The User Datagram Protocol

Stefan D. Bruda

CS 454/564, Fall 2023

TCP IS HEAVYWEIGHT

"Hi, I'd like to hear a TCP joke."
"Hello, would you like to hear a TCP joke?"
"Yes, I'd like to hear a TCP joke."
"OK, I will tell you a TCP joke."
"Are you ready to hear a TCP joke?"
"Yes, I am ready to hear a TCP joke."
"OK, I am about to send the TCP joke. It will last 10 seconds, has 2 characters, it does not have a setting, it ends with a punchline."
"OK, I am ready to get the TCP joke that will last 10 seconds, has 2 characters, does not have a setting, and ends with a punchline."
"I'm sorry, your connection has been timed out."
"Hello, would you like to hear a TCP joke?"

TCP IS HEAVYWEIGHT

“Hi, I’d like to hear a TCP joke.”
“Hello, would you like to hear a TCP joke?”
“Yes, I’d like to hear a TCP joke.”
“OK, I will tell you a TCP joke.”
“Are you ready to hear a TCP joke?”
“Yes, I am ready to hear a TCP joke.”
“OK, I am about to send the TCP joke. It will last 10 seconds, has 2 characters, it does not have a setting, it ends with a punchline.”
“OK, I am ready to get the TCP joke that will last 10 seconds, has 2 characters, does not have a setting, and ends with a punchline.”
“I’m sorry, your connection has been timed out.”
“Hello, would you like to hear a TCP joke?”

- The actual layer that transports data between machines is IP, which is a packet-switching, best-effort (unreliable) protocol
- TCP adds significant overhead to ensure reliability
  - Four-way handshake, sequence numbers, checksums, acknowledgments and retransmissions

UDP

“I like telling UDP jokes because I don’t care if you don’t get them.”

- Very similar to the TCP in terms of API
- Dissimilar with TCP in terms of innards (and hence programming techniques)
  - Many-to-many communication. Unlike TCP (point-to-point communication), UDP allows wide flexibility in the number of applications that can communicate with each other
    - multicast and broadcast facility
  - Unreliable delivery. A message can arrive in duplicate, or not arrive at all
  - No flow control. When messages arrive faster than they can be consumed, they are dropped
  - Message paradigm. Unlike TCP (stream paradigm) UDP communication is based on individual messages (datagrams)
  - Less overhead. UDP algorithms are simpler and thus communication is faster
- Your (informed) choice: one cannot choose between the sharply different TCP and UDP without taking into consideration the requirements of the application protocol

UDP CLIENTS

- Algorithm similar with TCP:
  1. Obtain the IP address and port number of the server (unchanged)
  2. Allocate a socket
  3. Choose a port for communication (arbitrary, unused)
  4. Specify the server to which messages are to be sent
  5. Communicate with the server (application protocol, send and receive messages)
  6. Close the socket

UDP clients

- Algorithm similar with TCP:
  1. Obtain the IP address and port number of the server (unchanged)
  2. Allocate a socket
  3. Choose a port for communication (arbitrary, unused)
  4. Specify the server to which messages are to be sent
  5. Communicate with the server (application protocol, send and receive messages)
  6. Close the socket

- Socket allocation:
  - Need to specify the protocol family and the socket type (UDP)

    ```
    #include <sys/types.h>
    #include <sys/socket.h>
    int sd = socket(PF_INET, SOCK_DGRAM, 0);
    ```
  - We end up with a socket descriptor

Connect to a server

```
int connectUDP(const char* host, const unsigned short port) {
    struct hostent *hinfo;
    struct sockaddr_in sin;
    int sd;
    const int type = SOCK_DGRAM;
    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    hinfo = gethostbyname(host);
    if (hinfo == NULL)
        return err_host;
    memcpy(&sin.sin_addr, hinfo->h_addr, hinfo->h_length);
    sin.sin_port = (unsigned short)htons(port);
    sd = socket(PF_INET, type, 0);
    if ( sd < 0 )
        return err_sock;
    int rc = connect(sd, (struct sockaddr *)&sin, sizeof(sin));
    if (rc < 0) {
        close(sd);
        return err_connect;
    }
    return sd;
}
```

