Vrije Universiteit Brussel

Software Languages Lab
Faculty of Computer Science
Vrije Universiteit Brussel (VUB)

# Meta-level Engineering for Debugging Asynchronous Applications in JavaScript

A Thesis presented for the Master degree in
Computer Science

Felipe Caicedo Moreno

Promotor: Prof. Dr. Elisa Gonzalez Boix
Advisor: Dr. Carlos Noguera


June 2014

# Abstract

The emergence of AJAX introduced the asynchronous paradigm to web-based applications, allowing developers to implement new solutions for interacting with the server in an asynchronous way. However, the paradigm also introduced some problems given that JavaScript has a concurrency model based on an event-loop [16]. In fact, web-based applications introduce different communicating event loops that process different types of events. Such model allows implementing non-blocking web pages, but divides the control flow in two parts: the request process and the reception process, which makes the code difficult to understand and maintain.

Taking into account the importance of the debuggers because they help developers understand the control flow of a program and identify and resolve discrepancies with the intended behaviour, the problems noted above lead us to identify three challenges that a tool for debugging asynchronous JavaScript programs must face: message-oriented, open debugging and heterogeneity that the current debugging tools for JavaScript applications cannot overcome.

This thesis presents a reflective model that we call MIAJ – Meta-level Infrastructure for Asynchronous JavaScript Applications. It is designed for giving support in debugging asynchronous JavaScript programs, and overcoming the challenges of debugging applications based on communicating event-loops. This infrastructure reifies the communication traces capturing interactions (messages) exchanged between event loops. The model combines ideas from classic channel-oriented reflective frameworks [1] with a transmitter-receptor model from AmbientTalk's/M language [3]. It supports, by default, web abstractions such as DOM, distributed communication or MySQL. MIAJ allows developers to reify any type of event (e.g. jQuery, promises, AJAX, etc.), and ends with the heterogeneity of web-based applications. It furthermore provides support for establishing the causality flow between the events processed by the communicating event loops. Based on our architecture, we propose a distributed debugger for asynchronous JavaScript applications, the first one for online use, to the best of our knowledge. This implements the traditional debugging features such as *state inspection* (allowing us to pause the message processing), *stepping* (allowing us to process paused messages), *causal link browsing* (allowing us to establish the happened-before relationship between messages), and *open debugging* (allowing actors to disconnect and reconnect to the debugging session and to keep control of the processed messages while disconnected).

# Acknowledgments

I would like to give special thanks to my project supervisors Elisa Gonzalez and Carlos Noguera, in the first place for allowing me to participate in this thesis, and in the second place for their patience, support, guidance and on-going work during its elaboration.

Also thanks to the whole team of SOFT Lab for their help; providing feedback with presentations, resources and suggestions. The quality of this thesis would not be the same without all the help offered, especially by Elisa and Carlos.

Secondly, I would like to thank my family because they are my real support. To my mother, a tireless campaigner and my personal advisor; to my father, a tireless motivator and inspiration, and to my siblings, a key part in all I do.

And last but not least, many thanks to my dance partner for her support during the stressful moments while I was working on this thesis.

# Content table

# List of figures

# List of tables

# 1 Introduction

## 1.1 Context

No more than 15 years ago, all web applications worked synchronously. One known example is Gmail: every time a user wanted to check an incoming email, he or she could click on a button called *Refresh inbox.* The action of that button was to reload the page. Indeed, any action or request made by the user implied opening a new page or reloading the same one, thus blocking the web browser from processing every request.

Nowadays, checking new emails does not require reloading a page, and indeed, does not require any user action since servers can push information to clients. The asynchronous paradigm introduced new solutions, but also introduced some problems because JavaScript has a concurrency model based on an event loop [16]. This has the advantage of non-blocking pages, but makes codes more difficult to understand and maintain.

Asynchronous models also introduced a type of function called *callbacks*, which are used for evaluating the return value of asynchronous messages. Now imagine the following process in asynchronous applications: obtaining, filtering and finally showing the information from a source. As we can see, the processes are inter-dependent (e.g. we cannot filter the information before obtaining it) and implemented with callbacks as follows:

```
getData(function(data){
    filterData(data,function(filteredData){
            showData(filteredData,function(){
            });
    });
});
```

As shown in the code snippet, the control flow of the application is now driven by the activation of different callbacks. This leads to the phenomena called *callback hell* or *pyramid of doom* [11]. Such phenomena complicated both software development and maintenance, including debugging.

In view of the potential benefits of using asynchronous programming in JavaScript, the number of these implementations has risen considerably in the past few years and with it, new patterns for taming centralized and distributed asynchronous JavaScript. However, the variety of tools for debugging these applications has not risen accordingly.

Debugger tools have an important role in the software development process; they provide support for understanding the flow of a program (e.g. How should a developer interpret the tree

of asynchronous calls?) as in the previous example, and detect and resolve discrepancies with the intended behaviour.

On the one hand, current debugging tools for JavaScript applications such as Chrome DevTools [8], allows us to debug JavaScript applications only in the client browser, not allowing interaction with the server, and limiting the debugging process. On the other hand, we can find tools such FireDetective [32], which allows us to debug both sides of the communication, but always *after* the application execution (we call these debuggers *offline*) and only for AJAX-oriented messages.

The goal of this thesis is to investigate debugging support for JavaScript applications, which allows us to reify all communication between the different components of a web-application, independently of the underlying technology being used.

## 1.2 Motivation

In order to have an overview of the challenges that suppose debugging asynchronous JavaScript applications, we introduce the case scenario below. This will be used as a running example for the rest of the thesis.

### 1.2.1 Case Scenario: Library application

Consider the case of a public library website, where the user can browse and borrow books. The system has to check certain preconditions before allowing a user to borrow a book. First, the system has to check whether the user is prohibited or blocked from borrowing books (e.g. because the user returned a book one month late). Secondly, the system checks whether there is enough copies of the book in stock. Finally, whether the user has not borrowed the allowed maximum amount of books.

Implemented in a distributed asynchronous model, the process is as follows. When the user clicks the button *borrow book,* the system launches the three asynchronous requests to check the three aforementioned preconditions. These requests are received and processed by some machine and returned to the client. Finally, each response received is collected and when the three have arrived, the result is reported to the DOM as shown in Figure 1.

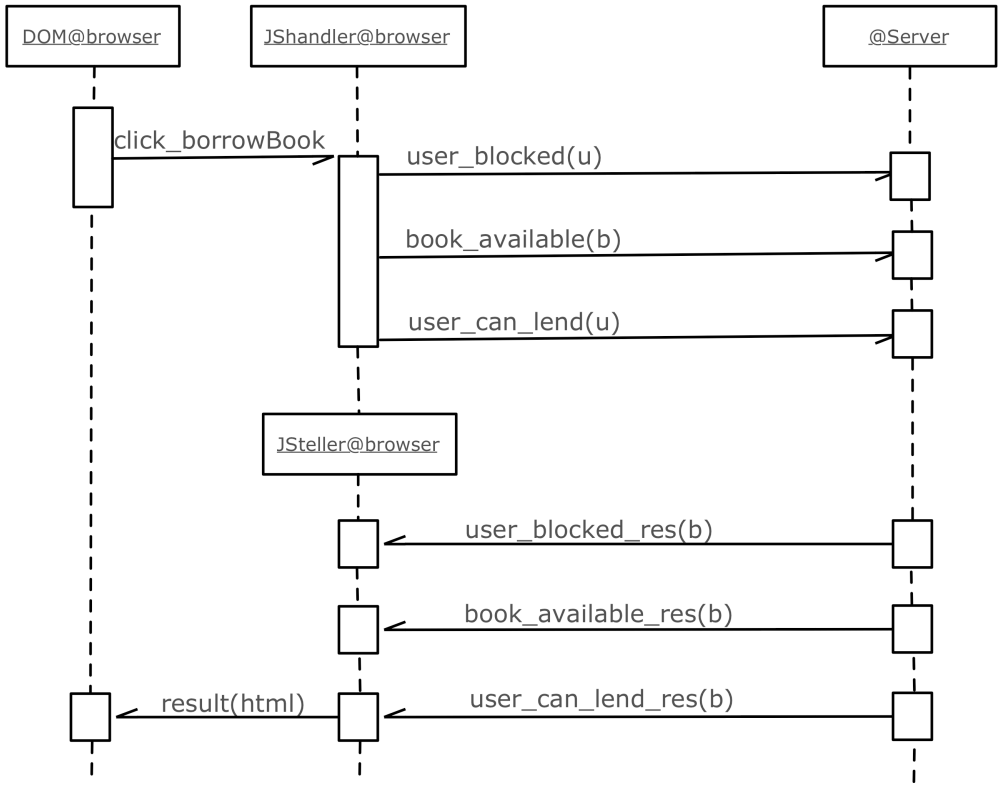


Figure 1: Borrowing a book process

As shown, what applies here are instructions or statements executed in parallel, with no apparent relationship between them. The three requests are sent at the same time, and appear to have no relationship between them. Moreover, the control flow is divided into two parts, the request process (e.g. *user blocked*) and the reception process (e.g. *user_blocked_res*).

