Brent

Sun RPC Software Architecture

Bob Lyon

Company Confidential

ABSTRACT

This document discusses the layering of NDR, RPC, Authentication and various transport methods in Sun's RPC environment. Actual code is presented as examples.

Introduction

The document, along with some code, gives concrete examples of the principles described in "Sun Network Data Representation and Remote Procedure Call". The more important principles are worth repeating here. They are:

1. NDR is not a protocol. It is a set rules that describe how basic data types are represented in stream of bits and how compound data types are constructed from the basic data types.

2. RPC is a protocol. In fact RPC is an open ended family of protocols. Each member of the family implements RPC on a different transport protocol. This paper discusses an implementation of RPC/UPD/IP. RPC/TCP/IP would be easy to implement.

3. Authentication was placed at RPC level so that issues can be addressed once by a central authority. (Authentication could have left up to individual service protocols which would have encouraged many similar, but not necessarily compatible versions authentication parameters.)

4. NDR and RPC are language independent, OS independent and machine independent. However, the rest of this document discusses an implementation of RPC that is tailored for the C/Unix environment. The code executes on both the 68000 and Vax families of machines.

The Big (Simple) Picture

The essence of procedure calling is that a well-known execution operates on a copy of the callers data (arguments) and produces a copy of the execution's output (results). In a C program, the binder magically links well known execution (routines) with the callers. The process's stack is used for copying arguments from caller to callee and copying results from callee to caller.

The stack is in effect, a communication area between caller and callee. The C compiler makes various guarantees about data placed on the stack such that caller and callee have consistent notations about the value of the data. The order that data is pushed onto (pulled off of) the stack depends upon the order that the C programmer lists his parameters (arguments).

Remote procedure calling captures the same essence as local procedure calling. It is made harder because (1) callers cannot be staticly bound to routines before run time, (2) communication of arguments and results are not as simple as stack manipulation, and (3) convenient short hand notations (like pointers) must be expanded to their real representations.

The NDR package translates C and machine specific data to (from) a standard representation for that data. The RPC package attempts to transfer the data between caller and callee and vise versa. The Authentication package is a wart that hangs on the side of the RPC package. It authenticates caller to callee and vice versa.

The Sun NDR Package (ndr.h)

The package is a tri-directional serializer. The directions are encode, decode and free. Encoded (decoded) data is streamed out to (in from) a ndr-stream which is specified by a set of primitive operation routines passed in a structure. These routines allow clients to create new flavor streams as desired without affecting the ndr package. ndr_stdio.h provides ndr_streams to and from the standard io package. ndr_mem.h allows ndr_streams to and from Unix process memory.

The ndr package provides a set of primitives that allow the client to translate between "basic" C data types and the standard representations. All ndr routines have the form:

```
ndr_xxxx(ndrs, data_ptr)
        NDR_STREAM *ndrs;
        <type> *data_ptr;
{
}
```

(Some primitives have more parameters.) ndrs is a pointer to a structure that contains the ndr-stream routines and a variable that contains the direction of the serialization. Typical clients will not need to worry about the construction of ndrs; they will merely pass the parameter off to primitive routines. Basic primitives include ndr_int, ndr_u_int, ndr_enum, ndr_long, ndr_u_long, ndr_short, ndr_u_short, and ndr_string. ndr_string takes a third parameter, maxsize which specifies the maximum allowable string length.

The client can then write his own ndr_routines for his particular data structures. The body of his routines would consist of calls to the ndr primitives and possibly other packages' ndr_routines. Two points are worth emphasizing: 1) the non-primitive ndr routines do not need to know about the direction of the serialization - they merely call primitive ndr routines, and 2) ndr routines made be recursive. An important aspect of the ndr design is that client programmer should be able to code their ndr data representation routines as quickly as they code their C structures.

Other, more complicated primitives provided are ndr_array, ndr_union, and ndr_reference.

→ try_ndr.c give an example of serializing (argc, argv) to stdout.

Now that the hardest part of RPC usage (data [en | de]coding) has been covered, on to the easy parts.

Client Authentication (auth.h)

The authentication package is very transparent to the normal rpc client except for (1) creation of a authentication session, and (2) recovery from authentication errors. Authentication errors are enumerated by the type AUTH_STATUS. The structure AUTH_CLIENT is declared so that the user may declare (or allocate) it, get it initialized by an appropriate authentication implementation, and finally pass it to the rpc package.

There will ultimately be a family of authentication implementations, each providing a different level of security. Currently, there only exists an authentication implementation that passes degenerate authentication parameters (no security). It is accessed via auth_none.h.

RPC Client Package (rpc.h)

Like authentication, rpc is mostly transparent. RPC_HANDLE is declared and initialized by the client. The handle contains information about the transport method, the remote host's address, and the program and version numbers of the desired procedures. The other types in rpc.h are used to convey status or remote procedure calling.

Remote procedures are called via the defined macro "rpc"; its usage is:

```
RPC_STATUS
rpc(handle, procedure, x_id, ndr_args, ndr_ptr, ndr_results, results_ptr)
    RPC_HANDLE *handler;
    u_long procedure;
    u_long *x_id;
    NDR_PROC ndr_args;
    caddr_t ndr_ptr;
    NDR_PROC ndr_results;
    caddr_t *results_ptr;
```

x_id is a pointer to a transaction id; if it is zero then it gets a random value. The arguments to the procedure are located at core address ndr_ptr and will be serialized (at the appropriate moment) by the routine ndr_args. If and when the remote call returns, the results will be placed at core address results_ptr; they will be deserialized by the routine ndr_results.

Currently, there is only one transport method implemented for rpc; it is accessed via rpc_udp.h. The transport method is UPD/IP, an unreliable protocol. (However, with reasonable use of transaction ids and a small cache at the "server" side, execute-at-most-once semantics can almost be guaranteed.) Creating a UPD base rpc handle involves passing rpcudp_create the following parameters: a target RPC_HANDLE, a remote address, a program and a version number, the number of "tries" (packet retransmissions) per call, the amount of time to wait between tries, and a pointer to a socket file number.

The authentication and rpc aspects of remote procedure calling are not symmetric between client (caller) and server (callee). Luckily, the details of authentication are as transparent to the service implementer as they are to the service client.

The remote program implementation works something like the following:

1. (Begin loop) a process waits on a socket for a packet to arrive.

2. When a packet arrives, the rpc service machinery verifies that it contains a well formed rpc call message. If not, then it loops. If so, it extracts important information like the authentication parameters, the program number, version number and procedure number.

3. The authentication parameters are handed to the authentication package for validation. If they are not valid, then an error message is sent to the caller and the loop is repeated.

4. The rpc machinery then tries to match the program and version numbers with those previously registered by a specific service. If there is no match, then an error message is returned to the caller and the loop is repreated.

5. Finally the specified service routine is called. It is passed the procedure number, a set of routines used for deserializing the parameters or serializing the results, and the identity of the caller (as determined by the validated authentication parameters).

6. When the service routine returns the loop is repreated.

The life of a service has three phases: first it is registered with the rpc service package; second, the package and all registered services are activated (the loop executes); finally, the services are unregistered.

The basic mechanism of the rpc service loop must be willing to share the ownership of the process with other software packages. This is because the Unix process may wish to wait for events that are independent from the call-message-arrived event.

Acknowledgements

The tri-directional ndr concept were first implemented at Xerox SDD. Transport independent rpc was proposed by the author and was rejected by Xerox.

Sun's ndr/auth/rpc implementation was done by the author. Dan Walsh and Bill Joy greatly influenced the software structure and the client interfaces.

Attachments

All header and source files are presented, including those that test or time various pieces of the rpc system. The code is currently available on pumpkinseed:~blyon/wnjrpc/*. The document is available on pumpkinseed:~blyon/rpc/memos/rpc_arch.txt.

```
CFLAGS = -O

all:    mallocheap.o                                                    \
        ndr.o ndr_stdio.o ndr_mem.o                                     \
        auth_none.o                                                     \
        rpc.o rpc_udp.o                                                 \
        try_ndr time_newndr time_raw_rpc rpc_client

mallocheap.o:   mallocheap.c types.h heap.h mallocheap.h

ndr.o:          ndr.c types.h heap.h ndr.h
ndr_stdio.o:    ndr_stdio.c types.h heap.h ndr.h ndr_stdio.h
ndr_mem.o:      ndr_mem.c types.h heap.h ndr.h ndr_mem.h

auth_none.o:    auth_none.c types.h auth.h auth_none.h

rpc.o:          rpc.c types.h heap.h ndr.h auth.h rpc.h rpc_msg.h
rpc_udp.o:      rpc_udp.c types.h heap.h mallocheap.h ndr.h ndr_mem.h  \
                auth.h rpc.h rpc_msg.h rpc_udp.h

try_ndr.o:      try_ndr.c types.h heap.h mallocheap.h ndr.h ndr_stdio.h
time_newndr.o:  time_newndr.c types.h heap.h mallocheap.h ndr.h ndr_mem.h
time_raw_rpc.o: time_raw_rpc.c types.h heap.h ndr.h ndr_mem.h          \
                auth.h rpc.h rpc_msg.h
rpc_client.o:   rpc_client.c types.h heap.h mallocheap.h ndr.h ndr_mem.h\
                auth.h auth_none.h rpc.h rpc_udp.h

try_ndr:        try_ndr.o mallocheap.o ndr.o ndr_stdio.o
        cc -o try_ndr try_ndr.o mallocheap.o ndr.o ndr_stdio.o

time_newndr:    time_newndr.o mallocheap.o ndr.o ndr_mem.o
        cc -o time_newndr time_newndr.o mallocheap.o ndr.o ndr_mem.o

time_raw_rpc:   time_raw_rpc.o ndr.o ndr_mem.o rpc.o
        cc -o time_raw_rpc time_raw_rpc.o ndr.o ndr_mem.o rpc.o

rpc_client:     rpc_client.o mallocheap.o ndr.o ndr_mem.o auth_none.o rpc.o
        cc -o rpc_client rpc_client.o mallocheap.o ndr.o ndr_mem.o      \
                auth_none.o rpc.o rpc_udp.o
```

```
/*      %M%     %I%     %E%     */

/*
 * Proposed additions to <sys/types.h>
 */

/* define needed due to 1.0 compiler bug */
#define VOID int
typedef enum bool_t { FALSE=(1==2), TRUE=(1==1) } bool_t;
typedef enum enum_t { __dontcare__ } enum_t;

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

