Computer Networks

Redes de Computadores

Bruno Carvalho
up201606517

João Malheiro
up201605926

Carlos Daniel Gomes
up201603404

December 23, 2018

Contents

Summary

This report was developed for the curricular course RCOM (Computer Networks), and contextualizes the work carried out in the last practical classes. We implement a simple FTP client and configure a private network of computers, a switch and a router.

Introduction

The first part of the project consists in implementing an FTP client application called *`download`*, capable of downloading a single file from an FTP server given an FTP URL. In this report we describe its architecture while also analyzing and discussing results.

The second part involves configuring a working private network of computers, connecting it to the Internet, and testing the FTP download application. At every step we will analyze the saved logs and describe the network architecture, while answering the presented questions.

Part 1: Download application

There are several auxiliary logging functions that are called to keep track of the download process and identify a point of failure if there is one.

Other than that, the whole application is built as a pipeline, without ramification. Errors found abort the program immediately, closing the connection if already open.

The download sequence is comprised of 7 steps:

1. Parse input FTP URL
2. Resolve hostname to server's IPv4 IP address and open protocol socket
3. Login to the server (user + password)
4. Enter passive mode and open passive socket
5. Retrieve command for file
6. Download file
7. Close connection to server

The URL is parsed by a regular expression which matches valid URLs with the *ftp* protocol, no port, and non-empty file paths — function *`parse_url()`*. Afterwards, the URL's hostname is resolved and the protocol connection is established — function *`ftp_open_control_socket()`*. After acknowledging the server's welcome response, we login with the provided or defaulted username as passwords — *`ftp_login()`*. Then we open a passive connection with a `PASV` command — *`ftp_open_passive_socket()`* — retrieve the requested file with `RETR` — *`send_retrieve()`* — and read the file from the socket connection until the end — *`download_file()`*.

An example log of the download of file `ftp://ftp.up.pt/pub/debian/README` can be found in Figure 5.

Part 2: Network

Exp 1. Configure an IP Network

We start by creating a private network 172.16.30.0/24 for computers *tux31* and *tux34*, connecting directly their interfaces `tux31@eth0` and `tux34@eth0`. Interface `tux31@eth0` is given IP address 172.16.30.1/24, and `tux34@eth0` is given 172.16.30.254/24. We test the connectivity of *tux31* and *tux34* with a simple `ping` command.

Cable configuration:
`tux31@eth0`—`tux34@eth0`

Commands:
tux31:
```
ifconfig eth0 up
ifconfig eth0 172.16.30.1/24
route add default gw 172.16.30.254
```
tux34:
```
ifconfig eth0 up
ifconfig eth0 172.16.30.254/24
```

What are the ARP packets and what are they used for?
ARP request packets are broadcast in a network by a host to retrieve the MAC address of the machine with a certain IPv4 address. The protocol specifies the packet is sent to every host machine in the network, and only the one with the requested IPv4 address responds with its MAC address. In other words, the ARP protocol serves to convert 32-bit logical IP addresses to 48-bit physical MAC addresses.

What are the MAC and IP addresses of ARP packets and why? The ARP request packet includes the IP and MAC addresses of its source host and the IP address of its target host, whose MAC address it wants to retrieve. The ARP reply includes all four addresses. For example, *tux31* sent an ARP request for 172.16.30.254 on network 172.16.30.0/24, which reached `tux34@eth0` (the only other host on the network). Its ARP reply was MAC address `00:21:5a:5a:7d:7a`.

What packets does the ping command generate? The `ping` commands generates ICMP `ECHO_REQUEST` packets, and expects ICMP `ECHO_REPLY` responses.

What are the MAC and IP addresses of the ping packets? The source IP is 172.16.30.1 (`tux31@eth0`) and the destination IP is 172.16.30.254 (`tux34@eth0`) for the `ECHO_REQUEST` packets, and the reverse of `ECHO_REPLY` packets. Since the packets are directly transmitted, the source and destination MAC addresses match the IP addresses.

How to determine if a receiving Ethernet frame is ARP, IP, ICMP? For Ethernet frames, check the EtherType header field on the MAC frame: `0x0800` for IPv4 and `0x0806` for ARP. Inside the IP packet, the protocol header field indicates the upper layer protocol: 1 for ICMP, 6 for TCP, 17 for UDP, etc.

00:0f:fe:8b:e4…	ff:ff:ff:ff:ff:ff	ARP	Who has 172.16.30.254? Tell 172.16.30.1
00:21:5a:5a:7d…	00:0f:fe:8b:e4:4d	ARP	172.16.30.254 is at 00:21:5a:5a:7d:74
172.16.30.1	172.16.30.254	ICMP	Echo (ping) request id=0x39cc, seq=1/256, ttl=64
172.16.30.254	172.16.30.1	ICMP	Echo (ping) reply id=0x39cc, seq=1/256, ttl=64
172.16.30.1	172.16.30.254	ICMP	Echo (ping) request id=0x39cc, seq=2/512, ttl=64
172.16.30.254	172.16.30.1	ICMP	Echo (ping) reply id=0x39cc, seq=2/512, ttl=64
172.16.30.1	172.16.30.254	ICMP	Echo (ping) request id=0x39cc, seq=3/768, ttl=64
172.16.30.254	172.16.30.1	ICMP	Echo (ping) reply id=0x39cc, seq=3/768, ttl=64
172.16.30.1	172.16.30.254	ICMP	Echo (ping) request id=0x39cc, seq=4/1024, ttl=64
172.16.30.254	172.16.30.1	ICMP	Echo (ping) reply id=0x39cc, seq=4/1024, ttl=64
172.16.30.1	172.16.30.254	ICMP	Echo (ping) request id=0x39cc, seq=5/1280, ttl=64
172.16.30.254	172.16.30.1	ICMP	Echo (ping) reply id=0x39cc, seq=5/1280, ttl=64

Figure 1: *tux31* pings *tux34*

How to determine the length of a receiving frame? For Ethernet frames there is no header field with the frame length — the whole frame must be read and the measured. For IPv4 packets, the header has two length fields: one for header length (4bits, offset 4 bits) and another for packet length (2 bytes, offset 16 bits).

What is the loopback interface and why is it important? 172.0.0.0/8 is a range of 2^{24} IPv4 addresses representing the host machine. Useful for testing or running client-server services in the host. It is generally not represented in the routing table. The address 172.0.0.1 is assigned the local loopback interface `lo`, but the whole range is private and may be used.

