情報システム工学 II(システム開発工学)

RPC プログラミング

Distributed Programming with Remote Procedure Call

山田泰司

taiji@aihara.co.jp

株式会社あいはら 研究開発チーム

RPC プログラミングとは

- **手続き呼出しのように、簡単に分散コンピューティングを行ないたい**
- **異なるアーキテクチャ間でデータ表現を統一して、やり取りを行ないたい**
- **明示的** (explicit) **ではない暗黙的** (implicit) **ネットワークプログラミングを行ないたい（下位層のソケット** API、XTI API **などは隠蔽して欲しい）**
- **オブジェクト指向プログラミングにおけるメッセージやメソッドのように、簡単に分散コンピューティングを行ないたい**

RPC(Remote Procedure Call) **の手法：**

- SunRPC → ONC RPC(on UDP, TCP) → ONC+ RPC(on TI:Transport Independent)
- XML-RPC(on HTTP) → SOAP: Simple Object Access Protocol(on HTTP, SMTP, FTP, etc)
- DCE-RPC: Distributed Computing Environment-Remote Procedure Calls, MS RPC
- DCOM:Distributed Component Object Model, Free DCE&DCOM
- CORBA: Common Object Request Broker Architecture,
  Java RMI:Remote Method Invocation, etc (via ORB)

SunRPC, ONC RPC とは

Sun Microsystems による RPC 実装：

- データ表現として外部データ表現 XDR: External Data Representation を定義
- ポートマッパ (portmap もしくは rpcbind) がサーバ上の遠隔手続きを管理
- NFS: Network File System, NIS(YP): Network Information Service, DRAC: Dynamic Relay Authorization Control など、特にサブネット内での RPC で利用
- NFS がホスト間でのファイル共有に早くから広く使われたため Unix では事実上の標準
- Windows でも SUA: Subsystem for Unix-based Application(aka SFU: Services for Unix) で Server 2003 R2 から標準搭載

NFS, NIS そのものの役割は他のディレクトリサービス、例えば、LDAP: Lightweight Directory Access Protocol や Open Directory(SASL: Simple Authentication and Security Layer, LDAP + NetInfo + /etc + NIS + Active Directory) に代わられつつある。

ONC+ RPC とは

- TLI:Transport Layer Interface によりトランスポート独立。さらに、IPv6 でも使用可
- マルチスレッドサーバ／クライアントに対応など

RPC 認証とは

1. Unix 認証 (AUTH_SYS, AUTH_UNIX)、DES 認証 (AUTH_DES) などが利用可能
2. さらに、Kerberos v5 認証 (AUTH_KERB) が利用可能の場合もある
3. また、RPCSEC_GSS API で柔軟で強固なセキュリティ管理が利用可能の場合もある

XDR（外部データ表現）とは

以下のような基本データ型を、異なるアーキテクチャで間で統一するためのインターフェース

- **void 型、論理型** (bool → bool_t)**、列挙型** (enum)
- **文字型** char, unsigned char → u_char
- **符号付き整数型** short, long, hyper(long long) → longlong_t
- **符号なし整数型** unsigned short → u_short, unsigned long → u_long, unsigned hyper(unsigned long long) → u_longlong_t
- **浮動小数点型** float, double, quadruple(long double)
- **固定長バイト列** opaque *var*[n] → char *var*[n]
- **可変長バイト列** opaque *var*<n> → struct { u_int *var*_len; char **var*_val; } *var*
- **文字列** string → char *
- **構造体** struct
- **共用体（但し、弁別型）**
  union *var* switch (enum typei) { case type0: *anytype v*; } →
  struct *var* { enum typei; union { *anytype v*; } *var*_u; }
- **基本データ型の固定長配列**
  anytype var[n] → *anytype var*[n]
- **基本データ型の可変長配列**
  anytype var<n> → struct { u_int *var*_len; *anytype* **var*_val; } *var*
- **定数型** const *var* = *anytype* → #define *var anytype* ***正確には** XDR **とは無関係**