```
/*      %M%     %I%     %E%     */

/*
 * Storage Management Routines.
 *
 * Copyright (C) 1984, Sun Microsystems, Inc.
 */

/*
 * Heaps provide the general storage mechanisms previously provided by
 * malloc and free and their ilk.
 *
 * Currently, only a toy implementation is provided.
 */

typedef struct heap {
        struct {
                caddr_t (*_makenode)(); /* storage allocator */
                VOID    (*_freenode)(); /* storage freer */
                VOID    (*_destroy)();
        } ops;
} HEAP;

#ifndef noheapmacros
#define makenode(hp, bytesize)                                  \
        (*(hp->ops._makenode))(hp, bytesize)
#define freenode(hp, node)                                      \
        (*(hp->ops._freenode))(hp, node)
#endif
```

```
/*      %M%     %I%     %E%     */

/*
 * Storage Management Routines.
 *
 * Copyright (C) 1984, Sun Microsystems, Inc.
 */

/*
 * Connects UNIX malloc and free the HEAP mechanism.
 */

HEAP *make_unix_heap();  /* takes no parameters */
```

```
/*      %M%     %I%     %E%     */

/*
 * Storage Management Routines.
 *
 * Copyright (C) 1984, Sun Microsystems, Inc.
 */

/*
 * Connects UNIX malloc and free the HEAP mechanism as stated in mallocheap.h.
 */

#include "types.h"
#define nomacros 1
#include "heap.h"
#include "mallocheap.h"
caddr_t malloc();
free();

VOID
noop_destroy() {}

VOID
unixfree(hp, ptr)
        HEAP *hp;
        caddr_t ptr;
{
        free(ptr);
}

caddr_t
unixmalloc(hp, size)
        HEAP *hp;
        u_int size;
{
        return (malloc(size));
}

HEAP unixheap;

HEAP *
make_unix_heap()
{
        unixheap.ops._makenode = unixmalloc;
        unixheap.ops._freenode = unixfree;
        unixheap.ops._destroy = noop_destroy;
        return (&unixheap);
}
```

```
/*      %M%     %I%     %E%     */

/*
 * Network Data Representation Serialization Routines.
 *
 * Copyright (C) 1984, Sun Microsystems, Inc.
 */

/*
 * NDR provides a conventional way for converting between C data
 * types and an external bit-string representation.  Library supplied
 * routines provide for the conversion on built-in C data types.  These
 * routines and utility routines defined here are used to help implement
 * a type encode/decode routine for each user-defined type.
 *
 * Each data type provides a single procedure which takes two arguments:
 *
 *      bool_t
 *      ndrproc(ndrs, argresp)
 *              NDR_STREAM *ndrs;
 *              <type> *argresp;
 *
 * The first argument is an instance of a NDR_STREAM, to which or from
 * which the data type is to be converted.  The second argument is
 * a pointer to the structure to be converted to or from the
 * NDR format or freed.  The NDR_STREAM contains an operation field
 * which indicates which of these operations are to be performed.
 *
 * The operation is included as part of a NDR_STREAM to speed things up.
 * This is a case where dynamic scope on ndrs and op would be perfect.
 * We write only one procedure per data type to make it easy
 * to keep the encode and decode procedures for a type consistent.  In many
 * cases the same code performs all operations on a user defined type,
 * because all the hard work is done in the component type routines.
 * decode as a series of calls on the nested data types.
 *
 * Not using a NDR protocol compiler at this level allows us to choose
 * when to trade space for time and use inline expansions of the encode
 * or decode procedures.  It also allows us to switch from using ndr to
 * using lower-level instance-specific primitives on the underlying data.
 * Thus we can mix use of NDR with standard i/o calls when using ndr_stdio.h.
 */

/*
 * Ndr operations.  NDR_ENCODE causes the type to be encoded into the
 * stream.  NDR_DECODE causes the type to be extracted from the stream.
 * NDR_FREE can be used if a NDR_DECODE creates a structure which has
 * nested substructure allocated with malloc() that should be free()'d.
 * Actually, user specific allocators and frees can be passed by the user.
 */
typedef enum ndr_op { NDR_ENCODE=0, NDR_DECODE=1, NDR_FREE=2 } NDR_OP;

/*
 * A NDR_PROC exists for each data type which is to be encoded or decoded.
 *
 * The second argument to the NDR_PROC is a pointer to an opaque pointer.
```

```
 * The opaque pointer generally points to a structure of the data type
 * to be decoded.  If this pointer is 0, then the type routines should
 * allocate dynamic storage of the appropriate size and return it.
 */
#ifdef cplus
bool_t  (*NDR_PROC)(NDRSTREAM *, caddr_t *);
#endif
typedef bool_t (*NDR_PROC)();

/*
 * Structure per NDR stream.
 * Contains operation which is being applied to the stream,
 * an operations handle to the paticular implementor (e.g. see ndr_mem.h
 * and ndr_stdio.h), and two private fields for the use of the
 * particular impelementation.  We have two private fields so that
 * ndr_mem doesn't need to allocate storage, a substantial savings
 * for this common case.
 *
 * The final buf pointer field is used after a call to inline() and
 * points to core containing the inline data.  This is described
 * more below.
 */
typedef struct ndr_stream {
        NDR_OP  op;                     /* operation; fast additional param */
        HEAP *hp;                       /* client supplied memory manager */
        struct ndr_ops {
                bool_t  (*getlong)();   /* get a long from underlying stream */
                bool_t  (*putlong)();   /* put a long to " */
                bool_t  (*getbytes)();  /* get some bytes from " */
                bool_t  (*putbytes)();  /* put some bytes to " */
                u_int   (*getpostn)();  /* returns bytes off from beginning */
                bool_t  (*setpostn)();  /* lets you reposition the stream */
                bool_t  (*inline)();    /* buf quick ptr to buffered data */
                VOID    (*endinline)(); /* end inline */
                VOID    (*destroy)();   /* free privates of this ndr_stream */
        } *ops;
        caddr_t private;                /* pointer to private data */
        caddr_t base;                   /* private used for position info */
        int     handy;                  /* extra private word */
        long    *buf;                   /* used during inlines */
} NDR_STREAM;

#ifndef nomacros
#define ndr_getposition(ndrs)                                   \
        (*(ndrs)->ops->getpostn)(ndrs)
#define ndr_setposition(ndrs, pos)                              \
        (*(ndrs)->ops->setpostn)(ndrs, pos)
#define ndr_inline(ndrs, len)                                   \
        (*(ndrs)->cps->inline)(ndrs, len)
#define ndr_endinline(ndrs)                                     \
        if ((ndrs)->ops->endinline))                            \
                (*(ndrs)->ops->endinline)(ndrs)                 \
        else                                                    \
                (ndrs)->buf = 0;
#define ndr_destroy(ndrs)                                       \
        if ((ndrs)->ops->destroy))                              \
```

```
                (*(ndrs)->ops->destroy)(ndrs)
#endif

/*
 * Predefined procedures operating with primitive data types on a NDR_STREAM.
 *      define ASSUMPTIONS:
 *      1) sizeof and bit representations of (short, long, unspecified)
 *              ints are the same of their unsigned counterparts.
 *      2) enums are ints.
 *      3) (The Preprocessor does not know C):
 *              sizeof(int) == sizeof(long int).  Therefore, the primitves
 *              ndr_int and ndr_u_int must be re#defined is this moves
 *              to a machine like the 11/70.
 */
#ifdef cplus
bool_t  ndr_null(NDR_STREAM *, caddr_t);  /* always returns TRUE */
bool_t  ndr_int(NDR_STREAM *, int *);
bool_t  ndr_u_int(NDR_STREAM *, unsigned *);
bool_t  ndr_bool(NDR_STREAM *, bool_t *);
bool_t  ndr_enum(NDR_STREAM *, enum_t *);
bool_t  ndr_long(NDR_STREAM *, long *);
bool_t  ndr_u_long(NDR_STREAM *, u_long *);
bool_t  ndr_short(NDR_STREAM *, short *);
bool_t  ndr_u_short(NDR_STREAM *, u_short *);
#endif
bool_t  ndr_null(), ndr_long(), ndr_short(), ndr_u_short(), ndr_bool();
#ifndef nomacros
#define ndr_int(ndrs, ip)       ndr_long(ndrs, (long *)ip)
#define ndr_u_int(ndrs, up)     ndr_long(ndrs, (long *)up)
#define ndr_enum(ndrs, ep)      ndr_int(ndrs, (long *)ep)
#define ndr_u_long(ndrs, ulp)   ndr_long(ndrs, (long *)ulp)
#else
bool_t  ndr_int(), ndr_u_int, ndr_enum(), ndr_u_long();
#endif

/*
 * Predefined procedures for operating on array data types.
 * 2nd parameter references pointer to storage.
 *      If it is null, then routine malloc's necessary storage.
 * 3rd parameter receives number of elements in resulting array.
 * 4th parameter gives maximum element-length of the array.
 *
 * For ndr_array, last parameter is routine to unpack each element.
 * This is similar to a lisp map function.  The last u_int is the
 * sizeof of arrays' elements.
 *
 * ndr_bytes represents the array of bytes as a network string.
 */
#ifdef cplus
bool_t  ndr_array(NDR_STREAM *, caddr_t *, u_int*, u_int, u_int, NDR_PROC);
bool_t  ndr_bytes(NDR_STREAM *, caddr_t *, u_int *, u_int);
#endif
bool_t  ndr_array(), ndr_bytes();

