```
**********************************************************
   33959 Fri Apr 25 08:35:53 2014
new/usr/src/lib/libnsl/rpc/rpcb_clnt.c
4729 __rpcb_findaddr_timed should try rpcbind protocol 4 first
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License, Version 1.0 only
   6  * (the "License").  You may not use this file except in compliance
   7  * with the License.
   8  *
   9  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
  10  * or http://www.opensolaris.org/os/licensing.
  11  * See the License for the specific language governing permissions
  12  * and limitations under the License.
  13  *
  14  * When distributing Covered Code, include this CDDL HEADER in each
  15  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  16  * If applicable, add the following below this CDDL HEADER, with the
  17  * fields enclosed by brackets "[]" replaced with your own identifying
  18  * information: Portions Copyright [yyyy] [name of copyright owner]
  19  *
  20  * CDDL HEADER END
  21  */

  23 /*
  24  * Copyright 2014 Gary Mills
  25  * Copyright 2006 Sun Microsystems, Inc.  All rights reserved.
  26  * Use is subject to license terms.
  27  */

  29 /* Copyright (c) 1983, 1984, 1985, 1986, 1987, 1988, 1989 AT&T */
  30 /* All Rights Reserved */
  31 /*
  32  * Portions of this source code were derived from Berkeley
  33  * 4.3 BSD under license from the Regents of the University of
  34  * California.
  35  */

  36 #pragma ident   "%Z%%M% %I%     %E% SMI"

  37 /*
  38  * interface to rpcbind rpc service.
  39  */

  41 #include "mt.h"
  42 #include "rpc_mt.h"
  43 #include <assert.h>
  44 #include <rpc/rpc.h>
  45 #include <rpc/rpcb_prot.h>
  46 #include <netconfig.h>
  47 #include <netdir.h>
  48 #include <rpc/nettype.h>
  49 #include <syslog.h>
  50 #ifdef PORTMAP
  51 #include <netinet/in.h>         /* FOR IPPROTO_TCP/UDP definitions */
  52 #include <rpc/pmap_prot.h>
  53 #endif
  55 #ifdef ND_DEBUG
  56 #include <stdio.h>
  57 #endif
  54 #include <sys/utsname.h>
  55 #include <errno.h>
  56 #include <stdlib.h>
```



```
  57 #include <string.h>
  58 #include <unistd.h>

  60 static struct timeval tottimeout = { 60, 0 };
  61 static const struct timeval rmttimeout = { 3, 0 };
  62 static struct timeval rpcbrmttime = { 15, 0 };

  64 extern bool_t xdr_wrapstring(XDR *, char **);

  66 static const char nullstring[] = "\000";

  68 extern CLIENT *_clnt_tli_create_timed(int, const struct netconfig *,
  69                         struct netbuf *, rpcprog_t, rpcvers_t, uint_t, uint_t,
  70                         const struct timeval *);

  72 static CLIENT *_getclnthandle_timed(char *, struct netconfig *, char **,
  73                         struct timeval *);


  76 /*
  77  * The life time of a cached entry should not exceed 5 minutes
  78  * since automountd attempts an unmount every 5 minutes.
  79  * It is arbitrarily set a little lower (3 min = 180 sec)
  80  * to reduce the time during which an entry is stale.
  81  */
  82 #define CACHE_TTL 180
  83 #define CACHESIZE 6

  85 struct address_cache {
  86         char *ac_host;
  87         char *ac_netid;
  88         char *ac_uaddr;
  89         struct netbuf *ac_taddr;
  90         struct address_cache *ac_next;
  91         time_t ac_maxtime;
  92 };
_____unchanged_portion_omitted_

 139 /*
 140  *      It might seem that a reader/writer lock would be more reasonable here.
 141  *      However because getclnthandle(), the only user of the cache functions,
 142  *      may do a delete_cache() operation if a check_cache() fails to return an
 143  *      address useful to clnt_tli_create(), we may as well use a mutex.
 144  */
 145 /*
 146  * As it turns out, if the cache lock is *not* a reader/writer lock, we will
 147  * block all clnt_create's if we are trying to connect to a host that's down,
 148  * since the lock will be held all during that time.
 149  */
 150 extern rwlock_t rpcbaddr_cache_lock;

 152 /*
 153  * The routines check_cache(), add_cache(), delete_cache() manage the
 154  * cache of rpcbind addresses for (host, netid).
