用します。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、やがて同期が取れなくなることがあります。その場合、クライアント側の RPC サブシステムは RPC _AUTHERROR を受け取るので、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。サーバーのニックネームテーブルのサイズは限られており、必要に応じてエントリをフラッシュできます。その場合、クライアントは元の資格を再送信しなければなりません。サーバーはクライアント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資格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)

次の例で資格を説明します。

例 **B–2** AUTH_DES 認証プロトコル

```
/*
 * 資格は 2 種類ある。1 つはクライアントがフルネットワーク
```

例 **B–2** AUTH_DES 認証プロトコル　（続き）

```
 * 名に使用する資格。 もう 1 つはサーバーがクライアントに付けた
 * ニックネーム (単なる符号なし整数) に使用する資格。クライアントは
 * サーバーとの最初のトランザクションでそのフルネームを使用する
 * 必要がある。その際サーバーはクライアントへニックネームを返す。
 * クライアントはそのサーバーとのその後のトランザクションで
 * このニックネームを使用できる。ニックネームは必ず使用しなければ
 * ならないわけではないが、パフォーマンス上の理由から
 * 使用するほうが望ましい。
 */
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];

/*
 * ネットワークユーザー名の長さの最大値
 */
const MAXNETNAMELEN = 255;

/*
 * フルネームにはクライアントのネットワーク名と、 暗号化された会話鍵と
 * ウィンドウが含まれる。ウィンドウは資格の有効期限を示す。
 * ベリファイアのタイムスタンプに記載された時刻にウィンドウの時間を
 * 加えた時刻が過ぎると、サーバーは要求を期限切れとし、許可しない。
 * 要求を再送信しないように、最初のトランザクションを除き、
 * サーバーは前回よりも大きな値のタイムスタンプを要求する。
 * 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を
 * チェックする代わりに、ウィンドウベリファイアがウィンドウより
 * 1 だけ少ないかどうかを調べる。
 */
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」を意味する */

/*
 * 資格はフルネームかニックネームのどちらか。
 */
unionauthdes_credswitch(authdes_namekindadc_namekind){
     case ADN_FULLNAME:
         authdes_fullname adc_fullname;
     case ADN_NICKNAME:
         unsigned int adc_nickname;
};

/*
```

例 **B–2** AUTH_DES 認証プロトコル　　*(続き)*

```
 * タイムスタンプは 1970 年 1 月 1 日深夜 0 時からの秒数を符号化したもの。
 */
struct timestamp {
     unsigned int seconds;      /* 秒 */
     unsigned int useconds;     /* マイクロ秒 */
};

/*
 * ベリファイア：クライアントの種類
  */
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：サーバーの種類
 * サーバーはクライアントが渡したタイムスタンプから 1 秒
 * 少ない（暗号化された）タイムスタンプを返す。サーバーはまた、
 * クライアントに今後のトランザクションで使用する
 * ニックネーム（暗号化されていない）を通知する。
 */
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
```

これらの 2 つの計算は同じ機能を持つことを次に示します。(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、公開鍵の使用を最小限に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 ビットなので、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

AUTH_KERB の S 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。

- KGETKCRED クライアントによって提供されたオーセンティケータを検査するために、サーバー側の RPC が使用する
- KSETKCRED は、一次名、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
- KGETUCRED は UNIX 固有です。一次名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します。

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、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. チケットとセッション鍵を取得するためのローカル kerbd への KSETKCRED
2. フルネーム資格とベリファイアを使用する、サーバーの NFS サービスへの NFSPROC_GETATTR。 サーバーは、呼び出しを受信し、ローカルの kerbd へ KGETKCRED 呼び出しを行な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、次のとおりです。

- ベリファイアが無効な場合は AUTH_BADCRED が返されます。これは、資格内に記述されている *win* とベリファイア内の *win +1* が一致しない場合や、タイムスタンプがウィンドウ範囲外の場合です。
- 再送信が検出されると AUTH_REJECTEDCRED が返されます。
- ベリファイアが誤伝送されるとAUTH_BADVERF が返されます。

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である小さい整数を割り当てます。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。以前の名前が変換できない場合は、ユーザーは匿名と指定されます。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。次の場合について検討します。

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。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 を返す
- サービスチケットの期限が切れると、AUTH_TIMEEXPIRE を返す

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

次の AUTH_KERB の例は AUTH_DES と多くの点で似ていることが次のコーディング例からわかります。両者の違いに注意してください。

例 **B–3** AUTH_DES 認証プロトコル

```
#define AUTH_KERB 4
/*
 * 資格は 2 種類ある。1 つはクライアントが
 *
```

例 **B–3** AUTH_DES 認証プロトコル　(続き)

```
Kerberos チケット (以前に暗号化された) を送信する際の資格で、
 * もう 1 つはサーバーがクライアントに与えた
 * ニックネーム (単純な符号なし整数)。
 * クライアントはサーバーとの最初のトランザクションではそのフルネームを
 * 使用する必要がある。その際、サーバーはクライアントへニックネームを返す。
 * クライアントはそれ以降のサーバーとのすべてのトランザクションで
 * このニックネームを使用できる (チケットが期限切れとなるまで)。
 * ニックネームは使用しなくてもかまいませんが、パフォーマンス上の理由から
 * 使用するほうが望ましい。
 */
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
 * ウィンドウが含まれる。ウィンドウは資格の有効期限。
 * ベリファイアのタイムスタンプに記載された時刻に
 * ウィンドウの時間を加えた時刻が過ぎると、サーバーは要求を期限切れとし、
 * 許可しない。要求を再送信しないように、最初のトランザクションを除き、
 * サーバーは前回よりも大きな値のタイムスタンプを要求する。
 * 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を
 * チェックする代わりに、ウィンドウベリファイアがウィンドウより 1 だけ
 * 少ないかどうかを調べる。
 */
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 */
};
/*
 * 資格はフルネームまたはニックネームのどちらか。
 */
union authkerb_credswitch(authkerb_namekind akc_namekind){
     case AKN_FULLNAME:
         authkerb_fullname akc_fullname;
     case AKN_NICKNAME:
         unsigned long akc_nickname;
};

/*
 * タイムスタンプは 1970 年 1 月 1 日午前 0 時からの秒数を符号化したもの。
 */
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
 */
struct authkerb_verf_clnt {
```

例 **B–3** AUTH_DES 認証プロトコル　　(続き)

```
     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
 * サーバーは、クライアントがサーバーに渡したタイムスタンプと同じ
 * タイムスタンプ（暗号化された）を返す。 サーバーはまた、
 * クライアントへ今後のトランザクションで使用する
 * ニックネーム（暗号化されていない）を通知する。
 */
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 データ型は形式言語で記述する必要があるのと同じように、XDR データ型に対して動作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

次のコーディング例は、シンプルな `ping` プログラムの仕様を示します。

例 **B–4** RPC 言語を使用する `ping` サービス

```
/*
 * シンプルな ping プログラム
 */
program PING_PROG {
     version PING_VERS_PINGBACK {
         void
         PINGPROC_NULL(void) = 0;
         /*
         * 呼び出し側 ping は往復時間をミリ秒単位で返します。
         * 処理が時間切れとなった場合は -1 を返します。
         */
        int
         PINGPROC_PINGBACK(void) = 1;
         /* void - 上記はこの呼び出しへの引数 */
```

例 **B–4** RPC 言語を使用する ping サービス　　*(続き)*

```
     } = 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、処理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。これは古いクライアントプログラムと互換性を持たせるために便利です。

RPCL 構文

RPC 言語 (RPCL) は C に似ています。この節では、例を含め 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
```

このファイルは 6 つの型の定義を認識します。

```
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、次のテーブルを記述する定義が追加されている点を除けば XDR 言語と同一です。

表 **B–2** RPC 言語の定義

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`

RPC 言語では、

- `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ません。
- 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ません。
- 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せん。
- 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ります。
- プログラム、バージョン、および手続きには、符号なし定数のみを割り当てることができます。

RPCL 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
```

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;
```

RPCL 定数

整数の定数を使用する場合は、XDR シンボリック定数を使用できます。たとえば、配列サイズの指定に使用します。

```
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
```

RPCL 型の定義

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;
```

RPCL 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
```

RPCL 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;
```

RPCL 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点に注意してください。次はコンパイルできないプログラムの例です。

```
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];
```

または以下を使用します。

```
struct data {int dummy [10]};
```

RPCL 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囲みます。

```
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。サイズ指定は省略できます。この場合、配列は任意の長さとなります。

```
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 `struct` へコンパイルされます。

```
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。

RPCL ポインタ宣言

XDR でポインタ宣言は、C で行われる場合とまったく同様に行われます。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。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;
```

RPCL 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。この typedef では、アイテムを宣言する際に、 struct coord の代わりに coord を使用できます。

RPCL 共用体

XDR 共用体は識別型の共用体で、C 共用体とは異なります。これらは Pascal の可変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。エラーが発生しなければ、データのブロックを返します。 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;
     };
```

この共用体は次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 の共用体構成要素の名前が、接尾辞 _u を除いて型の名前と同じ名称であることに注意してください。

RPCL 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、引数が渡されないことを意味しています。

次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これらの#define 文になります。

```
#define TIMEPROG 0x20000044
#define TIMEVERS 1
#define TIMEGET 1
#define TIMESET 2
```

RPCL 言語規則の例外

RPC 言語規則には次の例外があります。

RPCL の C 形式モード

C 形式モードでは、rpcgen への void 引数の渡し方が異なります。値が void の場合、引数が渡される必要はありません。

RPCL のブール型

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;
```

RPCL 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。最大サイズは省略できます。この場合、文字列は任意の長さとなります。

次に例を示します。

```
string name<32>;   --> char *name;
string longname<>; --> char *longname;
```

NULL 文字列は渡されません。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) は渡されます。

RPCL の隠されたデータ

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配列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;
```

RPCL の void

void 宣言では、変数を指定できません。void 宣言には、void 以外何も記述しません。 void 宣言は次の 2 箇所でのみ行います。 リモートプロシージャの引数または戻り値として、共用体の定義とプログラムの定義 (引数が渡されないなどで使用される) です。たとえば、引数は渡されません。

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、リモート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 プログラムでは、マシンが異なる場合、アドレスも異なるため、これらのプログラムすべてに直接ブロードキャスト通信を行うことは不可能です。ところが、rpcbind のアドレスはわかっています。 そのため、特定のプログラムへブロードキャスト通信を行うには、クライアントは送信先マシン上にある rpcbind プロセスへメッセージを送信します。 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の応答を取り出すと、それをクライアントに送信します。

次のコーディング例は、RPC 言語の rpcbind プロトコル仕様を示します。

例 **B–5** rpcbind プロトコル仕様 (RPC 言語で記述)

```
/*
 * rpcb_prot.x
 * rpc 言語で記述した RPCBIND プロトコル
 */
/*
 * (プログラム、バージョン、ネットワーク ID) の汎用アドレスへの割り当て
 */
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<>;               /* このサービスの所有者 */ };
/* 割り当てのリスト */
struct rpcblist {
```

例 **B–5** rpcbind プロトコル仕様 (RPC 言語で記述)　　*(続き)*

```
    rpcb rpcb_map;
    struct rpcblist *rpcb_next;
};

/* リモート呼び出しの引数 */
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 */
};

/* リモート呼び出しの戻り値 */
struct rpcb_rmtcallres {
    string addr<>;                  /* リモート汎用アドレス */
    opaque results<>;               /* 戻り値 */
};

/*
 * rpcb_entry には、特定のトランスポート上のサービスの
 * マージされたアドレスと関連付けられた netconfig 情報を含みます。
 * RPCBPROC_GETADDRLIST は rpcb_entry のリストを返します。
 * r_nc_* フィールドで使用できる値については、netconfig.h を
 * 参照してください。
 */
struct rpcb_entry {
    string          r_maddr<>;      /* サービスのマージされたアドレス */
    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するアドレスのリスト */
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 内の手続きに 1 を足した数 */
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 に関するすべての状態のリンクリスト */
struct rpcbs_addrlist {
```

例 **B–5** rpcbind プロトコル仕様 (RPC 言語で記述)　　*(続き)*

```
    rpcprog_t prog;
    rpcvers_t vers;
    int success;
    int failure;
    string netid<>;
    struct rpcbs_addrlist *next;
};

/* rmtcall に関するすべての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;   /* 直接的に呼び出すか、間接的に呼び出すか */
    string netid<>;
    struct rpcbs_rmtcalllist *next;
};

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;
};

/*
 * 監視する rpcbind のバージョン 1 つに対して
 * 1 つの rpcb_stat 構造体が返される。
 */
typedef rpcb_stat rpcb_stat_byvers[RPCBVERS_STAT];
/* rpcbind 手続き */
program RPCBPROG {
    version RPCBVERS {
        void
        RPCBPROC_NULL(void) = 0;

        /*
         * [r_prog, r_vers, r_addr, r_owner,r_netid] の組み合わせを登録。
         * セキュリティー上の理由から、rpcbind サーバーはこの手続きの要求を
         * ループバックトランスポートのみで受け付ける。成功の場合は真を、
         * 失敗の場合は偽を返す。
         */
        bool
        RPCBPROC_SET(rpcb) = 1;

        /*
```

例 **B–5** rpcbind プロトコル仕様 (RPC 言語で記述)　*(続き)*

```
	 * [r_prog, r_vers, r_owner, r_netid] の組み合わせを登録解除。
	 * vers がゼロの場合、すべてのバージョンが登録解除される。
		 * セキュリティー上の理由から、rpcbind サーバーは
		 * この手続きの要求をループバックトランスポート
	 * のみで受け付ける。成功の場合は真を、失敗の場合は偽を返す。
	 */
	bool
	RPCBPROC_UNSET(rpcb) = 2;

	/*
	 * [r_prog, r_vers, r_netid] の組み合わせが登録されている
		 * 汎用アドレスを返す。r_addr を指定すると、
		 * r_addr へマージされた汎用アドレスを返す。
	 * r_owner は無視する。失敗の場合は偽を返す。
	 */
	string
	RPCBPROC_GETADDR(rpcb) = 3;

	/* すべての割り当てのリストを返す。 */

    rpcblist
	RPCBPROC_DUMP(void) = 4;

	/*
	 * リモートマシン上の手続きを呼び出す。
		 * 登録されていない場合はこの手続きは
	 * 何も出力しない。つまり、エラー情報を返さない。
	 */
	rpcb_rmtcallres
	RPCBPROC_CALLIT(rpcb_rmtcallargs) = 5;

	/*
	 * rpcbind サーバーシステム上の時刻を返す。
	 */
	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
```

例 **B–5** rpcbind プロトコル仕様 (RPC 言語で記述)　　*(続き)*

```
RPCBPROC_GETADDR(rpcb) = 3;

rpcblist_ptr
RPCBPROC_DUMP(void) = 4;

/*
 * 注: RPCBPROC_BCAST は CALLIT と同じ機能を持つ。
* 新しい名前の目的は RPCBPROC_BCAST はブロードキャスト RPC に
 * 使用し、RPCBPROC_INDIRECT は間接呼び出しに使用する
         * ことを示すため。
 */
rpcb_rmtcallres
RPCBPROC_BCAST(rpcb_rmtcallargs) = RPCBPROC_CALLIT;

unsigned int
RPCBPROC_GETTIME(void) = 6;

struct netbuf
RPCBPROC_UADDR2TADDR(string) = 7;


string
RPCBPROC_TADDR2UADDR(struct netbuf) = 8;


/*
 * RPCBPROC_GETADDR と同じ機能を持つが、
         * バージョン番号がわからなければ、
 * アドレスが返されない点が異なる。
 */
string
RPCBPROC_GETVERSADDR(rpcb) = 9;


/*
 * リモートマシン上の手続きを呼び出す。登録されていない場合は、
 * この手続きは出力を行う。つまり、エラー情報を返す。
 */
rpcb_rmtcallres
RPCBPROC_INDIRECT(rpcb_rmtcallargs) = 10;

/*
 * RPCBPROC_GETADDR と同じ機能を持つが、
         * この組み合わせ (prog, vers) へ
 * 登録されたアドレスのリストを返す点が異なる。
 */
rpcb_entry_list_ptr
RPCBPROC_GETADDRLIST(rpcb) = 11;

/*
 * rpcbind サーバーの動作に関する統計情報を返す。
 */
rpcb_stat_byvers
```

