```
**********************************************************
   35819 Mon Jul 21 19:19:06 2014
new/usr/src/lib/libnsl/rpc/rpcb_clnt.c
4729 __rpcb_findaddr_timed should try rpcbind protocol 4 first
Reviewed by: Marcel Telka <marcel@telka.sk>
**********************************************************
```

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

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

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

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

/*
 * interface to rpcbind rpc service.
 */

#include "mt.h"
#include "rpc_mt.h"
#include <assert.h>
#include <rpc/rpc.h>
#include <rpc/rpcb_prot.h>
#include <netconfig.h>
#include <netdir.h>
#include <netdb.h>
#include <rpc/nettype.h>
#include <syslog.h>
#ifdef PORTMAP
#include <netinet/in.h>         /* FOR IPPROTO_TCP/UDP definitions */
#include <rpc/pmap_prot.h>
#endif
#ifdef ND_DEBUG
#include <stdio.h>
#endif
#include <sys/utsname.h>
```



```
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

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

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

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

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

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


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

struct address_cache {
        char *ac_host;
        char *ac_netid;
        char *ac_uaddr;
        struct netbuf *ac_taddr;
        struct address_cache *ac_next;
        time_t ac_maxtime;
};
______unchanged_portion_omitted_

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

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

static struct address_cache *
check_cache(char *host, char *netid)
{
        struct address_cache *cptr;

        /* READ LOCK HELD ON ENTRY: rpcbaddr_cache_lock */

        assert(RW_READ_HELD(&rpcbaddr_cache_lock));
```

```
        for (cptr = front; cptr != NULL; cptr = cptr->ac_next) {
                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) {
                                /*
```

```
322                                 * case where a client handle is created
323                                 * without a targaddr and the handle is
324                                 * requested with a targaddr
325                                 */
326                                if (ad_cache->ac_uaddr != NULL) {
327                                        *targaddr = strdup(ad_cache->ac_uaddr);
328                                        if (*targaddr == NULL) {
329                                                syslog(LOG_ERR,
330                                                "_getclnthandle_timed: strdup "
331                                                "failed.");
332                                                rpc_createerr.cf_stat =
333                                                    RPC_SYSTEMERROR;
334                                                (void) rw_unlock(
335                                                    &rpcbaddr_cache_lock);
336                                                return (NULL);
337                                        }
338                                } else {
339                                        *targaddr = NULL;
340                                }
341                        }
342                        (void) rw_unlock(&rpcbaddr_cache_lock);
343                        return (client);
344                }
345                if (rpc_createerr.cf_stat == RPC_SYSTEMERROR) {
346                        (void) rw_unlock(&rpcbaddr_cache_lock);
347                        return (NULL);
348                }
349                addr_to_delete.len = addr->len;
350                addr_to_delete.buf = malloc(addr->len);
351                if (addr_to_delete.buf == NULL) {
352                        addr_to_delete.len = 0;
353                } else {
354                        (void) memcpy(addr_to_delete.buf, addr->buf, addr->len);
355                }
356        }
357        (void) rw_unlock(&rpcbaddr_cache_lock);
358        if (addr_to_delete.len != 0) {
359                /*
360                 * Assume this may be due to cache data being
361                 *  outdated
362                 */
363                (void) rw_wrlock(&rpcbaddr_cache_lock);
364                delete_cache(&addr_to_delete);
365                (void) rw_unlock(&rpcbaddr_cache_lock);
366                free(addr_to_delete.buf);
367        }
368        rpcbind_hs.h_host = host;
369        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

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

380        for (j = 0; j < nas->n_cnt; j++) {
381                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
382        client = _clnt_tli_create_timed(RPC_ANYFD, nconf, addr, RPCBPROG,
383            RPCBVERS4, 0, 0, tp);
384        if (client)
385                break;
386        }
423 #ifdef ND_DEBUG
424        if (!client) {
425                clnt_pcreateerror("rpcbind clnt interface");
426        }
427 #endif

