

# CSIS0234B Computer and Communication Networks (Class B)

## Reading for Tutorial 3

### Datagram communications

So far we have used the network by having clients making connections to servers, using the reliable byte-stream semantic. When such a semantic is used, every message is replied, or “acknowledged”, by the underlying OS. The OS will retransmit messages if an acknowledgement is not seen. This is used to guarantee that messages always arrive at the remote host, in the right order.

Reliable semantic makes programming easy, but involves overhead that sometimes is unjustified. For example, when audio information is transferred through the network, it is usually better to miss some sound samples than to delay everything to wait for missed samples to get retransmitted. Also, sometimes the concept of a “connection” is completely meaningless, as we will see later in the course when we talk about broadcasting and multicasting.

In such cases, we want the “raw power” of the underlying network, with the OS adding only minimal overheads. They are called “best-effort” (or “unreliable”) **datagram** services. The TCP/IP protocol suite provides one such datagram service, called UDP (User Datagram Protocol). Similar to TCP, a number of system calls are provided for sending and receiving datagrams. Many of them are the same as those used for TCP, so please read this in conjunction with the tutorial readings for tutorial 1.

#### 1. Setting up a UDP socket

UDP sockets are created using the `socket()` system call, just like TCP sockets. The only difference is that `SOCK_DGRAM` is used, instead of `SOCK_STREAM`, as the `type` argument. As for TCP, you can set `protocol` to 0 to select the default UDP protocol. If you want to be explicit, set it to `IPPROTO_UDP` instead.

Just like TCP servers, UDP servers must have a “well-known port” for other computers to send messages to. This well-known port is established using `bind()`, as before. In general, clients do not need to bind to a particular address: the OS will choose an arbitrary unused address when an unbound socket is first used for communication.

#### 2. Sending and Receiving without connection

UDP allows messages to be sent and received without making a connection. Without a connection, one needs to specify the recipient address whenever a message is sent. This can be done by the `sendto()` system call, essentially combining the functionalities of `connect()` and `write()`. Similarly, the sender is known only when a message is received. This can be done by the `recvfrom()` system call, combining the functionalities of `accept()` and `read()`. The prototypes look as follows:

```
int sendto(int sockfd, const void *buf, size_t nbytes, int flags,
           const struct sockaddr *to, socklen_t tolen);
int recvfrom(int sockfd, void *buf, size_t nbytes, int flags,
             struct sockaddr *from, socklen_t *fromlen);
```

The arguments `sockfd`, `buf` and `nbytes` have exactly the same meaning as in `read` and `write`. Similarly, the `to` and `tolen` arguments have exactly the same meaning as the `serv_addr`

and `serv_addr` arguments in `connect()`; while the `from` and `fromlen` arguments have exactly the same meaning as the `addr` and `addrlen` arguments in `accept()`. Instead of repeating the tedious details, we just refer you to tutorial 1 readings and man pages.

The `flag` argument allows additional options to be set for one system call (like disallowing blocking, receiving data without removing the data, etc). We will seldom need any flag, and in such cases we should use 0 as `flag`.

### 3. UDP server

The skeleton of a UDP interactive server usually looks like this:

```
int sockfd = socket(PF_INET, SOCK_DGRAM, 0);
struct sockaddr_in servaddr;
/* fill in the local socket address structure for binding */
if (bind(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr)) == -1)
    /* handle error */
char msg[MAX_MSG_SIZE]; /* MAX_MSG_SIZE should be defined before */
char reply[MAX_REPLY_SIZE]; /* MAX_REPLY_SIZE should be defined before */
struct sockaddr_in client_addr;
sockaddr_t len;
for (;;) {
    len = sizeof(client_addr); /* size of client address */
    int num_recv_bytes = recvfrom(sockfd, msg, MSG_SIZE, 0,
        (struct sockaddr *)&client_addr, &len);
    if (num_recv_bytes == -1)
        /* handle error */
    /* Process the data */
    if (sendto(sockfd, reply, REPLY_SIZE, 0,
        (struct sockaddr *)&client_addr, len) == -1)
        /* handle error */
}
```

There is an important difference between this server from a typical TCP server. It never calls `accept()`, which is natural as the functionality is already in `recvfrom()`. It never calls `listen()`, since there is no “connections” to keep. On the other hand, many clients may send messages to the same UDP socket at the same time, and the OS needs to keep these message at some place when the server is doing something else. Each UDP socket has a receive buffer and each datagram that arrives for this socket is placed in that socket receive buffer. When the process calls `recvfrom()`, the next datagram from the buffer is returned to the process in a FIFO (first-in, first-out) order. This buffer has a size limit (in Linux, by default it is 65535 bytes). If a message is too long to fit in the receive buffer, excess bytes will be discarded. The size of the receive buffer can be increased by setting the socket option `SO_RCVBUF` with `setsockopt()`<sup>1</sup>:

```
int new_size = 12 * 1024; /* the new size of receive buffer */
setsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &new_size, sizeof(new_size));
```

