```
/*                                                                      tab:8
 *
 * NetLink.h - header file for C++ version of network link module
 *
 * "Copyright (c) 2006-2011 by Steven S. Lumetta."
 *
 * Permission to use, copy, modify, and distribute this software and its
 * documentation for any purpose, without fee, and without written agreement is
 * hereby granted, provided that the above copyright notice and the following
 * two paragraphs appear in all copies of this software.
 *
 * IN NO EVENT SHALL THE AUTHOR OR THE UNIVERSITY OF ILLINOIS BE LIABLE TO
 * ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL
 * DAMAGES ARISING OUT  OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION,
 * EVEN IF THE AUTHOR AND/OR THE UNIVERSITY OF ILLINOIS HAS BEEN ADVISED
 * OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 * THE AUTHOR AND THE UNIVERSITY OF ILLINOIS SPECIFICALLY DISCLAIM ANY
 * WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
 * MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.  THE SOFTWARE
 * PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND NEITHER THE AUTHOR NOR
 * THE UNIVERSITY OF ILLINOIS HAS ANY OBLIGATION TO PROVIDE MAINTENANCE,
 * SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS."
 *
 * Author:          Steve Lumetta
 * Version:         2
 * Creation Date:   Wed Jan 28 09:20:15 2009
 * Filename:        NetLink.h
 * History:
 *      SL      1       Wed Jan 28 09:20:15 2009
 *              Adapted from C version.
 *      SL      2       Thu Jan 27 08:59:27 2011
 *              Changed connection constructor to use strings via getaddrinfo.
 */

#if !defined(_NETLINK_H)
#define _NETLINK_H

#include <exception>

#include <netinet/in.h>
#include <string.h>
#include <unistd.h>

namespace network_link_module {

/* Error codes are visible: do not reorder--only append new codes. */
typedef enum {
    UNINITIALIZED    = -1, /* module initialization not done yet */

    NO_ERROR         = 0, /* success */

    /* errors returned by initialize */
    INIT_FAILED      = 1, /* module initialization failed     */

    /* errors returned from reads and writes */
    WAS_CLOSED       = 2, /* remote machine closed connection */
    READ_WRITE_ERROR = 3, /* error occurred during read/write */

    NL_NUM_ERRORS
} err_t;

/*
 * Well-known TCP port numbers, added here for convenience.
 * See /etc/services for more.
 */
enum {
    ECHO        = 7,
    FTP_DATA    = 20,
    FTP_CONTROL = 21,
    SSH         = 22,
    TELNET      = 23,
    HTTP        = 80,
    SFTP        = 115,
    IMAP2       = 143,
    SNMP        = 161,
    LDAP        = 389,
    HTTPS       = 443,
    IMAPS       = 993
};


// exception hierarchy
class NetLinkException                       : public std::exception {};

  // module initialization failed
  class NetLinkInitializationFailedException : public NetLinkException {};

  // bad arguments passed to interface function
  class NetLinkBadArgumentsException         : public NetLinkException {};

  // inactive NetLink or NetLinkServer used
  class NetLinkInactiveException             : public NetLinkException {};


class NetLinkEndpoint {

    // data are the same for both servers and normal connections
    protected:
        int                fd;        // file descriptor (-1 means invalid)
        struct sockaddr_in addr;      // remote Internet address (local for
                                      //    server, with 0 IP address)


    // module initialization (primarily to avoid termination by SIGPIPE)
    private:
        static err_t initialize ();
        static err_t init_success;
        static void maybe_initialize () {
            if (UNINITIALIZED == init_success) {
                init_success = initialize ();
            }
        }


    // interface functions available for both servers and normal connections
    public:

        // check module initialization status
        static err_t init_status () {
            // If initialization were only tried in constructor, can't avoid
            // exceptions on failure, and this call would be pointless.