388        if (client) {
389                tmpaddr = targaddr ? taddr2uaddr(nconf, addr) : NULL;
390                add_cache(host, nconf->nc_netid, addr, tmpaddr);
391                if (targaddr) {
392                        *targaddr = tmpaddr;
393                }
394        }
395        netdir_free((char *)nas, ND_ADDRLIST);
396        return (client);
397 }

399 /*
400  * This routine will return a client handle that is connected to the local
401  * rpcbind. Returns NULL on error.
442  * rpcbind. Returns NULL on error and free's everything.
402  */
403 static CLIENT *
404 local_rpcb(void)
405 {
406        static struct netconfig *loopnconf;
407        static char hostname[MAXHOSTNAMELEN + 1];
448        static char *hostname;
408        extern mutex_t loopnconf_lock;

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

416                if ((hostname[0] == '\0') && (gethostname(hostname,
417                    sizeof (hostname)) < 0)) {
418                        syslog(LOG_ERR, "local_rpcb: gethostname failed.");
458                if (hostname == NULL) {
459 #if defined(__i386) && !defined(__amd64)
460                        if ((_nuname(&utsname) == -1) ||
461 #else
462                        if ((uname(&utsname) == -1) ||
```

```
#endif
                        ((hostname = strdup(utsname.nodename)) == NULL)) {
                            syslog(LOG_ERR, "local_rpcb : strdup failed.");
                        rpc_createerr.cf_stat = RPC_UNKNOWNHOST;
                        (void) mutex_unlock(&loopnconf_lock);
                        return (NULL);
                }
                }
                nc_handle = setnetconfig();
                if (nc_handle == NULL) {
                        /* fails to open netconfig file */
                        rpc_createerr.cf_stat = RPC_UNKNOWNPROTO;
                        (void) mutex_unlock(&loopnconf_lock);
                        return (NULL);
                }
                while (nconf = getnetconfig(nc_handle)) {
                        if (strcmp(nconf->nc_protofmly, NC_LOOPBACK) == 0) {
                                tmpnconf = nconf;
                                if (nconf->nc_semantics == NC_TPI_CLTS)
                                        break;
                        }
                }
                if (tmpnconf == NULL) {
                        rpc_createerr.cf_stat = RPC_UNKNOWNPROTO;
                        (void) mutex_unlock(&loopnconf_lock);
                        return (NULL);
                }
                loopnconf = getnetconfigent(tmpnconf->nc_netid);
                /* loopnconf is never freed */
                (void) endnetconfig(nc_handle);
        }
        (void) mutex_unlock(&loopnconf_lock);
        return (getclnthandle(hostname, loopnconf, NULL));
}
_____unchanged_portion_omitted_

/*
 * From the merged list, find the appropriate entry
 */
static struct netbuf *
got_entry(rpcb_entry_list_ptr relp, struct netconfig *nconf)
{
        struct netbuf *na = NULL;
        rpcb_entry_list_ptr sp;
        rpcb_entry *rmap;

        for (sp = relp; sp != NULL; sp = sp->rpcb_entry_next) {
                rmap = &sp->rpcb_entry_map;
                if ((strcmp(nconf->nc_proto, rmap->r_nc_proto) == 0) &&
                    (strcmp(nconf->nc_protofmly, rmap->r_nc_protofmly) == 0) &&
                    (nconf->nc_semantics == rmap->r_nc_semantics) &&
                    (rmap->r_maddr != NULL) && (rmap->r_maddr[0] != NULL)) {
                        na = uaddr2taddr(nconf, rmap->r_maddr);
#ifdef ND_DEBUG
                        fprintf(stderr, "\tRemote address is [%s].\n",
                                rmap->r_maddr);
                        if (!na)
                                fprintf(stderr,
                                    "\tCouldn't resolve remote address!\n");
#endif
                        break;
                }
        }
        return (na);
}
```