 155  */

 157 static struct address_cache *
 158 check_cache(char *host, char *netid)
 159 {
 160         struct address_cache *cptr;

 162         /* READ LOCK HELD ON ENTRY: rpcbaddr_cache_lock */

 164         assert(RW_READ_HELD(&rpcbaddr_cache_lock));
 165         for (cptr = front; cptr != NULL; cptr = cptr->ac_next) {
 166                 if ((strcmp(cptr->ac_host, host) == 0) &&
```

```
                    (strcmp(cptr->ac_netid, netid) == 0) &&
                    (time(NULL) <= cptr->ac_maxtime)) {
#ifdef ND_DEBUG
                        fprintf(stderr, "Found cache entry for %s: %s\n",
                                host, netid);
#endif
                        return (cptr);
                }
        }
        return (NULL);
}
```

*unchanged_portion_omitted_*

```
static void
add_cache(char *host, char *netid, struct netbuf *taddr, char *uaddr)
{
        struct address_cache  *ad_cache, *cptr, *prevptr;

        ad_cache = malloc(sizeof (struct address_cache));
        if (!ad_cache) {
                goto memerr;
        }
        ad_cache->ac_maxtime = time(NULL) + CACHE_TTL;
        ad_cache->ac_host = strdup(host);
        ad_cache->ac_netid = strdup(netid);
        ad_cache->ac_uaddr = uaddr ? strdup(uaddr) : NULL;
        ad_cache->ac_taddr = malloc(sizeof (struct netbuf));
        if (!ad_cache->ac_host || !ad_cache->ac_netid || !ad_cache->ac_taddr ||
            (uaddr && !ad_cache->ac_uaddr)) {
                goto memerr1;
        }

        ad_cache->ac_taddr->len = ad_cache->ac_taddr->maxlen = taddr->len;
        ad_cache->ac_taddr->buf = malloc(taddr->len);
        if (ad_cache->ac_taddr->buf == NULL) {
                goto memerr1;
        }

        (void) memcpy(ad_cache->ac_taddr->buf, taddr->buf, taddr->len);
#ifdef ND_DEBUG
        (void) fprintf(stderr, "Added to cache: %s : %s\n", host, netid);
#endif

/* VARIABLES PROTECTED BY rpcbaddr_cache_lock:  cptr */

        (void) rw_wrlock(&rpcbaddr_cache_lock);
        if (cachesize < CACHESIZE) {
                ad_cache->ac_next = front;
                front = ad_cache;
                cachesize++;
        } else {
                /* Free the last entry */
                cptr = front;
                prevptr = NULL;
                while (cptr->ac_next) {
                        prevptr = cptr;
                        cptr = cptr->ac_next;
                }

#ifdef ND_DEBUG
                fprintf(stderr, "Deleted from cache: %s : %s\n",
                        cptr->ac_host, cptr->ac_netid);
#endif
                free(cptr->ac_host);
                free(cptr->ac_netid);
                free(cptr->ac_taddr->buf);
```



```
                free(cptr->ac_taddr);
                if (cptr->ac_uaddr)
                        free(cptr->ac_uaddr);

                if (prevptr) {
                        prevptr->ac_next = NULL;
                        ad_cache->ac_next = front;
                        front = ad_cache;
                } else {
                        front = ad_cache;
                        ad_cache->ac_next = NULL;
                }
                free(cptr);
        }
        (void) rw_unlock(&rpcbaddr_cache_lock);
        return;
memerr1:
        if (ad_cache->ac_host)
                free(ad_cache->ac_host);
        if (ad_cache->ac_netid)
                free(ad_cache->ac_netid);
        if (ad_cache->ac_uaddr)
                free(ad_cache->ac_uaddr);
        if (ad_cache->ac_taddr)
                free(ad_cache->ac_taddr);
        free(ad_cache);
memerr:
        syslog(LOG_ERR, "add_cache : out of memory.");
}
```

*unchanged_portion_omitted_*

```
/*
 * Same as getclnthandle() except it takes an extra timeout argument.
 * This is for bug 4049792: clnt_create_timed does not timeout.
 *
 * If tp is NULL, use default timeout to get a client handle.
 */
static CLIENT *
_getclnthandle_timed(char *host, struct netconfig *nconf, char **targaddr,
                                                        struct timeval *tp)
{
        CLIENT *client = NULL;
        struct netbuf *addr;
        struct netbuf addr_to_delete;
        struct nd_addrlist *nas;
        struct nd_hostserv rpcbind_hs;
        struct address_cache *ad_cache;
        char *tmpaddr;
        int neterr;
        int j;

/* VARIABLES PROTECTED BY rpcbaddr_cache_lock:  ad_cache */

        /* Get the address of the rpcbind.  Check cache first */
        addr_to_delete.len = 0;
        (void) rw_rdlock(&rpcbaddr_cache_lock);
        ad_cache = check_cache(host, nconf->nc_netid);
        if (ad_cache != NULL) {
                addr = ad_cache->ac_taddr;
                client = _clnt_tli_create_timed(RPC_ANYFD, nconf, addr,
                    RPCBPROG, RPCBVERS4, 0, 0, tp);
                if (client != NULL) {
                        if (targaddr) {
                                /*
                                 * case where a client handle is created
                                 * without a targaddr and the handle is
```

```
323                                  * requested with a targaddr
324                                  */
325                                 if (ad_cache->ac_uaddr != NULL) {
326                                         *targaddr = strdup(ad_cache->ac_uaddr);
327                                         if (*targaddr == NULL) {
328                                                 syslog(LOG_ERR,
329                                                 "_getclnthandle_timed: strdup "
