

49202 Communication Protocols

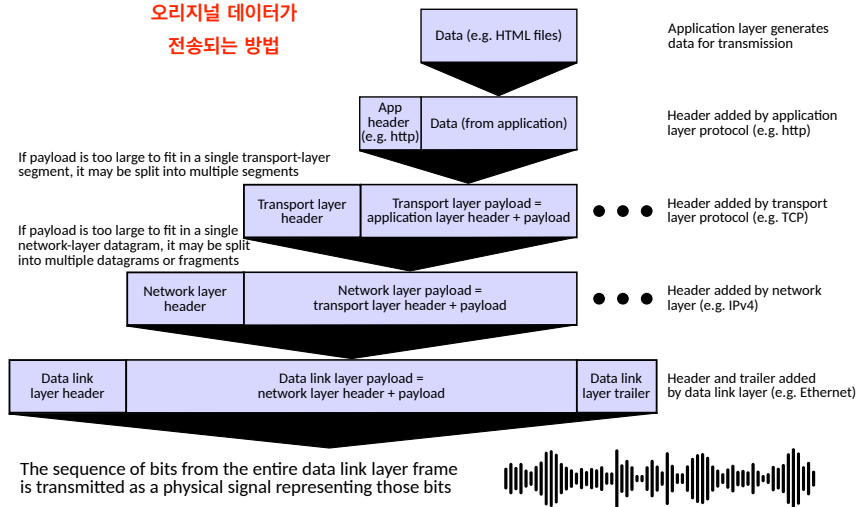
The Application Layer - Examples

Daniel R. Franklin

Faculty of Engineering & IT

March 5, 2024

Protocol headers, overhead and efficiency



Protocol headers, overhead and efficiency

- Anything which is **not payload** is **overhead**
- This includes all headers and trailers added by each layer in the protocol stack
- In some cases, one layer may need to split the data (payload) passed to it from higher layers in the stack - e.g.
 - TCP splits up data written to it from the application layer into multiple *segments* if the size of the written data exceeds the maximum segment size (MSS)
 - IP splits up data written to it from the transport layer into multiple *fragments* if the size exceeds the underlying networks' maximum transmission unit (MTU)
- If such splitting occurs, each piece of the original payload needs its own set of (TCP or IP) headers so that the pieces can be properly reassembled.

Calculating overheads and/or efficiency

- We can do some **simple calculations** to understand how **efficient** a particular protocol combination is
- For example, we can express the overhead as a fraction of the total packet size
- We could also express the efficiency as the fraction or percentage of the total packet size which contains useful payload
- Alternatively, given the percentage overhead or efficiency, you may be asked to calculate some unknown component of the overhead.
- Note that our definition of *efficiency* here only takes into account the active transmitting time of the network interface - an alternative definition of efficiency is the fraction of *total time* that we are spending sending useful data - this can also account for inter-frame gaps and other idle time.
- We will explore this later!

Example 1: calculating overheads and/or efficiency

Example: A **DNS query** is 33 bytes in length. It is being sent using UDP over IPv4 over Ethernet. Calculate the percentage overhead, assuming that Ethernet header + trailer is 26 bytes, IPv4 header is 20 bytes and UDP header is 8 bytes.

Solution:

$$\%O = \frac{O_{UDP} + O_{IPv4} + O_{Ether}}{O_{UDP} + O_{IPv4} + O_{Ether} + Payload} \times 100 = \frac{8 + 20 + 26}{8 + 20 + 26 + 33} \times 100 = 62\%$$

- An important observation: the payload is small, so most of this Ethernet frame is overhead!
- Efficiency is $\nu = 100 - \%O = 38\%$

Example 2: calculating overheads and/or efficiency

Example: Calculate the % overhead for an Ethernet frame carrying the maximum amount of traffic over TCP and IPv4, assuming the MTU of the Ethernet network has been set to 1500 bytes.

Solution: We assume the TCP header size is 20 bytes, the IPv4 header is also 20 bytes and the Ethernet header/trailer is 26 bytes. If the Ethernet frame size equals the MTU of 1500 bytes, which includes the IP and TCP header, then the payload size must be $1500 - 20 - 20 = 1460$ bytes. Ethernet 은 IP 와 TCP를 포함하고 있음.

$$\%O = \frac{O_{TCP} + O_{IPv4} + O_{Ether}}{O_{TCP} + O_{IPv4} + O_{Ether} + Payload} \times 100 = \frac{20 + 20 + 26}{20 + 20 + 26 + 1460} \times 100 = 4.3\%$$

- When the payload is large, most of the Ethernet frame is payload - so overheads are low and efficiency is high!
- Efficiency is $\nu = 100 - \%O = 95.7\%$

Example 3: calculating overheads and/or efficiency

Example: Suppose the protocol overhead is 25.15% of an Ethernet frame. Calculate the size of the IPv4 header if the payload size is 256 bytes, Ethernet header is 22 bytes, Ethernet trailer is 4 bytes and TCP header is 20 bytes (note: IPv4 headers can be between 20 and 60 bytes in length).