rpcgen とは

```
$ man rpcgen
```

RPC プロトコルコンパイラ：C に似た RPC 言語から C コードを生成するツール

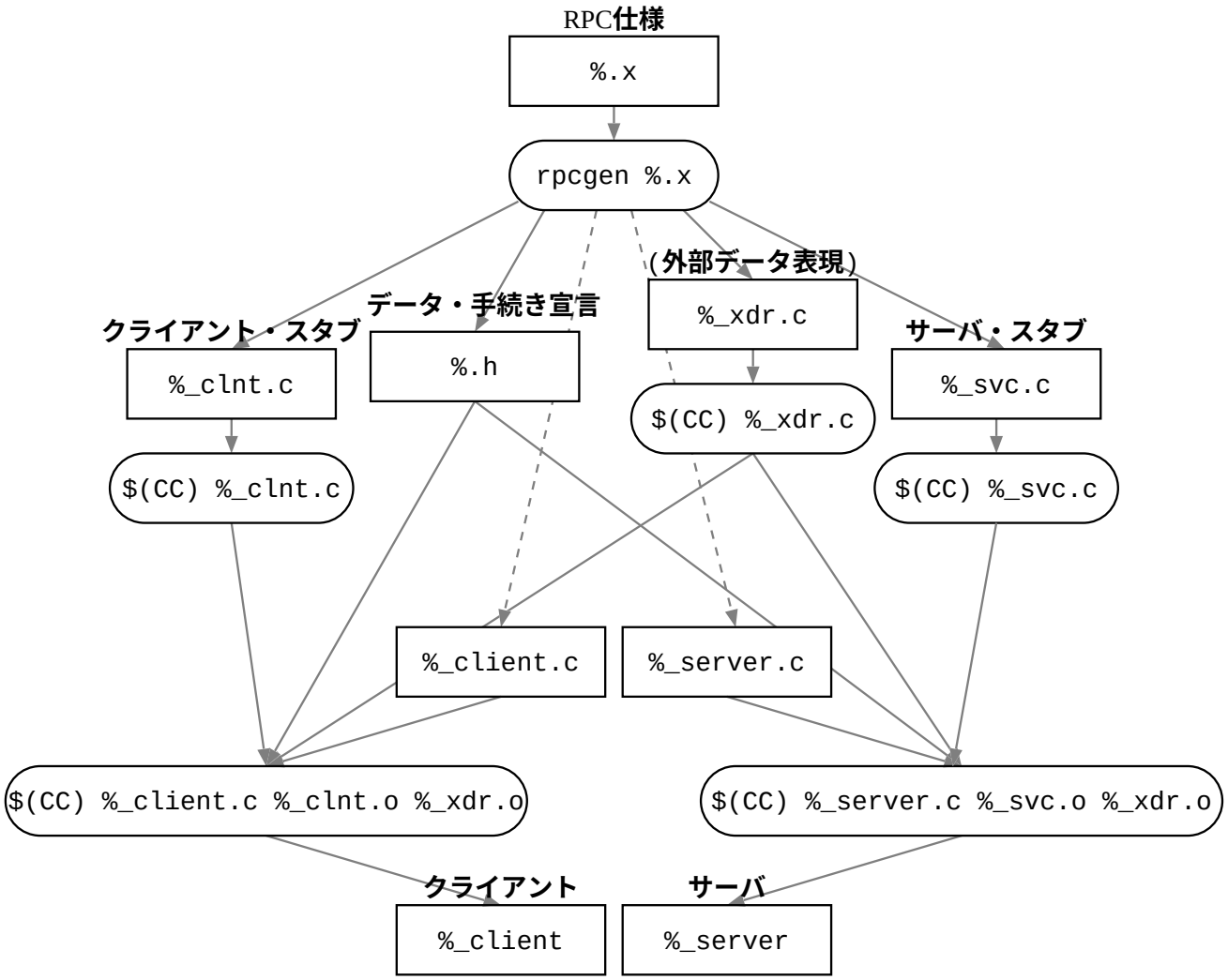
- ▷ -C オプションで ANSI C89 スタイルかつ C++対応の C コードを生成
- ▷ -a オプションでサンプルコードも生成
- ▷ -N オプションで複数の引数に対応したスタイルの C コードを生成
- ▷ -M オプションでマルチスレッド（ＭＴ）に対応したスタイルの C コードを生成

```
$ rpcgen -C -a test00.x
$ ls test00*
test00.h
test00_client.c
test00_clnt.c
test00_server.c
test00_svc.c

$ rpcgen -C -a test01.x
$ ls test01*
test01.h
test01_client.c
test01_clnt.c
test01_server.c
test01_svc.c
test01_xdr.c
```