例 **B–5** rpcbind プロトコル仕様 (RPC 言語で記述)　　*(続き)*

```
        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。この節では、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。プログラムは次を渡します。 プログラム番号 *prog*、バージョン番号 *vers*、ネットワーク ID *netid*、および、プログラムがサービス要求を待機する汎用アドレス *uaddr*。

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 にはできません。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リモートプロシージャを呼び出すことができます。 RPCBPROC_CALLIT は、rpcbind の汎用アドレス経由での任意のリモートプログラムへのブロードキャスト通信をサポートします。

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、リモートプロシージャへの引数を指定します

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、リモートプログラムの汎用アドレスと、リモートプロシージャ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。RPCBPROC_UADDR2TADDR は uaddr2taddr() と同じ機能を持ちます。 マニュアルページ netdir(3NSL) を参照してください。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。RPCBPROC_TADDR2UADDR は taddr2uaddr() と同じ機能を持ちます。マニュアルページ netdir(3NSL) を参照してください。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 は、以前の手続きのほかに、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。しかし、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などのエラーが起こった場合、エラー情報を返す点が異なります。ブロードキャスト RPC でこの手続きを使用しないでください。 この手続きは間接 RPC 呼び出しのみで使用します。
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 サーバーの動作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。

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。

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。この付録では次の内容について説明します。

- 285 ページの「XDR プロトコルの概要」
- 287 ページの「XDR のデータ型宣言」
- 300 ページの「XDR 言語仕様」

XDR プロトコルの概要

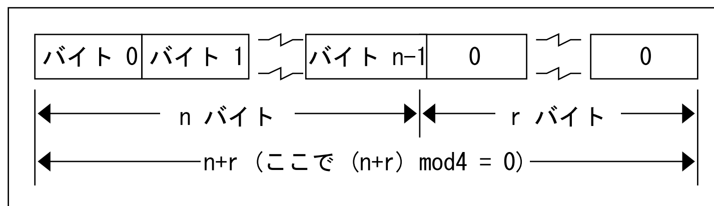
外部データ表現 (external data representation: XDR) は、データの記述と符号化の標準規約です。XDR プロトコルは、異なるコンピュータアーキテクチャ間のデータ伝送に利用できます。これまで、種々のマシン間のデータ通信に使用されてきました。XDR は、ISO の参照モデルのプレゼンテーション層 (第 6 層) に対応するもので、X.409「ISO 抽象構文表記」におおむね従っています。XDR と X.409 との一番大きな違いは、XDR が暗黙的データ型を使用するのに対して、X.409 は明示的データ型を使用する点です。

XDR では、言語を使用してデータ形式を記述しますが、データの記述のためにだけしか使用できません。XDR はプログラミング言語ではありません。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、バイトまたは (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。全項目を表現するには、複数の 4 バイト(32ビット) のデータが必要です。各バイトには `0` から `n -1` の番号が付けられています。 バイトは、バイト `m+1` の前に常にバイト `m` が位置するという関係が保たれる、複数のバイトストリームへと読み書きされます。`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。次の図で、ボックスの間の省略記号 (. . .) の場所には必要に応じて 0 以上の追加バイトが入ります。



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。

基本ブロックサイズが 4 であるということは、コンピュータで標準 XDR が使用できないことを意味する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、300 ページの「XDR 言語仕様」の節を参照してください。

一部のデータ型については、具体的なデータ記述例も示しました。 より詳しいデータ記述例については、303 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

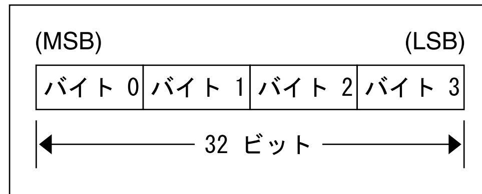
XDR の符号付き整数は、[-2147483648, 2147483647] の範囲の整数が符号化された 32 ビットデータです。整数は 2 の補数で表されます。最上位バイト (MSB) と最下位バイト (LSB) はそれぞれバイト 0 とバイト 3 です。

宣言

整数は次のように宣言します。

```
int identifier;
```

符号付き整数の符号化



符号なし整数

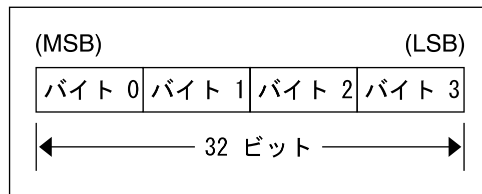
XDR の符号なし整数は、[0, 4294967295] の範囲の正の整数が符号化された 32 ビットデータです。整数は符号なしの 2 進数で表されます。最上位バイト (MSB) と最下位バイト (LSB) はそれぞれバイト 0 とバイト 3 です。

宣言

符号なし整数は次のように宣言します。

`unsigned int` *identifier;*

符号なし整数の符号化



列挙型

列挙型のデータ表現方法は符号付き整数と同じで、整数のサブセットを記述する際に便利です。列挙型の符号化は 287 ページの「符号付き整数の符号化」 に示したものと同じです。

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ないでください。

ブール型

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。ブール型の符号化は 287 ページの「符号付き整数の符号化」 に示したものと同じです。

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;
```

hyper 整数と符号なし hyper 整数

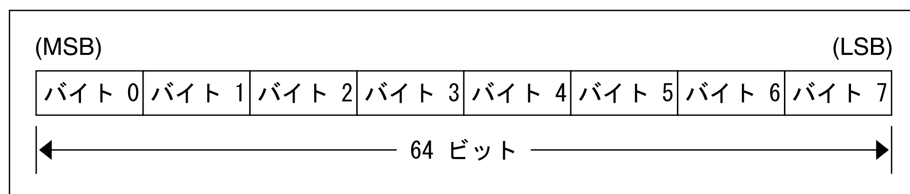
標準規約では 64 ビット (8 バイト) の整数として hyper int と unsigned hyper int を定義しています。その表現方法は明らかに、以前に説明した integer と unsigned integer を拡張したものです。hyper 整数は 2 の補数で表されます。最上位バイト (MSB) と最下位バイト (LSB) はそれぞれバイト 0 とバイト 7 です。

宣言

Hyper 整数は次のように宣言します。

```
hyper int identifier;
unsigned hyper int identifier;
```

Hyper 整数の符号化



浮動小数点

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`: 数値の符号を表します。値 0 は正、1 は負を表します。 このフィールドには 1 ビットが入ります。

`E`: 数値の指数部 (基数は 2) を表します。このフィールドには 8 ビットが入ります。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。このフィールドには 23 ビットが入ります。

したがって、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

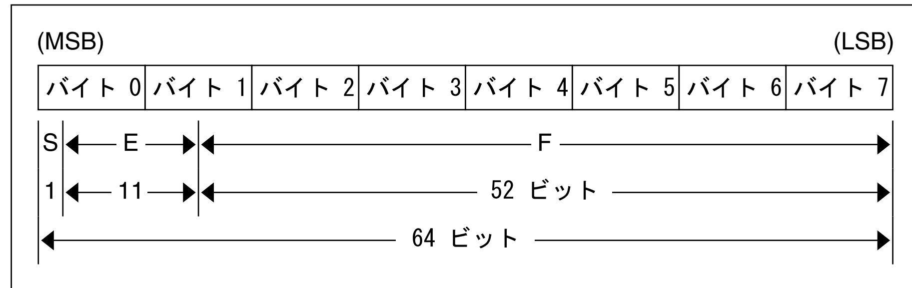
単精度浮動小数点データは次のように宣言します。

`float` *identifier*`;`

倍精度浮動小数点データは次のように宣言します。

`double` *identifier*`;`

倍精度浮動小数点データの符号化



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、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 (オーバーフロー)、正規化されない値 (アンダーフロー) の符号化については、IEEE 標準規約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

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`: 数値の符号を示します。値 0 は正、1 は負を表します。このフィールドには 1 ビットが入ります。

`E`: 数値の指数部 (基数は 2) を表します。このフィールドには 15 ビットが入ります。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。このフィールドには 111 ビットが入ります。

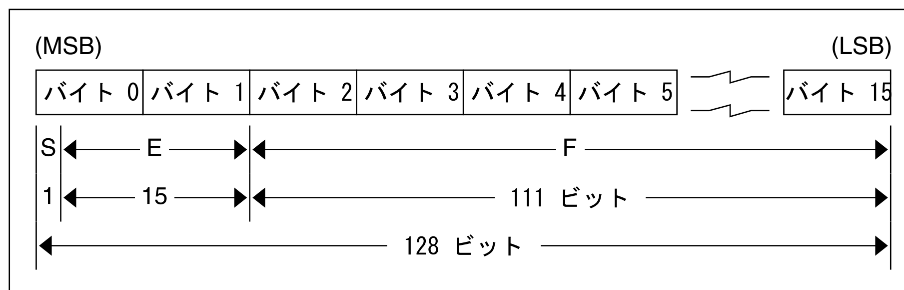
したがって、4 倍精度浮動小数点型の値は次のように記述されます。

```
(-1)**S * 2**(E-Bias) * 1.F
```

宣言

`quadruple` *identifier*;

4 倍精度浮動小数点データの符号化



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、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 (オーバーフロー)、正規化されない値 (アンダーフロー) の符号化については、IEEE 標準規約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

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

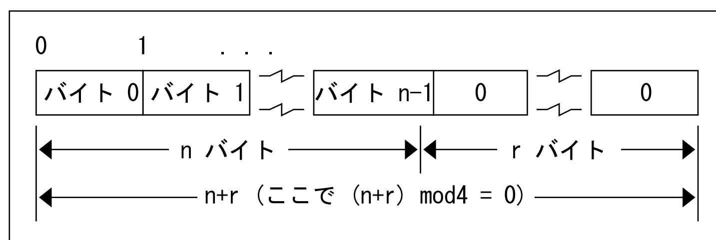
隠されたデータは次のように宣言します。

`opaque` *identifier*`[n];`

この宣言内で、定数 `n` は隠されたデータを格納するため必要となる静的なバイト数です。`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。

固定長の隠されたデータの符号化

`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。



可変長の隠されたデータ

標準規約では、可変長 (カウント付き) の隠されたデータも定義されます。このようなデータは、`n` バイトの任意のバイトシーケンス (バイト番号は `0` から `n-1`) が、次に示すように符号なし整数 `w` に符号化され、その後に、`n` バイトのシーケンスが続くように定義されています。

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` はシーケンス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

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>;`

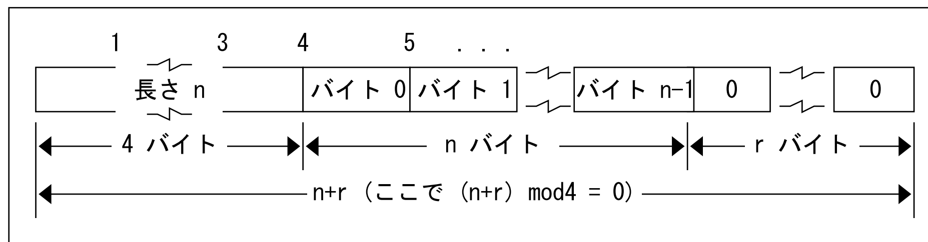
または

`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>;`

可変長の隠されたデータの符号化



指定した最大バイト数以上の長さを符号化しないでください。

カウント付きバイト文字列

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>;

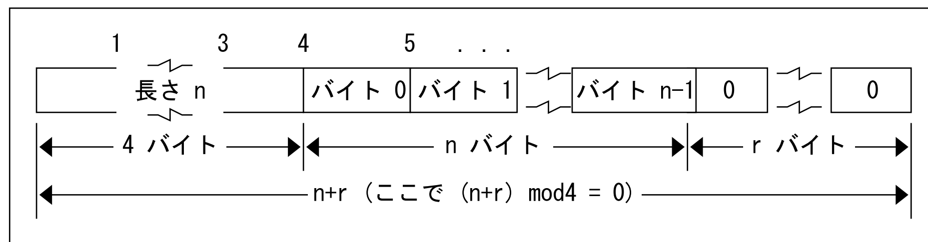
または

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>;

文字列の符号化



指定した最大バイト数以上の長さを符号化しないでください。

固定長配列

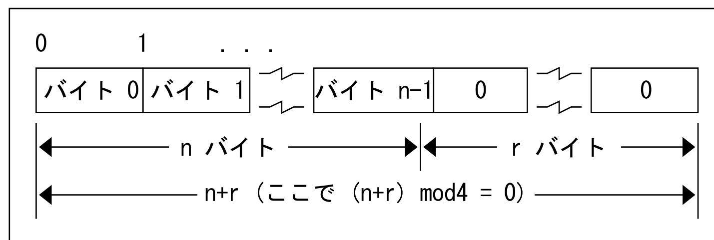
固定長配列の要素番号は 0 〜 n-1 で、個々の配列要素が 0 〜 n-1 の番号順に符号化されます。各配列要素のバイト数は 4 の倍数になっています。全要素が同一のデータ型であっても、要素のサイズが異なることがあります。たとえば、文字列の固定長配列の場合、要素のデータ型はすべて string 型ですが、個々の要素の長さは異なります。

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];`

固定長配列の符号化



可変長配列

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。

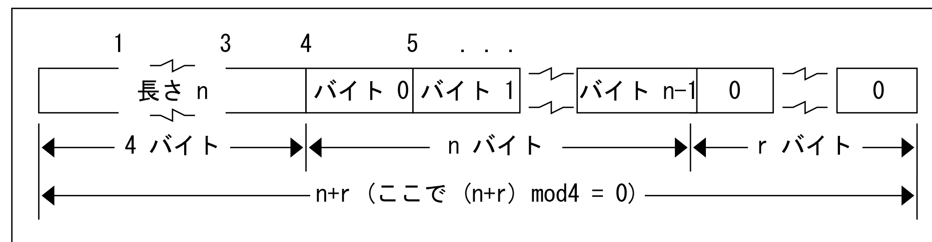
```
type-name identifier<m>;
```

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。

カウント付き配列の符号化



指定した最大バイト数以上の長さを符号化しないでください。

構造体

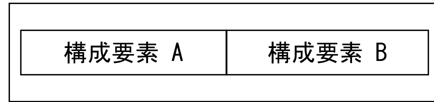
構造体の構成要素は、構造体の宣言で並べた順に符号化されます。各構成要素のサイズはそれぞれ異なる可能性がありますが、各々が 4 の倍数に調整されます。

宣言

構造体は次のように宣言します。

```
struct {
        component-declaration-A;
        component-declaration-B;
        ...
  } identifier;
```

構造体の符号化



識別型の共用体

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

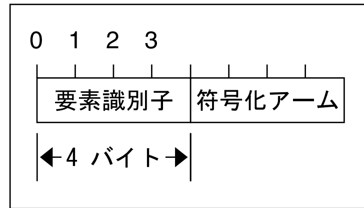
識別型の共用体は次のように宣言します。

```
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。

識別型共用体の符号化



void

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

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。const 型は、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

オプションデータ共用体は、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;
```

この構文は次の共用体と同じです。

```
union switch (bool opted) {
     case TRUE:
     type-name element;
     case FALSE:
     void;
} identifier;
```

このオプションデータ構文は、次の可変長配列宣言とも同じです。これはブール型 opted を配列の長さと解釈できるためです。

```
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

この節では XDR 言語の仕様について説明します。

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。

- 特殊文字として、|、(、)、[、]、* を使用する
- 終端記号は、引用符 (") で囲んだ文字列または文字とする
- 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
- 代替項目は縦棒 (|) で区切って並べる
- 省略可能な項目は角括弧で囲む
- 項目をまとめてグループ化するときは、括弧で囲む
- 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

仕様では次の決まりが使用されます。

- コメントは、/* と */ で囲む
- 空白は項目と項目の区切りに使用し、意味を持たない
- 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

- 定数は 1 つ以上の 10 進数のシーケンスで、次のコーディング例のように、オプションで先頭にマイナス記号 (-) を付けることができる。

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 ")" "{"
```

例 **C–1** XDR 仕様　　*(続き)*

```
    ( "case" value ":" declaration ";" )
    ( "case" value ":" declaration ";" )*
    [ "default" ":" declaration ";" ]
    "}"

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

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ません。

bool	float	switch
cas	hyper	typedef
chas	int	union
const	opaque	unassigned
default	quadruple	void
double	string	
enum	struct	

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す。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ります。

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る必要があります。

同様に、構造体と共用体の宣言の有効範囲内では、変数名は一意にする必要があります。構造体と共用体の宣言が入れ子になっている場合は、新しい有効範囲ができます。

共用体の要素識別子は、整数を表す型にする必要があります。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りません。case で指定する値は、要素識別子の型に応じた値にする必要があります。また、union 宣言の有効範囲内で case の値を 2 回以上指定することはできません。

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。

例 **C–2** XDR ファイルデータ構造体

```
const MAXUSERNAME = 32;/* ユーザー名の長さの最大値 */
const MAXFILELEN = 65535;  /* ファイルの長さの最大値 */
const MAXNAMELEN = 255;    /* ファイル名の長さの最大値 */

/* ファイルのタイプ: */
enum filekind {
     TEXT = 0, /* ASCII データ */
     DATA = 1, /* raw データ */
     EXEC = 2  /* 実行可能ファイル */
 };

/* ファイルタイプ別のファイル情報: */
union filetype switch (filekind kind) {
     case TEXT:
         void;                            /* 追加情報なし */
     case DATA:
         string creator<MAXNAMELEN>;      /* データ作成者 */
     case EXEC:
         string interpreter<MAXNAMELEN>; /*プログラムインタプリタ*/
};