```
/*
 * Quick check to see if rpcbind is up.  Tries to connect over
 * local transport.
 */
bool_t
__rpcbind_is_up(void)
{
        char hostname[MAXHOSTNAMELEN + 1];
        struct utsname name;
        char uaddr[SYS_NMLN];
        struct netbuf *addr;
        int fd;
        struct t_call *sndcall;
        struct netconfig *netconf;
        bool_t res;

        if (gethostname(hostname, sizeof (hostname)) < 0)
#if defined(__i386) && !defined(__amd64)
        if (_nuname(&name) == -1)
#else
        if (uname(&name) == -1)
#endif
                return (TRUE);

        if ((fd = t_open("/dev/ticotsord", O_RDWR, NULL)) == -1)
                return (TRUE);

        if (t_bind(fd, NULL, NULL) == -1) {
                (void) t_close(fd);
                return (TRUE);
        }

        /* LINTED pointer cast */
        if ((sndcall = (struct t_call *)t_alloc(fd, T_CALL, 0)) == NULL) {
                (void) t_close(fd);
                return (TRUE);
        }

        uaddr[0] = '\0';
        (void) strlcpy(uaddr, hostname, sizeof (uaddr) - 5);
        (void) strcpy(uaddr, name.nodename);
        (void) strcat(uaddr, ".rpc");
        if ((netconf = getnetconfigent("ticotsord")) == NULL) {
                (void) t_free((char *)sndcall, T_CALL);
                (void) t_close(fd);
                return (FALSE);
        }
        addr = uaddr2taddr(netconf, uaddr);
        freenetconfigent(netconf);
        if (addr == NULL || addr->buf == NULL) {
                if (addr)
                        free(addr);
                (void) t_free((char *)sndcall, T_CALL);
                (void) t_close(fd);
                return (FALSE);
        }
        sndcall->addr.maxlen = addr->maxlen;
        sndcall->addr.len = addr->len;
        sndcall->addr.buf = addr->buf;

        if (t_connect(fd, sndcall, NULL) == -1)
                res = FALSE;
        else
                res = TRUE;

        sndcall->addr.maxlen = sndcall->addr.len = 0;
```

```
619         sndcall->addr.buf = NULL;
620         (void) t_free((char *)sndcall, T_CALL);
621         free(addr->buf);
622         free(addr);
623         (void) t_close(fd);

625         return (res);
626 }


629 /*
630  * An internal function which optimizes rpcb_getaddr function.  It returns
631  * the universal address of the remote service or NULL.  It also optionally
689  * An internal function which optimizes rpcb_getaddr function.  It also
632  * returns the client handle that it uses to contact the remote rpcbind.
633  * The caller will re-purpose the client handle to contact the remote service.
634  *
635  * The algorithm used: First try version 4.  Then try version 3 (svr4).
636  * Finally, if the transport is TCP or UDP, try version 2 (portmap).
637  * 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.
638  * With this algorithm, we get performance as well as a plan for
639  * obsoleting version 2.
640  *
699  * For all other transports, the algorithm remains as 4 and then 3.
700  *
641  * XXX: Due to some problems with t_connect(), we do not reuse the same client
642  * handle for COTS cases and hence in these cases we do not return the
643  * client handle.  This code will change if t_connect() ever
644  * starts working properly.  Also look under clnt_vc.c.
645  */
646 struct netbuf *
647 __rpcb_findaddr_timed(rpcprog_t program, rpcvers_t version,
648         struct netconfig *nconf, char *host, CLIENT **clpp, struct timeval *tp)
649 {
650         static bool_t check_rpcbind = TRUE;
651         CLIENT *client = NULL;
652         RPCB parms;
653         enum clnt_stat clnt_st;
654         char *ua = NULL;
655         uint_t vers;
656         struct netbuf *address = NULL;
657         void *handle;
658         rpcb_entry_list_ptr relp = NULL;
659         bool_t tmp_client = FALSE;
717         uint_t start_vers = RPCBVERS4;

661         /* parameter checking */
662         if (nconf == NULL) {
663                 rpc_createerr.cf_stat = RPC_UNKNOWNPROTO;
664                 /*
665                  * Setting rpc_createerr.cf_stat is sufficient.
666                  * No details in rpc_createerr.cf_error needed.
667                  */
668                 return (NULL);
669         }

671         parms.r_addr = NULL;

673         /*
674          * Use default total timeout if no timeout is specified.
675          */
676         if (tp == NULL)
```



```
677                 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;

