CS305 · Computer Networking

Assignment 2

何泽安 (He Zean)
12011323
Mar. 30, 2022

Q. 1

The *MAIL FROM:* header is a part of the SMTP handshaking protocol, which is sent by the client to identify the sender for the server, and may let the server do relevant checks about the sender (may like checking its privilege or so) and get ready for receiving the mail message. While the *From:* header is a part of the mail message itself – SMTP protocol doesn't care about it, but may help the mail applications to parse related information.

Q. 2

SMTP uses a line consisting of a single period to indicate the message ends, which goes straightly in the next line of the last line in the message body. While HTTP put the size of message body into the *Content-Length:* header, let the client receive and calculate when should split this message out from the TCP/UDP stream.

HTTP can NOT use the same method as SMTP, since SMTP requires the email message body be consists of 7-bit ASCII codes, that are readable, while there is no such limitation for HTTP, it almost impossible for the client to "parse" the message before receiving all of it, and with the Content-Type header. Therefore, it's too difficult for HTTP clients to recognize "a period at the end of message".

Q. 3

We can check the items in the cache regularly (frequently) and thus get n records. For each item (domain), statistic the time n_i they appears in these n records. Then we can roughly represent the "cache rate" as $\frac{n_i}{n}$, the higher "cache rate" is, the more frequent the site been accessed.

We first explain its availability. We know that DNS records have TTLs, which means that if a site is less frequently visited, at some point, one client may query the DNS server for its IP address, at this point, the DNS server also add this record into cache. Unfortunately, this site may be so unpopular that nobody access it in a large scale of time, of cause this record should already been deleted from cache. Then in the next several checks, this domain is not in the cache, say, n_i should be low, therefore leads a low "cache rate". However, if one site is popular, it's possible that once after the cache expires (pass TTL), one client will request its DNS record again before long, therefore the cache got a sudden new record and for we observers, this domain is always in the cache, leading a high n_i.

However, we know that TTL of DNS records varies, site with TTL longer than average itself makes the observe result be larger than actual, vice versa. However, with a large n, we can decrease this affect.

Q. 4

For C-S distribution, the time cannot be shorter than the time for client to send N copies of the file, and cannot be shorter than the maximum download time within the clients. We take the lower bound of this relation:

$$D_{CS} = \max\{\frac{NF}{u_s}, \frac{F}{d_{min}}\}$$

For P2P distribution, the time cannot be shorter than time for ① the server to send the whole file; ② the slowest client to download the whole file and ③ the time for using every machine to upload N copies of files in total. Therefore, the lower bound of P2P is:

$$D_{CS} = \max\{\frac{F}{u_s}, \frac{F}{d_{min}}, \frac{NF}{u_s+\Sigma_{i=1}^{N} u_i}\}$$

The time listed above are in the unit of second.

N \u	300Kbps		700Kbps		2Mbps	
	C-S	P2P	C-S	P2P	C-S	P2P
10	7680	7680	7680	7680	7680	7680
100	51200	25903.56	51200	15616.20	51200	7680
1000	512000	47559.78	512000	21524.85	512000	7680

Q. 5

(a) N files (there's no sense that we mix a high quality video with a low quality audio, vice versa, thus we first sort the videos and audios by quality, then match the pair from high to low).

(b) $2N$ files trivially, N videos + N audios.

Q. 6

(a) Since the server isn't ready to listen and reply to the port, it failed to do the handshaking process to establish the TCP connection. As for the socket API, it may raise an ConnectionRefusedError to report this.

```
clientSocket.connect((serverName, serverPort))
====
ConnectionRefusedError: [Errno 61] Connection refused
```

(b) It first runs successfully (UDP as a 无连接的 protocol, does not establish a connection, it just send the message to the specified "server" ip & port), when it reaches the recvfrom line, the program may be

blocked, or "paused", since this operation is blocking and let the program wait, until the server, or else, send a message to this port (socket).

```
modifiedMessage, serverAddress = clientSocket.recvfrom(2048)
```

(c) **TCP:** similar to question (a), the client try to contact *port1* of server, while the server program runs on *port2*, for the server, it doesn't know a client send the TCP connection request and do nothing, while the client may not establish the TCP connection and raise an error, if no program is listening *port1*, or receive unexpected response and may therefore crashed if another program runs on that port and reply something unexpected. **UDP:** similar to question (b), the server do nothing but keep listening *port2*, and the client first send a message to *port1*, then keep waiting for the response.

Q. 7

For the Host B, its source port is y and the destination port is x.

Q. 8

If the application's service concerns the timeliness of the transmission more than the accuracy of the transmitted packet content (typically, the transmitted content is large, and such content is somewhat tolerant of noise / packet loss), it may prefer UDP more than TCP. The RDT mechanism makes TCP slower than UDP, which is terrible for the case having much data to send (in a short time scale), what's worth, the server also have to prepare sufficient cache for resending packages. Plus, we knows that TCP provides congestion control, that if the client already in some bad network condition, this feature let the server send packages slower, makes things worse.

Q. 9

	0	1	0	1	0	0	1	1
+	0	1	1	0	0	1	1	0
	1	0	1	1	1	0	0	1
+	0	1	1	1	0	1	0	0
(1	*0*	*0*	*1*	*0*	*1*	*1*	*0*	*1)*
wrap	0	0	1	0	1	1	1	0
1's cp	**1**	**1**	**0**	**1**	**0**	**0**	**0**	**1**

We notice that after two addition, there's an one overflow, we then need to wrap back, say, add 1 to the last 8 bits. Then do the 1's complement and get *11010001*.

The mechanism of taking the 1's complement can help the CPU do a faster check. The adder in ALU is faster enough that after adding all the data bytes, it only costs about 1 clock cycle to add the checksum,

then directly check if all bits are 1 (this also costs little), otherwise, CPU needs to spend more time to check the sum of data bytes and the checksum (not 1's complement, but the sum itself) bit-by-bit. The overall cost of using 1's complement as checksum is much less than the sum itself.

To detect the error, just add all the data 8-bit bytes, then add up the checksum (in this process, each overflow are handled by taking a wrap back). Finally check if the sum is all 1, say 11111111, otherwise, there must exists error(s).

1-bit error can always been detected, as it always leads the sum cannot be all 1, simple math problem. But 2-bit error may not be detected. Consider the counter example that, the first two words given in the question are changed to 01010001**0** and 01100111**1**, with the same checksum, the sum is still all 1.