Modular Asynchronous Web Programming: Advantages & Challenges

Wiliam Rocha
Shibaura Institute of Technology
Japan
xl15607@shibaura-it.ac.jp

Hiroaki Fukuda
Shibaura Institute of Technology
Japan
hiroaki@shibaura-it.ac.jp

Paul Leger
Universidad Catolica del Norte
Chile
pleger@ucn.cl

ABSTRACT

Because of the success of the Internet technologies, traditional standalone applications like Spreadsheet and Drawing are now provided as Web Applications. These adopt asynchronous programming that provides high responsive user interactions. At the same time these applications can grow and make their maintenance harder, turning Modular Programming an attractive practice because of its concept of dividing concerns in separated modules. However, it's difficult to combine asynchronous methods and modular programming because the first requires uncoupling a module into two sub-modules, which are non-intuitively connected by a callback method. It can spawn the creation of other two issues: callback spaghetti and callback hell. Some proposals have been developed to reduce the issues about modular programming. In this paper, we compare and evaluate them applying them to a non-trivial open source application, the FlickrSphere. Then, we will discuss our experience.

General Terms

Algorithms, Measurement, Design, Reliability, Experimentation, Verification

Keywords

Asynchronous programming; aspect-oriented programming

1. INTRODUCTION

Due the growth of high speed networks, traditional standalone applications such as drawing and spreadsheet software are now provided using web technologies, namely Web Applications. Such modern applications adopt asynchronous techniques such as AJAX, providing high responsive user interaction. At the same time, as the scale of such applications grows, its maintenance becomes more complex and there is where modular programming shines because it allows separating concerns into modules[8], meaning that changes made in one concern does not affect the others (e.g., other modules), turning the maintenance easier and safer. The basic idea of asynchronous programming is to decompose a blocking operation that awaits for its completion into a non-blocking operation that immediately returns the control by a callback method. Therefore, this practice can reveal two issues: callback spaghetti[6] and callback hell[7]. Callback spaghetti refers to the concern of the implementation when we have a complex and tangled control structure to the following executions over many callback methods. Callback hell refers to deeply-nested callbacks that have dependencies on data returned from previous asynchronous invocations. Also, the combination of Asynchronous Programming and Modular Programming can lead to structural problems that conflicts the concept of each one, because modules will need to be divided in two sub-modules that will be non-intuitively connected by a callback method called after the asynchronous function ends its execution and in a large scale application, the execution flow will be really trick to be tracked. Some proposals have been presented to crawl these issues such as async/await from C#[1], Promise pattern from JavaScript[3] and SyncAS from ActionScript[4]. Although the Promise and SyncAS versions were developed as Web Applications, the async/await version was developed as a common Windows Application. But it's important to denote that the async/await constructs work in a Web Application project too. This paper evaluates these proposals applying them to a non-trivial open source application called FlickrSphere[9], originally implemented in ActionScript3, using nested and iterative asynchronous functions which will bring some drawbacks such as Callback spaghetti and Callback hell. Then, our experiences will be discussed about the implementations.

2. ASYNCHRONOUS PROBLEMS

Nowadays, asynchronous programming is widely-adopted by programmers. This section briefly compare asynchronous programming and synchronous programming.

2.1 Synchronous Programming

Synchronous programming is the common style taught to programmers. Listing 1 shows one example of an application. The *ImageViewer* class contains one method: *showFromURL*. It basically downloads data from an image to display it. It's important to assume that the method download from *Request* class is a blocking operation that downloads data and takes significant time. The control flow is clear because each instruction needs to end and then advance.

BICT 2015, December 03-05, New York City, United States
Copyright © 2016 ICST
DOI 10.4108/eai.3-12-2015.2262472

Listing 1: Synchronous version of a remote image viewer.

```
class ImageViewer {
    function showFromURL(url :URL,) : void {
        var e : Event = new Request().download(url);
        var img : Image = convertToImage(e.data);
        show(img);
    }
}
```

2.2 Asynchronous Programming

Asynchronous programming style has more complicated control flows. Listing 2 shows the rewritten program of Listing 1 replacing a blocking operation (*download*) with a non-blocking operation (*downloadAsync*). Two major changes can be found. First, invoking *convertToImage* is removed from *showFromURL* because *send*, that invokes *downloadAsync*, returns immediately without any data. Instead, the reference of the *convertToImage* is passed as a callback by using next, which is defined in *Request* class. Second, the show function call must be moved to *convertToImage* because *showFromURL* does not contain the image. Summarizing, a module that uses a non-blocking operation requires defining the following instructions as a callback. As a consequence, if the next instructions are far from the call-site area in the code, understanding control flow will be harder.

