Transport Selection and Name-to-Address Mapping 4

This chapter covers selecting transports and resolving network addresses. It describes interfaces that enable you to specify the available communication protocols for an application. The chapter also describes additional functions that provide direct mapping of names to network addresses.

How Transport Selection Works	*page 112*
Name-to-Address Mapping	*page 120*
Using the Name-to-Address Mapping Routines	*page 122*

Note – In this chapter the terms *network* and *transport* are used interchangeably to refer to the programmatic interface that conforms to the transport layer of the OSI Reference Mode. The term network is also used to refer to the physical collection of computers connected through some electronic medium.

Transport Selection Is Multithread Safe

The interface described in this chapter is multithread safe. This means that applications that contain transport selection function calls can be used freely in a multithreaded application.

4

Transport Selection

A distributed application must use a standard interface to the transport services if it is to be portable to different protocols. Transport selection services provide an interface that allows an application to select which protocols to use. This makes an application "protocol" and "medium independent."

Transport selection makes it easy for a client application to try each available transport until it establishes communication with a server. Transport selection lets server applications accept requests on multiple transports, and in doing so, communicate over a number of protocols. Transports may be tried in either the order specified by the local default sequence or in an order specified by the user.

Choosing from the available transports is the responsibility of the application. The transport selection mechanism makes that selection uniform and simple.

How Transport Selection Works

The transport selection component is built around:

- A network configuration database (the `/etc/netconfig` file), which contains an entry for each network on the system
- Optional use of the `NETPATH` environment variable

The `NETPATH` variable is set by the user; it contains an ordered list of transport identifiers. The transport identifiers match the `netconfig` network ID field and are links to records in the `netconfig` file. The `netconfig` file is described in "/etc/netconfig File" on page 113. The network selection interface is a set of access routines for the network-configuration database.

One set of library routines accesses only the `/etc/netconfig` entries identified by the `NETPATH` environment variable:

```
setnetpath()
getnetpath()
endnetpath()
```

They are described in "NETPATH Access to netconfig Data" on page 116 and in `getnetpath(3N)`. These routines let the user influence the selection of transports used by the application.

To avoid user influence on transport selection, use the routines that access the `netconfig` database directly. These routines are described in "Accessing netconfig" on page 118 and in `getnetconfig(3N)`.

```
setnetconfig()
getnetconfig()
endnetconfig()
```

The following two routines manipulate `netconfig` entries and the data structures they represent.

```
getnetconfigent()
freenetconfigent()
```

`/etc/netconfig` *File*

The `netconfig` file describes all transport protocols on a host. The entries in the `netconfig` file are explained briefly in Table 4-1 and in more detail in `netconfig(4)`.

Table 4-1 The `netconfig` File

Entries	Description
`network ID`	A local representation of a transport name (such as `tcp`). Do not assume that this field contains a well-known name (such as `tcp` or `udp`) or that two systems use the same name for the same transport.
`semantics`	The semantics of the particular transport protocol. Valid semantics are: `tpi_clts` – connectionless `tpi_cots` – connection oriented `tpi_cots_ord` – connection oriented with orderly release
`flags`	May take only the values, `v`, or hyphen `(-)`. Only the visible flag (`-v`) is defined.
`protocol family`	The protocol family name of the transport provider (for example, `inet` or `loopback`).

Table 4-1 The `netconfig` File

Entries	Description
`protocol name`	The protocol name of the transport provider. For example, if *protocol family* is `inet`, then *protocol name* is `tcp`, `udp`, or `icmp`. Otherwise, the value of *protocol name* is a hyphen (`-`).
`network device`	The full path name of the device file to open when accessing the transport provider.
`name-to-address translation libraries`	Names of the shared objects. This field contains the comma-separated file names of the shared objects that contain name-to-address mapping routines. Shared objects are located through the path in the `LD_LIBRARY_PATH` variable. A "`-`" in this field indicates the absence of any translation libraries.

Code Example 4-1 shows a sample `netconfig` file. Use of the `netconfig` file has been changed for the `inet` transports, as described in the commented section in the sample file. This change is also described in "Name-to-Address Mapping" on page 120.