### 1.2.2 Challenges of Debugging Distributed Asynchronous JavaScript

Debuggers are essential tools for developers; to begin with, they provide support for understanding the control flow of a program, and detect and resolve discrepancies with the intended behaviour. In sequential programs, all instructions or statements are executed in sequence, building a stack of execution, which can be used, for example, for examining previous states of the execution. Moreover, distributed programs involve different machines exchanging messages between them through the use of communication channels. This implies that the stack of execution now is distributed, which complicates the debugging process.

How to reproduce a bug if the application has different starting points and the whole application is not executed in the same space of memory is a question that becomes even more difficult to answer when the program is asynchronous.

JavaScript is based on the event-loop concurrency model [16], where all the requests/responses or messages are processed one by one by a single thread called event loop. A web-based distributed application like library involves the interaction of different event-loops.

Figure 1 shows three different threads of execution or event loops. One can consider that the DOM runs in an event loop interacting with the main JavaScript event loop that runs the client logical code, so, when a click occurs, a message is scheduled in the JavaScript event loop. In other words, when a click is detected in the DOM and processed by the JavaScript event loop (Message C In the image below), the callback associated to the event forward the three requests to the server event loop.

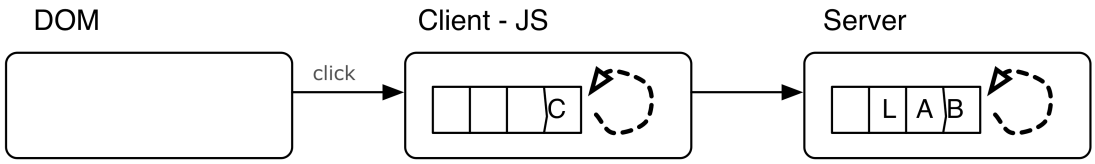


Figure 2: Communicating event loops

Suppose that there is a bug that provides the false information that a book is always available. If we want to find such a bug we should take into account different things:

- The debugging tool must go beyond the client browser, which means that the debugging information is in different spaces of memory.
  In this example, it is possible that the JavaScript logical code is reporting the information incorrectly to the DOM, but it could also be possible that the information received from the server is incorrect.
- All messages are treated as independent messages by each event loop, so, supposing that the problem is in the server, how does the client know that a given response was triggered by the request A (book_available)?
- Event loops never wait for responses locking the process, which sequentially means no information between turns (a turn is the moment the event loop processes a message and is atomic).
- The debugging tool needs to deal with disconnections and reconnections like MANET applications for being able of analysing all the messages. If, while debugging a client, debugging session disconnects, the debugging tool loses valuable information for locating the bug.

We now further discuss the challenges that a debugger for distributed asynchronous JavaScript application needs to face. Prior work has identified the following:

**Message-oriented debugging.** In sequential programs establishing the happened-before [12] relationship between messages is relatively easy because the stack of the program reflects the control flow of execution. However in asynchronous applications, establishing that relationship requires other techniques because the call stack is always empty at the beginning and end of processing a message (or turn). Notice that in Figure 1 the order of the requests and responses is as follows:

- *Click -> user_blocked -> book_available -> user_can_lend -> user_blocked_res -> book_available_res -> user_can_lend_res*

But if we want to create the happened-before relationship between these messages, it is not enough to follow their order. The debugger must establish which request caused each one of the three responses. The happened-before relationship of the messages from Figure 1 should therefore be:

- *Click*
  - *User_blocked -> user_blocked_res*
  - *Book_available -> book_available_res*
  - *User_can_lend -> user_can_lend_res*

**Open debugging.** Reproducing a bug is one of the hardest challenges while debugging an application. Suppose that there is a bug in the *borrow book* request. Examining the causality relationship between messages may be enough for finding it. But what happens when the developer is trying to reproduce a bug, and the client disconnects from the debugging session while processing the messages *book_available* and *book_available_res*?

The response to that question is that it would be impossible to locate the bug, because the bug could come from these messages. Moreover, the happened-before relationship could not be created because there are messages missing.

For this reason it is necessary to deal with disconnections and reconnections. If an actor is being debugged and disconnects from the debugging session, the messages missing while disconnected should be reported to the debugger when the actor reconnects.

Apart from these two challenges, web-based asynchronous applications present an additional challenge that we call Heterogeneous debugging. The event loop of JavaScript not only processes one kind of event since each event can be generated by different technologies (e.g. DOM events are generated by jQuery, and the communication between client and server could by means Sockets or AJAX), meaning that its structure can be different.

In Figure 1 we can observe three kinds of events: the click, the asynchronous request, and the modification in the DOM for reporting the results. Now, suppose that a developer wants to pause the execution of a program using a debugging tool. Pausing the execution in a program implies that the actor must have a mailbox for saving the events that are going to be paused. But is it necessary to have different kinds of events in a mailbox for each type of event? Thus allowing the developer to pause the execution depending on the type of event?

A debugger for JavaScript applications should deal with and take into account this heterogeneity, because in real applications there are more than three types of asynchronous events.

## 1.3 Goals and Contributions

A main problem to tackle is how to reify communication traces in order to capture interactions (messages) exchanged between event loops; for this reason, our first goal is to explore a meta-level architecture that allows us to reify communication traces, and based on this support, we aim to develop the first prototype for debugging distributed asynchronous JavaScript applications. To this end, we departure from the AmbientTalk debugger REME-D [3,26] and from the classic oriented reflective framework [1].

### 1.3.1 Contributions

We now highlight the contributions of this work:

- Design of a meta-level architecture on top of web abstractions for asynchronous JavaScript applications in order to reify communications traces. This is further described in Chapter 4.
- Application of this meta-level architecture capturing communication interactions of the following JavaScript technologies: Socket.io-Client [24], Socket.io-Server [24], jQuery [10] on top of DOM (click events and HTML mutation), and MySQL for Node.js [23].
- Design and implementation of a debugger based on Ambient oriented programming principles.

# 2 Distributed Asynchronous Applications in JavaScript

This chapter discusses how web applications work and which elements are relevant to consider in the definition of a meta-level infrastructure to reify communication traces.

## 2.1 Introduction

Establishing the causal relationship between synchronous computations is not that difficult because, as mentioned before, the call stack reflects the order execution. Asynchronous applications are completely different because computation is split in message processing and return values captured by promises [5] or callbacks.

Moreover, JavaScript uses a concurrency model based on an event-loop [16] (which means that all the computation is processed by its event loop), and processes different kind of events. This fact makes it even more difficult to establish the causality between computation. This heterogeneity of JavaScript led us to describe the events that could present problems while constructing the causality flow between computations.

**AJAX**. Asynchronous JavaScript And XML [18] was the first technology supporting asynchronous communication with the server. It is a programming technique for creating interactive web applications. The requests or messages are sent using the API XMLHttpRequest (or XHR) [17], and allows web applications to request and receive data from the server in the background without having to reload the web browser.