Exp 2. Implement two virtual LANs in a switch

Now we are going to create a second private network 172.16.31.0/24 for *tux32*, and connect our two networks to two virtual LANs in the switch: vlan 30 for network 172.16.30.0/24 and vlan 31 for network 172.16.31.0/24. Naturally, we connect `tux31@eth0` and `tux34@eth0` to vlan 30 and `tux32@eth0` to vlan 31. Notice there is no connection between the networks yet, so *tux32* cannot communicate with neither *tux31* or *tux34*.

How many broadcast domains are there? How can you conclude it from the logs? The two broadcast domains are the two networks 172.16.30.0/24 and 172.16.31.0/24 because they are not connected.

Cable configuration:
`tux31@eth0—sw Fa0/1`
`tux34@eth0—sw Fa0/4`
`tux32@eth0—sw Fa0/2`

Commands:
tux32:
```
ifconfig eth0 up
ifconfig eth0 172.16.31.1/24
```
switch:
```
configure terminal
vlan 30
exit
vlan 31
exit
interface fastethernet 0/1
switchport mode access
switchport access vlan 30
exit
interface fastethernet 0/4
switchport mode access
switchport access vlan 30
exit
interface fastethernet 0/2
switchport mode access
switchport access vlan 31
end
```

Exp 3. Configure a Router in Linux

In this configuration we enable a new interface `tux34@eth2` and connect it to network 172.16.31.0/24 through vlan 31. This way *tux34* connects the two networks, and enabling IP forwarding allows *tux31* and *tux32* to communicate.

What routes are there in the tuxes? What are their meaning? The routes listed by command `ip route` are as follows:

- *tux31*: `default via 172.16.30.254 dev eth0`
  Default gateway of *tux31*. If no other route matches the destination of an IP packet being sent, it is sent to 172.16.30.254 (through interface `eth0`).
- *tux31*: `172.16.30.0/24 dev eth0`
  Means *tux31* is directly connected to all hosts in the network 172.16.30.0/24 through network interface `eth0`.
- *tux34*: `172.16.30.0/24 dev eth0`
  Means *tux34* is directly connected to network 172.16.30.0/24 through interface `eth0`.
- *tux34*: `172.16.31.0/24 dev eth2`
  Means *tux34* is directly connected to network 172.16.31.0/24 through interface `eth2`.

Cable configuration:
`tux31@eth0—sw Fa0/1`
`tux34@eth0—sw Fa0/4`
`tux34@eth2—sw Fa0/7`
`tux32@eth0—sw Fa0/2`

Commands:
tux34:
```
ifconfig eth2 up
ifconfig eth2 172.16.31.253/24
echo 1 > /.../ip_forward
echo 0 > /.../icmp_echo_ignore_broadcasts
```
tux32:
```
route add -net 172.16.30.0/24 gw 172.16.31.253
```
switch:
```
configure terminal
interface fastethernet 0/7
switchport mode access
switchport access vlan 41
end
```

- *tux32*: `172.16.30.0/24 via 172.16.31.253 dev eth0`
  Means that *tux32* is (indirectly) connected to the network 172.16.30.0/24 through gateway 172.16.31.253. So any IP packet whose destination is the network 172.16.30.0/24 will be sent to address 172.16.31.253 (who *tux32* expects to forward).
- *tux32*: `172.16.31.0/24 dev eth0`
  Means *tux32* is directly connected to network 172.16.31.0/24 through interface `eth0`.

What information does an entry of the forwarding table contain? The primary information are the *Destination* (host or networks) and the *Gateway*. Packets destined to a certain *Destination* are routed to the respective *Gateway*. For gateway entries, the host is not directly connected to the *Destination* but it always directly connected to the *Gateway*.

The table displayed by `route -n` shows further information. *Metric* scores a given route in terms of cost. It can contain any number of values that help a router or host determine the best route among multiple routes to a destination. A packet will generally be sent through the route with the lowest metric. The most basic metric is typically based on path length, hop count or delay. Some *Flags* are U for Up, meaning the route is up; G for Gateway, meaning the route is to a gateway; H for host, meaning the route's destination is a complete host address; D means the route was created by a redirect; and M means the route was modified by a redirect. *Iface* is the network interface used for the route, naturally.

What ICMP packets are observed and why? When *tux31* pings *tux32*, ICMP `ECHO_REQUEST` packets are sent with source 172.16.30.1 and destination 172.16.31.1. These are routed from `tux31@eth0` through *tux34* to `tux32@eth0`, and back for the `ECHO_REPLY` packets. These packets can be observed in both of *tux34*'s interfaces.

What ARP messages, and associated MAC addresses, are observed and why? ARP Request/Reply pairs which are required for IP communication:
- *tux31* (172.16.30.1) asks MAC address of 172.16.30.254 (*tux34*)
- *tux34* (172.16.31.253) asks MAC address of 172.16.31.1 (*tux32*)

When the ARP table is clear, the link layer needs to request the MAC address of the destination IP with an ARP request.

```
00:0f:fe:8b:e4:4d ff:ff:ff:ff:ff:ff ARP    Who has 172.16.30.254? Tell 172.16.30.1
00:21:5a:5a:7d:74 00:0f:fe:8b:e4:4d ARP    172.16.30.254 is at 00:21:5a:5a:7d:74
172.16.30.1       172.16.31.1       ICMP   Echo (ping) request  id=0x44cf, seq=1/256, ttl=64
172.16.31.1       172.16.30.1       ICMP   Echo (ping) reply    id=0x44cf, seq=1/256, ttl=63
fc:fb:fb:3a:fa:86 01:80:c2:00:00:00 STP    Conf. Root = 32768/30/fc:fb:fb:3a:fa:80  Cost = 0
172.16.30.1       172.16.31.1       ICMP   Echo (ping) request  id=0x44cf, seq=2/512, ttl=64
172.16.31.1       172.16.30.1       ICMP   Echo (ping) reply    id=0x44cf, seq=2/512, ttl=63
172.16.30.1       172.16.31.1       ICMP   Echo (ping) request  id=0x44cf, seq=3/768, ttl=64
172.16.31.1       172.16.30.1       ICMP   Echo (ping) reply    id=0x44cf, seq=3/768, ttl=63

00:01:02:21:83:0e ff:ff:ff:ff:ff:ff ARP    Who has 172.16.31.1? Tell 172.16.31.253
00:21:5a:61:30:63 00:01:02:21:83:0e ARP    172.16.31.1 is at 00:21:5a:61:30:63
172.16.30.1       172.16.31.1       ICMP   Echo (ping) request  id=0x44cf, seq=1/256, ttl=63
172.16.31.1       172.16.30.1       ICMP   Echo (ping) reply    id=0x44cf, seq=1/256, ttl=64
fc:fb:fb:3a:fa:87 01:80:c2:00:00:00 STP    Conf. Root = 32768/31/fc:fb:fb:3a:fa:80  Cost = 0
172.16.30.1       172.16.31.1       ICMP   Echo (ping) request  id=0x44cf, seq=2/512, ttl=63
172.16.31.1       172.16.30.1       ICMP   Echo (ping) reply    id=0x44cf, seq=2/512, ttl=64
172.16.30.1       172.16.31.1       ICMP   Echo (ping) request  id=0x44cf, seq=3/768, ttl=63
172.16.31.1       172.16.30.1       ICMP   Echo (ping) reply    id=0x44cf, seq=3/768, ttl=64
```

