References

http://javabeat.net/introduction-to-comet-and-reverse-ajax/

http://www.ibm.com/developerworks/library/wa-reverseajax1/

http://www.ibm.com/support/knowledgecenter/SSAL2T_8.2.0/com.ibm.cics.tx.doc/concepts/c_properties_websrvces.html

---

Reverse AJAX

Need for Reverse AJAX

The main idea behind Reverse AJAX is to allow servers to push data to clients whenever an event occurs. It bypasses the limitation of a traditional AJAX request which is stateless and must be initiated by the client.

Much more suitable for real time applications where clients must constantly fetch data from the server

Reverse AJAX techniques

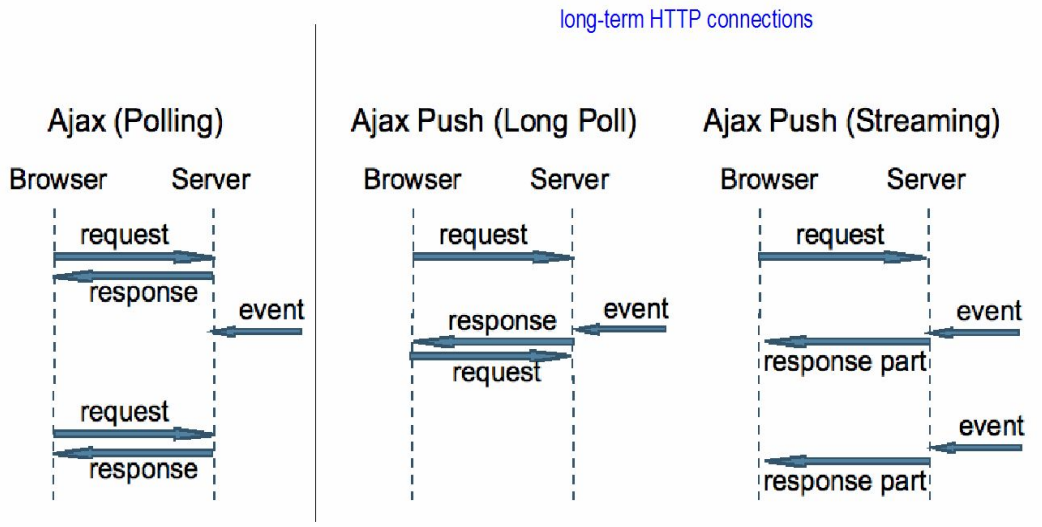
1. **Long polling**: the server makes a request, but the server holds onto the response unless it has an update. As soon as the server has an update, it sends it and then the client can send another request. Disadvantage is the additional header data that needs to be sent back and forth causing additional overhead.

2. **Piggybacking**: a modification to polling - when the client makes a request, the server can respond in two ways: send a normal response, or, send a mixed response.
    a. normal response is sent when the server has no event-related data to send
    b. when an event occurs, then the next request made by the client will see a response from the server containing the normal response as well as the event related response

    The advantage of piggybacking over long polling is it eliminates the need to make unnecessary requests, but the latency is much higher.

3. **Http streams / Comet**: Comet is a web application model where a request is sent to a server and then kept alive for a long time until a time-out or a server event occurs. With

Comet, web servers can send data to client without the client having to explicitly request it.



COMET implementations

1. **Comet using iframes** (with comments)

index.html

```
<!DOCTYPE html>
<html>
	<head>
		<script>
			function init()
			{
				ifr = document.getElementById('ifr');
			}

			obj =
			{
				monitor: function()
				{

					document.getElementsByTagName("button")[0].style.disp
					lay = "none";
					ifr.src = "monitor.php"; // set the iframe's src,
this forces a call to the server
				},

				updateMain: function(str)
				{
					... // update the main window here with some string
```

```
                        },

                        heartbeat: function()
                        {
                               if (timer)
                                      clearTimeout(timer);

                               timer = setTimeout("monitor", 10000);
                        }
                 }
                 ...
          </script>
    </head>
    <body>
          <iframe id='ifr' > </iframe>
          <button onclick="obj.monitor()">Monitor</button>

    </body onload="init()">
</html>
```