rpcgen で生成されるＣコード

rpcgen で生成されるＣコードからクライアント／サーバプログラムまでのファイルとコマンド：



RPC 仕様ファイルの例

test_x/test00.x

```
program SOMEPROG {
  version SOMEVERS {
    double SOMETHINGPROC(double) = 1; /* proc# */
  } = 1; /* ver# */
} = 0x20000000; /* rpc prog# */
```

test_x/test01.x

```
struct vec_double {
  double d[2];
};
program SOMEPROG {
  version SOMEVERS {
    vec_double SOMETHINGPROC(vec_double) = 1; /* proc# */
  } = 1; /* ver# */
} = 0x20000000; /* rpc prog# */
```

test_x/test02.x

```
struct vec_double {
  double d[2];
};
program SOMEPROG {
  version SOMEVERS {
    /* 複数の引数 … rpcgen へ '-N' オプションが必要となる */
    vec_double SOMETHINGPROC(int, vec_double) = 1; /* proc# */
  } = 1; /* ver# */
} = 0x20000000; /* rpc prog# */
```

RPC 仕様ファイルの例（弁別型共用体）

test_x/test03.x

```
enum aftype {
  AFTYPE_INET = 1,
  AFTYPE_INET6 = 2
};
union address switch (enum aftype af) {
  case AFTYPE_INET:
    unsigned long s;
  case AFTYPE_INET6:
    unsigned char s6[16];
};

program ADDRESSPROG {
  version ADDRESSVERS {
    void ADDRESSSETPROC(address) = 1; /* proc# */
    address ADDRESSGETPROC(void) = 2; /* proc# */
  } = 1; /* ver# */
} = 0x20000000; /* rpc prog# */
```

test_x/test03.h **の一部**

```
struct address {
  aftype af;
  union {
    u_long s;
    u_char s6[16];
  } address_u;
};
```

クライアント側プログラム

クライアント側プログラムにて、自前で書く必要のあるコード、つまり、クライアント側スタブ手続きの使い方。
test_x/test01_client.c

```
        :
char *host = "localhost";
CLIENT *clnt;
vec_double *result_1;
vec_double somethingproc_1_arg;


/* クライアント・ハンドルの生成 */
clnt = clnt_create(host, SOMEPROG, SOMEVERS, "netpath"/* もしくは"udp","tcp","datagram_v"など */);
if (clnt == (CLIENT *)NULL) {
  clnt_pcreateerror(host); /* コネクション前用エラー出力 */
  //fprintf(stderr, "%s\n", clnt_spcreateerror(host)); /* これでも良い */
  exit(1);
}
      :
/* リモート手続き (somethingproc version 1) 呼出し */
result_1 = somethingproc_1(&somethingproc_1_arg, clnt);
if (result_1 == (vec_double *)NULL) {
  clnt_perror(clnt, "call failed"); /* コネクション後用エラー出力 */
  //fprintf(stderr, "%s", clnt_sperror(clnt, "call failed")); /* これでも良い */
}
      :
/* クライアント・ハンドルの破壊 */
clnt_destroy(clnt);
      :
```