330                                                 "failed.");
331                                                 rpc_createerr.cf_stat =
332                                                     RPC_SYSTEMERROR;
333                                                 (void) rw_unlock(
334                                                     &rpcbaddr_cache_lock);
335                                                 return (NULL);
336                                         }
337                                 } else {
338                                         *targaddr = NULL;
339                                 }
340                         }
341                         (void) rw_unlock(&rpcbaddr_cache_lock);
342                         return (client);
343                 }
344                 if (rpc_createerr.cf_stat == RPC_SYSTEMERROR) {
345                         (void) rw_unlock(&rpcbaddr_cache_lock);
346                         return (NULL);
347                 }
348                 addr_to_delete.len = addr->len;
349                 addr_to_delete.buf = malloc(addr->len);
350                 if (addr_to_delete.buf == NULL) {
351                         addr_to_delete.len = 0;
352                 } else {
353                         (void) memcpy(addr_to_delete.buf, addr->buf, addr->len);
354                 }
355         }
356         (void) rw_unlock(&rpcbaddr_cache_lock);
357         if (addr_to_delete.len != 0) {
358                 /*
359                  * Assume this may be due to cache data being
360                  *  outdated
361                  */
362                 (void) rw_wrlock(&rpcbaddr_cache_lock);
363                 delete_cache(&addr_to_delete);
364                 (void) rw_unlock(&rpcbaddr_cache_lock);
365                 free(addr_to_delete.buf);
366         }
367         rpcbind_hs.h_host = host;
368         rpcbind_hs.h_serv = "rpcbind";
384 #ifdef ND_DEBUG
385         fprintf(stderr, "rpcbind client routines: diagnostics :\n");
386         fprintf(stderr, "\tGetting address for (%s, %s, %s) ... \n",
387                 rpcbind_hs.h_host, rpcbind_hs.h_serv, nconf->nc_netid);
388 #endif

370         if ((neterr = netdir_getbyname(nconf, &rpcbind_hs, &nas)) != 0) {
371                 if (neterr == ND_NOHOST)
372                         rpc_createerr.cf_stat = RPC_UNKNOWNHOST;
373                 else
374                         rpc_createerr.cf_stat = RPC_N2AXLATEFAILURE;
375                 return (NULL);
376         }
377         /* XXX nas should perhaps be cached for better performance */

379         for (j = 0; j < nas->n_cnt; j++) {
380                 addr = &(nas->n_addrs[j]);
401 #ifdef ND_DEBUG
402 {
403         int i;
```



```
404         char *ua;

406         ua = taddr2uaddr(nconf, &(nas->n_addrs[j]));
407         fprintf(stderr, "Got it [%s]\n", ua);
408         free(ua);

410         fprintf(stderr, "\tnetbuf len = %d, maxlen = %d\n",
411                 addr->len, addr->maxlen);
412         fprintf(stderr, "\tAddress is ");
413         for (i = 0; i < addr->len; i++)
414                 fprintf(stderr, "%u.", addr->buf[i]);
415         fprintf(stderr, "\n");
416 }
417 #endif
381         client = _clnt_tli_create_timed(RPC_ANYFD, nconf, addr, RPCBPROG,
382             RPCBVERS4, 0, 0, tp);
383         if (client)
384                 break;
385         }
423 #ifdef ND_DEBUG
424         if (!client) {
425                 clnt_pcreateerror("rpcbind clnt interface");
426         }
427 #endif

387         if (client) {
388                 tmpaddr = targaddr ? taddr2uaddr(nconf, addr) : NULL;
389                 add_cache(host, nconf->nc_netid, addr, tmpaddr);
390                 if (targaddr) {
391                         *targaddr = tmpaddr;
392                 }
393         }
394         netdir_free((char *)nas, ND_ADDRLIST);
395         return (client);
396 }

398 /*
399  * This routine will return a client handle that is connected to the local
400  * rpcbind. Returns NULL on error.
442  * rpcbind. Returns NULL on error and free's everything.
401  */
402 static CLIENT *
403 local_rpcb(void)
404 {
405         static struct netconfig *loopnconf;
406         static char *hostname;
407         extern mutex_t loopnconf_lock;

409 /* VARIABLES PROTECTED BY loopnconf_lock: hostname loopnconf */
451 /* VARIABLES PROTECTED BY loopnconf_lock: loopnconf */
410         (void) mutex_lock(&loopnconf_lock);
411         if (loopnconf == NULL) {
412                 struct utsname utsname;
413                 struct netconfig *nconf, *tmpnconf = NULL;
414                 void *nc_handle;

416                 if (hostname == NULL) {
417 #if defined(__i386) && !defined(__amd64)
418                         if ((_nuname(&utsname) == -1) ||
419                             ((hostname = strdup(utsname.nodename)) == NULL)) {
420 #else
421                         if ((uname(&utsname) == -1) ||
422                             ((hostname = strdup(utsname.nodename)) == NULL)) {
423 #endif
464                             ((hostname = strdup(utsname.nodename)) == NULL)) {
424                                 syslog(LOG_ERR, "local_rpcb : strdup failed.");
```

```
425                                 rpc_createerr.cf_stat = RPC_UNKNOWNHOST;
426                                 (void) mutex_unlock(&loopnconf_lock);
427                                 return (NULL);
428                         }