Solution: Again we start with

$$\%O = \frac{O_{TCP} + O_{IPv4} + O_{Ether}}{O_{TCP} + O_{IPv4} + O_{Ether} + Payload} \times 100$$

now, re-arrange to make O_{IPv4} the subject of the formula:

$$\begin{aligned} O_{IPv4} &= \frac{(O_{TCP} + O_{Ether})(\frac{\%O}{100} - 1) + Payload \times \frac{\%O}{100}}{1 - \frac{\%O}{100}} \\ &= \frac{(20 + 26)(\frac{25.15}{100} - 1) + 256 \times \frac{25.15}{100}}{1 - \frac{25.15}{100}} = 40 \text{ bytes} \end{aligned}$$

The domain name system (DNS)

- One of the most important protocols on the Internet today is DNS
- It provides a mechanism for accessing a machine by *name* rather than by *number*
- DNS is a distributed, hierarchical database mapping IP addresses to names:
 - Sometimes you may have multiple names associated with an IP address
 - Sometimes you may have multiple IP addresses associated with a name

DNS - how it works

- If you attempt to establish a connection with **www.uts.edu.au**:
 - Your client software (for example, your web browser) attempts to do a **name lookup** using an operating system call - e.g. on Unix via gethostbyname()
 - Your operating system checks its cache and/or local hosts file to see if it knows the IP address
 - If that fails, it queries the local name server for the IP address of this host

DNS - how it works

- When it receives the query, the DNS server checks its own cache. If the requested name is in the cache, the IP address is sent back to the client immediately
- If not, it checks whether it is the responsible (**authoritative**) server for this domain name. If so, it performs the lookup and returns the IP address to the client.
- If not, it sends the request (randomly) to one of thirteen special global DNS servers called the **Root DNS Servers**.
- These root servers are run by an organisation called **Internic** (Network Solutions), and are hosted in secure locations with massive redundancy.


DNS - how it works

- These root servers know the addresses of all **top-level DNS servers** - e.g. .edu, .com, .au, .info, etc. Each of these is known as a **top-level domain (TLD)**;
- In some cases, the root server might know the answer to the query itself. If so, results will be returned to your local DNS server to be forwarded back to you.

DNS - how it works

- In most cases, the the root servers **delegate** the request to another DNS server which has the responsibility for that TLD;
- When this happens, information about the **next** server to query is sent back to your local DNS server as an **NS record**;
- Your DNS server then re-sends the request to the to the server listed in the NS record sent by the root DNS (one of those responsible for **.au** in our **www.uts.edu.au** example).

DNS - how it works

- The **.au** DNS server checks its records to see who is responsible for **www.uts.edu.au** domain. The relevant NS record is sent back to your local DNS server; 
- If that DNS server doesn't know about **.uts.edu.au** but *does* know about **.edu.au**, then the NS record for **.edu.au** will be sent back to your local DNS server;
- Eventually the DNS server for **.uts.edu.au** domains is contacted and asked for the address of **www.uts.edu.au**. This record is sent to your DNS server, which finally sends it to your client.

DNS - how it works

- What about load balancing?
- Suppose the authoritative server doesn't send the same IP address every time...
- What about virtual hosting?
- There is no problem with having multiple addresses resolve to the same IP address. HTTP 1.1+ includes the requested hostname - so the server can work out “who” it is supposed to be

Reverse DNS lookups

- Reverse DNS lookups (finding the name(s) associated with a particular IP address) are quite similar.
- The lookup is done numerically - your local DNS server knows the IP addresses for which it has names...

Reverse DNS lookups

- If you need a reverse DNS lookup of (say) 216.239.37.99, your DNS server again queries the root server, which has a list of DNS servers for the 216.x.y.z class A subnet.
- Your DNS server forwards the request to one of those, and so on until a DNS server is found which knows the hostname for that address.

