

CS 5523 Lecture 6: An overview of UDP and multicast

- *Questions on Laboratory 1*
- *UDP overview*
- *Common UDP applications*
- *UDP in Java*
- *UDP client-server communication in UNIX (simple-request)*
- *UDP client-server communication in UNIX (request-reply)*
- *UDP failure models*

UDP overview

- *abstraction is that of a datagram*
- *sender first creates a socket bound to (local port, local IP address)*
- *sender can use the socket to send to any destination*
- *destination (host, port) is explicitly included in each message*
- *the receiver must have a socket bound to the specified port*

What is the datagram abstraction?

Common Internet applications that use UDP:

- *Traceroute*
- *RIP (routing)*
- *BOOTP (bootstrap protocol)*
- *DHCP (bootstrap protocol)*
- *NTP (time protocol)*
- *TFTP (trivial FTP)*
- *SNMP (network management)*
- *DNS (name service)*
- *NFS (distributed file system)*
- *Sun RPC (remote procedure call)*
- *DCE RPC (remote procedure call)*

Figure 4.3 (updated from web site)
UDP client writes a request and (hopefully) reads a reply

```
import java.net.*;
import java.io.*;
public class UDPClient{
    public static void main(String args[]){
        DatagramSocket aSocket = null;
        try {
            aSocket = new DatagramSocket();
            byte [] m = args[0].getBytes();
            InetAddress aHost = InetAddress.getByName(args[1]);
            int serverPort = 6789;
            DatagramPacket request = new DatagramPacket(m,args[0].length(),
                aHost, serverPort);

            aSocket.send(request);
            byte[] buffer = new byte[1000];
            DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
            aSocket.receive(reply);
            System.out.println("Reply: " + new String(reply.getData()));
        }catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
        catch (IOException e){System.out.println("IO: " + e.getMessage());}
        finally {if(aSocket != null) aSocket.close();}
    }
}
```

Figure 4.4 (updated from web site)
UDP server repeatedly receives a request and sends it back to the client

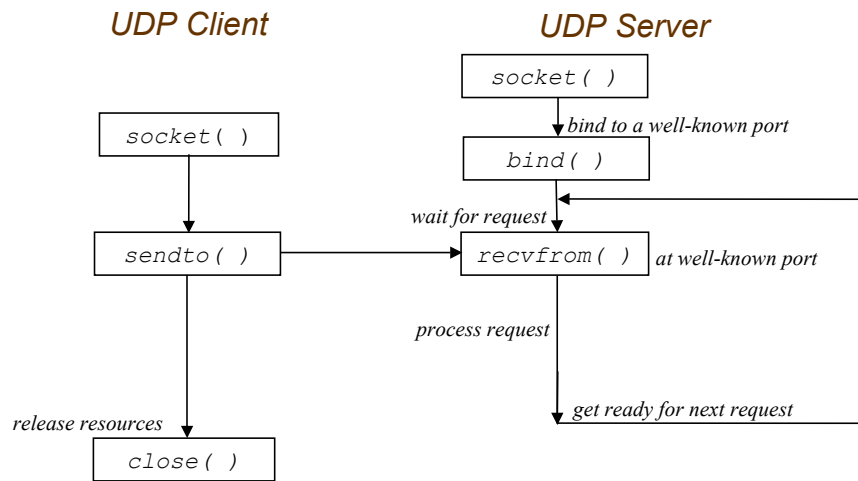
```
import java.net.*;
import java.io.*;
public class UDPServer{
    public static void main(String args[]){
        DatagramSocket aSocket = null;
        try{
            aSocket = new DatagramSocket(6789);
            byte[] buffer = new byte[1000]; // create socket at agreed port
            while(true){
                DatagramPacket request = new DatagramPacket(buffer, buffer.length);
                aSocket.receive(request);
                DatagramPacket reply = new DatagramPacket(request.getData(),
                    request.getLength(), request.getAddress(), request.getPort());
                aSocket.send(reply);
            }
        }catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
        }catch (IOException e) {System.out.println("IO: " + e.getMessage());}
        }finally {if(aSocket != null) aSocket.close();}
    }
}
```

UDP and Multicast in Unix

*The implementations that we are discussing are extracted from
UICI (Universal Internet Communication Interface)*

*from Practical Unix Programming by K. and S. Robbins
2nd Edition*

Basic UDP communication (simple request)



The client uses the server's well-known port.