monitor.php

```
<?php

    date_default_timezone_set("Asia/Calcutta"); // set the timezone to silence
warnings

    ob_start(); // start output buffering

    $oldtime = filemtime("file.txt"); // use filemtime() to get the last modified
time of the file

    while(1)
    {
          clearstatcache(); // clear the cache
          echo "<script>parent.obj.heartbeat();</script>";
          ob_flush(); flush();

          $newtime = filemtime("file.txt");

          if($newtime > $oldtime)
          {
                $str = ...;
                echo "<script> parent.obj.updateMain(\"$str\"); </script>"; // echo
a script that is sent to the iframe and executed immediately. Each script sent will be
executed
```

```
                    ob_flush(); flush(); // flush the output buffer, ignoring this will
cause the php script to fail
             }
      }
?>
```

The PHP script monitors a file continuously and pushes updates to the client when modifications have been made. But what if there have been no modifications? The server connection will time out. In order to circumvent this, the server sends periodic heartbeats by calling parent.obj.heartbeat() to tell the client it's still alive.

2. Comet using AJAX

This is essentially long polling. The request is kept open for a while until the server sends back data.

index.html

```
<!DOCTYPE html>
<html>
      <head>
            <script>
                  function monitor() // create and send the AJAX request
                  {
                        xhr.onreadystatechange = updateWindow;
                        xhr.open("GET", monitor.php", true);
                        xhr.send();
                  }

                  function updateWindow()
                  {
                        if(readyState == 3 && status == 200) // this code is
executed as long as the request is open and the server is sending back data
                        {
                              if(xhr.responseText.indexOf("Fatal") == -1) // this
is needed to ensure that the data coming back from the server isn't erroneous
                              {
                                    ... // append responseText to the document body
                              }
                        }

                        else if(readyState == 4 && status == 200)
                        {
                              // request has finished, close and restart
                              xhr.abort();
                              monitor();
```

```
                        }
                }
                ...
        </script>
    </head>
    <body>
        <button onclick="monitor()">Monitor</button>

    </body onload="init()">
</html>
```

monitor.php

```
<?php
    date_default_timezone_set("Asia/Calcutta"); // set the timezone to silence
warnings
    clearstatcache(); // to prevent caching of AJAX responses
    ob_start(); // start output buffering
    $oldtime = filemtime("file.txt"); // use filemtime() to get the last modified
time of the file

    while(1)
    {
        clearstatcache(); // clear the cache
        // no need for any heartbeat
        $newtime = filemtime("file.txt");

        if($newtime > $oldtime)
        {
            $str = ...;
            echo "... $str ...";

            ob_flush(); flush(); // flush the output buffer, ignoring this will
cause the php script to fail
        }
    }
?>
```

The issue with using AJAX is that, if an error occurs, the status still comes back as 200. This can be misleading, so we'd need to write extra code to analyse the response and ensure it's not an error.

3. Comet using SSEs

Features / advantages of SSE:

- unidirectional - the server sends data to the client
- pure HTTP - no tedious code required for error checking

- no polling required
- automatic reconnection - circumvent the "30 second rule" of disconnection that apache employs

index.html

```
<!DOCTYPE html>
<html>
	<head>
		<script>
			function monitor() // create and send the AJAX request
			{
				sse = EventSource("monitor.php");
				document.addEventListener("update", updateWindow);
				sse.onerror = function() { alert("Error
occurred..."); }; // error handler
				setTimeout(close, 10000);
			}

			function updateWindow(event)
			{
				... // do something with event.data
			}

			function close() { sse.close(); }

		</script>
	</head>
	<body>
		<button onclick="monitor()">Monitor</button>

	</body onload="init()">
</html>
```

monitor.php

```
<?php
	header("Content-type:text/event-stream"); // this is an EXTREMELY
important header

	date_default_timezone_set("Asia/Calcutta"); // set the timezone to
silence warnings
	ob_start(); // start output buffering
	$oldtime = filemtime("file.txt"); // use filemtime() to get the last
modified time of the file
```

```
	while(1)
	{
		$newtime = filemtime("file.txt");

		if($newtime > $oldtime)
		{
			$str = ...;
			echo "event:update\n";
			echo "data:$str\n\n"; // use double newline char to
signify end of stream
			ob_flush(); flush();
		}
	}
?>
```

