UDP Lecture Notes

Matt Warner

1 Overview

- Transport Layer OSI model
- User datagram protocol (UDP)
- UDP progamming
- Example UDP client/server programs

2 Review: Network Layer

The network layer, also called the Internet Protocol (IP) layer, provides host to host transmission service, where hosts are not necessarily adjacent.

The IP layer provides services:

- *Addressing*, were hosts have global addresses: IPv4, IPv6
- *Routing* and *forwarding*, which involves finding a path from host to host.

3 Transport Layer

The internet has two main protocols in the transport layer, a connectionless protocol and a connection-oriented one. These protocols complement each other.

The connectionless protocol is UDP. It does almost nothing beyond sending packets between applications, letting applications build their own protocols on top as needed.

The connection-oriented one is TCP. It does almost everything. It makes connections and adds reliability with retransmissions, along with flow control and congestion control, all on behalf of the applications that use it.

4 UDP

The internet protocol suite supports a connectionless transport protocol called **UDP (User Datagram Protocol)**. UDP provides a way for applications to send encapsulated IP datagrams without having to establish a connection.

UDP transmits **segments** consisting of an 8-byte header followed by the payload. The header is shown in Fig. 6-27. The two **ports** serve to identify the endpoints within the source and destination machines. When a UDP packet arrives, its payload is handed to the process attached to the destination port. This attachment occurs when the BIND primative or something similar is used.

Think of ports as mailboxes that applications can rent to receive packets. We will have more to say about them when we describe TCP, which also uses ports. In fact, the main value of UDP over just using raw IP is the addition of the source and destination ports. Without the port fields, the transport layer would not know what to do with each incoming packet. With them, it delivers the embedded segment to the correct application.

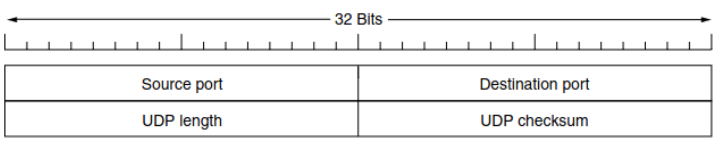


Figure 6-27. The UDP header.

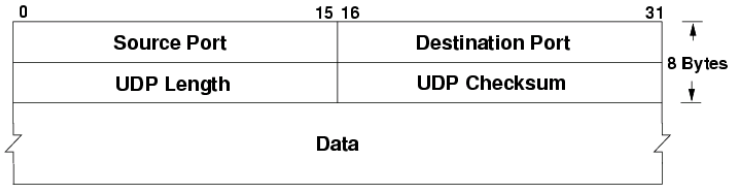


Figure 6-28. UDP Packet

Here are some things to consider with UDP:

- Data may or may not be delivered.
- Data may change
- Data may not be in order
- UDP uses checksums to discover if the data has been corrupted, but will not do anything about it.
- Transmits information in one direction from source to destination without verifying the readiness or state of the receiver.

5 UDP programming

The endpoint on a host in programmingc an be refered to as a socket.

The *Socket* is the end-point of the communications link. It has a clear identification. That is:

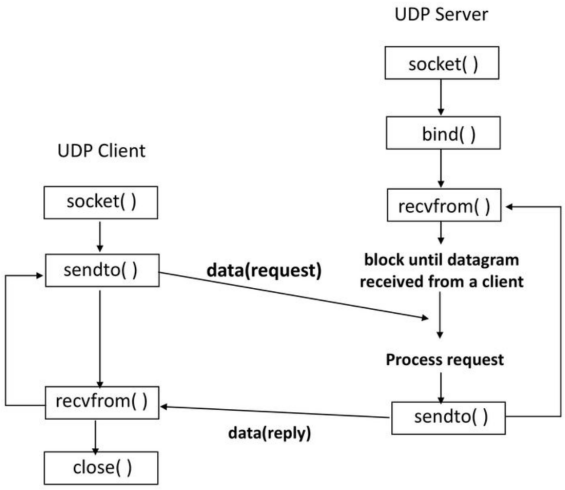
IP address + port number

The socket can receive data, and it can send data. The typical logic is that we have two programs. One is the server, and the other is the client. The server waits to receive a datagram from any client. The client puts on its address the IP & Port it should go to. The server receives it, processes it, then responds to the client with a datagram.