Client in UDP communication - Unix

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "uici.h"
#include "uiciudp.h"
#define BUFSIZE 1024

int main(int argc, char *argv[])
{
    ssize_t nbytes;
    char request[BUFSIZE];
    int requestfd;
    int rlen;
    u_port_t serverport;

    if (argc != 3) {
        fprintf(stderr, "Usage: %s servername serverport\n", argv[0]);
        return 1;
    }
}
```

Client in UDP communication – Unix (continued)

```
serverport = (u_port_t) atoi(argv[2]);

/* Create an unbound UDP communication endpoint */
if ((requestfd = u_open_udp(0)) < 0) {
    fprintf(stderr, "Failed to create UDP endpoint: %s\n", u_strerror(requestfd));
    return 1;
}

/* Create a request containing the process ID */
sprintf(request, "[%ld]\n", (long)getpid());
rlen = strlen(request);

/* Use a simple-request protocol to send a request to (server, serverport) */
nbytes = u_sendtohost(requestfd, request, rlen, argv[1], serverport);
if (nbytes >= 0)
    return 0;
fprintf(stderr, "Send error: %s\n", u_strerror(nbytes));
return 1;
}
```

Opening a UDP port and binding it

```
int u_open_udp(u_port_t port)
{
    int reuse = 1;
    int sock;
    struct sockaddr_in server;

    if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
        return -1;
    if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &reuse, sizeof(reuse)) == -1) {
        close(sock);
        return U_ESOCKOPT;
    }

    if (port > 0) {
        server.sin_family = AF_INET;
        server.sin_addr.s_addr = htonl(INADDR_ANY);
        server.sin_port = htons((short)port);
        if (bind(sock, (struct sockaddr *)&server, sizeof(server)) < 0) {
            close(sock);
            return U_EBIND;
        }
    }
    return sock;
}
```

Sending to a specified host using UDP

```
ssize_t u_sendtohost(int fd, void *buf, size_t nbytes, char *hostn,
                    u_port_t port)
{
    struct hostent *hp;
    struct sockaddr_in remote;

    if (isdigit((int)(*hostn))) {
        remote.sin_addr.s_addr = inet_addr(hostn);
    }
    else {
        if (!(hp = gethostbyname(hostn)))
            return U_EHOST;
        memcpy((char *)&remote.sin_addr, hp->h_addr_list[0], hp->h_length);
    }
    remote.sin_port = htons((short)port);
    remote.sin_family = AF_INET;

    return u_sendto(fd, buf, nbytes, &remote);
}
```

Sending to a host (specified by name)

```
ssize_t u_sendtohost(int fd, void *buf, size_t nbytes, char *hostn,
                    u_port_t port)
{
    struct hostent *hp;
    struct sockaddr_in remote;

    if (isdigit((int)(*hostn))) {
        remote.sin_addr.s_addr = inet_addr(hostn);
    }
    else {
        if (!(hp = gethostbyname(hostn)))
            return U_EHOST;
        memcpy((char *)&remote.sin_addr, hp->h_addr_list[0], hp->h_length);
    }
    remote.sin_port = htons((short)port);
    remote.sin_family = AF_INET;

    return u_sendto(fd, buf, nbytes, &remote);
}
```

Sending to a host (specified by address)

```
ssize_t u_sendto(int fd, void *buf, size_t nbytes, u_buf_t *ubufp)
{
    int len;
    struct sockaddr *remotep;
    int retval;

    len = sizeof(struct sockaddr_in);
    remotep = (struct sockaddr *)ubufp;
    while ( ((retval = sendto(fd, buf, nbytes, 0, remotep, len)) == -1) &&
            (errno == EINTR) )
        ;
    return retval;
}
```

Server in UDP – Unix (simple request)

```
#include "uici.h"
#include "uiciudp.h"
#define BUFSIZE 1024

int main(int argc, char *argv[])
{
    char buf[BUFSIZE];
    char host_info[BUFSIZE];
    ssize_t nbytes_received;
    u_port_t port;
    int requestfd;
    u_buf_t sender_info;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s port\n", argv[0]);
        return 1;
    }
}
```

Server in UDP – Unix (continued)

```
/* Create a udp communication endpoint associated with port */
port = (u_port_t) atoi(argv[1]);
if ((requestfd = u_open_udp(port)) == -1) {
    fprintf(stderr, "Failed to create UDP endpoint: %s\n",
            u_strerror(requestfd));
    return 1;
}