Performance of AJAX

Potential issues with AJAX - Client side

1. Request is never sent
   - the browser drops requests

2. Request never returns
   - happens due to timeouts. One solution is to wait for a fixed time interval before aborting and resending the request. With XHR, this can be done using the timeout and ontimeout properties

3. Request returns, but takes too long
   - the server is overloaded with multiple requests

4. Response from the server but an error has occurred
   - permission issues
   - error during processing
   - overloaded
   - malformed data received

If multiple requests are made and the responses are out of order, then a lot of bandwidth is wasted if the ordering was important.

In cases 1 through 3, the reason could be due to the client sending an avalanche of requests in which case either

i) server chokes requests (meaning there's either a timeout or a huge latency)

ii) browser drops requests due to the **2 connection limitation** imposed by HTTP 1.1.

The 2 connection limitation

According to HTTP 1.1; at any point of time there can only be a maximum of 2 simultaneous connections from a single client to a particular domain.

Reason for imposition of the limit

This is done so that clients do not overload servers with too many request (or else the server might classify it as a DDoS attack and block the client altogether).

Solutions to the 2-connection limitation

1. Use a priority queue on the client side so that requests are not dropped by the browser*

2. Use sub-domains, eg; mail.google.com and drive.google.com

3. Split up resources across multiple servers and use Access-Control-* headers

4. Increase the max no. of connections from the browser's side; this is just a hack and not recommended (change param in about:config for Firefox)

5. Send requests in batches; this requires code to pack and unpack requests, not to mention the speed of the slowest request determines the speed of the entire batch. Also, if one request fails, the entire batch fails.

* Code for below

Priority Queue for Multiple Requests (just give this a cursory glance)

prqueue.js

```
function PriorityQueue() { this.items = new Array(); }


PriorityQueue.prototype =

{

	get: function() { return this.items.shift(); },
```

```
	put: function(request) {
		this.items.push(request);
		this.prioritize();
	},

	prioritize: function() // sort based on priority
	{
		this.items.sort(
				function(r1, r2) {
					return r1.priority - r2.priority;
				}
			);
	},

	remove: function(request) // remove a particular request
	{
		for(var i in this.items) {
			if(this.items[i] == request) {
				this.items.splice(i, 1);
			}
		}
	},

	size: function() {
		return this.items.length;
	},

	peek: function(k) { return this.items[k]; },
}
```

reqmanager.js

```
RequestManager = (function()
{
	manager = {
```

```
		active: new Array(), // store currently processed requests, has a limit of 2 at any time
		pending: new PriorityQueue(), // requests that are pending at the moment
		MAX_INTERVAL: 500,
		MAX_AGE: 500,

		sendToServer: function() // check if more requests can be sent
		{
			if(this.active.length < 2) // max no. of connections is 2
			{

				newReq = this.pending.get()
				newReq.xhr = ....
				newReq.xhr.onreadystatechange = processResponse;
				...

				this.active.push(newReq)
			}
		},

		processResponse: function() {
			for(var i in this.active) {
				if(this.active[i].xhr.readyState == 4) {
					if(... .status == 200) {
						// no error
					}
					else {
						// error occurred
					}

					this.active.splice(i, 1); // since the request is completed, remove it
				}
			}
```

```
		},

		sendToPendingQueue: function(request) // convenience function
		{
			this.pending.put(request);
		},

		agePromote: function() // increase the priority of requests that have been in the queue for a long time
		{
			for(var i in this.pending) {
				curReq = this.pending[i];
				curReq.age += this.MAX_AGE;

				if(curReq.age > this.MAX_AGE) {
					curReq.age = 0;
					curReq.priority--;
					this.pending.prioritize();
				}
			}
		}
	}

	setInterval(this.agePromote, this.MAX_INTERVAL);
	setInterval(this.sendToServer, this.MAX_INTERVAL);

	return manager;
})();
```