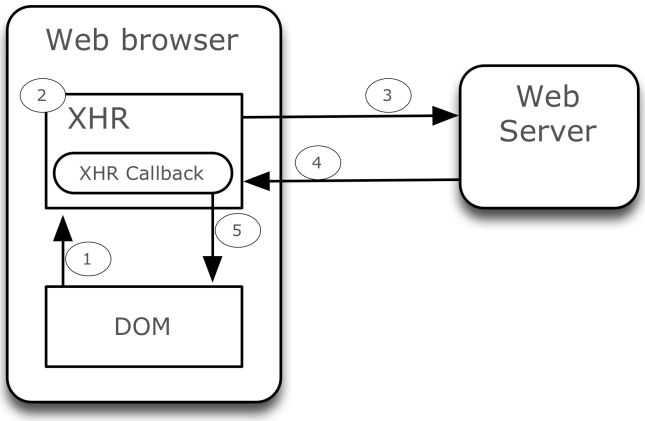


Figure 3: AJAX interactions

The process of sending an asynchronous message using AJAX is as shown in Figure 3. First an XHR object is created by some interactions from the DOM (for instance a click on the button *borrow book*). One of the parameters at the time of creating that object is a callback, which will gather the result of remote calls. Second, the object XHR requests a page from the server and when the reply arrives, the callback is executed, modifying the DOM if needed. The actions in points 3 and 4 represent remote communication and occur in the background for the client, as such the web browser is not blocked and the user can still interact with the web page.

One disadvantage of using AJAX for messaging is that it does not provide support for intercepting communications initiated from the server. AJAX techniques are normally used with thread-based servers, which can process different requests at the same time by dedicating a single thread for each request. And while a request is being processed the thread blocks requests from processing in the same thread. However, this kind of server cannot initiate communication. On the other hand, single-threaded servers such as Node.js [20] are never blocked during the processing of a request, since it is an event-loop and can initiate communication.

We can observe the difference between both types of servers in Figure 4. The result for every request received by a single-thread server would be an OK, the real result (Res1) is returned when processed (initiating the communication).

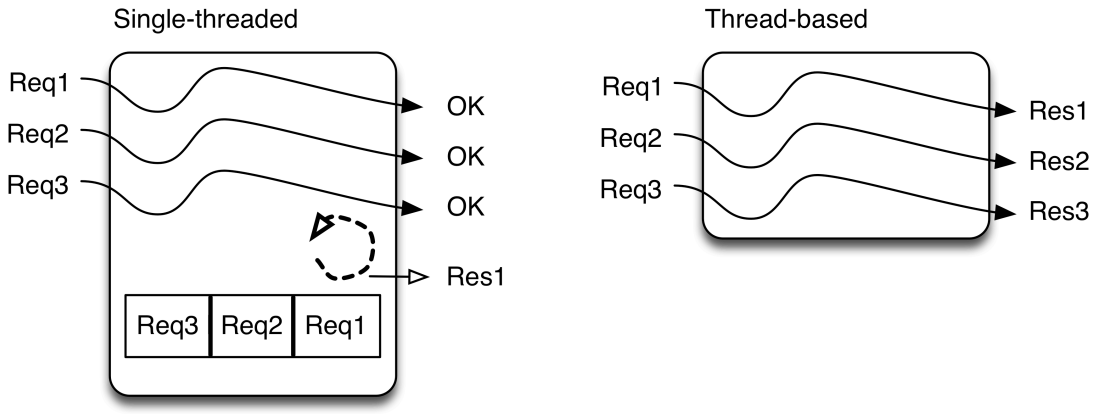


Figure 4: Non-blocking I/O VS blocking I/O

As is shown in the previous figure, all the responses of the requests in servers that use single-threads are always *OK* with no data. Using this kind of server, AJAX would take this data as correct, being actually *incorrect*. The response will be returned when the event-loop processes the requests and AJAX does not provide mechanisms for catching that response.

In simple terms, an AJAX request is an asynchronous request sent to the server; while this request is being processed, the server is blocked. It is only when the server is unblocked that the response returns to the client (the server returns the response if, and only if, it is unblocked). This is how AJAX captures the responses.

**Push-based Communication** is a model used for exchanging asynchronous messages between two points. This model solves the problem detected in AJAX since, in this case, both clients and server can initiate the communication.

This model is formed by two elements: subscribers and publishers. The subscribers, as the name suggests, subscribes to messages sent by the publishers.

```
subscribe('book_available_res', function (book) {
        //Show book info
});

publish('book_available', {id_book: 1});
```

In the previous fragment of code, we can observe that the application is acting as subscriber and publisher. On the one hand, the application subscribes to the event *book_available_res*; every time it receives these messages the callback passed as a parameter is executed for evaluating the return value. On the other hand, the application publishes a message called *book_available* for getting the book information. Note that the communication is not blocking: the message is pushed up to the server, and when it is ready the server will return the corresponding information initiating communication.

On the other side of the communication, the server needs to subscribe to the event *book_available* and publish the message *book_available_res* with the book information.

```
subscribe('book_available', function (book) {
        var res = getBook(book.id_book);
        publish('book_available_res', res);
});
```

Unfortunately, the use of this model could lead us to the *callback hell* or *pyramid of doom* [11] described in Chapter 1, because the return values are captured by callbacks.

**Promises**. To liberate the callback hell, JavaScript uses promises inspired by Argus's Promises [6]. These allow programmers to write asynchronous applications in a parallel way to synchronous applications, because these serve as a proxy future value.

A promise is a proxy of the returning value of an asynchronous request, in that way programmers can use the return value without waiting for the result. Therefore, using promises for processing a depending chain of asynchronous requests such as the example in Chapter 1 liberates the problem of dealing with depending callbacks.

```
Q.fcall(getData)
.then(filterData)
.then(processData)
.then(function (data){
        //shows data
},function(error){
        //handle errors between step 1 and step 3
}).done();
```

The example above uses the library Q [22] for JavaScript and represents the same example as Chapter 1. We can observe that the problem of dealing with callbacks disappears in this example. The first step *getData* returns a promise, but its result must also be a promise for filtering the data. The process goes on and on until the last promise which shows the data.

With regards to the state of the promises, it is said that a promise is pending while it is waiting for a value and resolved when it receives the corresponding value. A promise can be rejected as well, for this reason a second callback is executed for handling the error. The example above shows it.

## 2.2 Concurrency in JavaScript

As mentioned in section 1.2, JavaScript programs run inside a single-threaded event-loop [16], this event-loop process all the events executed in the web browser independently of its type.

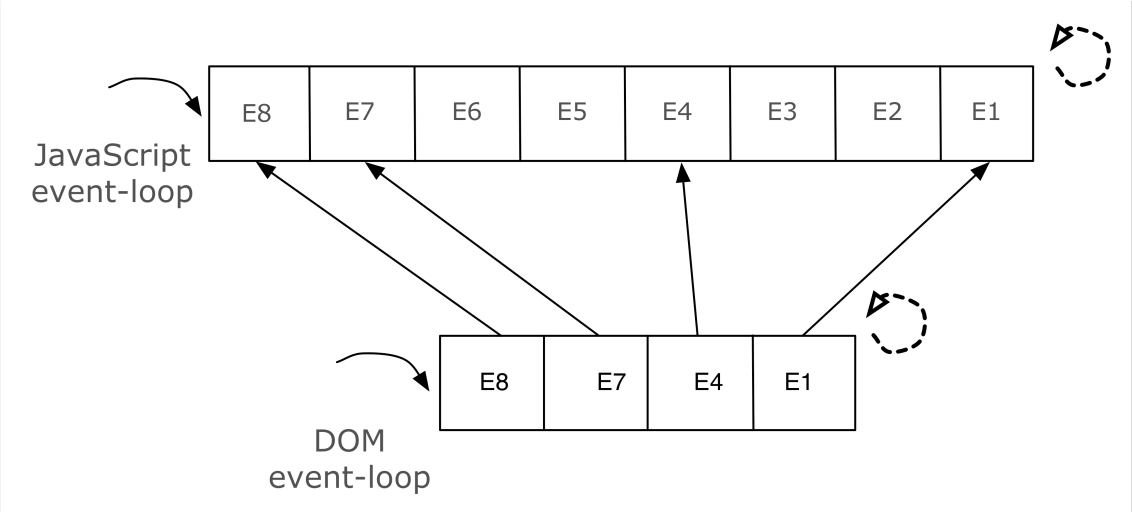


Figure 5: Event-loops in client side

In Figure 5, we can observe that the events 1, 4, 7 and 8 are events from the DOM and for the eyes of the programmer are processed by the DOM event-loop. However, what really happens is that all the events are added to the main event-loop in order to be processed. For the DOM, these events will be processed as if they were in an independent event-loop, but in a global context, other types of events can be executed first.

