Writing effective asynchronous XmlHttpRequests

Thibaud Lopez Schneider
thibaud_lopez@yahoo.com
May 27, 2008

The XmlHttpRequest object is very popular in web development as it allows execution of all sorts of functionality in the background of web pages. The Internet is full of tutorials and examples that illustrate how to implement it, and it is being standardized by the W3C.

However, when it comes to assemble a large indefinite number of XmlHttpRequests, in intricate ways like in conditional statements or in loops, and in synchronous mode mixed with asynchronous mode, then the tutorials and examples fall short, and web developers are left alone to their own trials and errors.

This document proposes simple rules to effectively transform code from synchronous XmlHttpRequests to asynchronous. This document is important where stability, quality and usability are important. The result could also be integrated in AJAX tools.

Table of contents

Definitions

Rules

Single request

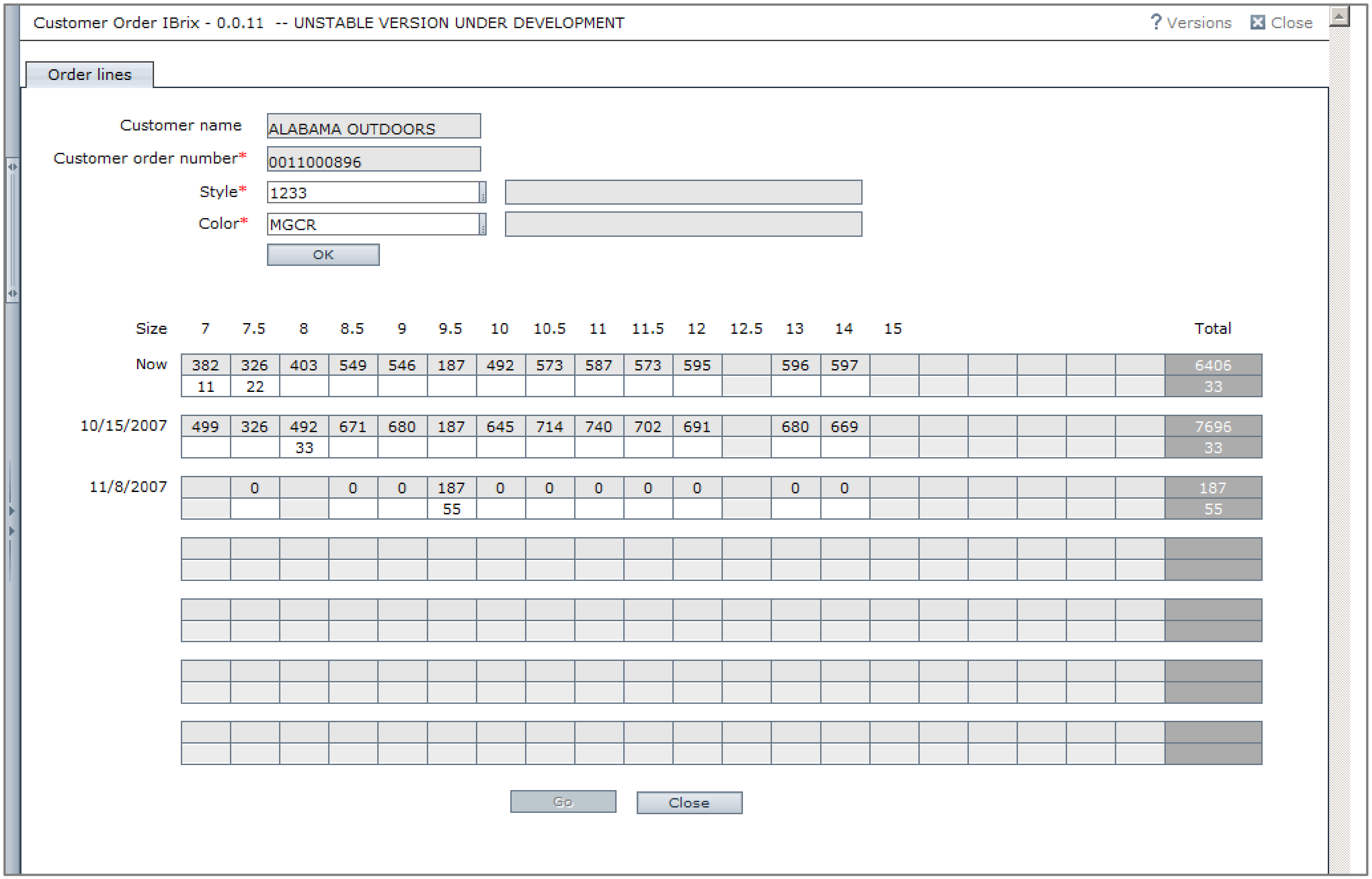
Multiple requests

Conditional statements

Loop statements

Dilemma

How do I send 100 XmlHttpRequests?



Dilemma

How do I send 100 XmlHttpRequests?

- Let's suppose I have a shopping cart with something like 100 items, let's suppose I only have the possibility to write my business logic on the client-side (not on the server-side), and let's suppose I will process each item of the shopping cart with one XmlHttpRequest. Now, how do I send 100 XmlHttpRequests? Remember we are in a fictitious case were we can only write that business logic on the client-side.