            maybe_initialize ();
            return init_success;
        }
```

```
        // returns TCP port number (local for a NetLinkServer,
        //     remote for a NetLink)
        int16_t tcp_port () {return ntohs (addr.sin_port);}

        // returns true if NetLinkEndpoint can be used
        bool is_active () {return (-1 != fd);}


    // internal function
    private:

        // mark endpoint as no longer usable
        void mark_inactive () {fd = -1;}


    // internal functions for use by derived classes
    protected:

        // shut down an endpoint (if not already done)
        void close_endpoint () {
            if (is_active ()) {
                close (fd);
                mark_inactive ();
            }
        }

        NetLinkEndpoint () {
            // check for module initialization
            maybe_initialize ();
            // throw exception if initialization failed
            if (INIT_FAILED == init_success) {
                throw NetLinkInitializationFailedException ();
            }
            // initialize descriptor and address data: file descriptor
            // defaults to invalid; address defaults to "any" (local)
            // Internet (IPv4) address with port 0 (invalid); flags
            // default to none
            mark_inactive ();
            addr.sin_family      = AF_INET;
            addr.sin_addr.s_addr = INADDR_ANY;
            addr.sin_port        = htons (0);
            memset (&addr.sin_zero, 0, sizeof (addr.sin_zero));
        }
        ~NetLinkEndpoint () {close_endpoint ();}
};


// NetLink class needs to know about NetLinkServer so as to be able
// to construct new connections by accepting connections from a server.
class NetLinkServer;


// a connection
class NetLink : public NetLinkEndpoint {

    // internal function used to make TCP connections within constructors
    private:
        void connectNetLink (uint32_t ip_addr, uint16_t tcp_port);

    // interface functions
    public:

        // Connect to a specific IP address and port number.
        NetLink (uint32_t ip_addr, uint16_t tcp_port);

        // Connect to a host name and port number.
        NetLink (const char* host_name, const char* tcp_port);

        // Accept (wait for) an incoming connection from a NetLinkServer.
        NetLink (NetLinkServer* server);

        // Get IP address of remote machine (in host byte order). */
        uint32_t remote_ip_address () {return ntohl (addr.sin_addr.s_addr);}

        // Read a fixed number of bytes (NetLink becomes inactive on failure).
        err_t blocking_read (void* buf, int32_t n_bytes);

        // Write a fixed number of bytes (NetLink becomes inactive on failure).
        err_t blocking_write (const void* buf, int32_t n_bytes);
};


// a server
class NetLinkServer : public NetLinkEndpoint {

    // interface functions
    public:

        // Create a server at a specified TCP port number.
        NetLinkServer (uint16_t tcp_port);

    // Constructor for accepting connections needs to access protected
    //    NetLinkEndpoint data.
    friend NetLink::NetLink (NetLinkServer* server);
};


} // network_link_module_namespace


/*
 * N.B. I have explicitly avoided exposing the file descriptor associated
 * with a NetLinkEndpoint.  The intended use of this module is for each
 * NetLink and NetLinkServer to be driven by a separate thread and thus to
 * obviate the need for select/poll across file descriptor sets.
 *
 * Non-blocking access is also mostly unnecessary in this model.  However,
 * note that two NetLinks sending large amounts of data to one another
 * may deadlock.  Either: disallow such behavior, use separate R/W threads
 * for one or both NetLinks, or extend the module to support non-blocking
 * access.
 */

#endif /* _NETLINK_H */
```

```
/*                                                                      tab:8
 *
 * NetLink.cc - source file for C++ version of network link module
 *
 * "Copyright (c) 2006-2011 by Steven S. Lumetta."
 *
 * Permission to use, copy, modify, and distribute this software and its
 * documentation for any purpose, without fee, and without written agreement is
 * hereby granted, provided that the above copyright notice and the following
 * two paragraphs appear in all copies of this software.