The JavaScript event-loop may process events such as the following [31]:

- **Page requests.** Occurs when a web page is loaded. The first time that we access a certain page, and every time we access another page through a link.
- **Resources requests.** In HTML, it is possible to include style files or JavaScript files to offer a visual design of the web page. When this happens, every file included is a resource request. All the files are loaded synchronously but not physically visible for the user.
- **Script invocation.** It is possible to include JavaScript code directly within HTML, not including the file that contains it.
- **DOM events**. Occurs every time the DOM changes or when an HTML element is clicked. Includes all the mouse and keyboard events.
- **Timeouts**. In JavaScript, it is possible to execute a function in X milliseconds or every X milliseconds.

This process is really important at the moment of establishing the relationship happened-before because it guarantees correlativity between events.

Regarding the server-side, more than one independent event-loop can be possible, unlike the client-side.

## 2.3 Conclusions

In this chapter, we presented the most important elements and characteristics of JavaScript applications, and the abstractions and structure of the web applications that should be taken into account in order to define a well-designed meta-level infrastructure.

# 3 Related Work

This chapter discusses the existing debugging tools, and their limitations for debugging distributed asynchronous JavaScript applications. However, it is analysed which one of its characteristics could be useful or should be taken into account in the design of the new tool.
On the other hand, others models of meta-level engineering relevant for this thesis are analysed and discussed.

## 3.1 Existing Debugging Tools

We now discuss current debugging tools that can deal with distributed asynchronous JavaScript programs and we also review relevant techniques for distributed asynchronous debugging besides JavaScript. Therefore, we are going to separate these into two categories: debugging asynchronous JavaScript applications and debugging non-JavaScript applications.

### 3.1.1 Debugging Asynchronous JavaScript Applications

Several debuggers are currently available for debugging JavaScript code; among others we can find Firebug [21] Chrome DevTools[8] or Theseus [15]. They provide traditional debugging support such as DOM inspection and modification, monitoring, breakpoints, variable inspection or step-by-step execution among others.

These tools only allow client-side code debugging and DOM interaction. This means that for debugging in the server-code, other tools should be used, meaning additional and independent processes for debugging a distributed application.

Therefore, if a developer wants to debug a distributed application, he or she should use two or more instances of different debugging tools in different spaces of memory, as well as the likelihood of using different features. The related interaction between client and server is needed to understand application behaviour, which in this case does not exist.

It is important to highlight that some of the tools do not directly deal with libraries as jQuery. Firebug, for example, can use an extension called FireQuery [2] that supports code implemented with that library. Note that there are a great number of libraries that do not have direct support of these tools.

#### 3.1.1.1 FireDetective

FireDetective [32] is an offline debugger that records the execution traces of a JavaScript program executed both by the client and server. This tool uses the call level detail, recording the names of the executed functions and methods, and the order in which they are called, enabling the tool to reconstruct a call tree representation.

Figure 6 shows the overall architecture of FireDetective, which is divided into three components:

- A Firefox add-on for recording the JS traces and information about the abstractions, which are specific to the web-domain.
- The server-tracer, installed in a Java Platform Enterprise Edition web server.
- The visualizer, both Firefox add-on and server-tracer, redirect the collected information to it (using sockets) as shown in the figure below. The visualizer processes and displays the given information in real-time.

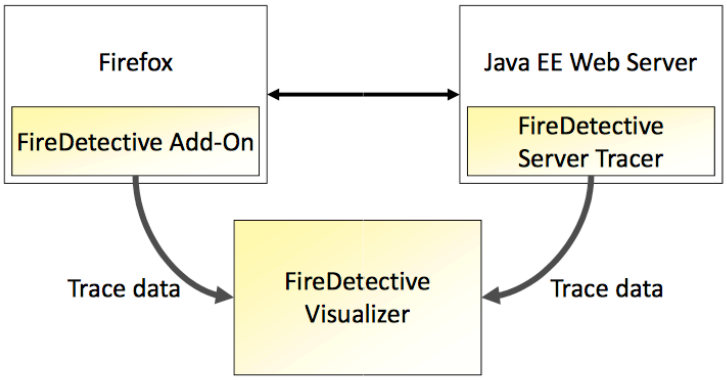


Figure 6: FireDetective Architecture. Figure extracted from [32].

FireDetective provides support for determining the workflow causality. It allows tracing client-server AJAX requests using only thread-based servers. However, there are some important points to highlight:

- This implementation only supports Java EE Web servers. So, it is not possible to apply distributed debugging using servers such as PHP.
- It is true that its model can be translated to other Web servers such as PHP or Ruby, but cannot be applied to servers that use event loop concurrency (as explained in section 2.1). So, to make this kind of web technologies extensible requires redesigning from scratch.
- It is an offline debugger; as such, it cannot be used to manipulate the behaviour of the program by means of traditional features such as pausing or breakpoints. The debugging process can only be initiated when the application is finished.

- Only client-server interactions can be debugged. Interactions with the Databases or file handling remain outside of the debugging process.
- Libraries such as Q for dealing with promises are not supported.

### 3.1.2 Debugging Asynchronous non-JavaScript Applications

#### 3.1.2.1 Causeway

Causeway [25] is a message-oriented distributed debugger that provides an offline view from trace files generated by the debugging process. It was created specifically for distributed applications built as communicating event-loops, in particular E [16]. However, applicable to JavaScript, it depends on generating the trace files with the corresponding code instrumentation. The figure below shows the four different views for the communication traces: the process order view, the message order tree, the stack frame view and the source code view.