/*
 * ndr_opaque allows the specification of a fixed size sequence of
```

```
 * opaque bytes.  The 2nd parameter points to the opaque object and the
 * 3rd gives its byte length.
 */
#ifdef cplus
bool_t  ndr_opaque(NDR_STREAM *, caddr_t, u_int);
#endif
bool_t  ndr_opaque();

/*
 * ndr_string deals with "C strings" - arrays of bytes that are
 * terminated by a NULL character.  The 2nd parameter references
 * pointer to storage; If pointer is null, then routine malloc's
 * necessary storage.  The last parameter is the max allowed length
 * of the string as specified by the protocol.
 */
#ifdef cplus
bool_t  ndr_string(NDR_STREAM *, char **, u_int);
#endif
bool_t  ndr_string();

/*
 * Support routine for discriminated unions.
 * You create an array of ndrdiscrim structures, terminated with
 * a entry with a null procedure pointer.  The routine gets
 * the discriminant value and then searches the array of structures
 * looking for a procedure to unpack the discriminant.  If there is
 * no specific routine, then a default routine may be called.
 * If there is no specific or default routine it is an error.
 */
#define NULL_NDR_PROC ((NDR_PROC)0)
typedef struct ndr_discrim {
        enum_t  value;
        NDR_PROC proc;
} NDR_DISCRIM;
#ifdef cplus
bool_t  ndr_union(NDR_STREAM *, enum_t *, caddr_t, NDR_DISCRIM *, NDR_PROC);
#endif
bool_t  ndr_union();

/*
 * ndr_reference is for recursively translating a structure that is
 * referenced by a pointer inside the structure that is currently being
 * translated.  2nd parameter references pointer to storage.
 *      If it is null, then routine malloc's necessary storage.
 * The 3rd parameter is the sizeof the referneced structure.
 * The last parameter is routine to unpack the referneced structure.
 */
#ifdef cplus
bool_t  ndr_reference(NDR_STREAM *, caddr_t *, u_int, NDR_PROC);
#endif
bool_t  ndr_reference();

/*
 * In-line routines for fast encode/decode of primitve data types.
 * Caveat emptor: these use single memory cycles to get the
 * data from the underlying buffer, and will fail to operate
```

```
 * properly if the data is not aligned.  The standard way to use these
 * is to say:
 *      if (ndr_inline(ndrs, count) == FALSE)
 *              return (FALSE);
 *      <<< macro calls >>>
 * where ''count'' is the number of bytes of data occupied
 * by the primitive data types.
 *
 * N.B. and frozen for all time: each data type here uses 4 bytes
 * of external representation.
 */
#ifndef nomacros
#define NDR_GET_LONG(ndrs)              (*((ndrs)->buf)++)
#define NDR_PUT_LONG(ndrs, v)           (*((ndrs)->buf)++ = (v))

#define NDR_GET_BOOL(ndrs)              ((bool_t)NDR_GET_LONG(ndrs))
#define NDR_GET_ENUM(ndrs, t)           ((t)NDR_GET_LONG(ndrs))
#define NDR_GET_U_LONG(ndrs)            ((u_long)NDR_GET_LONG(ndrs))
#define NDR_GET_SHORT(ndrs)             ((short)NDR_GET_LONG(ndrs))
#define NDR_GET_U_SHORT(ndrs)           ((u_short)NDR_GET_LONG(ndrs))

#define NDR_PUT_BOOL(ndrs, v)           NDR_PUT_LONG((ndrs), (long)(v))
#define NDR_PUT_ENUM(ndrs, v)           NDR_PUT_LONG((ndrs), (long)(v))
#define NDR_PUT_U_LONG(ndrs, v)         NDR_PUT_LONG((ndrs), (long)(v))
#define NDR_PUT_SHORT(ndrs, v)          NDR_PUT_LONG((ndrs), (long)(v))
#define NDR_PUT_U_SHORT(ndrs, v)        NDR_PUT_LONG((ndrs), (long)(v))
#endif
```

```
/*      %M%     %I%     %E%     */

/*
 * Generic NDR routines impelmentation.
 *
 * Copyright (C) 1984, Sun Microsystems, Inc.
 */

#include "types.h"              /* should be <sys/types.h> */
#include "heap.h"               /* should be ??? */
#include "ndr.h"                /* should be <rpc/ndr.h> */
/* constants specific to the ndr "protocol" */
#define NDR_FALSE       ((long) 0)
#define NDR_TRUE        ((long) 1)
#define BYTES_PER_NDR_UNIT      (4)
#define NULL    ((caddr_t) 0)

char ndr_zero[BYTES_PER_NDR_UNIT] = { 0, 0, 0, 0 };  /* for unit alignment */

#ifdef nomacros
bool_t
ndr_u_int(ndrs, up)
        NDR_STREAM *ndrs;
        u_int *up
{
        register bool_t (*proc)() = (sizeof(u_int) == sizeof(u_long)) ?
                ndr_u_long : ndr_u_short;

        return (proc(ndrs, up));
}

bool_t
ndr_int(ndrs, ip)
        NDR_STREAM *ndrs;
        int *ip;
{
        register bool_t (*proc)() = (sizeof(int) == sizeof(long)) ?
            ndr_long : ndr_short;

        return (proc(ndrs, ip));
}

bool_t
ndr_enum(ndrs, ep)
        NDR_STREAM *ndrs;
        enum_t *ep;
{

        return (ndr_int(ndrs, (int *) ep));
}

bool_t
ndr_u_long(ndrs, ulp)
        NDR_STREAM *ndrs;
        u_long *ulp;
{
```

```
        return (ndr_long(ndrs, (long *)ulp));
}
#endif

bool_t
ndr_null(ndrs, addr)
        NDR_STREAM *ndrs;
        caddr_t addr;
{
        return (TRUE);
}

bool_t
ndr_long(ndrs, lp)
        register NDR_STREAM *ndrs;
        long *lp;
{

        switch (ndrs->op) {

        case NDR_ENCODE:
                return ((*ndrs->ops->putlong)(ndrs, lp));

        case NDR_DECODE:
                return ((*ndrs->ops->getlong)(ndrs, lp));

        case NDR_FREE:
                return (TRUE);
        }
        return (FALSE);
}

bool_t
ndr_short(ndrs, sp)
        register NDR_STREAM *ndrs;
        short *sp;
{
        long l;

        switch (ndrs->op) {

        case NDR_ENCODE:
                l = (long) *sp;
                return ((*ndrs->ops->putlong)(ndrs, &l));

        case NDR_DECODE:
                if (!(*ndrs->ops->getlong)(ndrs, &l))
                        return (FALSE);
                *sp = (short) l;
                return (TRUE);

        case NDR_FREE:
                return (TRUE);
        }
        return (FALSE);
```

```
}

bool_t
ndr_u_short(ndrs, usp)
        register NDR_STREAM *ndrs;
        u_short *usp;
{
        u_long l;

        switch (ndrs->op) {

        case NDR_ENCODE:
                l = (u_long) *usp;
                return ((*ndrs->ops->putlong)(ndrs, &l));

        case NDR_DECODE:
                if (!(*ndrs->ops->getlong)(ndrs, &l)) return (FALSE);
                *usp = (u_short) l;
                return (TRUE);

        case NDR_FREE:
                return (TRUE);
        }
        return (FALSE);
}

bool_t
ndr_bool(ndrs, bp)
        register NDR_STREAM *ndrs;
        bool_t *bp;
{
        long lb;

        switch (ndrs->op) {

        case NDR_ENCODE:
                lb = *bp ? NDR_TRUE : NDR_FALSE;
                return ((*ndrs->ops->putlong)(ndrs, &lb));

        case NDR_DECODE:
                if (!(*ndrs->ops->getlong)(ndrs, &lb)) return (FALSE);
                *bp = (lb == NDR_FALSE) ? FALSE : TRUE;
                return (TRUE);

        case NDR_FREE:
                return (TRUE);
        }
        return (FALSE);
}

bool_t
ndr_opaque(ndrs, cp, cnt)
        register NDR_STREAM *ndrs;
        caddr_t cp;
        register u_int cnt;
{
```

```
        register u_int rndup = cnt % BYTES_PER_NDR_UNIT;
        static crud[BYTES_PER_NDR_UNIT];

        if (rndup > 0) rndup = BYTES_PER_NDR_UNIT - rndup;
        switch (ndrs->op) {

        case NDR_ENCODE:
                if (!(*ndrs->ops->putbytes)(ndrs, cp, cnt)) return (FALSE);
                return ((*ndrs->ops->putbytes)(ndrs, ndr_zero, rndup));

        case NDR_DECODE:
                if (!(*ndrs->ops->getbytes)(ndrs, cp, cnt)) return (FALSE);
                return ((*ndrs->ops->getbytes)(ndrs, crud, rndup));

        case NDR_FREE:
                return (TRUE);
        }
        return (FALSE);
}

bool_t
ndr_string(ndrs, cpp, maxsize)
        register NDR_STREAM *ndrs;
        char **cpp;
        u_int maxsize;
{
        register char *sp = *cpp;  /* sp is the actual string pointer */
        u_int size;

        /* first deal with the length since ndr strings are counted-strings */
        if ((ndrs->op) == NDR_ENCODE) size = strlen(sp);
        if (!ndr_u_int(ndrs, &size)) return (FALSE);
        if ((size > maxsize) && (ndrs->op != NDR_FREE)) return (FALSE);