reqclient.js

```
for(var i = 0; i < 10; i++) {
	newRequest =
	{
		url: ...,
```

```
            method: ...,
            age: ...,
            xhr: ...,
            data: ...,
            priority: ...
        };

        RequestManager.sendToServer(newRequest)
}
```

Potential issues with AJAX - Server side

1. **Racing**: If multiple AJAX requests are made which involve manipulating the superglobal $_SESSION, then this could lead to racing. Luckily, PHP has built in locking mechanisms

2. **Server availability**; **and use of the HEAD call:**
   If a request is to be made with a lot of data, it is better to ensure the server is offline before making such an expensive request. This is accomplished by sending a simple HEAD call to the server. Server responds with headers, not data (a simple heartbeat).

3. **Client availability**: similar to point 2; this can also be solved in the same request.

Content Optimization

Http Compression

When data to be sent from server is too large to be cached, the best method is compression.

The browser tells the server which formats it can understand using the Accept-Encoding request header. Eg;

Accept-Encoding: gzip, deflate

The server tells the browser what the format of the data is using the Content-Encoding header. Eg;

Content-Encoding: deflate

Some terminology

1. TTFB - Time to First Byte

The interval between the time the request is made, and the time the first byte of the response arrives

2. TTLB - Time to Last Byte

The interval between the time the request is made, and the time the last byte of the response arrives

Code example

compress.html

```
...
<script type="text/javascript" src="jquery.js"></script>
<script>
	function getData()
	{
		$.post("getfile.php", showContents); // jquery
	}

	function showContents()
	{
		... // display contents
	}
</script>
...

<body>
	<button onclick="getData()"> Get Data </button>
</body>
```

compress.php

```
<?php
	header("Content-Encoding:deflate"); // very important to specify the compression
type
	// header("Content-Encoding:gzip")

	$contents = file_get_contents("data.txt");

	$contents = gzdeflate($contents, 9); // 9 is the highest level of compression
	// $contents = gzencode($contents, 9);

	echo $contents;
>?
```

By carrying out compression, there will be a small delay before the client receives the first byte, so TTFB will be longer. However, since the amount of compressed data being transferred is so little compared to before, the TTLB will be much smaller.

If the same data is being accessed by multiple users, it would be better to store the pre-compressed version so that the TTFB is also significantly reduced.

Caching

AJAX calls by default are not cached by the browser. But we can specify caching by setting either the Expires (HTTP 1.0) or Cache-control (HTTP 1.1) response headers.

i) In the case of "**Expires"**, you specify a date in the form

D, d M Y H:i:s

Eg; Sun, 19 Oct 2014 14:12:12 GMT

Browsers might sometimes keep the cached copy even after expiry. This can be prevented by sending a get request using a random parameter in the URL.

Eg; http://website.com?randomstr=timestamp

Since the timestamp always changes, the browser is forced to go to the server.

```
<?php
      $exprtime = date("D, d M Y H:i:s", time() + 10);
      header("Expires: $exprtime");
      ...
?>
```

ii) In the case of "**Cache-control"**, we specify the parameter "max-age" which is the no. of seconds for which the cached copy is valid.

```
<?php
      $time = 10;
      header("Cache-control: max-age=$time");
      ...
?>
```

Note: these headers will result in only GET requests getting cached. POST requests are never cached. You'd need to cache it manually using localStorage.

Other points on caching

1. Use external JS scripts whenever possible. This is because external scripts can be cached. This significantly reduces loading times in the future.