/* 完全なファイル: */
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 */
};
```

linda 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表に示すように符号化されます。

表 **C–1** 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 `Filekind` 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	linda	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。

付録 D

RPC コーディング例

この章では、このマニュアルの rpcgen と RPC に関して説明した章で使用したサンプルプログラムを示します。擬似プログラムであるというような但し書きが特にない限り、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。Sun 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)

例 **D–1** rpcgen プログラム: dir.x

```
/*
 *      dir.x: リモート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 ツールの機能を説明するために使用する
 *       rpcgen ソースモジュールです。ヘッダーファイル (.h) と
 *       付属するデータ構造体の両方を生成するため、
 *      rpcgen -h -T スイッチを付けてコンパイルします。
 */
const MAXNAMELEN = 255;     /*ディレクトリエントリの長さの最大値*/

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*/
/*
 * 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の名前 */
     namelist next;       /* 次のエントリ */
 };

/*
```

例 D–1 rpcgen プログラム: dir.x　(続き)

```
 * READDIR 操作の戻り値:
 *     真に移植性の高いアプリケーションでは、
 *       このサンプルプログラムのように UNIX の
 *      errno を返す方法に頼るのではなく、決められたエラーコードの
 *       リストを使用する。
 *      この例では、成功したリモート呼び出しと
 *       失敗したリモート呼び出しを区別するために共用体が使用されている。
 */
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: 他には何も返さない*/
};

/*
 * ディレクトリプログラムの定義
 */
program DIRPROG {
     version DIRVERS {
         readdir_res
         READDIR(nametype) = 1;
     } = 1;
 } = 0x20000076;
```

例 D–2 リモート dir_proc.c

```
/*
 * dir_proc.c: リモート readdir の実装
 */
#include <rpc/rpc.h>      /* 必ず必要 */
#include <dirent.h>
#include "dir.h"          /* rpcgen が作成 */

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

/* ARGSUSED1*/
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
```

例 **D–2** リモート dir_proc.c　（続き）

```
     */
    dirp = opendir(*dirname);
    if (dirp == (DIR *)NULL) {
         res.errno = errno;
         return (&res);
     }
    /*
     * 以前の戻り値を解放
     */
    xdr_free(xdr_readdir_res, &res);
    /*
     * ディレクトリエントリを収集する。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;
     /* 戻り値を返す */
     res.errno = 0;
     closedir(dirp);
     return (&res);
}
```

例 **D–3** rls.c クライアント

```
/*
 * rls.c: リモートディレクトリリストクライアント
 */

#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;
```

例 D–3 rls.c クライアント　(続き)

```
     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 を呼び出す際に使用する
 * クライアント「ハンドル」を作成。
 */
    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);
     }

/* リモートプロシージャの呼び出しに成功 */

    if (result->errno != 0) {
    /*
     * リモートシステムのエラーが発生。エラーメッセージを出力し、終了
     */
    }
    if (result->errno < sys_nerr)
         fprintf (stderr, "%s : %s\n", dir,
         sys_enlist[result->errno]);
         errno = result->errno;
         perror(dir);
         exit(1);
     }

/* ディレクトリリストの取得に成功。リストを出力 */
    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)

例 **D–4** rpcgen プログラム: time.x

```
/*
 * time.x: リモートタイムプロトコル
 */
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)

例 **D–5** rpcgenプログラム: 2 つの数値の合計を求める

```
/* このプログラムは新しい rpcgen の機能の一部を示すため、
 * 2 つの数を追加する手続きを示します。この場合 add() は 2 つの引数を
 * と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。

例 **D–6** rpcgen プログラム: spray.x

```
/*
 * 著作権 (c) 1987、1991 Sun Microsystems, Inc.
 */
/* spray.x より*/

#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する。
 * 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 */

/*
 * 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 {
        /*
         * 単にデータをスローし、カウンタを増分します。
         * この呼び出しは終了しないため、クライアントは必ず
         * タイムアウトとなります。
         */
        void
```

例 **D–6** rpcgen プログラム: spray.x　(続き)

```
        SPRAYPROC_SPRAY(sprayarr) = 1;

        /*
         * カウンタ値と最終にクリアしたときからの経過時間を取得
         */
        spraycumul
        SPRAYPROC_GET(void) = 2;

        /*
         * カウンタをクリアし、経過時間をリセット
         */
        void
        SPRAYPROC_CLEAR(void) = 3;
    } = 1;
} = 100012;
```

メッセージ表示プログラムとそのリモートバージョン

例 **D–7** printmesg.c

```
/* printmsg.c: コンソールへメッセージを出力 */
#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);
}

/* コンソールへメッセージを出力 */
```

例 D–7 printmesg.c　(続き)

```
/*
 * メッセージを実際に出力したかどうかを示すブール値を返す。
 */
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);
}
```

例 D–8 printmesg.c のリモートバージョン

```
/*  * rprintmsg.c: "printmsg.c” のリモートバージョン */
#include <stdio.h>
#include <rpc/rpc.h> /* 必ず必要 */
#include "msg.h”     /* 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);
    }
    /*
     * コマンド行引数の値を保存
     */
    server = argv[1];
    message = argv[2];
/*
 * コマンド行で指定したサーバー上の MESSAGEPROG を呼び出すために使用する
 * クライアント「ハンドル」を作成
 */
    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。
```

例 **D–8** printmesg.c のリモートバージョン　(続き)

```
         * エラーメッセージを出力し終了。
         */
        clnt_pcreateerror(server);
        exit(1);
    }
    /* サーバー上でリモートプロシージャ "printmessage" を呼び出す */
    result = printmessage_1(&message, cl);
    if (result == (int *)NULL) {
    /*
     * サーバーの呼び出し中にエラーが発生
     * エラーメッセージを出力し終了。
     */
        clnt_perror(cl, server);
        exit(1);
    }
    /* リモートプロシージャの呼び出しに成功 */
    if (*result == 0) {
        /*
         * サーバーはメッセージの出力に失敗。
         * エラーメッセージを出力し終了。
         */
        fprintf(stderr,"%s"
    }
/* サーバーのコンソール上にメッセージを出力 */
    printf("Message delivered to %s!\n", server);
    exit(0);
}
```

例 **D–9** rpcgen プログラム: msg.x

```
/* msg.x: リモートメッセージ出力プロトコル */
program MESSAGEPROG {
     version MESSAGEVERS {
         int PRINTMESSAGE(string) = 1;
     } = 1;
} = 0x20000001;
```

例 **D–10** mesg_proc.c

```
/*
 * msg_proc.c: リモートプロシージャ "printmessage" の実装
 */

#include <stdio.h>
#include <rpc/rpc.h>     /* 必ず必要 */
#include "msg.h”     /* rpcgen が 生成 */

/*
 * "printmessage" のリモートバージョン
 */
/*ARGSUSED1*/
int printmessage_1(msg, req)
```

例 **D–10** mesg_proc.c （続き）

```
    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;
        return (&result);
    }
    fprintf(f, "%sen", *msg);
     fclose(f);
     result = 1;
     return (&result);
}
```

バッチコードの例

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 */
```

例 **D–11** バッチを使用するクライアントプログラムの例　　*(続き)*

```
    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);
}
```

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();                      /* この関数は戻らない */
    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");
                /* 呼び出し側にエラー発生を通知 */
                svcerr_decode(transp);
                break;
            }
```

例 **D–12** バッチを使用するサーバープログラムの例　　*(続き)*

```
            /* 文字列 s をレンダリングするコード */
            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場合にメッセージを出力しない */
                break;
            }
    /* 文字列 s をレンダリングするコードをここに置くが、応答は送信しない */
            break;
        default:
            svcerr_noproc(transp);
            return;
    }
    /* 引数を復号化する間に、割り当てられていた文字列を解放 */
    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。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。

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;
{
    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;
```

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　　*(続き)*

```
    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

Solaris 環境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。

Solaris 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は論理通信チャネルです。特定のネットワークアドレス上で待機することにより、プロセスは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リモート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。このプロトコルはどのトランスポートにも実現できるので、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、リモートプログラムの動的結合を可能にします。

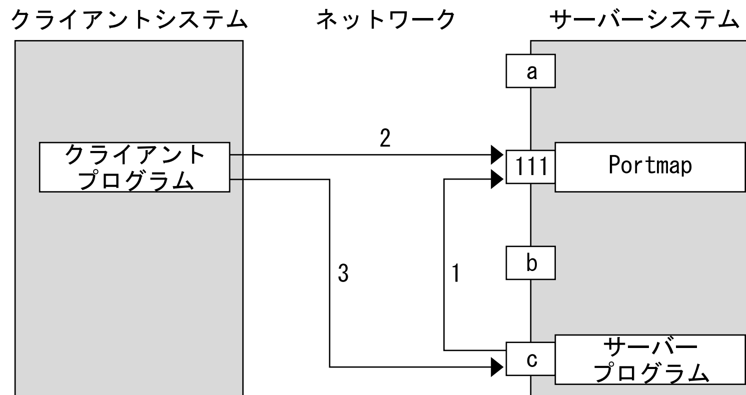


図 **E–1** 典型的な Portmap シーケンス (TCP/IP のみ)

上記の図は、次のプロセスを示しています。

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、リモート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ーを実行しておけば、その他のリモートプログラムのポート番号はポートマッパー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、111 はポートマッパーに割り当てられたポート番号です。

ポートマッパーはブロードキャスト RPC にも役立ちます。特定の RPC プログラムは通常、異なるマシンへは異なるポート番号を割り当てるため、これらのすべてのプログラムへ直接ブロードキャスト通信を行うことはできません。これに対して、ポートマッパー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ーにメッセージを送信します。各ポートマッパーは、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。次のコーディング例で、portmap プロトコル仕様を示します。

例 **E–1** portmap プロトコル仕様 (RPC 言語で記述)

```
const PMAP_PORT = 111;        /* ポートマッパーのポート番号 */
 /*
  * ポート番号のマッピング（プログラム、バージョン、プロトコル）
  */
 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  */
 /*
  * マッピングのリスト
  */
 struct pmaplist {
     pmap map;
     pmaplist *next;
 };
 /*
  * callit への引数
  */
 struct call_args {
     rpcprog_t prog;
 rpcvers_t vers;
 rpcproc_t proc;
     opaque args<>;
 };
 /*
 * callit の戻り値
 */
 struct call_result {
```

例 **E–1** portmap プロトコル仕様 (RPC 言語で記述)　　(続き)

```
     rpcport_t port;
     opaque res<>;
 };
 /*
 * ポートマッパー手続き
 */
 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。次の節ではポートマッパーの各手続きを説明します。

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、マシン上で最初に使用可能になるとローカルのポートマッププログラムに自分自身を登録します。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。PMAPPROC_SET 手続きは、割り当てが正しく設定されれば TRUE を、設定されない場合は FALSE を返します。 rpc_soc(3NSL) 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ープログラムで自身の登録を解除しなければなりません。PMAPPROC_UNSET の引数と戻り値は PMAPPROC_SET の引数と戻り値と同一の意味を持ちます。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(3NSL) の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*、とトランスポートプロトコル番号 *prot* を、PMAPPROC_GETPORT 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(3NSL) のマニュアルページの pmap_getport() 関数も参照してください。

PMAPPROC_DUMP

PMAPPROC_DUMP 手続きはポートマッパーのデータベース内のすべてのエントリを列挙します。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(3NSL) のマニュアルページの pmap_getmaps() 関数も参照してください。

PMAPPROC_CALLIT

PMAPPROC_CALLIT 手続きを使用すると、呼び出し側がリモートプロシージャのポート番号がわからなくても、同一マシン上にあるリモートプロシージャを呼び出すことができます。この手続きは、既知のポートマッパーのポートを使用して、任意のリモートプログラムへのブロードキャスト通信をサポートします。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、リモートプロシージャへの引数です。rpc_soc(3NSL) の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、リモートプログラムのポート番号と、リモートプロシージャからの戻り値を返します。

ポートマッパーがリモートプログラムとの通信に使用できるのは UDP/IP だけです。

付録 F

SAF を使用したポートモニタープログラムの作成

この付録は、ポートモニター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) の下で動作す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。

この付録の内容は、次のとおりです。

- 325 ページの「SAF の概要」
- 326 ページの「SAC の概要」
- 329 ページの「SAF ファイル」
- 330 ページの「SAC とポートモニターのインタフェース」
- 332 ページの「ポートモニターの管理インタフェース」
- 339 ページの「構成ファイルとスクリプト」
- 344 ページの「ポートモニターのサンプルプログラム」
- 349 ページの「論理ダイアグラムとディレクトリ構造」

SAF の概要

サービスアクセス機能 (SAF) はサービスアクセス手順を一般化するため、ローカルシステム上へのログインアクセスとローカルサービスへのネットワークアクセスを同じように管理できます。 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ー、リスナー、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ーを使用できます。

ポートモニター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ーで決まります。したがって、ユーザーは独自のポートモニターを開発してインストールすれば、システムを必要に応じて拡張できます。この拡張性は SAF の重要な特徴です。

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ません。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ーを管理し、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
- ポートモニター
- ポートモニターごとの構成スクリプト (オプション)
- ポートモニター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

サービスアクセスコントローラ (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、ポートモニターを、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 にポートモニターの状態を変更させることができます。また、sacadm を使用して、ポートモニターを 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ー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ーを一意的に識別できるタグと、各ポートモニター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ーの起動
- 管理下のポートモニターをポーリングして、必要に応じて回復手続きを実行

ポートモニターの基本機能

ポートモニターは、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。ポートモニターの主目的は、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。ネットワーク上のアドレス (TSAP または PSAP)、固定された端末回線、着信電話回線などがポートとなります。何をポートとみなすかは、ポートモニター自体が定義します。

ポートモニターはいくつかの基本機能を実行します。これらの機能の一部は SAF への適合が必要となります。 また、ポートモニター自体の条件や設計により指定される機能もあります。

ポートモニターの主な機能を 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

ポートモニターの第 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、ポートモニターの開発者が定義します。ポートモニターは複数のポートを同時に管理できます。

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

ポートモニターの第 2 の機能は、アクティビティ指示に対応す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、ポートモニターがモニタ固有のアクションを取るための指示です。ポートモニター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ーが、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。ポートモニターがどのような指示を検出するかは、ポートモニターの開発者が定義します。
2. もう 1 つのアクティビティは着信サービス要求です。ポートモニター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。

ポートモニターのその他の機能

この項では、ポートモニター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

ポートモニター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、ポートモニターは使用可能と使用不可の 2 つの内部状態を保持しています。ポートモニターの起動時の状態は、sac で設定した環境変数 `ISTATE` の値で決まります。330 ページの「SAC とポートモニターのインタフェース」 を参照してください。

ポートモニター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。ポートモニターが 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ー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。

ポートモニターの有効と無効の切り換えは動的な操作です。この操作で、ポートモニターの内部状態が変わります。ただし、次にポートモニター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。次にポートモニターを起動しても、ポートの状態は残ります。

utmpx エントリの作成

ポートモニターは、起動するサービスの utmpx エントリを作成します。そのとき、エントリのタイプフィールドは USER_PROCESS に設定されます。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。utmpx 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、utmpx エントリは DEAD_PROCESS に設定されなければなりません。

ポートモニターのプロセス ID とファイルのロック

ポートモニターは、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

ポートモニターは、ポートモニターの管理ファイル _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、ポートモニター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、ポートモニターの開発者が定義します。

ポートモニターの終了

ポートモニターはシグナル SIGTERM を受け取ると、手続きに従って自分自身を終了させます。ポートモニターの終了シーケンスを次に示します。

1. ポートモニター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. ポートモニターの状態を有効に切り換える指示はすべて無視します。
3. ポートモニターは、管理下の全ポートの制御を停止します。ポートモニターの以前のインスタンスが停止中は、ポートモニターの新しいインスタンスが、正しく起動される必要があります。
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、ポートモニター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。

ポートモニターの管理ファイル

ポートモニター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ー固有の管理コマンドとを使用して管理します。

ポートモニター `listen` に固有の管理コマンドは `nlsadmin()` です。ポートモニター `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ーを作成するときは、これらのコマンドと同様の機能を持つ、ポートモニター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

ポートモニター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存在し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。

ポートモニターのプライベートファイル

ポートモニターは、ディレクトリ `/var/saf/tag` (tag はポートモニターの名前) 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 とポートモニターのインタフェース

SAC は、起動するポートモニターごとに次の 2 つの環境変数を作成します。

- PMTAG
- ISTATE

  SAC は、この変数を一意的に識別できるポートモニタータグに設定します。ポートモニターは 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、ポートモニターの起動時の内部状態を指定するのに使用します。 ISTATEは、ポートモニターが使用可能状態で起動するときに enabled に、使用不可状態で起動するときに disabled に設定されます。sac はポートモニターの正常なポーリングを定期的に実行します。