679         /*
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         /*
680          * Check if rpcbind is up.  This prevents needless delays when
681          * accessing applications such as the keyserver while booting
682          * disklessly.
683          */
684         if (check_rpcbind && strcmp(nconf->nc_protofmly, NC_LOOPBACK) == 0) {
685                 if (!__rpcbind_is_up()) {
686                         rpc_createerr.cf_stat = RPC_PMAPFAILURE;
687                         rpc_createerr.cf_error.re_errno = 0;
688                         rpc_createerr.cf_error.re_terrno = 0;
689                         goto error;
690                 }
691                 check_rpcbind = FALSE;
692         }

694         /*
695          * 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
696          */
697         parms.r_prog = program;
698         parms.r_vers = version;
699         parms.r_owner = (char *)&nullstring[0]; /* not needed; */
```



```
700         /* just for xdring */
701         parms.r_netid = nconf->nc_netid; /* not really needed */

703         /*
704          * If a COTS transport is being used, try getting address via CLTS
705          * transport.  This works only with version 4.
706          */
707         if (nconf->nc_semantics == NC_TPI_COTS_ORD ||
708             nconf->nc_semantics == NC_TPI_COTS) {
709                 tmp_client = TRUE;
710                 if ((handle = __rpc_setconf("datagram_v")) != NULL) {
871                 void *handle;
711                         struct netconfig *nconf_clts;
873                 rpcb_entry_list_ptr relp = NULL;

713                         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) {
714                                 if (strcmp(nconf_clts->nc_protofmly,
715                                     nconf->nc_protofmly) != 0) {
716                                         continue;
717                                 }
718                                 /*
719                                  * Sets rpc_createerr.cf_error members
720                                  * on failure
721                                  */
722                                 client = _getclnthandle_timed(host, nconf_clts,
723                                     &parms.r_addr, tp);
884                                         client = _getclnthandle_timed(host,
885                                                 nconf_clts, &parms.r_addr,
886                                                 tp);
724                                 break;
725                         }
726                         __rpc_endconf(handle);
727                 }
891                         if (client == NULL)
892                                 goto regular_rpcbind;   /* Go the regular way */
728         } else {
729                 /* Sets rpc_createerr.cf_error members on failure */
730                 client = _getclnthandle_timed(host, nconf, &parms.r_addr, tp);
731         }

733         if (client != NULL) {

735                 /* Set rpcbind version 4 */
894                         /* This is a UDP PORTMAP handle.  Change to version 4 */
736                 vers = RPCBVERS4;
737                 CLNT_CONTROL(client, CLSET_VERS, (char *)&vers);

897                 }
739                 /*
740                  * We also send the remote system the address we used to
741                  * contact it in case it can help it connect back with us
742                  */
743                 if (parms.r_addr == NULL) {
744                         parms.r_addr = strdup(""); /* for XDRing */
745                         if (parms.r_addr == NULL) {
746                                 syslog(LOG_ERR, "__rpcb_findaddr_timed: "
747                                     "strdup failed.");
748                                 /* Construct a system error */
749                                 rpc_createerr.cf_error.re_errno = errno;
750                                 rpc_createerr.cf_error.re_terrno = 0;
751                                 rpc_createerr.cf_stat = RPC_SYSTEMERROR;
```

```
908                             address = NULL;
752                             goto error;
753                     }
754             }

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

759             /* Sets error structure members in client handle */
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);