クライアント側プログラム（複数の引数版）

クライアント側プログラムにて、自前で書く必要のあるコード、つまり、クライアント側スタブ手続きの使い方。
test_x/test02_client.c

```
        :
char *host = "localhost";
CLIENT *clnt;
vec_double *result_1;
int somethingproc_1_arg1;
vec_double somethingproc_1_arg2;

/* クライアント・ハンドルの生成 */
clnt = clnt_create(host, SOMEPROG, SOMEVERS, "netpath"/* もしくは"udp","tcp","datagram_v"など */);
if (clnt == (CLIENT *)NULL) {
  clnt_pcreateerror(host); /* コネクション前用エラー出力 */
  //fprintf(stderr, "%s\n", clnt_spcreateerror(host)); /* これでも良い */
  exit(1);
}
      :
/* リモート手続き (somethingproc version 1) 呼出し */
result_1 = somethingproc_1(somethingproc_1_arg1, somethingproc_1_arg2, clnt);
if (result_1 == (vec_double *)NULL) {
  clnt_perror(clnt, "call failed"); /* コネクション後用エラー出力 */
  //fprintf(stderr, "%s", clnt_sperror(clnt, "call failed")); /* これでも良い */
}
      :
/* クライアント・ハンドルの破壊 */
clnt_destroy(clnt);
      :
```

サーバ側プログラム

サーバ側プログラムにて、自前で書く必要のあるコード、つまり、サーバ側スタブ手続きの作り方。
test_x/test01_server.c

```
        :
vec_double *
somethingproc_1_svc(vec_double *argp, struct svc_req *rqstp)
{
  static vec_double result;

  /*
    ここに、*argp を入力、result を出力とする、なんらかのコードを書く事になる。
  */
  return &result;
}
        :
```

test_x/test02_server.c**（複数の引数版）**

```
        :
vec_double *
somethingproc_1_svc(int arg1, vec_double arg2, struct svc_req *rqstp)
{
  static vec_double result;

  /*
    ここに、argp1, argp2 を入力、result を出力とする、なんらかのコードを書く事になる。
  */
  return &result;
}
        :
```

クライアント側プログラム（ＭＴ版）

クライアント側プログラムにて、自前で書く必要のあるコード、つまり、クライアント側スタブ手続きの使い方。
`test_x/test01_client.c(-M` **オプション要**`)`

```
        :
    char *host = "localhost";
    CLIENT *clnt;
    enum clnt_stat retval_1;
    vec_double result_1;
    vec_double somethingproc_1_arg;

    /* クライアント・ハンドルの生成 */
    clnt = clnt_create(host, SOMEPROG, SOMEVERS, "netpath"/* もしくは"udp","tcp","datagram_v"など */);
    if (clnt == (CLIENT *)NULL) {
      clnt_pcreateerror(host); /* コネクション前用エラー出力 */
      //fprintf(stderr, "%s\n", clnt_spcreateerror(host)); /* これでも良い */
      exit(1);
    }
        :
    /* リモート手続き (somethingproc version 1) 呼出し */
    retval_1 = somethingproc_1(&somethingproc_1_arg, &result_1, clnt);
    if (retval_1 != RPC_SUCCESS) {
      clnt_perror(clnt, "call failed"); /* コネクション後用エラー出力 */
      //fprintf(stderr, "%s", clnt_sperror(clnt, "call failed")); /* これでも良い */
    }
        :
    /* クライアント・ハンドルの破壊 */
    clnt_destroy(clnt);
        :
```

サーバ側プログラム（ＭＴ版）

サーバ側プログラムにて、自前で書く必要のあるコード、つまり、サーバ側スタブ手続きの作り方。
`test_x/test01_server.c(-M` **オプション要**)

```
        :
bool_t
somethingproc_1_svc(vec_double *argp, vec_double *result, struct svc_req *rqstp)
{
  /*
    ここに、*argp を入力、*result を出力とする、なんらかのコードを書く事になる。
  */
  return retval;
}

int
someprog_1_freeresult(SVCXPRT *transp, xdrproc_t xdr_result, caddr_t result)
{
  xdr_free(xdr_result, result);
  /*
    ここに必要ならば、さらに解放に関するコードを書く。
  */
  return (TRUE);
}
        :
```