Figure 2: *tux31* pings *tux32*, seen from `tux34@eth0` and `tux34@eth2`

The observed MAC addresses are then:
- `tux31@eth0` (172.16.30.1): `00:0f:fe:8b:e4:4d`
- `tux34@eth0` (172.16.30.254): `00:21:5a:5a:7d:74`

- `tux34@eth2` (172.16.31.253): `00:01:02:21:83:0e`
- `tux32@eth0` (172.16.31.1): `00:21:5a:61:30:63`

What are the IP and MAC addresses associated to ICMP packets and why? Since ICMP packets are IPv4 packets – they live in the network layer – their IP addresses are fixed. When *tux31* pings *tux32*, the source IP is 172.16.30.1 and destination IP is 172.16.31.1. However, the MAC addresses reflect the hops the packet takes through the network. From *tux31* to *tux34*, the source MAC address is that of `tux31@eth0` and the destination MAC address is that of `tux34@eth0`. From *tux34* to *tux32*, the source MAC address is that of `tux34@eth2` and the destination MAC address is that of `tux32@eth0`. For the `ECHO_REPLY` packets it's the reverse, naturally.

Exp 4. Configure a Commercial Router and Implement NAT

To allow both networks to communicate with the Internet we connect one of the router's interfaces to network 172.16.31.0/24 and implement NAT, with allowance for *tux31* and *tux32* but not for *tux34*.

Cable configuration:
```
tux31@eth0—sw Fa0/1
tux34@eth0—sw Fa0/4
tux34@eth2—sw Fa0/7
tux32@eth0—sw Fa0/2
rt Giga0/1—sw Fa0/9
rt Giga0/0—172.16.1.254
```

Commands:
tux44:
```
route add default gw 172.16.31.254
```
tux42:
```
route add default gw 172.16.31.254
```
switch:
```
configure terminal
interface fastethernet 0/9
switchport mode access
switchport access vlan 31
end
```

How to configure a static route in a commercial router? The command's basic syntax is

```
ip route DestinationIP Mask GatewayIP
```

What are the paths followed by the packets in the experiments carried out and why?

- *tux31* pings 172.16.30.254 at *tux34*:
  172.16.30.1(*tux31*) → 172.16.30.254(*tux34*)
- *tux31* pings 172.16.31.253 at *tux34*:
  172.16.30.1(*tux31*) → 172.16.30.254(*tux34*)
- *tux31* pings 172.16.31.1 at *tux32*:
  172.16.30.1(*tux31*) → 172.16.30.254(*tux34*) → 172.16.31.1(*tux32*)
- *tux31* pings 172.16.31.254 at *router*:
  172.16.30.1(*tux31*) → 172.16.30.254(*tux34*) → 172.16.31.254(*rt*)
- *tux31* pings 172.16.1.39 at *router*:
  172.16.30.1(*tux31*) → 172.16.30.254(*tux34*) → 172.16.31.254(*rt*)
- *tux31* pings 172.16.1.254 (with NAT implemented in the router):
  172.16.30.1(*tux31*) → 172.16.30.254(*tux34*) → 172.16.31.254(*rt*) → 172.16.1.254(*lab*)
- When *tux32* pings *tux31* with redirects disabled and no gateway route to 172.16.30.0/24:
  172.16.31.1(*tux32*) → 172.16.31.254(*rt*) → 172.16.31.253(*tux34*) → 172.16.30.1(*tux31*)
  The ping reply skips the router — *tux34* forwards directly to *tux32* from *tux31*.
- When *tux32* pings *tux31* with redirects enabled but no gateway route to 172.16.30.0/24, the first ping is routed to 172.16.30.254(*rt*) — like the previous case — but further pings are routed directly to 172.16.30.253(*tux34*) — like the next case — because of the ICMP Redirect sent by the router.
- When *tux32* pings *tux31* with a gateway route to 172.16.30.0/24 through *tux34*: 172.16.31.1(*tux32*) → 172.16.31.253(*tux34*) → 172.16.30.1(*tux31*)
  See also Figure 6.

How to configure NAT in a commercial router? Specify which interface is private with `ip nat inside` and which interface is public with `ip nat outside`. Then create an access list of IP addresses which are allowed to pass through the NAT:

```
ip nat pool ovrld 172.16.1.39 172.16.1.39 prefix 24
ip nat inside source list 1 pool ovrld overload
access-list 1 permit 172.16.30.0 0.0.0.7
access-list 1 permit 172.16.31.0 0.0.0.7
```

Because we chose mask 0.0.0.7 for the access list for each network, *tux34* is not allowed past the NAT and its packets addressed to the Internet are rejected.

What does NAT do? NAT (*Network Address Translation*) translates IP addresses from a private network (172.16.30.0/24 and 172.16.31.0/24 in our case) to a public IP address (172.16.1.39), mapping each private IP address and port pair to a port on the public IP address. In other words, NAT implements a *bidirectional function* between pairs (`PrivateIP`, `PrivatePort`), where `PrivateIP` is an allowed IP address present in the access list, to pairs (`PublicIP`, `PublicPort`), where `PublicIP` in our case is always 172.16.1.39. This function's *mapping* is generated on demand and stored in the NAT *mapping table*. Packets arriving in the router from the private network have their IP and port translated according to this *mapping* and then routed to their destination; those arriving from the public network are translated back to their private IP and ports accordingly.

Exp 5. DNS

With Internet access and a proper DNS server configured, *tux31* and *tux32* can now resolve and ping domains on the Internet. *tux34* can't if we keep the access-list specified in the previous section, as 172.16.31.253 is not allowed past the router's NAT.

How to configure the DNS service at an host? Edit `/etc/resolv.conf` by hand or with `resolveconf`, providing a nameserver which implements the DNS protocol. In the lab we use domain netlab.fe.up.pt.