Connect to a server

```
int connectUDP(const char* host, const unsigned short port) {
    struct hostent *hinfo;
    struct sockaddr_in sin;
    int sd;
    const int type = SOCK_DGRAM;
    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    hinfo = gethostbyname(host);
    if (hinfo == NULL)
        return err_host;
    memcpy(&sin.sin_addr, hinfo->h_addr, hinfo->h_length);
    sin.sin_port = (unsigned short)htons(port);
    sd = socket(PF_INET, type, 0);
    if ( sd < 0 )
        return err_sock;
    int rc = connect(sd, (struct sockaddr *)&sin, sizeof(sin));
    if (rc < 0) {
        close(sd);
        return err_connect;
    }
    return sd;
}
```

Connected and unconnected UDP sockets

Client applications can use a UDP socket in connected and unconnected mode

- To enter connected mode, the client calls `connect` to specify the remote endpoint address
- To communicate using an unconnected socket we have to specify the remote endpoint address each time we send a message
- This is the only difference between connected and unconnected sockets
  - a call to `connect` does not initiate any packet exchange
  - it just stores the remote address for future use
  - even if the call succeeds there is no guarantee that the address is valid, that the server is up, or that the server is reachable

Communicate with the Server

- We assume hereby that we have a “connected” socket
- We then send data using `send` and receive responses using `recv`
- Each time we call send, UDP sends a single message to the server containing all the data to be sent
- There is no longer the case that we might receive the answer in pieces
- Each call to `recv` returns a complete message, we no longer need repeated calls
  - If the receiving buffer is large enough, we end up with our original message
  - If the message is too large for the buffer. . .

COMMUNICATE WITH THE SERVER

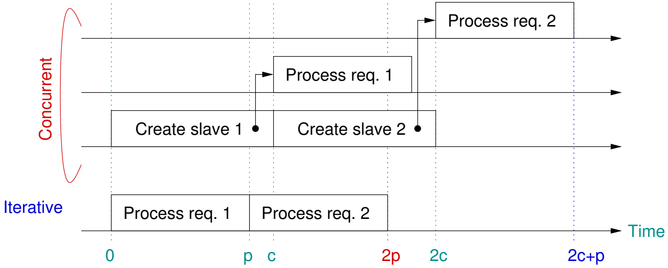
- We assume hereby that we have a “connected” socket
- We then send data using `send` and receive responses using `recv`
- Each time we call send, UDP sends a single message to the server containing all the data to be sent
- There is no longer the case that we might receive the answer in pieces
- Each call to `recv` returns a complete message, we no longer need repeated calls
  - If the receiving buffer is large enough, we end up with our original message
  - If the message is too large for the buffer. . .
    the bytes that cannot be stored are discarded without feedback

Closing the connection

- `close` closes the connection and destroys the socket
  - The machine on which the `close` occurs does not inform its peer about the event
  - The peer should be aware of this and know how long should it keep the data structures for the interaction with the client
- We can think of using `shutdown` to partially close the socket
  - Unfortunately, such a call is useless

- `close` closes the connection and destroys the socket
  - The machine on which the `close` occurs does not inform its peer about the event
  - The peer should be aware of this and know how long should it keep the data structures for the interaction with the client
- We can think of using `shutdown` to partially close the socket
  - Unfortunately, such a call is useless
  - The purpose of partial close was to inform the peer
  - UDP does not do this, the peer does not receive any indication of what happened on the other end (no end-of-file, no `SIGPIPE`)

- In principle, your UDP server is just your usual server
  - We can have concurrent or iterative servers
  - We can build our servers stateless or stateful
- In practice, many combination do not make a lot of sense
  - It is hard to argue for a stateful UDP server
  - Under UDP, it is often the case that process/thread creation is too expensive



  - Few UDP servers have concurrent implementations in practice

A UDP CONCURRENT SERVER

1. create and bind the master socket
2. leave the master socket unconnected
3. repeat forever:
    1. call `recvfrom` to receive the next request from a client
        - no call to `accept` is involved
        - no slave socket is created
    2. fork/create thread?
    3. (in child thread) do:
        1. form a reply according to the application protocol
        2. send the reply back to the client through the master socket using `sendto`
        3. terminate
    4. continue with the loop

- A concurrent implementation does not make a lot of sense
    - The child terminates after serving one request
    - About the only reason for concurrency is a time consuming step 3.3.1

