Practices on Computer Networks and Distributed Systems

Wall clock time and Distributed Systems

1. Sending and receiving ICMP timestamps in C with the Raw Sockets API

Raw Sockets allow programmatic access to the IP layer. In this practical we use Raw Sockets to write a program that sends and receives specific types of ICMP messages with the intention of retrieving the time of of day of another host that will ultimately allow it to have its clock synchronized. Synchronizing a clock entails being able to speed it up and to slow it down. The POSIX-compatible function that in Linux allows the clock to be sped up/slowed down in order for the clock to be synchronized is adjtime().

Exercise 1.

a. Check the type of clock synchronization client that is in use in your Linux

```
$ systemctl status systemd-timesyncd
```

If your system is using the full fledged Linux NTP client, follow the relevant instructions in section 2, below.

b. Assuming that your system is using systemd-timesyncd, read the resulting listing and confirm the IP address of the time server that your system was using until now. Check that the transport protocol is UDP and that the associated port is 123.

c. Stop the NTP client so that the time synchronization program we wish to build doesn't conflict with it in taming the clock:

```
$ systemctl stop systemd-timesyncd
```

d. Print the current system's daytime with a μs precision:

```
$ date --rfc-3339=ns
```

e. Set the clock time 2 minutes later than the current time; see this example:

```
$ date -s '2021-10-22 09:20:00'
```

f. Our example time synchronization program is based on the ICMP protocol's Timestamp request and Timestamp response messages. In order for our program to send ICMP Timestamp request message, we will open a Raw Socket upon our stack's IP protocol which will allows to craft one such

message and later have it sent over the socket towards the Internet host that we want to use as reference for our local clock.

Download the raw sock icmptimestamp.c program from paloalto.junileon.es/ds/lab/icmptimestamp.c and test it. The program should contact the host which IP you entered into the program and, if the ICMP Timestamp request makes it to that host and the response is successfully delivered to our local host, then, the program will calculate the amount of μs necessary to set your local clock in sync with the remote system's clock. The process of updating the local time is achieved by calling Linux function adjtime(). That function will speed up or slow down the clock to achieve the synchronization without ever setting the clock backwards, which is absolutely forbidden. Run the program once and observe that in a few minutes of slowing it down, it will reach the target time. Use the following example as orientation:

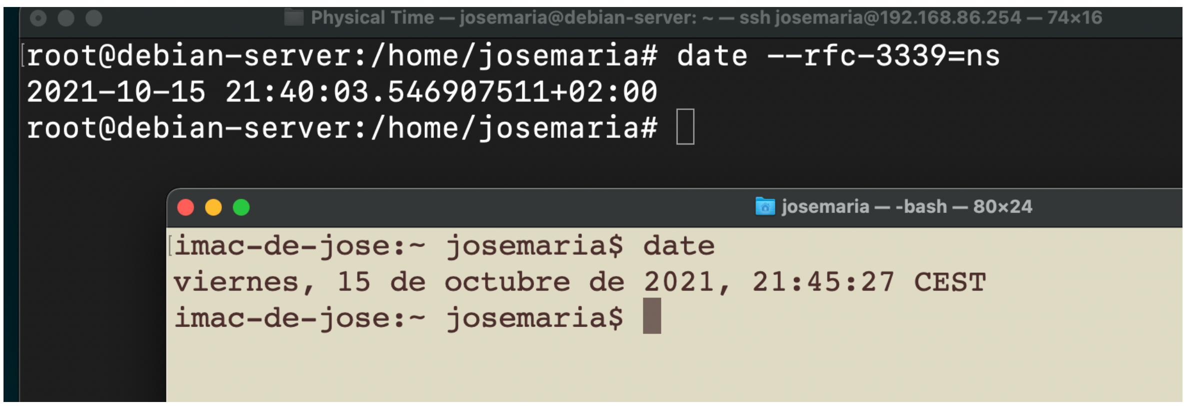

Fig. 1. The Linux clock is about 5 min behind the clock of the OS-X system

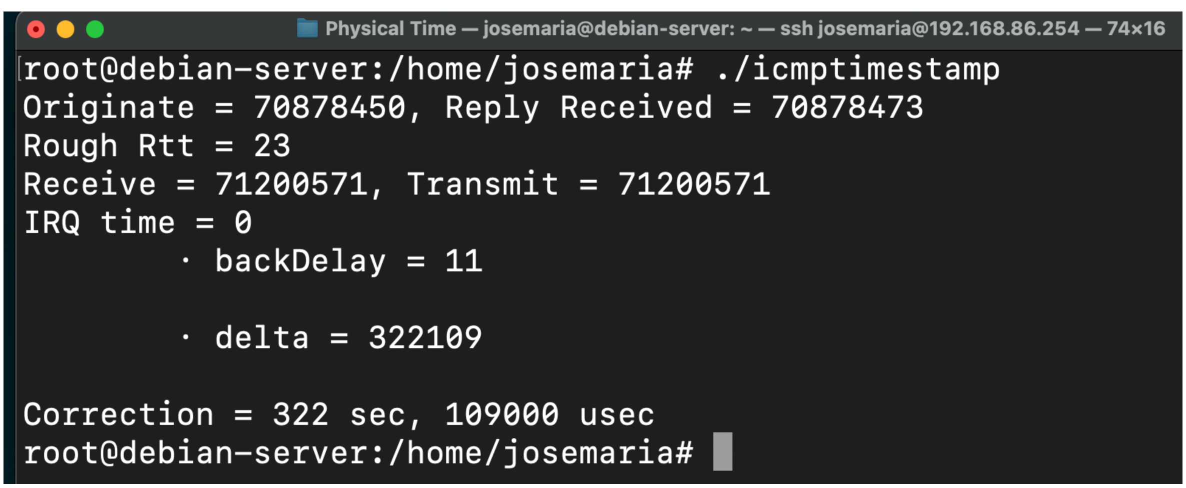

Fig. 2. The icmptimestamp program has sent an ICMP Timestamp request message to 193.146.101.46 and has received an ICMP Timestamp reply. The local clock is now being sped up to reach the clock time of that host.

```
$ sudo su
# ./icmptimestamp
```

The instructor will provide you a guidelines to complete this exercise.

Exercise 2. Have icmptimestamp program emulate the behavior of the Christian's algorithm by repeatedly retrieving the time from another system, then calculating the average and finally *properly* using adjtime() for adjusting the clock. You will observe that repeated calls to the program time retrieving function fail, then, we will have to solve this problem by modifying the program so that it runs fine in repeated calls? Wireshark will be able to report to you the error condition that you will have to discover and which prevents the aforementioned repeated calls from functioning all right (Make sure you *sudo* the resulting program, for, Raw Sock programs need superuser privileges to be executed).

1. If using OS-X, you need to enable the following property:

```
$ sysctl net | fgrep icmp | fgrep times
net.inet.icmp.timestamp: 0
```

2. Enable it:

```
$ sudo sysctl -w net.inet.icmp.timestamp=1
Password:
net.inet.icmp.timestamp: 0 -> 1
```

3. Check it:

```
$ sysctl net | fgrep icmp | fgrep times
net.inet.icmp.timestamp: 1
```

4. Start Wireshark, set filter 'icmp'

5. Compile the icmp timestamp requester program (gcc …: Ok) and run under Linux and have the program contact the MacBook Air at 192.168.2.? that we just reconfigured:

$ sudo testicmpts 192.168.2.106

6. Monitor the icmp traffic with WS and check the program's output

2. Querying the state of a Unix/Linux NTP Client

As we reviewed in the Lectures, the Internet protocol used today for synchronizing clocks is NTP (Network Time Protocol), we can query its basic parameters by issuing the OS-X and Linux command ntpq and can estimate the Rtt with our NTP server by using ping and traceroute. Before practicing with these commands, let's peek at the RFC of the NTP protocol.

Exercise 3. Donwload and skim NTP's RFC and tell what transport it uses and the port number used by NTP servers.

Exercise 4. Monitor the NTP traffic from your host by running Wireshark, use a display filter 'ntp'; you may have to wait a few minutes before some NTP traffic appears. Confirm that the transport and the port you established in the preceding question are correct. Depict an NTP protocol graph.

Exercise 5. Decode fields in the NTP request packet sent by your client, then, respond to the following questions related to the NTP concepts developed on the lectures:

a. What NTP stratum does your server belong in?

b. What physical connection exists between your server and the physical atomic clock?

c. Explain the meaning of the Reference, Origin and Receive Timestamps

d. What is the field width of the aforementioned timestamps? They're represented with fixed-point numbers that represent a number of seconds, what's the smallest number of seconds representable? And the maximum?

