University of Glasgow | School of Computing Science

Assessed Coursework

Course Name	NS3			
Coursework Number	Summative Exercise 2			
Deadline	Time:	9:00am	Date:	9 March 2012
% Contribution to final course mark	4%			
Solo or Group ✓	Solo	✓	Group	
Anticipated Hours	4			
Submission Instructions	Submit via Moodle, in a .tar.gz archive formatted as instructed in the NS3 Lab 4 handout.			
Please Note: This Coursework cannot be Re-Done				

Code of Assessment Rules for Coursework Submission

Deadlines for the submission of coursework which is to be formally assessed will be published in course documentation, and work which is submitted later than the deadline will be subject to penalty as set out below.

The primary grade and secondary band awarded for coursework which is submitted after the published deadline will be calculated as follows:

(i) in respect of work submitted not more than five working days after the deadline
   a. the work will be assessed in the usual way;
   b. the primary grade and secondary band so determined will then be reduced by two secondary bands for each working day (or part of a working day) the work was submitted late.

(ii) work submitted more than five working days after the deadline will be awarded Grade H.

Penalties for late submission of coursework will not be imposed if good cause is established for the late submission. You should submit documents supporting good cause via MyCampus.

Penalty for non-adherence to Submission Instructions is 2 bands

You must complete an "Own Work" form via https://webapps.dcs.gla.ac.uk/ETHICS for all coursework UNLESS submitted via Moodle

NS3 Lab 4 – UDP Programming

Dr Colin Perkins
School of Computing Science
University of Glasgow
`http://csperkins.org/teaching/ns3/`

29 February 2012

Introduction

The laboratory sessions for Networked Systems 3 (NS3) will introduce you to network programming in C on Unix/Linux systems. There are weekly labs for this course, during which you will complete several exercises. These exercises will build on your knowledge of C programming and pthreads from the Advanced Programming 3 course last semester, and on the material in the NS3 lectures. There are a mixture of formative and summative exercises. The formative exercises are intended to give you practice in programming networked systems in C; they are not assessed. The two summative exercises are assessed, and are worth a total of 20% of the marks for this course.

This is NS3 lab 4, on UDP programming. It comprises one formative exercise and one summative exercise. The formative exercise should be completed during the timetabled laboratory session in week 8 of the semester. The summative exercise that should be completed during the timetabled laboratory sessions in weeks 8 and 9 of the semester, and during other hours as necessary. This work is assessed, and is worth 4% of the marks for this course.

Background: UDP

The user datagram protocol (UDP) provides an unreliable and connectionless datagram service to applications. It is primarily used by local-area request-response protocols such as the DNS, or for applications such as voice-over-IP that prefer timeliness to reliability. UDP behaviour stands in sharp contrast to TCP, which provides a reliable, connection oriented, stream abstraction.

A UDP socket can be created using the `socket()` system call in the usual manner, but specifying `SOCK_DGRAM` as the socket type:

```
int fd = socket(AF_INET, SOCK_DGRAM, 0);
```

Once a UDP socket has been created, it should be bound to a known port if it is expected to act as a server that sends and receives datagrams. This is done using the `bind()` system call, in exactly the same was as for a TCP socket. The arguments to `bind()` indicate the local address and port to which the socket should be bound. Since UDP is connectionless, there is no need to call the `listen()`, `accept()`, or `connect()` functions.

UDP datagrams can be transmitted using the `sendto()` function. This takes as arguments a file descriptor representing the socket, a buffer of data to the transmitted, the length of that buffer, and the addresses and port to which that buffer should be sent. The destination address is specified as a `struct sockaddr *`, and a corresponding size. If sending to a unicast address, this should be looked up in the DNS using `getaddrinfo()` in much the same way that you look up the address used in a TCP `connect()` (but taking just the first address returned), and using this as the destination of the datagram. Since the destination address is specified in the `sendto()` call, it is possible to send each datagram to a different destination.

```
int                fd;
char               buffer[...];
int                buflen = sizeof(buffer);
struct sockaddr_in addr = ...;
...
if (sendto(fd, buffer, buflen, 0,
           (struct sockaddr *) addr,
           sizeof(addr)) < 0) {
  // Error...
}
```

The `recvfrom()` function can be used to receive UDP datagrams. This works in much the same way as `read()`, except that it also takes an empty address structure (`struct sockaddr *`) that is filled in with source address and port from the received datagram. This address can be used in a `sendto()` call to send a reply. Each received datagram can come from a different source address.

```
int             fd;
char            buffer[...];
int             buflen = sizeof(buffer);
struct sockaddr addr;
socklen_t       alen = sizeof(addr);
int             rlen;
...
rlen = recvfrom(fd, buffer, buflen, 0, &addr, &alen);
if (rlen < 0) {
  // Error...
}
```

A UDP socket must be closed in the usual way, once you have finished using it, using the `close()` system call.

Formative Exercise 5: UDP Client/Server Example

The formative exercise for this lab demonstrates how to build the most simple UDP-based client-server application. You should write two programs:

`udp_hello_server` The server should listen for datagrams on UDP port 5008. It should read the first datagram received, print the contents of that datagram to the screen, close the socket, then exit.