What packets are exchanged by DNS and what information is transported? The host sends DNS queries to the nameserver and receives DNS responses. Information transported in DNS messages include, among other things, the question — with the domain name and type of record requested — the answer — resolved IP, resource records of the domain name — and the authority.

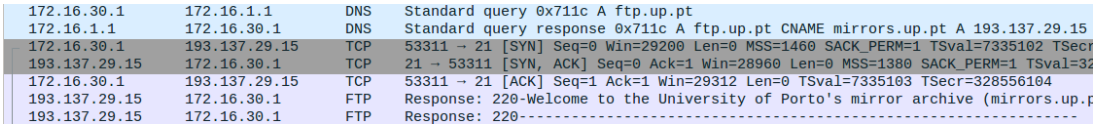
```
172.16.30.1     172.16.1.1      DNS   Standard query 0x5deb A fe.up.pt
172.16.1.1      172.16.30.1     DNS   Standard query response 0x5deb A fe.up.pt A 10.227.240.205
```

Figure 3: DNS lookup by *tux31*

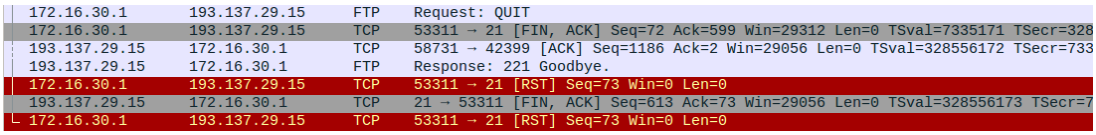
Exp 6. TCP connections

How many TCP connections are opened by the FTP application? In what connection is transported the FTP control information? The FTP protocol uses two TCP connections: the control connection, where the client sends the server FTP commands and receives FTP replies (server port 21), and the data connection, opened after a `PASV` command, through which the server sends retrieved files to the client.

What are the phases of a TCP connection? The *connection establishment*; *connection established*; and *connection termination* phases. The initiation is a *3-way handshake*, and the termination is a *4-way handshake* with a timeout.



(a) TCP connection establishment



(b) TCP connection termination

Figure 4: TCP connection phases

Is the throughput of a TCP data connection disturbed by the appearance of a second TCP connection? How? Yes, as evidenced from the throughput statistics at *`tux31`* and *`tux32`*, see Figure 7, where they download the same file from the same server. *`tux32`* starts the download about 10 seconds later than *`tux31`*, dropping its throughput to half. When *`tux31`* concludes the download, the throughput at *`tux32`* doubles, as it is not allowed the bandwidth that was reserved for *`tux31`*.

Conclusions

The private network we configured in part 2 was simple but allows us to understand throughly multiple aspects and concepts of computer networking, namely routing, protocols, layers and addresses.

The *`download`* application we developed is straightforward, reports on all stages, all FTP control messages, and all errors it encounters, and closes the connection with the server gracefully even in those cases.

Appendix: Statistics

```
1. FTP URL: ftp://ftp.up.pt/pub/debian/project/trace/mirrors.up.pt
 Defaulting username to anonymous
 Defaulting password to upstudent-rcom
 url.protocol: ftp
 url.username: anonymous
 url.password: upstudent-rcom
 url.hostname: ftp.up.pt
 url.pathname: pub/debian/project/trace/
 url.filename: mirrors.up.pt
 url.port: 21
2. Resolve hostname ftp.up.pt and setup control socket
 2.1. Resolved hostname ftp.up.pt successfully
  host->h_name: mirrors.up.pt
  host->h_addrtype: 2 [IPv4=2]
 2.2. Protocol IP Address: 193.137.29.15
      Establishing control connection with FTP server
 2.3. Opened control socket for FTP's protocol connection
 2.4. Connected control socket to 193.137.29.15:21
    [REPLY] 220-Welcome to the University of Porto's mirror archive (mirrors.up.pt)
    [REPLY] 220----------------------------------------------------------------
    [REPLY] 220-
    [REPLY] 220-All connections and transfers are logged. The max number of connections is 200.
    [REPLY] 220-
    [REPLY] 220-For more information please visit our website: http://mirrors.up.pt/
    [REPLY] 220-Questions and comments can be sent to mirrors@uporto.pt
    [REPLY] 220-
    [REPLY] 220-
    [REPLY] 220
 2.5. Successfully established control socket connection
3. Send USER and PASS login commands to FTP server
    [COMMD] USER anonymous
    [REPLY] 331 Please specify the password.
 3.1. Server confirmed user anonymous
    [COMMD] PASS upstudent-rcom
    [REPLY] 230 Login successful.
 3.2. Server acknowledges login, proceeding
4. Establish passive connection with FTP server
    [COMMD] PASV
    [REPLY] 227 Entering Passive Mode (193,137,29,15,229,4).
 4.1. Entered Passive Mode: (193,137,29,15,229,4)
 4.2. Passive IP Address: 193.137.29.15:58628
 4.3. Opened passive socket for FTP's passive connection
 4.4. Connected passive socket to 193.137.29.15:58628
 4.5. Successfully established passive socket connection
5. Retrieve file from Server (retrieve command)
    [COMMD] RETR pub/debian/project/trace/mirrors.up.pt
    [REPLY] 150 Opening BINARY mode data connection for pub/debian/project/trace/mirrors.up.pt (802 bytes).
 5.1. Confirmed, server retrieved pub/debian/project/trace/mirrors.up.pt
6. Download file mirrors.up.pt into current directory
 6.1. Opened output file successfully
      Reading...
 6.2. Done.
```

Figure 5: Successful download of `ftp://ftp.up.pt/pub/debian/project/trace/mirrors.up.pt`