Code Example 4-1 Sample `netconfig` File

```
# The "Network Configuration" File.
#
# Each entry is of the form:
#
#<net <semantics>  <flags> <proto    <proto   <device>    <nametoaddr_libs>
# id>                       family>   name>
#
# The "-" in <nametoaddr_libs> for inet family transports indicates redirection
# to the name service switch policies for "hosts" and "services. The "-" may be
# replaced by nametoaddr libraries that comply with the SVR4 specs, in which
# case the name service switch will be used for netdir_getbyname, netdir_
# getbyaddr, gethostbyname, gethostbyaddr, getservbyname, and getservbyport.
# There are no nametoaddr_libs for the inet family in Solaris anymore.
#
udp     tpi_clts      v       inet      udp      /dev/udp      -
#
tcp     tpi_cots_ord  v       inet      tcp      /dev/tcp      -
#
icmp    tpi_raw       -       inet      icmp     /dev/icmp     -
#
rawip   tpi_raw       -       inet      -        /dev/rawip    -
```

```
#
ticlts tpi_clts      v      loopback  -        /dev/ticlts    straddr.so
#
ticots tpi_cots      v      loopback  -        /dev/ticots    straddr.so
#
ticotsord tpi_cots_ord v    loopback  -        /dev/ticotsord straddr.so
#
```

Network selection library routines return pointers to `netconfig` entries. The `netconfig` structure is shown in Code Example 4-2.

Code Example 4-2 The `netconfig` Structure

```
struct netconfig {
    char          *nc_netid;    /* network identifier */
    unsigned long nc_semantics; /* semantics of protocol */
    unsigned long nc_flag;      /* flags for the protocol */
    unsigned long nc_protofmly; /* family name */
    unsigned long nc_proto;     /* proto specific */
    char          *nc_device;   /* device name for network id */
    unsigned long nc_nlookups;  /* # entries in nc_lookups */
    char          **nc_lookups; /* list of lookup libraries */
    unsigned long nc_unused[8];
};
```

Valid network IDs are defined by the system administrator, who must ensure that network IDs are locally unique. If they are not, some network selection routines can fail. For example, it is not possible to know which network `getnetconfigent("udp")` will use if there are two `netconfig` entries with the network ID `udp`.

The system administrator also sets the order of the entries in the `netconfig` database. The routines that find entries in `/etc/netconfig` return them in order, from the beginning of the file. The order of transports in the `netconfig` file is the default transport search sequence of the routines. Loopback entries should be at the end of the file.

The `netconfig` file and the `netconfig` structure are described in greater detail in `netconfig(4)`.

4

The NETPATH *Environment Variable*

An application usually uses the default transport search path set by the system administrator to locate an available transport. However, when a user wants to influence the choices made by an application, the application can modify the interface by using the environment variable NETPATH and the routines described in the section, "NETPATH Access to netconfig Data." These routines access only the transports specified in the NETPATH variable.

NETPATH is similar to the PATH variable. It is a colon-separated list of transport IDs. Each transport ID in the NETPATH variable corresponds to the network ID field of a record in the netconfig file. NETPATH is described in environ(5).

The default transport set is different for the routines that access netconfig through the NETPATH environment variable (described in the next section) and the routines that access netconfig directly. The default transport set for routines that access netconfig via NETPATH consists of the visible transports in the netconfig file. For routines that access netconfig directly, the default transport set is the entire netconfig file. A transport is visible if the system administrator has included a v flag in the flags field of that transport's netconfig entry.

NETPATH *Access to* netconfig *Data*

Three routines access the network configuration database indirectly through the NETPATH environment variable. The variable specifies the transport or transports an application is to use and the order to try them. NETPATH components are read from left to right. The functions have the following interfaces:

```
#include <netconfig.h>

void *setnetpath(void);
struct netconfig *getnetpath(void *);
int endnetpath(void *);
```

A call to setnetpath() initializes the search of NETPATH. It returns a pointer to a database that contains the entries specified in a NETPATH variable. The pointer, called a handle, is used to traverse this database with getnetpath(). setnetpath() must be called before the first call to getnetpath().

When first called, `getnetpath()` returns a pointer to the `netconfig` file entry that corresponds to the first component of the `NETPATH` variable. On each subsequent call, `getnetpath()` returns a pointer to the `netconfig` entry that corresponds to the next component of the `NETPATH` variable; `getnetpath()` returns `NULL` if there are no more components in `NETPATH`. A call to `getnetpath()` without an initial call to `setnetpath()` causes an error; `getnetpath()` requires the pointer returned by `setnetpath()` as an argument.

`getnetpath()` silently ignores invalid `NETPATH` components. A `NETPATH` component is invalid if there is no corresponding entry in the `netconfig` database.

If the `NETPATH` variable is unset, `getnetpath()` behaves as if `NETPATH` were set to the sequence of default or visible transports in the `netconfig` database, in the order in which they are listed.

`endnetpath()` is called to release the database pointer to elements in the `NETPATH` variable when processing is complete. `endnetpath()` fails if `setnetpath()` was not called previously. Code Example 4-3 shows the `setnetpath()`, `getnetpath()`, and `endnetpath()` routines.