- First, I wrote one request to process one item from the shopping cart. Then, I enclosed that request in a loop to process the entire shopping cart. It was easy and it worked. But because the requests were synchronous the browser hung for several minutes while the loop was processing. Meanwhile, the users saw a blank screen, killed the browser and tried again, resulting in user frustration and double requests.

- So I changed the request to asynchronous. Now the browser sent all the requests at almost the same time. Unfortunately, the ERP couldn't process more than one order at a time, it accepted the first request, locked the database, and rejected the other requests.

- Then I started asking myself tricky questions like how do I create a loop of asynchronous requests knowing that at each iteration I have to wait for the response of the previous request? And what happens if one of the requests fails?

- As I couldn't find any help on the Internet I set myself to write this document and make it public.

Anatomy of one XmlHttpRequest

1. Synchronous request

Definition

- An XmlHttpRequest can be **synchronous**. The browser sends the request, pauses the code, waits until it receives the response or until the connection times out, and then resumes the code.
- Synchronous requests are the most natural and easy to code. They are preferred when high readability and maintainability are important.
- But during that process the browser freezes (or hangs) resulting in a blank screen. It’s not even possible to give feedback to the user (indicate activity, show progress). So the user may kill the browser and try again, resulting in user frustration and double requests. That’s poor usability.

1. Synchronous request

Example

```
// send the request
var request = createXmlHttpRequest();
request.open("GET", "page.html", false);
request.setRequestHeader("Content-Type", "text/plain");
request.send();

// do something with the response
request.responseText;
```

Design pattern 1

1. Synchronous request

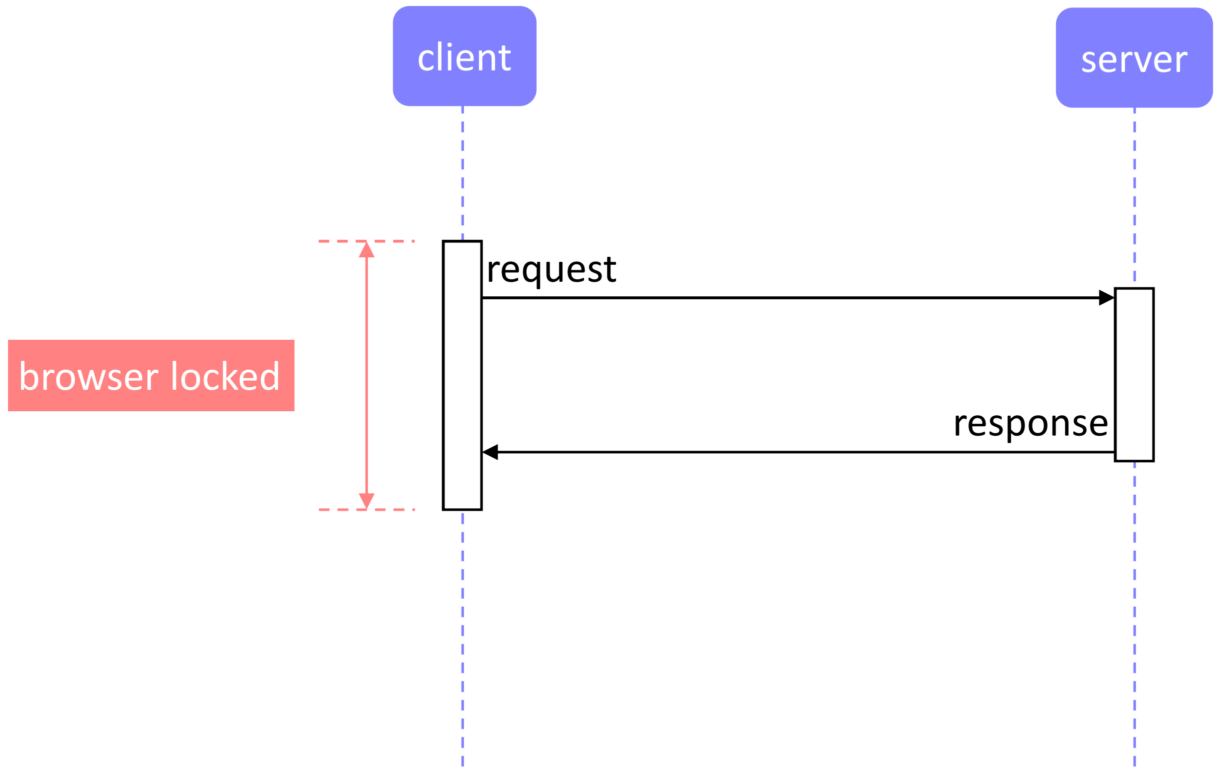
Pseudo-code

```
function f() {
    request
    response
}
```

```
function f() {
    pre-request
    request
    something
    response
    post-response
}
```

Variation

1. Synchronous request
Sequence diagram



2. Asynchronous request
Definition

- On the other hand, an XmlHttpRequest can be **asynchronous**. The browser sends the request, and continues to execute the code. Then, when the browser receives the response it fires an event that is caught by an event handler.
- The browser does not freeze so it's possible to give feedback to the user (indicate activity, show progress).
- But as the caller and the event handler are in two separate functions it affects return values (cannot return values to the caller), local variable accessors (response doesn't have access to request's local variables unless if closure or transferred), looping (the post-loop of asynchronous requests is executed before the responses which can result in unexpected behavior), exceptions, and branching statements. So the code must be adapted and it's less easy to write and read, mostly if there are multiple intricate requests.