`udp_hello_client` Your client should send the text "Hello, world!" in a UDP datagram to port 5008 of a host named on the command line, then it should close the socket. The client should take the name of the machine on which the server is running as its single command line argument (i.e., if the server is running on machine `bo720-1-01` you should run your client using the command `hello_client bo720-1-01`.

Run your client and server, and demonstrate that you can send the text "Hello, world!" from one to the other. Try this with client and server running on the same machine, and with them running on two different machines.

Background: Multicast

In addition to standard point-to-point (unicast) transmission, UDP can also be used with the IP multicast service. Multicast groups are identified by IP addresses in the range 224.0.0.0 to 239.255.255.255. IP addresses in this range differ from other IP addresses in that they identify a *group* of receivers, rather than a single host. A UDP datagram sent to a multicast address is delivered to all hosts that have joined that group. A host may join a multicast group by calling the `setsockopt()` function with the address of the group to join (this is done after binding to a port):

```
struct ip_mreq  imr;

inet_pton(AF_INET, "224.0.0.22", &(imr.imr_multiaddr.s_addr));
imr.imr_interface.s_addr = INADDR_ANY;

if (setsockopt(fd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
                                     &imr, sizeof(imr)) < 0) {
  // Error...
}
```

Once a host has joined a multicast group, it will receiv datagrams sent to that group address. A host does not need to join a group in order to send to that group.

Note that multicast addresses generally do not have DNS entries, so you must specify them as raw IP addresses (using `inet_pton()`) rather than looking them up in the DNS using `getaddrinfo()`. Multicast addresses cannot be used with TCP connections, since TCP is a point-to-point protocol, and only supports one sender and one receiver.

A host can leave a multicast group by calling the `setsockopt()` function with the `IP_DROP_MEMBERSHIP` option:

```
if (setsockopt(fd, IPPROTO_IP, IP_DROP_MEMBERSHIP,
                                    &imr, sizeof(imr)) < 0) {
  perror("Unable to join group");
  return 1;
}
```

Receivers should leave any multicast groups that they have joined before they `close()` the underlying sockets.

Summative Exercise 2: Multicast Chat

The second summative exercise demonstrates how to build a simple UDP multicast chat application. You should write two programs:

`chirp` The chirp client sends a chat message to the group. It is invoked with a single command line argument containing the messages to be sent, for example: `chirp "hello world"`. The message to be sent must be enclosed in quotes, else the shell will interpret it as several arguments to pass to the chirp command. The maximum length message that should be supported is 1000 characters.

The chirp client will send a single datagram to multicast group 224.0.0.22 port 5010 (you do not need to join a multicast group to send to it). The contents of that datagram should be the text "`FROM`" (without the quotes), followed by a single space, your username, then a single newline `\n` character. Following the newline is the text passed as the command line argument, then another `\n` to signal end of message. You can retrieve your username using the `getlogin()` function.

Once your chirp client has sent the datagram, it should close the socket and exit.

`chirp_listener` The server should listen for datagrams sent to multicast group 224.0.0.22 on UDP port 5010. It should read each datagram received, and extract the username and message text. For each message, it should print the date and time, a dash, the username, another dash, then the message (e.g., if the received datagram contains "`FROM csp\nhello world\n`", print something like "`21-02-2012 22:49:05 - csp - hello world`").

Check that the contents of the message, and the username, contain printable characters before displaying them, in case a chirp client is sending badly formed datagrams (display a ? in place of any non-printable characters). Ensure your listener program is robust to receiving other malformed packets, and does not crash.

Test your `chirp` and `chirp_listener` programs, ensuring that you can chat with other members of the class.

Submission

You should prepare an electronic copy of your source code and Makefile (do not submit compiled binaries) archived as a `.tar.gz` file that expands into a directory named after your 7-digit matriculation number followed by "-submission2". For example, if your matriculation number is 0301234, your archive should expand to create a directory "0301234-submission2" with your files inside. You can create the archive using a command such as:

```
tar cvzf 0301234-submission2.tar.gz 0301234-submission2/
```

Ask one of the lab demonstrators if you are unsure how to create the archive. If your archive is formatted correctly, you should see something like the following when running the `tar ztf` command:

```
$ tar ztf 0301234-submission2.tar.gz
0301234-submission2/
0301234-submission2/Makefile
0301234-submission2/chirp.c
0301234-submission2/chirp_listener.c
$
```

(the `0301234-submission2/` prefix shows that the archive expands into a subdirectory with the appropriate name for this matriculation number).

This work is assessed, and is worth 4% of the marks for this course. Submissions should be made via Moodle. The deadline for submissions is 9:00am on Friday 9 March 2012. As per the Code of Assessment policy regarding late submissions, submissions will be accepted for up to 5 working days beyond this due date. Any late submissions will be marked as if submitted on time, yielding a band value between 0 and 22; for each working day the submission is late, the band value will be reduced by 2. Submissions received more than 5 working days after the due date will receive an H (band value of 0). Submissions that are not made via Moodle, or that are in archives which do not meet the above guidelines will be penalised two bands. This penalty will be applied in addition to any late submission penalty.