Figure 6: *tux32* pings *tux31*, seen from `tux32@eth0`

172.16.31.1	172.16.30.1	ICMP	Echo (ping) request	id=0x4b3b, seq=1/256, ttl=64
172.16.31.254	172.16.31.1	ICMP	Redirect	(Redirect for host)
172.16.31.1	172.16.30.1	ICMP	Echo (ping) request	id=0x4b3b, seq=1/256, ttl=63
172.16.30.1	172.16.31.1	ICMP	Echo (ping) reply	id=0x4b3b, seq=1/256, ttl=63
fc:fb:fb:3a:fa…	01:80:c2:00:00:00	STP	Conf. TC + Root = 32768/31/fc:fb:fb:3a:fa:80	Cost
172.16.31.1	172.16.30.1	ICMP	Echo (ping) request	id=0x4b3b, seq=2/512, ttl=64
172.16.31.254	172.16.31.1	ICMP	Redirect	(Redirect for host)
172.16.30.1	172.16.31.1	ICMP	Echo (ping) reply	id=0x4b3b, seq=2/512, ttl=63
172.16.31.1	172.16.30.1	ICMP	Echo (ping) request	id=0x4b3b, seq=3/768, ttl=64
172.16.30.1	172.16.31.1	ICMP	Echo (ping) reply	id=0x4b3b, seq=3/768, ttl=63
fc:fb:fb:3a:fa…	01:00:0c:cc:cc:cc	CDP	Device ID: tux-sw3 Port ID: FastEthernet0/2	
fc:fb:fb:3a:fa…	01:80:c2:00:00:00	STP	Conf. Root = 32768/31/fc:fb:fb:3a:fa:80	Cost = 0
172.16.31.1	172.16.30.1	ICMP	Echo (ping) request	id=0x4b3b, seq=4/1024, ttl=64
172.16.31.254	172.16.31.1	ICMP	Redirect	(Redirect for host)
172.16.30.1	172.16.31.1	ICMP	Echo (ping) reply	id=0x4b3b, seq=4/1024, ttl=63
172.16.31.1	172.16.30.1	ICMP	Echo (ping) request	id=0x4b3b, seq=5/1280, ttl=64
172.16.31.254	172.16.31.1	ICMP	Redirect	(Redirect for host)
172.16.30.1	172.16.31.1	ICMP	Echo (ping) reply	id=0x4b3b, seq=5/1280, ttl=63

(a) With ICMP redirects disabled and no gateway route

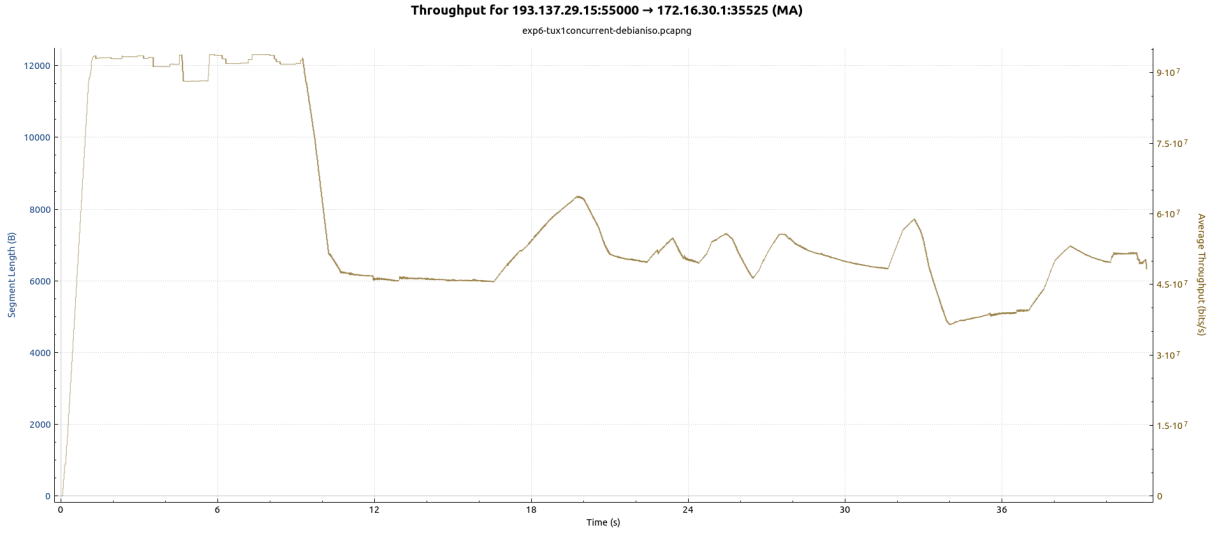
172.16.31.1	172.16.30.1	ICMP	Echo (ping) request	id=0x4bae, seq=1/256, ttl=64
172.16.31.254	172.16.31.1	ICMP	Redirect	(Redirect for host)
172.16.30.1	172.16.31.1	ICMP	Echo (ping) reply	id=0x4bae, seq=1/256, ttl=63
172.16.31.1	172.16.30.1	ICMP	Echo (ping) request	id=0x4bae, seq=2/512, ttl=64
172.16.30.1	172.16.31.1	ICMP	Echo (ping) reply	id=0x4bae, seq=2/512, ttl=63
fc:fb:fb:3a:fa:84	01:80:c2:00:00:00	STP	Conf. Root = 32768/31/fc:fb:fb:3a:fa:80	Cost = 0
172.16.31.1	172.16.30.1	ICMP	Echo (ping) request	id=0x4bae, seq=3/768, ttl=64
172.16.30.1	172.16.31.1	ICMP	Echo (ping) reply	id=0x4bae, seq=3/768, ttl=63
172.16.31.1	172.16.30.1	ICMP	Echo (ping) request	id=0x4bae, seq=4/1024, ttl=64
172.16.30.1	172.16.31.1	ICMP	Echo (ping) reply	id=0x4bae, seq=4/1024, ttl=63
fc:fb:fb:3a:fa:84	01:80:c2:00:00:00	STP	Conf. Root = 32768/31/fc:fb:fb:3a:fa:80	Cost = 0
fc:fb:fb:3a:fa:84	fc:fb:fb:3a:fa:84	LOOP	Reply	
172.16.31.1	172.16.30.1	ICMP	Echo (ping) request	id=0x4bae, seq=5/1280, ttl=64
172.16.30.1	172.16.31.1	ICMP	Echo (ping) reply	id=0x4bae, seq=5/1280, ttl=63