764             switch (clnt_st) {
765             case RPC_SUCCESS: /* Call succeeded */
766                     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)) {
767                     xdr_free((xdrproc_t)xdr_rpcb_entry_list_ptr,
768                         (char *)&relp);
769                     if (address != NULL) {
770                             /* Program number and version number matched */
771                             goto done;
772                     }
773                     /* 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);
774                     /*
775                      * XXX: should have returned with RPC_PROGUNAVAIL
776                      * or perhaps RPC_PROGNOTREGISTERED error but
929                      * XXX: should have perhaps returned with error but
777                      * since the remote machine might not always be able
778                      * to send the address on all transports, we try the
779                      * regular way with version 3, then 2
932                      * regular way with regular_rpcbind
780                      */
781                     /* Try the next version */
782                     break;
783             case RPC_PROGVERSMISMATCH: /* RPC protocol mismatch */
784                     clnt_geterr(client, &rpc_createerr.cf_error);
785                     if (rpc_createerr.cf_error.re_vers.low > vers) {
786                             rpc_createerr.cf_stat = clnt_st;
787                             goto error;  /* a new version, can't handle */
788                     }
789                     /* Try the next version */
790                     break;
791             case RPC_PROCUNAVAIL: /* Procedure unavailable */
792             case RPC_PROGUNAVAIL: /* Program not available */
793             case RPC_TIMEDOUT: /* Call timed out */
794                     /* Try the next version */
795                     break;
796             default:
797                     clnt_geterr(client, &rpc_createerr.cf_error);
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 {
798                     rpc_createerr.cf_stat = RPC_PMAPFAILURE;
941                     clnt_geterr(client, &rpc_createerr.cf_error);
799                     goto error;
```



```
800                             break;
801                     }
944             }

803             } else {
946     regular_rpcbind:

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

808             } /* End of version 4 */

810             /* Destroy a temporary client */
811             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 */
812                     CLNT_DESTROY(client);
813                     client = NULL;
814                     free(parms.r_addr);
815                     parms.r_addr = NULL;
816             }
817             tmp_client = FALSE;

819             /*
820              * Try version 3
821              */

823             /* Now the same transport is to be used to get the address */
824             if (client == NULL) {
825                     /* Sets rpc_createerr.cf_error members on failure */
826                     client = _getclnthandle_timed(host, nconf, &parms.r_addr, tp);
827             }
960                     if (client == NULL) {
828             address = NULL;
829             if (client != NULL) {
962                             goto error;
963                     }
964             }
830                     if (parms.r_addr == NULL) {
831                             parms.r_addr = strdup("");      /* for XDRing */
832                             if (parms.r_addr == NULL) {
833                                     syslog(LOG_ERR, "__rpcb_findaddr_timed: "
834                                         "strdup failed.");
835                                     /* Construct a system error */
836                                     rpc_createerr.cf_error.re_errno = errno;
837                                     rpc_createerr.cf_error.re_terrno = 0;
970                             address = NULL;
838                                     rpc_createerr.cf_stat = RPC_SYSTEMERROR;
839                                     goto error;
840                             }
841                     }

843                     CLNT_CONTROL(client, CLSET_RETRY_TIMEOUT,
844                         (char *)&rpcbrmttime);
845                     vers = RPCBVERS; /* Set the version */
846                     CLNT_CONTROL(client, CLSET_VERS, (char *)&vers);
976             /* First try from start_vers and then version 3 (RPCBVERS) */

848                     /* Sets error structure members in client handle */
978             CLNT_CONTROL(client, CLSET_RETRY_TIMEOUT, (char *)&rpcbrmttime);
979             for (vers = start_vers;  vers >= RPCBVERS; vers--) {
980                     /* Set the version */
981                     CLNT_CONTROL(client, CLSET_VERS, (char *)&vers);
849                     clnt_st = CLNT_CALL(client, RPCBPROC_GETADDR,
```

```
                    (xdrproc_t)xdr_rpcb, (char *)&parms,
                    (xdrproc_t)xdr_wrapstring, (char *)&ua, *tp);
                                (xdrproc_t)xdr_wrapstring,
                                (char *)&ua, *tp);
            if (clnt_st == RPC_SUCCESS) {
                    if ((ua == NULL) || (ua[0] == NULL)) {
                            if (ua != NULL)
                                    xdr_free(xdr_wrapstring, (char *)&ua);

            switch (clnt_st) {
            case RPC_SUCCESS: /* Call succeeded */
                    if (ua != NULL) {
                            if (ua[0] != '\0') {
                                    address = uaddr2taddr(nconf, ua);
                            }
                            xdr_free((xdrproc_t)xdr_wrapstring,
                                (char *)&ua);