Listing 2: Asynchronous version of a remote image viewer.

```
class ImageViewer {
    function showFromURL(url: URL) : void {
        var request = new
        Request().next(convertToImage);
        request.send(url) ;
    }
    function convertToImage(e: Event) : Image {
        var img : Image = convertToImage(e.data);
        show(img);
    }
}
class Request {
    var nextF : Function ;
    function next(f: Function) : void {
        nextF = f;
    }
    function send(url: URL) : void {
        var loader = new Loader() ;
        loader.addEventListener(Loader.Complete,
            callback);
        loader.downloadAsync(url);
    }
    function callback(e: Event) : void {
        nextF(e);
    }
}
```

3. FLICKRSPHERE IN A NUTSHELL

FlickrSphere is an open source Web application implemented in ActionScript3. Since ActionScript3 runtime does not provide threads for concurrent executions, programmers need to use asynchronous programming if necessary. This section briefly describes the behaviour of FlickrSphere and its original implementation. It accepts keywords from users, then accesses the Flickr web service[2] to get all URLs of images matched by the keywords. After, it downloads all images according to these URLs. Every time an image is completely downloaded, FlickrSphere displays it in an animated circle on the screen. Figure 1 shows a screen-shot of the original FlickrSphere, where its main behaviour carries out nested and iterative asynchronous executions.

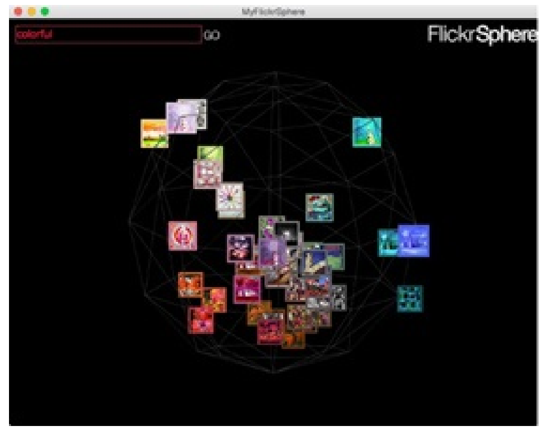

Figure 1: A screenshot of FlickrSphere.

4. APPLYING EXISTING PROPOSALS

This section presents different FlickrSphere implementations using existing proposals like async/await, Promise pattern and SyncAS. We will briefly explain each implementation and discuss them in a Qualitative and Quantitative tests.

4.1 Qualitative Evaluation

In the Qualitative Approach we discuss the level of Modularity, Expressiveness and Overload that each proposal offers to the programmer. Modularity refers to how we can concentrate one concern on one place. Expressiveness refers to how we can write programs naturally and intuitively. Overload refers to the difficulty that introduces each proposal.

4.1.1 *The async/await constructs*

The async/await constructs is a proposal that supplies writing programs with non-blocking operations in a synchronous fashion for C# 5.0. A method invocation is attached to *await* in order to keep the following executions as synchronous instructions processed when the asynchronous method execution is completed. The method usually contains an operation that takes a certain period of time. Meanwhile, a method definition with *async* modifier lets the compiler know if the method contains a method invocation that uses non-blocking operations. Listing 3 shows an example of the usage of async/await practice trying to get the number of images from the Flickr Web Service that match a keyword defined by the user. The method *Search* call the method with a non-blocking operation that returns a Task with the datatype required, like the *AccessFlickrSphereWS*. This task will be processed in other thread. You can see that inside this method there is a call for a third method called *DoIndependentWork* that will process in the main thread while the *getNumberOfImages* is running on the second thread. After the reserved word *await* is called with the task *getNumberOfImages*, the processor joins the threads.

Listing 3: Behaviour of FlickrSphere with async/await.

```
private async void Search(object sender,
    RoutedEventArgs e)
{
    int imagesFound = await AccessFlickrSphereWS();
}
async Task<int> AccessFlickrSphereWS()
{
```

```
    Task<string> getNumberOfImages = new
        HttpClient().GetStringAsync("url", "key");
    DoIndependentWork();
    string imagesFound = await getNumberOfImages;
    return Convert.ToInt32(imagesFound);
}
```

4.1.2 *Promise pattern*

One approach to deal with asynchronous issues adopted by JavaScript communities is the Promise pattern: a proxy object that represents an unknown (or future) result, not yet computed. The common term used for promise is *thenable*, as a programmer uses a then method to attach callback methods to a promise when it is fulfilled. Listing 4 shows the example of async/await written in Promise pattern. Promise requires decomposing a set of operations into methods, which will be called depending on the result of the blocking execution. Thereby, the method *search* from the object *searcher* will invoke the method *accessFlickrSphereWS* from the same object. This method call passes a callback (*downloadInfo*) for the *accessFlickrSphereWS* that will be invoked when the blocking operation finishes.

