

# CSIS0234B Computer and Communication Networks (Class B)

## Reading for Tutorial 3

### Datagram communications

In the last two tutorials we have examined how to make TCP connections in order to allow network communications. TCP communicates a “reliable byte-stream”: every message is replied (“acknowledged”) by the underlying OS. The OS will retransmit messages if an acknowledgment is not seen. This guarantees that messages always arrive at the remote host, in the right order, but involves some overhead.

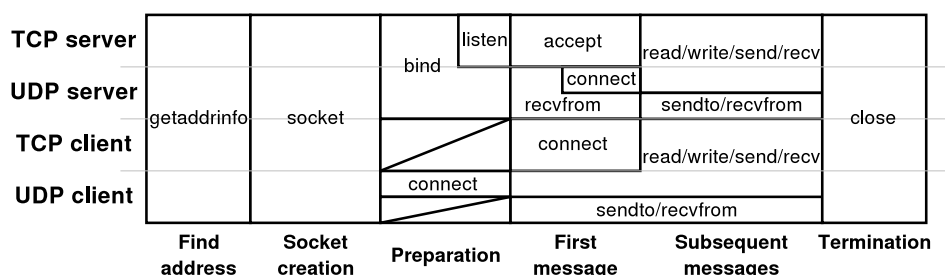
Such overhead is sometimes unjustified. For example, for real-time audio, it is usually better to simply omit corrupted sound samples than to retransmit them. Also, sometimes the concept of a “connection” is completely meaningless (we will see this later in multicasting). In such cases, we want the “raw power” of the underlying network, with the OS adding minimal overheads only. They are called “best-effort” (or “unreliable”) **datagram** services. The TCP/IP protocol suite provides UDP (User Datagram Protocol) for this purpose.

#### 1. Overview of Datagram communication

A number of system calls are provided for sending and receiving datagrams. Many of them are the same as those used for TCP, so you should read this note with the tutorial 1 notes. Like communicating using TCP, UDP communication usually involves a server and a client. The server must first get a socket and bind it to a port, so that the client can find it. Once the binding completes the port is ready for receiving messages, without getting a connection first. So the first message contains data and also the address of the client. Using the client address, the server can send data back to the client. Two system calls, `sendto()` and `recvfrom()`, are introduced for sending and receiving data with an address. In the reverse direction, a UDP client simply gets a UDP socket and start communicating with the server using `sendto()` and `recvfrom()`.

One can use the `connect()` system call for UDP sockets, both in the server side and the client side. Once a UDP socket is `connect()`'ed, one can use `read()` and `write()` to access the socket, as in TCP. We will discuss it further in Section 6 below.

The following summarizes the system/function calls for TCP and UDP communications.



#### 2. Getting addresses

A client can use `getaddrinfo()` to get an address structure of the server, and the server can use it to get an address structure suitable for `bind()`ing, just like TCP. The only difference is that we will use `SOCK_DGRAM` instead of `SOCK_STREAM` in the `ai_socktype` field of the `hints` argument. Once you get the `addrinfo` structure, you can create a socket and bind it or use it for `connect()` as if you are using TCP.

### 3. The `sendto()` and `recvfrom()` system calls

One can send a message to a particular port in a particular host using the `sendto()` system call, which combining the functionalities of `connect()` and `write()`. The receiver may receive a message from any port of any host using the `recvfrom()` system call, thus `recvfrom()` essentially combines the functionalities of `accept()` and `read()`. Their prototypes are 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 `servaddr` and `addrlen` arguments in `connect()`; while the `from` and `fromlen` arguments have exactly the same meaning as the `addr` and `addrlen_ptr` arguments in `accept()`.

Since no connections are maintained, the server needs to get the source address so as to send messages back to the clients. Therefore, instead of passing `NULL` in `from` and `fromlen`, the server must be prepared to receive the client address (at least for the first message of a client). Before a call to `recvfrom()`, `fromlen` should hold the number of bytes allocated for the `from` argument. On return, it is modified to indicate the actual size of the address stored in the structure. We can then use these information in the next `sendto()`. The `flag` argument allows additional options to be set for one system call (to never wait, to “peek” at the next message, etc). We normally don’t need them, so we pass 0 as `flag`.