 *
 * IN NO EVENT SHALL THE AUTHOR OR THE UNIVERSITY OF ILLINOIS BE LIABLE TO
 * ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL
 * DAMAGES ARISING OUT  OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION,
 * EVEN IF THE AUTHOR AND/OR THE UNIVERSITY OF ILLINOIS HAS BEEN ADVISED
 * OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 * THE AUTHOR AND THE UNIVERSITY OF ILLINOIS SPECIFICALLY DISCLAIM ANY
 * WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
 * MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.  THE SOFTWARE
 * PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND NEITHER THE AUTHOR NOR
 * THE UNIVERSITY OF ILLINOIS HAS ANY OBLIGATION TO PROVIDE MAINTENANCE,
 * SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS."
 *
 * Author:          Steve Lumetta
 * Version:         1
 * Creation Date:   Wed Jan 28 10:40:38 2009
 * Filename:        NetLink.cc
 * History:
 *      SL      1       Wed Jan 28 10:40:38 2009
 *              Adapted from C version.
 *      SL      2       Thu Jan 27 09:01:34 2011
 *              Changed connection constructor to use strings via getaddrinfo.
 */


/* Necessary for some systems to define sigaction. */
#define _POSIX_SOURCE

#include <netdb.h>
#include <netinet/in.h>
#include <pthread.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>

#include "NetLink.h"

using namespace network_link_module;


/*********************************************************************/
/* Module variables                                                  */


err_t NetLinkEndpoint::init_success = UNINITIALIZED;


/*********************************************************************/
/* Internal functions                                                */


/*
 * NetLink::connectNetLink (uint32_t ip_addr, uint16_t tcp_port)
 *   DESCRIPTION: initialize a new NetLink as a connection to
 *                a specific IP address and TCP port number
 *   INPUTS: ip_addr -- IP address of server, in network byte order
 *           tcp_port -- TCP port number of server, in network byte order
 *   OUTPUTS: none
 *   RETURN VALUE: a new netlink
 *   SIDE EFFECTS: none
 */
void NetLink::connectNetLink (uint32_t ip_addr, uint16_t tcp_port)
{
    /* Set address and port for connection. */
    addr.sin_addr.s_addr = ip_addr;
    addr.sin_port        = tcp_port;

    /* Create socket and try to make connection. */
    if (-1 == (fd = socket (PF_INET, SOCK_STREAM, 0)) ||
        -1 == (connect (fd, (struct sockaddr*)&addr, sizeof (addr)))) {

        /* Failure. */
        close_endpoint ();
    }
}


/*********************************************************************/
/* Module initialization function                                    */

/*
 * NetLinkEndpoint::initialize
 *   DESCRIPTION: initialize the NetLink module; called before
 *                any other interface function is called
 *   INPUTS: none
 *   OUTPUTS: none
 *   RETURN VALUE: NO_ERROR on success, or INIT_FAILED on failure
 *   SIDE EFFECTS: sets the process' SIGPIPE disposition
 */
err_t NetLinkEndpoint::initialize ()
{
    struct sigaction sa; /* used to set SIGPIPE behavior */

    /*
     * SIGPIPE is generated when a write system call is made to a socket
     * that has been closed remotely, and the default behavior for this
     * signal is to terminate the process.
     *
     * This behavior creates a security issue for any server: a client
     * can cause termination of the server by simply making a request
     * and immediately closing the connection.  The server reads the
     * request and issues a response (possibly even something like "go
     * away unless you want to authenticate yourself"), at which point
     * the server receives a SIGPIPE and terminates.
     *
     * SIGPIPE can be safely ignored; write will return an error, and
     * such exceptions can be handled by the caller as they see fit.