SunRPC **は極めて枯れた技術であるため、複数の引数、マルチスレッド対応について特段のメリットが得られるわけではない場合には** `-N, -M` **オプションを付けずに済ませられるようにコーディングすれば、広い可搬性を担保できる。**

RPC プログラム番号と rpcinfo コマンド

RPC プログラム番号

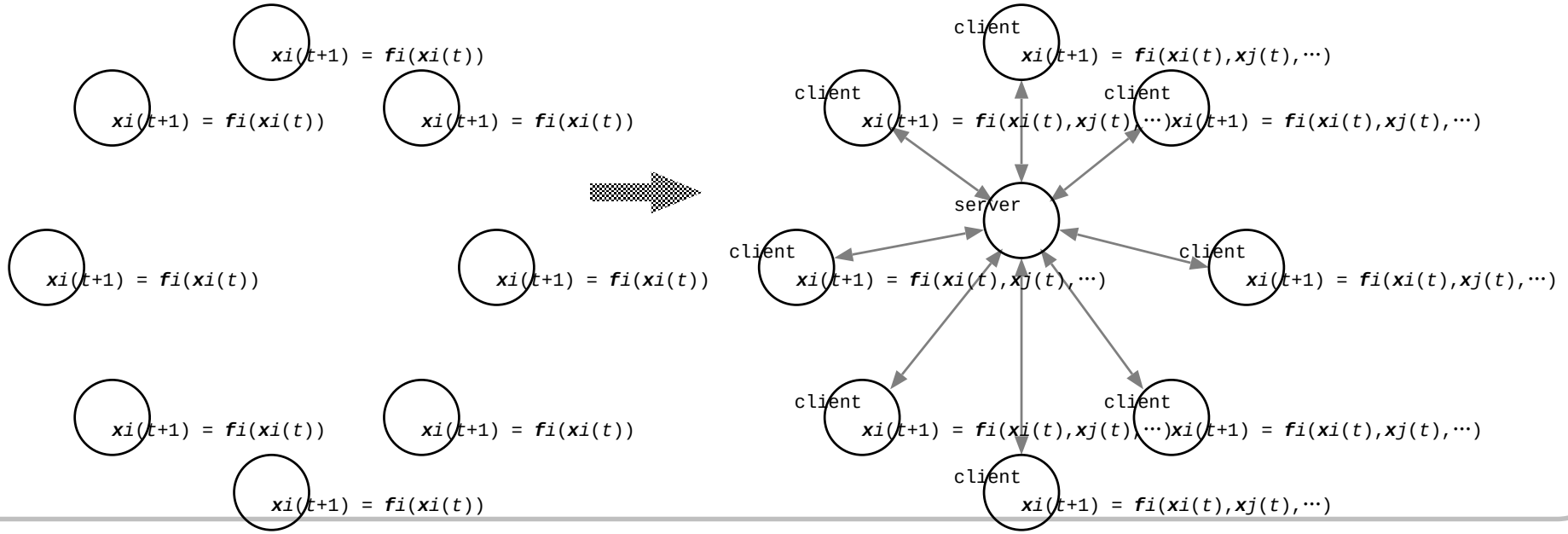
プログラム番号（16 進数、10 進数）の範囲		用途
0x00000000 〜 0x1fffffff	0 〜 536,870,911	Sun Microsystems による定義
0x20000000 〜 0x3fffffff	536,870,912 〜 1,073,741,823	ユーザ定義
0x40000000 〜 0x5fffffff	1,073,741,824 〜 1,610,612,735	システム開発用過渡的定義
0x60000000 〜 0xffffffff	1,610,612,736 〜 4,294,967,295	予約

```
$ man rpcinfo

$ rpcinfo -p
   program vers proto   port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
 536870912    1   udp  49562
 536870912    1   tcp  51416
```