Code Example 4-3 `setnetpath()`, `getnetpath()`, and `endnetpath()`

```
#include <netconfig.h>

void *handlep;
struct netconfig *nconf;

if ((handlep = setnetpath()) == (void *)NULL) {
    nc_perror(argv[0]);
    exit(1);
}
while ((nconf = getnetpath(handlep)) != (struct netconfig *)NULL) {
    /*
     * nconf now describes a transport provider.
     */
}
endnetpath(handlep);
```

The `netconfig` structures obtained through `getnetpath()` become invalid after the execution of `endnetpath()`. To preserve the data in the structure, use `getnetconfigent(nconf->nc_netid)` to copy them into a new data structure.

4

Accessing `netconfig`

Three functions access `/etc/netconfig` and locate `netconfig` entries. The routines `setnetconfig()`, `getnetconfig()`, and `endnetconfig()` have the following interfaces:

```
#include <netconfig.h>

void *setnetconfig(void);
struct netconfig *getnetconfig(void *);
int endnetconfig(void *);
```

A call to `setnetconfig()` initializes the record pointer to the first index in the database; `setnetconfig()` must be used before the first use of `getnetconfig()`. `setnetconfig()` returns a unique handle (a pointer into the database) to be used by the `getnetconfig()` routine. Each call to `getnetconfig()` returns the pointer to the current record in the `netconfig` database and increments its pointer to the next record. It can be used to search the entire `netconfig` database. `getnetconfig()` returns a `NULL` at the end of file.

You must use `endnetconfig()` to release the database pointer when processing is complete. `endnetconfig()` must not be called before `setnetconfig()`.

Code Example 4-4 `setnetconfig()`, `getnetconfig()`, and `endnetconfig()`

```
void *handlep;
struct netconfig *nconf;

if ((handlep = setnetconfig()) == (void *)NULL){
    nc_perror(argv[0]);
    exit(1);
}
/*
 * transport provider information is described in nconf.
 * process_transport is a user-supplied routine that
 * tries to connect to a server over transport nconf.
 */
while ((nconf = getnetconfig(handlep)) != (struct netconfig *)NULL){
    if (process_transport(nconf) == SUCCESS){
        break;
    }
}
endnetconfig(handlep);
```

The last two functions have the following interface:

```
#include <netconfig.h>
struct netconfig *getnetconfigent(char *);
int freenetconfigent(struct netconfig *);
```

`getnetconfigent()` returns a pointer to the `struct netconfig` structure corresponding to `netid`. It returns `NULL` if `netid` is invalid. `setnetconfig()` need not be called before `getnetconfigent()`.

`freenetconfigent()` frees the structure returned by `getnetconfigent()`. Code Example 4-5 shows the `getnetconfigent()` and `freenetconfigent()` routines.

Code Example 4-5 `getnetconfigent()` and `freenetconfigent()`

```
/* assume udp is a netid on this host */
struct netconfig *nconf;

if ((nconf = getnetconfigent("udp")) == (struct netconfig *)NULL){
    nc_perror("no information about udp");
    exit(1);
}
process_transport(nconf);
freenetconfigent(nconf);
```

Loop Through All Visible `netconfig` *Entries*

The `setnetconfig()` call is used to step through all the transports marked *visible* (by a `v` flag in the `flags` field) in the `netconfig` database. The transport selection routine returns a `netconfig` pointer.

Code Example 4-6 Looping Through Visible Transports

```
void *handlep;
struct netconfig *nconf;

if ((handlep = setnetconfig() == (void *) NULL) {
    nc_perror("setnetconfig");
    exit(1);
}
while (nconf = getnetconfig(handlep))
    if (nconf->nc_flag & NC_VISIBLE)
        doit(nconf);
(void) endnetconfig(handlep);
```

Looping Through User-Defined `netconfig` *Entries*

Users can control the loop by setting the `NETPATH` environment variable to a colon-separated list of transport names. If `NETPATH` is set as follows:

```
NETPATH=tcp:udp
```

The loop first returns the `tcp` entry, and then the `udp` entry. If `NETPATH` is not defined, the loop returns all visible entries in the `netconfig` file in the order in which they are stored. The `NETPATH` environment variable lets users define the order in which client-side applications try to connect to a service. It also lets the server administrator limit transports on which a service can listen.

Use `getnetpath()` and `setnetpath()` to obtain or modify the network path variable. Code Example 4-6 shows the form and use, which are similar to the `getnetconfig()` and `setnetconfig()` routines.