There is a complication here: the `struct sockaddr` type is designed when IPv6 is not in anybody’s dream, so it is not large enough to hold an IPv6 address. So although `from` is said to be a pointer to `struct sockaddr`, it should not really points to a real `struct sockaddr` (otherwise your program will only work for IPv4 networks). One way to deal with the problem is to look at the size of your own address, and use it to allocate the peer address structure. Alternatively, one may use the `sockaddr_storage` type, which be large enough all protocols. For example, a simple server that sends and receives one byte messages would look like this:

```
struct sockaddr_storage client_addr;
char mesg, reply;
for (;;) {
    socklen_t len = sizeof(client_addr); /* size of client address */
    int num_recv_bytes = recvfrom(sockfd, &mesg, 1, 0,
                                (struct sockaddr *)&client_addr, &len);
    if (num_recv_bytes == -1) err(1, "recvfrom");
    /* Process the data */
    if (sendto(sockfd, &reply, 1, 0,
               (struct sockaddr *)&client_addr, len) == -1)
        err(1, "sendto");
}
```

### 4. Receive buffers

There is an important difference between a UDP server and a TCP server: it never calls

`listen()`, so there is no `backlog` argument to set how many outstanding connections you would like to keep. Of course, at a time when many clients send messages to the same UDP socket, the OS needs to keep these messages in some buffer. So **every UDP socket has a receive buffer**. When a UDP message arrives at a UDP socket, it is queued there, waiting for a `recvfrom()` to get the message in a first-in first-out order.

This buffer has a size limit. If a message is too long to fit in the receive buffer, the message will be discarded. Note that this is very different from TCP: in TCP, what is communicated is a long byte-stream, so if a message is too long it will be fragmented into pieces, and the receiver will get them in separated calls to `read()`. In UDP, a message that does not fit into the buffer is never fragmented. Instead it is simply discarded. Likewise, blocking (the reverse of fragmenting) can occur for TCP, but will never occur for UDP. In other words, multiple UDP messages remain to be multiple UDP messages, they will never become a single larger UDP message.

In Linux, the receive buffer has, by default, a size of 65535 bytes. This translates to the normal maximum size of an IP packet (if the size is larger, the `sendto()` call fails immediately, with `errno` set to `EMSGSIZE`). However, most UDP messages are much shorter, to avoid having to be fragmented when travelling on an Ethernet network. For this the message should be less than 1400 bytes. So the default receive buffer is normally sufficient. On the other hand, applications which expect many outstanding UDP messages at the same time may increase the size of the receive buffer by setting the socket level (`SOL_SOCKET`) option `SO_RCVBUF`<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));
```

## 5. Sessions involving multiple messages

A simple UDP server does the following for each client: receives a single message from a client, processes the message, and completes the transaction by sending a single message back to the client. This can be done iteratively, or using a concurrent server approach or pre-forking server approach if the processing involves some waiting.

But this involves a lot of work if the client-server interaction involves more than one pair of message exchanges: it requires the server to remember each client and their progress. When a message comes, it has to be compared against the known client addresses to see whether it is one of them. If so the server must continue with the dialog in progress; and if not the server must start a new dialog. The server must also maintain multiple timeouts to prevent failed clients from building up.

A simple alternative is to use a separate process for each client. Once the server receives a client message, it will fork out a new process. Then the child process would deal with the interaction. This “concurrent server” approach has a small problem, though: if all children call `recvfrom()` on the same socket, incoming messages will go to an arbitrary child of the server, which is usually not the child allocated to deal with the message.

The answer is simple: let each child of the server use a different port! So when the server receives a connection, it forks a child, creates a temporary socket, and uses the temporary sock-

---

<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.

et to reply to the client. The client-server interaction will use this temporary server port. Once the interaction completes (or timeouts), the child dies, and the temporary socket is destroyed. For this to work, the client must look at the peer address of the first message received (using `recvfrom()`), and send subsequent messages to that address instead of the well-known server address.

## 6. Connected UDP sockets

Apart from the first message that is sent by the client, and the first message that is received by the client in case of a UDP session involving multiple messages, all UDP messages involve a known peer. However, the `recvfrom()` call allows any host and port to send a message to your port, so you should really check the address to see whether it is the intended peer before processing the message. Similarly, you have to specify the address of the peer whenever you use the `sendto()` system call, even though you are always talking to the same peer in the same session. Both are extra work for the programmer, so the `connect()` system call can be used to avoid all these. After `connect()`, the socket becomes a “connected socket”: one can use `read()` to receive data, the OS will only forward messages coming from the correct peer. One can also use `write()` to send data without specifying the peer address once again. On the other hand, use of `recvfrom()` and `sendto()` are restricted to cases where the address argument is set to `NULL` (“default”).<sup>1</sup>

It should be emphasized that no “connection” is made: even the existence of the peer is not checked. So the name “connect” is somewhat misleading. 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 messages with one peer.<sup>2</sup>

## 7. Error handling in connectionless services

Apart from errors that can be determined immediately (like sending an oversized message), 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 in 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 firewalls discard such error notification anyway.

---

<sup>1</sup>However, this is not checked in Linux, so you can actually use a connected socket to send to any peer, although message from any host other than the intended one will be discarded.

<sup>2</sup>The reverse name, “getpeername”, is actually a system call: given a TCP socket or a connected UDP socket you can use it to find the address of the peer.