429                         /* hostname is never freed */
430                 }
431                 nc_handle = setnetconfig();
432                 if (nc_handle == NULL) {
433                         /* fails to open netconfig file */
434                         rpc_createerr.cf_stat = RPC_UNKNOWNPROTO;
435                         (void) mutex_unlock(&loopnconf_lock);
436                         return (NULL);
437                 }
438                 while (nconf = getnetconfig(nc_handle)) {
439                         if (strcmp(nconf->nc_protofmly, NC_LOOPBACK) == 0) {
440                                 tmpnconf = nconf;
441                                 if (nconf->nc_semantics == NC_TPI_CLTS)
442                                         break;
443                         }
444                 }
445                 if (tmpnconf == NULL) {
446                         rpc_createerr.cf_stat = RPC_UNKNOWNPROTO;
447                         (void) mutex_unlock(&loopnconf_lock);
448                         return (NULL);
449                 }
450                 loopnconf = getnetconfigent(tmpnconf->nc_netid);
451                 /* loopnconf is never freed */
452                 (void) endnetconfig(nc_handle);
453         }
454         (void) mutex_unlock(&loopnconf_lock);
455         return (getclnthandle(hostname, loopnconf, NULL));
456 }
```

unchanged_portion_omitted_

```
545 /*
546  * From the merged list, find the appropriate entry
547  */
548 static struct netbuf *
549 got_entry(rpcb_entry_list_ptr relp, struct netconfig *nconf)
550 {
551         struct netbuf *na = NULL;
552         rpcb_entry_list_ptr sp;
553         rpcb_entry *rmap;

555         for (sp = relp; sp != NULL; sp = sp->rpcb_entry_next) {
556                 rmap = &sp->rpcb_entry_map;
557                 if ((strcmp(nconf->nc_proto, rmap->r_nc_proto) == 0) &&
558                     (strcmp(nconf->nc_protofmly, rmap->r_nc_protofmly) == 0) &&
559                     (nconf->nc_semantics == rmap->r_nc_semantics) &&
560                     (rmap->r_maddr != NULL) && (rmap->r_maddr[0] != NULL)) {
561                         na = uaddr2taddr(nconf, rmap->r_maddr);
562 #ifdef ND_DEBUG
603                         fprintf(stderr, "\tRemote address is [%s].\n",
604                                 rmap->r_maddr);
605                         if (!na)
606                                 fprintf(stderr,
607                                     "\tCouldn't resolve remote address!\n");
608 #endif
562                         break;
563                 }
564         }
565         return (na);
566 }
```

unchanged_portion_omitted_


```
641 /*
642  * An internal function which optimizes rpcb_getaddr function.  It returns
643  * the universal address of the remote service or NULL.  It also optionally
689  * An internal function which optimizes rpcb_getaddr function.  It also
644  * returns the client handle that it uses to contact the remote rpcbind.
645  * The caller will re-purpose the client to contact the remote service.
646  *
647  * The algorithm used: First try version 4.  Then try version 3 (svr4).
648  * Finally, if the transport is TCP or UDP, try version 2 (portmap).
649  * Version 4 is now available with all current systems on the network.
692  * The algorithm used: If the transports is TCP or UDP, it first tries
693  * version 2 (portmap), 4 and then 3 (svr4).  This order should be
694  * changed in the next OS release to 4, 2 and 3.  We are assuming that by
695  * that time, version 4 would be available on many machines on the network.
650  * With this algorithm, we get performance as well as a plan for
651  * obsoleting version 2.
652  *
699  * For all other transports, the algorithm remains as 4 and then 3.
700  *
653  * XXX: Due to some problems with t_connect(), we do not reuse the same client
654  * handle for COTS cases and hence in these cases we do not return the
655  * client handle.  This code will change if t_connect() ever
656  * starts working properly.  Also look under clnt_vc.c.
657  */
658 struct netbuf *
659 __rpcb_findaddr_timed(rpcprog_t program, rpcvers_t version,
660         struct netconfig *nconf, char *host, CLIENT **clpp, struct timeval *tp)
661 {
662         static bool_t check_rpcbind = TRUE;
663         CLIENT *client = NULL;
664         RPCB parms;
665         enum clnt_stat clnt_st;
666         char *ua = NULL;
667         uint_t vers;
668         struct netbuf *address = NULL;
669         void *handle;
670         rpcb_entry_list_ptr relp = NULL;
671         bool_t tmp_client = FALSE;
717         uint_t start_vers = RPCBVERS4;

673         /* parameter checking */
674         if (nconf == NULL) {
675                 rpc_createerr.cf_stat = RPC_UNKNOWNPROTO;
676                 return (NULL);
677         }

679         parms.r_addr = NULL;

681         /*
682          * Use default total timeout if no timeout is specified.
683          */
684         if (tp == NULL)
685                 tp = &tottimeout;

733 #ifdef PORTMAP
734         /* Try version 2 for TCP or UDP */
735         if (strcmp(nconf->nc_protofmly, NC_INET) == 0) {
736                 ushort_t port = 0;
737                 struct netbuf remote;
738                 uint_t pmapvers = 2;
739                 struct pmap pmapparms;

687         /*
742                  * Try UDP only - there are some portmappers out
743                  * there that use UDP only.