        /* now deal with the actual bytes */
        switch (ndrs->op) {

        case NDR_DECODE:
                if (sp == NULL) *cpp = sp = makenode(ndrs->hp, size+1);
                sp[size] = 0;
                /* and then falls into ... */

        case NDR_ENCODE:
                return (ndr_opaque(ndrs, sp, size));

        case NDR_FREE:
                if (sp != NULL) {
                        freenode(ndrs->hp, sp);
                        *cpp = NULL;
                }
                return (TRUE);
        }
        return (FALSE);
}

bool_t
```

```
ndr_bytes(ndrs, cpp, sizep, maxsize)
        register NDR_STREAM *ndrs;
        char **cpp;
        register u_int *sizep;
        u_int maxsize;
{
        register char *sp = *cpp;  /* sp is the actual string pointer */
        register u_int c;

        /* first deal with the length since ndr strings are counted-strings */
        if (!ndr_u_int(ndrs, sizep)) return (FALSE);
        c = *sizep;
        if ((c > maxsize) && (ndrs->op != NDR_FREE)) return (FALSE);

        /* now deal with the actual bytes */
        switch (ndrs->op) {

        case NDR_DECODE:
                if (sp == NULL) {
                        *cpp = sp = makenode(ndrs->hp, c+1);
                        sp[c] = 0;
                }
                else if (c < maxsize) sp[c] = 0;
                /* and then falls into ... */

        case NDR_ENCODE:
                return (ndr_opaque(ndrs, sp, c));

        case NDR_FREE:
                if (sp != NULL) {
                        freenode(ndrs->hp, sp);
                        *cpp = NULL;
                }
                return (TRUE);
        }
        return (FALSE);
}

bool_t
ndr_array(ndrs, addrp, sizep, maxsize, elsize, elproc)
        register NDR_STREAM *ndrs;
        caddr_t *addrp;
        u_int *sizep;
        u_int maxsize;
        u_int elsize;
        NDR_PROC elproc;
{
        register u_int i;
        register caddr_t target = *addrp;
        register u_int c;  /* the actual element count */
        register bool_t stat = TRUE;
        register int nodesize;

        /* like strings, arrays are really counted arrays */
        if (!ndr_u_int(ndrs, sizep)) return (FALSE);
        c = *sizep;
```

```
        if ((c > maxsize) && (ndrs->op != NDR_FREE)) return (FALSE);

        /*
         * now if we are deserializing, we may need to allocate an array.
         * We also save time by checking for a null array if we are freeing.
         */
        if (target == NULL)
            switch (ndrs->op) {
                case NDR_DECODE:
                        if (c == 0) return (TRUE);
                        nodesize = c * elsize;
                        *addrp = target = makenode(ndrs->hp, nodesize);
                        bzero(target, nodesize);
                        break;

                case NDR_FREE:
                        return (TRUE);
            }

        /* now we recursively transverse each element of array */
        for (i = 0; (i < c) && stat; i++) {
                stat = (*elproc)(ndrs, target);
                target += elsize;
        }

        /* finally, it may need freeing ... */
        if (ndrs->op == NDR_FREE) {
                freenode(ndrs->hp, *addrp);
                *addrp = NULL;
        }
        return (stat);
}

/*
 * ndr_union is basicly a run time macro expander.  Note that
 * the direction (op) is not inspected here.  This means that the
 * ndr client programmer could have done all herself.
 */
bool_t
ndr_union(ndrs, dscmp, unp, choices, dfault)
        register NDR_STREAM *ndrs;
        enum_t *dscmp;
        caddr_t unp;
        NDR_DISCRIM *choices;
        NDR_PROC dfault;
{
        register enum_t dscm;

        /* we deal with the discriminator;  it's an enum ... */
        if (!ndr_enum(ndrs, dscmp)) return (FALSE);
        dscm = *dscmp;

        /* next try to match a discription routine with the discriminator */
        for(; choices->proc != NULL_NDR_PROC; choices++) {
                if (choices->value == dscm)
                    return ((*(choices->proc))(ndrs, unp));
```

```
        }

        /* no match - so maybe use the default */
        return ((dfault == NULL_NDR_PROC) ? FALSE : (*dfault)(ndrs, unp));
}

/*
 * ndr_reference is a pointer chaser.  It also allocates (frees)
 * storage if requested to do so.  We must be careful about
 * chasing NULL pointers when freeing...
 */
bool_t
ndr_reference(ndrs, pp, size, proc)
        register NDR_STREAM *ndrs;
        caddr_t *pp;
        u_int   size;
        NDR_PROC proc;
{
        register caddr_t loc = *pp;
        register bool_t stat;

        if (loc == NULL) switch (ndrs->op) {
                case NDR_FREE:
                        return (TRUE);

                case NDR_DECODE:
                        *pp = loc = makenode(ndrs->hp, size);
                        bzero(loc, size);
                        break;
        }

        stat = (*proc)(ndrs, loc);

        if (ndrs->op == NDR_FREE) {
                freenode(ndrs->hp, loc);
                *pp = NULL;
        }
        return (stat);
}
```

```
/*      %M%     %I%     %E%     */

/*
 * NDR implementation using stdio library.
 *
 * Copyright (C) 1984, Sun Microsystems, Inc.
 */

/*
 * If you wish to use NDR routines with stdio, use ndrstdio_create
 * to initialize a NDR_STREAM (which you provide).
 */
#ifdef cplus
void    ndrstdio_create(NDR_STREAM *, FILE *, op, hp);
#endif
void    ndrstdio_create();

extern  struct ndr_ops ndrstdio_ops;
```

```
/*      %M%     %I%     %E%     */

/*
 * NDR implementation on standard i/o file.
 *
 * Copyright (C) 1984, Sun Microsystems, Inc.
 */

#include "types.h"              /* should be <sys/types.h> */
#include <stdio.h>
#include "heap.h"               /* should be ??? */
#include "ndr.h"                /* should be <rpc/ndr.h> */
#include "ndr_stdio.h"          /* should be <rpc/ndr_mem.h> */

#if defined(mc68000) && defined(macros)
#define htonl(x)        x
#define ntohl(x)        x
#else
long htonl(), ntohl();
#endif

static  bool_t ndrstdio_getlong(), ndrstdio_putlong();
static  bool_t ndrstdio_getbytes(), ndrstdio_putbytes();
static  u_int ndrstdio_getposition();
static  bool_t ndrstdio_setposition();
static  bool_t ndrstdio_inline();
static  VOID ndrstdio_endinline();
static  VOID ndrstdio_destroy();
struct  ndr_ops ndrstdio_ops =
    { ndrstdio_getlong, ndrstdio_putlong, ndrstdio_getbytes, ndrstdio_putbytes,
      ndrstdio_getposition, ndrstdio_setposition,
      ndrstdio_inline, ndrstdio_endinline, ndrstdio_destroy };

void
ndrstdio_create(ndrs, file, op, hp)
        register NDR_STREAM *ndrs;
        FILE *file;
        NDR_OP op;
        HEAP *hp;
{

        ndrs->op = op;
        ndrs->hp = hp;
        ndrs->ops = &ndrstdio_ops;
        ndrs->private = (caddr_t)file;
        ndrs->handy = 0;
        ndrs->base = 0;
        ndrs->buf = 0;
}

static VOID
ndrstdio_destroy(ndrs)
        register NDR_STREAM *ndrs;
{
        fflush((FILE *)ndrs->private);
        /* xx should we close the file ?? */
```

```
};

static bool_t
ndrstdio_getlong(ndrs, lp)
        NDR_STREAM *ndrs;
        register long *lp;
{

        if (fread((caddr_t)lp, sizeof (long), 1, (FILE *)ndrs->private) != 1)
                return (FALSE);
#ifndef mc68000
        *lp = ntohl(*lp);
#endif
        return (TRUE);
}

static bool_t
ndrstdio_putlong(ndrs, lp)
        NDR_STREAM *ndrs;
        long *lp;
{

#ifndef mc68000
        long mycopy = htonl(*lp);
        lp = &mycopy;
#endif
        if (fwrite((caddr_t)lp, sizeof (long), 1, (FILE *)ndrs->private) != 1)
                return (FALSE);
        return (TRUE);
}

static bool_t
ndrstdio_getbytes(ndrs, addr, len)
        NDR_STREAM *ndrs;
        caddr_t addr;
        u_int len;
{

        if ((len != 0) && (fread(addr, len, 1, (FILE *)ndrs->private) != 1))
                return (FALSE);
        return (TRUE);
}

static bool_t
ndrstdio_putbytes(ndrs, addr, len)
        NDR_STREAM *ndrs;
        caddr_t addr;
        u_int len;
{

        if ((len != 0) && (fwrite(addr, len, 1, (FILE *)ndrs->private) != 1))
                return (FALSE);
        return (TRUE);
}

static u_int
```

```
ndrstdio_getposition(ndrs)
        NDR_STREAM *ndrs;
{

        return ((u_int) ftell((FILE *)ndrs->private));
}

static bool_t
ndrstdio_setposition(ndrs, pos)
        NDR_STREAM *ndrs;
        u_int pos;
{

        return ((fseek((FILE *)ndrs->private, (long)pos, 0) < 0) ?
            FALSE : TRUE);
}

static bool_t
ndrstdio_inline(ndrs, len)
        register NDR_STREAM *ndrs;
        u_int len;
{

#ifdef notdef
        /*
         * Must do some work to implement this: must insure
         * enough data in the underlying stdio buffer,
         * that the buffer is aligned so that we can indirect through a
         * long *, and stuff this pointer in ndrs->buf.  Doing
         * a fread or fwrite to a scratch buffer would defeat
         * most of the gains to be had here and require storage
         * management on this buffer, so we don't do this.
         */
#else
        return (FALSE);
#endif
}

static VOID
ndrstdio_endinline(ndrs)
        NDR_STREAM *ndrs;
{
};
```

```
/*      %M%     %I%     %E%     */

/*
 * NDR implementation using memory buffers.
 *
 * Copyright (C) 1984, Sun Microsystems, Inc.
 */