（例題１）ONC RPC による分散型計算システム

まず、$k(k=2)$ 個の実数値が時間とともに変化する系があるものとする。その系ひとつを１クライアントとして動作させ、ひとつのサーバを介して、$m(m=16$ を最大とする$)$ 個のクライアントが互いに値の変化を授受できるような分散型計算システムを ONC RPC を使って構築せよ。但し、系はとりあえずは一様乱数でよい。また、Unix 認証に対応した版も作成せよ。さらに、トランスポート層を選べるようにせよ。



（例題１）の解答例

ex00some/some.x

```
const maxstate = 16;
struct state {
  double v[2];
};
struct some {
  unsigned int id;
  string name<64>;
  state st;
};
struct thing {
  unsigned int id;
  state sts<maxstate>;
};

program SOMEPROG {
  version SOMEVERS {
    thing SOMETHINGPROC(some) = 1; /* proc# */
  } = 1; /* ver# */
} = 0x20000000; /* rpc prog# */
```

```
$ rpcgen -C some.x
```

（例題１）の解答例（つづき）

ex00some/some.h **の主要部分 〜 生成されたヘッダファイルをまず見よ！**

```
#define maxstate 16

struct state {
  double v[2];
};
typedef struct state state;

struct some {
  u_int id;
  char *name;
  state st;
};
typedef struct some some;

struct thing {
  u_int id;
  struct {
    u_int sts_len;
    state *sts_val;
  } sts;
};
typedef struct thing thing;

#define SOMEPROG ((u_long)0x20000000)
#define SOMEVERS ((u_long)1)
#define SOMETHINGPROC ((u_long)1)

thing * somethingproc_1(some *, CLIENT *);              /* クライアント側スタブ手続き */
thing * somethingproc_1_svc(some *, struct svc_req *);  /* サーバ側スタブ手続き */
```

(**補足**) **現時点では** `char *name` **は特に使っていない。**

（例題１）の解答例（つづき）

ex00some/somec.c の主要部分 〜 クライアントプログラムを書く！

```
        :
#include "some.h"
#define irandom(lb, ub) ((lb) + (random()%((ub) - (lb) + 1)))
int main(int argc, char *argv[])
{
  CLIENT *clnt;
  char *host = "localhost";
  some s = { /* .id = */ 0, "" /* NULL だと境界違反になるので注意！ */, };
  thing *t;
  struct timespec ts = { /* .tv_sec = */ 0, /* .tv_nsec = */ 10000000 /* =10 ミリ秒 */ };
  int i, a;
        :
  for (i=0; i<sizeof(s.st.v)/sizeof(s.st.v[0]); i++) /* 自身の状態の初期化 */
    s.st.v[i] = irandom(1, 6);
  if (!(clnt = clnt_create(host, SOMEPROG, SOMEVERS, "udp"))) { /* クライアント・ハンドルの生成 */
    fprintf(stderr, "%s\n", clnt_spcreateerror(host));
    exit(1);
  }
  while (!0) {
    if (!(t=somethingproc_1(&s, clnt))) {
      fprintf(stderr, "%s", clnt_sperror(clnt, "somethingproc_1"));
      exit(1);
    }
    if (s.id == 0 && t->id != 0) /* id(>0) が始めて割り当てられたとき */
      s.id = t->id;
    printf("%u:%g,%g ", s.id, s.st.v[0], s.st.v[1]);    /* 自身の状態を表示 */
    for (i=0; i<t->sts.sts_len; i++)                    /* 他者の状態を表示 */
      printf("%u:%g,%g ", i, t->sts.sts_val[i].v[0], t->sts.sts_val[i].v[1]);
    printf("\n");
    nanosleep(&ts, NULL); /* 適度なウェイトを入れる */
    for (i=0; i<sizeof(s.st.v)/sizeof(s.st.v[0]); i++)  /* 自身の状態を更新 */
      s.st.v[i] = irandom(1, 6);
  }
  clnt_destroy(clnt); /* クライアント・ハンドルの破壊 */
  return 0;
}
```