2. Asynchronous request

Example

```
// send the request
var request = createXmlHttpRequest();
request.open("GET", "page.html", true);
request.onreadystatechange = handler;
request.send();

function handler() {
    if (request.readyState == 4 && request.status == 200) {
        // do something with the response
        request.responseText;
    } else {
        // do something else
    }
}
```

Design pattern 2

2. Asynchronous request
Pseudo-code

```
function f() {
   request
<- - - - - - - - - ->
   response
}
```

Split the synchronous (Design pattern 1) to get the asynchronous (Design pattern 2)

Rule #1

→

```
function f() {
   request
}

function f'() {
   response
}
```

2. Asynchronous request

Split

```
function f() {
    pre-request
    request
    something
    response
    post-response
}
```

Design pattern 1

```
function f() {
    pre-request
    request
}

function f'() {
    something
    response
    post-response
}
```

Correct split

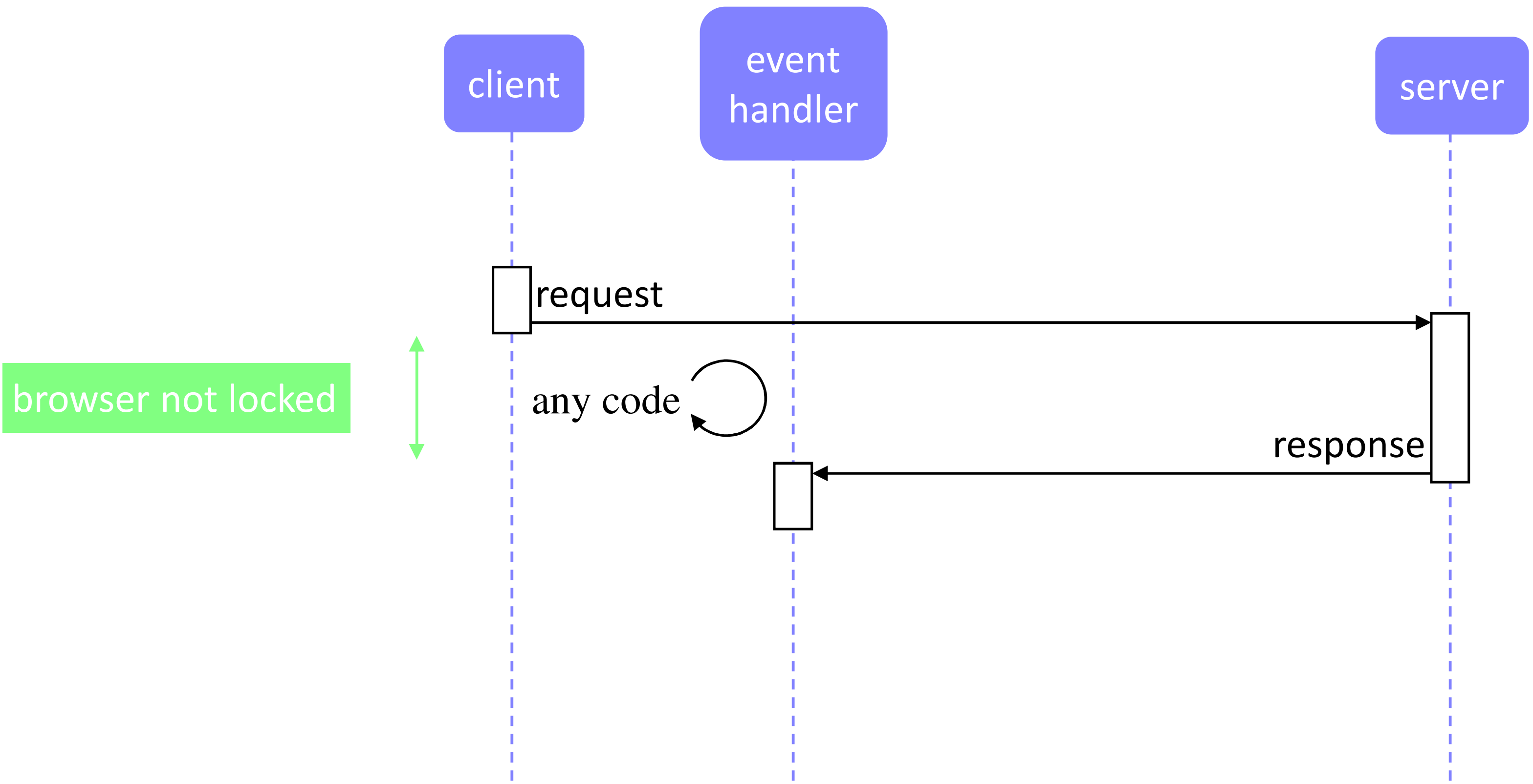
```
function f() {
    pre-request
    request
    something
}

function f'() {
    response
    post-response
}
```

Incorrect split

2. Asynchronous request

Sequence diagram



Anatomy of several XmlHttpRequests

Serial requests
Definition

- We can write code to send multiple requests in **serial**. The browser sends a request and receives the response. Only when it received the response it sends another request and receives another response, etc., until all the requests are sent.
- Serial XmlHttpRequests can be synchronous or asynchronous. If they are asynchronous the code must be carefully crafted.
- Serial requests are the most natural to code as they mimic sequential programming.
- Serial is necessary when a request depends on the response of a previous request, or when the server cannot handle more than one request at a time.
- But serial requests don't benefit from the parallel processing abilities of servers.