     */

     /* Read the current action to initialize sigaction fields. */
     if (-1 == sigaction (SIGPIPE, NULL, &sa)) {
         return INIT_FAILED;
```

```
    }

    /* Set the behavior to ignore SIGPIPE. */
    sa.sa_handler = SIG_IGN;
    if (-1 == sigaction (SIGPIPE, &sa, NULL)) {
        return INIT_FAILED;
    }

    /* All done. */
    return NO_ERROR;
}


/*************************************************************************/
/* NetLink interface functions                                           */


/*
 * NetLink constructor (IP address and TCP port)
 *   DESCRIPTION: create a new NetLink to a specified IP address
 *                and TCP port number
 *   INPUTS: ip_addr -- the IP address of the server, in host byte order
 *           tcp_port -- the TCP port number, in host byte order
 *   OUTPUTS: none
 *   RETURN VALUE: none
 *   SIDE EFFECTS: none
 */
NetLink::NetLink (uint32_t ip_addr, uint16_t tcp_port)
{
    /* Reorder the bytes and pass it along. */
    connectNetLink (htonl (ip_addr), htons (tcp_port));
}


/*
 * NetLink constructor (host name and TCP port)
 *   DESCRIPTION: create a new NetLink to a specified host name
 *                and TCP port number
 *   INPUTS: host_name -- the name of the server (a string, e.g.,
 *                        "www.google.com")
 *           tcp_port -- the name of the TCP port (a string, e.g., "80")
 *   OUTPUTS: none
 *   RETURN VALUE: none
 *   SIDE EFFECTS: uses DNS to translate host name into IP address
 */
NetLink::NetLink (const char* host_name, const char* tcp_port)
{
    struct addrinfo     constraints;  /* constraints for address selection */
    struct addrinfo*    a_list;       /* list of matching addresses        */
    struct sockaddr_in* addr_in;      /* matching IPv4 address             */

    /* Check arguments. */
    if (NULL == host_name || NULL == tcp_port) {
        throw NetLinkBadArgumentsException ();
    }

    /* Set constraints to request a list of matching IPv4 TCP addresses. */
    memset (&constraints, 0, sizeof (constraints));
    constraints.ai_family   = AF_INET;
    constraints.ai_socktype = SOCK_STREAM;

    /* Obtain the address list. */
    /* Discarded getaddrinfo return value provides information
       on failure (FIXME). */
    if (0 == getaddrinfo (host_name, tcp_port, &constraints, &a_list) &&
        NULL != a_list) {

        /* Try only the first address returned. */
        addr_in = (struct sockaddr_in*)a_list->ai_addr;
        connectNetLink (addr_in->sin_addr.s_addr, addr_in->sin_port);

        /* Free returned list of addresses. */
        freeaddrinfo (a_list);
    }
}


/*
 * NetLink constructor (accept incoming connection from server)
 *   DESCRIPTION: accept an incoming connection to a NetLinkServer
 *   INPUTS: server -- the NetLinkServer
 *   OUTPUTS: none
 *   RETURN VALUE: a new netlink
 *   SIDE EFFECTS: if an error occurs, the server netlink is deactivated
 */
NetLink::NetLink (NetLinkServer* server)
{
    socklen_t len; /* len of socket address passed/returned */

    /* Check arguments. */
    if (!server->is_active ()) {
        throw NetLinkInactiveException ();
    }

    len = sizeof (addr);
    if (-1 == (fd = accept (server->fd, (struct sockaddr*)&addr, &len)) ||
        sizeof (addr) != len) {

        /*
         * Something happened.  Close both the new NetLink and the
         * NetLinkServer.
