# Dynamically Detecting Uncaught Errors in Asynchronous Javascript Functions

Rabee Sohail Malik, Raheel Arif

## Abstract

With the widespread use of AJAX, the use of asynchronous function calls in client-side Javascript is now quite common. However, in current implementations of Javascript, the entire stacktrace is not printed in case of an exception being thrown in an asynchronously called function which makes debugging difficult. This project solves this problem by using Jalangi, a dynamic analysis framework for Javascript.

## 1. Introduction

Javascript is the most common, client-side scripting language used in web browsers (Flanagan 2011). Asynchronous Javascript and XML (AJAX) is a technology which allows the creation of asynchronous webpages, which further allows for the content on a webpage to change dynamically without having to reload the entire webpage (Chris Ullman 2007). An example of an AJAX call is given in Listing 1. In this code, a new request object is made and an anonymous function is assigned to the onreadystatechange variable of this object. This function is called asynchronously when the state of the request object is changed. When this function is called, the main thread of execution is not interrupted.

Currently, when an exception is thrown in an asynchronously called function, the full stacktrace of the program is not printed. This is because it is not obvious which function has initialized the request of which the asynchronous function has been called. The work done in this project prints such a stacktrace for asynchronous function.We define the stacktrace of an asynchronous function to be the current functions on the program stack plus the program stack at the time when the asynchronous function was initialized. An example stacktrace for the code given in Listing 2 is given in Listing 3. The entry function is *send*. It is worth noting the stacktrace does not print the true function stack of the program as the asynchronous function is not called by the function that it is initialized in. However, this stacktrace still contains valuable information which is helpful in debugging as it gives information about where the asynchronous function has been initialized. The asynchronous function is labelled as anonymous in the stacktrace because it does not have a name, however the line number in the code where it is initialized can be used as an identifier. The approach outlined in this paper has been implemented for two kinds of asynchronous function calls, AJAX request and *setTimeOut* function calls. An example of *setTimeOut* is shown in Listing 4. The first argument to the *setTimeOut* function is the function which is to be called after a predefined time interval. This time is passed as the second argument to the *setTimeOut* function. In this paper, the approach is explained using examples of AJAX request calls but the implementation is the same for *setTimeOut* asynchronous function as well, with only a few minor changes. The rest of this report is organized as follows: Section 2 discusses the approach that is used to solve this problem, in Section 3, the results of the approach when applied to a real project are described and in Section 4 the conclusion of this project is presented.

**Listing 1.** .

```
function send(json){
    var request = new XMLHTTPRequest();
    request.open("POST","/endpoint.php");
    request.onreadystatechange = function(){
        if (request.readyState === 4
            && request.status === 200){
            console.log("RequestSuccessfull");
        }
    }
    request.send();
}
```

**Listing 2.** .

```
function afterSendRequest(){
    console.log("AJAXRequestSent");
}
function send(json){
    sendRequest(json)
}

function throwException(){
    throw exception();
}

function sendRequest(json){
    var request = new XMLHTTPRequest();
    request.open("POST","/endpoint.php");
    request.onreadystatechange = function(){
        if (request.status !== 200) {
          throwException();
        }
    }
    request.send(json);
    afterSendRequest();
}
```

**Listing 3.** Stacktrace of example code in Listing 2

```
Send, Line 4
sendRequest, Line 12
Anonymous, Line 15
throwException(), Line 8
```

**Listing 4.** Example setTimeOut function call .

```
function example(){
   setTimeOut(
      function(){
         console.log("AsyncFuncCalled");
      }, 3000);
}
```

## 2. Approach

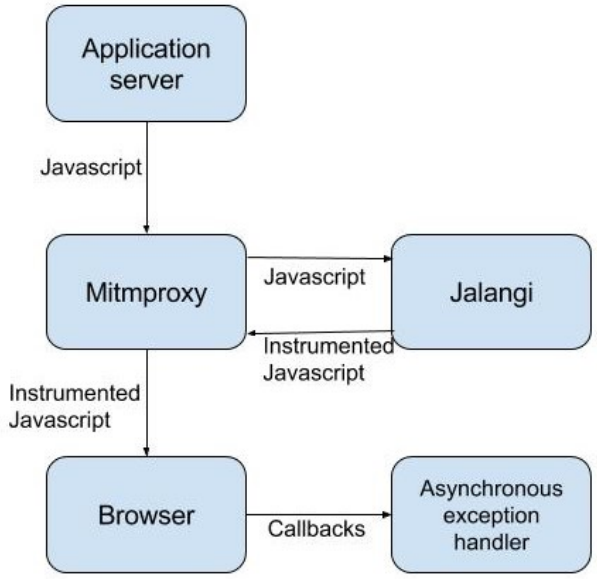


**Figure 1.** How the different components work together

Jalangi2 is used for dynamic analysis and for on-the-fly instrumentation of Javascript, mitmproxy is used. An overview of how the different components work together is given in Figure 1. Mitmproxy instruments the code before it reaches the browser, and when the code is executed on the browser, callbacks are made to the Asynchronous Exception Handler on certain events.