/*
 * If you have some data to be interpreted as network data representation
 * or to be converted to network data representation in a memory buffer,
 * then this is the package for you.
 *
 * The procedure ndrmem_create initializes a stream descriptor for a
 * memory buffer.
 */
#ifdef cplus
void    ndrmem_create(NDR_STREAM *, caddr_t, u_int, op, hp);
#endif

extern  struct ndr_ops ndrmem_ops;
```

```
/*      %M%     %I%     %E%     */

/*
 * NDR implementation on memory buffers.
 *
 * Copyright (C) 1984, Sun Microsystems, Inc.
 */

#include "types.h"              /* should be <sys/types.h> */
#include "heap.h"               /* should be ??? */
#include "ndr.h"                /* should be <rpc/ndr.h> */
#include "ndr_mem.h"            /* should be <rpc/ndr_mem.h> */

#if defined(mc68000) && defined(macros)
#define htonl(x)        x
#define ntohl(x)        x
#else
long htonl(), ntohl();
#endif

static  bool_t ndrmem_getlong(), ndrmem_putlong();
static  bool_t ndrmem_getbytes(), ndrmem_putbytes();
static  u_int ndrmem_getposition();
static  bool_t ndrmem_setposition();
static  bool_t ndrmem_inline();
static  VOID ndrmem_destroy();
static  VOID ndrmem_endinline();
struct  ndr_ops ndrmem_ops =
    { ndrmem_getlong, ndrmem_putlong, ndrmem_getbytes, ndrmem_putbytes,
      ndrmem_getposition, ndrmem_setposition,
      ndrmem_inline, ndrmem_endinline, ndrmem_destroy };

void
ndrmem_create(ndrs, addr, size, op, hp)
        register NDR_STREAM *ndrs;
        caddr_t addr;
        u_int size;
        NDR_OP op;
        HEAP *hp;
{

        ndrs->op = op;
        ndrs->hp = hp;
        ndrs->ops = &ndrmem_ops;
        ndrs->private = ndrs->base = addr;
        ndrs->handy = size;
        ndrs->buf = 0;
}

static VOID
ndrmem_destroy(ndrs)
        NDR_STREAM *ndrs;
{
};

static bool_t
```

```
ndrmem_getlong(ndrs, lp)
        register NDR_STREAM *ndrs;
        long *lp;
{

        if ((ndrs->handy -= sizeof (long)) < 0)
                return (FALSE);
        *lp = ntohl(*((long *)(ndrs->private)));
        ndrs->private += sizeof (long);
        return (TRUE);
}

static bool_t
ndrmem_putlong(ndrs, lp)
        register NDR_STREAM *ndrs;
        long *lp;
{
        register long *dest_lp = ((long *)(ndrs->private));

        if ((ndrs->handy -= sizeof (long)) < 0)
                return (FALSE);
        *dest_lp = htonl(*lp);
        ndrs->private += sizeof (long);
        return (TRUE);
}

static bool_t
ndrmem_getbytes(ndrs, addr, len)
        register NDR_STREAM *ndrs;
        caddr_t addr;
        u_int len;
{

        if ((ndrs->handy -= len) < 0)
                return (FALSE);
        bcopy(ndrs->private, addr, len);
        ndrs->private += len;
        return (TRUE);
}

static bool_t
ndrmem_putbytes(ndrs, addr, len)
        register NDR_STREAM *ndrs;
        caddr_t addr;
        u_int len;
{

        if ((ndrs->handy -= len) < 0)
                return (FALSE);
        bcopy(addr, ndrs->private, len);
        ndrs->private += len;
        return (TRUE);
}

static u_int
ndrmem_getposition(ndrs)
```

```
        register NDR_STREAM *ndrs;
{

        return ((u_int)ndrs->private - (u_int)ndrs->base);
}

static bool_t
ndrmem_setposition(ndrs, pos)
        register NDR_STREAM *ndrs;
        u_int pos;
{
        register caddr_t newaddr = ndrs->base + pos;
        register caddr_t lastaddr = ndrs->private + ndrs->handy;

        if ((long)newaddr > (long)lastaddr) return (FALSE);
        ndrs->private = newaddr;
        ndrs->handy = (int)lastaddr - (int)newaddr;
        return (TRUE);
}

static bool_t
ndrmem_inline(ndrs, len)
        register NDR_STREAM *ndrs;
        int len;
{

        if ((ndrs->handy -= len) < 0)
                return (FALSE);
        ndrs->buf = (long *)ndrs->private;
        ndrs->private += len;
        return (TRUE);
}

static VOID
ndrmem_endinline(ndrs)
        NDR_STREAM *ndrs;
{
}
```

```
/*      %M%     %I%     %E%     */

/*
 * Sun Authentication interface.
 * The data structures are completely opaque to the client.  The client
 * is required to pass a AUTH_CLIENT * to routines that create rpc
 * "sessions".
 *
 * Copyright (C) 1984, Sun Microsystems, Inc.
 */


/* why authentication may fail */

typedef enum auth_status {
/* failed at remote end because .... */
        AUTH_BADCRED=1,                 /* bogus credentials (seal broken) */
        AUTH_REJECTEDCRED=2,            /* client should begin new session */
        AUTH_BADVERF=3,                 /* bogus verifier (seal broken) */
        AUTH_REJECTEDVERF=4,            /* verifier expired or was replayed */
        AUTH_TOOWEAK=5,                 /* rejected due to security reasons */
/* may fail locally ... */
        AUTH_INVALIDRESP=6,             /* bogus response verifier */
        AUTH_FAILED=7                   /* some unknown reason */
} AUTH_STATUS;

union des_block {
        struct {
                u_long high;
                u_long low;
        } key;
        char c[8];
};

bool_t ndr_deskey();

#define MAX_AUTH_SIZE ((unsigned) 100)
#define MAX_AUTH_BYTES 400

struct opaque_auth {  /* opaque to the client */
        enum_t flavor;
        caddr_t base;
        u_int length;  /* not to exceed MAX_AUTH_SIZE */
};

extern struct opaque_auth null_auth;
bool_t ndr_opaque_auth();

typedef struct auth_client {
        struct opaque_auth credential;
        struct opaque_auth verfier;
        union des_block conversation_key;
        struct auth_ops {
                VOID (*get_next_verf)();
                bool_t (*validate_verf)();
                VOID    (*destroy)();           /* destroy this structure */
```

```
        } *ops;
} AUTH_CLIENT;

#ifndef nomacros
#define get_next_verfier(auth)                                  \
        ((*(auth).ops->get_next_verf)(auth))
#define validate_verfier(auth, verf)                            \
        ((*(auth).ops->validate_verf)(auth, verf))
#define destroy_auth(auth)                                      \
        ((*(auth).ops->destroy)(auth))
#endif
```

```
/*      %M%     %I%     %E%     */

/*
 * Creates an authentication client that passes "null"
 * credentials and verifiers to remote systems.
 *
 * Copyright (C) 1984, Sun Microsystems, Inc.
 */

#ifdef cplus
void    authnone_create(AUTH_CLIENT *);
#endif

void    authnone_create();
```

```
/*      %M%     %I%     %E%     */

/*
 * Creates an authentication client that passes "null"
 * credentials and verifiers to remote systems.
 *
 * Copyright (C) 1984, Sun Microsystems, Inc.
 */

#include "types.h"
#include "auth.h"
#include "auth_none.h"

static VOID none_next_verf(), none_destroy();
static bool_t none_validate_verf();
static struct auth_ops ops =
        { none_next_verf, none_validate_verf, none_destroy};

static VOID
none_next_verf()
{
}

static bool_t
none_validate_verf()
{

        return (TRUE);
}

static VOID
none_destroy()
{
}

void
authnone_create(c)
        register AUTH_CLIENT *c;
{

        c->credential = c->verfier = null_auth;
        c->ops = &ops;
}
```

```
/*      %M%     %I%     %E%     */

/*
 * Remote procedure call interface.
 *
 * Copyright (C) 1984, Sun Microsystems, Inc.
 */

/*
 * Rpc call returns a RPC_STATUS.  This should be looked at more,
 * since each implementation is required to live with this (implementation
 * independent) list of errors.
 */
typedef enum rpc_status {
        RPC_SUCCESS=0,                  /* call succeeded */
/* errors locally */
        RPC_CANTENCODEARGS=1,           /* can't encode arguments */
        RPC_CANTDECODERES=2,            /* can't decode results */
        RPC_CANTSEND=3,                 /* failure in sending call */
        RPC_CANTRECV=4,                 /* failure in receiving result */
        RPC_TIMEDOUT=5,                 /* call timed out */
/* errors from remote end */
        RPC_VERSMISMATCH=6,             /* rpc versions not compatible */
        RPC_AUTHERROR=7,                /* authentication error */
        RPC_PROGUNAVAIL=8,              /* program not available */
        RPC_PROGVERSMISMATCH=9,         /* program version mismatched */
        RPC_PROCUNAVAIL=10,             /* procedure unavailable */
        RPC_CANTDECODEARGS=11,          /* decode arguments error */
/* unspecified error */
        RPC_FAILED=12
} RPC_STATUS;

typedef struct rpc_error {
        RPC_STATUS status;
        union {
                AUTH_STATUS why;        /* why the auth error occurred */
                struct {
                        u_long low;     /* lowest verion supported */
                        u_long high;    /* highest verion supported */
                } versions;
                struct {                /* maybe meaningful if RPC_FAILED */
                        long s1;
                        long s2;
                } lb;                   /* life boot & debugging only */
        } u;
} RPC_ERROR;

/*
 * Rpc handle.  Created by individual implementations, see e.g. rpc_udp.h.
 * Client is responsible for initializing auth, see e.g. auth_none.h.
 */
typedef struct rpc_handle {
        AUTH_CLIENT auth;
        struct rpc_ops {
                RPC_STATUS (*call)();   /* call remote procedure */
                VOID    (*abort)();     /* abort a call */
```

```
                RPC_ERROR (*geterr)();  /* get more specific error code */
                VOID    (*destroy)();   /* destroy this structure */
        } *ops;
        caddr_t private;
} RPC_HANDLE;

