# AJAX PERFORMANCE: GENERAL NOTES BASED ON CLASSROOM DISCUSSIONS.

**Potential problem with AJAX applications - Requests.**

1. Request never returns. Why? (See point 5)

a. Timeouts - Solution is to wait for a fixed interval before aborting the request or probably trying at a later point of time.

Implementation? ( Setup a timeout function for the timeout period. If the response is received within the timeout, cancel the function. Otherwise the function will fire anyway after the timeout interval).

Otherwise use the timeout and the ontimeout properties on the XHR object. The timeout property takes the time in milliseconds after which the XHR request is deemed to be timedout. The ontimeout property refers to a function which fires when the request times out.

2. Request returns too slowly.(See point 5). Detailed solution in later parts of this document.

3. Response returns but an error has occurred.

a) Server may have permission probs.

b) Server is overloaded and sent back text indicating the same.

c) Web Server may throw errors.

d) Server returns malformed data.

If multiple requests were made and the order of the responses is important, then things get very complicated. What if responses arrive out of order. Whole lot of bandwidth is wasted because the data makes no sense now.

4. Request is not made at all (In spite of a call by the client. The browser plays party-crasher. See point 5)

5. An avalanche of requests is made by a client (infact by all clients)

a) Server chokes (in which case the responses are very slow or they timeout).

b) Each client (browser) decides that there are too many in its "queue" and therefore silently drops some requests. This is because of the 2-connection limit imposed by http 1.1.

**HTTP 1.1 says: At any point in time, there can only be 2 simultaneous open requests from each single client to a particular domain.**

**Browsers do not obey this strictly. For example, firefox, chrome have the limit set at 6.**

Solution:

a) The client must decide which of the requests is most important. Also, the client must have its own queue so that requests are not dropped by the browser. We build a practical solution to this. Check the request manager example we did in class. (prqueue.js, prqueue.html, reqmanager.js and reqclient.js)

(if order of responses is important, this can still cause prbs. We will then need to add an additional property called “depends” to each JSON request object and do logic on that)

b) Use sub-domains on the server.

c) Split up resources over many servers and use the "Access-control-allow-origin" header. But server side cost starts going up.

d) Increase max connections from browser side, but we are just mimic-ing the fly-over method of traffic control in Bangalore. In firefox this can be done by typing “about:config” in the address bar and then modifying the **network.http.max-persistent-connections-per-server** field.

e) Send Requests in batches. This will require code to pack and unpack requests. Disadvantages? (The speed of the slowest request determines the speed of the batch)

**Other Performance issues:**

1. If server script encounters an error, some server environments return a 200 as status. So merely checking for 200 and readyState == 4 may not be enough at all. Also, if a server script times out while executing (maybe it got caught accidentally in an infinite loop), 200 can be returned.

One of the simplest ways of introducing the first level of error check is to check for the MIME type of the returned value. If it is not what we expect, then we might want to take alternate action. Otherwise we have to analyze data a little bit to make sure it is ok, before using it completely.

2. Order of responses is not guaranteed. Order of requests may be what you want, but the server may take varying times in order to process these requests. "Out of order" responses may be meaningless.

We need a priority based queue. Priorities must be assigned after proper thinking. But even then this can cause problem with the last two requests, because for the browser both these are equal priority. Hence it is better to assign dependencies to requests and then perform logic when out of order responses are received.

**Server side problems:**

1. Race conditions. Example $\_SESSION php variable. Luckily PHP has inbuilt session-locking mechanism. But the $\_SESSION is one per web server. If more than one web-server is used, then the usual thread-safe programming must be done by the developer. (locking etc)

2. Server availability - Check if the server is actually up before making the request. This is crucial if you are sending a lot of data with a request. You want to be sure the server is ready to take it. Otherwise it can be a waste of bandwidth.

Solution: This could be a done with a simple HEAD call to save bandwidth. The server responds with a heartbeat (only the header part, body is not sent). Frequency of the HEAD call is a crucial factor.