Listing 4: Main behaviour of FlickrSphere with Promise.

```
var searcher = function () {
   var accessFlickrSphereWS = function (k) {
      return new Promise(function (resolve, reject) {
         try {
            var obj = downloadInfo("Flickr_Url",
                keyword, null, errorHandler);
            resolve(obj);
         } catch (e) {
            reject(e);
         }
      });
   };
   var callback = function(o) {
      doSomething(o.countImages);
   };
   return {
      search: function (keyword) {
         accessFlickrSphereWS(keyword).then(callback,
             err);
      },
   }
};
```

4.1.3 *SyncAS*

SyncAS is a proof-of-concept library to provide virtual block, which enables a programmer to virtually block a method execution without blocking the execution of the program. A programmer specifies the points where an execution should be stopped and restarted using an aspect-oriented approach[5]. As a consequence, programmers can write programs as synchronous fashion even if they use non-blocking operations.

Listing 5: Main behaviour of FlickrSphere with SyncAS.

```
class Searcher {
   function search(key: String) : void {
      var obj : WSHelper = new WSHelper();
      var countImages : int =
          obj.accessFlickrWS("Flickr_Url", key);
      doSomething(countImages);
   }
}
class WSHelper {
   function accessFlickrWS(url: String, key: String) {
      // Code to access the WebService
   }
}
```

Listing 5 shows the rewritten code with SyncAS. Similar to async/await, SyncAS enables virtually blocking a method invocation that uses non-blocking operations. With SyncAS, we can write *Search* in a synchronous manner without the need to add constructs like found in async/await. Instead, the *WSHelper.accessFlickrWS* method is a method that contains a non-blocking operation, thereby, we need to flag a method as virtually block-able and restart them when *accessFlickrWSComplete* is finished as follows:

```
SyncAS.addAsyncOperation(
   "WSHelper.accessFlickrSphereWS",
   "WSHelper.accessFlickrSphereWSComplete");
```

4.1.4 *Discussion*

As shown in Table 1, these proposals are evaluated on Modularity, Expressiveness, and Overload.

The async/await has high expressiveness because we explicitly add the reserved word *async* to the method signature and the reserved word *await* to the non-blocking operation. Meanwhile, because these programmers explicitly need to write async/await in order to control executions, they will end up mixing asynchronous and synchronous methods. As a consequence, the modularity of this proposal is not considered high (i.e., Middle).

The Promise is native supported for most of the grade A browsers and has a simple implementation, which make its overload really low. However, callback methods are necessary to follow the promise style, bringing modularity issues like callback spaghetti (Low expressiveness and Middle modularity).

The SyncAS has similar features to async/await. However, a programmer who provides asynchronous methods also needs to provide aspects that control asynchronous executions. This fact means that besides the Overload is almost the same for the three solutions, SyncAS can be highlighted due its higher modularity when compared to async/await and enables dividing programmers into two categories: non-asynchronous programmers who just work with non-blocking operations, and asynchronous programmers who manage asynchronous executions. Meanwhile, SyncAS does not provide loops, forcing the use of self-recursion solutions, leading to non-intuitive programs. Therefore the expressiveness is lower than async/await, but higher than Promise (Middle).

Table 1: Comparison by Modularity, Expressiveness, Overload

	async/await	Promise	SyncAS
Modularity	Middle	Middle	High
Expressiveness	High	Low	Middle
Overload	Thread level	Very low	Additional closure execution

Table 2: Comparison by Performance in milliseconds

	async/await	Promise	SyncAS
Async	411.3	668	400
Sync	302.4	510	300

4.2 Quantitative Evaluation

In this section we will discuss their acting when downloading a single image of 5kB up to 11kB in a scenario of 100Mbps. The benchmark test was made by adding a listener that registers the beginning of the download method and its end, repeated one hundred times to get the average.

4.2.1 The async/await constructs

The async/await presented good performance for downloading images, usually taking 411.3(ms). At the same time, when we executed our application without it, the time fell to 302.4(ms). When using the async/await, the circle kept moving randomly. And when without that, the circle stopped and waited until all images were downloaded.