Exercise 6. Issue the following command which queries your NTP client configuration, then, respond to the ensuing questions

```
$ ntpq -pn
```

a. What's the polling frequency used by your client?

b. What is the delay between your client and the NTP server?

c. What does jitter mean? Jitter represents the variability of the delay, does that make sense?

d. What's the Rtt resulting from the execution of ping against the server? Is it equal to the ntpq's delay field?

3. Wall clock time and the ICMP protocol

One of the functions of the icmp protocol allows a system to find out another system's wall clock time so that the difference between both clocks can be calculated and thus, eventually, compensated for. The next experiment consists of two computers, Mac and Linux connected via Internet, the Mac use wants to discover how much its clock is skewed versus that of Linux, to that end, she executes the following Unix (OS-X) command:

```
$ sudo timedc
  Passwd:
  timedc> clockdiff 192.168.2.122
  timedc: 192.168.2.122 will not tell us the date
  time on 192.168.2.122 is 47161 ms. behind time on Mac-2.local
```

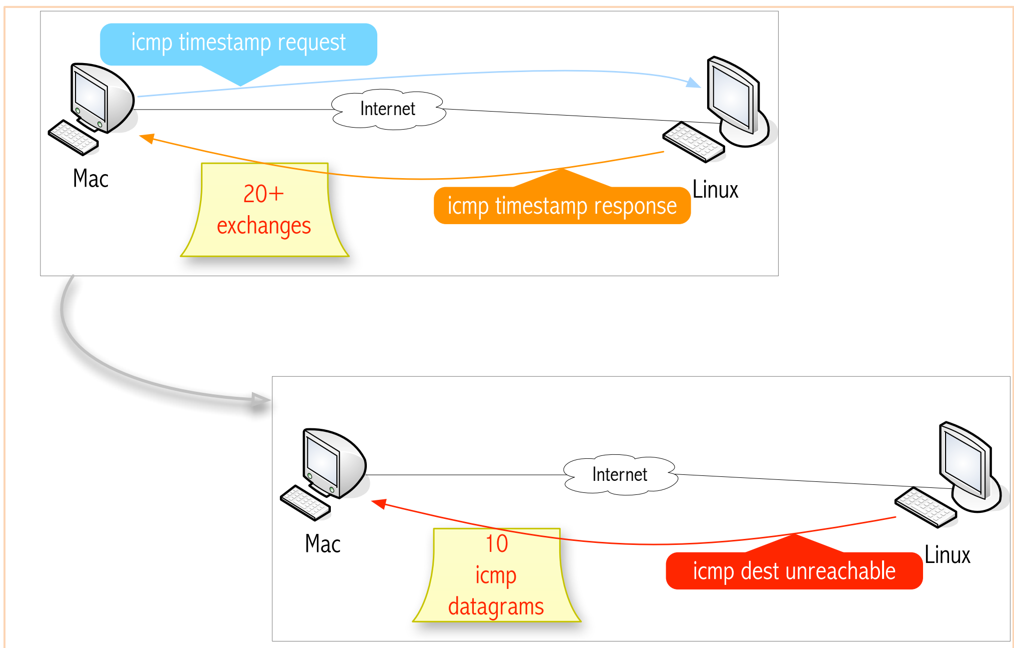

Fig. 3. The Mac computer sends several back-to-back icmp timestamp requests to the Linux computer, after that, the Mac receives 10 icmp destination unreachable messages from Linux

The timedc at Mac (192.168.2.106) command controls the UNIX timed daemon which sends icmp timestamp requests to Linux (192.168.2.122) and, for each one of them it expects to receive an icmp timestamp response; then, the command, interprets the contents of the icmp response and computes the time difference, which in this case results in Linux being 47161 ms *behind* Mac[1]. See the following timed UNIX man page for more detail:

[1] Usually, instead of saying "Mac is behind", "Mac is slow" is used in the field of computer wall clock time.

```
Chema — less — 80×24

TIMED(8)                  BSD System Manager's Manual                 TIMED(8)

NAME
     timed -- time server daemon

SYNOPSIS
     timed [-dtM] [-i network | -n network] [-F host ...]

DESCRIPTION
     The timed utility is a time server daemon which may be invoked at boot
     time via launchd(8).  (See launchd.plist(5) for more information about
     configuring daemons to run at boot.)  It synchronizes the host's time
     with the time of other machines, which are also running timed, in a local
     area network.  These time servers will slow down the clocks of some
     machines and speed up the clocks of others to bring them to the average
     network time.  The average network time is computed from measurements of
     clock differences using the ICMP timestamp request message.
```

Fig.2. The OS-X **$ man timed** highlighting that **timed** is a server program capable of slowing down or speeding up other system's clocks to have them within average network time.

