Appendix A Network Programming with Unix Sockets
(Revision number 2)

A.1 The problem

Write a simple client server program for two processes (Client and Server) to communicate using Unix sockets. Client executes on a machine whose Internet name is **`beehive.cc.gatech.edu`**; Server executes on a machine whose Internet name is **`mit.edu`**; the port number to use with the socket at the Server is 2999. The sockets use TCP/IP as the underlying protocol for communication. Once connected, the Client sends a string "Hello World! Client is Alive!" On receiving this message, the Server replies with a string "Got it! Server is Alive!" The Client and Server print what they successfully sent as well as what they successfully received, respectively, on **`stdout`**, close the connection, and terminate.

A.2 Source files provided

In the subsequent sections, the client code, server code, and Makefile are given. Make sure to use the right binary output for your target machine. For simplicity, use the same target machines for both the client and the server. For example, a binary file built on x86_64 architecture will not run on the i686 unless you use -m32 option for compatibility. (try 'uname -m' to check out architecture). The Makefile provided in the appendix uses "-m32" as default.

If you are not sure about any system calls in the example, use the Unix man-pages for detailed explanation. Not sure about the 'man' command itself? Type 'man man'. For example, if you want know about the socket system call, type 'man socket' on any Unix machine.

A.3 Makefile

```
############################################################
##
## Makefile: CS2200 Client/Server Example
##                  Using Unix Sockets
## Author:    Junsuk Shin
##
```

```
#############################################################
CFLAGS  = -Wall -pedantic -m32
LFASGS  = -m32
CC      = gcc
RM      = /bin/rm -rf

SERVER  = server
CLIENT  = client

SERVER_SRC  = server.c
CLIENT_SRC  = client.c
SRCS    = $(SERVER_SRC) $(CLIENT_SRC)

SERVER_OBJ  = $(patsubst %.c,%.o,$(SERVER_SRC))
CLIENT_OBJ  = $(patsubst %.c,%.o,$(CLIENT_SRC))
OBJS    = $(SERVER_OBJ) $(CLIENT_OBJ)

# pattern rule for object files
%.o: %.c
        $(CC) -c $(CFLAGS) $< -o $@

#-------------------------------------------------------------
all: $(SERVER) $(CLIENT)

$(SERVER): $(SERVER_OBJ)
        $(CC) $(LFASGS) -o $@ $<

$(CLIENT): $(CLIENT_OBJ)
        $(CC) $(LFASGS) -o $@ $<

clean:
        $(RM) $(OBJS) $(SERVER) $(CLIENT) core*

.PHONY: depend
depend:
        makedepend -Y -- $(CFLAGS) -- $(SRCS) 2>/dev/null

# DO NOT DELETE

server.o: example.h
client.o: example.h
```

A.4 Common header file

```
#ifndef EXAMPLE_H
#define EXAMPLE_H

#define SERVER_PORT 2999
#define SERVER_MSG  "Got it!  Server is Alive!"
#define CLIENT_MSG  "Hello World!  Client is Alive!"

#define MAXPENDING  5
#define BUFF_SIZE   128

#endif
```

