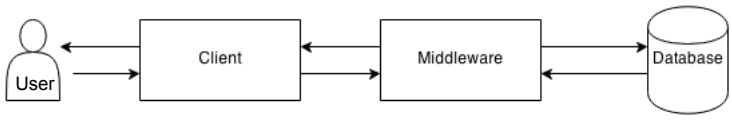
University of Glasgow

Messaging

Web Application Development 2

System Architecture



- We have not yet specified:
  - how messages are going to be sent, and
  - in what format these messages are going to be
- Different interactions will require using different message formats and protocols

At the Application Layer

- We shall be focusing on a protocol for communication and the payload formats:

1. Hypertext Transfer Protocol (HTTP)

- The application protocol used to send messages
- This is a specific URL scheme
- Follows a Request-Response pattern
- Provides a number of ways to make a request (GET, POST)

2. User Agent Specific Protocols (UASP)

- These package/wrap the information that will be sent when providing a response
- Such as XML, XHTML, JSON

MESSAGES

Request – Response Pattern

- Or **Request-Reply** pattern is way to exchange messages
- A requester sends a request message and the receiver of that message provides a message in response
- Typically, this is performed in a synchronous fashion (as in HTTP)
- However, maybe asynchronously (e.g. HTTP/2)

Responses and Requests

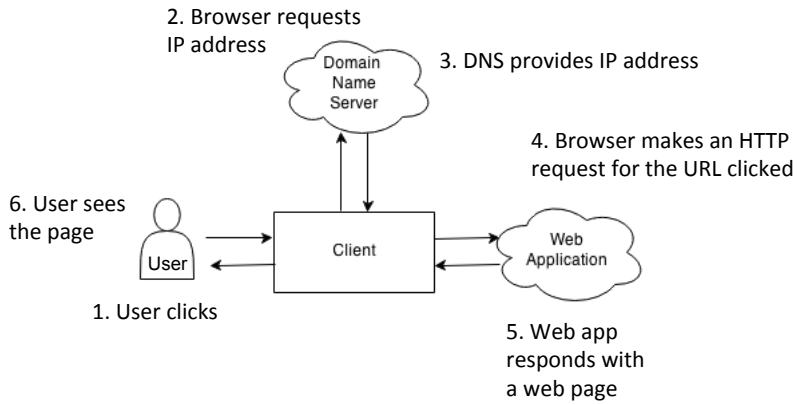
- The following happens when the **user agent**
  - i.e., the web browser/client
- is asked to **send** a **message**,
  - i.e., when the user clicks a link,
- First, the URL is turned into an **IP address**
  - **Request:** ask Domain Name System for IP
  - **Response:** returns the IP for the URL
    - i.e., www.gla.ac.uk maps to 130.209.34.12

Responses and Requests

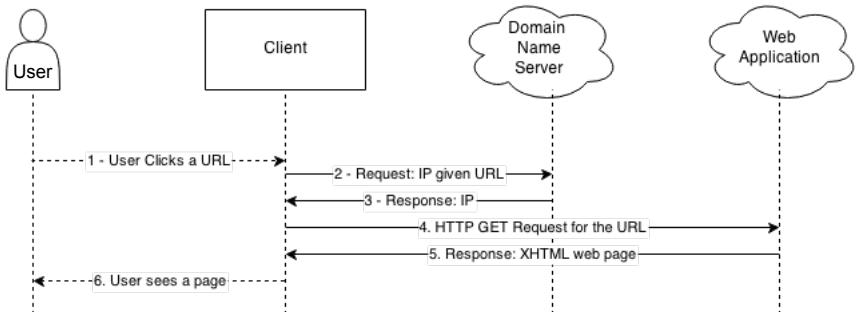
- Second, a **TCP connection** is opened on a particular port on the node at that IP address
  - port 80 for HTTP (standard)
  - port 443 for HTTPS (encrypted through TLS)
- Next, a request is made using a specific URL Scheme (e.g., HTTP) and sent using that TCP connection
  - **Request:** Get the home page of the specified URL
  - **Response:** Returns the XHTML for the home page