SAC は FIFO 経由でポートモニターと通信します。 ポートモニターはカレントディレクトリ内で SAC からのメッセージを受け取るための _pmpipe と、応答メッセージを SAC に送るための ../_sacpipe をオープンします。

メッセージ形式

この節では、SAC からポートモニターに送るメッセージ (SAC メッセージ) と、ポートモニターから SAC に送るメッセージ (ポートモニターメッセージ) 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。例 F–2 を参照してください。

SAC のメッセージ

SAC から送られるメッセージの形式は、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 からポートモニター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 使用可能メッセージ
- SC_DISABLE 使用不可メッセージ
- ポートモニターの _pmtab ファイルを読む必要があることを示す SC_READDB 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。332 ページの「メッセージクラス」 を参照してください。Solaris 環境では常に *sc_size* を 0 に設定する必要があります。

ポートモニターは、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。

ポートモニターメッセージ

ポートモニターから 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;             /* ポートモニターの現在の状態 */
     char pm_maxclass;            /* このポートモニターが解釈できる
                                   最大メッセージクラス */
     char pm_tag[PMTAGSIZE + 1];  /* ポートモニターのタグ */
     int pm_size;                 /* オプションデータ部分のサイズ */
};
```

ポートモニターから 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ーのタグが、pm_state フィールドにはポートモニター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 起動中
- PM_ENABLED 使用可能
- PM_DISABLED 使用不可
- PM_STOPPING 停止中

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 つのフィールドについては、332 ページの「メッセージクラス」 の説明を参照してください。Solaris 環境では、常に pm_maxclass を 1 へ、sc_size を 0 へ設定してください。

ポートモニターから先にメッセージを送ることはありません。ポートモニターは、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。前述のメッセージはすべて class 1 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いません。必要な情報はすべてメッセージヘッダーに入っています。

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 からポートモニター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。どのポートモニターもクラス 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。ポートモニターは、それに対する sac への応答の中で、ポートモニター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、ポートモニターに対して、pm_maxclass の値より大きいクラスのメッセージは送りません。ポートモニター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 環境では、常に pm_maxclass を 1 へ設定してください。

どのポートモニターも、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、このポートモニターはクラス 1、2、3 のメッセージを理解できます。ポートモニター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。ポートモニターからの応答は、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ー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 は pmmsg 構造体の要素です。sc_size は sacmsg 構造体の要素です。 これらの要素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。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 環境では、常に sc_size と pm_size を 0 へ設定してください。

ポートモニター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ー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。SAC 管理ファイル内でエントリを作成することにより、システムへポートモニターを追加できます。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 の管理ファイル内の各エントリには次の表に示す情報が含まれます。

表 F–1 サービスアクセスコントローラ _sactab ファイル

フィールド	説明
PMTAG	個々のポートモニターを一意的に識別できるタグ。ポートモニター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ー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	ポートモニターのタイプ。各ポートモニターには、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ーグループを示す。有効なポートモニタータイプの例として ttymon と listen が挙げられる。タイプ指示子を使用すると、関連ポートモニター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ータグがどのタイプのポートモニター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 は、起動時にポートモニターを有効にしない。 -x は、ポートモニターを起動しない。 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、ポートモニターは起動され有効に設定される。
RCNT	ポートモニター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ーは再起動しない。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、ポートモニター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	ポートモニターを起動するコマンドの文字列。文字列の最初の要素 (コマンド自体) は、フルパスで指定しなければならない。

ポートモニターの管理ファイル _pmtab

各ポートモニターは、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 はポートモニターのタグ) には、ポートモニターのプライベートファイルを入れることができます。

各ポートモニターは自分の管理ファイルを持っています。この管理ファイル内へサービスエントリを追加、削除、変更するには、 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ーはそのファイルを読み直します。ポートモニター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を定義します。

どのタイプのポートモニター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、サービスの起動時に割り当てられる ID、たとえば、root が必ず入っていなければなりません。

サービスタグとポートモニター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。別のポートモニターのサービスを指定する際に同じサービスタグを使用できます。レコードにはポートモニターに固有のデータも入っています。たとえば、ポートモニター 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。ポートモニターのタイプごとに、ポートモニター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。コマンド ttyadm は ttymon の形式を提供し、nlsadmin は listen の形式を提供します。 ユーザーが定義するポートモニターでも、同じような管理コマンドを提供する必要があります。

ポートモニター管理ファイルの各サービスエントリには、次の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ーの中でのみ一意になるようにします。別のポートモニターのサービス (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、ポートモニター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。次の表にサービスエントリを定義します。

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 を指定すると、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 を指定すると、このサービスの utmpx エントリを作成する。デフォルトではサービスの utmpx 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

表 **F–2** SVCTAG サービスエントリ　(続き)

サービスエントリ	説明
PMSPECIFIC	ポートモニター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ー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、utmpx を作成するのが適当でない場合は、ポートモニターが -u フラグを無視します。サービスによっては、utmpx 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 サービスです。

各ポートモニター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 はポートモニター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、ポートモニターの管理ファイルの形式がわかります。このコメント行は、ポートモニター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、つまりポートモニター管理用の管理コマンドです。 sacadm(1M) マニュアルページを参照してください。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ーを管理します。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ー情報を SAC 管理ファイルから取り出して印刷する
- ポートモニターの追加と削除を行う
- ポートモニターの有効と無効を切り換える
- ポートモニター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
- ポートモニター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
- SAC に管理ファイルの再読み取りを要求する

ポートモニターの管理コマンド pmadm

pmadm は、SAF 階層の下位レベル、つまりサービス管理用の管理コマンドです。pmadm(1M) マニュアルページを参照してください。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ー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

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ー (-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、ポートモニター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ー情報を説明しました。新たなポートモニターを追加するときは、_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、ポートモニター側の _pmtab エントリを正しくフォーマットする必要があります。

各ポートモニターには、この 2 つの処理を行うための管理コマンドが必要です。ポートモニターの設計者は、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ー固有部分に必要な情報を、ポートモニターの _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、ポートモニター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。

ポートモニターとサービスのインタフェース

ポートモニター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、最初にポートモニターに子プロセスの 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。ポートモニターは、exec() が起こる前に、ポートモニター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。これはライブラリルーチン doconfig() を呼び出すことによって実行されます。

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。

ポートモニターに必要な条件

ポートモニターを開発するには、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。ポートモニター自体のほかに、管理コマンドを提供する必要があります。

起動時の環境

ポートモニター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
- /var/adm/utmpx に、タイプが LOGIN_PROCESS のエントリを持つ
- ポートモニターの正しい初期状態を示すため、環境変数 ISTATE が enabled または disabled へ設定されている
- 環境変数 PMTAG はポートモニターの割り当てられたタグに設定されている。
- ポートモニター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
- ポートモニターは、/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 はポートモニターのタグ)
- ポートモニターはユーザー ID が 0 (root) で動作している

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ーの主要ファイルが存在します。

表 **F–3** ポートモニターの主要ファイル

ファイル	説明
`_config`	ポートモニターの構成スクリプト。ポートモニター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`	ポートモニターが自身のプロセス ID を書き込むファイル
`_pmtab`	ポートモニターの管理ファイル。このファイルには、ポートモニター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`	ポートモニターが SAC からのメッセージを受け取る際に経由する FIFO
`svctag`	`svctag` というタグを持つサービスの構成スクリプト
`../_sacpipe`	ポートモニターが SAC へメッセージを送信する際に経由する FIFO

ポートモニターの実行すべきタスク

ポートモニターは、ポートモニター固有機能のほかに次のタスクを実行します。

- 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` 受信時に正常終了する。
- SAC とのメッセージ交換プロトコルに従う。

ポートモニター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、ポートモニターは `-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

この節では構成ファイルとスクリプトについて説明します。

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 (ポートモニター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() はエラーで終了します。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() はエラーで終了します。

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、解釈される言語でコマンドスクリプトを記述することにより、システム上のすべてのサービスの環境をカスタマイズするために使用できます。この言語についてはこの章で説明するほか、doconfig(3NSL) マニュアルページにも説明があります。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。

ポートモニターごとの構成ファイル

ポートモニター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、ポートモニターの環境と、ポートモニター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。ポートモニター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。

ポートモニターごとの構成スクリプトは、ポートモニターが起動されたときに翻訳されて実行されます。ポートモニターが 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。

ポートモニター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、ポートモニター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(3NSL) 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ー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。次の 5 つのキーワードが定義されています。assign、push、 pop、runwait、および run です。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 文字を超えることはできません。

assign キーワード

assign キーワードは環境変数の定義に使用します。

`assign` *variable=value*

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 キーワード

push キーワードは *fd* が指定するストリームへ STREAMS モジュールをプッシュするために使用します。 doconfig(3NSL) マニュアルページを参照してください。

`push` *module1*[, *module2*, *module3*, ...]

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 以降も同様)。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 キーワード

pop キーワー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ために使用します。

`pop [module]`

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。引数を指定すると、モジュールは、指定したモジュールがストリームの一番上にくるまで、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 キーワード

runwait キーワードは、コマンドを実行してその終了を待ちます。

`runwait` *command*

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。*command* が見つからないか、実行できなかったとき、または *command* は存在してもステータスが 0 以外のときは、runwait コマンドは実行エラーとなります。

runキーワード

run キーワードは、runwait と同じですが、*command* の終了を待たない点が異なります。

run *command*

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やこれらのコマンドと同様、実行のための別プロセスを生成しません。sh(1) の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 とポートモニター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ー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

システムごとの構成スクリプトは sacadm コマンドで管理します。

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。

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
```

ポートモニターごとの構成スクリプト

モジュールごとの構成スクリプトは sacadm コマンドで管理します。

```
sacadm -g -p pmtag [ -z script ]
```

-g オプションは、ポートモニター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、script に指定したファイルがポートモニター pmtag に対するポートモニター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 -p オプションと -z オプションを組み合わせて指定すると、*pmtag* に指定したポートモニターのポートモニターごとの構成スクリプトと同じファイル *script* がインストールされます。あるいは、/etc/saf/pmtag/_config が存在する場合、*script* の内容に _config が置き換わります。-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。

_config ファイル内では、コマンド /usr/bin/daemon は STREAMS のマルチプレクサーを構築してまとめるデーモンプロセスを起動するとみなされます。この構成スクリプトをインストールすると、ポートモニターが必要とするコマンドを、ポートモニター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

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、ポートモニターによって解釈されて実行されます。

```
pmadm -g -p pmtag -s svctag [ -z script ]
 pmadm -g -s svctag -t type -z script
```

注 – SAC は自分自身の構成ファイル (_sysconfig) と、ポートモニター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、ポートモニターが実行します。

-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、ポートモニター (*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、ポートモニター (*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、ポートモニタータイプ (*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。

次に示すサービスごとの構成スクリプトでは、2 つの設定を制御します。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
```

ポートモニターのサンプルプログラム

次のコーディング例は、SAC からのメッセージに応答するだけの NULL ポートモニターです。

例 **F–1** ポートモニターのサンプル

```
# 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]; /* ポートモニターのタグ */
FILE *Fp; /* ログファイルのファイルポインタ */
FILE *Tfp; /* pid ファイルのファイルポインタ */
char State; /* ポートモニターの現在の状態*/

main(argc, argv)
    int argc;
    char *argv[];
{
    char *istate;
    strcpy(Tag, getenv("PMTAG"));
/*
 * ポートモニターのプライベートディレクトリ内でログファイルを開く。
 */
    sprintf(Scratch, "/var/saf/%s/log", Tag);
    Fp = fopen(Scratch, "a+");
    if (Fp == (FILE *)NULL)
        exit(1);
    log(Fp, "starting");
```

例 **F–1** ポートモニターのサンプル　　(続き)

```
/*
 * 初期状態 (「enabled」または「disabled」のどちらか) を取得し、
 * それに従って State を設定する。
 */
    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、ロックして、ポートモニターが
 * アクティブであることを示す。
 */
    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。この関数は戻らない。
 */
    handlepoll();
    pause();
    fclose(Tfp);
    fclose(Fp);
}

handlepoll()
{
    int pfd; /* 着信パイプのファイル記述子 */
    int sfd; /* 送信パイプのファイル記述子 */
    struct sacmsg sacmsg; /* 着信メッセージ */
    struct pmmsg pmmsg; /* 送信メッセージ */
/*
```

例 **F–1** ポートモニターのサンプル　　(続き)

```
 * SAC からの着信メッセージ用にパイプをオープン。
 */
    pfd = open("_pmpipe", O_RDONLY|O_NONBLOCK);
    if (pfd < 0) {
        log(Fp, "_pmpipe open failed");
        exit(1);
    }
/*
 * SAC への送信メッセージ用にパイプをオープン。
 */
    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。クラス 1 のメッセージのみをサポートする。 */
    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る。
 */
     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それに応じて応答する。
 */
         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:
                 /*内部の状態が次のように変化することに注意。*/
                 log(Fp, "Got SC_ENABLE message");
                 pmmsg.pm_type = PM_STATUS;
                 State = PM_ENABLED;
                 pmmsg.pm_state = State;
                 break;
             case SC_DISABLE:
                 /*内部の状態が次のように変化することに注意。*/
                 log(Fp, "Got SC_DISABLE message");
                 pmmsg.pm_type = PM_STATUS;
                 State = PM_DISABLED;
                 pmmsg.pm_state = State;
                 break;
             case SC_READDB:
                 /*
```

例 **F–1** ポートモニターのサンプル　　(続き)

```
                 * 正常に機能するポートモニターは、
                                 * ここで、_pmtab を読み取り、
                 * 正しいアクションを行う。
                 */
                 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;
         }
/*
 * ポーリングに対して現在の状態を示す応答を送信。
 */
         if (write(sfd, &pmmsg, sizeof(pmmsg)) != sizeof(pmmsg))
             log(Fp, "sanity response failed");
     }
}
/*
 * 一般的なログ関数
 */
log(fp, msg)
    FILE *fp;
    char *msg;
{
    fprintf(fp, "%d; %s\n", getpid(), msg);
    fflush(fp);
}
```

次のコーディング例は sac.h ヘッダーファイルを示します。

例 **F–2** sac.h ヘッダーファイル

```
/* utmpx ID のバイト長 */
# define IDLEN 4
/* utmpx ID で使用できるワイルド文字 */
# define SC_WILDC 0xff
/* ポートモニターのタグの最大バイト長 */
# define PMTAGSIZE 14
/*
 * doconfig() の rflag 値
 */
/* assign を許可しない */
# define NOASSIGN 0x1
/* run または runwait を許可しない */
# define NORUN 0x2
/*
```

例 **F–2** sac.h ヘッダーファイル　　*(続き)*

```
 * SAC へのメッセージ (ヘッダーのみ) このヘッダーは常に固定。
 * size フィールド (pm_size) はヘッダーの後ろに続くメッセージの
 * データ部分のサイズを定義する。このオプションデータ部分の形式は、
 * メッセージのタイプ (pm_type) によって厳密に定義される。
 */
struct pmmsg {
    char pm_type;               /* メッセージのタイプ */
    unchar pm_state;            /* ポートモニターの現在の状態 */
    char pm_maxclass;
             /* このポートモニターが理解できる最大メッセージクラス */
    char pm_tag[PMTAGSIZE + 1]; /* ポートモニターのタグ */
    int pm_size;                /* オプションデータ部分のサイズ */
};
/*
 * pm_type の値
 */
# define PM_STATUS 1 /* 状態応答 */
# define PM_UNKNOWN 2 /* 未知のメッセージを受信 */
/*
 * pm_state 値
 */
/*
 * クラス 1 の応答
 */
# define PM_STARTING 1   /* 起動状態のポートモニター */
# define PM_ENABLED 2    /* 使用可能状態のポートモニター */
# define PM_DISABLED 3   /* 使用不可状態のポートモニター */
# define PM_STOPPING 4   /* 停止状態のポートモニター */
/*
 * ポートモニターへのメッセージ
 */
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 */
};
/*
 * 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。
* 拡張性を持たせるため、コマンドは各クラスに分類される。
* 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。
* どのクラスもヘッダーは同じ。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る。
* オプションデータ部の形式は、コマンドで自動的に決まる。
* 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。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る。
* 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
* 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。
* (すなわち、クラス 1 のコマンドには必ずクラス 1 で応答する。)
 */
/*
 * クラス 1 コマンド(現在はクラス 1 のコマンドのみが存在)
```

例 **F–2** `sac.h` ヘッダーファイル　*(続き)*