#if defined(cplus) && defined(nomacros)
RPC_STATUS rpc(RPC_HANDLE *, u_long, u_long *, NDR_PROC, caddr_t, NDR_PROC, caddr_t);
void    rpc_abort(RPC_HANDLE *);
RPC_ERROR rpc_geterr(RPC_HANDLE *);
void    rpc_destroy(RPC_HANDLE *);
#endif

#ifndef nomacros
#define rpc(rh, proc, tid, argf, argp, resultf, resultp)                \
    ((*(rh)->ops->call)((rh), (proc), (tid), (argf), (argp), (resultf), (resultp)))
#define rpc_abort(rh)                                                   \
    ((*(rh)->ops->abort)(rh))
#define rpc_geterr(rh)                                                  \
    ((*(rh)->ops->geterr)(rh))
#define rpc_destroy(rh)                                                 \
    ((*(rh)->ops->destroy)(rh))
#else
RPC_STATUS rpc();
void    rpc_abort();
RPC_ERROR  rpc_geterr();
void    rpc_destroy();
#endif
```

```
/*      %M%     %I%     %E%     */

/*
 * This header handles the rpc message definition, its serializer and
 * some common rpc utility routines.
 * Types and functions in here are for various implementations of rpc -
 * they are NOT for the rpc client or rpc service implementations!
 *
 * Copyright (C) 1984, Sun Microsystems, Inc.
 */

#define RPC_SERVICE_PORT ((u_short) 2048)

/*
 * now for a bottom up definition of an rpc message.  The declaration were
 * lifted directly from the (proposed) design spec.
 */

typedef enum rpc_direction { call=0, reply=1 } RPC_DIRECTION;

typedef struct rpc_call_msg_body {
        u_long rpc_version;  /* must be equal to one */
        struct opaque_auth cred;
        struct opaque_auth verf;
        u_long program;
        u_long version;
        u_long procedure;
        /* protocol specific - provided by client */
        caddr_t args;
        NDR_PROC ndr_args;
} RPC_CALL_MSG_BODY;

typedef enum rpc_reply_type { msg_accepted=0, msg_denied=1 } RPC_REPLY_TYPE;

typedef enum rpc_accept_type { success=0, prog_unavail=1, prog_mismatch=2,
        proc_unavail=3, garbage_args=4 } RPC_ACCEPT_TYPE;

typedef enum rpc_reject_type { rpc_mismatch=0, auth_error=1 } RPC_REJECT_TYPE;

typedef struct rpc_accepted_reply {
        struct opaque_auth verf;
        RPC_ACCEPT_TYPE acpt_stat;
        union {
                struct {
                        u_long low;
                        u_long high;
                } versions;
                struct {
                        caddr_t where;
                        NDR_PROC proc;
                } results;
                /* and many other null cases */
        } u;
} RPC_ACCEPTED_REPLY;

typedef struct rpc_rejected_reply {
```

```
        RPC_REJECT_TYPE rjct_stat;
        union {
                struct {
                        u_long low;
                        u_long high;
                } versions;
                AUTH_STATUS why;  /* why authentication did not work */
        } u;
} RPC_REJECTED_REPLY;

typedef struct rpc_reply_msg_body {
        RPC_REPLY_TYPE rt;
        union {
                RPC_ACCEPTED_REPLY ar;
                RPC_REJECTED_REPLY dr;
        } u;
} RPC_REPLY_MSG_BODY;

/* "the" rpc message ... */
typedef struct rpc_message {
        long x_id;
        RPC_DIRECTION direction;
        union {
                RPC_CALL_MSG_BODY cmb;
                RPC_REPLY_MSG_BODY rmb;
        } u;
} RPC_MESSAGE;

#ifdef cplus
bool_t ndr_rpc_call_message(NDR_STREAM *, RPC_MESSAGE *);
bool_t ndr_rpc_reply_message(NDR_STREAM *, RPC_MESSAGE *);
void rpc_reply_to_rpc_error(RPC_MESSAGE *, RPC_ERROR *);
#endif

bool_t  ndr_rpc_call_message(), ndr_rpc_reply_message();
void    rpc_reply_to_rpc_error();  /* given a reply message, fills in the error */
```

```
/*      %M%     %I%     %E%     */

/*
 * This program implements the rpc message definition, its serializer and
 * some common rpc utility routines.
 * The routines are meant for various implementations of rpc -
 *  they are NOT for the rpc client or rpc service implementations!
 * Because authentication stuff is easy and is part of rpc, the opaque
 * routines are also in this program.
 *
 * Copyright (C) 1984, Sun Microsystems, Inc.
 */

#include "types.h"              /* should be <sys/types.h> */
#include "heap.h"
#include "ndr.h"
#include "auth.h"
#include "rpc.h"
#include "rpc_msg.h"

/* * * * * * * * * * * * * * * * Basic Authentication * * * * * * * * * * * */

struct opaque_auth null_auth;

union opaque_word {   /* size of this must be 32 bits - the basic ndr size */
        u_long l_val;
        char c_val[4];
};

static bool_t
ndr_opaque_word(ndrs, op)
        NDR_STREAM *ndrs;
        union opaque_word *op;
{

        return (ndr_opaque(ndrs, op, sizeof (union opaque_word)));
}

bool_t
ndr_opaque_auth(ndrs, ap)
        register NDR_STREAM *ndrs;
        register struct opaque_auth *ap;
{

        if (!ndr_enum(ndrs, &(ap->flavor))) return (FALSE);
        return (ndr_array(ndrs, &(ap->base), &(ap->length),
            MAX_AUTH_SIZE, sizeof (union opaque_word), ndr_opaque_word));
}

bool_t
ndr_deskey(ndrs, blkp)
        register NDR_STREAM *ndrs;
        register union des_block *blkp;
{

        if (!ndr_u_long(ndrs, &(blkp->key.high))) return (FALSE);
```

```
        return (ndr_u_long(ndrs, &(blkp->key.low)));
}

/* * * * * * * * * * * * * * * * Common RPC * * * * * * * * * * * * * * * * * * * */

static bool_t
ndr_rpc_accepted_reply(ndrs, ar)
        register NDR_STREAM *ndrs;
        register RPC_ACCEPTED_REPLY *ar;
{

        /* personalized union, rather than calling ndr_union */
        if (!ndr_opaque_auth(ndrs, &(ar->verf))) return (FALSE);
        if (!ndr_enum(ndrs, &(ar->acpt_stat))) return (FALSE);
        switch (ar->acpt_stat) {

        case success:
                return ((*(ar->u.results.proc))(ndrs, ar->u.results.where));

        case prog_mismatch:
                if (!ndr_u_long(ndrs, &(ar->u.versions.low))) return (FALSE);
                return (ndr_u_long(ndrs, &(ar->u.versions.high)));
        }
        return (TRUE);  /* TRUE => open ended set of problems */
}

static bool_t
ndr_rpc_rejected_reply(ndrs, rr)
        register NDR_STREAM *ndrs;
        register RPC_REJECTED_REPLY *rr;
{

        /* personalized union, rather than calling ndr_union */
        if (!ndr_enum(ndrs, &(rr->rjct_stat))) return (FALSE);
        switch (rr->rjct_stat) {

        case rpc_mismatch:
                if (!ndr_u_long(ndrs, &(rr->u.versions.low))) return (FALSE);
                return (ndr_u_long(ndrs, &(rr->u.versions.high)));

        case auth_error:
                return (ndr_enum(ndrs, &(rr->u.why)));
        }
        return (FALSE);
}

NDR_DISCRIM reply_dscrm[3] = {
        { (enum_t)msg_accepted, ndr_rpc_accepted_reply },
        { (enum_t)msg_denied, ndr_rpc_rejected_reply },
        { __dontcare__, NULL_NDR_PROC } };
bool_t
ndr_rpc_reply_message(ndrs, rmsg)
        register NDR_STREAM *ndrs;
        register RPC_MESSAGE *rmsg;
{
```

```
        if (
            ndr_u_long(ndrs, &(rmsg->x_id)) &&
            ndr_enum(ndrs, &(rmsg->direction)) &&
            (rmsg->direction == reply) )
            return (ndr_union(ndrs, &(rmsg->u.rmb.rt), &(rmsg->u.rmb.u),
                reply_dscrm, NULL_NDR_PROC));
        return (FALSE);
}

bool_t
ndr_rpc_call_message(ndrs, cmsg)
        register NDR_STREAM *ndrs;
        register RPC_MESSAGE *cmsg;
{

        if (
            ndr_u_long(ndrs, &(cmsg->x_id)) &&
            ndr_enum(ndrs, &(cmsg->direction)) &&
            (cmsg->direction == call) &&
            ndr_u_long(ndrs, &(cmsg->u.cmb.rpc_version)) &&
            (cmsg->u.cmb.rpc_version == 1) &&
            ndr_opaque_auth(ndrs, &(cmsg->u.cmb.cred)) &&
            ndr_opaque_auth(ndrs, &(cmsg->u.cmb.verf)) &&
            ndr_u_long(ndrs, &(cmsg->u.cmb.program)) &&
            ndr_u_long(ndrs, &(cmsg->u.cmb.version)) &&
            ndr_u_long(ndrs, &(cmsg->u.cmb.procedure)) )
            return ((*(cmsg->u.cmb.ndr_args))(ndrs, cmsg->u.cmb.args));
        return (FALSE);
}

static void
accepted(acpt_stat, error)
        register RPC_ACCEPT_TYPE acpt_stat;
        register RPC_ERROR *error;
{

        switch (acpt_stat) {

        case prog_unavail:
                error->status = RPC_PROGUNAVAIL;
                return;

        case prog_mismatch:
                error->status = RPC_PROGVERSMISMATCH;
                return;

        case proc_unavail:
                error->status = RPC_PROCUNAVAIL;
                return;

        case garbage_args:
                error->status = RPC_CANTDECODEARGS;
                return;

        case success:
                error->status = RPC_SUCCESS;
```

```
                return;
        }
        /* something's wrong, but we don't know what ... */
        error->status = RPC_FAILED;
        error->u.lb.s1 = (long)msg_accepted;
        error->u.lb.s2 = (long)acpt_stat;
}

static void
rejected(rjct_stat, error)
        register RPC_REJECT_TYPE rjct_stat;
        register RPC_ERROR *error;
{

        switch (rjct_stat) {

        case rpc_mismatch:
                error->status = RPC_VERSMISMATCH;
                return;

        case auth_error:
                error->status = RPC_AUTHERROR;
                return;
        }
        /* something's wrong, but we don't know what ... */
        error->status = RPC_FAILED;
        error->u.lb.s1 = (long)msg_denied;
        error->u.lb.s2 = (long)rjct_stat;
}

void
rpc_reply_to_rpc_error(msg, error)
        /* given a reply message, fills in the error */
        register RPC_MESSAGE *msg;
        register RPC_ERROR *error;
{