                            if (address != NULL) {
                                    goto done;
                            }
                            /* NULL universal address */
                            /* But client call was successful */
                            clnt_geterr(client, &rpc_createerr.cf_error);
                            /* address unknown */
                            rpc_createerr.cf_stat = RPC_PROGNOTREGISTERED;
                            goto error;
                    }
#ifndef PORTMAP
                    clnt_geterr(client, &rpc_createerr.cf_error);
                    rpc_createerr.cf_stat = RPC_UNKNOWNPROTO;
                    goto error;
                    address = uaddr2taddr(nconf, ua);
#ifdef ND_DEBUG
                    fprintf(stderr, "\tRemote address is [%s]\n", ua);
                    if (!address)
                            fprintf(stderr,
                                "\tCouldn't resolve remote address!\n");
#endif
                    /* Try the next version */
                    break;
            case RPC_PROGVERSMISMATCH: /* RPC protocol mismatch */
                    clnt_geterr(client, &rpc_createerr.cf_error);
#ifdef PORTMAP
                    if (rpc_createerr.cf_error.re_vers.low > vers) {
                            rpc_createerr.cf_stat = clnt_st;
                            goto error;  /* a new version, can't handle */
                    }
#else
                    rpc_createerr.cf_stat = clnt_st;
                    goto error;
#endif
                    /* Try the next version */
                    break;
#ifdef PORTMAP
            case RPC_PROCUNAVAIL: /* Procedure unavailable */
            case RPC_PROGUNAVAIL: /* Program not available */
            case RPC_TIMEDOUT: /* Call timed out */
                    /* Try the next version */
                    break;
#endif
            default:
                    clnt_geterr(client, &rpc_createerr.cf_error);
                    rpc_createerr.cf_stat = RPC_PMAPFAILURE;
                    goto error;
                    break;
```



```
                }
        } /* End of version 3 */
#ifndef PORTMAP
        /* cf_error members set by creation failure */
        rpc_createerr.cf_stat = RPC_PROGNOTREGISTERED;
#endif
        /*
         * Try version 2
         */
                        xdr_free((xdrproc_t)xdr_wrapstring, (char *)&ua);

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

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

                        if (client != NULL) {
                                CLNT_DESTROY(client);
                                client = NULL;
                                free(parms.r_addr);
                                parms.r_addr = NULL;
                        }
                        if ((handle = __rpc_setconf("udp")) == NULL) {
                                /* Construct an unknown protocol error */
                                rpc_createerr.cf_stat = RPC_UNKNOWNPROTO;
                        if (!address) {
                                /* We don't know about your universal address */
                                rpc_createerr.cf_stat = RPC_N2AXLATEFAILURE;
                                goto error;
                        }
                        goto done;
                }
                if (clnt_st == RPC_PROGVERSMISMATCH) {
                        struct rpc_err rpcerr;

                        /*
                         * The following to reinforce that you can
                         * only request for remote address through
                         * the same transport you are requesting.
                         * ie. requesting unversial address
                         * of IPv4 has to be carried through IPv4.
                         * Can't use IPv6 to send out the request.
                         * The mergeaddr in rpcbind can't handle
                         * this.
                         */
                        for (;;) {
                                if ((newnconf = __rpc_getconf(handle))
                                    == NULL) {
                                        __rpc_endconf(handle);
                                        /*
                                         * Construct an unknown protocol
                                         * error
                                         */
                                        rpc_createerr.cf_stat =
                                            RPC_UNKNOWNPROTO;
                        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 */
 960                                goto error;
 961                        }
 962                        /*
 963                         * here check the protocol family to
 964                         * be consistent with the request one
 965                         */
 966                        if (strcmp(newnconf->nc_protofmly,
 967                            nconf->nc_protofmly) == 0)
 968                                break;
 969                }

 971                        /* Sets rpc_createerr.cf_error members on failure */
 972                        client = _getclnthandle_timed(host, newnconf,
 973                            &parms.r_addr, tp);
 974                        __rpc_endconf(handle);
 975                        tmp_client = TRUE;
 976                }
 977                if (client == NULL) {
 978                        /*
 979                         * rpc_createerr. cf_error members were set by
 980                         * creation failure
 981                         */
1023        if ((address == NULL) || (address->len == 0)) {
 982                        rpc_createerr.cf_stat = RPC_PROGNOTREGISTERED;
 983                        tmp_client = FALSE;
 984                        goto error;
 985                }