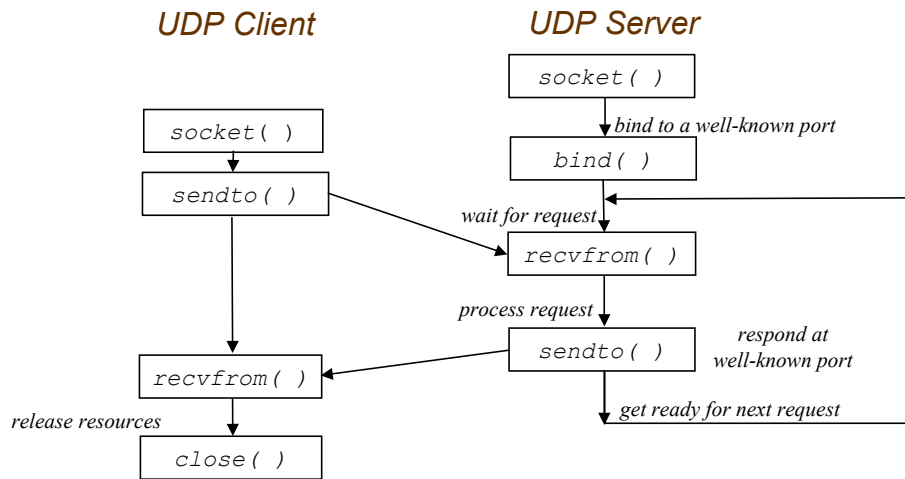
/* Process client requests */
for ( ; ; ) {
    nbytes_received = u_recvfrom(requestfd, buf, BUFSIZE, &sender_info);
    if (nbytes_received < 0) {
        perror("Server receive error");
        continue;
    }
    u_get_host_info(&sender_info, host_info, BUFSIZE);

    if ((u_write(STDOUT_FILENO, host_info, strlen(host_info)) == -1) ||
        (u_write(STDOUT_FILENO, buf, nbytes_received) == -1)) {
        perror("Error echoing to standard output");
    }
}
}
```

Summary of UDP properties:

- *Message size - determined by application up to 2^{16} bytes*
- *Blocking properties are determined by socket options not UDP*
- *Timeouts - can be set on a socket to prevent indefinite waiting*
- *The recvfrom doesn't specify which senders, anyone can send*

UDP communication model 2 (request-reply)



*The client uses the server's well-known port.
The server uses client address from request for response.*

UDP client using request-reply

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "uici.h"
#include "uiciudp.h"
#define BUFSIZE 1024

int request_reply(int requestfd, void* request, int reqlen,
                 char* server, int serverport, void *reply, int replen);

int main(int argc, char *argv[])
{
    ssize_t nbytes;
    char reply[BUFSIZE];
    char request[BUFSIZE];
    int requestfd;
    u_port_t serverport;

    if (argc != 3) {
        fprintf(stderr, "Usage: %s servername serverport\n", argv[0]);
        return 1;
    }
}
```

UDP client using request-reply (continued)

```
serverport = (u_port_t) atoi(argv[2]);

        /* Create an unbound UDP communication endpoint */
if ((requestfd = u_open_udp(0)) < 0) {
    fprintf(stderr, "Failed to create UDP endpoint: %s\n", u_strerror(requestfd));
    return 1;
}

sprintf(request, "[%ld]\n", (long)getpid()); /* Create request with process ID */

/* Use request-reply protocol to send a message */
nbytes = request_reply(requestfd, request, strlen(request)+1,
                       argv[1], serverport, reply, BUFSIZE);

if (nbytes < 0) {
    fprintf(stderr, "request_reply failed: %s\n", u_strerror(nbytes));
    return 1;
}

/* Echo the reply for debugging */
if (u_write(STDOUT_FILENO, reply, nbytes) == -1) {
    perror("Unable to echo server reply");
    return 1;
}
return 0;
}
```

Request-reply

```
int request_reply(int requestfd, void* request, int reqlen,
                 char* server, int serverport, void *reply, int replen)
{
    ssize_t nbytes;
    u_buf_t sender_info;

    /* Send the request */
    nbytes = u_sendtohost(requestfd, request, reqlen, server, serverport);
    if (nbytes < 0)
        return (int)nbytes;

    /* Wait a response, restart if from wrong server */
    while ((nbytes = u_rcvfrom(requestfd, reply, replen, &sender_info)) >= 0 )
        if (u_compare_host(&sender_info, server, serverport)) /* sender match */
            break;
    return (int)nbytes;
}
```

UDP failure model:

- *Omission failures*
- *Messages can be delivered out of order*

Does UDP have integrity?

Does UDP have validity?

Timelines for simple-request and request-reply

- *What are the possible scenarios?*
- *What can go wrong?*
- *How can they be fixed?*

Scenarios

■ Events:

- A – Client sends a request message*
- B – Server receives request message*
- C – Server processes the request*
- D – Server sends the reply message*
- E – Client receives reply message*
- F – Request message is lost*
- G – Reply messages is lost*
- H – Client crashes*
- I – Server crashes*

■ Which sequences are possible: *ABCED, ABCDG, ABCI, ABCGD?*

■ Draw the time lines associated with the event sequences

Strategies for dealing with UDP failures

- *Request-reply*
- *Timeouts*
- *Discarding duplicates*
- *Request-reply-acknowledge*

What types of failure to each of these address?

Why is repeating request-reply with time-outs a problem in UDP?

What is an idempotent operation?

Why is request-reply over TCP fundamentally different from UDP?

For next time:

- *Read CDK Chapter 4*
- *Be prepared to discuss the following questions:
3.1, 3.7 and 3.14*