Create and bind a UDP server socket

```
int passiveUDP(const unsigned short port, const int backlog,
               const unsigned long int ip_addr) {
    struct sockaddr_in sin;
    int   sd;
    const int type = SOCK_DGRAM;
    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = ip_addr;  // usually INADDR_ANY
    sin.sin_port = (unsigned short)htons(port);
    sd = socket(PF_INET, type, 0);
    if ( sd < 0 )
        return err_sock;
    if ( bind(sd, (struct sockaddr *)&sin, sizeof(sin)) < 0 )
        return err_bind;
    // if ( listen(sd, backlog) < 0 )
    //     return err_listen;
    return sd;
}
```

Unconnected sockets

- An unconnected socket does not store the coordinates of a peer
  - So the servers must use this kind of socket (they have more than one peer usually)
  - The clients may use unconnected sockets (especially when they communicate with more than one server)
- To send through an unconnected socket, we use

```
int sendto(int socket, const void *msg, size_t len, int flags,
           const struct sockaddr *to, socklen_t tolen)
```

- How do we obtain the address of the peer?

UNCONNECTED SOCKETS (CONT'D)

How do we obtain the address of the peer?

- We create it, as we did for the call to `bind`

Unconnected sockets (cont'd)

How do we obtain the address of the peer?

- We create it, as we did for the call to `bind`
- This is how we implement broadcast: we create a `sockaddr` structure containing the IP address `INADDR_BROADCAST`

UNCONNECTED SOCKETS (CONT'D)

How do we obtain the address of the peer?

- We create it, as we did for the call to `bind`
- This is how we implement broadcast: we create a `sockaddr` structure containing the IP address `INADDR_BROADCAST`
- When we send a reply, `recvfrom` gives us the reply address
- In addition to the buffer that holds the received message, a second buffer is filled in with the address of the sender

```
int recvfrom(int socket, void *buf, size_t len, int flags,
             struct sockaddr *from, socklen_t *fromlen);
```

- So we do something like this:

```
// other data (sd, request, rsize, etc.)
// declared and initialized as appropriate
struct sockaddr_in peer; socklen_t psize;
r = recvfrom(sd, request, rsize, 0,
             (struct sockaddr*)&peer, &psize);
// check r, prepare the buffer reply
sendto(sd, reply, strlen(reply), 0,
       (struct sockaddr*)&peer, psize);
```

On UDP unreliability

- Our client and server algorithms ignore one crucial aspect of UDP communication: unreliability
    - UDP communication semantics: unreliable, or best effort delivery
- Clients and servers must implement reliable communication all by themselves
    - We can use timeout and retransmission mechanisms
    - But then this introduces the problem of duplicate packets, which must also be handled
    - Adding reliability can be difficult, and is closely related to the semantics of the application protocol
- Reliability can be approached in two ways:
    - Ignore the problem. Do nothing, and so accept the possibility of dropped messages
    - Deal with the problem. Implement control algorithms using message sequencing, acknowledgments, timeouts and retransmissions
        - We thus end up with another implementation of TCP (so we would be better off using TCP in the first place)

TO USE OR NOT TO USE

- One should in principle prefer TCP
  - Useful features already implemented: reliability, point-to-point, flow control
  - Less burden on the application programmer
  - In fact most Internet services use TCP precisely for these reasons
- Major reasons for not using TCP: speed and bandwidth
  - TCP introduces a significant communication overhead (in terms of both bandwidth and time)
  - Some applications do not tolerate this overhead
    - Typical examples: games, real-time video streaming, VOIP
  - This kind of applications will typically use UDP
  - They usually use the “do not care” approach to reliability!
    - We do not care about a frame dropped now and then; it is more important that say, the video stream is delivered in real time
    - If we start implementing reliable communication we introduce the same kind of overhead we had problems with in the first place

To use or not to use (cont'd)

- Another reason for using UDP: broadcast/multicast capabilities
  - Good example: the DHCP protocol
  - A machine can obtain its IP address and other routing information automatically from a DHCP server
  - However, the machine cannot contact the server directly
    - It has no idea how to send packets to a precise destination at all!
    - Indeed, it does not even know its IP address
  - DHCP is thus a UDP application
    - The client broadcasts blindly a "discovery packet" (impossible under TCP)
    - The quickest DHCP server within reach responds with the IP address, routing information, etc. . . with a broadcast message of its own

- The concept of multiservice servers extends of course to UDP servers
- The same idea as for TCP: multiple threads listen to multiple ports and serve different types of clients
  - The difference is that there are no slave UDP threads
- In addition, it is often the case in practice that we have multiprotocol servers
  - That is, servers that accept both TCP and UDP clients
  - Typically, such a server serves the same kind of requests arriving on both TCP and UDP ports