        /* optimized for normal, successful case */
        switch (msg->u.rmb.rt) {

        case msg_accepted:
                if (msg->u.rmb.u.ar.acpt_stat == success) {
                        error->status = RPC_SUCCESS;
                        return;
                };
                accepted(msg->u.rmb.u.ar.acpt_stat, error);
                break;

        case msg_denied:
                rejected(msg->u.rmb.u.dr.rjct_stat, error);
                break;

        default:
                error->status = RPC_FAILED;
                error->u.lb.s1 = (long)(msg->u.rmb.rt);
                break;
```

```
        }
        switch (error->status) {

        case RPC_VERSMISMATCH:
                error->u.versions.low = msg->u.rmb.u.dr.u.versions.low;
                error->u.versions.high = msg->u.rmb.u.dr.u.versions.high;
                break;

        case RPC_AUTHERROR:
                error->u.why = msg->u.rmb.u.dr.u.why;
                break;

        case RPC_PROGVERSMISMATCH:
                error->u.versions.low = msg->u.rmb.u.ar.u.versions.low;
                error->u.versions.high = msg->u.rmb.u.ar.u.versions.high;
                break;
        }
}
```

```
/*      %M%     %I%     %E%     */

/*
 * UDP based RPC implementation
 *
 * Copyright (C) 1984, Sun Microsystems, Inc.
 *
 * Client passes RPC_HANDLE * and rpcudp_create fills it in.
 * Usage: rpcudp_create(rpch, remoteaddr, remotelen, program, version,
        tries, wait_per_try, sock_ptr);
 * If *sock_ptr<0, *sock_ptr is set to a newly created UPD socket.
 * NB: *sock_ptr is copied into a private area.
 * NB: It is the clients responsibility to close *sock_ptr.
 */
#if defined(cplus)
bool_t rpcudp_create(RPC_HANDLE *, struct sockaddr_in *,
        int, u_long, u_long, u_int, struct time_val, int *);
#endif
bool_t rpcudp_create();

/* ... */
```

```
/*      %M%     %I%     %E%     */

/*
 * Implements a UPD/IP based, client side RPC.
 *
 * Copyright (C) 1984, Sun Microsystems, Inc.
 */

#include "types.h"              /* should be <sys/types.h> */
#include "heap.h"
#include "mallocheap.h"
#include "ndr.h"
#include "ndr_mem.h"
#include "auth.h"
#include "rpc.h"
#include "rpc_msg.h"
#include "rpc_udp.h"
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <netdb.h>
#define UDPMSGSIZE 1400
#define NULL ((caddr_t)0)

static RPC_STATUS upd_callit();
static VOID abort();
static RPC_ERROR get_error();
static VOID destroy();
static struct rpc_ops upd_ops = { upd_callit, abort, get_error, destroy };

struct private_data {
        int sock;
        struct sockaddr_in remoteaddr;
        int remotelen;
        u_int tries;
        struct timeval wait_per_try;
        RPC_ERROR error;
        RPC_MESSAGE call_msg;
        NDR_STREAM outndrs;
        NDR_STREAM inndrs;
        HEAP *hp;
        char outbuf[UDPMSGSIZE];
        char inbuf[UDPMSGSIZE];
};

static RPC_STATUS
upd_callit(h, proc, x_id, ndr_args, args_ptr, ndr_results, results_ptr)
        register RPC_HANDLE *h;
        u_long proc;
        register u_long *x_id;
        NDR_PROC ndr_args;
        caddr_t args_ptr;
        NDR_PROC ndr_results;
        caddr_t results_ptr;
{
        register struct private_data *p =
```

```
        (struct private_data *) h->private;
    register NDR_STREAM *ndrs = &(p->outndrs);
    register u_int outlen, inlen;
    int readfds, fromlen;
    register int mask;
    register int i = 0;
    struct sockaddr_in from;
    RPC_MESSAGE reply_msg;

    if (*x_id == 0) *x_id = (u_long)random();
    p->call_msg.x_id = *x_id;
    get_next_verfier(h->auth);
    p->call_msg.u.cmb.cred = h->auth.credential;
    p->call_msg.u.cmb.verf = h->auth.verfier;
    p->call_msg.u.cmb.procedure = proc;
    p->call_msg.u.cmb.args = args_ptr;
    p->call_msg.u.cmb.ndr_args = ndr_args;
    if (ndr_rpc_call_message(ndrs, &(p->call_msg)) == FALSE)
        return (p->error.status = RPC_CANTENCODEARGS);
    outlen = ndr_getposition(ndrs);
    while (TRUE) {

        if (sendto(p->sock, p->outbuf, outlen, 0,
            &(p->remoteaddr), p->remotelen) != outlen)
            return (p->error.status = RPC_CANTSEND);
        /*
         * sub-optimal code appears inside the loop because we have
         * some clock time to spare while the packets are in flight.
         * (We assume that this is actually only executed once.)
         */
        ndr_setposition(ndrs, 0);
        reply_msg.u.rmb.u.ar.verf = null_auth;
        reply_msg.u.rmb.u.ar.u.results.where = results_ptr;
        reply_msg.u.rmb.u.ar.u.results.proc = ndr_results;
        ndrs = &(p->inndrs);
        ndr_setposition(ndrs, 0);
        ndrs->op = NDR_DECODE;
        mask = 1 << p->sock;
rcv_again:
        fromlen = sizeof(struct sockaddr);
        readfds = mask;
        switch (select(32, &readfds, NULL, NULL, &(p->wait_per_try))) {

        case 0:
                if (++i < p->tries) continue;
                return (p->error.status = RPC_TIMEDOUT);

        case -1:
                return (p->error.status = RPC_CANTRECV);
        }
        if ((readfds & mask) == 0) goto rcv_again;
        inlen = recvfrom(p->sock, p->inbuf, UDPMSGSIZE, 0,
            &from, &fromlen);
        if (inlen < sizeof (u_long)) goto rcv_again;
        /* see if reply transaction id matches sent id */
        if (*((u_long *)(p->inbuf)) != *((u_long *)(p->outbuf)))
```

```
                    goto rcv_again;
                /* we now assume we have the proper reply */
                break;
        }

        /* now decode and validate the response */
        if (ndr_rpc_reply_message(ndrs, &reply_msg)) {
                rpc_reply_to_rpc_error(&reply_msg, &(p->error));
                if (p->error.status == RPC_SUCCESS  &&
                    validate_verfier(h->auth, reply_msg.u.rmb.u.ar.verf)
                    == FALSE) {
                        p->error.status = RPC_AUTHERROR;
                        p->error.u.why = AUTH_INVALIDRESP;
                }
        }
        else
           p->error.status = RPC_CANTDECODERES;
        ndrs->op = NDR_FREE;
        reply_msg.u.rmb.u.ar.u.results.proc = ndr_null;
        ndr_rpc_reply_message(ndrs, &reply_msg);
        return (p->error.status);
}

static RPC_ERROR
get_error(h)
        RPC_HANDLE *h;
{
        register struct private_data *p =
            (struct private_data *) h->private;

        return (p->error);
}

static VOID
abort(h)
        RPC_HANDLE *h;
{
}

static VOID
destroy(h)
        RPC_HANDLE *h;
{
        register struct private_data *p =
            (struct private_data *) h->private;

        close(p->sock);
        (*(p->inndrs.ops->destroy))(&(p->inndrs));
        (*(p->outndrs.ops->destroy))(&(p->outndrs));
        (*(p->hp->ops._destroy))(p->hp);
}

bool_t
rpcudp_create(h, remoteaddr, remotelen, program, version, tries, wait_per_try, sock_ptr)
        RPC_HANDLE *h;
        struct sockaddr_in *remoteaddr;
```

```
        int remotelen;
        u_long program;
        u_long version;
        u_int tries;
        struct timeval wait_per_try;
        register int *sock_ptr;
{
        HEAP *hp = make_unix_heap();
        register struct private_data *p =
            (struct private_data *) makenode(hp, sizeof (struct private_data));
        struct timeval now;
        struct timezone dummy;

        h->ops = &upd_ops;
        h->private = (caddr_t) p;
        p->hp = hp;
        p->remoteaddr = *remoteaddr;
        p->remotelen = remotelen;
        p->tries = tries;
        p->wait_per_try = wait_per_try;
        p->call_msg.direction = call;
        p->call_msg.u.cmb.rpc_version = 1;
        p->call_msg.u.cmb.program = program;
        p->call_msg.u.cmb.version = version;
        ndrmem_create(&(p->outndrs), p->outbuf, UDPMSGSIZE, NDR_ENCODE, hp);
        ndrmem_create(&(p->inndrs), p->inbuf, UDPMSGSIZE, NDR_DECODE, hp);
        p->sock = (*sock_ptr < 0) ?
            (*sock_ptr = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) :
            *sock_ptr;
        if (p->sock < 0) return (FALSE);
        gettimeofday(&now, &dummy);
        srandom((int)(getpid() ^ random() ^ now.tv_sec ^ now.tv_usec));
        return (TRUE);
}
```