Design pattern 3

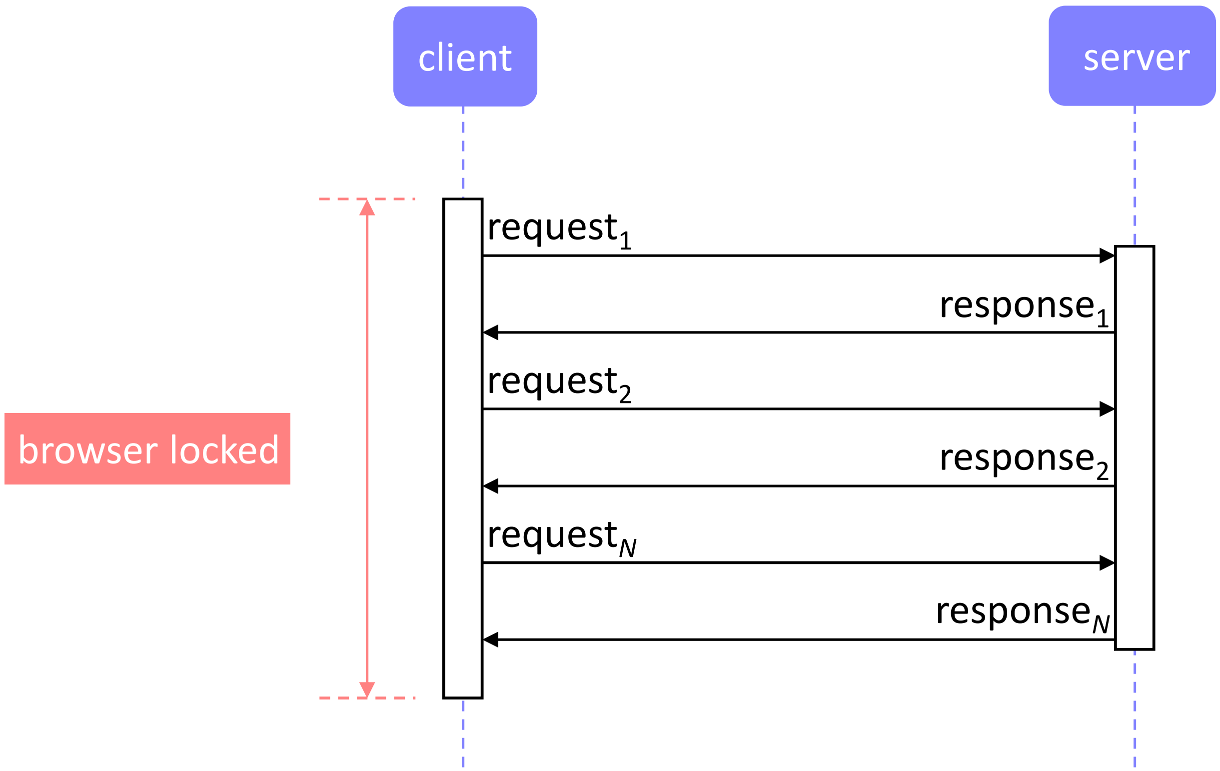
3. Serial synchronous requests

Pseudo-code

```
function f() {
    pre-requests
    request₁
    response₁
    request₂
    response₂
    …
    requestN
    responseN
    post-requests
}
```

3. Serial synchronous requests

Sequence diagram



Design pattern 4

4. Serial asynchronous requests

Pseudo-code

```
function f() {
    pre-requests
    request₁
    response₁
    request₂
    response₂
    …
    requestN
    responseN
    post-requests
}
```