```
 */
# define SC_STATUS 1    /* ステータス要求 *
# define SC_ENABLE 2    /* 使用可能にする要求 */
# define SC_DISABLE 3   /* 使用不可にする要求 */
# define SC_READDB 4    /* pmtab 読み取り要求 */
/*
 * Saferrno の errno 値。pmadm と sacadm の両方が Saferrno を使用し、
 * これらの値はこの両者で共有される点に注意してください。
 */
# define E_BADARGS 1   /* 引数が無効か、コマンド行の形式が不正。 */
# define E_NOPRIV 2    /* ユーザーに操作特権がない。 */
# 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       /* エントリがすでに存在する */
# define E_PMRUN 7     /* ポートモニターが動作中。 */
# define E_PMNOTRUN 8  /* ポートモニターが動作していない。 */
# 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-1 に 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ーをどのように生成するのかを示しています。これは、複数のモニターが並行して動作できるようにするため、異なるプロトコルを同時に提供することを意味します。

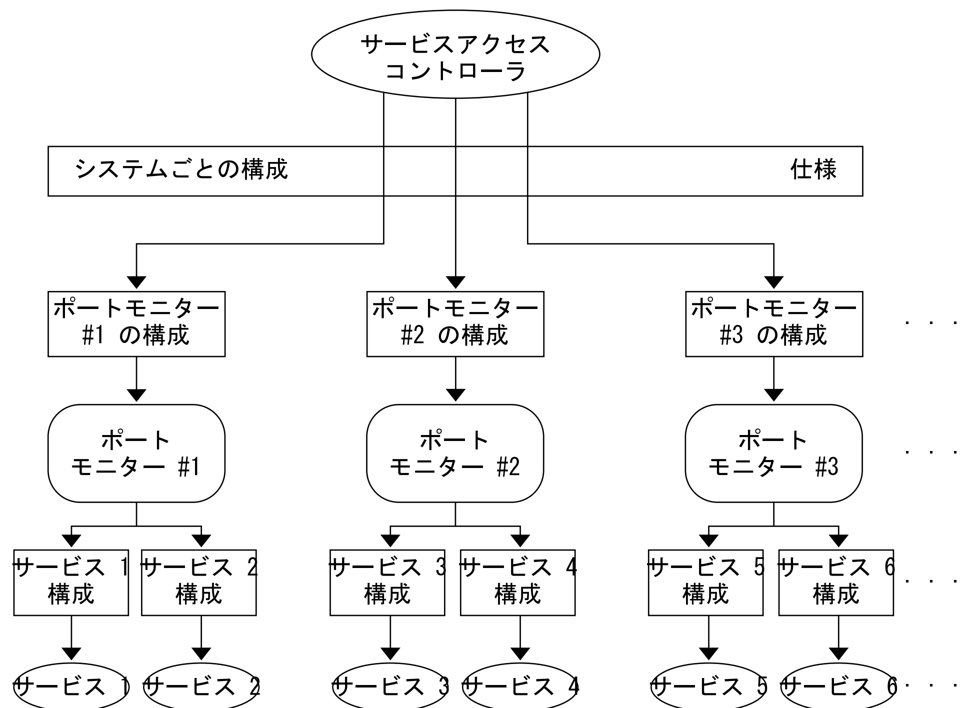


図 **F–1** SAF 論理のフレームワーク

次の図は 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。 図の後に、ディレクトリ構造内のファイルおよびディレクトリについて説明します。

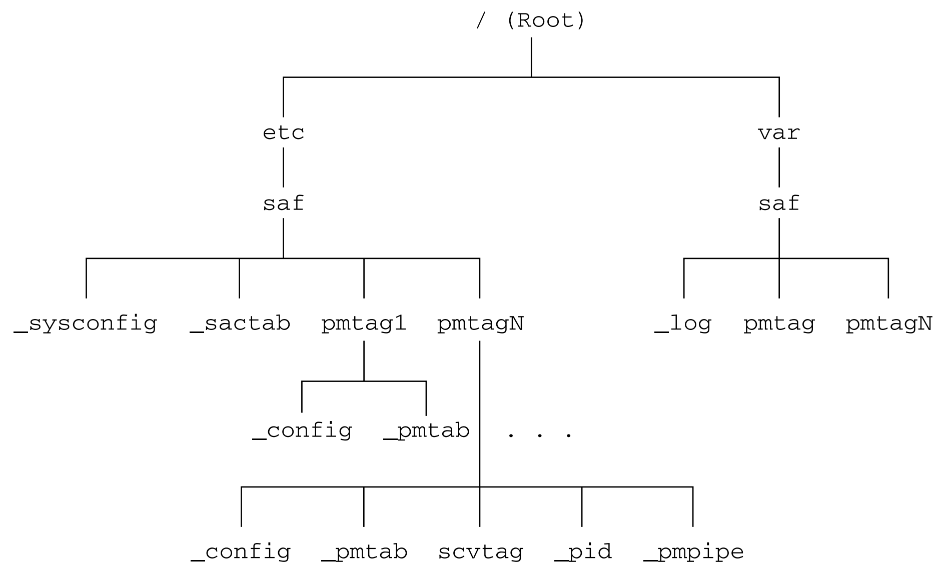


図 **F–2** SAF ディレクトリ構造

SAF ディレクトリ構造内のスクリプトとファイルは次のとおりです。

- /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ーの情報が入っています。
- /etc/saf/pmtag - ポートモニター *pmtag* のホームディレクトリ
- /etc/saf/pmtag/_config - ポートモニター *pmtag* のポートモニターごとの構成スクリプト
- /etc/saf/pmtag/_pmtab - ポートモニター *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 - *pmtag* 経由で使用可能な、サービス *svctag* のサービスごとの構成スクリプトが置かれます。
- /etc/saf/pmtag/_pid - このファイルにポートモニターはそのプロセス ID とカレントディレクトリを書き込んだり、アドバイサリロックを設定します。
- /etc/saf/pmtag/_pmpipe - ポートモニターが SAC と ../_sacpipe からのメッセージを受け取ったり、SAC へ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 - ポートモニター *pmtag* が作成するファイル、たとえばそのログファイルなどのためのディレクトリ

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。

MT 安全
: スレッド化された環境で呼び出すことのできるインタフェースの特徴。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できる。

MT ホット
: ライブラリや呼び出しが自動的にスレッドを作成するインタフェースの特徴。

ping
: リモートシステム上のアクティビティーを確認するサービス。コンピュータはホストへ小さなプログラムを送信し、戻ってくるパス上に時間をメモします。

RPC/XDR
: マシンに依存しないデータ構造体の標準規格。「RPC 言語 (RPCL)」を参照。

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 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 ライブラリ
: コンパイル時にリンクエディタへ指定されるネットワークサービスライブラリ。libnsl。RPC パッケージともいう。

TI-RPC
: トランスポートに依存しない RPC。SunOS 5.0 およびその互換バージョンでサポートされる RPC のバージョン。

TS-RPC
: トランスポート固有の RPC。SunOS 4.0 およびその互換バージョンでサポートされる RPC のバージョン。TS-RPC は SunOS 5.0 およびその互換バージョンでもサポートされます。

XDR 言語
: データ記述言語およびデータ表現プロトコル 。

仮想回路トランスポート	トランスミッションコントロールプロトコル (TCP) によって促進されるプロセス間の明確な接続。アプリケーション同士が物理的回線があるかのように互いに通信できるようにする仮想回路。
クライアント	コンピュータサーバーのリソースへリモートアクセスするプロセス。
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。
サーバー	リソースを管理し、クライアントへサービスを提供するネットワークデバイス。
シリアライズ	データをマシン固有の表現形式から XDR 形式へ変換すること。
接続型トランスポート	接続の確立、データ転送、接続解除の定義された 3 段階を経て通信を行うインターコネクトモデルの特徴。「ストリームトランスポート」を参照。
データグラムトランスポート	メッセージとインターネットソースおよびそれに関連付けられた宛先アドレス。データグラムトランスポートは接続型トランスポートに比べオーバーヘッドが小さくなりますが、信頼性も低くなると考えられます。データ転送はバッファサイズにより制限されます。
デシリアライズ	データを XDR 形式からマシン固有の表現形式へ変換すること。
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。
ネットワーククライアント	サービスに対するリモートプロシージャ呼び出しを行うプロセス。
ネットワークサーバー	リソースを管理してクライアントへサービスを提供する、ネットワークデバイス。
ネットワークサービス	1 つ以上のリモートサービスプログラムの集合。
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。
汎用アドレス	TCP/IP などのネットワークのタイプの 16 進アドレス。ポートモニターを構成し、ネットワーク上の印刷クライアントからの出力要求を検査します。
非接続トランスポート	接続を確立せずに通信を行うインターコネクトモデルの特徴。「データグラムトランスポート」を参照。
ホスト	リモート地にあるコンピュータやワークステーションがアクセスするコンピュータシステム。データは通常ホストに置かれますが、ネットワーク内ではリモート地にホストを置き、ネットワークへ情報を提供することもできます。
リモートプログラム	1 つ以上のリモートプロシージャを実現するプログラム。

索引

数字・記号

32–ビットシステム, 75
64 ビットシステム, 77
1 suffix, 48
_1suffix (接尾辞), 82
4 倍精度浮動小数点, XDR 言語, 292

A

add.x source file, 58
`add.x` ソースファイル, 59, 61, 65, 67, 69
ADDPROG プログラム, 309
ah_cred フィールド, 101
ah_key フィールド, 123
ah_verf フィールド, 101
ANSI C 標準
  rpcgen ツール, 44, 58, 69
AUTH_BADCRED エラー, 265
AUTH_DES 認証, 122, 258, 263
  Diffie-Hellman 暗号, 262, 263
  Diffie-Hellman 暗号化, 123
  XDR 言語のプロトコル, 260, 263
  エラー, 260
  会話鍵, 259, 261, 263
  共通鍵, 262, 263
  サーバー, 123, 124
  資格, 122, 261
  時刻の同期, 260
  時刻の同期化, 123
  ニックネーム, 260, 261, 262
  ハンドル, 122, 123
  ベリファイア, 259, 260, 262
AUTH_KERB 認証, 124
  NFS, 264, 265
  XDR 言語でのプロトコル, 265
  XDR 言語のプロトコル, 267
  暗号化, 124, 125
  エラー, 265
  資格, 124, 125, 265, 266
  時刻の同期, 266, 267
  時刻の同期化, 124
  ニックネーム, 125, 265, 266, 267
  ベリファイア, 124, 125, 265, 266, 267
AUTH_NONE 認証, 256
AUTH_REJECTEDVERF エラー, 265
AUTH_SHORT ベリファイア, 257, 258
AUTH_SYS 認証, 257, 258
AUTH_TIMEEXPIRE エラー, 265
AUTH_TOOWEAK エラー, 265
AUTH_UNIX (AUTH_SYS) 認証, 258

B

bcast.c プログラム, 113, 114
broacast RPC, 113

C

C
  rpcgen ツール, 69
    ANSI C 準拠, 44, 58, 69
    C 形式モード, 44, 57, 59
    前処理命令, 56

rpcgen ツール (続き)
rpcgen ツール および
C 形式モード, 62
rpcgen ツールおよび
C 形式モード, 275
前処理命令, 57, 71
CBC (暗号文ブロックチェイン) モード, 124
cd コマンド, 342
circuit_v transport タイプ, 38
cl_auth フィールド, 101
clnt.c 接尾辞, 50
clnt_create ルーチン, コード例, 50
clnt_destroy ルーチン, 記述済み, 49
clnt_perror ルーチン, 78
clnt_sperror ルーチン, 78
config ファイル, 338
config ファイル, 343
_config ファイル, 339, 340
cpp 命令, rpcgen ツール, 57
C 形式モード
rpcgen ツール, 44, 57, 59, 62, 275

D

datagram_n トランスポートタイプ, 38
datagram_vトランスポートタイプ, 38
DES 暗号化, 123, 259
Diffie-Hellman 暗号, 259, 262, 263
Diffie-Hellman 暗号化, 123
dir.x プログラム, 53
dir.x プログラム, 51, 305, 306
dir_proc.c ルーチン, 54
dir_proc.c ルーチン, 53
doconfig 関数, 328, 337, 339

E

ECB (電子コードブック) モード, 124, 125
errors
認証
AUTH_KERB, 265
/etc/gss/mech, 137
/etc/gss/qop, 137
/etc/inet/inetd.conf ファイル, 138
/etc/netconfig データベース, 37, 148
/etc/netconfig データベース, 71
/etc/rpc データベース, 34
/etc/saf//_config ファイル, 338, 339, 340, 343
/etc/saf/_pid ファイル, 328, 338, 345
/etc/saf//_pmpipe ファイル, 330, 338
/etc/saf//_pmtab ファイル, 329, 333, 335, 338
/etc/saf/_sactab ファイル, 333
/etc/saf//svctag ファイル, 335, 338, 339
/etc/saf/_sysconfig ファイル, 339, 340, 342, 343

F

free ルーチン, 56

I

inetd ポートモニター, 138
inetd ポートモニター, rpcgen ツール, 51
inetd ポートモニター
rpcgen ツール, 72
RPC サービス, 139
ISTATE 環境変数, 327, 330, 337
IXDR_GET_LONG, 77
IXDR_PUT_LONG, 77
.i接尾辞, 74

K

kerbd デーモン, 263, 264
Kerberos 認証, 21
KGETKCRED 手続き, 263, 264
KGETUCRED 手順, 265
KGETUCRED 手続き, 263, 265
KSETKCRED 手続き, 263, 264

L

libc ライブラリ, 147, 148
libnsl ライブラリ, 50, 52, 147, 148
librpcsvc ライブラリ, 80
lib ライブラリ, 55
limits, スレッドの最大数, 160
listen ポートモニター, 138

inetd ポートモニター (続き)
　rpcgen ツール，　51, 72
　使用，　139, 140
lthread ライブラリ，　156

M

makefile テンプレート
　rpcgen ツール，　44, 58
msg.h ヘッダーファイル，　50
`msg.x` プログラム，　62
msg_clnt.c ルーチン，　50
msg_svc.c プログラム，　50
msg_svc.c ルーチン，　50
MT RPC プログラミング
　マルチスレッド RPC プログラミングを参照
MT 自動モード，　157, 159, 162
　rpcgen ツール，　44, 58, 68
　コード例，　160, 162
　サービストランスポートハンドル，　159
MT 対応コード，　21
　rpcgen ツール，　44, 58, 62
　rpcgen ツールおよび，　67
　クライアント，　44, 62, 64, 65, 67
　サーバー，　43, 44, 64, 65, 67, 158
MT (マルチスレッド) 対応コード
　クライアント，　79
　サーバー，　79
MT ユーザーモード，　21, 157, 159, 162
mutex ロック, マルチスレッドモードおよび,
　　158

N

netconfig データベース，　37, 71, 148
NETPATH 環境変数，　37, 71, 89
nettype パラメータ，　38
Network Information Services Plus
　NIS+ (Network Information Services Plus)を
　　参照
Newstyle (C 形式) モード
　rpcgen ツール，　44, 57, 59, 62
NFS
　Kerberos 認証，　264, 265
NFSPROC_GETATTR 手続き，　264
NFSPROC_STATVFS 手続き，　265
NIS+，　201, 216
　アプリケーションプログラミングインタ
　　フェース (API)，　201, 205
　エラーメッセージ表示関数，　201, 204
　オブジェクト，　201
　　サンプルプログラム，　206, 216
　　操作関数，　201
　概要，　29
　グループ，　200
　　サンプルプログラム，　207, 208, 213, 216
NIS+ (
　グループ
　　操作関数，　201
NIS+
　コンパイル，　206
　サーバー，　198
　　関数，　201, 203
　　サンプルプログラム，　216
　サポートされていないマクロ，　206
　サンプルプログラム，　205, 216
　セキュリティー，　199, 200
　その他の関数，　201, 204, 205
　データベースアクセス関数，　201, 203, 204
　テーブル，　198, 199, 200
　　アクセス関数，　201, 202
　　サンプルプログラム，　208, 210
　同期，　204
　ドメイン，　198, 200
　　関数，　203
　トランザクションログ関数，　201, 204
　名前空間管理コマンド，　200
　ネームサービススイッチ，　200
　ローカル名関数，　201, 202, 204, 205
NIS+ (Network Information Services Plus)，　21
NIS+ Plus
　グループ
　　操作関数，　203
NIS+ 情報, オブジェクト，　200
nlsadmin コマンド，　329
NULL トランスポートタイプ，　38
NULL 引数，　82
NULL ポインタ，　237
NULL 文字列，　275

O

ONC+ 概要，　27, 29

P

pid ファイル， 328, 338
_pid ファイル， 345
ping プログラム， 267, 268
pmadm コマンド， 140, 329, 334, 336, 343, 344
PMAPPROC_CALLIT 手続き， 323
PMAPPROC_DUMP 手続き， 323
PMAPPROC_GETPORT 手続き， 323
PMAPPROC_NULL 手続き， 322
PMAPPROC_SET 手続き， 322
PMAPPROC_UNSET 手続き， 323
pm_maxclass フィールド， 332
pmmsg 構造体， 331
_pmpipe ファイル， 330, 338
pm_size フィールド， 331, 332
_pmtab， 334
_pmtab ファイル， 329, 333, 335, 338
PMTAG 環境変数， 330, 337
pop 構成スクリプトキーワード， 341
printmsg.c プログラム
  シングルプロセスバージョン， 311, 312
printmsg.c プログラム, 単一プロセスバージョン， 46
printmsg.c プログラム
  単一プロセスバージョン， 45
printmsg.c プログラム, リモートバージョン， 51
printmsg.c プログラム
  リモートバージョン， 46, 312, 314
PROGVERS_ORIG プログラム名， 140
PROGVERS プログラム名， 140