A.5 Client source code

```
#include "example.h"
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <netdb.h>

const char usage[] = "Usage: client [-h] [-p <server port>]
[-s <server address>]\n";

char *server_addr = NULL;
int server_port = SERVER_PORT;

void print_usage(void) {
    printf("%s",usage);
    exit(EXIT_FAILURE);
}

void read_options(int argc, char *argv[]) {
    int c;

    /* read command line options */
    while ( (c=getopt(argc,argv,"hp:s:")) != -1 ) {
        switch(c) {
        case 'p':
            server_port = atoi(optarg);
            if ( server_port <= 0 ) {
                print_usage();
            }
```

```
            break;
        case 's':
            server_addr = optarg;
            break;
        case 'h':
        default:
            print_usage();
        }
    }
    if ( server_addr == NULL ) {
        print_usage();
    }
}

int connect_to(char *host, int port) {
    int sock;
    struct addrinfo hint;
    struct addrinfo *addr;
    char port_str[8];

    /*
     * making connection from a client side is simpler than
     * a server side
     * It follows 2 steps:
     *      1) make socket (socket)
     *      2) make connection (connect)
     */
    memset(&hint, 0, sizeof(hint));
    hint.ai_family = PF_INET;
    hint.ai_socktype = SOCK_STREAM;
    snprintf(port_str,8,"%d",port);

    /* First, need to find out network address with a given
     * host name. host can be any form such as
     * host name – tokyo.cc.gatech.edu or
     * dotted decimal notation – 192.168.1.1.
     * For a descriptive name, gethostbyname() or
     * gethostbyname_r() can be used. The behavior of
     * gethostbyname() when passed a numeric address
     * string is unspecified. For a dotted notation,
     * inet_addr() can be used.
     * Since the return value from above 2 system calls are
     * different, it
     * should be handled in a different way.
     * In this example, getaddrinfo() is used, and it
     * simply handles both cases.
     */
```

```
    /* The getaddrinfo() function shall translate the name
     * of a service location (for example, a host name)
     * and/or a service name and shall return
     * a set of socket addresses and associated information
     * to  be  used  in creating a socket with which to
     * address the specified service.
     */
    getaddrinfo(host, port_str, &hint, &addr);

    /* Second, make socket
     * (use addrinfo set by getaddrinfo() )
     * It can be simply
     * socket(PF_INET, SOCK_STREAM, IPPROTO_TCP) for a
     * tcp/ip connection.
     */
    if((sock = socket(addr->ai_family,
                      addr->ai_socktype,
                      addr->ai_protocol)) == -1) {
        perror("socket");
        return -1;
    }

    /* Third, connect a socket
     * Since, getaddrinfo() sets what we need, use them
     * here.
     * For example, if gethostbyname() is used to resolve
     * network address, you might use a different address
     * here.
     */
    if(connect(sock,
               addr->ai_addr,
               addr->ai_addrlen) == -1 ) {
        perror("connect");
        return -1;
    }

    free(addr);

    return sock;
}

int main(int argc, char *argv[]) {
    int     socket;
    int     sent_size, recv_size;
    char    buffer[BUFF_SIZE];
```

```
read_options(argc,argv);

socket = connect_to(server_addr, server_port);
if ( socket <= 0 ) {
    return EXIT_FAILURE;
}

/* send a message from a socket */
/* Usually, recv/send doesn't fail, but always needs to
 * check (good programming habit!!!)
 */
sent_size = send(socket,
            CLIENT_MSG,
            strlen(CLIENT_MSG)+1,0);

if ( sent_size == -1 ) {
    perror("send");
    exit(EXIT_FAILURE);
}
printf("Message sent to %s\n\t%s\n",
                                server_addr,CLIENT_MSG);

/* Receive a message from a connected socket */

/*
 * buffer for the message needs to be provided.
 * It returns the length of the message written to
 * the buffer unless there is an error.
 * By default, it's blocking call. If interested,
 * check out fcntl(), O_NONBLOCK, and EAGAIN return
 * value.
 */
recv_size = recv(socket,buffer,BUFF_SIZE,0);
if ( recv_size == -1 ) {
    perror("recv");
    exit(EXIT_FAILURE);
}
printf("Message received from %s\n\t%s\n",
                                    server_addr,buffer);

/* Close sockets */

/*
 * Not really necessary since the program terminates,
 * But it's good habit. Same for file descriptors and
 * sockets. Also, there's a limitation for such
```

```
     * descriptors that a user can open. Simply, you cannot
     * open file/make socket above the limitation.
     */
    close(socket);
    return EXIT_SUCCESS;
}
```

A.6 Server source code

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include "example.h"

union sock {
    struct sockaddr s;
    struct sockaddr_in i;
};

const char usage[] =
                "Usage: server [-h] [-p <port number>]\n";
int port = SERVER_PORT;
char client_ip[16];

void print_usage(void) {
    printf("%s",usage);
    exit(EXIT_FAILURE);
}

void get_options(int argc, char *argv[]) {
    int c;

    /* read command line options      */
    /* "hp:" means -h and -p <string> */
    while ( (c=getopt(argc,argv,"hp:")) != -1 ) {
        switch(c) {
        case 'p':
            port = atoi(optarg);
            if ( port <= 0 ) {
                print_usage();
            }
            break;
```

```
        case 'h':
        default:
            print_usage();
        }
    }
}

int create_server_socket(int port) {
    struct sockaddr_in serv_addr;
    int serv_sock;
    int opt = 1;

    /* Creation of server socket usually follows these 3
     * steps:
     *     1) make socket / create endpoint of
     *        communication
     *     2) bind a name (address) to a socket
     *     3) listen for incoming socket connections
     */
    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = PF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port = htons(port);

    /* make TCP socket */
    /* PF_INET : IP protocol family
     *            For more options, check out
     *                          /usr/include/bits/socket.h
     *            (e.g. PF_UNIX for unix domain socket)
     * SOCK_STREAM : sequenced, reliable connection
     *                (e.g. SOCK_DGRAM for connectionless,
     *                      unreliable connection)
     * IPPROTO_TCP : Transmission Control Protocol/TCP
     *                (e.g. IPPROTO_UDP, IPPROTO_RSVP, etc)
     *                check out /usr/include/netinet/in.h
     *                          /usr/include/linux/in.h
     */
    if ((serv_sock=socket(PF_INET,
                          SOCK_STREAM,
                          IPPROTO_TCP)) == -1 ) {
        perror("socket");
        return -1;
    }