Other DNS functions

- Other interesting information can be stored in DNS:
 - For example, mail servers have a special type of record called MX (mail exchange)
 - Others include NS (as you've seen), SOA (start of Authority), RP (responsible person), TXT (some text information about a host) etc.
 - DNS now includes an authentication mechanism for combatting spam emails

The DNS protocol

- DNS preferentially uses UDP - however, if a request via UDP times out, the request is re-tried on TCP. In both cases port 53 is used.
- DNS requests are not text-based. You cannot easily perform a DNS request using a `telnet` client.
- Refer to the RFCs for a detailed description of the packet structure (RFC 1034 and 1035)

The DNS protocol

The basic structure is: header, question, answer, authority, additional

- Header: includes an ID field, Q/A flag, count of questions, answers etc. - and various other flags.
- Question: the text of the query being made or answered, e.g. google.com
- Answers (if any): the IP address(es) corresponding to that hostname, any CNAMEs (canonical names) for the same machine etc. etc.

The DNS protocol

- Authorities (if any): the names of name server(s) responsible for that DNS record
- Additional (if any): for example, the IP addresses of those servers, if known
- To see the details of a DNS lookup taking place, try the command
`dig +trace www.uts.edu.au`

Hypertext transfer protocol (HTTP)

- HTTP is essentially a protocol for file transfer
- HTTP/1.0 uses an individual TCP connection for each file to be transferred - not very efficient if many small files need to be sent. Very obsolete!
- HTTP/1.1 uses a single TCP connection for multiple related transfers, and supports many more options
- HTTP/2 adds pre-emptive server-side pushing of additional expected data to a client plus header compression
- HTTP/3 runs over UDP-based QUIC protocol rather than TCP for reduced latency and higher efficiency; it implements a similar congestion-control algorithm to Google's BBR flavour of TCP (we will discuss TCP and UDP over the next two weeks) and eliminates the head-of-line problem.
- HTTP supports optional use of Transport Layer Security (TLS) for authentication and content encryption (HTTPS).

Example HTTP 1.0 session

- To keep things simple, we will demonstrate an HTTP 1.0 session.
 - Suppose we wish to access one of the web servers in the Mininet lab topology
 - The URL is `http://h3.commprotocols.net/`
 - If we type this into the browser window, we must first do a DNS lookup to obtain the IP address of the server (you will see this in Wireshark).
 - Once you get a response, your browser will ask the operating system to establish a TCP connection to the server's address on Port 80 (HTTP)
 - Your web browser will send a request to the web server over this connection

Example HTTP 1.0 session in telnet

telnet

```
telnet h3.commprotocols.net 80
Trying 10.0.3.1...
Connected to h1.commprotocols.net
Escape character is '^]'.
GET / HTTP/1.0

HTTP/1.1 200 OK
Server: nginx/1.24.0
Date: Wed, 28 Feb 2024 22:53:04 GMT
Content-Type: text/html
Content-Length: 621
Last-Modified: Tue, 10 Sep 2019 05:18:58 GMT
Connection: close
ETag: "5d773242-26d"
Accept-Ranges: bytes


<!DOCTYPE html>
<html>
<head>
<title>Communication Protocols Demo Web Page</title>
...
```

- The client issues the **GET / HTTP/1.0** request (press enter twice) to request the root of the website (/)
- The server responds with **HTTP/1.1 200 OK** to indicate success, followed by some metadata - including the server details, length of the content and when it was last modified
- Finally, the HTML code is sent (in the lab VM you can look at the files in **/var/www/html**)

Other HTTP commands

- HEAD - just request the header, don't send the file (useful for debugging)
- PUT and POST - used for sending data back to the server from the client, such as the content of web forms
- DELETE - remove a file (if allowed)
- TRACE - test to see if a server is alive (should respond with TRACE)
- LINK, UNLINK - associate or disassociate header information with a document