R

raw RPC, 下位レベルを使ったプログラムのテスト， 105
READDIR 手続き， 51, 56, 305, 308
rls.c ルーチン， 56
RPC， 21
  iインタフェースルーチン， 87
  アドレス登録， 21
  アドレス変換， 39
  アドレスルックアップサービス， 36, 40, 41
  一時的 RPC プログラム番号， 251
  一時的な RPC プログラム番号， 144
  インタフェースルーチン， 34, 35, 79, 80, 88
RPC (続き)
  TI-RPC インタフェースルーチンを参照
    エキスパートレベル， 94, 98
    下位レベルのデータ構造， 100
    キャッシュサーバー， 100
    単純， 87
    中間レベル， 36, 92
    標準， 87
    ボトムレベルl， 99
  エラー， 78, 250
  間接， 280
  情報の取り出し， 41, 143
  トランスポート選択， 38
  トランスポートタイプ， 38
  名前からアドレスへの変換， 39, 40
  ネットワーク選択， 37
  バッチ， 115, 118, 252, 314, 317
  非同期モード， 111, 113
  標準， 32, 256
  複数のクライアントバージョン， 143
  複数のサーバーバージョン， 140
  ブロードキャスト
  ブロードキャスト RPCを参照
  プロトコル
  TI-RPC プロトコルを参照
  ポートモニターの使用， 140
  ポーリングルーチン， 111, 113
  リモートプロシージャの識別， 34
  リモートプロシージャの特定， 249, 250
RPC (, リモートプロシージャを特定， 252
RPC
  レコードマーク標準， 256
RPC_AUTHERROR エラー， 260
rpcbind デーモン, アドレスの登録， 40
rpcbind ルーチン， 21
rpcbind ルーチン， 21
  時刻サービス， 259
RPCBPROC_CALLIT 手続き， 41
RPCBPROC_GETTIME 手続き， 259
RPC_CLNT 前処理命令， 56
rpc_createerr グローバル変数， 89
rpcgen tool, MT (マルチスレッド) 自動モード， 159
rpcgen ツール， 21, 43, 75, 78, 309
  cpp 命令， 57
  C および， 69
    ANSI C 準拠， 44, 58, 69
    C 形式モード， 44, 57, 59, 62, 275

C および (続き)
前処理命令, 56, 57, 71
MT (マルチスレッド) 自動モード, 44, 58, 68, 162
MT (マルチスレッド) 対応コード, 44, 58, 62, 67
Newstyle (C 形式) モード, 44, 57, 59, 62
TI-RPC 対 TS-RPC, 147
TI-RPC 対 TS-RPC ライブラリ選択, 44, 58
TI-RPC または TS-RPC のライブラリ選択, 68
xdr_inline() カウンタ, 69
xdr_inline カウント, 58
XDR ルーチン生成, 217
XDR ルーチンの生成, 51, 56
印刷メッセージプログラム, 51
オプションの出力, 43
クライアント側のスタブルーチン
クライアント側のスタブルーチンを参照
クライアントハンド
クライアントハンドルを参照
クライアントプログラム
クライアントプログラムを参照
コマンド行でステートメントを定義する, 71
コンパイルモード, 44, 59, 62
サーバー側のスタブルーチン
サーバー側のスタブルーチンを参照
サーバーハンドル
サーバーハンドルを参照
サーバープログラム
サーバープログラムを参照
時刻サーバープログラム, 56, 68, 309
スタブルーチン
クライアント側のスタブルーチンを参照
スプレイパケットプログラム, 310, 311
タイムアウトの変更, 73
チュートリアル, 45, 58
ディスパッチテーブル, 74, 75
ディレクトリリストプログラム, 51, 56, 305, 308
デバッギング, 77
デバッグ, 70, 71, 78
デフォルト
C プリプロセッサ, 57
MT 対応, 44, 62
クライアント側のタイムアウト期間, 73
コンパイル モード, 44
デフォルト (続き)
サーバー終了間隔, 72
出力, 43
引数引き渡しモード, 59, 61
ライブラリ選択, 68
テンプレート, 44, 57, 58, 60, 61
認証, 70, 73, 122, 125
ネットワークタイプ/トランスポート選択, 70
バッチコードの例, 314, 317
ハンドコーディング対, 83
ハンドル
クライアントハンドルを参照
引数, 47, 48, 59, 62, 82, 84, 87, 275
複雑なデータ構造, 51
複雑なデータ構造の引渡し, 56
フラグ, 58
-A (MT 自動モード), 58, 68
-a (テンプレート), 57, 58
-b (TS-RPC ライブラリ), 58, 68
-i (xdr_inline() カウント), 69
-M (MT 対応コード), 58, 62
-N (C 形式モード), 57, 59
-Sc (テンプレート), 57, 58
-Sm (テンプレート), 57, 58
-Ss (テンプレート), 57, 58
リスト済み, 57
プリプロセッサ指示, 70
ブロードキャスト呼び出しサーバー応答, 71
プログラミング技法, 70, 78
変数宣言, 271
ポインタ, 47, 48
ポートモニターサポート, 51, 72
前処理命令, 56, 57, 71
メッセージ表示プログラム, 311, 314
メッセージプログラムの印刷, 45
ライブラリ
libnsl, 50, 52, 147, 148
TI-RPC または TS-RPC ライブラリの選択, 44, 68
rpcgenツール
ライブラリ
TI-RPC または TS-RPC ライブラリの選択, 58
rpcgen ツール
利点, 44
リモートプロシージャ呼び出しの障害, 50

rpcgen ツール (続き)
リモートプロシージャ呼び出しの命名, 48
ローカル手続きをリモートプロシージャに変換, 45, 51
rpcgen ツールl, socket 機能, 68
rpc_gss_principal_t principal 構造体名, 132
rpc_gss_principal_t 主体名の構造, 131
RPC_HDR 前処理命令, 56
RPCPROGVERSMISMATCH エラー, 142
RPCSEC_GSS セキュリティタイプ
etc/gss/mech/ ファイル, 137
/etc/gss/qop ファイル, 137
サービス
完全性, 126
/rpcsvc ディレクトリ, 252
RPC_SVC 前処理命令, 56
RPC_TBL 前処理命令, 56
RPC/XDR
RPC 言語 (RPCL)を参照
RPC_XDR 前処理命令, 56
RPC 言語, リファレンス, 304
RPC 言語 (RPCL), 21, 267, 268, 276
C, 43
C 形式モード, 275
portmap プロトコル仕様, 321, 322
RPC 言語(RPCL)
RPC 言語規則の例外, 274
RPC 言語 (RPCL)
void, 276
XDR 言語, 267
XDR 言語対, 304
XDR 言語との比較, 269
概要, 304
隠されたデータ, 275
RPC 言語(RPCL)
型定義, 270
可変長配列, 271
キーワード, 52
RPC 言語 (RPCL)
規則の例外, 276
共用体, 273
RPC 言語(RPCL)
共用体, 52
RPC 言語 (RPCL)
構造体, 272
RPC 言語(RPCL)
構造体, 52
RPC 言語 (RPCL)
構文, 268, 269
固定長配列, 271
RPC 言語(RPCL)
固定長配列, 271
RPC 言語 (RPCL)
識別型共用体, 273
RPC 言語(RPCL)
識別型共用体, 273
識別した共用体, 52
RPC 言語 (RPCL)
仕様, 267, 276
宣言, 270, 272
単純宣言, 270
定義, 269
定数, 270
で記述されたプロトコル例, 46
配列, 271, 272
ブール値, 275
プログラム宣言, 273, 274
ポインタ, 272
文字列, 47, 275
RPC 言語(RPCL)
列挙, 52
RPC 言語 (RPCL)
列挙型, 269, 270
RPC コール, レコードマーク標準, 256
RPC (リモートプロシージャ呼び出し)
アドレス translation, 147
一時的な RPC プログラム番号, 143
インタフェースルーチン
トップレベル, 49, 50
エラー, 50
障害, 50
名前からアドレスへの変換, 147
複数のクライアントバージョン, 142
複数のサーバーバージョン, 141
rstat プログラム, マルチスレッド, 156
runwait 構成スクリプトキーワード, 341

S

SAC
sac.h ヘッダーファイル, 347, 349
sacadm コマンド, 140, 333, 335, 342, 343
_sacpipe ファイル, 330, 338
_sactab ファイル, 333

SAC (続き)
キーファイル, 333
起動, 338, 340
主要ファイル, 337
メッセージインタフェース, 330, 332, 338, 344
sac.h ヘッダーファイル, 347, 349
sacadm コマンド, 140, 326, 333, 335, 342, 343
_sacpipe ファイル, 330, 338
_sactab ファイル, 333
SAC (サービスアクセスコントローラ), sacadm コマンド, 326
SAF, 21
SAC (サービスアクセスコントローラ), 328, 330, 332
概要, 328
管理インタフェース, 338, 342, 344
pmadm コマンド, 139, 140, 329, 334, 336 , 343, 344
_pmtabファイル, 338
_pmtab ファイル, 329, 335
sacadm コマンド, 140, 333, 335, 342, 343
_sactab ファイル, 333
サービスインタフェース, 336
主要ファイル, 337
ポートモニターが実行すべきタスク, 338
ポートモニター実装条件, 337
モニター固有のコマンド, 336
管理コマンド
モニター固有のコマンド, 336
構成スクリプト, 339, 344
インストール, 342, 344
置き換え, 342, 344
記述言語, 340, 342
SAF)
構成スクリプト
サービスごと, 344
SAF
構成スクリプト
サービスごと, 328, 329, 337, 338, 339, 340, 343
システムごと, 339, 340, 342, 343
出力, 342, 344
ポートモニターごと, 338, 339, 340, 343
コーディング例, 344
使用ファイル, 329, 333, 335, 337
ディレクトリ構造, 351
SAF (続き)
ポートモニター機能, 328
ポートモニターの機能, 338
ポートモニターの終了, 328, 338
メッセージインタフェース, 330, 332, 338, 344
論理ダイアグラム, 349, 350
SAF (サービスアクセス機能), 325
SAC (サービスアクセスコントローラ), 326
概要, 325
管理インタフェース, 332
SAF (サービスアクセス機能
管理インタフェース
_pmtab ファイル, 333
SAF (サービスアクセス機能)
管理インタフェース
sacadm コマンド, 326
モニタ固有のコマンド, 336
ポートモニターの機能, 326
sc_size フィールド, 332
socket 機能
TS-RPC (トランスポート特定リモートプロシージャ呼び出し)を参照
spray.x (スプレイパケット) プログラム, 310, 311
STREAMS モジュール
ポートモニター構成, 341
ポートモニターの構成, 343
Sun RPC
TI-RPC (トランスポート独立リモートプロシージャ呼び出し)を参照
svc.c 接尾辞, 50
svctag ファイル, 335, 338, 339
SVCXPRT サービストランスポートハンドル, 138, 159
_svc 接尾辞, 69
sysconfig ファイル, 342, 343
_sysconfig ファイル, 339, 340, 342

T

/tag ディレクトリ, 21
TCP, 21
TCP (, nettype パラメータ, 38
TCP
portmap シーケンス, 320, 321
portmap ポート番号, 322

TCP (続き)
サーバークラッシュ, 249
TCP/IP ストリーム
XDR, 239, 240, 256
TCP/IP プロトコル
TCP (Transport Control Protocol)を参照
TCP (トランスポート制御プロトコル)
pUDP アプリケーションを TS-RPC から
TI-RPC への移行, 145
RPC プロトコル, 248
tcp トランスポートタイプ, 38
thread.h ファイル, 162
`time.x` プログラム, 68
time.x プログラム, 56, 309
TI-RPC, 21
raw、下位レベルを使ったプログラム, 105
アドレス変換, 39, 40
アドレスルックアップサービス, 36, 40, 41
一時的 RPC プログラム番号, 251
一時的な RPC プログラム番号, 143, 144
インタフェースルーチン, 34, 35, 79, 80, 87,
88
エキスパートレベル, 94
下位レベルのデータ構造, 100
キャッシュサーバー, 100
単純, 79, 87
中間レベル, 36, 92
トップレベル, 91
標準, 35, 87
ボトムレベル, 99
情報の取り出し, 41, 143
データ表現, 33
トランスポート選択, 38
トランスポートタイプ, 38
名前からアドレスへの変換, 39, 40
ネットワーク選択, 37
プロトコル, 33, 247, 249, 256
XDR 言語, 253, 256
手続きの識別, 34
トランスポートプロトコルと意味論,
248
認証, 250, 251
バージョン番号, 250
TI-RPC (
プロトコル
プロシージャの特定, 249
TI-RPC
プロトコル
プロシージャの特定, 250
プロシージャを特定, 252
レコードマーク標準, 256
割り当てと相互認識の独立性 , 249
呼び出しセマンティクス, 33
ライブラリ選択、rpcgen ツール, 68
リモートプロシージャの識別, 34
リモートプロシージャの特定, 249, 250
リモートプロシージャを特定, 252
TI-RPC (トランスポート独立リモートプロシー
ジャ呼び 出し)
アドレス変換, 147
TI-RPC (トランスポート独立リモートプロシー
ジャ呼び出し)
インタフェースルーチン
トップレベル, 49
TI-RPC(トランスポート独立リモートプロシー
ジャ呼び出し)
インタフェースルーチン
トップレベル, 50
TI-RPC (トランスポート独立リモートプロシー
ジャ呼び 出し)
名前からアドレスへの変換, 147
TI-RPC (トランスポート独立リモートプロシー
ジャ呼び出し)
ライブラリ選択、rpcgen ツール, 58
ライブラリ選択、rpcgen ツールおよび, 44
TLI ファイル記述子
オープンした~を渡す, 94, 97
TS-RPC から TI-RPC への移行, 145
TS-RPC から TI-RPCへの移行
`libc` ライブラリおよび, 147
libnsl ライブラリ, 147
TI-RPC と TS-RPCの相違点, 147, 151
TS-RPC から TI-RPC への移行
アプリケーション, 145
および旧インタフェース, 147
関数の互換性一覧表, 148
TS-RPC からTI-RPC への移行
関数の互換性のリスト, 150
TS-RPC から TI-RPCへの移行
コード比較例, 151
TS-RPC からTI-RPC への移行
名前からアドレスのマッピング, 147
TS-RPC から TI-RPC への移行
利点, 146

TS-RPC (トランスポート特定リモートプロシージャ呼び出し), ライブラリ選択、rpcgen ツール, 58
ttyadm コマンド, 329
t時間を指定した作成, 中間レベルのインタフェース, 93

U

UDP, 21
  nettype パラメータ, 38
  portmap ポート番号, 322
  サーバー作成ルーチン, 98
  ブロードキャスト RPC および, 113
UDP/IP プロトコル
  UDPを参照
udp トランスポートタイプ, 38
UDP (ユーザーデータグラムプロトコル)
  RCP プロトコル, 248
  UDP アプリケーションを TS-RPC から TI-RPC への移行, 145
  クライアント作成ルーチン, 97
  サーバー作成ルーチン, 100
ulimit コマンド, 342
umask コマンド, 342
/usr/include/rpcsvc ディレクトリ, 252
/usr/share/lib ディレクトリ, 55
utmpx エントリ
  作成, 328, 335, 338

V

/var/saf/ ディレクトリ, 329, 333
void 宣言
  RPC 言語, 276
  XDR 言語, 298
void 引数, 274, 275

X

XDR, 21
  rpcgen ツール, 51, 56
  から変換 (デシリアライズ), 84, 108, 109
  処理内容の決定, 238
XDR (続き)
  ストリーム
    RPC システムによる作成, 222
    アクセス, 238
    インタフェース, 240, 242
    新規作成, 240, 242
    標準入出力, 238
    フィルタ以外のプリミティブ, 237
    メモリー, 238
    レコード (TCP/IP), 239, 240, 256
  データの標準形式, 221
  に変換 (シリアライズ), 51, 56, 108, 110
  によるメモリー割り当て, 108, 110
  プリミティブ, 222
  プリミティブルーチン, 84
    隠されたデータ, 233
    共用体, 234
    固定長配列, 234
    識別型の共用体, 234
    バイト配列, 229
    配列, 232, 234
    フィルタ以外, 237
    ポインタ, 236, 237
    文字列, 228
  ブロックサイズ, 286
  プロトコル
  XDR 言語を参照
  への変換 (シリアライズ), 84
  変換コスト, 221
  変換 (シリアライズ), 224
  変換(シリアライズ), 222
  変換 (デシリアライズ), 224
  変換(デシリアライズ), 223
  ボックス図による説明, 286
  ライブラリ, 21, 222, 224
  リンクリスト, 242
  ルーチンの最適化, 238
xdr_array ルーチン, 232
xdr_bytes ルーチン, 229
XDR_DECODE 処理, 228
XDR_ENCODE 処理, 228
XDR_FREE 処理, 228
xdr_inline カウント, 58, 69
xdr_prefix, 52
xdrs-x_op フィールド, 238
xdr_type (オブジェクト) 表記法, 124
XDR (外部データ表現), 21
  rpcgen ツール, 56