The exercises that follow illustrate the mentioned icmp protocol exchanges. Before starting to work the exercises, please, download the RFC where the ICMP protocol was documented by the IETF, it will be necessary to decode the WS (Wireshark) traces included alongside the exercises (Mac computer's host name is Brainstorm and the Linux computer's is Josephus).

Exercise 6. Observe the Request/Response sequence caused by the execution of timedc at Mac, what protocol encapsulates those ICMP R/R messages? (See fig. 3).

Exercise 7. Consult the RFC that you just downloaded and map the fields of the Type 13 message to those of the WS trace at Fig. 4.

Exercise 8. According to the WS trace of Fig. 5., how many bits make up any of the three relevant timestamp fields of the included ICMP Type 14 message.

Exercise 9. See Fig. 6 which contains the last message of the WS trace and provide a speculative explanation of it.

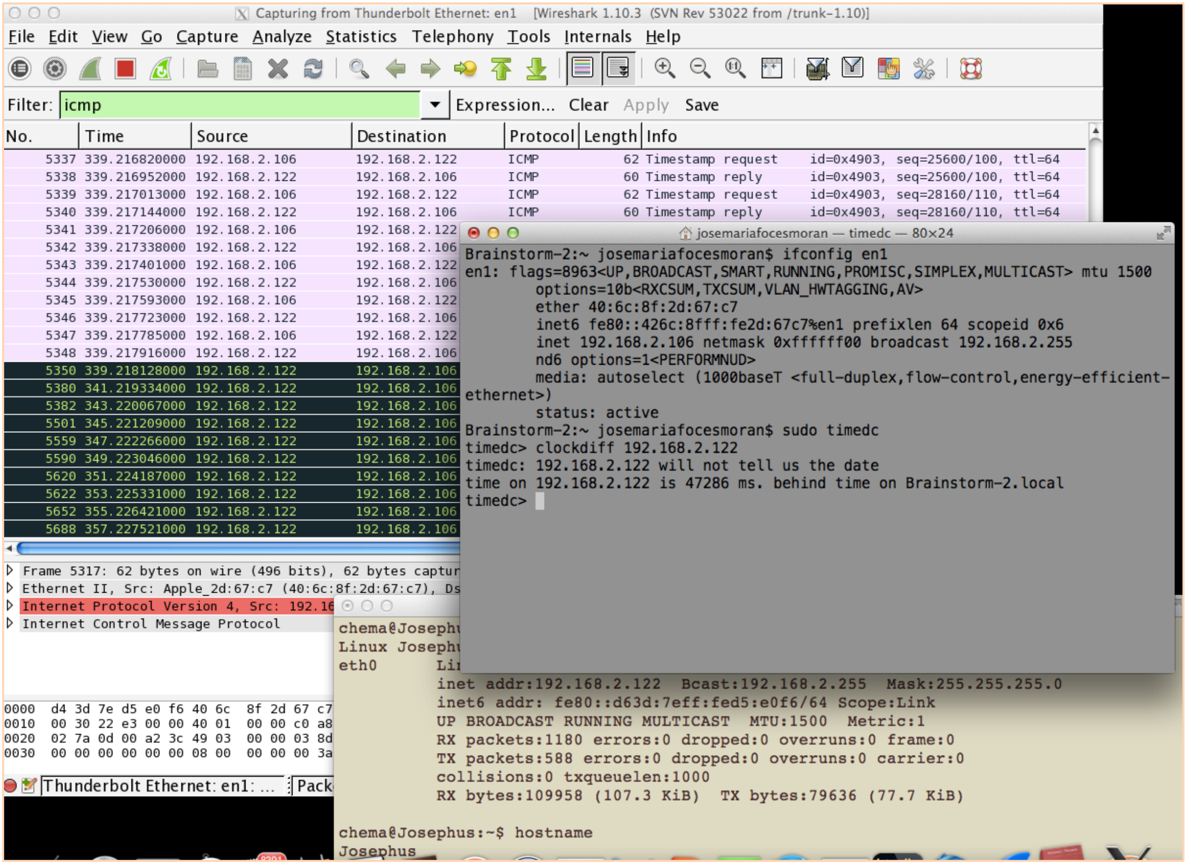

Fig.3. timed daemons exchanging icmp timestamp messages to bring the slave's clock (Linux computer whose name is Josephus) into sync with the server (The MAC computer whose name is brainstorm).