（例題１）の解答例（つづき）

ex00some/somed.c ～ サーバプログラムを書く！

```
#include "some.h"

struct somed {
  int n;                        /* 割り当てたクライアント数 */
  state states[maxstate];       /* すべてクライアントの状態 */
} me = { 0, };

thing *somethingproc_1_svc(some *s, struct svc_req *rqstp)
{
  static thing t = { 0, };

  /*
   以下が some *s を入力、thing t を出力とするコード
  */
  if (!s->id && !(me.n < sizeof(me.states)/sizeof(state)))
    return &t;
  /* me.n が maxstate を越えないなら新規参入 (id == 0) について id を割り当てる。さもなくば return */
  if (!s->id)
    t.id = ++me.n;
  else
    t.id = s->id;
  me.states[t.id-1] = s->st; /* 配列 states の添字は id-1 としている */
  t.sts.sts_len = me.n;
  t.sts.sts_val = me.states;
  return &t; /* クライアント数とすべてクライアントの状態が返る */
}
```

（例題１）の解答例（つづき）

ex00some/makefile ～ **汎用的な** GNU `makefile` **を一度書いておくとラク！**

```
RPC_X=some.x
RPC_GENS=\
$(RPC_X:.x=.h)  \
$(RPC_X:%.x=%_xdr.c)    \
$(RPC_X:%.x=%_svc.c)    \
$(RPC_X:%.x=%_clnt.c)   \

RPC_SRCS=$(filter %.c,$(RPC_GENS))
RPC_OBJS=$(RPC_SRCS:.c=.o)
RPCD=$(RPC_X:%.x=%d)
RPCC=$(RPC_X:%.x=%c)

all: $(RPC_GENS) $(RPC_OBJS) $(RPCD) $(RPCC)

%.h %_xdr.c %_svc.c %_clnt.c: %.x
        rpcgen -C $<
$(RPCD): $(RPC_X:%.x=%d.c) $(RPC_X:%.x=%_xdr.o) $(RPC_X:%.x=%_svc.o)
$(RPCC): $(RPC_X:%.x=%c.c) $(RPC_X:%.x=%_xdr.o) $(RPC_X:%.x=%_clnt.o)

clean:
        rm -f $(RPC_GENS) $(RPC_OBJS) $(RPCD) $(RPCC)
```

```
$ make CC=gcc LDLIBS='-lnsl -lrt'
gcc -c -o some_xdr.o some_xdr.c
gcc -c -o some_svc.o some_svc.c
gcc -c -o some_clnt.o some_clnt.c
gcc somed.c some_xdr.o some_svc.o  -lnsl -lrt -o somed
gcc somec.c some_xdr.o some_clnt.o -lnsl -lrt -o somec
$ ./somed
$ ./somec
```

（例題１）の解答例（Unix 認証対応版）

ex01some/somec.c の主要部分

```
    if (!(clnt = clnt_create(host, SOMEPROG, SOMEVERS, "tcp"))) { /* クライアント・ハンドルの生成 */
      fprintf(stderr, "%s\n", clnt_spcreateerror(host)); exit(1);
    }
    auth_destroy(clnt->cl_auth); /* まず既定の認証情報 (AUTH_NONE) を破壊 */
    clnt->cl_auth = authunix_create_default(); /* Unix 認証情報を取得して、クライアント・ハンドルに割り当て */
    /*{ // 上記は以下とほぼ等価である
      char hostname[256];
      gethostname(hostname, sizeof(hostname));
      clnt->cl_auth = authunix_create(hostname, getuid(), getgid(), 0, NULL);
    }*/
```

ex01some/somed.c の主要部分

```
  thing *somethingproc_1_svc(some *s, struct svc_req *rqstp)
  {
    struct authunix_parms *au;

    switch (rqstp->rq_cred.oa_flavor) { /* Unix 認証情報の使用例、表示してるだけ */
    case AUTH_NONE: printf("AUTH_NONE:\n"); break;
    case AUTH_UNIX:
      au = (struct authunix_parms *)rqstp->rq_clntcred;
      printf("AUTH_UNIX: host %s, uid %ld, gid %ld\n",
             au->aup_machname, (long)au->aup_uid, (long)au->aup_gid);
      break;
    }
```