744                  */
```

```
745                 if (strcmp(nconf->nc_proto, NC_TCP) == 0) {
746                         struct netconfig *newnconf;
747                         void *handle;

749                         if ((handle = __rpc_setconf("udp")) == NULL) {
750                                 rpc_createerr.cf_stat = RPC_UNKNOWNPROTO;
751                                 return (NULL);
752                         }

754                         /*
755                          * The following to reinforce that you can
756                          * only request for remote address through
757                          * the same transport you are requesting.
758                          * ie. requesting unversial address
759                          * of IPv4 has to be carried through IPv4.
760                          * Can't use IPv6 to send out the request.
761                          * The mergeaddr in rpcbind can't handle
762                          * this.
763                          */
764                         for (;;) {
765                                 if ((newnconf = __rpc_getconf(handle))
766                                                                 == NULL) {
767                                         __rpc_endconf(handle);
768                                         rpc_createerr.cf_stat =
769                                             RPC_UNKNOWNPROTO;
770                                         return (NULL);
771                                 }
772                                 /*
773                                  * here check the protocol family to
774                                  * be consistent with the request one
775                                  */
776                                 if (strcmp(newnconf->nc_protofmly,
777                                     nconf->nc_protofmly) == NULL)
778                                         break;
779                         }

781                         client = _getclnthandle_timed(host, newnconf,
782                                         &parms.r_addr, tp);
783                         __rpc_endconf(handle);
784                 } else {
785                         client = _getclnthandle_timed(host, nconf,
786                                         &parms.r_addr, tp);
787                 }
788                 if (client == NULL)
789                         return (NULL);

791                 /*
792                  * Set version and retry timeout.
793                  */
794                 CLNT_CONTROL(client, CLSET_RETRY_TIMEOUT, (char *)&rpcbrmttime);
795                 CLNT_CONTROL(client, CLSET_VERS, (char *)&pmapvers);

797                 pmapparms.pm_prog = program;
798                 pmapparms.pm_vers = version;
799                 pmapparms.pm_prot = strcmp(nconf->nc_proto, NC_TCP) ?
800                                     IPPROTO_UDP : IPPROTO_TCP;
801                 pmapparms.pm_port = 0;  /* not needed */
802                 clnt_st = CLNT_CALL(client, PMAPPROC_GETPORT,
803                                     (xdrproc_t)xdr_pmap, (caddr_t)&pmapparms,
804                                     (xdrproc_t)xdr_u_short, (caddr_t)&port,
805                                     *tp);
806                 if (clnt_st != RPC_SUCCESS) {
807                         if ((clnt_st == RPC_PROGVERSMISMATCH) ||
808                             (clnt_st == RPC_PROGUNAVAIL))
809                                 goto try_rpcbind; /* Try different versions */
810                         rpc_createerr.cf_stat = RPC_PMAPFAILURE;
```



```
811                         clnt_geterr(client, &rpc_createerr.cf_error);
812                         goto error;
813                 } else if (port == 0) {
814                         address = NULL;
815                         rpc_createerr.cf_stat = RPC_PROGNOTREGISTERED;
816                         goto error;
817                 }
818                 port = htons(port);
819                 CLNT_CONTROL(client, CLGET_SVC_ADDR, (char *)&remote);
820                 if (((address = malloc(sizeof (struct netbuf))) == NULL) ||
821                     ((address->buf = malloc(remote.len)) == NULL)) {
822                         rpc_createerr.cf_stat = RPC_SYSTEMERROR;
823                         clnt_geterr(client, &rpc_createerr.cf_error);
824                         if (address) {
825                                 free(address);
826                                 address = NULL;
827                         }
828                         goto error;
829                 }
830                 (void) memcpy(address->buf, remote.buf, remote.len);
831                 (void) memcpy(&address->buf[sizeof (short)], &port,
832                                                         sizeof (short));
833                 address->len = address->maxlen = remote.len;
834                 goto done;
835         }
836 #endif

838 try_rpcbind:
839         /*
688          * Check if rpcbind is up.  This prevents needless delays when
689          * accessing applications such as the keyserver while booting
690          * disklessly.
691          */
692         if (check_rpcbind && strcmp(nconf->nc_protofmly, NC_LOOPBACK) == 0) {
693                 if (!__rpcbind_is_up()) {
694                         rpc_createerr.cf_stat = RPC_PMAPFAILURE;
695                         rpc_createerr.cf_error.re_errno = 0;
696                         rpc_createerr.cf_error.re_terrno = 0;
697                         goto error;
698                 }
699                 check_rpcbind = FALSE;
700         }

702         /*
703          * First try version 4.
855          * Now we try version 4 and then 3.
856          * We also send the remote system the address we used to
857          * contact it in case it can help to connect back with us
704          */
705         parms.r_prog = program;
706         parms.r_vers = version;
707         parms.r_owner = (char *)&nullstring[0]; /* not needed; */
708         /* just for xdring */
709         parms.r_netid = nconf->nc_netid; /* not really needed */

711         /*
712          * If a COTS transport is being used, try getting address via CLTS
713          * transport.  This works only with version 4.