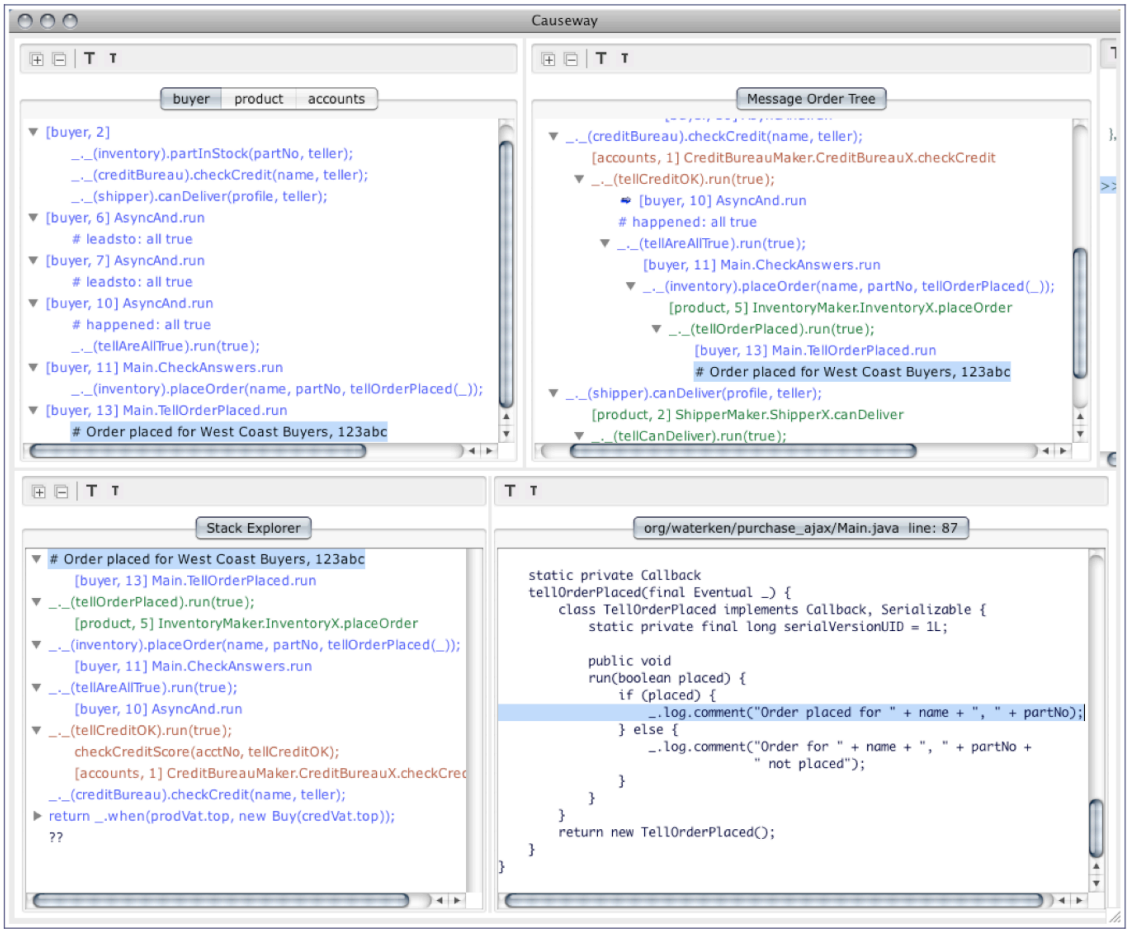


Figure 7: Causeway user interface. Extracted from [25].

**Process Order View.** In this view we can observe the different process within the debugging session (buyer, product and accounts in this case). Selecting the tab of the corresponding process. We can observe a 2-level tree of the events in chronological order. All the parent items are received events.

**Message Order Tree**. This view shows the happened-before relationship between events, showing which events caused others.

**Stack Explorer.** The stack explorer shows the events that caused the event selected in the message order view. It is also a 2-level tree and the parent node represents a sent event.

**Source Code View**. This view shows the source code that caused the message selected in the message order tree.

In comparison to FireDetective, it is applicable to different web technologies. Developers can generate the traces and use the views previously explained for debugging a given application. However, since it is offline, it does not allow us to use the traditional debugging features such as breakpoints or pausing.

#### 3.1.2.2 REME-D

REME-D [4,26] –Reflective Epidemic Message-Oriented Debugger – is a distributed debugger that supports the features of Ambient-Oriented programming [27] paradigms; non-blocking communications, ad-hoc networks and frequent network disconnections. It is a debugger that adapts the notions of sequential debugging, such as step-by-step execution or state introspection for ambient-oriented debugging.

The tool moreover combines these features with message-oriented architecture based on event-driven debuggers such as Causeway.

As a debugger for ambient-oriented programming, built on an event-loop concurrency model, its principles could be translated into other languages built on the same model, such as distributed asynchronous JavaScript. For this reason, the two challenges that address enabling distributed debugging in ambient-oriented programs are exactly the same for debugging distributed asynchronous JavaScript applications. These challenges are message-oriented debugging and open debugging.

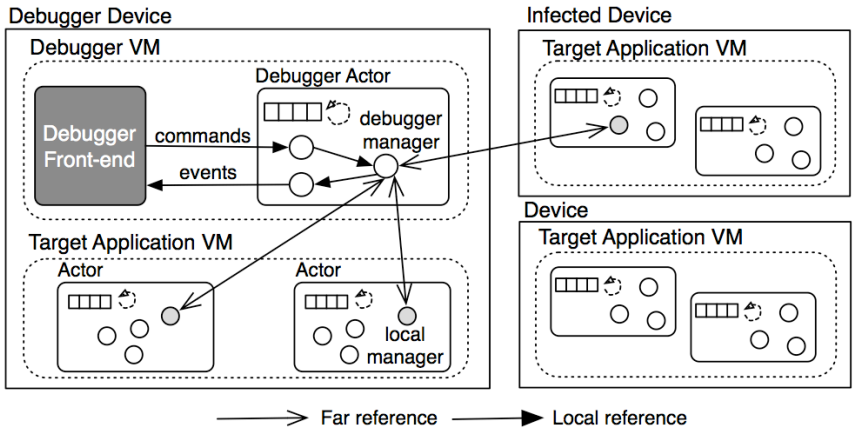


Figure 8: REME-D architecture. Figure extracted from [4].

In the image above, we can observe the architecture of REME-D, which consists of an actor called *debugger device* (which initiates the debugger session), infected devices joined to the debugging session dynamically and devices that run the program, but do not form part of the debugging session. A device can decide if wants to be debugged.

Each actor within the debugging session has a dedicated object (grey objects) called local managers; these implement the main debugging features of REME-D such as state inspection or stepping. Moreover, all the local managers maintain bidirectional communication with the debugger manager.

This debugger is able to identify every distributed message and establish the relation happened-before among them. Furthermore, it supports open debugging sessions.

So far, it is possible to translate all the features of REME-D to a debugger for asynchronous distributed JavaScript programs. Those are: state inspection, stepping, causal link browsing and epidemic debugging. However, the REME-D architecture is not directly applicable with web-based applications due to two reasons:

- The model of communication in Ambient Talk applications is P2P while in web applications it is client-server. Therefor, it would not be possible to use a client as a debugger manager because within the client-server model, clients cannot acquire communication between them. It is true that this communication could be established using the server, but that could create a problem of dependency; if an error occurs in the server, and the server is not able to process requests, the debug messages will never arrive to the debugger manager.

- Ambient Talk has remote object references called *far references*. In the Web, the concept of remote object references does not exist. In fact, there are more than one kind of asynchronous message as mentioned in section 1.2.

## 3.2 Meta-level Engineering

In JavaScript, there are no reflexive facilities to deal with asynchronous operations. As a result, the aim of this thesis is to review the meta-level engineering in JavaScript and in distributed communication.

### 3.2.1 Introduction

Meta-programming consists of writing programs that manipulate other programs [9]. When it is an object that describes, manipulates or implements other objects, it is called *meta-object.* The base-object is the object that the meta-object is about.

One of the most known meta-programs is the debugger; this allows programmers to change the behaviour of a certain program, whether executing it step-by-step or setting breakpoints in some part of the code.

Reflection, therefore, allows meta-programmers to examine the structure of a program and its data. Thus, when examining properties in JavaScript, the default behaviour is to include inherent properties that can intervene in the correctness of new implementations because these inherent properties must normally be ignored.

### 3.2.2 Meta-level Engineering in JavaScript

Reflection in JavaScript signifies intercepting method calls, proxying methods or adding news on the fly. For instance, suppose you desire to intercept all the calls to the method *console.log* and redirect the information to print to a remote visualizer. This is possible by proxying the method *console.log* and changing its default behaviour as shown in the code below. We can observe that the behaviour of the method is overridden, before printing the information in the console (*oldLog.apply*), the information will be sent to a remote visualizer.

```
var oldLog = console.log;
console.log = function(){
        redirectInfo(arguments);
        oldLog.apply(this,arguments);
}
```

#### 3.2.2.1 The proxy pattern

A simple definition of proxy is an interface used for representing an object or method, thereby protecting direct access to it. The classical proxy pattern allows proxying both at the method and object level. However if an object is proxied using this pattern, all the methods of the object must be proxied individually using the same technique. Proxying an object with this pattern does not imply that its methods are proxied as well.

The next piece of code shows a useful application of the pattern. In this case we are proxying a method that belongs to a non-instantiable object such as *window*.

```
1 var protectedAl = window.alert; //original reference to the function
2 var proxy = function(){ //defining the proxy
3       arguments[0] = 'Captured: '+arguments[0]; //New behaviour
4       protectedAl.apply(this,arguments); //Calling the original reference
5 };
6 window.alert = proxy; //Overriding the original function with the proxy
```

In the third line of code, the behaviour of the *alert* function is changed by adding the word "Captured" at the beginning of the string received as a parameter. In the fourth line, the proxy controls the access to the original function. Direct access to the original method *alert* does not exist anymore for programmers.

#### 3.2.2.2 Proxy API

Van Cutsem et al. [28,29] introduced a reflective API called Proxy for proxying both objects and functions in the programming language JavaScript.