 987                /*
 988                 * Set version and retry timeout.
 989                 */
 990                CLNT_CONTROL(client, CLSET_RETRY_TIMEOUT, (char *)&rpcbrmttime);
 991                CLNT_CONTROL(client, CLSET_VERS, (char *)&pmapvers);

 993                pmapparms.pm_prog = program;
 994                pmapparms.pm_vers = version;
 995                pmapparms.pm_prot = (strcmp(nconf->nc_proto, NC_TCP) != 0) ?
 996                    IPPROTO_UDP : IPPROTO_TCP;
 997                pmapparms.pm_port = 0;  /* not needed */

 999                /* Sets error structure members in client handle */
1000                clnt_st = CLNT_CALL(client, PMAPPROC_GETPORT,
1001                    (xdrproc_t)xdr_pmap, (caddr_t)&pmapparms,
1002                    (xdrproc_t)xdr_u_short, (caddr_t)&port, *tp);

1004                if (clnt_st != RPC_SUCCESS) {
1005                        clnt_geterr(client, &rpc_createerr.cf_error);
1006                        rpc_createerr.cf_stat = RPC_RPCBFAILURE;
1007                        goto error;
1008                } else if (port == 0) {
1009                        /* Will be NULL universal address */
1010                        /* But client call was successful */
1011                        clnt_geterr(client, &rpc_createerr.cf_error);
1012                        rpc_createerr.cf_stat = RPC_PROGNOTREGISTERED;
1013                        goto error;
1014                }
1015                port = htons(port);
1016                CLNT_CONTROL(client, CLGET_SVC_ADDR, (char *)&remote);
1017                if (((address = malloc(sizeof (struct netbuf))) == NULL) ||
1018                    ((address->buf = malloc(remote.len)) == NULL)) {
1019                        /* Construct a system error */
1020                        rpc_createerr.cf_error.re_errno = errno;
```



```
1021                        rpc_createerr.cf_error.re_terrno = 0;
1022                        rpc_createerr.cf_stat = RPC_SYSTEMERROR;
1023                        free(address);
1024                        address = NULL;
1025                        goto error;
1026                }
1027                (void) memcpy(address->buf, remote.buf, remote.len);
1028                (void) memcpy(&address->buf[sizeof (short)], &port,
1029                    sizeof (short));
1030                address->len = address->maxlen = remote.len;
1031                goto done;
1032        } else {
1033                /*
1034                 * This is not NC_INET.
1035                 * Always an error for version 2.
1036                 */
1037                if (client != NULL && clnt_st != RPC_SUCCESS) {
1038                        /* There is a client that failed */
1039                        clnt_geterr(client, &rpc_createerr.cf_error);
1040                        rpc_createerr.cf_stat = clnt_st;
1041                } else {
1042                        /* Something else */
1043                        rpc_createerr.cf_stat = RPC_UNKNOWNPROTO;
1044                        /*
1045                         * Setting rpc_createerr.cf_stat is sufficient.
1046                         * No details in rpc_createerr.cf_error needed.
1047                         */
1048                }
1049        }
1050 #endif

1052 error:
1053        /* Return NULL address and NULL client */
1054        address = NULL;
1055        if (client != NULL) {
1029        if (client) {
1056                CLNT_DESTROY(client);
1057                client = NULL;
1058        }

1060 done:
1061        /* Return an address and optional client */
1062        if (client != NULL && tmp_client) {
1063                /* This client is the temporary one */
1034        if (nconf->nc_semantics != NC_TPI_CLTS) {
1035                /* This client is the connectionless one */
1036                if (client) {
1064                CLNT_DESTROY(client);
1065                client = NULL;
1066        }
1067        if (clpp != NULL) {
1040        }
1041        if (clpp) {
1068                *clpp = client;
1069        } else if (client != NULL) {
1043        } else if (client) {
1070                CLNT_DESTROY(client);
1071        }
1046        if (parms.r_addr)
1072        free(parms.r_addr);
1073        return (address);
1074 }
_____unchanged_portion_omitted_
```