XDR (外部データ表現) (続き)
ファイルのデータ構造, 303
リンクリスト, 300
XDR 言語, 21, 287, 291
4 倍精度浮動小数点, 292
AUTH_DES 認証プロトコル, 260
RPC 言語, 267
RPC 言語対, 304
RPC メッセージプロトコル, 253, 256
void, 298
オプションデータの共用体, 299
概要, 285, 286
カウント付きバイト文字列, 294, 295
隠されたデータ, 292, 294
型の定義, 298, 299, 302
可変長の隠されたデータ, 293, 294
可変長配列, 295
キーワード, 302
共用体, 297, 299, 302
構造体, 296, 302
構文, 302, 303
固定長の隠されたデータ, 292
固定長配列, 295
コメント, 300
識別型共用体, 297, 299, 302
識別子, 300
宣言, 287, 300
定数, 298, 301, 302
認証プロトコル, 263
の仕様, 300
配列, 295, 296, 302
ブール型, 289
符号なし整数, 288
浮動小数点, 290, 292
文字列, 294, 295
列挙型, 288
.x 接尾辞, 52

あ

アクセス制御
認証, 121
ポートモニター, 327
アドレス, 39
RPC サービスの割り当て, 21
概要, 319, 320
管理関数, 150
アドレス (続き)
サーバーのアドレスをクライアントに渡す, 94
情報の取り出し, 41
登録, 21
登録解除, 323
として引数を引き渡す, 82
トランスポート (netbuf), 40
名前からアドレスへの変換ルーチン, 39
ネットワーク, 319, 320
汎用, 39, 281, 282, 320
引数を渡す, 47, 48
ユーザーの結合アドレスを渡す, 97
ルックアップサービス, 40, 41
アプリケーション, TS-RPC から TI-RPC への移行, 145
アプリケーションプログラミングインタフェース (API)
NIS+, 201, 205
暗号, AUTH_DES 認証 (Diffie-Hellman), 263
暗号化
AUTH_DES 認証 (Diffie-Hellman), 123, 259, 262
AUTH_KERB 認証, 124, 125
プライバシサービス, 126
暗号化ブロックチェイン (CBC) モード, 124

い

一時的なプログラム番号, 143, 144
一時的プログラム番号, 251
一覧表示
NIS+ objects, 211
NIS+ オブジェクト, 205, 212
NIS+ サーバー, 203
NIS+ 主体, 203
NIS+ テーブルオブジェクト, 210, 212
rpcbind割り当て, 279
意味論, TI-RPC コール, 249
印刷
システムコンソールへのメッセージ, 45, 51
インターネットプロトコル
TCP (トランスポート制御プロトコル)を参照
インタフェース
RPC (リモートプロシージャ呼び出し)、インタフェースルーチンを参照
インデックステーブル, rpcgen ツール, 56

え
エキスパートレベルのインタフェースルーチン
(RPC), 98
クライアント, 97
サーバー, 98
エキスパートレベルのインタフェースルーチン
ルーチン (RPC), 94
概要, 94
エラー
NIS+ エラーメッセージ表示関数, 201, 204
RPC, 50, 78, 250
クライアントハンドルの作成, 89
認証
AUTH_DES, 260
AUTH_KERB, 265
複数のクライアントバージョン, 142

お
オープンした TLI ファイル記述子
渡す, 94, 97
オブジェクト (NIS+), 201
サンプルプログラム, 206, 216
オブジェクト(NIS+), サンプルプログラム, 206
オブジェクト (NIS+)
操作関数, 201, 205
オプションデータの共用体, XDR 言語, 299

か
外部データ表現, 21
下位レベルのデータ構造, 100
会話鍵
AUTH_DES 認証, 259, 261, 263
カウント付きバイト文字列, 21
隠されたデータ
XDR コーディング例, 233
宣言
RPC 言語, 275
XDR 言語, 292, 294
可視トランスポートタイプ, 38
型の定義
RPC 言語, 270
XDR 言語, 298, 299, 302
可変長の隠されたデータ, 21
XDR 言語, 293, 294
可変長配列宣言, 21
RPC 言語, 271
XDR 言語, 295
間接 RPC, 280
完全性, 126

き
キーワード
RPC 言語, 52
XDR 言語, 302
記憶域の非割り当て, 21
キャッシュ, サーバー, 100
キャッシュに書き込む, NIS+, 201
共用体
XDR コーディング例, 234
宣言
RPC 言語, 52, 273
XDR 言語, 297, 299, 302

く
クライアント
NIS+, 200
TS-RPC 対 TI-RPC, 151, 152
トランザクション ID, 249
バッチ, 115, 117, 314, 315
複数のバージョン, 142, 143
マルチスレッド, 155
対応, 79
ユーザーモード, 162, 164, 166
マルチスレッド対応
対応, 62, 64, 65, 67
クライアント側のスタブルーチン
rpcgen ツール, 43, 49, 50
MT 自動モード, 68
前処理命令, 56
クライアント側のタイムアウト期間, rpcgen
ツール, 73
クライアント側のテンプレート
rpcgen ツール, 44, 57, 58, 60, 61
クライアント側プログラム
rpcgen ツール
ANSI C 準拠, 69

クライアントスタブルーチン
rpcgen ツール
C 形式モード, 60, 61
MT 対応, 62, 63
MT 未対応, 63, 64
クライアント認証, 21
クライアントのタイムアウト期間, 35
時間を指定したクライアントの作成, 93
クライアントのタイムアウト期限, 時間を指定したクライアントの作成, 90
クライアントハンドル, 35, 36, 37
下位レベルのデータ構造, 100, 101
作成, 88
エキスパートレベルのインタフェース, 94, 97
中間レベルのインタフェース, 92, 93
作成
トップレベルのインタフェース, 90
作成
トップレベルのインタフェース, 35, 49, 50
ボトムレベルのインタフェース, 99
破棄
エキスパートレベルのインタフェースルーチン, 97
トップレベルのインタフェース, 49, 90
クライアントプログラム
rpcgen ツール, 51
MT 対応, 65, 67
概要, 45, 48
ディレクトリリストサービス, 54, 56
ディレクトリリストプログラム, 308
デバッギング, 77
デバッグ, 78
複雑なデータ構造の引き渡し, 54
複雑なデータ構造の引渡し, 56
メッセージ印刷コードの例, 48, 51
rpcgen ツールおよび
MT 対応, 44
単純インタフェース, 82
リモートコピー, 106, 107
クラッシュ
サーバー, 249, 260
グループ (NIS+), 200
サンプルプログラム, 207, 208, 213, 216
操作関数, 201, 203
クロック同期, 21

こ

構成スクリプト, 21
構成スクリプトキーワードの実行, 342
構成スクリプトキーワードをプッシュする, 340
合成データ型フィルタ, XDR, 228
構造体宣言, 21
RPC 言語, 52, 272
XDR 言語, 296, 302
構築されたデータ型フィルタ, XDR, 86
構文
RPC 言語, 268, 269
XDR 言語, 302, 303
コール意味論, TI-RPC, 249
コールバック手続き, 143
RPCSEC_GSS, 135
一時的な RPC プログラム番号および, 143, 144
に使用, 144
コールバックプロシージャ
NIS+, 211, 212
互換性
ライブラリ関数, 148, 150
固定長の隠されたデータ, XDR 言語, 292
固定長配列, 21
XDR コーディング例, 234
XDR サンプルコード, 233
宣言
RPC 言語, 271
XDR 言語, 295
コネクションオリエンテッドトランスポート, クライアントハンドル作成, 37
コピー
NIS+ オブジェクト, 205
NIS+ データベースエントリ, 204
NIS+ テーブルエントリオブジェクト, 202
リモート, 105, 108
コメント, XDR 言語, 300
コンパイル
NIS+, 206
rpcgen ツール, 44, 59, 62

さ

サーバー, 21
NIS+, 198, 203, 204
およびポートモニター, 138, 140

サーバー (続き)
キャッシュ, 100
クラッシュ, 249, 260
終了間隔、rpcgen ツール, 72
ディスパッチテーブル, 70, 74, 75
トランザクション ID, 249
認証, 119, 120, 121, 124
認証および, 123
バッチ, 117, 118, 314, 316
複数のバージョン, 140, 141
ポーリングルーチン, 111, 113
マルチスレッド, 155
自動モード, 157, 159, 162
対応, 43, 79, 158
ユーザーモード, 157, 159, 162, 166
マルチスレッド対応
自動モード, 44, 58, 68
対応, 44, 64, 65, 67
サーバーservers, マルチスレッド, 157
サーバー側のスタブルーチン, 21
rpcgen ツール, 43, 45, 50
ANSI C 準拠, 69
MT 自動モード, 68
MT 対応, 43, 64, 65
前処理命令, 56
サーバー側のテンプレート
rpcgen ツール, 44, 57, 58, 61
サーバートランスポートハンドル, 101
サーバーハンドル, 21, 36
下位レベルのデータ構造, 101
作成, 35, 37, 149
エキスパートレベルのインタフェース, 97, 98
中間レベルのインタフェース, 93
トップレベルのインタフェース, 90, 91
廃棄, 149
サーバープログラム
rpcgen ツール
C 形式モード, 61
MT 自動モード, 68
MT 対応, 43, 44, 67
概要, 45, 50
クライアントの認証, 73
ディレクトリリストサービス, 53, 54
ディレクトリリストプログラム, 305
rpcgenツール
ディレクトリリストプログラム, 307
rpcgenツール (続き)
rpcgen ツール
デバッギング, 77
ネットワークタイプ/トランスポート選択, 70
複雑なデータ構造の引き渡し, 53, 54
ブロードキャスト呼び出し応答, 71
ブロードキャスト呼び出しへの応答, 70
rpcgen ツール1
クライアント認証, 70
rpcgen ツールおよび
クライアント認証, 119
一時的な RPC プログラム, 144
および rpcgen ツール
クライアント認証, 120
デバッグ, 78
単純インタフェース, 82
リモートコピー, 107, 108
サービス, 126
サービスアクセス機能
SAFを参照
サービスアクセスコントローラ
SACを参照
サービスディスパッチルーチン, 認証, 120
サービストランスポートハンドル (SVCXPRT), 159
再帰データ構造体, 245
再帰的データ構造, 242
再帰的データ構造体, 300
最大
スレッドの最大数, 160
ブロードキャスト要求サイズ, 113
削除, 21
NIS+ オブジェクト, 205
NIS+ オブジェクト、名前空間から, 213
NIS+ グループオブジェクト, 203, 213, 216
NIS+ グループメンバー, 203
NIS+ テーブルエントリオブジェクト, 202, 213, 216
XDR ストリーム, 237
アドレス登録, 21
関連, 37
ポートモニター, 326
ポートモニターサービス, 334
ホストからNIS+ ディレクトリ, 203
マッピング, 36
作成
NIS+ グループオブジェクト, 203, 207, 208

作成 (続き)
NIS+ ディレクトリオブジェクト, 207
NIS+ データベース, 203
NIS+ テーブルオブジェクト, 208, 210
utmpx エントリ, 328, 335, 338
サンプルコード
時刻サーバープログラム, 309
ディレクトリリストプログラム, 305, 308
サンプルプログラム, 305, 309, 317
スプレイパケットプログラム, 310, 311
ディレクトリリストプログラム, 305
バッチコード, 314, 317
メッセージ表示プログラム, 311, 314

し

資格
AUTH_DES, 261
AUTH_DES, 122, 261
AUTH_KERB, 124, 125, 265, 266
ウィンドウ, 122, 124
ウィンドウ (寿命), 260, 261, 266
資格ウィンドウ
AUTH_DES 認証, 261
AUTH_KERB 認証, 266
資格のウィンドウ
AUTH_DES 認証, 122
AUTH_KERB 認証, 124
ウィンドウベリファイア, 260
資格の存在時間
資格のウィンドウを参照
時間によるクライアントの作成, 35, 36
時間を指定したクライアントの作成, トップレベルのインタフェース, 90
識別, 21
ポートモニターサービス, 334
リモートプロシージャ, 34
識別型共用体
宣言
RPC 言語, 273
XDR 言語, 273, 297, 302
識別型の共用体, XDR コーディング例, 234
識別子, XDR 言語, 300
時刻
ping プログラム, 267, 268
現在時刻を取得, 259
時刻サーバープログラム, 56, 68, 309
時刻サービス
rpcbind ルーチン, 259
中間レベルのクライアント, 92, 93
中間レベルのサーバー, 93
トップレベルのクライアント, 90
トップレベルのサーバー, 91
トップレベルのサービス, 90
時刻の同期
AUTH_DES 認証, 260
AUTH_KERB 認証, 266
AUTH_KERB 認証, 267
時刻の同期化
AUTH_DES 認証, 123
AUTH_KERB 認証, 124
自動 MT モード, 21
修正
変更を参照
出力
ポートモニター構成スクリプト, 342, 344
使用可能にする, ポートモニター, 330
状態の取り出し
情報の取り出しを参照
承認, NIS+, 199
情報の取り出し
RPC, 41
アドレス, 41
サーバーコールバック, 143
リモートホストの状態, 156
情報レポート
NIS+, 203
シリアライズ, 21, 51, 56, 84, 87, 108, 110, 219, 222, 224
シングルスレッドモード
デフォルトとして, 157
ポーリングルーチン, 113
シングルスレッドモデル, ポーリングルーチンおよび, 111

す

スタブルーチン
クライアント側のスタブルーチンを参照
ステートメントを定義する、コマンド行、rpcgen ツール, 71
ストリーム
XDR (外部データ表現)、ストリームを参照
ストリームトランスポート, 21

スレッド
　MT 自動モードを参照
スレッドライブラリ, スレッド,　156

せ
制限, ブロードキャスト要求サイズ,　113
整数,　21
　XDR 言語,　219, 220
整数フィルタ XDR,　226, 227
セキュリティ,　21
　サービス,　126
　メカニズム,　126
セキュリティ?, NIS+,　199
接続型端点,　102
接続型トランスポート,　21
　nettype パラメータ,　38
　およびポートモニター,　138
　サーバーハンドルの作成,　37
　使用するとき,　39
　リモートコピーコード例,　105, 108
接続トランスポート, UDP,　281
セマンティクス, TI-RPC 呼び出し,　33
宣言
　RPC 言語,　270, 276
　XDR 言語,　287, 300

そ
相互認識, TI-RPC,　249

た
タイムアウト期間
　rpcgen ツール,　70, 73
単純インタフェース (RPC), XDR 変換,　84
単純インタフェースのルーチン,　34
単純インタフェースルーチン (RPC),　79, 87
　XDR 変換,　87
　サーバー,　82
　ハンドコード登録ルーチン,　83
単純宣言, RPC 言語,　270
端点, 接続型,　102

ち
置換
　変更を参照
中間レベルのインタフェースルーチン,　36
中間レベルのインタフェースルーチン (RPC),
　92
チュートリアル
　rpcgen ツール,　45, 57

つ
追加
　NIS+ グループメンバー,　203
　NIS+ データベースエントリ,　204
　NIS+ テーブルエントリオブジェクト,　202,
　209, 210
　アドレスの登録,　40
　ポートモニター,　326
　ポートモニターサービス,　334
ツリー,　300

て
定数
　RPC 言語,　270
　XDR 言語,　298, 301, 302
ディスパッチテーブル
　rpcgen ツール,　74, 75
/ ディレクトリ,　329, 333
ディレクトリ
　SAF,　351
　リモートディレクトリリストサービス,　51,
　56
　リモートディレクトリリストプログラム,
　305, 308
データ型
　任意のデータ型を引き渡す,　84, 87
データグラムトランスポート,　21
　nettype パラメータ,　38
　およびブロードキャスト RPC,　113
データ構造,　21
　MT 対応,　162
　xdr_inline によるパッキング,　58
　XDR 形式に変換,　87
　XDR 形式への変換,　220
　下位レベル,　100