When an object is proxied using this API, the behaviour of the object is exactly the same as the original object, intercepting all the messages sent to it. This implies that all the methods can be intercepted. This solves the problem of proxying an object with the proxy pattern. In this way, every method should be proxied individually.

The API introduces two types or proxies: *generic wrappers,* which are representations of objects in the same address space and *virtual objects*, which emulate other objects without having to be present in the same address space.

The Proxy API supports intercession; even though the proxied object has exactly the same behaviour as the original object, the methods are intercepted, which means that their behaviour can be changed or specialized [9] and also supports introspection, enabling developers to examine the object properties and methods.

In the code below we can see how to proxy an object using this API.

```
function makeTracer(obj) {
var proxy = Proxy(obj, {
get: function(tgt, name, rcvr) {
       console.log(tgt, 'get', name);
    return Reflect.get(tgt, name, rcvr);
},
set: function(tgt, name, val, rcvr) {
       console.log(tgt,'set', name, val);
       return Reflect.set(tgt, name, val, rcvr);
}});
return proxy;
}
```

```
var obj = new Obj();
obj = makeTracer(obj);
obj.prop = ‘hello’;
```

The constructor receives an object as a parameter and returns the new proxied object. Afterward, every time an object uses an *accessor* method for receiving object property or a *modifier* method for modifying an object property (as in the second fragment of code), the information will be printed in the console.

However, when proxying an object, we have the two-body problem: the proxy has its own identity and the programmers have access to both original and proxied objects. Every time that a proxied object is needed, it is necessary to create a new instance invoking *makeTracer*.

### 3.2.3 Meta-level Engineering Architectures for Distributed Computing

**Channel reification [1].** This model is an extension of the message reification model. It is a reflective model for distributed computation consisting in reifying as objects the abstractions that represent a channel (those that provide a service through a logical channel).

In the figure below, we can observe the main difference with the meta-object model; in the meta-object model, the meta-object reifies one specific object (e.g. the meta-object MA, reifies the object A), and in the channel reification model, the reification is between the communication of two objects (e.g. the meta-object C reifies the communication between objects A and B).

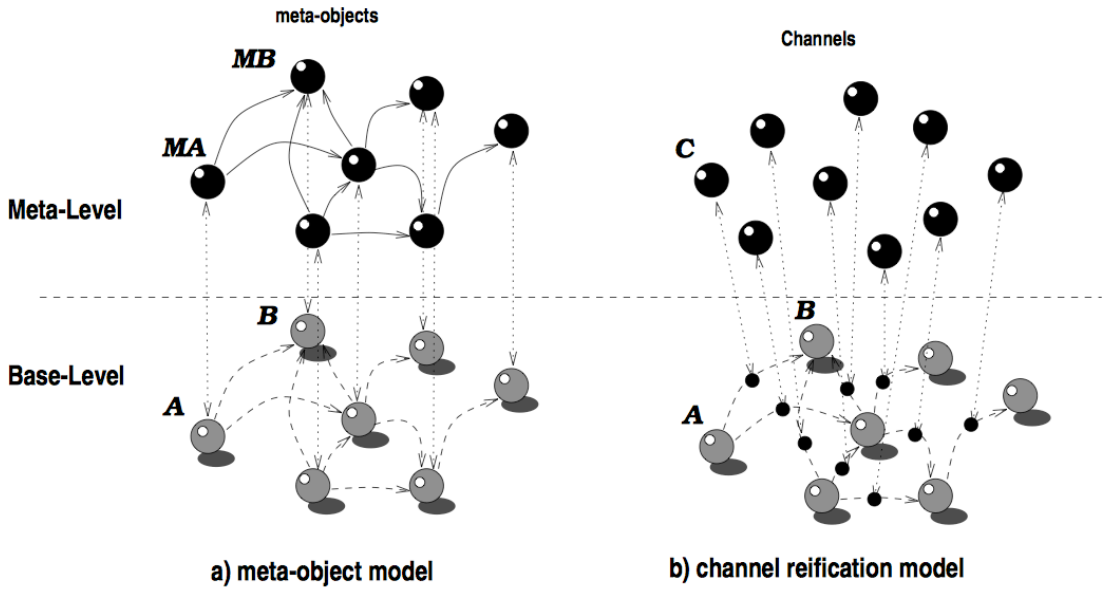


Figure 9: Meta-object VS Channel reification. Extracted from [1].

We can observe in detail in the figure below that when a service request occurs, the model reifies that communication into a channel, thus intercepting and controlling the sender and receiver actions of it, moreover, different channels can handle different method calls, which gives a reflective behaviour for each method.

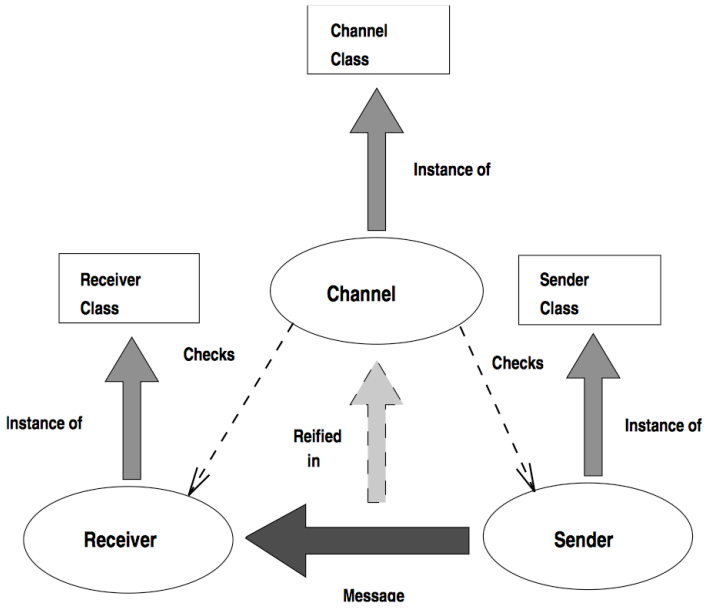


Figure 10: Channel reification model scheme. Extracted from [1]

**Meta-level architecture of AmbientTalk.** [3], Gonzalez introduces the transmitter-receptor model. The idea of this model is that both ends (C and S) reify the communication as a pair of meta-objects encapsulating all aspects of interactions between senders and receivers. The responsibility of dealing with network failures or sending and receiving messages belongs to the meta-objects. In Figure 11 we can observe its architecture.

By manipulating these meta-objects (marked in grey in Figure 11), developers can intercept and change the behaviour of the remote communication between both objects.
On the one hand, the transmitter reifies the communication channel at the client-side, thus controlling how the client object sends the messages. This object can perform some actions before sending the message.

On the other hand, the receptor reifies the service object for controlling the distributed operations with the service object: message sending and message reception. Being a service object, normally the requests received imply a response. This object can also perform some actions before sending or receiving a message.

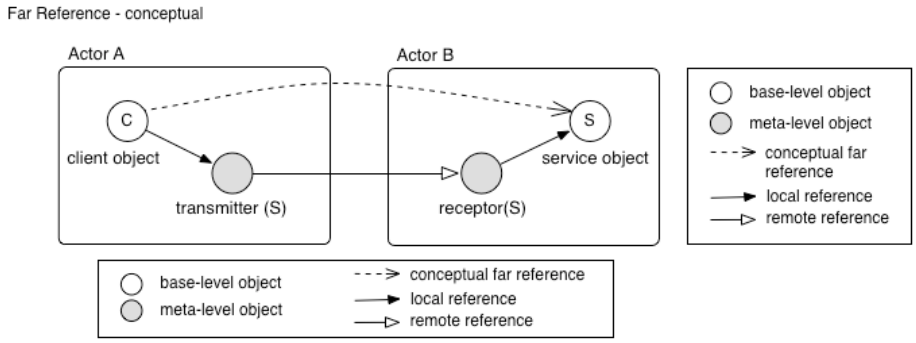


Figure 11: REME-D, far references architecture. Extracted from [3]

In this model we do not have the two-body problem such as the Proxy API, all interactions are captured by the transmitter and receptor meta-objects.