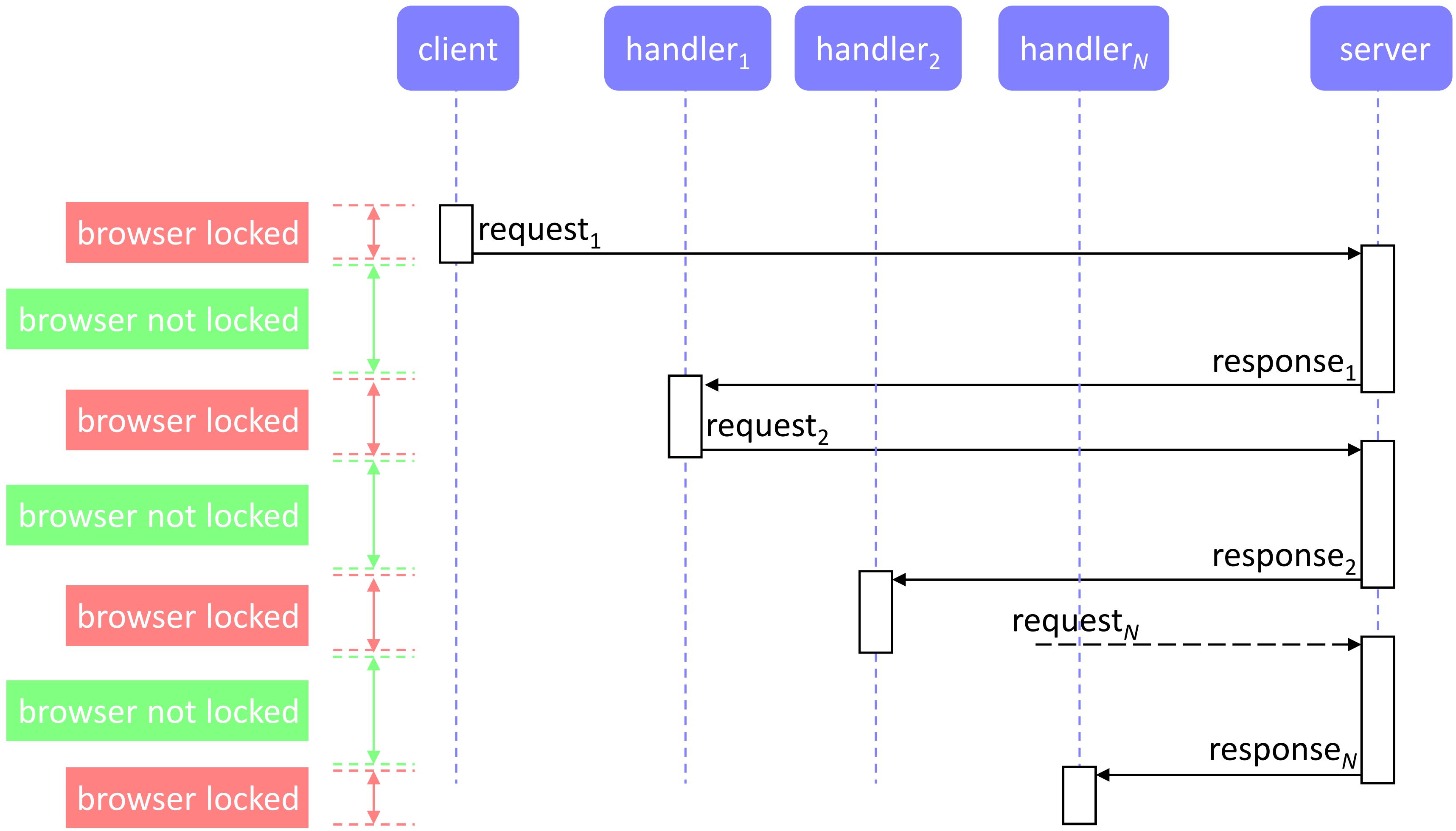
Take the synchronous (Design pattern 3) and apply Rule #1 to get the asynchronous (Design pattern 4)

→

```
function f() {
    pre-requests
    request₁
}
function f'₁() {
    response₁
    request₂
}
function f'₂() {
    response₂
    request₃
}
…
function f'N() {
    responseN
    post-requests
}
```

4. Serial asynchronous requests

Sequence diagram



Parallel requests
Definition

- We can also write code to send multiple requests in **parallel**. The browser sends all the requests at once (not exactly concurrently but in cascade), then receives the responses in random order.
- By definition, parallel XmlHttpRequests can only be asynchronous, not synchronous.
- Parallel is ideal when the server can handle multiple requests at once.
- But with parallel, the requests must not depend on the responses, and the order in which the responses are received must not matter.