---

<sup>1</sup>In linux, the value set is actually twice the value specified. This can be verified by calling `getsockopt()` immediately after calling `setsockopt()`. The reason is that Linux use the receive buffer to hold internal control information about the buffer, which is not the case for other Unix variants. Most programs do not expect this, so the OS artificially double the value set for `SO_RCVBUF`, accounting for the need for storing extra information.

#### 4. UDP clients

UDP clients is even simpler: it just need to send a message to the server and wait message from it. E.g.,

```
int sockfd = socket(PF_INET, SOCK_DGRAM, 0);
struct sockaddr_in servaddr;
/* fills in the socket address structure */
if (sendto(sockfd, mesg, strlen(mesg), 0, (struct sockaddr *)&servaddr,
          sizeof(servaddr)) == -1)
    /* handle error
int num_rcv_bytes = recvfrom(sockfd, reply, REPLY_SIZE, 0, NULL, NULL);
/* Process the reply */
```

Just like `accept`, one can set the address arguments of `recvfrom` to `NULL` if the program is not interested in knowing the address of the remote side, as we see here. However, this means that if another program somehow sends a datagram to this port, it will be incorrectly regarded as the reply from the server. To avoid that, one can actually get the address upon `recvfrom`, and compare it against the address in `servaddr`. Or one can use connected sockets as described below.

#### 5. Connections with UDP

All these are fine if UDP is used for a single interaction. But if a client needs to interact with the same server many times, it can be troublesome to have to specify `to_addr` and `from_addr` on every interaction. Furthermore, the corresponding servers are quite troublesome to write, since messages from different clients all appear from the same socket. One way to avoid this is to make a **new socket that only send and receive message with one peer**. This is achieved with the `connect()` system call, with exactly the same usage as the `connect()` call when used for TCP connections. The resulting socket is said to be **connected**. Typically, servers `fork()` before using `connect()`, so that the parent can continue listening. For connected sockets, one can use `read()` and `write()` for communication. On the other hand, use of `recvfrom()` and `sendto()` are restricted to cases where the address argument is set to `NULL` (“default”).

Unlike `connect()` in TCP sockets, here no “connection” is made with the other party. In fact even the existence of peer is not checked. The name “connect” probably makes it confusing. A name like “`setpeername()`” might be more appropriate, reflecting what is done by the OS: make sure the socket is used only to send and receive packet with one peer.

A client using `connect()` looks like this:

```
if (connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr)) == -1)
    /* handle error */
write(sockfd, mesg, strlen(mesg));
int num_rcv_bytes = read(sockfd, reply, REPLY_SIZE);
/* Process the reply */
```

#### 6. Size limitation of the UDP packets

The length of an UDP packets is bounded by a fixed size, depending on the underlying network. For IP networks this maximum length is about 64K bytes. If this limit is exceeded, the `sendto()` call fails immediately, with `errno` set to `EMSGSIZE`. Most UDP packets are much

smaller, so that they can travel on Ethernet without having to be fragmented (thus improving efficiency). For this the packet size should be less than 1400 bytes.

## 7. Error handling in connectionless services

Apart from errors that can be determined immediately, like the one we just see in the last section, in general errors are not detected during a sending operation. Thus the sender can immediately perform other operations, without waiting for the underlying network system to confirm that the operation is performed. But this also means that errors cannot be detected for a sending operation. E.g., if a UDP message is sent to a port which no program is listening, the remote side OS may generate a reply (technically, ICMP reply) informing the sender that nobody is listening. This cannot be detected during a send operation. We say such errors are “**asynchronous**”.

Instead, such errors may be detected during the next **receive** operation. When an asynchronous error occurs, the corresponding socket is found, and an error is queued in the socket. The user program using the socket is notified during the next operation (usually, receive) that it performs.

There is one subtlety, though. An unconnected UDP socket can be communicating with many peers, and there is no way to tell which peer has a problem. Consequently, asynchronous errors are simply discarded (rather than reported) for unconnected UDP sockets. Applications using unconnected sockets must thus deal with errors using other means. This is probably not a big deal, since many fire walls discard such error notification anyway.