データ構造 (続き)
再帰的, 242
データ構造体
rpcgen ツール, 51, 56
xdr_inline によるパッキング, 69
構造体宣言, 21
再帰, 245
再帰的, 300
データ表現, 21
TI-RPC, 33
データベースアクセス関数 (NIS+), 201, 203, 204
テーブル (NIS+), 198, 199, 200
アクセス関数, 201, 202
サンプルプログラム, 208, 210
デーモン
kerbd, 264
rpcbind, 40
デシリアライズ, 21
デシリアライズが可能, 223
テスト
NIS+ グループ, 203
下位レベル raw RPC を使ったプログラム, 105
手続き
RPC 手続き, 269
RPC プログラムとして登録, 83
RPC プログラムとしての登録, 35
手続き番号, 21
エラー条件, 250
手続きリスト, RPC 言語, 269
デバッギング, rpcgen ツール, 77
デバッグ
raw モード, 105
rpcgen ツールおよび, 71
および rpcgen ツール, 78
デフォルト
rpcgen ツール, 21
シングルスレッドモード, 157
スレッドの最大数, 160
電子コードブック (ECB) モード, 124
テンプレート
rpcgen ツール, 44, 57, 58, 60, 61

と
同期, 時間同期を参照, 21

テンプレート (続き)
NIS+, 204
統計
情報の取り出しを参照
動的なプログラム番号, 143, 144
動的プログラム番号, 251
動的割り当て, 320
登録, 149
RPC プログラムとして登録, 83
RPC プログラムとしての手続き, 35, 82
アドレス
portmap ルーチンを参照
認証番号, 256
ハンドコード登録ルーチン, 83
プログラムのバージョン番号, 140
プログラム番号, 252
登録削除, 149
特定
リモートププロシージャ, 250
リモートプロシージャ, 249, 252
トップレベルのインタフェースルーチン, 35
トップレベルのインタフェースルーチン (RPC), 49, 88, 91
トップレベルのインタフェースルーチン(RPC), 50
トップレベルのインタフェースルーチン (RPC)
概要, 88
クライアント, 90
トップレベルのインタフェースルーチン(RPC)
クライアント, 49, 50
トップレベルのインタフェースルーチン (RPC)
サーバー, 90, 91
ドメイン (NIS+), 198, 200
関数, 203, 205
トランザクション ID, 34, 249
トランザクションログ関数 (NIS+), 201
トランスポート選択
RPC, 38
rpcgen ツール, 70
トランスポートタイプ, 21
rpcgen ツール, 70
インタフェース, 88
トランスポート特定リモートプロシージャ呼び出し, 21
トランスポートハンドル, 21
SVCXPRT サービス, 138, 159
サーバー, 101
トランスポートプロトコル, 21

トランスポートハンドル (続き)
RPC プロトコル, 248
トランスポートレベルのインタフェースファイル記述子
オープンした~を渡す, 94, 97

な

名前からアドレスへの変換, 147
名前サービス
NIS+ (Network Information Services Plus)を参照

に

ニックネーム
AUTH_DES, 261
AUTH_DES, 260, 262
AUTH_KERB, 265, 266
AUTH_KERB, 125, 265, 267
入出力ストリーム, XDR, 238
認証, 21, 125, 150, 256, 267
AUTH_DES, 122, 124, 263
AUTH_KERB, 124, 125, 267
AUTH_NONE, 256
AUTH_SHORT, 257, 258
AUTH_SYS (AUTH_UNIX), 257, 258
NIS+, 199
rpcgen ツール, 73
RPC プロトコル, 250, 251
アクセス制御, 121
エラー
AUTH_DES, 260
AUTH_KERB, 265
概要, 256
下位レベルのデータ構造および, 101
サーバー, 119, 120, 121, 123, 124
サービスディスパッチおよびルーチン, 120
サービスディスパッチルーチン, 119
サポートされている方法, 118
資格
AUTH_DES, 122, 261
AUTH_KERB, 124, 125, 265, 266
ウィンドウ, 122, 124, 260
ウィンドウ(寿命), 261, 266
資格 (続き)
時刻の同期
AUTH_DES 認証, 260
AUTH_KERB 認証, 266, 267
時刻の同期化
AUTH_DES 認証, 123, 259
AUTH_KERB 認証, 124
ニックネーム, 125
AUTH_DES, 260, 261, 262
AUTH_KERB, 125, 265, 266, 267
認証番号の登録, 256
認証番号の割り当て, 256
破棄, 119
ハンドル, 101, 122, 123
ベリファイア
AUTH_DES, 259, 260
AUTH_KERB, 124, 125, 265, 266, 267
AUTH_SYS, 257, 258

ね

ネームサービススイッチ, 200
ネット名, 122, 258
ネットワークアドレス, 21
ネットワーク選択
RPC, 37
rpcgen ツール, 70
ネットワークパイプ, 219
ネットワークファイルシステム
NFS (ネットワークファイルシステム)を参照
ネットワーク名, 122, 258

は

バージョン, 21
RPC 言語, 269
ライブラリ関数, 150
ライブラリ関数の互換性, 148
バージョン番号, 21
エラー条件, 250
登録, 140
複数のクライアントバージョン, 142
複数のクライアント番号, 143
複数のサーバーバージョン, 140, 141
ポートモニター, 335, 336
マップ, 21

バージョン番号 (続き)
メッセージプロトコル, 250
割り当て, 140
バージョンリスト, RPC 言語, 269
廃棄, サーバーハンドル, 149
バイト配列, XDR, 229
パイプ
_pmpipe ファイル, 330, 338
_sacpipe ファイル, 330, 338
ネットワーク, 219
配列, 21
XDR 形式に変換, 108, 110
XDR 形式への変換, 86
XDR コーディング例, 229, 232, 233, 234
宣言
RPC 言語, 271, 272
XDR 言語, 295, 296, 302
バインド, 21
破棄, 21
クライアント認証ハンドル, 119
クライアントハンドル, 49, 90, 97
バッチ, 118, 252, 314, 317
バッファサイズ
送信と受信を指定する, 94, 97
パラメータ
引数を参照
番号, 21
番号フィルタ、XDR, 21
ハンドル, 21
認証, 101, 122, 123
汎用アドレス, 39, 281, 282, 320

ひ

引数 (リモートプロシージャ)
void, 274, 275
値による引き渡し, 62
引数(リモートプロシージャ), 値による引き渡し, 59
引数 (リモートプロシージャ)
アドレスで渡す, 47, 48, 82
オープンした TLI ファイル記述子を渡す, 94, 97
サーバーのアドレスをクライアントに渡す, 94
任意のデータ型を引き渡す, 84, 87
ユーザーの結合アドレスを渡す, 97
引き渡すパラメータ
引数を参照
非接続型トランスポート
nettype パラメータ, 38
クライアントハンドル作成, 37
サーバーハンドル作成, 37
使用するとき, 38
非接続トランスポート, 21
UDP (ユーザーデータグラムプロトコル), 21
日付サービス
単純用トップレベルのクライアント, 90
中間レベルのクライアント, 92, 93
中間レベルのサーバー, 93
トップレベルのサーバー, 90, 91
非同期モード, 111, 113
表示
システムコンソールへのメッセージ, 311, 314
標準
ANSI C 標準、rpcgen ツール, 44, 58, 69
RPC, 32, 256
XDR データの標準形式, 221
レコードマーク標準, 256
標準インタフェースルーチン, 35
中間レベルのルーチン, 36
標準インタフェースルーチン (RPC), 35, 87
MT 対応, 79
エキスパートレベルのルーチン, 94, 98
下位レベルのデータ構造, 100
中間レベルのルーチン, 92
トップレベルのルーチン, 88, 91
標準インタフェースルーチン(RPC)
トップレベルのルーチン, 49, 50
標準インタフェースルーチン (RPC)
ボトムレベルのルーチン, 99
標準規格, 命名方法の標準規格, 259

ふ

ファイル記述子、オープンした TLI を渡す, 94, 97
ファイルシステム, 21
ファイルのデータ構造, XDR 言語, 303
フィルタ (XDR)
隠されたデータ, 233
共用体, 234

フィルタ (XDR) (続き)
合成 (複合) データ型, 228
構築された (複合) データ型, 86
整数, 226, 227
配列, 229, 232, 233, 234
番号, 84
浮動小数点, 227
文字列, 86, 228, 229
列挙型, 227, 228
フィルタ番号、XDR, 84
ブール型, XDR 言語, 289
ブール値, RPC 言語, 275
複合データ型フィルタ
XDR, 86, 228
複雑なデータ構造, 21
rpcgen ツール, 51, 56
xdr_inline によるパッキング, 58, 69
複数のクライアントバージョン, 142, 143
複数のサーバーバージョン, 140, 141
複製サーバー
NIS+, 198, 200
符号なし整数, XDR 言語, 288
浮動小数点
XDR 言語, 290, 292
浮動小数点フィルタ, XDR プリミティブ, 227
プライバシ, 126
フラグ
rpcgen ツール、フラグを参照
ブロードキャスト RPC, 35, 71, 114, 115, 149
概要, 253
へのサーバーの応答, 114, 115
ルーチン, 115
プログラム宣言
RPC 言語, 273, 274
プログラム定義, RPC 言語, 269
プログラム番号, 21, 250, 252
一時的な (動的に割り当てた), 251
一時的な (動的に割り当てられた), 143, 144
エラー条件, 250
登録, 252
マップ
portmap ルーチンを参照
割り当て, 251, 252
プロトコル, 21
AUTH_DES, 260
AUTH_DES, 263
RPC 言語による指定, 46

へ
ヘッダーファイル
rpcgen ツール, 56
rpcgen ツールおよび, 50
ベリファイア
AUTH_DES, 259, 260, 262
AUTH_KERB, 267
AUTH_KERB, 124, 125, 265, 266
AUTH_SYS, 257, 258
変換
XDR 形式, 108, 219
XDR形式, 222, 223, 224
XDR 形式から, 91, 108, 109
XDR 形式から変換, 84
XDR 形式に, 51, 56, 87, 110
XDR 形式に変換, 84
アドレス, 39, 147
ローカル手続きをリモートプロシージャに, 45, 51
変更
NIS+ テーブルエントリオブジェクト, 202
ポートモニター構成スクリプト, 342, 344
ポートモニターサービス, 334
変数宣言, 271

ほ
ポインタ
RPC 言語, 272
XDR コーディング例, 236, 237
リモートプロシージャ, 47, 48
ポートデータ
XDR (外部データ表現)を参照
ポート番号, 21
TCP/IP プロトコル, 281, 322
UDP/IP プロトコル, 281, 322
登録されたサービス用に調べる, 319
ポートモニター, 138
_pmpipe ファイル, 330, 338
rpcgen ツール, 51, 72
utmpx エントリ作成, 338
utmpx エントリの作成, 328, 335
アクティビティの監視, 327
管理インタフェース, 332, 338
管理機能, 327
管理コマンド
pmadm, 139, 140, 329, 334, 336, 343, 344

管理コマンド (続き)
sacadm,　139, 326, 333, 335, 342, 343
モニター固有のコマンド,　336
モニタ固有のコマンド,　336
管理ファイル
_pmtab,　335
_pmtab,　329, 333, 338
_sactab,　333
機能,　326, 328, 338
構成スクリプト,　339, 344
インストール,　342, 344
置き換え,　344
記述言語,　340, 342
サービスごと,　328, 329, 337, 338, 339, 340, 343, 344
システムごと,　339, 340, 342, 343
出力,　342, 344
置換,　342
ポートモニターごと,　338, 339, 340, 343
コーディング例,　344
サービスインタフェース,　336
サービスの削除,　334
サービスの識別,　334
サービスの追加,　334
システムへのアクセス制限,　327
実装条件,　337
終了,　328, 338
使用可能にする,　330
使用不可にする,　330
タイプ,　333
追加,　326
バージョン番号,　335, 336
外す,　326
ファイル
管理,　329, 333, 335, 337
サービスごとの構成,　328, 329, 337, 338, 339, 343, 344
システムごとの構成,　339, 340, 342, 343
主要,　337
プライベート,　329, 333
プロセス ID,　328, 338
ポートモニターごとの構成,　338, 339, 340, 343
プライベートファイル,　329, 333
プロセス ID 及びロックファイル,　328, 338
ポートモニターサービスの変更,　334
無効化,　328
ポートモニター (続き)
メッセージインタフェース,　330, 332, 338, 344, 349
有効化,　328
ポートモニター listen, 管理コマンド,　329
ポートモニター ttymon,　329
ポートモニター構成スクリプトのインストール,　342, 344
ポートモニターの終了,　328, 338
ポートモニターの無効化,　328
ポートモニターを使用不可にする,　330
ポーリングルーチン,　111, 113
ボトムレベルのインタフェースルーチン (RPC),　99

ま

前処理命令
rpcgen ツール,　56, 57, 71
マスターサーバー
NIS+,　198, 200, 216
マップ,　21, 36
マルチスレッド RPC プログラミング,　21, 155
クライアント
対応,　79
サーバー,　155, 157
対応,　79, 158
マルチスレッド RPC プログラム
クライアント,　155
ユーザーモード,　162, 164, 166
サーバー,　159
自動モード,　157, 159, 162
タイミングダイアグラム,　158
ユーザーモード,　157, 159, 162, 166
スレッドの最大数,　160
パフォーマンス改善,　160, 166
ライブラリ,　156
マルチスレッド自動モード
MT 自動モードを参照
マルチスレッド対応 RPC プログラミング
rpcgen ツール,　58, 62, 68
rpcgen ツールおよび,　44
クライアント
対応,　44, 62, 64, 65, 67
サーバー
自動モード,　58, 68

マルチスレッド対応RPC プログラミング
サーバー
自動モード, 44
マルチスレッド対応 RPC プログラミング
サーバー
対応, 43, 64, 65, 67
マルチスレッド対応 RPCプログラミング
サーバー
対応, 44
マルチスレッド対応コード
MT 対応コードを参照
マルチスレッドユーザーモード, 157, 159, 162

め
命名, 21
rpcgen によるクライアント側スタブプログラム, 50
rpcgenによるクライアント側スタブプログラム, 50
rpcgen によるサーバープログラム, 50
rpcgen によるリモートプロシージャ呼び出し, 48
rpcgen のテンプレートファイル, 59
ネット名, 122, 258
バージョン番号識別, 140
命名方法, 標準規格, 259
メインサーバー機能, 72
メカニズム、セキュリティ, 126
メッセージインタフェース (SAF), 330, 332, 338, 344, 349
メッセージクラス, 332
メモリ, 解放, 87
メモリー, 224
XDR による割り当て, 108, 110
XDR プリミティブ要件, 226, 228
解除
XDR_FREE 処理, 228
解放, 65
開放
clnt_destroy ルーチン, 49
解放
free ルーチン, 56
NIS+, 202, 203, 204
xdr_free ルーチン, 56
メモリーストリーム, XDR, 238
メモリ、解放, 21
メモリの非割り当て
メモリ、解放を参照
メモリの割り当て, 21

も
文字列宣言
RPC 言語, 47, 275
XDR 言語, 294, 295
文字列表現, 21
XDR ルーチン, 86, 228

ゆ
有効化
サーバーのキャッシュ, 100
ポートモニター, 328
ユーザー
数, 80
ネットワーク上にいる, 121
ユーザー MT モード, 157, 159, 162
ユーザー数, リモートホスト上にある, 80
ユーザーの数, ネットワーク上にいる, 121
ユーザーの結合アドレス, 渡す, 97

よ
呼び出しセマンティクス, TI-RPC, 33
呼び出し側, サービスアクセス機能を用いた記述, SAF (サービスアクセス機能)を参照, 21
呼び出し側の無効化, 21

ら
ライブラリ
lib, 55
libc, 147, 148
libnsl, 50, 52, 147, 148
librpcsvc, 80
rpcgen ツール
TI-RPC または TS-RPC ライブラリの選択, 44, 58, 68
rpcgen ツールおよび, 147
libnsl, 148

rpcgen ツールおよび (続き)
libnsl, 50, 52
RPC 関数, 148, 150
XDR, 21, 222, 224
スレッド, 156

り
リスト, 21
portmap マップ, 323
rpcbind マッピング, 34
リモートディレクトリリストサービス, 51, 56
リモートディレクトリリストプログラム, 305, 308
リソース解放, 21
リモート時刻プロトコル, 56, 68
リモートディレクトリリストサービス, 51, 56
リモートプロシージャ
識別, 34
特定, 249, 250, 252
ローカル手続きを変換, 45, 51
リモートプロシージャ呼び出し
RPCを参照
リンクリスト
XDR, 242, 245, 300

れ
レコードストリーム
XDR, 239, 240, 256
レコードマーク標準, 256
列挙, RPC 言語, 52
列挙型
RPC 言語, 269, 270
XDR 言語, 288
列挙型フィルタ
XDR プリミティブ, 227, 228
レポート
情報の取り出しを参照

ろ
ローカル手続き
リモートプロシージャに変換, 45, 51
ロード
バインドを参照
ログ関数
NIS+ トランザクション, 201, 204
ロック
mutex、マルチスレッドモードおよび, 158
ポートモニター ID とロックファイル, 328, 338

わ
割り当て
TI-RPC, 249
動的, 320