4.2.2 Promise pattern

The promise pattern makes use of C# Page Methods to download the images to the disk and because the Server Side code just has this function, the process is quite fast, 668(ms). At the same time, without Promise it fell to 510(ms). Also, it was responsive and the browser didn't freeze.

4.2.3 SyncAS

The SyncAS had a slightly better performance than async/await, having 400(ms) when present and 300(ms) when not. The numbers can be considered promising for a real application.

4.2.4 Discussion

In a scenario of 100Mbps and personal computers, we may consider that the overhead won't be that high due a real application would have other settings and processing capabilities. Async/await and SyncAS did equal numbers in different machines and operating systems, but still can be good options for programmers depending on what they are developing and the ideas discussed in the qualitative test must be considered for that decision. On the other hand, Promise pattern had good numbers too. Of course we cannot compare it with async/await and SyncAS due JavaScript relies on its browser and may run slower or not. Considering each proposal presented and their performance, the three technologies have attractive characteristics for the purpose of each language and can be applied with gains.

5. CONCLUSIONS

Asynchronous programming makes it possible to create high-responsive solutions for Web purposes. However, because it uses callbacks to manage the program flow, this solution has its drawbacks due its higher level of complexity and can be considered a modern goto statement. In addition, introducing asynchronous programming into module based programming requires dividing a method into call-site and its callback continuation, creating complex control flows. In order to solve these drawbacks, some proposals are available, however, issues related to modular programming, expressiveness and complexity are still present. We evaluated and compared three proposals: async/await, Promise pattern, and SyncAS, applying them to a application called FlickrSphere. From a modular programming viewpoint, SyncAS is better than other two proposals because can encapsulate non-blocking operations in a module completely. From an expressiveness viewpoint, async/await is better due to supporting of loops (e.g., for) and making it clear on differentiating what is asynchronous and what isn't. Also, the Promise pattern is only useful when developers need a lightweight solution. And considering the performance gains, SyncAS and async/await had the same numbers, but Promise had an worse result maybe due the browser-dependency. Besides that, all of them couldn't perform better than synchronous programs, but this small difference between the synchronous and asynchronous can be ignored due its improvement for responsiveness.

6. ACKNOWLEDGMENTS

This work was partially supported by JSPS KAKENHI Grant Number 26330089 and CAPES Foundation, Ministry of Education of Brazil, Brasilia - DF, Zip Code 70.040-020.

7. REFERENCES

[1] Bierman, G., Russo, C., Mainland, G., Meijer, E., Torgersen, M.: *Pause 'n' play: Formalizing asynchronous C#.* In: Proceedings of the 26th European Conference on Object-Oriented Programming. pp. 233-257. ECOOP'12, Springer-Verlag, Berlin, Heidelberg (2012), `http://dx.doi.org/10.1007/978-3-642-31057-7_12`;

[2] Flickr: `http://www.flickr.com/`;

[3] Friedman, D., Wise, D.: *The Impact of Applicative Programming on Multiprocessing.* Technical report (Indiana University, Bloomington. Computer Science Dept.), Indiana University, Computer Science Department (1976); `http://books.google.co.jp/books?id=ZIhtHQAACAAJ`;

[4] Fukuda, H., Leger, P.: *A library to modularly control asynchronous executions.* In: Proceeding of the 30th ACM/SIGAPP Symposium On Applied Computing, pp. 1648-1650, ACM Digital Library Salamanca, Spain Apr. 13-17, 2015;

[5] Kiczales, G., Irwin, J., Lamping, J., Loingtier, J., Lopes, C., Maeda, C., Mend- hekar, A.: *Aspect Oriented Programming.* In: Special Issues in Object-Oriented Programming. Max Muehlhaeuser (general editor) et al. (1996);

[6] Mikkonen, T., Taivalsaari, A.: *Web applications - spaghetti code for the 21st century.* In: Proceedings of the 2008 Sixth International Conference on Software Engineering Research, Management and Applications. pp. 319-328. SERA '08, IEEE Computer Society, Washington, DC, USA (2008), `http://dx.doi.org/10.1109/SERA.2008.16`;

[7] Ogden, M.: *Callback hell.*, `http://callbackhell.com/`;

[8] Parnas, D.L.: *On the criteria to be used in decomposing systems into modules.* Commun. ACM 15(12), 1053-1058 (Dec 1972), `http://doi.acm.org/10.1145/361598.361623`;

[9] Yossy: be interactive: Flickrsphere, `http://www.libspark.org/svn/as3/Thread/tags/v1.0/samples/flickrsphere/fla/FlickrSphere.html`;