To be able to print the stacktrace, it is necessary to keep track of what functions are called. To this end, two callbacks of Jalangi are used, *functionEnter* and *functionExit*. *functionEnter* is called just after a function is called and *functionExit* is called immediately after a function exits. The name of function being entered or exited is passed as an argument to the callback. Using *functionExit*, it can also be determined if a function exited normally or if it threw an exception. Using this information, a simple stack can be maintained from which a function name can be popped off every time *functionExit* is called and a function name can

**Listing 5.** Pseudocode for building tree of function calls

```
currentNode = null
Tree = new Tree()

functionEnter(functionName)
{
    Node n = new Node(functionName);
    if(currentNode != null)
    {
        n.parent = currentNode;
        currentNode.descendants.add(n)
    }
    currentNode = n
}

functionExit(exitWithException)
{
    if(! exitWithException)
    {
        currentNode = currentNode.parent
    }
    else
    {
        Node n = currentNode;
        while( n.parent != null)
        {
            print(n.functionName)
            n = n.parent
        }
    }
}
```

be pushed on every time *functionEnter* is called. In case, a function terminates because of an exception, the contents of the stack can be printed. However, this approach would only work for code which does not have any asynchronous functions. This is because code with asynchronous functions has multiple threads of executions and each of those would have their own stack.

To solve this problem, a tree is used, in which each node $n$ represents a function $f$, and the descendants of $n$ are all the functions called by $f$. The pseudocode for building such a tree is given in Listing 5. The function currently executing is represented by *currentNode*. When *functionEnter* is called, the new function is made a child of *currentNode*, its parent is set to *currentNode* and finally, *currentNode* is assigned the node representing the new function. When *functionExit* is called, the *currentNode* is made equal to the parent of *currentNode*. If a function exits due to an exception, the stacktrace is printed by walking back from *currentNode*, all the way up to the root node and printing the name of the function represented by the nodes each step of the way.

Though this approach solves the problem of handling multiple threads of execution, it would still not work for asynchronous function calls. This is because there is no relation between the time when an asynchronous function is executed and the time when the asynchronous function is initialized. In practice, asynchronous functions are executed by the Javascript runtime only after the main thread of execution has finished executing (Kamani). This would result in all functions exiting before the asynchronous func-

**Listing 6.** Pseudocode for adding the node representing an asynchronous function

```
putFieldPre(baseObject, fieldName, value){
    if(baseObject.type == AJAXRequestObject
        && fieldName == "onreadystatechange")
    {
        Node n = new node();
        n.function = value;
        currentNode.addChild(n);
    }
}
```

tion is called and there would be no record of the function in which the asynchronous function was assigned to the *onreadystatechange* variable of the base object.

To solve this problem, two pieces of information are required: 1) determining when an asynchronous function has been assigned so that it can be added in the appropriate location in the tree and 2) on the execution of an asynchronous function, determining the location of that function in the tree. The approach for solving these two problems is detailed below.

### 2.1 Asynchronous Function Assignment

As can be seen in Listing 2, before an AJAX request is sent, the field *onreadystatechange* of the AJAX request object is written to. The value of that field is set to the asynchronous callback function, and it is called every time the value *onreadystate* field of the request object is changed. Jalangi2 provides a callback called *putFieldPre* which is called every time a field of an object is written to. The base object, the field name and the value being written are provided as arguments to this callback. Using this callback, the node representing the asynchronous function is added to the tree at the point of initialization by checking for the type of the base object and the name of the field that is being written to. This node is simply made the child of *currentNode*. *currentNode* here represents the node in which the asynchronous function has been initialized. The value that is being written, the asynchronous function itself that is, is also stored in the same node. It is important to note that the asynchronous function is not assigned to the *currentNode* variable as it as has not yet been called, and has only been assigned to the *onreadystatechange* field of the request object.

The pseudocode for adding the node representing an asynchronous function is shown in Listing 6.

### 2.2 Asynchronous Function Execution

There is no relation between the time when an asynchronous function is initialized, and when it was called. Therefore, when an asynchronous function is called, it is necessary to search the tree for the node representing the asynchronous function and make that node the *currentNode*. The *functionEnter* callback of Jalangi2 also provides the function body of the function being called. As previously stated, upon initialization, the asynchronous function itself is stored in its representative node. To find the node representing the called asynchronous function, the tree is traversed depth first and the field *prototype* field of the function passed to the *functionEnter* callback is compared to the *prototype* field of the function stored in each node. In case of a match, the matched node is assigned to *currentNode*.

The *functionEnter* callback is also passed an argument which indicates whether the function exited normally or whether it exited due to an exception. In case of the latter, the tree is walked from *currentNode* to the root node and the name of the function each node represents is printed. Thus, the stacktrace of the program is printed.