Parallel synchronous requests

N/A

Design pattern 5

5. Parallel asynchronous requests

Pseudo-code

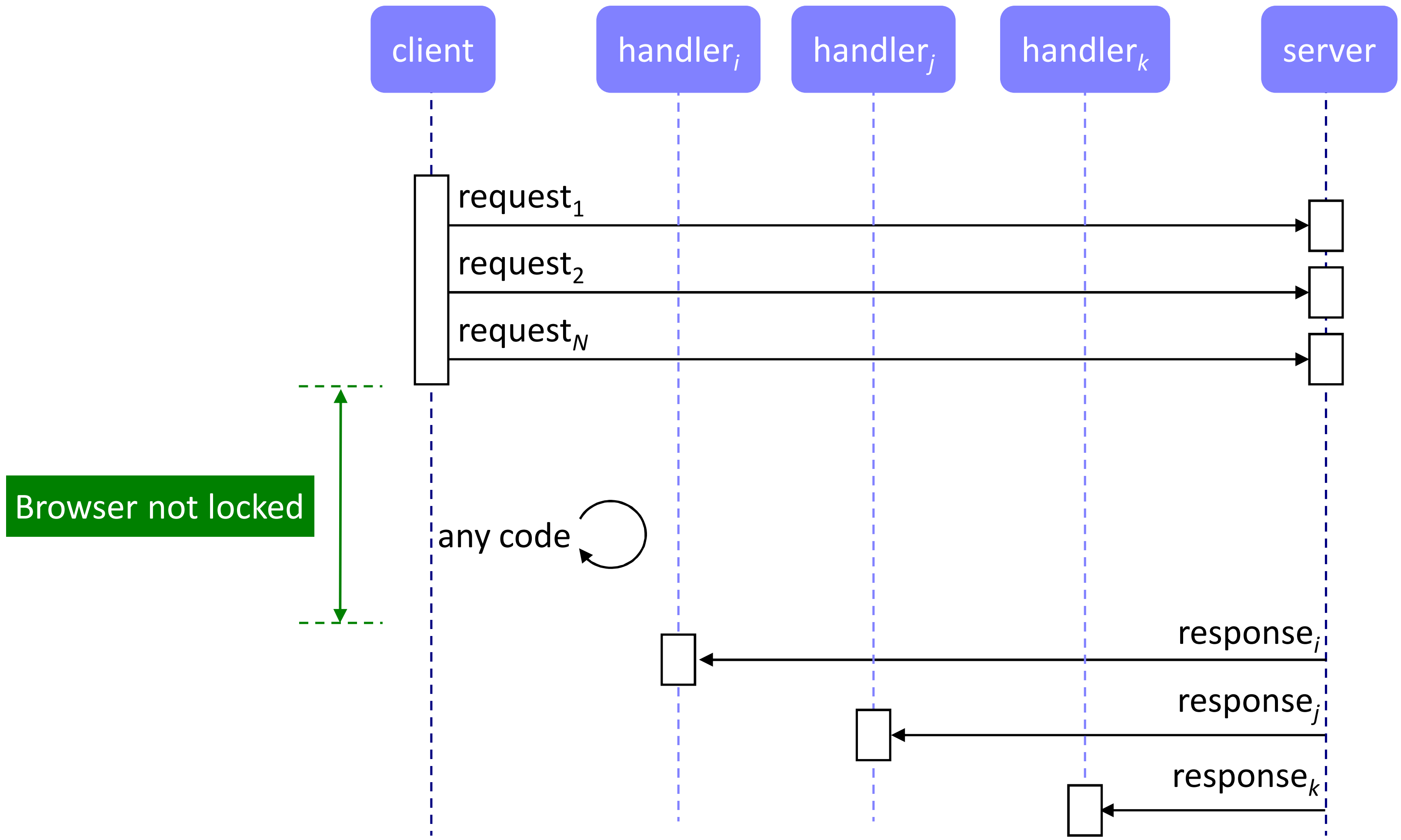
```
function f() {
      pre-requests
      request_1
      request_2
      …
      request_N
}
function f'_1() {
      response_1
      if (N responses handled) {
            f''()
      }
}
function f'_2() {
      response_2
      if (N responses handled) {
            f''()
      }
}
…
function f'_N() {
      response_N
      if (N responses handled) {
            f''()
      }
}
function f''() {
      post-requests
}
```

```
function f() {
      pre-requests
      request_1
      request_2
      …
      request_N
      i = 0
}
function f'_1() {
      response_1
      i++
      if (i == N) {
            f''()
      }
}
function f'_2() {
      response_2
      i++
      if (i == N) {
            f''()
      }
}
…
function f'_N() {
      response_N
      i++
      if (i == N) {
            f''()
      }
}
function f''() {
      post-requests
}
```

Variation

5. Parallel asynchronous requests

Sequence diagram



Conditional statements

if-then-else, *switch*

6. Synchronous *if-then-else* of one request

Pseudo-code

```
function f() {
    pre-if
    if (condition) {
        request
        response
    } else {
        something
    }
    post-if
}
```

Design pattern 6

6. Synchronous *if-then-else*

Pseudo-code

```
function f() {
    pre-if
    if (condition1) {
        request1
        response1
    } else if (condition2) {
        request2
        response2
    …
    } else if (conditionN) {
        requestN
        responseN
    } else {
        something
    }
    post-if
}
```

7. Asynchronous *if-then-else* of one request

Pseudo-code

```
function f() {
   pre-if
   if (condition) {
      request
      response
   } else {
      something
   }
   post-if
}
```

Split the synchronous (Design pattern 6) and copy the post-if to get the asynchronous (Design pattern 7)

Rule #2

```
function f() {
   pre-if
   if (condition) {
      request
   } else {
      something
      post-if
   }
}
function f'() {
   response
   post-if
}
```

Correct split & copy

```
function f() {
   pre-if
   if (condition) {
      request
   } else {
      something
   }
   post-if
}
function f'() {
   response
   post-if
}
```

Incorrect copy

```
function f() {
   pre-if
   if (condition) {
      request
   } else {
      something
      f''()
   }
}
function f'() {
   response
   f''()
}
function f''(){
   post-if
}
```

Variation of correct split & copy

Design pattern 7

7. Asynchronous *if-then-else*

Pseudo-code

```
function f() {
      pre-if
      if (condition_1) {
            request_1
      } else if (condition_2) {
            request_2
      …
      } else if (condition_N) {
            request_N
      } else {
            something
            f''()
      }
}
function f'_1() {
      response_1
      f''()
}
function f'_2() {
      response_2
      f''()
}
…
function f'_N() {
      response_N
      f''()
}
function f''() {
      post-if
}
```

Design pattern 8

8. Synchronous *switch*

Pseudo-code

```
function f() {
    pre-switch
    switch(expression) {
        case i:
            request_i
            response_i
            break
        case j:
            request_j
            response_j
            break
        default:
            something
    }
    post-switch
}
```

Design pattern 9

9. Asynchronous *switch*

Pseudo-code

```
function f() {
    pre-switch
    switch(expression) {
     case i:
          request_i
          response_i
          break
     case j:
          request_j
          response_j
          break
     default:
          something
     }
    post-switch
}
```

Take the synchronous (Design pattern 8) and apply Rules #1 and #2 to get the asynchronous (Design pattern 9)