714          */
715         if (nconf->nc_semantics == NC_TPI_COTS_ORD ||
716             nconf->nc_semantics == NC_TPI_COTS) {
717                 tmp_client = TRUE;
718                 handle = __rpc_setconf("datagram_v");
719         } else {
720                 handle = __rpc_setconf(nconf->nc_proto);
721         }
```

```
723         if (handle != NULL) {
871                 void *handle;
724                 struct netconfig *nconf_clts;
873                 rpcb_entry_list_ptr relp = NULL;

726                 while ((nconf_clts = __rpc_getconf(handle)) != NULL) {
875                 if (client == NULL) {
876                         /* This did not go through the above PORTMAP/TCP code */
877                         if ((handle = __rpc_setconf("datagram_v")) != NULL) {
878                                 while ((nconf_clts = __rpc_getconf(handle))
879                                     != NULL) {
727                         if (strcmp(nconf_clts->nc_protofmly,
728                             nconf->nc_protofmly) != 0) {
729                                 continue;
730                         }
731                         client = _getclnthandle_timed(host, nconf_clts,
732                             &parms.r_addr, tp);
884                                         client = _getclnthandle_timed(host,
885                                             nconf_clts, &parms.r_addr,
886                                             tp);
733                         break;
734                 }
735                 __rpc_endconf(handle);
736         }
737         if (client != NULL) {

739                 /* Set rpcbind version 4 */
891                         if (client == NULL)
892                                 goto regular_rpcbind;   /* Go the regular way */
893                 } else {
894                         /* This is a UDP PORTMAP handle.  Change to version 4 */
740                 vers = RPCBVERS4;
741                 CLNT_CONTROL(client, CLSET_VERS, (char *)&vers);

897                 }
743                 /*
744                  * We also send the remote system the address we used to
745                  * contact it in case it can help it connect back with us
746                  */
747                 if (parms.r_addr == NULL) {
748                         parms.r_addr = strdup(""); /* for XDRing */
749                         if (parms.r_addr == NULL) {
750                                 syslog(LOG_ERR, "__rpcb_findaddr_timed: "
751                                     "strdup failed.");
752                                 rpc_createerr.cf_stat = RPC_SYSTEMERROR;
908                                 address = NULL;
753                                 goto error;
754                         }
755                 }

757                 CLNT_CONTROL(client, CLSET_RETRY_TIMEOUT,
758                     (char *)&rpcbrmttime);
913                 CLNT_CONTROL(client, CLSET_RETRY_TIMEOUT, (char *)&rpcbrmttime);

760                 clnt_st = CLNT_CALL(client, RPCBPROC_GETADDRLIST,
761                     (xdrproc_t)xdr_rpcb, (char *)&parms,
762                     (xdrproc_t)xdr_rpcb_entry_list_ptr, (char *)&relp, *tp);
763                 switch (clnt_st) {
764                 case RPC_SUCCESS: /* Call succeeded */
765                         address = got_entry(relp, nconf);
917                                 (xdrproc_t)xdr_rpcb_entry_list_ptr,
918                                 (char *)&relp, *tp);
919                 if (clnt_st == RPC_SUCCESS) {
920                         if (address = got_entry(relp, nconf)) {
766                         xdr_free((xdrproc_t)xdr_rpcb_entry_list_ptr,
```



```
767                             (char *)&relp);
768                         if (address != NULL) {
769                                 /* Program number and version number matched */
770                                 goto done;
771                         }
772                         /* Program and version not found for this transport */
925                         /* Entry not found for this transport */
926                         xdr_free((xdrproc_t)xdr_rpcb_entry_list_ptr,
927                                 (char *)&relp);
773                         /*
774                          * XXX: should have returned with RPC_PROGUNAVAIL
775                          * or perhaps RPC_PROGNOTREGISTERED error but
929                          * XXX: should have perhaps returned with error but
776                          * since the remote machine might not always be able
777                          * to send the address on all transports, we try the
778                          * regular way with version 3, then 2
932                          * regular way with regular_rpcbind
779                          */
780                         /* Try the next version */
781                         break;
782                 case RPC_PROGVERSMISMATCH: /* RPC protocol mismatch */
783                         clnt_geterr(client, &rpc_createerr.cf_error);
784                         if (rpc_createerr.cf_error.re_vers.low > vers) {
785                                 rpc_createerr.cf_stat = RPC_PROGVERSMISMATCH;
786                                 goto error;  /* a new version, can't handle */
787                         }
788                         /* Try the next version */
789                         break;
790                 case RPC_PROCUNAVAIL: /* Procedure unavailable */
791                 case RPC_PROGUNAVAIL: /* Program not available */
792                 case RPC_TIMEDOUT: /* Call timed out */
793                         /* Try the next version */
794                         break;
795                 default:
934                         goto regular_rpcbind;
935                 } else if ((clnt_st == RPC_PROGVERSMISMATCH) ||
936                             (clnt_st == RPC_PROGUNAVAIL)) {
937                         start_vers = RPCBVERS;  /* Try version 3 now */
938                         goto regular_rpcbind; /* Try different versions */