2. Manually cache data wherever possible. This way, dependencies on the browser are reduced. Check to see if localStorage has a cached copy of some data before making requests for it

Miscellaneous

1. GET vs POST: In all browsers except Firefox, POST is a two step process. First, the headers are sent, followed by the data. GET, on the other hand, uses only one TCP packet. Since POST is more expensive, GET should be used wherever possible.

2. When using images, so not set the src attribute unless required. Even setting src="" (empty attribute) wastes a call.

3. Code minification:
    - remove white spaces, comments, etc
    - replace long winded variables names with one/two letter names
    - change colour names to hex-code
    - minimize DOM access; store them in variables and use

Web Services

Software that allows interoperability of machines over a network. The two types of web services are:

1. **RESTful** - based on REST (Representational State Transfer)
2. **WS-*** - based on SOAP (Simple Object Access protocol)

Need for web services

- faster product development - we don't keep reinventing the wheel
- cross platform accessible (platform independent, it is built on top of HTTP)

Web services vs websites

1. Web services are used by other programs; websites are used by human beings
2. Websites aim to provide and present information; web services have no concept of presentation
3. Web services return data in XML or JSON format; websites usually return content in HTML (or is converted to HTML)

Note: Webapps are a more intelligent form of websites, as they can distinguish between clients (not stateless).

Properties of a web service

1. **Self-contained**: no additional software required on the client side

2. **Self-describing**: a WSDL (Web Service Description Language) file provides all the info needed to implement/invoke the service
3. **Discoverable**: clients can easily discover and invoke the service
4. **Cross platform**: they can be implemented in different environments
5. **Inherently open and standards based**: XML and HTTP are the major technical foundations for web services
6. **Modular**: simpler web services can be aggregated to form more complex ones

SOAP based WS

XML-based messaging protocol. It defines a set of rules for structuring messages that can be used for simple one-way messaging, but is particularly useful for performing RPC style request-response dialogues.

3 main components of SOAP

1. **SOAP** (Simple Object Access Protocol); it is XML based

2. **WSDL** (Web Services Description language); it is also XML based, it is used to locate and describe web services

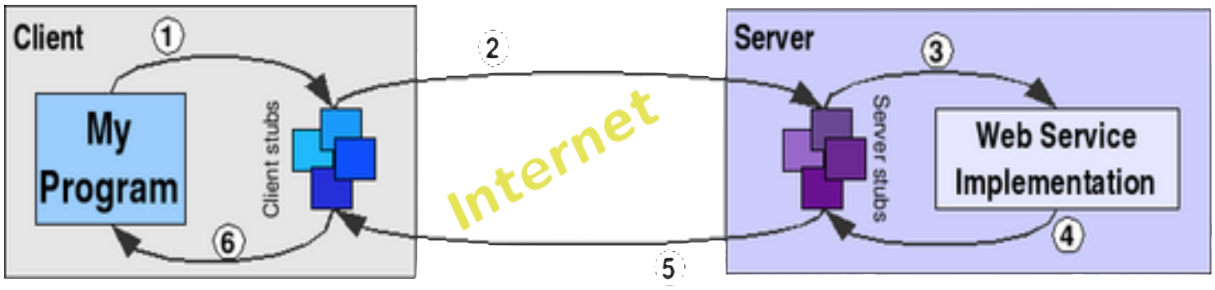
3. **UDDI** (Universal Description, Discovery and Integration); it is a directory service where companies can register and search for web services.

Working

1. Web service "publishes" a service to the UDDI
2. Client searches for web service in the UDDI
3. UDDI returns WSDL
4. Client uses WSDL to Bind to the service provider; provider returns a string representation of an object (called "serialization" or "marshalling")
5. RPC dialogue, which is translated to SOAP requests-response using stubs.

Necessity of stubs

Required to interpret SOAP requests and generate SOAP responses ([de]serialization of objects).