For a full ports list see http://www.webopedia.com/quick_ref/portnumbers.asp

Flow of Messages

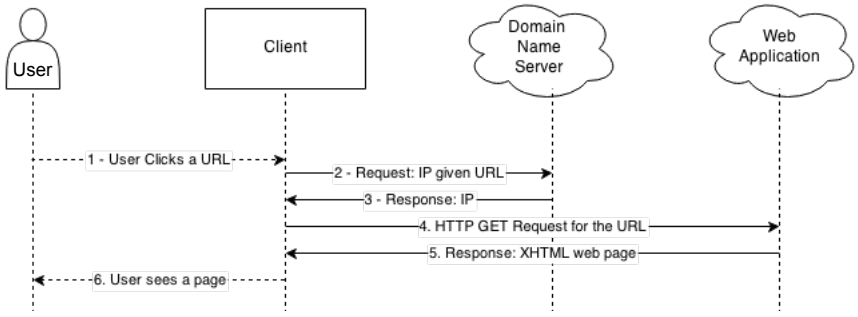


Sequence Diagrams



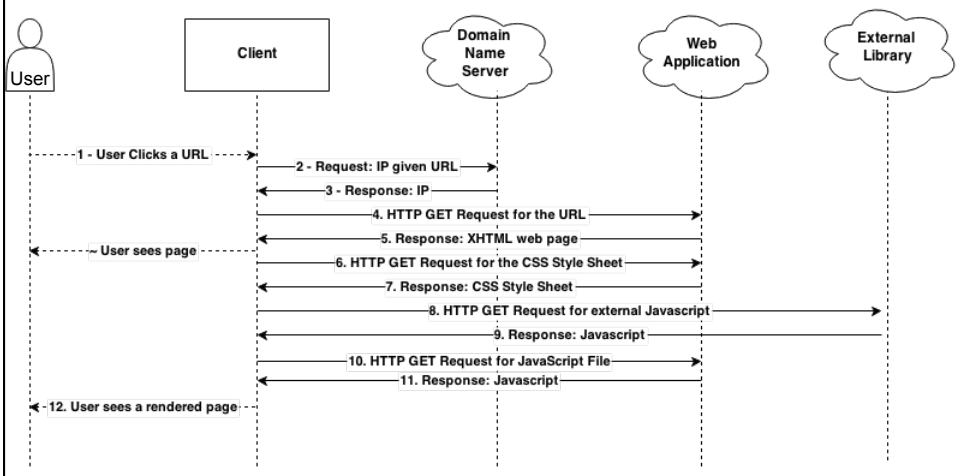
- Sys. Arch. Diagram aggregates over all messages, only showing the flow, but hides a lot of detail
- Sequence Diagrams provide a better way to show the flow of messages
- Label all messages – the more precise the better

Question



What other messages would be sent and received when loading up a web page?

Example



- Further requests for Style Sheets and JavaScript

Communicating with the Database

- Web Apps typically make requests to a Database
- Since we are using Django, requests are made indirectly, through the Object Relational Mapping (ORM)
- The actual request maybe via HTTP or some other protocol
- But we can specify it as a ORM Request and ORM Response

Protocols

- **Requests can be made using various protocols**
  - **http**: this common protocol indicates a file that a web browser can format and display - an HTML file, image file, sound file, etc.
  - **https**: utilises Transport Layer Security for secure communication and always sends data in encrypted form
  - **file**: this indicates a file which is not in a recognised web format and will be displayed as text

Protocols

- **Other protocols include:**
  - **ftp**: file transfer protocol is used to refer to web sites from which files can be extracted and downloaded to the client machines
  - **mailto**: if selected, such a link generates a form in which an e-mail message can be constructed and sent to a designated user
  - **news**: the resource is a news group or article
  - **telnet**: generates a telnet session to this server
- For a full list, see http://www.iana.org/assignments/uri-schemes/uri-schemes.xhtml

HTTP

- **HTTP** or **Hyper Text Transfer Protocol** is used to deliver virtually all files and other data using **8 bit characters**
  - Usually, HTTP takes place through TCP/IP sockets
- HTTP is used to transmit **resources**, not just files
  - A resource is some chunk of information that can be identified by a URI
- HTTP functions as a **request-response pattern** in the **client-server** computing model

Request and Response Messages

- Under HTTP, communication is as follows:
  - An HTTP client opens a connection and sends a **request message** to an HTTP server
    - Typically the request is either a GET or POST
  - The server then returns a **response message**, usually containing the resource that was requested
    - Typically, in XHTML, XML, but also in other formats like JSON
  - After delivering the response, the server **closes the connection**, making HTTP a stateless protocol
    - i.e., not maintaining any connection information between transactions

http GET

- GET appends the data to the URL as key-value pairs as follows:
  - URL ? key1 = value1 & key2 = value2
- Special characters within the values are replaced, e.g. %20 for space
  - this is called **url-encoding**
- The user can see, copy and bookmark a URL, thus it is easy for them to 'resubmit' the page
- Therefore GET should be used for pages which don't change anything on the server
- e.g. it's fine for information requests
  - GET pro.pl?name=John%20Smith&address=5%20Queen%20Street HTTP/1.1
  - GET /base/query/?query=big+java HTTP/1.1

http POST

- **POST sends data packaged as part of the message**
  - must be used for multipart/form-data, e.g., file uploading
  - should be used for programs with side effects
    - e.g., database update, purchase requested, sending email
  - or if there are non-ASCII characters in the data (e.g., accented letters)
  - if the data set is large (GET can have problems with >1kb)
  - you want to hide data from users – although they can always view the source
- POST uses the message body to achieve this and so has the following header lines:

  Content-Type: application/x-www-form-urlencoded
  Content-Length: 26 // number of characters

  and then the body, e.g.:

  name=John%20Smith&address=5%20Queen%20Street
  - NB. that structure (key1 = value1 & key2 = value2 &....) is what *x-www-form-urlencoded* refers to

GET vs. POST

- What is the difference between these methods?
- When would you use one over another?
  - Use GET for safe and idempotent requests
  - Use POST for requests that are not safe or not idempotent
    - A safe operation is an operation which does not change the data requested
    - An idempotent operation is one in which the result will be the same no matter how many times you request it

Other HTTP Methods

- **The http protocol has numerous other methods too:**
  - **HEAD** is just like GET, except it asks the server to return the response headers only
    - useful to check characteristics of a resource without actually downloading it
  - **PUT** for storing data on the server
  - **DELETE** for deleting a resource on the server
  - **OPTIONS** for finding out what the server can do - e.g., switch to secure connections
  - **TRACE** for debugging connections
  - **CONNECT** for establishing a link through a proxy

Stateless Communication

- "HTTP is a stateless protocol"
- HTTP does not require the server to retain any information about the client/user
  - All requests are independent
- This is a problem if we want to maintain a session

Dealing with Statelessness

- **Common Solutions to overcome statelessness**
  - **Client Side**: use HTTP Cookies
    - Cookies are tokens stored on the client and can be included in the request
    - Best to store a session-id in the cookie that the server can use to retrieve information about the user
    - Rather than actual data about the user, etc on their client
  - **Server Side**: with hidden variables when the page is a form
    - i.e., through POSTs
  - **URL encoding:** store a session-id within the URL
    - http://.../doing_task?session_id=unique_session_code