（考察）このように Unix 認証は、単に uid, gid を渡しているだけで、アクセス制限をサポートするものではないことがわかる。セキュリティ管理をするにはさらに他の手段を検討する、ホストアドレス制限を検討する、などが必要となる。

（例題１）の解答例（トランスポート層選択対応版）

ex02some/somec.c **の主要部分**

```
        :
    char *host = "localhost", *proto = "netpath";
        :
    int i, a;

    for (a=1; a<argc; a++)
      if (strcmp(argv[a], "-h") == 0) {
        fprintf(stderr, "usage:\n");
        fprintf(stderr, "\t%s [-h] [-s server] [-p nettype]\n", argv[0]);
        exit(1);
      }
      else if (strcmp(argv[a], "-s") == 0 && a+1 < argc)
        host = argv[++a];
      else if (strcmp(argv[a], "-p") == 0 && a+1 < argc)
        proto = argv[++a];
        :
    if (!(clnt = clnt_create(host, SOMEPROG, SOMEVERS, proto))) {
      fprintf(stderr, "%s\n", clnt_spcreateerror(host));
      exit(1);
    }
        :
```

```
$ ./somec -p tcp
$ ./somec -p circuit_v
```

ONC+ RPC, TLI, netconfig, TI-RPC

```
$ man rpc
$ man netconfig
```

char *proto = "nettype"の nettype に指定できるトランスポート名：

nettype	説明
netpath	NETPATH 環境変数依存、NETPATH が未定義なら visible
visible	/etc/netconfig で visible フラグ (v) が設定されているトランスポートから選択
datagram_v	/etc/netconfig でコネクションレスデータグラムトランスポート (tpi_clts) かつ visible から選択
circuit_v	/etc/netconfig で接続指向 (順序なし｜順序付き) トランスポート (tpi_cots|tpi_cots_ord) かつ visible から選択
datagram_n	/etc/netconfig でコネクションレスデータグラムトランスポート (tpi_clts) かつ netpath から選択
circuit_n	/etc/netconfig で接続指向 (順序なし｜順序付き) トランスポート (tpi_cots|tpi_cots_ord) かつ netpath から選択
udp	UDP/IPv4
tcp	TCP/IPv4

```
$ less /etc/netconfig
udp6       tpi_clts      v    inet6    udp    /dev/udp6       -
tcp6       tpi_cots_ord  v    inet6    tcp    /dev/tcp6       -
udp        tpi_clts      v    inet     udp    /dev/udp        -
tcp        tpi_cots_ord  v    inet     tcp    /dev/tcp        -
rawip      tpi_raw       -    inet     -      /dev/rawip      -
ticlts     tpi_clts      v    loopback -      /dev/ticlts     straddr.so
ticotsord  tpi_cots_ord  v    loopback -      /dev/ticotsord  straddr.so
ticots     tpi_cots      v    loopback -      /dev/ticots     straddr.so
```

RPC プログラミング：まとめ

単純な分散型計算システムの作成などを通して、

- rpcgen の使用方法
- ONC RPC のクライアント側通信スタブ手続きの使い方
- ONC RPC のサーバ側通信スタブ手続きの作り方
- ONC RPC の Unix 認証の使い方
- 複数の引数、マルチスレッド、TI-RPC などの話題と実際

参考文献：

- 「TCP/IP によるネットワーク構築 Vol.III〜クライアントサーバプログラミングとアプリケーション（村井 純 他 訳、共立出版）」
- 「UNIX ネットワークプログラミング Vol.2〜IPC:プロセス間通信（篠田 陽一 訳、ピアソン・エデュケーション）」
- 「ONC+ 開発ガイド（Sun Microsystems）」
  `http://docs.sun.com/app/docs/doc/816-3978`

その他のリソース：

`abmail/libexec/ - http://www.aihara.co.jp/~taiji/abmail/`

では、IPv4,IPv6 アドレスレンジデータベースの管理・検索を ONC+ RPC で実現し、RPC のセキュリティ管理としてホストアドレス制限を採用している。