(b) With ICMP redirects enabled

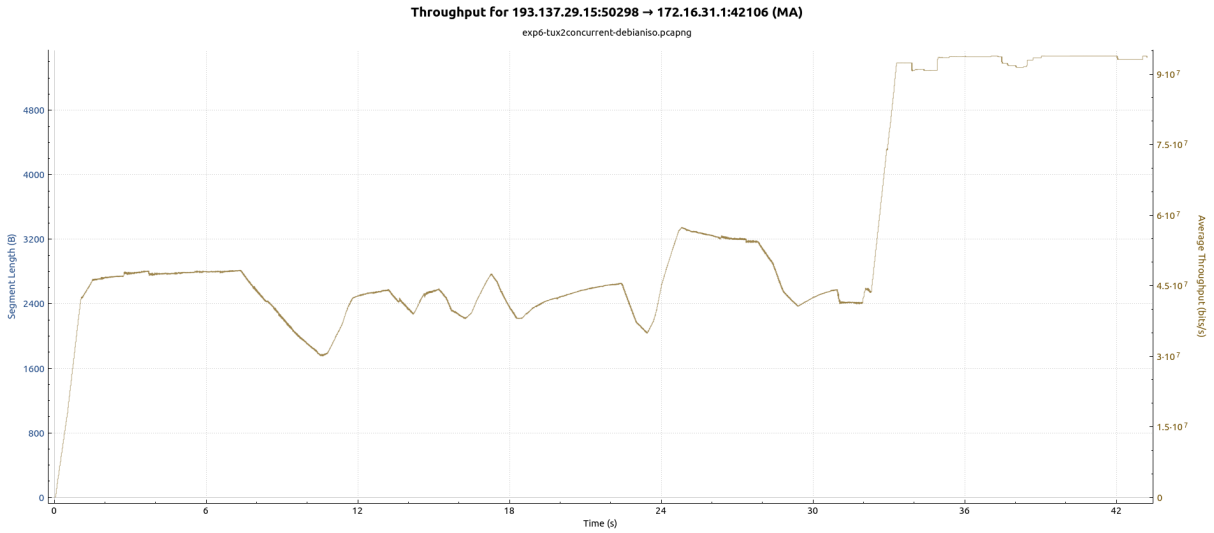
172.16.31.1	172.16.30.1	ICMP	Echo (ping) request	id=0x4c27, seq=1/256, ttl=64
172.16.30.1	172.16.31.1	ICMP	Echo (ping) reply	id=0x4c27, seq=1/256, ttl=63
172.16.31.1	172.16.30.1	ICMP	Echo (ping) request	id=0x4c27, seq=2/512, ttl=64
172.16.30.1	172.16.31.1	ICMP	Echo (ping) reply	id=0x4c27, seq=2/512, ttl=63
fc:fb:fb:3a:fa…	01:80:c2:00:00:00	STP	Conf. Root = 32768/31/fc:fb:fb:3a:fa:80	Cost = 0
172.16.31.1	172.16.30.1	ICMP	Echo (ping) request	id=0x4c27, seq=3/768, ttl=64
172.16.30.1	172.16.31.1	ICMP	Echo (ping) reply	id=0x4c27, seq=3/768, ttl=63
172.16.31.1	172.16.30.1	ICMP	Echo (ping) request	id=0x4c27, seq=4/1024, ttl=64
172.16.30.1	172.16.31.1	ICMP	Echo (ping) reply	id=0x4c27, seq=4/1024, ttl=63
fc:fb:fb:3a:fa…	01:80:c2:00:00:00	STP	Conf. Root = 32768/31/fc:fb:fb:3a:fa:80	Cost = 0
172.16.31.1	172.16.30.1	ICMP	Echo (ping) request	id=0x4c27, seq=5/1280, ttl=64
172.16.30.1	172.16.31.1	ICMP	Echo (ping) reply	id=0x4c27, seq=5/1280, ttl=63

(c) With gateway through *tux34*

Figure 7: Throughputs of concurrent download of 310 MB file in *tux31* and *tux32*.