         */
        close_endpoint ();
        server->close_endpoint ();
    }
}


/*
 * NetLink::blocking_read
 *   DESCRIPTION: read a fixed number of bytes from a NetLink;
 *                blocks until all bytes are read, the NetLink is closed,
 *                or an error occurs
 *   INPUTS: buf -- a buffer into which to read the bytes
 *           n_bytes -- the number of bytes to be read
 *   OUTPUTS: buf -- the bytes read from the NetLink (if successful)
 *   RETURN VALUE: NO_ERROR, or an error number on failure
 *   SIDE EFFECTS: if an error occurs, the NetLink is shut down; buf
 *                 contents may change even if an error occurs
 */
err_t
NetLink::blocking_read (void* buf, int32_t n_bytes)
{
    int one_read;   /* bytes read in one call to read */
    int bytes_read; /* bytes read so far              */
    char* bptr;     /* pointer to buffer              */
```

```
    /* Check arguments. */
    if (NULL == buf || 0 >= n_bytes) {
        throw NetLinkBadArgumentsException ();
    }
    if (!is_active ()) {
        throw NetLinkInactiveException ();
    }

    /*
     * Repeat read call until either n_bytes are read, socket is closed,
     * or error occurs.
     */
    for (bptr = (char*)buf, bytes_read = 0; n_bytes > bytes_read;
         bptr += one_read, bytes_read += one_read) {
        one_read = read (fd, bptr, n_bytes - bytes_read);
        if (0 >= one_read) {
            close_endpoint ();
            return (0 == one_read ? WAS_CLOSED : READ_WRITE_ERROR);
        }
    }

    /* Success! */
    return NO_ERROR;
}


/*
 * NetLink::blocking_write
 *   DESCRIPTION: write a fixed number of bytes to a NetLink;
 *                blocks until all bytes are written, the NetLink is closed,
 *                or an error occurs
 *   INPUTS: n -- the NetLink
 *           buf -- a buffer containing the bytes to be written
 *           n_bytes -- the number of bytes to be written
 *   OUTPUTS: none
 *   RETURN VALUE: NO_ERROR, or an error number on failure
 *   SIDE EFFECTS: if an error occurs, the NetLink is shut down
 */
err_t
NetLink::blocking_write (const void* buf, int32_t n_bytes)
{
    int one_write;     /* bytes written in one call to write */
    int bytes_written; /* bytes written so far               */
    const char* bptr;  /* pointer to buffer                  */

    /* Check arguments. */
    if (NULL == buf || 0 >= n_bytes) {
        throw NetLinkBadArgumentsException ();
    }
    if (!is_active ()) {
        throw NetLinkInactiveException ();
    }

    /*
     * Repeat write call until either n_bytes are written, socket is
     * closed, or error occurs.
     */
    for (bptr = (const char*)buf, bytes_written = 0; n_bytes > bytes_written;
         bptr += one_write, bytes_written += one_write) {
        one_write = write (fd, bptr, n_bytes - bytes_written);
        if (0 >= one_write) {
            close_endpoint ();
            return (0 == one_write ? WAS_CLOSED : READ_WRITE_ERROR);
        }
    }

    /* Success! */
    return NO_ERROR;
}


/*************************************************************************/
/* NetLink interface functions                                           */


/*
 * NetLinkServer constructor (TCP port)
 *   DESCRIPTION: create a new NetLinkServer bound to a specific local
 *                TCP port number
 *   INPUTS: port -- the local TCP port number, in host byte order
 *   OUTPUTS: none
 *   RETURN VALUE: none
 *   SIDE EFFECTS: none
 */
NetLinkServer::NetLinkServer (uint16_t tcp_port)
{
    int one = 1; /* used to turn on address reuse */

    /* Set port as requested before attempting to bind. */
    addr.sin_port = htons (tcp_port);

    /*
     * Create socket, ask for reuse of address/port pair (avoid waiting),
     * bind to requested port, and enter server mode to wait for incoming
     * connections (does NOT block until accept is called).
     */
    if (-1 == (fd = socket (PF_INET, SOCK_STREAM, 0)) ||
        -1 == setsockopt (fd, SOL_SOCKET, SO_REUSEADDR, &one, sizeof (one)) ||
        -1 == bind (fd, (struct sockaddr*)&addr, sizeof (addr)) ||
        -1 == listen (fd, 100)) {

        /* Something failed!  Clean up and return failure. */
        close_endpoint ();
    }
}
```