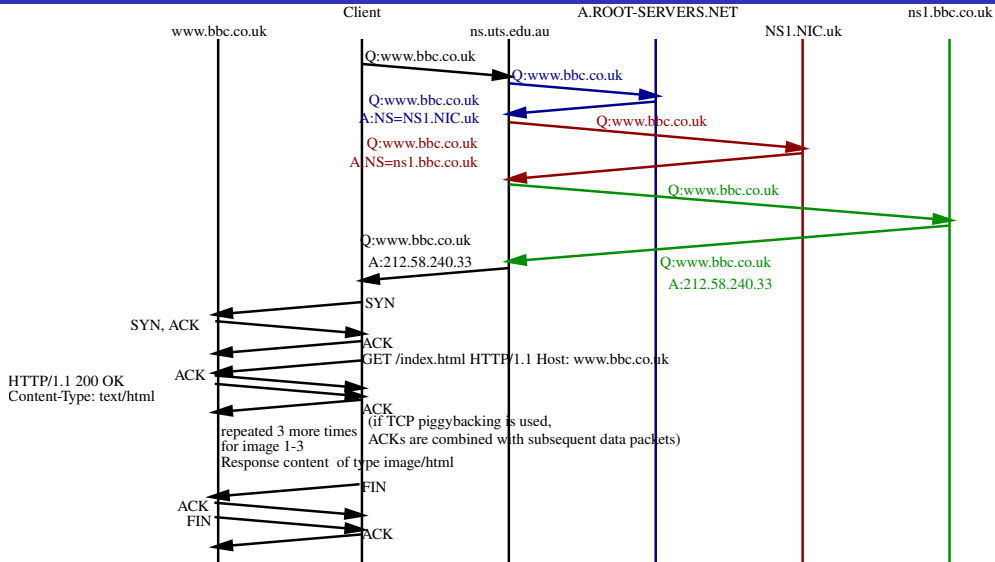
HTTP response codes

- 
- 100-199 are 'information' messages (e.g. 100 Continue)
 - 200-299 are 'success' messages (e.g. 200 OK, 201 Created)
 - 300-399 are 'redirection' messages (e.g. 301 Moved Permanently)
 - 400-499 are 'error' messages (e.g. 404 Not Found, 400 Bad Request)
 - 500-599 are 'server error' messages (e.g. 500 Internal Server Error)

HTTP/1.1

- HTTP/1.1 was developed to overcome a few major design flaws in HTTP/1.0
- New features include:
 - Persistent connections - you can transfer multiple files over a single session (e.g. HTML, CSS, javascript, images etc.)
 - 'Virtual hosts' - i.e., you can host multiple sites on one web server - all using the same IP address
 - Partial downloads of files can be performed - allows resumption of failed downloads.
- Performance has also been improved.
- HTTP/1.1 remains the most common flavour of HTTP.

HTTP + DNS - Example



HTTPS

- HTTP is a plain-text protocol - requests and responses are transmitted in the clear
- This is obviously not very secure - anyone with Wireshark or similar can intercept content (credit card details, usernames/passwords...)
- The the client can't be sure that it is talking to the right server (e.g. `http://www.google.com`) and server can't trust the client
- The solution is to insert a **shim layer** between HTTP and the transport layer - this is called **transport layer security** (TLS)
- TLS uses digital certificates to provide **server authentication** (and optionally **client authentication**), **confidentiality** (data encryption) and **data integrity verification**

Public key cryptography

K **privit**

k **Public**

- We are not going deep into the maths!
- The basic idea: **two keys** are generated by each user, one, K_{priv} is kept **private** and the other, K_{pub} is made **public**. K_{pub} can be freely shared (e.g. included with a message to be transmitted)
- The keys are generated using the RSA algorithm, and have the following important properties:
 - Data encrypted with K_{priv} can only be decrypted with K_{pub}
 - Data encrypted with K_{pub} can only be decrypted with K_{priv}
 - K_{priv} cannot be derived from K_{pub} (and vice versa)

Public key cryptography

- The RSA algorithm is quite slow - about $100\times$ slower compared to a symmetric algorithm such as AES
- **However**, it can be used to securely exchange a symmetric key (a 'shared secret' key), solving the **key distribution problem**
- Now, if A and B want to privately exchange messages, A uses B's public key to encrypt their message, and B uses their private key to decrypt
- Similarly, B can use A's public key to encrypt the reply, and A decrypts it using their private key.

Cryptographic certificates

- Public key cryptography can be used to establish a chain of trust
- The owner of a web server purchases a **digital certificate** from a trusted provider called a **Certification Authority** (CA) who verifies the identity of the owner
- This certificate includes information about the identity of the owner, including their public key, and has been **signed** by the CA.
- A **digital signature** is a cryptographic hash (similar to a checksum) of the content of the document (here, the certificate) which has been encrypted by the signer's private key
- If you have the signer's **public** key (the CA in this case), you can calculate the hash yourself, then decrypt the signature and compare - if they are the same, the signature is valid
- We can therefore trust that the public key belongs to the owner listed in the certificate.

HTTPS (HTTP + TLS)

- Now, when the web browser connects to the server (over a TCP socket, by default on port 443), it can use the server's public key (from the certificate, which it can verify) to encrypt the request
- The request includes a master secret key
- From this, the server uses a **key derivation function** (KDF) to generate a set of four keys to be used in both directions for performing (1) message authentication coding (MAC - used for integrity verification) and (2) symmetric encryption/decryption
- The byte stream is broken into chunks, each of which has a MAC appended - the whole record (message + MAC) is then encrypted using the symmetric key
- To avoid replay attacks, random data (a **nonce**) plus a sequence number is used during calculation of the MAC.
- Results: **authentication**, **confidentiality** and **message integrity**.