Name-to-Address Mapping

Name-to-address mapping lets an application obtain the address of a service on a specified host, independent of the transport used. Name-to-address mapping consists of the following functions:

```
netdir_getbyname()
netdir_getbyaddr()
netdir_free()
taddr2uaddr()
uaddr2taddr()
netdir_options()
```

The first argument of each routine points to a netconfig structure that describes a transport. The routine uses the array of directory-lookup library paths in the netconfig structure to call each path until the translation succeeds.

The libraries are described in Table 4-2 on page 121. The routines described in the section, “Using the Name-to-Address Mapping Routines,” are defined in `netdir(3N)`.

Note – The following libraries no longer exist in the Solaris 2.x environment: `tcpip.so`, `switch.so`, and `nis.so`. For more information on this change, see `nsswitch.conf(4)` and the NOTES section of `gethostbyname(3N)`.

Table 4-2 Name-to-Address Libraries

Library	Transport Family	Description
–	`inet`	For networks of the protocol family `inet`, its name-to-address mapping is provided by the name service switch based on the entries for *hosts* and *services* in the file `nsswitch.conf`. For networks of other families, the "-" indicates a non-functional name-to-address mapping.
`straddr.so`	`loopback`	Contains the name-to-address mapping routines of any protocol that accepts strings as addresses, such as the loopback transports.

`straddr.so` *Library*

Files for the library are created and maintained by the system administrator. The `straddr.so` files are `/etc/net/`*transport-name*`/hosts` and `/etc/net/`*transport-name*`/services`. *transport-name* is the local name of the transport that accepts string addresses (specified in the *network ID* field of the `/etc/netconfig` file). For example, the host file for `ticlts` would be `/etc/net/ticlts/hosts`, and the service file for `ticlts` would be `/etc/net/ticlts/services`.

Even though most string addresses do not distinguish between *host* and *service*, separating the string into a host part and a service part is consistent with other transports. The `/etc/net/`*transport-name*`/hosts` file contains a text string that is assumed to be the host address, followed by the host name. For example:

```
joyluckaddr         joyluck
carpediemaddr       carpediem
thehopaddr          thehop
pongoaddr           pongo
```

For loopback transports, it makes no sense to list other hosts because the service cannot go outside the containing host.

The /etc/net/transport-name/services file contains service names followed by strings identifying the service address. For example:

```
rpcbind         rpc
listen          serve
```

The routines create the full string address by concatenating the host address, a period (.), and the service address. For example, the address of the listen service on pongo is pongoaddr.serve.

When an application requests the address of a service on a particular host on a transport that uses this library, the host name must be in /etc/net/*transport*/hosts, and the service name must be in /etc/net/*transport*/services. If either is missing, the name-to-address translation fails.

Using the Name-to-Address Mapping Routines

This section provides an overview of what routines are available to use. The routines return or convert the network names to their respective network addresses. Note that netdir_getbyname(), netdir_getbyaddr(), and taddr2uaddr() return pointers to data that must be freed by calls to netdir_free().

```
int netdir_getbyname(struct netconfig *nconf,
                     struct nd_hostserv *service,
                     struct nd_addrlist **addrs);
```

netdir_getbyname() maps the host and service name specified in *service* to a set of addresses consistent with the transport identified in *nconf*. The nd_hostserv and nd_addrlist structures are defined in netdir(3N). A pointer to the addresses is returned in *addrs*.

To find all addresses of a host and service (on all available transports), call netdir_getbyname() with each netconfig structure returned by either getnetpath() or getnetconfig().

```
int netdir_getbyaddr(struct netconfig *nconf,
                     struct nd_hostservlist **service,
                     struct netbuf *netaddr);
```

netdir_getbyaddr() maps addresses into host and service names. The function is called with an address in *netaddr* and returns a list of host-name and service-name pairs in *service*. The nd_hostservlist structure is defined in netdir(3N).

```
void netdir_free(void *ptr, int struct_type);
```

The `netdir_free()` routine frees structures allocated by the name-to-address translation routines. The parameters can take the values shown in Table 4-3.

Table 4-3 `netdir_free()` Routines

struct_type	***ptr***
`ND_HOSTSERV`	pointer to an `nd_hostserv` structure
`ND_HOSTSERVLIST`	pointer to an `nd_hostservlist` structure
`ND_ADDR`	pointer to a `netbuf` structure
`ND_ADDRLIST`	pointer to an `nd_addrlist` structure

```
char *taddr2uaddr(struct netconfig *nconf, struct netbuf *addr);
```

`taddr2uaddr()` translates the address pointed to by *addr a*nd returns a transport-independent character representation of the address (“universal address”). *nconf s*pecifies the transport for which the address is valid. The universal address can be freed by `free()`.

```
struct netbuf *uaddr2taddr(struct netconfig *nconf, char *uaddr);
```

The “universal address” pointed to by *uaddr* is translated into a `netbuf` structure; *nconf* specifies the transport for which the address is valid.

```
int netdir_options(struct netconfig *nconf, int option, int fd,
                   char *point_to_args);
```

`netdir_options()` interfaces to transport-specific capabilities (such as the broadcast address and reserved port facilities of TCP and UDP). *nconf* specifies a transport. *option* specifies the transport-specific action to take. *fd* may or may not be used depending upon the value of *option*. The fourth argument points to operation-specific data.