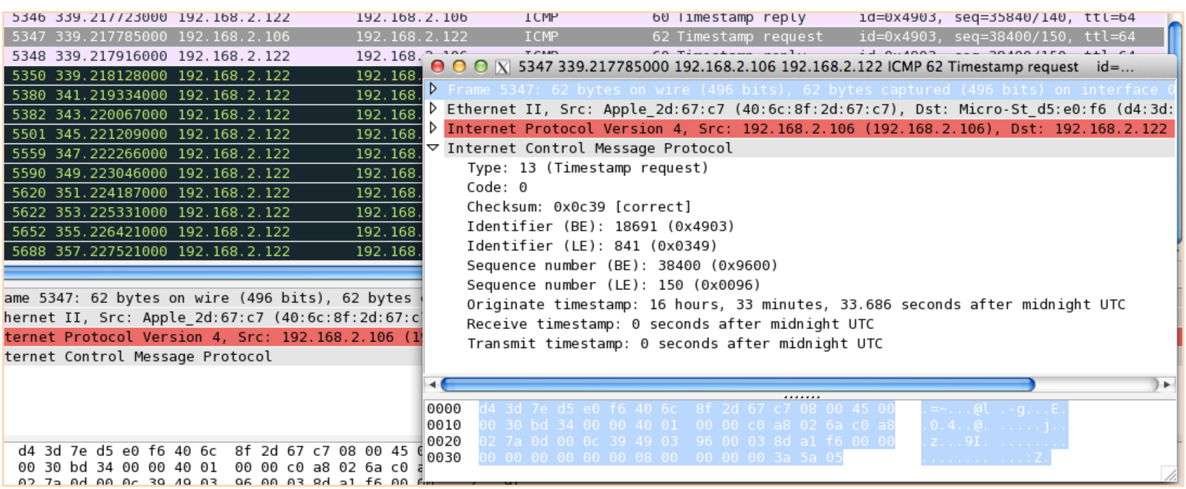

Fig.4. timed daemons exchanging icmp timestamp messages to bring the slave's clock (Linux computer whose name is Josephus) into sync with the server (The MAC computer whose name is brainstorm).

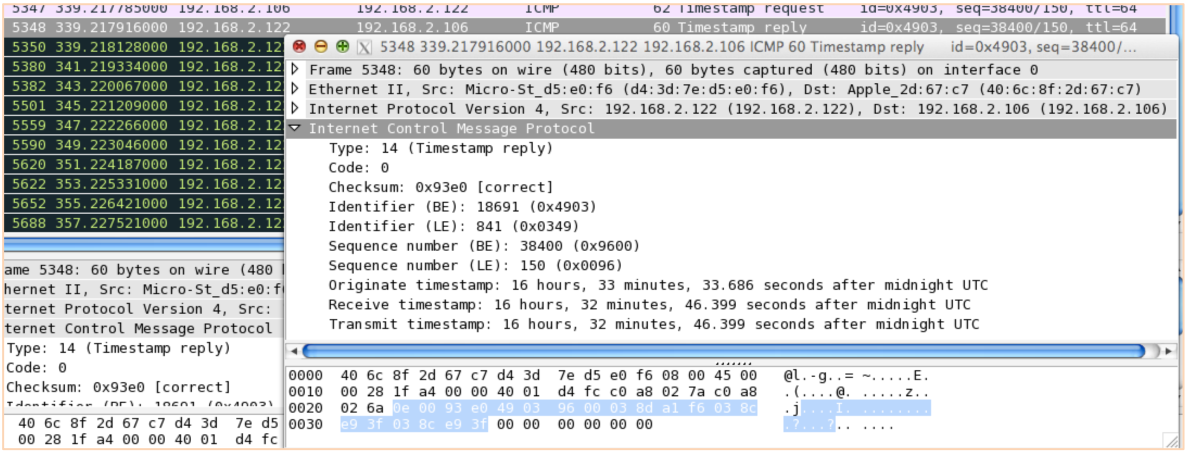

Fig.5. Timestamp reply, concrete ICMP type 14 message fields appearing in WS trace