```
/*      %M%     %I%     %E%     */

/*
 * TESTS Network Data Representation Serialization Routines.
 * by serializing argc, argv.
 *
 * Copyright (C) 1984, Sun Microsystems, Inc.
 */

#include "types.h"
#include "heap.h"
#include "mallocheap.h"
#include "ndr.h"
#include "ndr_stdio.h"
#include <stdio.h>

#define MAXSTRINGSIZE ((u_int) 24)
#define MAXSTRINGS ((u_int) 128)

bool_t
note_string(ndrs, sp)
        NDR_STREAM *ndrs;
        char **sp;
{
        return (ndr_string(ndrs, sp, MAXSTRINGSIZE));
}

main (argc, argv)
        int argc;
        char **argv;
{
        NDR_STREAM ndrsobject;
        register NDR_STREAM *ndrs = &ndrsobject;
        HEAP *stdheap = make_unix_heap();
        register int i;

        /* set up ... */
        ndrsobject.hp = stdheap;
        ndrstdio_create(ndrs, stdout, NDR_ENCODE);

        /* do it */
        if (!ndr_array(ndrs, &argv, &argc, MAXSTRINGS,
            sizeof(char *), note_string)) {
                fflush(stdout);
                abort();
        }

        /* test degenerate strings */
        for (i=0; i < argc; i += 2) {
                argv[i][0] = 0;
        };
        if (!ndr_array(ndrs, &argv, &argc, MAXSTRINGS,
            sizeof(char *), note_string)) {
                fflush(stdout);
                abort();
        }
```

```
        /* test degenerate array */
        argc = 0;
        argv = NULL;
        if (!ndr_array(ndrs, &argv, &argc, MAXSTRINGS,
            sizeof(char *), note_string)) {
                fflush(stdout);
                abort();
        }
        fflush(stdout);
        exit(0);
}
```

```
/*      %M%     %I%     %E%     */

/*
 * TIMES Network Data Representation Serialization Routines.
 * by (de)serializing argc, argv.
 *
 * Copyright (C) 1984, Sun Microsystems, Inc.
 */

#include "types.h"
#include "heap.h"
#include "mallocheap.h"
#include "ndr.h"
#include "ndr_mem.h"
#include <sys/time.h>
#include <stdio.h>

#define MAXSTRINGSIZE ((u_int) 24)
#define MAXSTRINGS ((u_int) 128)

char buff[4000];
int sargc;
char **sargv;
NDR_STREAM *gndrs;

bool_t
note_string(ndrs, sp)
        NDR_STREAM *ndrs;
        char **sp;
{
        return (ndr_string(ndrs, sp, MAXSTRINGSIZE));
}

doit(dir)
        NDR_OP dir;
{

        gndrs->op = dir;
        if (!ndr_setposition(gndrs, 0)) abort();
        if (!ndr_array(gndrs, &sargv, &sargc, MAXSTRINGS,
            sizeof(char *), note_string)) abort();
}

main (argc, argv)
        int argc;
        char **argv;
{
        NDR_STREAM gndrs_object;
        register NDR_OP dir = NDR_ENCODE;
        struct timeval begin, end;
        struct timezone dummy;
        register int i;

        sargc = argc;
        sargv = argv;
        gndrs = &gndrs_object;
```

```
        ndrmem_create(gndrs, buff, 4000, dir, NULL);

        doit(dir);
        gettimeofday(&begin, &dummy);
        for (i = 0; i < 1000; i++) doit(dir);
        gettimeofday(&end, &dummy);
        end.tv_sec -= begin.tv_sec;
        fprintf(stderr, "%d calls takes %ld seconds.\n", i,  end.tv_sec);
        fflush(stderr);

        dir = NDR_DECODE;
        doit(dir);
        gettimeofday(&begin, &dummy);
        for (i = 0; i < 1000; i++) doit(dir);
        gettimeofday(&end, &dummy);
        end.tv_sec -= begin.tv_sec;
        fprintf(stderr, "%d calls takes %ld seconds.\n", i,  end.tv_sec);
        fflush(stderr);

        exit(0);
}
```

```
/*      %M%     %I%     %E%     */

/*
 * TIMES rpc message serialization
 * by serializing / deserializing / and freeing null rpc call & reply
 * messages.
 *
 * Copyright (C) 1984, Sun Microsystems, Inc.
 */

#include <stdio.h>
#include <sys/time.h>
#include "types.h"
#include "heap.h"
#include "ndr.h"
#include "ndr_mem.h"
#include "auth.h"
#include "rpc.h"
#include "rpc_msg.h"

#define BUFSIZE 400

main()
{
        RPC_MESSAGE call_msg;
        register RPC_MESSAGE *cm = &call_msg;
        RPC_MESSAGE reply_msg;
        register RPC_MESSAGE *rm = &reply_msg;
        char buff[BUFSIZE];
        NDR_STREAM ndrsobject;
        register NDR_STREAM *ndrs = &ndrsobject;
        register int i;
        struct timeval begin, end;
        struct timezone dummy;

        call_msg.direction = call;
        call_msg.u.cmb.rpc_version = 1;
        call_msg.u.cmb.args = NULL;
        call_msg.u.cmb.ndr_args = ndr_null;
        call_msg.u.cmb.cred = call_msg.u.cmb.verf = null_auth;
        reply_msg.direction = reply;
        reply_msg.u.rmb.rt = msg_accepted;
        reply_msg.u.rmb.u.ar.verf = null_auth;
        reply_msg.u.rmb.u.ar.acpt_stat = success;
        reply_msg.u.rmb.u.ar.u.results.where = NULL;
        reply_msg.u.rmb.u.ar.u.results.proc = ndr_null;
        gettimeofday(&begin, &dummy);
        for (i=0; i<10000; i++) {
                ndrmem_create(ndrs, buff, BUFSIZE, NDR_ENCODE, NULL);
                if (!ndr_rpc_call_message(ndrs, cm)) abort();
                (*(ndrs->ops->destroy))(ndrs);
                ndrmem_create(ndrs, buff, BUFSIZE, NDR_DECODE, NULL);
                if (!ndr_rpc_call_message(ndrs, cm)) abort();
                ndrs->op = NDR_FREE;
                if (!ndr_rpc_call_message(ndrs, cm)) abort();
                (*(ndrs->ops->destroy))(ndrs);
```

```
                ndrmem_create(ndrs, buff, BUFSIZE, NDR_ENCODE, NULL);
                if (!ndr_rpc_reply_message(ndrs, rm)) abort();
                (*(ndrs->ops->destroy))(ndrs);
                ndrmem_create(ndrs, buff, BUFSIZE, NDR_DECODE, NULL);
                if (!ndr_rpc_reply_message(ndrs, rm)) abort();
                ndrs->op = NDR_FREE;
                if (!ndr_rpc_reply_message(ndrs, rm)) abort();
                (*(ndrs->ops->destroy))(ndrs);
        }
        gettimeofday(&end, &dummy);
        end.tv_sec -= begin.tv_sec;
        fprintf(stderr, "%d round trips take %ld seconds.\n", i,  end.tv_sec);
        fflush(stderr);
        exit(0);
}
```

```
/*      %M%     %I%     %E%     */

/*
 * rpc_client.c, last modified by blyon, date: Jan 12, 1984.
 *
 * oroginal author: blyon, date Dec 29, 1983.
 *
 * tests and times client side of rpc.
 */

#include "types.h"                      /* should be <sys/types.h> */
#include "heap.h"
#include "mallocheap.h"
#include "ndr.h"
#include "ndr_mem.h"
#include "auth.h"
#include "auth_none.h"
#include "rpc.h"
#include "rpc_udp.h"
#include <stdio.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <netdb.h>
#define SERVICE_PORT ((u_short) 2048)
#define PROG ((long unsigned) 1)
#define VERS ((long unsigned) 1)
#define PROC ((long unsigned) 2)

main (argc, argv)
        int argc;
        char **argv;
{
        struct sockaddr_in server_addr;
        register int i, addrlen;
        register RPC_STATUS rpc_stat;
        struct hostent *he;
        u_long x_id;
        struct timeval begin, end, timeout;
        struct timezone dummy;
        RPC_HANDLE client_handle;
        register RPC_HANDLE *client = &client_handle;
        int socket = -1;

        if (argc != 2) {
                fprintf(stderr, "usage: %s [-s | server_name]\n", argv[0]);
                exit(0);
        };
        timeout.tv_sec = 2;
        if ((he = gethostbyname(argv[1])) == NULL) {
                fprintf(stderr, "cannot get addr for '%s'\n", argv[1]);
                exit(0);
        };
        addrlen = sizeof(struct sockaddr_in);
        bcopy(he->h_addr, &server_addr.sin_addr, he->h_length);
        server_addr.sin_family = AF_INET;
```

```
        server_addr.sin_port = htons(SERVICE_PORT);
        authnone_create(&client_handle.auth);
        if (rpcudp_create(client, &server_addr, addrlen, PROG, VERS,
            3, timeout, &socket) == FALSE) abort();
        for (i=0; i<100; i++) {
                x_id = 0;
                rpc_stat =
                   rpc(client, PROC, &x_id, ndr_null, NULL, ndr_null, NULL);
                fprintf(stderr, " %d", rpc_stat); fflush(stderr);
        };
        gettimeofday(&begin, &dummy);
        for (i=0; i<1000; i++) {
                x_id++;
                rpc_stat =
                    rpc(client, PROC, &x_id, ndr_null, NULL, ndr_null, NULL);
        };
        gettimeofday(&end,  &dummy);
        end.tv_sec -= begin.tv_sec;
        fprintf(stderr, "%d calls takes %ld seconds.\n", i,  end.tv_sec);
        fflush(stderr);
        exit(0);
};  /* main */
```