Networking

UDP: User Datagram Protocol

- RFC 768 [Postel 1980]: about three pages.
- provides no reliability
  - it sends the datagram to the IP layer, but there is no guarantee that:
    - it will reach its destination
    - it will reach unspoiled its destination

UDP encapsulation

IP datagram

UDP datagram

UDP - checksum

pseudo header*

IP: Internet Protocol
**TFTP**

*Trivial File Transfer protocol*

- uses UDP as its transport mechanism
- mainly used to bootstrap diskless systems
- RFC 1350 [Sollins 1992] is the official spec.
  - RFC 2347, 2348, 2349 specify newer extensions.
- lock-step protocol

---

**format**

<table>
<thead>
<tr>
<th>IP header</th>
<th>UDP header</th>
<th>TFTP message</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode (2 bytes)</td>
<td>null terminated string</td>
<td></td>
</tr>
<tr>
<td>read 01</td>
<td>filename 2 bytes</td>
<td>octet: binary/raw ascii: convert nl to cr/nl</td>
</tr>
<tr>
<td>write 02</td>
<td>mode 1 byte</td>
<td></td>
</tr>
<tr>
<td>data 03</td>
<td>data 0 to 512 bytes</td>
<td></td>
</tr>
<tr>
<td>ack 04</td>
<td>path 2 bytes</td>
<td></td>
</tr>
<tr>
<td>error 05</td>
<td>null terminated message</td>
<td></td>
</tr>
</tbody>
</table>

---

**the protocol**

Client

- read request
- octet 0
- data
- ack 04
- data 01
- data
- block
- null terminated string
- block size
- block
- time out

Server

- data 01
- data
- block
- null terminated string
- block size
- block
- time out

---

**TFTP ...**

- is a stop and wait protocol
- each data-block has a block number
  - used in the acknowledge response
- lost packets are detected with timeout and retransmission implemented on the sender side.
- has no checksum / data integrity check
  - handled by the UDP layer
- has no security

---

**why are protocols so difficult?**

RRQ

- data 1

ACK 1

- data 2

ACK 2

- data 2

ACK 2

- data 3

The sorcerer's apprentice syndrome

---

**the Fix**

- ignore duplicate ACKs
tftp extensions

<table>
<thead>
<tr>
<th>IP header</th>
<th>UDP header</th>
<th>TFTP message</th>
</tr>
</thead>
</table>

opcode (2 bytes)

filename | mode | offset | ulen | opc1 | opc2 | ulen2 |

= ACK | option1 | value1 | option2 | value2 |

DNS
The Domain Name System

- Server
  - manage a distributed data base
  - process queries/requests
- Client:
  - does queries
  - uses the resolver library functions
  - ie: gethostbyname(...), gethostbyaddr(...)
format ...

- **identification**: set by the client and returned by the server.
- **flags**:

```
QR opcode IA PC RD RA MBZ Rcode
```

DNS - Summary

- essential when host is connected to the internet.
- hierarchical tree that forms the DNS name space.
- all DNS queries and responses have the same message format.

Clients & Servers

- **Client**:
  - in general, an application that initiates a peer-to-peer communication.
  - usually invoked by the 'end user'

- **Server**:
  - waits for incoming requests from a client.
  - performs necessary work and
  - probably returns a result.

Concurrent Vs. Iterative

- **concurrent-server**
  - handles multiple requests at one time.
- **iterative-server**
  - process one request at a time.
Connection [oriented|less]

- connectionless:
  - UDP - User Datagram Protocol
  - the burden of the data integrity is on the application.
- connection-oriented:
  - TCP - Transport Control Protocol
  - the application is free to deal with higher things.

Server types

- iterative, connectionless
  - the most common
    - usually stateless
    - trivial amount of processing
- iterative, connection-oriented
  - less common
    - trivial amount of data but
    - need reliable transport

TCP - Transmission Control Protocol

- connection oriented
  - exactly two end points.
    - no broadcast/multicast
  - the two applications must establish a connection with each other before data can be exchanged.
- reliable
  - byte stream
    - 8-bit bytes with no interpretation
    - there is no record boundaries.
reliable ...

- preserves sequence
  - IP datagrams can arrive out of order
  - segments are resequenced if necessary
- drops duplicates
  - since IP datagrams can get duplicated
- flow control
  - each end of the connection has a finite amount of buffer space.
  - the receiving side allows the other end to send as much data as it has buffer for.

TCP encapsulation

<table>
<thead>
<tr>
<th>IP datagram</th>
<th>TCP segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP header</td>
<td>TCP header</td>
</tr>
<tr>
<td>TCP data</td>
<td></td>
</tr>
</tbody>
</table>

TCP Header

<table>
<thead>
<tr>
<th>16-bit source port number</th>
<th>16-bit destination port number</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit sequence number</td>
<td>32-bit acknowledgment number</td>
</tr>
<tr>
<td></td>
<td>4-bit header length</td>
</tr>
<tr>
<td>6-bit flags</td>
<td>16-bit window size</td>
</tr>
<tr>
<td>16-bit TCP checksum</td>
<td>16-bit urgent pointer</td>
</tr>
<tr>
<td>options (if any)</td>
<td>data (if any)</td>
</tr>
</tbody>
</table>

TCP Header ...

- each segment contains a source and destination port number.
- together with the source and destination IP number from the IP header we get an unique identification of each connection.
- socket: IP address + port number
- socket pair: source + destination sockets.

connection establishment

1. the client dials a #
2. the server answers, Hello?
3. who's calling?
Connection Establishment
the three way handshake

1. the client sends a SYN segment specifying the port # of the server it wants to connect to, and its ISN - Initial Sequence Number
2. the server responds with its own SYN segment containing its ISN. The server also ACKs the client's SYN by ACKing the client's ISN+1
3. the client must ACK this SYN from the server by ACKing the server's ISN+1.

---

Segments, Streams and Sequence numbers

ISN + 2
last byte successfully sent acknowledged

-----
data stream

ready to be sent
sent but not acked
last byte that can be sent before an ack is received

---

TCP - Interactive data flow

client

keystroke

server

data byte

echo of data byte

ack of data byte

display

server

echo

ack of echoed byte

ack of echoed byte

ack of echoed byte