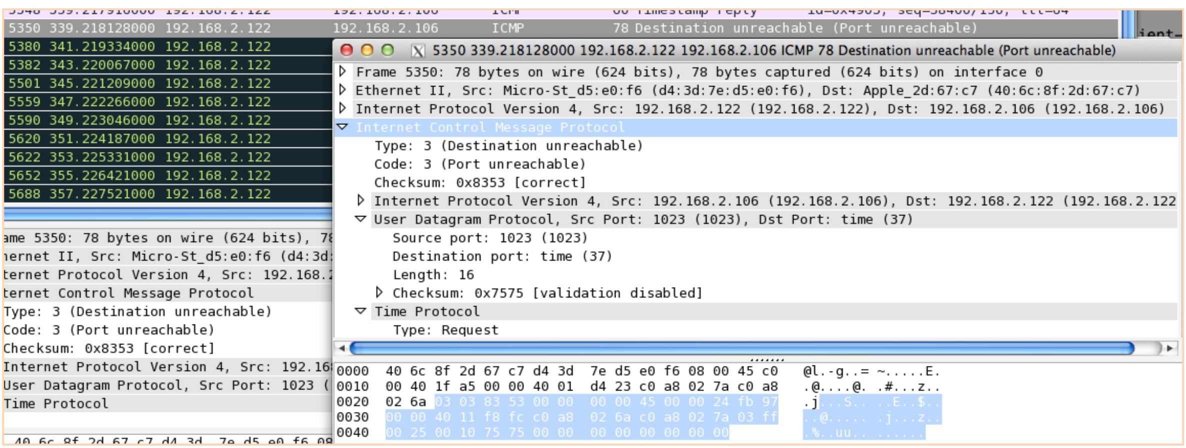

Fig.6. Last icmp message

Appendix A: Source code of the icmptimestamp.c program

```
/*
 * Parts of this source code were taken from the
 * W. Richard Stevens' book on Unix Network Programming
 * (Prentice-Hall 1998)
 *
 * Refactorization, additional comments and adaptation
 * for the purposes of the CN Lab by José María Foces Moran 2014
 *
 * Technical details about the structure of the ICMP datagram and IP
packets
 * may be obtained from RFC
 */


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#include <sys/types.h>
#include <sys/param.h>

#include <sys/time.h>
#include <sys/file.h>

#include <sys/socket.h>
#include <arpa/inet.h>

#include <netinet/in_systm.h>
#include <netinet/in.h>

#include <netinet/ip.h>
#include <netinet/ip_icmp.h>

#include <netdb.h>
#include <unistd.h>

#include <errno.h>

#define TRUE 1

/*
 * #Bytes of data, following ICMP header (Stevens' UNIX Net Programming
book)
 * Data that goes with ICMP echo request
 */
int datalen = 12;

struct sockaddr targetHost; //generic sock address for the target host
int rawSocket;

int nsent = 0;
int pid;

#define MAXIPHEADERLENGTH 60
```

```
#define MAXICMPPAYLOADLENGTH 76
#define MAXIPPACKETSIZE (65536 - 60 - 8)
#define MAGICSEQN 512

struct timeval originateTimeval, receiveTimeval;
long tsorig, tsrecv;

long tsdiff;


int initSocket(char *targetHostIpDDN) {

    int rawSock;
    struct sockaddr_in *inetTargetHost; //inet socket address for target
host
    struct protoent *protocol;

    bzero((char *) &targetHost, sizeof (struct sockaddr));
    inetTargetHost = (struct sockaddr_in *) &targetHost;
    inetTargetHost->sin_family = AF_INET;
    inet_aton(targetHostIpDDN, &(inetTargetHost->sin_addr));

    protocol = getprotobyname("icmp");

    rawSock = socket(AF_INET, SOCK_RAW, protocol->p_proto);

    return rawSock;

} //end of initSock()

unsigned char *createPacket(int *packlen) {

    unsigned char *packet;

    *packlen = datalen + MAXIPHEADERLENGTH + MAXICMPPAYLOADLENGTH;

    packet = (unsigned char *) malloc((unsigned int) *packlen);

    return packet;

}//end of createPacket()

/*
 * The source code of function internetChecksum are original from
 * W. Richard Stevens' book on Unix Network Programming
 * (Prentice-Hall 1998) in its entirety, no modification has been
 * carried out by JMFoces whatsoever except for the function name
 */
unsigned short internetChecksum(u_short *addr, int len) {
    int nleft = len;
    int sum = 0;
    unsigned short *w = addr;
    unsigned short answer = 0;

    while (nleft > 1) {
        sum += *w++;
        nleft -= 2;
    }
```

```
    if (nleft == 1) {
        *(unsigned char *) (&answer) = *(unsigned char *) w;
        sum += answer;
    }

    sum = (sum >> 16) + (sum & 0xffff);
    sum += (sum >> 16);
    answer = ~sum;
    return (answer);
}

void sendRequest() {
    int len;
    struct icmp *icmp;
    unsigned char requestPacket[MAXIPPACKETSIZE];

    icmp = (struct icmp *) requestPacket;
    icmp->icmp_type = ICMP_TSTAMP;
    icmp->icmp_code = 0;