939                 } else {
796                         rpc_createerr.cf_stat = RPC_PMAPFAILURE;
797                         clnt_geterr(client, &rpc_createerr.cf_error);
798                         goto error;
799                         break;
800                 }
944         }

802         } else {
946 regular_rpcbind:

804                 /* No client */
805                 tmp_client = FALSE;

807         } /* End of version 4 */

809         /* Destroy a temporary client */
810         if (client != NULL && tmp_client) {
948         /* Now the same transport is to be used to get the address */
949         if (client && ((nconf->nc_semantics == NC_TPI_COTS_ORD) ||
950             (nconf->nc_semantics == NC_TPI_COTS))) {
951                 /* A CLTS type of client - destroy it */
811                 CLNT_DESTROY(client);
812                 client = NULL;
813                 free(parms.r_addr);
814                 parms.r_addr = NULL;
815         }
```

```
816         tmp_client = FALSE;

818         /*
819          * Try version 3
820          */

822         /* Now the same transport is to be used to get the address */
823         if (client == NULL) {
824                 client = _getclnthandle_timed(host, nconf, &parms.r_addr, tp);
825         }
960                 if (client == NULL) {
826         address = NULL;
827         if (client != NULL) {
962                         goto error;
963                 }
964         }
828                 if (parms.r_addr == NULL) {
829                         parms.r_addr = strdup("");      /* for XDRing */
830                         if (parms.r_addr == NULL) {
831                                 syslog(LOG_ERR, "__rpcb_findaddr_timed: "
832                                     "strdup failed.");
970                         address = NULL;
833                                 rpc_createerr.cf_stat = RPC_SYSTEMERROR;
834                                 goto error;
835                         }
836                 }

838                 CLNT_CONTROL(client, CLSET_RETRY_TIMEOUT,
839                     (char *)&rpcbrmttime);
840                 vers = RPCBVERS; /* Set the version */
976         /* First try from start_vers and then version 3 (RPCBVERS) */

978         CLNT_CONTROL(client, CLSET_RETRY_TIMEOUT, (char *)&rpcbrmttime);
979         for (vers = start_vers;  vers >= RPCBVERS; vers--) {
980                 /* Set the version */
841                 CLNT_CONTROL(client, CLSET_VERS, (char *)&vers);
842                 clnt_st = CLNT_CALL(client, RPCBPROC_GETADDR,
843                     (xdrproc_t)xdr_rpcb, (char *)&parms,
844                     (xdrproc_t)xdr_wrapstring, (char *)&ua, *tp);
845                 switch (clnt_st) {
846                 case RPC_SUCCESS: /* Call succeeded */
847                         if (ua != NULL) {
848                                 if (ua[0] != '\0') {
849                                         address = uaddr2taddr(nconf, ua);
850                                 }
851                                 xdr_free((xdrproc_t)xdr_wrapstring,
852                                     (char *)&ua);
984                                     (xdrproc_t)xdr_wrapstring,
985                                     (char *)&ua, *tp);
986                 if (clnt_st == RPC_SUCCESS) {
987                         if ((ua == NULL) || (ua[0] == NULL)) {
988                                 if (ua != NULL)
989                                         xdr_free(xdr_wrapstring, (char *)&ua);

854                                 if (address != NULL) {
855                                         goto done;
856                                 }
857                                 /* We don't know about your universal addr */
991                                 /* address unknown */
858                                 rpc_createerr.cf_stat = RPC_PROGNOTREGISTERED;
859                                 goto error;
860                         }
861                         /* Try the next version */
862                         break;
863                 case RPC_PROGVERSMISMATCH: /* RPC protocol mismatch */
864                         clnt_geterr(client, &rpc_createerr.cf_error);
```



```
865                         if (rpc_createerr.cf_error.re_vers.low > vers)
866                                 goto error;  /* a new version, can't handle */
867                         /* Try the next version */
868                         break;
869                 case RPC_PROCUNAVAIL: /* Procedure unavailable */
870                 case RPC_PROGUNAVAIL: /* Program not available */
871                 case RPC_TIMEDOUT: /* Call timed out */
872                         /* Try the next version */
873                         break;
874                 default:
875                         clnt_geterr(client, &rpc_createerr.cf_error);
876                         rpc_createerr.cf_stat = RPC_PMAPFAILURE;
877                         goto error;
878                         break;
879                 }
880         } /* End of version 3 */
995                         address = uaddr2taddr(nconf, ua);
996 #ifdef ND_DEBUG
997                         fprintf(stderr, "\tRemote address is [%s]\n", ua);
998                         if (!address)
999                                 fprintf(stderr,
1000                                     "\tCouldn't resolve remote address!\n");
1001 #endif
1002                        xdr_free((xdrproc_t)xdr_wrapstring, (char *)&ua);

882         /*
883          * Try version 2
884          */

886 #ifdef PORTMAP
887         /* Try version 2 for TCP or UDP */
888         if (strcmp(nconf->nc_protofmly, NC_INET) == 0) {
889                 ushort_t port = 0;
890                 struct netbuf remote;
891                 uint_t pmapvers = 2;
892                 struct pmap pmapparms;

894                 /*
895                  * Try UDP only - there are some portmappers out
896                  * there that use UDP only.