3. Client availability - This is crucial to save resources on the server. For instance, the server may have fetched huge amounts of data from database and kept it in its memory. The question is: How long should the server keep this data in its memory? What if the browser has been closed by the user? Or what if the user is simply inactive? Hence the client should also let the server know that it is alive so that the server keeps the data in memory.

(Both 2 and 3 can be accomplished with one request. Otherwise the server may be wasting a lot of resources.)

**Patterns:**

Be careful with Periodic Refresh, Multi-stage downloads and predictive fetches. With periodic refresh, the frequency value is vital for good performance. In predictive fetches, wrong predictions can end up fetching a lot of data which may be useless later on.

**CONTENT OPTIMIZATION:**

Apart from saving bandwidth by making calls judiciously, we must also optimize content (payload) that is transferred back by the server.

**Remember: Send very little, less often.**

**Http Compression**

1. We can use compression techniques like gzip or deflate so that the text payload can be significantly reduced.

2. The "Accept-encoding" header can be used by the client to specify whether it can accept compressed data. Most browsers accept gzip and deflate.

3.

**Markup/CSS/JS Optimization:**

This can also make your code unreadable (atleast for a start :) and can discourage potential "copy masters")

a. Mark up:

* Remove indentation, unnecessary comments.
* Remap color values appropriately - Big words can use hex codes while small words can be retained.
* Be careful with removing quotes around attributes, short-closing some tags (dangerous with <script>, <textarea>)

b. CSS Optimizations:

* Remove all whitespaces, Comments
* Remap colors to shortest values.
* Use Short-hand notations as much as possible

(ex: instead of saying “font-size: 26px; font-family: Arial; font-weight: bold;” you can say

“font: bold 26px Arial;”

c. Javascript Optimizations.

* A single line can have the entire script. Therefore you normally have two versions of the file. One for the developer and one for the end-user (who need not understand the code at all)
* Remove all comments. They are useless to the end user.
* Remove all whitespaces. Be careful with ';' character.
* Perform code optimizations. (x=x+1 can be replaced with x++;)
* You can replace meaningful variables with s, x, y etc using a tool and then deliver that to the user.
* Remap built-in objects to save space.

ex: w=window, n=navigator ap = appname;

if(w.n.ap == "Netscape")

{

}

* Minimise DOM access as much as you can. Try to store DOM elements in variables.

d. Use external Javascripts. The first download will be a little slow. But the browser can cache the script and if requested by subsequent pages, can use the cached copy of the script. Use need-based script downloads.

**OTHER POINTS:**

1. Choose the right format for downloading data. (XML/JSON/Text)
2. If you are using images, do not set the "src" attribute until it is required. **Merely setting the src to an empty string also wastes a call.**

The following is very bad.....

**<img src=""/>** and later in code, you do **img.src =** [**http://somesrvr/img.jpg**](http://somesrvr/img.jpg)**;**

1. Try to use "GET" as the method as often as you can. Experiments have shown that POST is actually a two step process (headers are sent first and then the body) unlike GET which takes only ONE TCP packet. The YAHOO Mail team found that all browsers except Firefox perform POST as a two-step process – headers are sent first, server responds with 100 (continue) and then the body is sent by the browser. Firefox performs POST in one step.
2. Effective use of caching:

a) Predictive Fetch pattern needs to be implemented judiciously. (Can prefetch and cache both on client and server side).

b) External Javascripts can be cached. This will slow down the first download/execution but the browser can cache downloaded JS and subsequent page downloads (which use the same JS) will be significantly faster because the JS files are already there with the browser.

c) Use the “Expires” header judiciously to cache information. (But this means the browser can arbitrarily choose to persist the data even after the expiry date.). Hence you will need to use the URL manipulating method to force a new fetch for the cached content. The “Expires” header is very effective in caching static data received through AJAX.

Alternately, use the “Cache-control” header (http 1.1). This is much better than the “Expires” header.

d) POST responses are NOT cached in spite of the "Expires" header being set by the server (Or for that matter the Cache-control header). You have to MANUALLY cache the POST responses.

e) Manually cache wherever possible. This way, dependency on browser is much reduced (cross browser issues are eliminated too). Before you make new requests, check for cached data in the browser’s memory and use it if you can.