    icmp->icmp_seq = MAGICSEQN;
    icmp->icmp_id = pid;

    gettimeofday(&originateTimeval, NULL);
    tsorig = (originateTimeval.tv_sec % (24 * 3600)) * 1000 +
originateTimeval.tv_usec / 1000;
    icmp->icmp_otime = htonl(tsorig);
    icmp->icmp_rtime = 0;
    icmp->icmp_ttime = 0;

    len = datalen + 8;

    icmp->icmp_cksum = internetChecksum((u_short *) icmp, len);

    sendto(rawSocket, (char *) requestPacket, len, 0, &targetHost, sizeof
(struct sockaddr));

}

int processPacket(char *buf, int n, struct sockaddr_in *from) {
    int headerLength;
    struct icmp *icmp;
    struct ip *ip;
    struct timeval delta;

    //Compute IP header length
    ip = (struct ip *) buf;
    headerLength = ip->ip_hl << 2;

    //Subtract header length from n
    n -= headerLength;
    icmp = (struct icmp *) (buf + headerLength);

    /*
     * Discard all ICMP packets which ICMP type is not
     * RFC 792 type value 14 for timestamp reply message represented by
     * ICMP_TSTAMPREPLY constant
     */
```

```
    if (icmp->icmp_type == ICMP_TSTAMPREPLY) {

        if (icmp->icmp_seq != MAGICSEQN)
            printf("Spurious sequence  received %d\n", icmp->icmp_seq);
        if (icmp->icmp_id != getpid())
            printf("Spurious id received %d\n", icmp->icmp_id);

        //Receive timestamp
        tsrecv = ntohl(icmp->icmp_rtime);

        //Difference between Receive timestamp and originate timestamp:
        tsdiff = tsrecv - tsorig; // ms

        printf("Originate = %ld, receive = %ld\n",
                ntohl(icmp->icmp_otime), ntohl(icmp->icmp_rtime));
        printf("Adjustment = %ld ms\n", tsdiff);

        delta.tv_sec = tsdiff / 1000;
        delta.tv_usec = (tsdiff % 1000) * 1000;
        printf("Correction = %ld sec, %ld usec\n", delta.tv_sec,
delta.tv_usec);

        /* adjtime() makes small adjustments to the system time,
         * as returned by gettimeofday(2), advancing or retarding it by
the time
         * specified by the timeval delta
         *
         * See the man page for adjtime
         */
        adjtime(&delta, (struct timeval *) 0);

        return (0); //Timestamp reply
    } else
        return (-1); //Not timestamp reply
}

void receiveResponse(int packetLength, unsigned char *packet) {
    struct sockaddr_in from;
    int nbytes;
    int fromlen;

    while (TRUE) {
        fromlen = sizeof (from);
        nbytes = recvfrom(rawSocket, (char *) packet, packetLength, 0,
                (struct sockaddr *) &from, &fromlen);

        if (nbytes < 0) {
            printf("Bytes received < 0");
            fflush(stdout);

            if (errno == EINTR)
                continue;
            else
                perror("recvfrom error");
        }

        if (processPacket((char *) packet, nbytes, &from) == 0)
            exit(0);
```

```
    }


}

int main() {
    unsigned char *packet;
    int packetLength;

    pid = getpid();

    rawSocket = initSocket("192.168.2.106");

    packet = createPacket(&packetLength);

    sendRequest();

    receiveResponse(packetLength, packet);

} //end of main()
```