897                  */
898                 if (strcmp(nconf->nc_proto, NC_TCP) == 0) {
899                         struct netconfig *newnconf;

901                         if (client) {
902                                 CLNT_DESTROY(client);
903                                 client = NULL;
904                                 free(parms.r_addr);
905                                 parms.r_addr = NULL;
906                         }
907                         if ((handle = __rpc_setconf("udp")) == NULL) {
908                                 rpc_createerr.cf_stat = RPC_UNKNOWNPROTO;
1004                        if (!address) {
1005                                /* We don't know about your universal address */
1006                                rpc_createerr.cf_stat = RPC_N2AXLATEFAILURE;
909                                 goto error;
910                         }
1009                        goto done;
1010                }
1011                if (clnt_st == RPC_PROGVERSMISMATCH) {
1012                        struct rpc_err rpcerr;

912                         /*
913                          * The following to reinforce that you can
914                          * only request for remote address through
915                          * the same transport you are requesting.
```

```
 916                         * ie. requesting unversial address
 917                         * of IPv4 has to be carried through IPv4.
 918                         * Can't use IPv6 to send out the request.
 919                         * The mergeaddr in rpcbind can't handle
 920                         * this.
 921                         */
 922                        for (;;) {
 923                                if ((newnconf = __rpc_getconf(handle))
 924                                    == NULL) {
 925                                        __rpc_endconf(handle);
 926                                        rpc_createerr.cf_stat =
 927                                            RPC_UNKNOWNPROTO;
1014                        clnt_geterr(client, &rpcerr);
1015                        if (rpcerr.re_vers.low > RPCBVERS4)
1016                                goto error;  /* a new version, can't handle */
1017                } else if (clnt_st != RPC_PROGUNAVAIL) {
1018                        /* Cant handle this error */
 928                                        goto error;
 929                                }
 930                                /*
 931                                 * here check the protocol family to
 932                                 * be consistent with the request one
 933                                 */
 934                                if (strcmp(newnconf->nc_protofmly,
 935                                    nconf->nc_protofmly) == NULL)
 936                                        break;
 937                        }

 939                        client = _getclnthandle_timed(host, newnconf,
 940                            &parms.r_addr, tp);
 941                        __rpc_endconf(handle);
 942                        tmp_client = TRUE;
 943                }
 944                if (client == NULL)
1023        if ((address == NULL) || (address->len == 0)) {
 945                        rpc_createerr.cf_stat = RPC_PROGNOTREGISTERED;
 946                        tmp_client = FALSE;
 947                        goto error;

 949                /*
 950                 * Set version and retry timeout.
 951                 */
 952                CLNT_CONTROL(client, CLSET_RETRY_TIMEOUT, (char *)&rpcbrmttime);
 953                CLNT_CONTROL(client, CLSET_VERS, (char *)&pmapvers);

 955                pmapparms.pm_prog = program;
 956                pmapparms.pm_vers = version;
 957                pmapparms.pm_prot = strcmp(nconf->nc_proto, NC_TCP) ?
 958                    IPPROTO_UDP : IPPROTO_TCP;
 959                pmapparms.pm_port = 0;  /* not needed */
 960                clnt_st = CLNT_CALL(client, PMAPPROC_GETPORT,
 961                    (xdrproc_t)xdr_pmap, (caddr_t)&pmapparms,
 962                    (xdrproc_t)xdr_u_short, (caddr_t)&port, *tp);
 963                if (clnt_st != RPC_SUCCESS) {
 964                        rpc_createerr.cf_stat = RPC_PMAPFAILURE;
 965                        clnt_geterr(client, &rpc_createerr.cf_error);
 966                        goto error;
 967                } else if (port == 0) {
 968                        rpc_createerr.cf_stat = RPC_PROGNOTREGISTERED;
 969                        goto error;
 970                }
 971                port = htons(port);
 972                CLNT_CONTROL(client, CLGET_SVC_ADDR, (char *)&remote);
 973                if (((address = malloc(sizeof (struct netbuf))) == NULL) ||
 974                    ((address->buf = malloc(remote.len)) == NULL)) {
 975                        rpc_createerr.cf_stat = RPC_SYSTEMERROR;
```



```
 976                        clnt_geterr(client, &rpc_createerr.cf_error);
 977                        if (address != NULL) {
 978                                free(address);
 979                                address = NULL;
 980                        }
 981                        goto error;
 982                }
 983                (void) memcpy(address->buf, remote.buf, remote.len);
 984                (void) memcpy(&address->buf[sizeof (short)], &port,
 985                    sizeof (short));
 986                address->len = address->maxlen = remote.len;
 987                goto done;
 988        }
 989 #endif

 991 error:
 992        /* Return NULL address and NULL client */
 993        address = NULL;
 994        if (client) {
 995                CLNT_DESTROY(client);
 996                client = NULL;
 997        }

 999 done:
1000        /* Return an address and optional client */
1001        if (client != NULL && tmp_client) {
1002                /* This client is the temporary one */
1034        if (nconf->nc_semantics != NC_TPI_CLTS) {
1035                /* This client is the connectionless one */
1036                if (client) {
1003                CLNT_DESTROY(client);
1004                client = NULL;
1005        }
1040        }
1006        if (clpp) {
1007                *clpp = client;
1008        } else if (client) {
1009                CLNT_DESTROY(client);
1010        }
1011        if (parms.r_addr)
1012                free(parms.r_addr);
1013        return (address);
1014 }
```

_____*unchanged_portion_omitted_*