RESTful WS

REST stands for Representational State Transfer, it is an architectural model based on some constraints. In this model, there are "resources", "representations" of those resources and "connectors" connecting resources to those representations.

Constraints

1. Client-server architecture
2. Stateless
3. Client must have the ability to cache responses
4. Layered architecture
5. All modules must have a "uniform interface" so that they can evolve independently.

Since the WWW satisfies these constraints, we can conclude that the WWW is also RESTful.

Working

- The URI of the web service acts as the object and it supports CRUD (Create; Read; Update; Delete). A single PHP script usually carries out CRUD.
- 4 URIs are provided, one for each option. These URIs are mapped to a single PHP script which then does the job. Return format of data is always JSON or XML.
- The representation is the only view the client has of the resource. Client can manipulate the resource through its representation.
- The URI -> PHP script mapping is done by the server's config file (.htaccess).

.htaccess

```
...
RewriteEngine On
RewriteRule ^get/usn/(\d\d\d)\.(json) usnproc.php?usn=$1&ret=$2
RewriteRule ^update usnproc.php
```

usnclient.php

```
<?php
	/*
	Why is this file needed? Because, we can't directly make PUT requests to usnproc.php, so we send a GET request to usnclient with data we want to 'put', and usnclient.php builds the PUT query and sends it to usnproc.php
	*/

	extract($_GET);
```

```
	$data = array("usn" => $usn, "grd" => $grd);

	// Build the query string
	$data = http_build_query($data);

	// Build the header fields
	$header = array("Content-type:application/x-www-form-urlencoded");

	// Build the request object
	$request = array(
		"http"=>array(
			"method"=>"PUT",
			"header"=>$header,
			"content"=>$data,
		)
	);

	// One more step before we actually fire the request
	$context = stream_context_create($req);

	// file_get_contents(url, flag, context)
	$retval = file_get_contents("http://localhost/priv/updt", false,
$context);

	echo $retval;
?>
```

usnproc.php

```
<?php

	$method = $_SERVER["REQUEST_METHOD"]; // use this variable to get the
CRUD operation

	if($method == "GET")
	{
		extract($_GET);

		response = array();
		// open a file and look for the usn matching $usn and return it
		response['name'] = ...;
		response['gpa'] = ...;

		echo json_encode(response);
```

```
	}

	else if($method == "PUT")
	{
		$data = file_get_contents("php://input"); // returns http query
string
		$data = explode("&", $data);
		... // do something with $data

		echo json_encode(response);
	}
?>
```

Comparing SOAP and REST

1. RESTful services are easier to implement; SOAP is much more complex to implement

2. REST messages are shorter than SOAP (and hence take up less bandwidth)

3. REST messages are more human readable than SOAP

Web Security

Possible risks

1. Phishing
2. Man-in-the-middle; packet sniffing, session hijacking, **heartbleed**
3. DDoS attacks
4. DNS poisoning
5. SQL injection
6. XSS (Cross Server Scripting)
7. CSRF (Cross Server Request Forgery)
8. Expired certificates

HTTP Authorization

Two types of authorization

1. **Basic**
   - unencrypted **base64 encoding**
2. **Digest**
   - encrypted by hashing (username, pwd, server supplied nonce value, HTTP method, URI)
   - much safer, but much more complicated

Popular auth method
Basic + HTTPS (secure socket layer)

HTTP status 401
FORBIDDEN - restrict access to protected pages and data

CSRF

Malicious websites attempt to carry out unwanted actions on trusted websites that the user has been authenticated for.

Solution is to directly enter links into the browser rather than click on them.

SQL Injection

Sending carefully engineered data to break the server and leak information from the database. Happens when servers do not sanitize their inputs.

Solutions
1. Sanitize inputs
2. Do not run more than 1 query at any time
3. Return vague errors only, do not give hackers any indication of what is happening behind the scenes
4. Do not run queries in privileged mode
5. Escape quotes, so that hackers cannot manipulate query strings

XSS

Vulnerability on the client side; malicious links can prevent default operation and inject scripts to read cookies.