Table 4-4 shows the values used for *option*:

Table 4-4 Values for `netdir_options`

Option	Description
`ND_SET_BROADCAST`	Sets the transport for broadcast (if the transport supports broadcast).
`ND_SET_RESERVEDPORT`	Lets the application bind to a reserved port (if allowed by the transport).
`ND_CHECK_RESERVEDPORT`	Verifies that an address corresponds to a reserved port (if the transport supports reserved ports).
`ND_MERGEADDR`	Transforms a locally meaningful address into an address to which client hosts can connect,

```
void netdir_perror(char *s);
```

`netdir_perror()` displays the message stating why one of the name-to-address mapping routines failed on `stderr`.

```
char *netdir_sperror(void);
```

`netdir_sperror()` returns a string containing the error message stating why one of the name-to-address mapping routines failed.

Code Example 4-7 shows network selection and name-to-address mapping.

Code Example 4-7 Network Selection and Name-to-Address Mapping

```
#include <netconfig.h>
#include <netdir.h>
#include <sys/tiuser.h>

struct nd_hostserv nd_hostserv;   /* host and service information */
struct nd_addrlist *nd_addrlistp; /* addresses for the service */
struct netbuf *netbufp;           /* the address of the service */
struct netconfig *nconf;          /* transport information*/
int i;                            /* the number of addresses */
char *uaddr;                      /* service universal address */
void *handlep;                 /* a handle into network selection */
/*
 * Set the host structure to reference the "date"
 * service on host "gandalf"
 */
nd_hostserv.h_host = "gandalf";
```

```
nd_hostserv.h_serv = "date";
/*
 * Initialize the network selection mechanism.
 */
if ((handlep = setnetpath()) == (void *)NULL) {
    nc_perror(argv[0]);
    exit(1);
}
/*
 * Loop through the transport providers.
 */
while ((nconf = getnetpath(handlep)) != (struct netconfig *)NULL) {
    /*
     * Print out the information associated with the
     * transport provider described in the "netconfig"
     * structure.
     */
    printf("Transport provider name: %s\n", nconf->nc_netid);
    printf("Transport protocol family: %s\n", nconf->nc_protofmly);
    printf("The transport device file: %s\n", nconf->nc_device);
    printf("Transport provider semantics: ");
    switch (nconf->nc_semantics) {
        case NC_TPI_COTS:
            printf("virtual circuit\n");
            break;
        case NC_TPI_COTS_ORD:
            printf("virtual circuit with orderly release\n");
            break;

        case NC_TPI_CLTS:
            printf("datagram\n");
            break;
    }
    /*
     * Get the address for service "date" on the host
     * named "gandalf" over the transport provider
     * specified in the netconfig structure.
     */
    if (netdir_getbyname(nconf, &nd_hostserv, &nd_addrlistp)
            != ND_OK) {
        printf("Cannot determine address for service\n");
        netdir_perror(argv[0]);
        continue;
    }
    printf("<%d> address of date service on gandalf:\n",
            nd_addrlistp->n_cnt);
```

```
        /*
         * Print out all addresses for service "date" on
         * host "gandalf" on current transport provider.
         */
        netbufp = nd_addrlistp->n_addrs;
        for (i = 0; i < nd_addrlistp->n_cnt; i++, netbufp++) {
            uaddr = taddr2uaddr(nconf,netbufp);
            printf("%s\n",uaddr);
            free(uaddr);
        }
        netdir_free( nd_addrlistp, ND_ADDRLIST );
    }
    endnetconfig(handlep);
```

Portability From Previous Releases

The list that follows contains the names of the functions with functionality unchanged from earlier releases.

```
gethostbyname()
gethostbyaddr()
gethostent()
getrpcbyname()
getrpcbynumber()
getservbyname()
getservbyaddr()
netdir_free()
netdir_getbyname()
netdir_getbyaddr()
netdir_options()
netdir_perror()
netdir_sperror()
taddr2uaddr()
uaddr2taddr()
```

Other porting issues are described in *ONC+ Developer's Guide*.