UDP and RAW Sockets

Alessandro Barenghi

Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano

barenghi - at - elet.polimi.it

May 20, 2014

Lesson contents

Overview

- Datagram oriented protocol (UDP over IPv4)
- Communication over IPv6
- Raw UDP packet construction
- ICMP ECHO responder, from scratch

UDP

Overview

- The User Datagram Protocol (UDP): connectionless protocol, no “session” concept
- The transferred data unit is the `datagram`
- No automatic retransmission in case of data loss, nor proper packet reordering at the receiving endpoint
- Known, as a jest, as the Unreliable Datagram Protocol

UDP

Applications

- Low latency communications: VoIP, Video Streaming, NTP Protocol
- Packet Broadcast: single message sent to all clients of a network
- Single packet query-answer: DNS, DHCP, SNMP and RIP
- Resource constrained environments: Trivial FTP Protocol

UDP

Primitives

- Socket creation is still managed by the same `socket` primitive
- The only change is the use of `SOCK_DGRAM` as socket type
- UDP is the result of the combination of `SOCK_DGRAM` with a `AF_INET` or `AF_INET6` domain
- The binding to a socket in order to listen from it is still done via a regular `bind` call
- No need for `connect` or `listen` calls as there is no connection

UDP

Sending

Sending data on a UDP socket is managed by the `sendto` primitive :

```
sendto( int sockfd, const void *buf, size_t len,
    int flags, const struct sockaddr *dest_addr,
    socklen_t addrlen);
```

- Same first 3 parameters as the `send` primitive
- The `dest_addr` parameter specifies the destination since no concept of “session” is bound to the socket

UDP

Receiving

Analogously, the data are received via the `recvfrom` primitive :

```
recvfrom(int sockfd, void *buf, size_t len, int flags,
     struct sockaddr *src_addr, socklen_t *addrlen);
```

- Same first 3 parameters as the `recv` primitive
- The `src_addr` parameter specifies source address of the datagram to be read
- Since there is no congestion control, messages from a sender can be cluttered by the remaining unread traffic

UDP

Issues

- Collisions among clients : different clients with the same ephemeral source port may clash
- Data loss / reordering : due to network latencies, some packets sent before may be delivered too late
- Simple fix: introducing a trivial acknowledgement mechanism
- If the acknowledgedment mechanism is too simple it may lead to Sorcerers's Apprentice Syndrome

UDP

TFTP

- The Trivial File Transfer Protocol is an UDP based file transfer protocol
- In order to provide minimal transfer warranties, it implements a simple acknowledgement mechanism:
    - The client sends a Read/Write ReQuest (RRQ/WRQ) to the server port 69 to initiate the communication
    - The server answers with the first data packet to the RRQ or with an ACK to a WRQ from a fresh ephemeral port
    - The client sends a numbered ACK in case of a RRQ session or the first data packet to be written in case of an WRQ
    - The server sends a numbered ACK for the first data to be written or the second data packet after receiving the client packet

UDP

TFTP

- The TFTP protocol looks reasonably sound :
  - All the data packets are ACKnowledged upon reception
  - Packet $n + 1$ cannot be received if packet n has not been acknowledged
  - The server side ephemeral port is freshly allocated by the server (no collisions)
- It is in fact widely used for transferring the kernel of a system performing a network based boot
- So, why do we still use TCP based FTP for reliable transfers? Can you see the flaw?

UDP

Sorcerer's Apprentice Syndrome

- The reception ordering invariant mandates that packet $n + 1$ cannot be received if packet n has not been acknowledged....
- ... but there's no rule for duplicates!
- What if :
  - The server sends the n-th data packet
  - The client sends the n-th acknowledgement, which gets delayed by network issues
  - The server times out and re-sends the n-th data packet
  - The client re-acknowledges the reception....
- Two duplicated "data streams" are created from a single one
- UDP has no congestion control so the situation is bound to get worse

IPv6

Motivation

- The 4th version of the Internet Protocol (IPv4) was conceived standardized in 1981
- At the time , 32 bits for the unique host identifier were thought to be more than sufficient
- Similarly, no mechanism for automatic address assignment was conceived, and DHCP was later employed to compensate the lack of it
- The protocol was so well designed that it exceeded all its usefulness expectations, until....

IPv6

Motivation

- The IPv4 address space was completely assigned (last /24 subclass assigned months ago)
- The prime solution to this problem is represented by IPv6, which is being pushed into adoption
- The Linux kernel has a stable and well tested IPv6 suite integrated and all the API are already in place
- It is sufficient to switch the type of protocol of a common `AF_INET` socket and set the addresses accordingly to get an IPv4 program working on IPv6

Raw Sockets

Motivation

- Raw access to sockets allows full freedom in crafting any kind of packet
- Useful for debugging purposes and testing corner cases
- Useful to implement a subset of a defined protocol in constrained environments
- Useful to check the correctness of packet filtering and mangling tools

Raw Sockets

Overview

- Raw sockets are just common sockets, employed while disabling any further processing by the kernel
- The data sent into a raw socket receives only Level 2 incapsulation and is then sent on Level 1
- Due to the intrinsic flexibility of this mechanisms (filtering policy overriding, IP spoofing...) only `root` is allowed to use them
- Since the portability of these sockets is an issue, it is strongly advised to use fixed length and endianness data types from C99

Raw Sockets

How to use them

- The socket is initialized with the `SOCK_RAW` socket type macro
- After the initialization, the kernel is notified not to rebuild the IP header via setting the `IP_HDRINCL` via `setsockopt`
- The packet is then crafted by hand by the developer
- It is recommended also to correctly compute the header checksum, even though the packet will be sent anyway

Raw Sockets

IP Header

- Reconstructing the IP header allows the spoofing of any field of the header, source address included
- Care should be taken to set correctly the three IP flags (Reserved, DF and MF) since they are bit-packed before the fragmentation offset
- The checksum of the packet must be computed after the whole packet has been put together

Raw Sockets

UDP Header

- In order to have a first example of packet crafting, we will be building an UDP packet
- The UDP header imposes only minimal overhead over the common TCP header
- Moreover, the checksum field is allowed to be set to zero (except if the Level 3 protocol is IPv6)