6 Socket system calls

`System call`	**Meaning**
`socket`	Create a new communication endpoint
`bind`	Attach a local address to a socket
`recvfrom`	Receive(read) some data over the connection
`sendto`	Send(write) some data over the connection
`close`	Release the connection

7 UDP communications pattern



8 System call: socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

socket() creates an endpoint for communication and returns a descriptor.

- **domain** is set to **AF_INET**
- **type** is set to **SOCK_DGRAM** for datagrams
- **protocol** is set to 0, i.e. default UDP
- Returns socket descriptor
    - used in **bind**, **sendto**, **recvfrom**, **close**.

9 system call: bind

Synopsis

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

- Assigns address to socket: IP number and port
- **struct sockaddr** holds address information
    - will accept **struct sockaddr_in** pointer
- **addrlen** specifies length of **addr** structure
- returns 0 on success, -1 otherwise

10 structure sockaddr: 16bytes

```c
struct sockaddr {
  short sa_family   /* address family */
  char sa_data[14];   /* address data */
};
struct sockaddr_in {
  short sin_family;    /* address family */
  unsigned short sin_port;    /* port number: 2 bytes */
  struct in_addr sin_addr     /* IP address: 4 bytes */
  char sin_zero[8]; /* pad to size of struct sockaddr */
};
```

11 structure sockaddr_in: members

```
// always AF_INET
sin_family    /* address family */

// htons(4444) ensures network order
sin_port  /* port number: 2 bytes */

sin_addr    /* Internet address: 4 bytes */

// 2 ways to construct sin_addr:
s_addr = INADDR_ANY /* Anything */
s_addr = inet_addr("127.0.0.1")
```

12 Helper Functions: Byte Order Conversion

- **htonl** (host to network long)
  - **Purpose:** Converts a 32-bit long integer from the host's byte order to network byte order.
  - **Details:** On little-endian systems, this function will reverse the byte order. On big-endian systems, it typically does nothing.
- **htons** (host to network short)
  - **Purpose:** Converts a 16-bit short integer from the host's byte order to network byte order.
  - **Details:** Adjusts the byte order of short integers for consistent data transmission across different systems.
- **ntohl** (network to host long)
  - **Purpose:** Converts a 32-bit long integer from network byte order to the host's byte order.
  - **Details:** Reverses the byte order of long integers received from the network to match the host system's byte order.
- **ntohs** (network to host short)
  - **Purpose:** Converts a 16-bit short integer from network byte order to the host's byte order.
  - **Details:** Reverses the byte order of 16-bit short integers received over a network to match the host system's byte order.

13 Helper Functions: Address Manipulation

- in_addr_t inet_addr(const char *cp)

14 System call: recvfrom

```
ssize_t recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from,
↪  socklen_t *fromlen)
```

Once you have a socket up and connected, you can read incoming data from the remote side using **recvfrom()** (for UDP **SOCK_DGRAM** sockets).

This function takes the socket descriptor, **s**, a pointer to the buffer **buf**, the size (in bytes) of the buffer, **len**, and a set of **flags** that control how the function works.

Additionally, the function takes a **struct sockaddr***, that will tell you where the data came from, and will fill in **fromlen** with the size of **struct sockaddr**.

15 System call: sendto

```
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags, const struct sockaddr
↪  *desk_addr, socklen_t addrlen)
```

- sends datagram **buf** of size **len** to socket **sockfd**
  will wait if there is no ready receiverflags specifies wait behavior, e.g.: 0 for default
- **dest_addr** holds address information of receiver
  - **struct sockaddr defines address structure**
  - **addrlen** specifies length of **dest_addr** structure
- returns the number of bytes send, i.e. size of datagram

16 System Call: close

```
int close(int fd)
```

- closes socket specified by socket descriptor, **fd**

17 Example: UDP Programming

- simple server: echo
  - receives datagrams, sends them back to sender
- simple client
  - sends datagram to server, receives responce