(a) Throughput in *tux31*



(b) Throughput in *tux32*

Appendix: Code

pipeline.c

```
#include "pipeline.h"
#include "debug.h"

#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <regex.h>

#include <netdb.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

//
// FTP Control Socket (User-Protocol Interpreter)
//
static FILE* controlstream = NULL;
static int controlfd = 0;

//
// FTP Passive Socket (Data Transfer)
//
static FILE* passivestream = NULL;
static int passivefd = 0;

//
// Input FTP URL
//
static url_t url;

static void close_control_socket();

static void close_passive_socket();

static void free_url();

//
// Send an FTP command to the control socket.
//
static int send_ftp_command(const char* command) {
    ssize_t count = 0, s = 0;
    ssize_t len = strlen(command);

    logftpcommand(command);

    do {
    count += s = send(controlfd, command + count, len - count, 0);
    if (s <= 0) libfail("Failed to write anything to control socket");
    } while (count < len);
```

```
    return 0;
}

//
// Read and discard reply lines from the FTP control socket stream
   until reading a
// line that starts with an FTP code not followed by a dash.
//
static int recv_ftp_reply(char* code, char** line) {
    char* reply = NULL;

    ssize_t read_size = 0;

    do {
    free(reply);

    reply = NULL;
    size_t n = 0;

    read_size = getline(&reply, &n, controlstream); // keeps \r\n
    if (read_size < 1) libfail("Failed to read reply from control
   socket stream");

    logftpreply(reply);
    } while ('1' > reply[0] || reply[0] > '5' || read_size < 4 ||
   reply[3] == '-');

    strncpy(code, reply, 3); code[3] = '\0';
    line != NULL ? *line = reply : free(reply);

    return 0;
}

//
// Extract substring matched by a capture group from the source string,
// returning NULL if the capture group didn't match.
//
static char* regexcap(const char* source, regmatch_t match) {
    ssize_t start = match.rm_so;
    ssize_t end = match.rm_eo;
    if (start == -1) return NULL;

    size_t len = end - start;

    char* buffer = malloc((len + 1) * sizeof(char));
    strncpy(buffer, source + start, len);
    buffer[len] = '\0';

    return buffer;
}

//
// 1. Parse program's input, the FTP URL
// For URL parsing we use a weak regular expression. It accepts all
   valid (ftp) URLs
// with non-empty filenames, although it also passes a whole bunch of
```

```
    invalid ones too.
// It gets the job done quickly. Note: it does not accept a port after
    the host.
//
int parse_url(const char* urlstr) {
    regex_t regex;
    // capturers:     1       23         4 5              6          78
            9
    //                  ftp ://[user      [:pass]       @] host
    /path/ to/   filename
    regcomp(&regex, "^(ftp)://(([^:@/ ]+)(:([^:@/ ]+))?@)?([^:@/
    ]+)/(([^/ ]+/)*)([^/ ]+)$",
    REG_EXTENDED | REG_ICASE | REG_NEWLINE);

    regmatch_t pmatch[10];

    // Regex-parse the input url
    int s = regexec(&regex, urlstr, 10, pmatch, 0);
    regfree(&regex);
    if (s != 0) fail("Invalid URL {no port, must have nonempty
    filename}");

    // Take captures
    url = (url_t){
    .protocol = regexcap(urlstr, pmatch[1]),
    .username = regexcap(urlstr, pmatch[3]),
    .password = regexcap(urlstr, pmatch[5]),
    .hostname = regexcap(urlstr, pmatch[6]),
    .pathname = regexcap(urlstr, pmatch[7]),
    .filename = regexcap(urlstr, pmatch[9]),
    .port = 21
    };

    atexit(free_url);

    progress("1. FTP URL: "CGREEN"ftp://%s/%s%s"CEND, url.hostname,
    url.pathname, url.filename);

    // Pick default username
    if (url.username == NULL) {
    url.username = strdup("anonymous"); // *** DEFAULT USER
    progress(" Defaulting username to %s", url.username);
    }

    // Pick default password
    if (url.password == NULL) {
    url.password = strdup("upstudent-rcom"); // *** DEFAULT PASS
    progress(" Defaulting password to %s", url.password);
    }

    // Extra: warning for the common test case where host is ftp.up.pt
    if (strcmp(url.username, "anonymous") && !strcmp(url.hostname,
    "ftp.up.pt")) {
    printf(CYELLOW"\n Warning: ftp.up.pt operates only in anonymous
    mode.\n\n"CEND);
    }
```

```
    progress(" url.protocol: %s", url.protocol);
    progress(" url.username: %s", url.username);
    progress(" url.password: %s", url.password);
    progress(" url.hostname: %s", url.hostname);
    progress(" url.pathname: %s", url.pathname);
    progress(" url.filename: %s", url.filename);
    progress(" url.port: %d", url.port);

    return 0;
}

//
// 2. Resolve hostname to server's IPv4 address and setup control socket
//
int ftp_open_control_socket() {
    char code[4];

    progress("2. Resolve hostname %s and setup control socket",
   url.hostname);

    // 2.1. resolve
    struct hostent* host = gethostbyname(url.hostname);
    if (host == NULL) libfail("Could not resolve hostname %s",
   url.hostname);

    progress(" 2.1. Resolved hostname %s successfully", url.hostname);
    progress("  host->h_name: %s", host->h_name);
    progress("  host->h_addrtype: %d [IPv4=%d]", host->h_addrtype,
   AF_INET);

    // 2.2. extract IPv4 address
    if (host->h_addrtype != AF_INET) fail("Expected IPv4 address");

    char* protocolip = inet_ntoa(*(struct in_addr*)host->h_addr);

    progress(" 2.2. Protocol IP Address: %s", protocolip);
    progress("      Establishing control connection with FTP server");

    // 2.3. socket()
    controlfd = socket(AF_INET, SOCK_STREAM, 0);
    if (controlfd == -1) libfail("Failed to open socket for control
   connection");

    controlstream = fdopen(controlfd, "r");
    atexit(close_control_socket);

    progress(" 2.3. Opened control socket for FTP's protocol
   connection");

    // 2.4. connect()
    struct sockaddr_in in_addr = {0};
    in_addr.sin_family = AF_INET; // IPv4 address
    in_addr.sin_port = htons(url.port); // host byte order -> network
   byte order
    in_addr.sin_addr.s_addr = inet_addr(protocolip); // to network
```

```
    format

    int s = connect(controlfd, (struct sockaddr*)&in_addr,
    sizeof(in_addr));
    if (s != 0) libfail("Failed to connect() control socket");

    progress(" 2.4. Connected control socket to %s:%d", protocolip, 21);

    // 2.5. recv() 220 === Service ready for new user
    recv_ftp_reply(code, NULL);
    if (strcmp(code, "220") != 0) unexpected("Expected reply 220, got
    %s", code);

    progress(" 2.5. Successfully established control socket
    connection");

    return 0;
}

//
// 3. Login user.
// Simply a USER command followed by a PASS command.
//
int ftp_login() {
    char code[4];

    progress("3. Send USER and PASS login commands to FTP server");

    // 3.1. send() USER username
    //      recv() 331 === User name okay, send password
    char* user_command = malloc((10 + strlen(url.username)) *
    sizeof(char));
    sprintf(user_command, "USER %s\r\n", url.username);

    send_ftp_command(user_command);
    free(user_command);

    recv_ftp_reply(code, NULL);
    if (strcmp(code, "331") != 0) unexpected("Expected reply 331, got
    %s", code);

    progress(" 3.1. Server confirmed user %s", url.username);

    // 3.2. send() PASS password
    //      recv() 230 === Login successful, proceed
    char* pass_command = malloc((10 + strlen(url.password)) *
    sizeof(char));
    sprintf(pass_command, "PASS %s\r\n", url.password);

    send_ftp_command(pass_command);
    free(pass_command);

    recv_ftp_reply(code, NULL);
    if (strcmp(code, "230") != 0) unexpected("Expected reply 230, got
    %s", code);
```

```
    progress(" 3.2. Server acknowledges login, proceeding");

    return 0;
}

//
// 4. Enter passive mode
//
int ftp_open_passive_socket() {
    char code[4];

    progress("4. Establish passive connection with FTP server");

    // 4.1. send() PASV
    //      recv() 227 Entering Passive Mode
    (IP3.IP2.IP1.IP0,Port1,Port0)
    send_ftp_command("PASV \r\n");

    char* line;
    recv_ftp_reply(code, &line);
    if (strcmp(code, "227") != 0) unexpected("Expected reply 227, got
    %d", code);

    regex_t regex;
    regcomp(&regex, "([0-9]+, ?[0-9]+, ?[0-9]+, ?[0-9]+, ?[0-9]+,
    ?[0-9]+)",
    REG_EXTENDED | REG_ICASE);

    regmatch_t pmatch[2];

    int s = regexec(&regex, line, 2, pmatch, 0);
    regfree(&regex);
    if (s != 0) fail("Unexpected Passive Mode response format: %s",
    line);

    char* substr = regexcap(line, pmatch[1]);
    free(line);

    progress(" 4.1. Entered Passive Mode: (%s)", substr);

    // 4.2. extract passive ip from response line
    int ip3, ip2, ip1, ip0, port1, port0;
    sscanf(substr, "%d, %d, %d, %d, %d, %d", &ip3, &ip2, &ip1, &ip0,
    &port1, &port0);
    free(substr);

    char passiveip[20] = {0};
    sprintf(passiveip, "%d.%d.%d.%d", ip3, ip2, ip1, ip0);
    int port = port1 * 256 + port0;

    progress(" 4.2. Passive IP Address: %s:%d", passiveip, port);

    // 4.3. socket()
    passivefd = socket(AF_INET, SOCK_STREAM, 0);
    if (passivefd == -1) libfail("Failed to open socket for passive
    connection");
```

```
    passivestream = fdopen(passivefd, "r+");
    atexit(close_passive_socket);

    progress(" 4.3. Opened passive socket for FTP's passive
    connection");

    // 4.4. connect()
    struct sockaddr_in in_addr = {0};
    in_addr.sin_family = AF_INET; // IPv4 address
    in_addr.sin_port = htons(port); // host bytes -> network bytes
    in_addr.sin_addr.s_addr = inet_addr(passiveip); // to network format

    s = connect(passivefd, (struct sockaddr*)&in_addr, sizeof(in_addr));
    if (s != 0) libfail("Failed to connect() passive socket");

    progress(" 4.4. Connected passive socket to %s:%d", passiveip,
    port);
    progress(" 4.5. Successfully established passive socket
    connection");

    return 0;
}

//
// 5. Send Retrieve command to FTP server
// This command instructs the server to send the filename through the
    passive connection.
//
int send_retrieve() {
    char code[4];

    progress("5. Retrieve file from Server (retrieve command)");

    // 5.1. send() RETR filepath
    //      recv() 150 File status okay, opening data transfer now
    size_t len = strlen(url.pathname) + strlen(url.filename);
    char* retr_command = malloc((10 + len) * sizeof(char));
    sprintf(retr_command, "RETR %s%s\r\n", url.pathname, url.filename);

    send_ftp_command(retr_command);
    free(retr_command);

    recv_ftp_reply(code, NULL);
    if (strcmp(code, "150") != 0) unexpected("Expected reply 150, got
    %s", code);

    progress(" 5.1. Confirmed, server retrieved %s%s", url.pathname,
    url.filename);

    return 0;
}

//
// 6. Download file retrieved.
// It is being sent through TCP to port 20, we just read the socket
```

```
// and forward to the output file stream until the transmission is over.
//
int download_file() {
    char code[4];

    progress("6. Download file %s into current directory",
    url.filename);

    // 6.1. Open output file in current directory
    FILE* out = fopen(url.filename, "w"); // streams are love, streams
    are life
    if (out == NULL) libfail("Failed to open output file in current
    directory");

    progress(" 6.1. Opened output file successfully");

    // 6.2. Canonical reading loop on passivefd, read the whole thing
    char buffer[1024]; // TCP caps at 1500B/p practically
    ssize_t read_size = 0;

    progress("      Reading...");

    while ((read_size = read(passivefd, buffer, sizeof(buffer))) > 0) {
    size_t write_size = fwrite(buffer, read_size, 1, out);
    if (write_size != 1) libfail("Failed to write to output file");
    }

    if (read_size < 0) libfail("Failed to read file on passive socket");

    fclose(out);

    progress(" 6.2. "CGREEN"Done."CEND);

    // 6.3. recv() 226 === Closing data connection, transfer complete
    recv_ftp_reply(code, NULL);
    if (strcmp(code, "226") != 0) unexpected("Expected reply 226, got
    %d", code);

    return 0;
}

//
// 7. Close the connection to the server.
//
static void close_control_socket() {
    send_ftp_command("QUIT \r\n");
fclose(controlstream);
}

static void close_passive_socket() {
    fclose(passivestream);
}

static void free_url() {
    free(url.protocol);
    free(url.username);
```

```
    free(url.password);
    free(url.hostname);
    free(url.pathname);
    free(url.filename);
}
```