    /* Set port as reusable */

    /*
```

```
     * Port may not be usable if it's not closed properly
     * (e.g. segfault, kill process) Specifies that the
     * rules used in validating addresses supplied to
     * bind() should allow reuse of local addresses
     */
    if (setsockopt(serv_sock,
                   SOL_SOCKET,
                   SO_REUSEADDR,
                   &opt,
                   sizeof(opt)) == -1 )
    {
        perror("setsockopt");
        return -1;
    }

    /* bind a name (server_addr) to a socket (serv_sock)
     * When a socket is created with socket(), it exists in
     * a name space (address family) but has no name
     * assigned.
     *
     * It is normally necessary to assign a local address
     * using bind before a SOCK_STREAM socket may receive
     * connections (accept()).
     */
    if (bind(serv_sock,
             (struct sockaddr *)&serv_addr,
             sizeof(serv_addr)) == -1 ) {
        perror("bind");
        return -1;
    }

    /*
     * Listen for socket connections and limit the queue of
     * incoming
     */
    if (listen(serv_sock,MAXPENDING) == -1 ) {
        perror("listen");
        return -1;
    }

    return serv_sock;
}


void read_client_ip(int sock) {
    union sock client;
    int client_len;
```

```
    client_len = sizeof(struct sockaddr);
    /* get the name of the peer socket */
    getpeername(sock,
                &(client.s),
                (socklen_t *)&client_len);

    /* inet_ntoa() convert the Internet host address to a
     * string in the Internet standard dot notation.
     */
    strncpy(client_ip,inet_ntoa(client.i.sin_addr),16);
}

int main(int argc, char *argv[]) {
    int                 server_sock, client_sock;
    int                 recv_size,sent_size;
    struct sockaddr_in  c_addr;
    unsigned int        c_len = sizeof(c_addr);
    char                buffer[BUFF_SIZE];

    get_options(argc,argv);

    server_sock = create_server_socket(port);

    /* Waiting on a client connection */

    /*
     * The  accept  function  is  used  with  connection-
     * based  socket  types:
     * (SOCK_STREAM, SOCK_SEQPACKET and SOCK_RDM).
     * It  extracts  the  first connection  request on the
     * queue of pending connections, creates a new
     * connected socket with mostly the same properties as
     *  s, and allocates a new file descriptor for the
     * socket, which is returned.
     */
    if ( (client_sock=accept(server_sock,
                             (struct sockaddr *)&c_addr,
                             &c_len)) == -1 ) {
        perror("accept");
        return EXIT_FAILURE;
    }

    read_client_ip(client_sock);

    /* Receive a message from a connected socket */
```

```
    /*
     * Buffer for the message needs to be provided .
     * It returns the length of the message written to
     * the buffer unless there is an error.
     * By default, it's blocking call. If interested,
     * check out fcntl(), O_NONBLOCK, and EAGAIN return
     * value.
     */
    recv_size = recv(client_sock,buffer,BUFF_SIZE,0);
    if ( recv_size == -1 ) {
        perror("recv");
        exit(EXIT_FAILURE);
    }
    printf("Message received from %s\n\t%s\n",
                                        client_ip,buffer);

    /* Send a message from a socket */
    /* Usually, recv/send doesn't fail, but always needs to
check
     * (good programming habit)
     */
    sent_size = send(client_sock,
                     SERVER_MSG,
                     strlen(SERVER_MSG)+1, 0);
    if ( sent_size == -1 ) {
        perror("send");
        exit(EXIT_FAILURE);
    }
    printf("Message sent to %s\n\t%s\n",
                                     client_ip,SERVER_MSG);

    /* Close sockets */

    /*
     * Not really necessary since the program terminates,
     * but it's good habit. Same for file descriptors and
     * sockets. Also, there's a limitation for such
     * descriptors that a user can open. Simply, you cannot
     * open file/make socket above the limitation.
     */
    close(client_sock);
    close(server_sock);

    return EXIT_SUCCESS;
}
```

A.7 Instantiating the client/server programs

The example program uses the following default for the server port:

- Server port: 2999
  This option is changeable via command line.

To run the program you have to start the client and server programs:

- Server
  run server (if necessary with different options)
  for e.g.,
  ./server (run with default options)
  ./server -p 3333 (to set up 3333 as server port number)

- Client
  run client (if necessary with different options)
  for e.g.,

```
./client -s < host ip | host name >
./client -p 3333 -s helsinki.cc.gatech.edu
```