Solution is to check on the server side for any malicious scripts that were sent along with the data.
On the client side, directly enter links into the browser rather than click on them.

Heartbleed

Vulnerability in **Heartbeat** module of OpenSSL 1.0.1. Sending (string, length) which is echoed back by the server.

Eg; Send ('hi', 2)
Recv 'hi'
Eg; Send ('hi', 500)
Recv 'hi ……. (and a lot of other sensitive data that is in memory) …

Happens because the server reads the length, calls malloc() and strcpy and then just returns the string.

Solution is to pass the string and call strlen() to get the length.

Semantic Web

Syntactic web

Information representation is carried out by computers. However, the interpretation and identification of relevant information is left to the user.

Some issues with the syntactic web

- HTML is format centric - the tags do not carry any semantic information
- lack of metadata - a lot of websites do not contain information about themselves
- low "recall precision" - pages with matching words may be returned by a good search engine, but these pages may just mention the words in passing; the match may not be strong.
- search results are sensitive to the vocabulary used. For example, a search for "TCP/IP" and "Protocol" would omit results where TCP/IP is described as a standard
- search results appear as a list of references to individual pages. In reality, these pages may belong to the same website.

Semantic web

Computers try to interpret information to an extent before presenting it.
According to W3C, "*it provides a common framework that allows data to be shared and reused across application, enterprise and community boundaries.*"

Main theme of the semantic web

1. **Concepts** - metadata, ontologies, web services
2. **Application** - semantic web apps
3. **Technology** - tools for ontology development

Note: **Ontology** is a formal definition of a term.

List of Topics

REVERSE AJAX: (------DONE------)
Need for Reverse AJAX - typical use cases.
Http Streaming vs Long Poll, timing diagrams.

Comet - using Hidden Frames - the need for heartbeat() function here.
- using XHR (the use of readyState == 3)
- using server-sent events (SSE).
- The advantages of using SSEs.
- The problem with simultaneous requests (http 1.1 restriction)due to Comet calls.

AJAX - PERFORMANCE: (------DONE------)

The "2 simultaneous requests only" limitation with http 1.1.
The reason for the limitation.
Impact on performance on client and server sides - possible workarounds
- Maintaining priority Queues,
- increasing connections at browser end
- using sub-domains.
- using separate servers with Access-control headers.
- packing requests into one.

(------DONE------)

{

Status code 200 does not mean all is ok. Why??
Race conditions on the server.
Use of the "HEAD" call.
Use of Compression techniques. TTFB, TTLB. Minimising TTFB.
Caching: The use of Cache-control and Expires headers. Difference between the two.
GET vs POST requests to save performance.
Techniques to reduce size of markup/javascript being transferred.
Minifying code.
Using external Javascripts.
Misuse of the "src" attribute in <img> tags

}

WEB SERVICES: (------DONE------)

Web Services vs Web Sites.
Properties of a Web Service - discoverable, self-describing etc.
Types of Web Services - RESTful and SOAP-based.
How SOAP-based services work. UDDI, SOAP and WSDL.
REST - meaning and architecture.
REST constraints.
RESTful APIs.
Standard guidelines for designing RESTful services.
JSON vs XML as data choice.
RESTful vs SOAP based services - Advantages and disadvantages.

SECURITY: (------DONE------)

- Restricting data access using HTTP Authentication.
- The configuration directives in Apache.
- The http status 401.
- Types of Web attacks-
  - Phishing
  - Man-in-the-middle, session-hijacking.
  - SQL Injection.
  - Cross Site Scripting - XSS. Variations (Permanent, reflection)
  - CSRF - Cross Site Request Forgery.
  - Difference between XSS and CSRF.
  - General countermeasures/precautions for protection.
  - The "Heartbleed" vulnerability
  - (SSL/https excluded for now)

SEMATIC WEB: (------DONE------)

- Differences between Syntactic Web and Semantic Web.
- Basic Working of the Semantic Web.