→

```
function f() {
     pre-switch
     switch(expression) {
          case i:
               request_i
               break
          case j:
               request_j
               break
          default:
               something
               f''()
     }
}
function f'_i() {
     response_i
     f''()
}
function f'_j() {
     response_j
     f''()
}
function f''() {
     post-switch
}
```

Loop statements

for, while, do-while

Design pattern 10

10. Serial synchronous *for*

Pseudo-code

```
function f() {
   pre-for
   for(initialExpression; condition; incrementExpression) {
      request
      response
   }
   post-for
}
```

10. Serial synchronous *for*

Pseudo-code

```
function f() {
    pre-for
    initialExpression
    f’()
}
function f’()
    if (condition) {
        request
        response
        incrementExpression
        f’()
    } else {
        f’’()
    }
}
function f’’() {
    post-for
}
```

Variation

Design pattern 11

11. Serial asynchronous *for*

Pseudo-code

```
function f() {
    pre-for
    initialExpression
    f'()
}
function f'()
    if (condition) {
        request
        response
        incrementExpression
        f'()
    } else {
        f''()
    }
}
function f''() {
    post-for
}
```

Take the synchronous (Design pattern 10) and apply Rules #1 and #2 to get the asynchronous (Design pattern 11)

→

```
function f() {
    pre-for
    initialExpression
    f'()
}

function f'()
    if (condition) {
        request
    } else {
        f'''()
    }
}

function f''() {
    response
    incrementExpression
    f'()
}

function f'''() {
    post-for
}
```

Parallel synchronous *for*

N/A

Design pattern 12

12. Parallel asynchronous *for*

Pseudo-code

```
function f() {
    pre-for
    for(initialExpression; condition; incrementExpression) {
        request
    }
}

function f’() {
    response
    if (all responses handled) {
        f’’()
    }
}

function f’’() {
    post-for
}
```

12. Parallel asynchronous *for*

Pseudo-code

```
function f() {
    pre-for
    i = j = 0
    for(initialExpression; condition; incrementExpression) {
        request
        i++
    }
}

function f’() {
    response
    j++
    if (i == j) {
        f’’()
    }
}

function f’’() {
    post-for
}
```

Variation

Design pattern 13

13. Serial synchronous *while*

Pseudo-code

```
function f() {
    pre-while
    while(condition) {
        request
        response
    }
    post-while
}
```

```
function f() {
    pre-while
    f’()
}
function f’() {
    if (condition) {
        request
        response
        f’()
    } else {
        f’’()
    }
}
function f’’() {
    post-while
}
```

Variation

Design pattern 14

14. Serial asynchronous *while*

Pseudo-code

```
function f() {
    pre-while
    f'()
}
function f'() {
    if (condition) {
        request
        response
        f'()
    } else {
        f''()
    }
}
function f''() {
    post-while
}
```

Take the synchronous (Design pattern 13) and apply Rules #1 and #2 to get the asynchronous (Design pattern 14)

→

```
function f() {
    pre-while
    f'()
}

function f'()
    if (condition) {
        request
    } else {
        f'''()
    }
}

function f''() {
    response
    f'()
}

function f'''() {
    post-while
}
```

Parallel synchronous *while*

```
N/A
```

Design pattern 15

15. Parallel asynchronous *while*

Pseudo-code

```
function f() {
    pre-while
    while(condition) {
        request
    }
}
function f'() {
    response
    if (all responses handled) {
        f''()
    }
}
function f''() {
    post-while
}
```

```
function f() {
    pre-while
    i = j = 0
    while(condition) {
        request
        i++
    }
}
function f'() {
    response
    j++
    if (i == j) {
        f''()
    }
}
function f''() {
    post-while
}
```

Variation

Design pattern 16

16. Serial synchronous *do-while*

Pseudo-code

```
function f() {
    pre-while
    do {
        request
        response
    } while(condition)
    post-while
}
```

```
function f() {
    pre-while
    f'()
}
function f'() {
    request
    response
    if (condition) {
        f'()
    } else {
        f''()
    }
}
function f''() {
    post-while
}
```

Variation

Design pattern 17

17. Serial asynchronous *do-while*

Pseudo-code

```
function f() {
    pre-while
    f’()
}
function f’() {
    request
    response
    if (condition) {
        f’()
    } else {
        f’’()
    }
}
function f’’() {
    post-while
}
```

Take the synchronous (Design pattern 16) and apply Rules #1 and #2 to get the asynchronous (Design pattern 17)

→

```
function f() {
    pre-while
    f’()
}
function f’() {
    request
}
function f’’() {
    response
    if (condition) {
        f’()
    } else {
        f’’’()
    }
}
function f’’’() {
    post-while
}
```

Parallel synchronous *do-while*

N/A

Design pattern 18

18. Parallel asynchronous *do-while*

Pseudo-code

```
function f() {
    pre-while
    do {
        request
    } while(condition)
}
function f'() {
    response
    if (all responses handled) {
        f''()
    }
}
function f''() {
    post-while
}
```

```
function f() {
    pre-while
    i = j = 0
    do {
        request
        i++
    } while(condition)
}
function f'() {
    response
    j++
    if (i == j) {
        f''()
    }
}
function f''() {
    post-while
}
```

Variation

Verifications

- A *for* of one iteration is like a single request
- A *while* is like a *for* without initialExpression and without incrementExpression
- A *do-while* with a condition equal to false is like a single request
- A single *if* with a condition equal to true is like a single request
- Multiple *if-then-else* with all conditions equal to true is like multiple serial requests

When to choose which pattern?