## 3. Results

The implementation of this approach was tested using an open-source website called "GoToVote". This source code of this website is publicly available on GitHub (GoT). Because no exceptions were encountered when browsing the website, exceptions were inserted to test the implementation. Two tests were conducted, one to test the AJAX asynchronous function call and one to test the *setTimeOut* asynchronous function call. Running both tests together did not yield correct results. This is most likely due to issues with Jalangi2 as it gives incorrect information for the second exception thrown. Removing the first exception but preserving everything else resulted in correct stacktrace being printed which again leads one to believe that it is Jalangi2 which behaves erratically in case of two exceptions.

The home page of GoToVote includes a Javascript file in which an asynchronous function is called due to an AJAX request. The first function to be called is an anonymous asynchronous function itself, which calls *run_get_centres*. Inside *run_get_centres*, an AJAX call is made. The asynchronous function passed to the AJAX request object calls a function called *parse_json* which calls another function *dropLastKElements*. Inside *dropLastKElements*, an exception was inserted. The screenshot of the resulting stacktrace is shown in Figure 2. The offset in the stacktrace gives the line number of the function call, relative to the *script* tag used to insert Javascript inside HTML. In case Javascript is placed in a separate file, the offset is simply the line number.

A similar test was done for testing asynchronous functions being passed to the *setTimeOut* function. The functions, however, were all artificially inserted as *setTimeOut* was not used in the original code. The screenshot of the resulting stacktrace is shown in Figure 3.

The steps for reproducing the test are outlined in the readme in the base folder of the project repository.

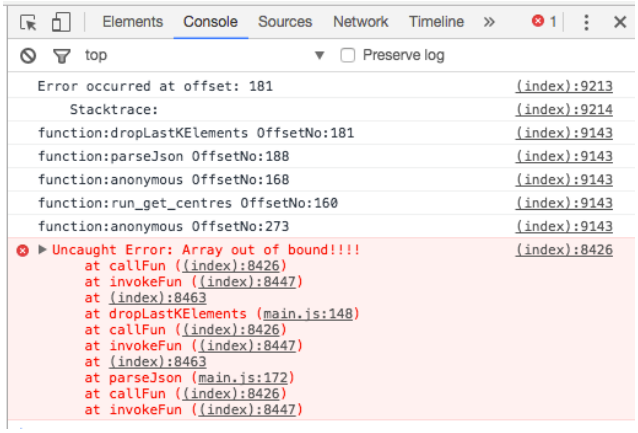


**Figure 2.** Stacktrace for the first test

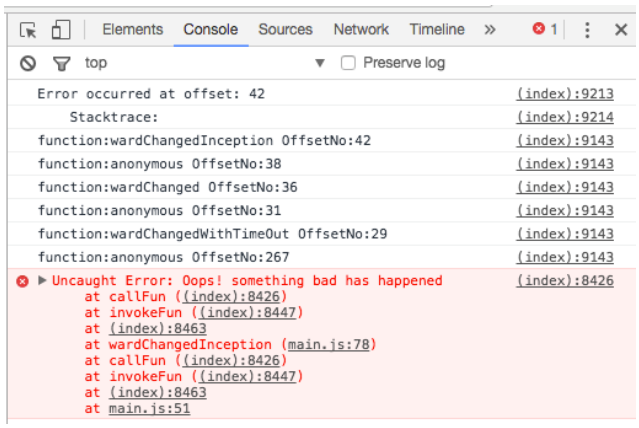


**Figure 3.** Stacktrace for second test

## 4. Limitations and Future Work

There are several ways in which the current approach and implementation can be improved.

Firstly, the storage of nodes in the tree can be optimized by removing nodes from the tree if all of their child functions have exited and the functions in question themselves have exited. This is possible because these functions will not be part of any stacktrace in the future.

Secondly, searching for asynchronous functions can be optimized. Instead of traversing the entire tree, a Map of asynchronous functions and their corresponding nodes in the tree can be maintained. This would bring the search time down from O(n) to O(1) where n is the number of functions calls in the program.

Thirdly, this approach assumes that the function assigned to onreadystatechange field of the AJAX request object is unique every time and is declared at the time of assignment as shown in Listing 1. This is not always true in practice. It is possible that a function declared elsewhere may be assigned to this field. It is also possible that the same function may be assigned to the onreadystatechange field of completely different base objects. In such a case, this approach would fail. It is left to future work to improve the approach to deal with such cases.

## References

Gotovote. `https://github.com/CodeForAfrica/GotToVote`. Accessed: 2017-02-19.

L. D. Chris Ullman. *Beginning Ajax*. Wiley, 2007.

D. Flanagan. *JavaScript: The Definitive Guide*. O'Reilly Media, 2011.

S. Kamani. How is javascript asynchronous and single threaded? `http://www.sohamkamani.com/blog/2016/03/14/wrapping-your-head-around-async-programming/`. Accessed: 2017-02-19.