Moreover, even though the model was designed for distributed object models the architecture leads us to think of the meta-level infrastructure we want to design. On the one hand, remote objects references such as in [27] do not exist in web applications and there are more than one kind of remote reference as explained in section 1.2. However in a web context, this architecture can be interpreted as client-server communication rather than remote objects peer-to-peer communication. Thus, the client object could be replaced for a client machine, and the server object for a server machine. Each side with a set of meta-objects (created in pairs) for reifying

the different types of asynchronous communication and encapsulating all the aspects related to their behaviour.

## 3.3 Conclusions

This chapter discusses the existing debugging tools for JavaScript and their limitations for debugging distributed asynchronous. On the one hand, some of them are only for the client or for the server, thus, the debugging process must be done separately and independently. On the other hand, the tools that can deal with distributed computing are based in specific technologies such AJAX or thread-based servers, which makes it difficult for portability for servers based on the event-loop concurrency model. Moreover, the latter are offline, which means that the classic debugging features cannot be applied.

Alternatively, we analysed other debugging tools, which use the same model (non-blocking event-loop concurrency), the meta-level architecture of the debugger REME-D and the classic channel-oriented reflective framework, in order to know which characteristics can be profitable for our design.

We identify that a debugger for distributed asynchronous JavaScript must meet the following criteria:

- **Online**. Offline support providing reification of client-server interactions have been explored in FireDetective [32]. Creating an online debugger enables developers to use traditional debugging features such as pausing, breakpoints or stepping.
- **Message-oriented**. Creating the happened-before relationship between messages is essential for the debugging process. While in synchronous programming, it is enough to follow the order of execution, in asynchronous programs the messages are processed by different event loops in an independent way, because there is no information between turns, and the control flow is separated into two steps, the message request and the evaluation of the result. The debugger should be able to establish the happened-before relationship.
- **Heterogeneity**. Signifies supporting all the web abstractions (or asynchronous messages) aforementioned in section 2.2, DOM interaction, timeouts, databases, remote requests, etc. Web applications follow a logic that other kinds of applications do not follow (different kinds of events).
  Also, it must support servers built on the event-loop concurrency model and servers that do not use that model.
- **Open debugging**. On the one hand, the debugger must support distribution; a distributed application executes part of its code in the browser and part in some remote node, however, while still being one single application. The debugger must

treat a distributed application transparently as a single one, taking into account the layer that separates both parts of the application without addressing this as a problem.

On the other hand, the debugger should deal with frequent disconnections and reconnections to the network. The messages processed by the actors, while remaining disconnected, should be taken into account for the debugger process in order to provide accurate interpretations.

Analysing both cases of the meta-level engineering section, we can consider a first approximation of how should the meta-level architecture for reifying the JavaScript asynchronous communications be. Having on both sides of the communication a list of channels that can specialize or modify the behaviour of the asynchronous events.

# 4 Meta-level Infrastructure for Asynchronous JavaScript Applications

In this chapter will be described MIAJ –Meta-level Infrastructure for Asynchronous JavaScript applications - the meta-level infrastructure for reifying asynchronous communications in JavaScript programs.

## 4.1 Introduction

Recall from the previous chapter that in JavaScript, all communication is serialized in one event loop and each page (or frame) runs in a JavaScript event loop, which receives different kinds of events (e.g. DOM, network, timers, etc.). Some of those events actually correspond to communication with other event loops, which are hidden from the programmer. For example, consider a remote communication with server.

Remote communications normally have two operations for guarantying that an event will be processed. We call these operations *send* and *receive* respectively.


Figure 12: Communicating event-loops

As shown in Figure 12, when the *send* operation is executed for sending the event *e1*, the *receive* operation queues up the given event to the server event-loop in order to be processed as well.

In this study, we propose a meta-level infrastructure, which reifies this underlying event loop communication by means of a well-defined abstraction. Our architecture consists of three components: channels (reifying different kinds of communication (e.g. implicit event loops)), meta-channels (reifying both ends of communication) and messages (reifying the events sent and received in the JS as first-class objects). The model combines ideas from classic channel-oriented reflective frameworks [1] with transmitter-receptor model from AmbientTalk's/M language [3].

## 4.2 MIAJ

MIAJ has been designed to reify communication traces of JavaScript programs. As we can observe in Figure 13, MIAJ consists of a set of channels, meta-channels and messages. Channels are objects reifying the web abstractions or libraries that produce or require events (e.g. DOM or Socket.io). Thereby, a channel has two operations: *send* for producing events and *receive* for catching them. Note that a channel can be distributed, which means that two channels implementing the two operations are needed, one for each side of the communication.
By default we support the following channels:

- jQueryDOM: reifying the DOM communication through the jQuery library. Clicks and DOM mutations are supported.
- Socket.io: reifying the distributed Socket.io communication on both sides of the communication.
- MySQL: reifying the asynchronous MySQL operations on the server side.

As can be noted, every channel produces and catches different kind of events; the same channel can produce different kinds of events such as jQueryDOM. This produces click events and DOM mutation. On the same lines Socket.io and MySQL produce events with different structure and information.

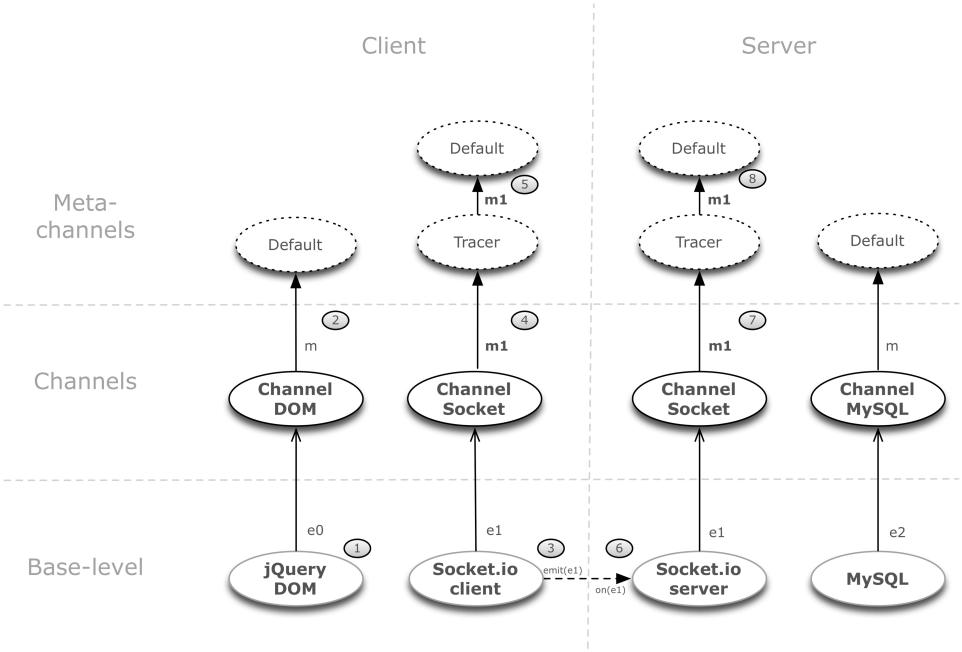


Figure 13: MIAJ overview

### 4.2.1 Meta-channels

Each channel has a list of meta-channels reifying both parts of the communication (*send* and *receive*); therefore, they intercept and modify the communication between event-loops.

By modifying them or creating news, developers can implement different and isolated behaviours for the events. So, for example, in Figure 13, the meta-channel Tracer is installed in the channel socket on both sides of the communication for tracing all the exchanged messages in the channel.

By default, there exists one meta-channel per supported channel, which implements the semantics of channels without the presence of MIAJ. For example, one of the default behaviours of JQueryDOM *receive* operations of a meta-channel is opening a new page after clicking on an element.

### 4.2.2 Messages

First-class objects called Messages reify communication between event loops.
When a channel intercepts or captures an event of a given abstraction or communication, the event is reified as a Message and delegated to the channel's meta-channels temporarily losing the responsibility of behaviour. We can observe step 3 of Figure 13: when the channel Socket intercepts the event *e1,* this is reified as the message *m1* and delegated to the *send* operation of the meta-channel Tracer.