- Are you able to write complex code? In which case you won't be afraid of coding multiple asynchronous and parallel requests. Or is readability and maintainability more important? In which case classic synchronous requests are preferred.

- Is feedback important for the user (indicate activity, show progress)? In which case asynchronous MUST be used. Or can the browser just hang? In which case synchronous CAN be used.

- Can the server handle concurrent requests? In which case parallel processing CAN be used. Or will it accept the first request, lock the resources, and reject the other requests? In which case serial processing MUST be used.

- Do the requests depend on previous responses? In which case serial processing MUST be used.

General considerations

- It's recommended to execute business logic close to the source, on the server-side (ex: ERP), or perhaps on the middleware (ex: WebSphere). But sometimes there's no other choice than to execute it on the client-side (ex: XmlHttpRequests)

- Parallel requests are not exactly concurrent. They are done in steps. The maximum number of open connections depends on the client. It's usually two as recommended in the HTTP RFC.

- The implementation of the Design patterns depends on the programming language, on the browser, and on the developer

- Exceptions must be handled otherwise patterns fail

- Use setTimeout() as jump statements to avoid exception propagation and stack overflow (?)

Future work

- Exception handling
- Operators, expressions
- Boolean algebra
- Finite State Machine
- Virtual CPU

Conclusion

I started by determining what is the relevant variable involved in executing a single XmlHttpRequest, it is synchronicity (synchronous or asynchronous). I then determined what is the relevant variable involved in executing two XmlHttpRequests, it is the type of transmission (serial or parallel). I then determined two rules to pass from synchronous to asynchronous.

Equipped with these two variables and two rules I was able to determine how to execute any arbitrary number of intricate XmlHttpRequests like in conditional statements and in loop statements.

Now, the resulting 18 design patterns allow me to write code that has the optimal usability, increased quality and robustness. But it may be at the cost of readability.

By answering a few simple questions I can also determine what is the best design pattern for any given case.

As the tutorials and documentation available on the Internet do not cover these cases I hope that developers will find this paper useful. Perhaps, one day AJAX frameworks and tools will provide the same.

References

- The XMLHttpRequest Object
  http://www.w3.org/TR/XMLHttpRequest
- AJAX Design Patterns
  http://codeidol.com/ajax/ajaxdp/
- Ajax Mistakes
  http://swik.net/Ajax/Ajax+Mistakes
- Asynchronous I/O
  http://en.wikipedia.org/wiki/Asynchronous_I/O
- Asynchronous Request Processing Framework
  http://www.jcorporate.com/expresso/doc/edg/edg_asyncprocess.html
- Functional programming
  http://www.stanford.edu/class/cs242/readings/backus.pdf
- [OOP] Bertrand Meyer, Object Oriented Programming
- Usable XMLHttpRequest in Practice
  http://www.baekdal.com/articles/usability/usable-XMLHttpRequest/
- Hypertext Transfer Protocol -- HTTP/1.1, RFC 2616
  http://www.ietf.org/rfc/rfc2616.txt
- Sequence
  http://en.wikipedia.org/wiki/Sequence
- Service-oriented architecture (SOA)
  http://en.wikipedia.org/wiki/Service-oriented_architecture
- Business Process Execution Language (BPEL)
  http://en.wikipedia.org/wiki/Business_Process_Execution_Language
- Orchestration (computers)
  http://en.wikipedia.org/wiki/Orchestration_(computers)
- Web Service Choreography
  http://en.wikipedia.org/wiki/Web_Service_Choreography
- XMLHttpRequest: Cross-browser implementation with sniffing capabilities
  http://www.ilinsky.com/articles/XMLHttpRequest/
- XMLHttpRequest Usability Guidelines
  http://www.baekdal.com/articles/Usability/XMLHttpRequest-guidelines/
- Design Patterns for AJAX Usability
  http://www.softwareas.com/ajax-patterns
- Solution Required for Problem in "Ajax in Struts-based Web Application"
  http://today.java.net/cs/user/create/cs_msg?x-lr=cs_msg/17186&x-lr2=a/219
- Handling Multiple XMLHTTPRequest Objects
  http://drakware.com/?e=2
- Multiple XMLHTTPRequest Objects Redux
  http://drakware.com/?e=3
- Multiple XHR requests (AJAX)
  http://www.digitalbonsai.com/xhrmulti.php
- Multiple XMLHttpRequest 2
  http://www.digitalbonsai.com/?itemid=5
- Foldblog: Handling Multiple XHRs
  http://ajaxian.com/archives/foldblog-handling-multiple-xhrs
  http://foldblog.blogspot.com/2006/01/ajax-handling-multiple-xmlhttprequests.html
- Introduction to Ajax, 13. Multiple Requests
  http://javascript.about.com/library/blajax13.htm
- Making simultaneous AJAX requests
  http://dominounlimited.blogspot.com/2006/08/making-simultaneous-ajax-requests.html
- Concurrent Ajax
  http://www.hunlock.com/blogs/Concurrent_Ajax
  http://p2p.wrox.com/topic.asp?TOPIC_ID=36547