pipeline.h

```
#ifndef PIPELINE_H___
#define PIPELINE_H___

typedef struct {
    char* protocol;
    char* username;
    char* password;
    char* hostname;
    char* pathname;
    char* filename;
    int port;
} url_t;

int parse_url(const char* urlstr);

int ftp_open_control_socket();

int ftp_login();

int ftp_open_passive_socket();

int send_retrieve();

int download_file();

#endif // PIPELINE_H___
```

main.c

```
#include "pipeline.h"

#include <stdlib.h>
#include <stdio.h>

int main(int argc, char** argv) {
    if (argc != 2) {
        printf("Expected 1 argument. Usage:\n  ./download ftp://...\n");
        return 0;
    }

    // 1. Parse input FTP URL
    parse_url(argv[1]);

    // 2. Resolve hostname to server's IPv4 IP address and open
   protocol socket
    ftp_open_control_socket();
```

```
    // 3. Login to the server (user + password)
    ftp_login();

    // 4. Enter passive mode and open passive socket
    ftp_open_passive_socket();

    // 5. Send retrieve command for file
    send_retrieve();

    // 6. Download file
    download_file();

    // 7. Close connection to server
    exit(EXIT_SUCCESS);
}
```

debug.c

```
#include "debug.h"

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

void progress(const char* format, ...) {
    va_list arglist;
    va_start(arglist, format);
    #if PRINT_PROGRESS
    vprintf(format, arglist);
    printf("\n");
    #endif
    va_end(arglist);
}

void logftpcommand(const char* line) {
    #if PRINT_FTP_COMMAND
    printf(CBLUE"    [COMMD] %s"CEND, line);
    #endif
}

void logftpreply(const char* line) {
    #if PRINT_FTP_REPLY
    printf(CPURP"    [REPLY] %s"CEND, line);
    #endif
}

void fail(const char* format, ...) {
    printf(CRED"FAIL: ");
    va_list arglist;
    va_start(arglist, format);
    vfprintf(stderr, format, arglist);
    va_end(arglist);
    printf("\n"CEND);
    exit(EXIT_FAILURE);
}
```

```
void libfail(const char* format, ...) {
    int err = errno;
    printf(CRED"ERROR: ");
    va_list arglist;
    va_start(arglist, format);
    vfprintf(stderr, format, arglist);
    va_end(arglist);
    printf("\n");
    errno = err;
    perror("Library error (errno)");
    printf(CEND);
    exit(EXIT_FAILURE);
}

void unexpected(const char* format, ...) {
    printf(CRED"UNEXPECTED SERVER RESPONSE: ");
    va_list arglist;
    va_start(arglist, format);
    vfprintf(stderr, format, arglist);
    va_end(arglist);
    printf("\n"CEND);
    exit(EXIT_FAILURE);
}
```

debug.h

```
#ifndef DEBUG_H___
#define DEBUG_H___

#include <stdarg.h>

//
// Linux terminal colors
//
#define CRED "\e[1;31m"

#define CGREEN "\e[1;32m"

#define CYELLOW "\e[1;33m"

#define CBLUE "\e[1;34m"

#define CPURP "\e[1;35m"

#define CEND "\e[m"

//
// Print what?
//
#define PRINT_PROGRESS 1

#define PRINT_FTP_COMMAND 1

#define PRINT_FTP_REPLY 1

//
```

```
// Logging functions
//
void progress(const char* format, ...);

void logftpcommand(const char* line);

void logftpreply(const char* line);

void fail(const char* format, ...);

void libfail(const char* format, ...);

void unexpected(const char* format, ...);

#endif // DEBUG_H___
```