In doing so, it guarantees:

- Reusing meta-channels in different channels. Developers could install the same meta-channel Tracer into the channel DOM, using exactly the same implementation. Otherwise a different meta-channel must be created for every channel and kind of event.
- Same channels can produces different kind of events. Note that the DOM channel intercepts different types of events (e.g. clicks or mutations). In this case, if the event is not reified as a message, developers will need to implement two different channels, one for reifying DOM mutations events and another for reifying links events.

The execution starts once a channel traps an event; this reifies the event as a message, and delegates this to its first meta-channel. When every meta-channel executes its behaviour, it delegates the message to the next meta-channel in the list of meta-channels; always depending on the type of the event (*send* or *receive*), these are delegated to the corresponding *send* and *receive* operations of every meta-channel.

The channel, thereby temporarily loses control of its events.

### 4.2.3 Channels Context

As referred to in Chapter 1, one of the challenges in asynchronous debugging is constructing the causality flow between computing, because each event-loop processes every message as independent (there is no information available between turns).

As we can observe in Figure 13, each channel is also independent (except the distributed channels such as Socket); a message in the DOM channel has nothing to do with a message in the Socket channel. However, in MIAJ, each channel has a reference to a sharing space that we call Context; if the DOM channel receives a click event, the channel can report that behaviour to the shared space, then, when the channel socket intercepts an asynchronous event, this latter channel could know that DOM was the last channel executed and even the information of the messages executed by it.

### 4.2.4 Execution Runtime

Let us explain the overall architecture with an example. Consider the case of a mouse click that triggers a callback, sending an asynchronous request using Socket.io [24] to a server. Moreover, that communication must be traced.

Every step is referenced in Figure 13 and is as follows:

1. The channel DOM intercepts the click event *e0* (incoming event).
2. The same channel reifies the event *e0* as the message *m*, and delegates this message to the *receive* operation of its first meta-channel, in this case the default. So, the callback is executed sending the corresponding asynchronous request (*emit(e1)*).
3. The channel Socket in the client side intercepts the event *e1*.
4. The channel Socket reifies the event *e1* as the message *m1*, and delegates this message to the *send* operation of its first meta-channel, Tracer in this case. So, the corresponding traces are generated for the message.
5. The meta-channel Tracer delegates the message to the *send* operation of the next one (The Default). This latter executes the default behaviour, which is sending the event to the server.
6. The channel Socket on the server side intercepts the event *e1* sent by the client. In distributed communication.

7. The channel Socket reifies the event as the message *m1*, and delegates the message to the *receive* operation of its first meta-channel (Tracer), generating the traces for the message *m1*.
8. The meta-channel Tracer delegates the message to the *receive* operation of the next one (The Default). The latter executes the default behaviour, which is executing a callback on the server-side.

## 4.3 APIs

### 4.3.1 MIAJ

The MIAJ API should enable developers to dynamically install channels in the infrastructure, providing them with a context in which to share information. In doing so, MIAJ provides the following four operations that we will detail individually below.

| | |
|---|---|
| **Subscribe (channel)** | Subscribes a channel to the MIAJ context. The channel can share information with other channels. |
| **Unsubscribe (channel)** | Unsubscribes a channel from the context. The channel loses contact with other channels. |
| **setChannelTurn (channel,message)** | When a channel executes its default behaviour, is said that is the turn of the given message. |
| **getLastTurn ()** | Gets the last turn of the last active channel |

Table 1: MIAJ API summary

- **Subscribe**. This operation installs a channel on MIAJ, setting the context of a channel for sharing information.
- **Unsubscribe**. This operation uninstalls a channel from MIAJ. Other channels will not have information about the channel uninstalled. However, the channel continues to reify the corresponding communication.
  Note that MIAJ cannot control the channel's behaviour; once a channel is instantiated the communication is immediately reified.
- **SetChannelTurn**. JavaScript is based on the event loop concurrency model. This means that two events cannot be processed at the same time as explained in section 2.2. If two events cannot be processed at the same tame, two channels will never be active at the same time. We say that a channel is active when the meta-channel Default is being executed.
  Setting a turn in MIAJ consists in saving the messages that are being processed by an active channel in a given moment. Taking up again the example of section 4.2.4; in step 2, DOM is the active channel executing the message *m,* in step 5, Socket is

the active channel processing the message *m1,* and in step 8, the active channel is Socket processing the same message but on the server side.

- GetLastTurn. Note that before a channel is active (execution of the last meta-channel), the other meta-channels could ask MIAJ for the last active channel. This method returns the oldest message processed by the last active channel.
  Keep in mind that a channel can be active by processing more than one message, for this reason every time that *getLastTurn* is called, the message returned will be not available anymore.

On the other hand, note that MIAJ should be installed on both sides of a distributed application.

### 4.3.2 Channel

The channels play the role of event emitters and event trappers. As such, they also implement the methods that represent the *send* and *receive* operation of a given web abstraction (e.g. for Socket.io are the emit and on methods respectively) for reifying the events and delegating the messages to the meta-channel. Moreover, they provide two more methods that represent the default behaviour of the channel. When the last meta-channel is executed, they seek the default behaviour of the channel and this is one of these two methods.

Therefore, a channel must implement the following methods:

| | |
|---|---|
| **setMetaChannels(mchArray)** | Sets the meta-channels of the channel from a given list of channels |
| **setContext (context)** | Sets the context of the channel |
| **getContext ()** | Gets the context of the channel |
| **send(message)** | Implementation of the default behaviour of the channel for the send operation |
| **reveive(message)** | Implementation of the default behaviour of the channel for the receive operation |

Table 2: Channel API summary

The implementation of a channel clearly depends on the complexity of the abstraction but the structure normally consists of three parts:

1. Overriding the methods that represent *send* and *receive,* in order to trap the events and reify them as messages. The overridden methods should delegate the message to the corresponding *send* or *receive* of the first of its meta-channels.
   Delegating a message consists of executing the corresponding *send* or *receive* operation of a meta-channel, passing the list of meta-channels without the first element (because this is the one that is being executed in that moment). By doing

so, it guarantees that every meta-channel will have control of the execution because it has knowledge of the list of meta-channels.

2. Implementation of the *send* and *receive* operations that represent the default behaviour of the channel will be used by the default meta-channel.
3. Implementation of three methods: *setContext* and *getContext*, for setting and getting the context of a channel, and *setMetaChannels*, for setting the list of meta-channels. This implementation is the same for all channels.

Take into account that in distributed communications such as Socket.io, the channels must be created in pairs in order to guarantee the consistency. The events are reified as messages and Socket.io does not understand messages.

### 4.3.3 Meta-channel

The meta-channels are the core of MIAJ because in a given moment they can control the execution of an event. These reify both ends of a communication channel (*send* and *receive*), and for this reason they must implement such operations. The implementation of both operations clearly depends on the necessities of the developer.

In every execution of *send* or *receive*, the meta-channel receives as parameters: a message (the reification of the event) for changing its behaviour or specializing it, receives the list of meta-channels that have not processed the given message (recall that the meta-channel should delegate the message to next meta-channel in the list), and a reference to the channel. Every meta-channel must have access to that reference for enabling access to the context of MIAJ.
If the meta-channels do not have access to this context, they will act as independent channels, and will not be able to share information.

A meta-channel, therefore typically consists of two operations:

| | |
|---|---|
| **send(message,mos,channel)** | Implements the behaviour for the send operation of the meta-channel |
| **receive(message,mos,channel)** | Implements the behaviour for the receive operation of the meta-channel |

Table 3: Meta-channel API overview

The implementation of the two operations, *send* and *receive* operations consists of two parts:

1. Implementing the custom behaviour of the meta-channel for the *send* and *receive* operations.
2. Delegating the message to the next meta-channel in the list. Except the default meta-channel because it is the last executed.

### 4.3.4 Message

Every channel produces different kinds of events with its own structure. In order to reuse the meta-channels in distinct channels, it is necessary that all the events produced by all the channels have the same structure.

All asynchronous events in JavaScript have one common property: name and data they transport. We call this the *name selector* (e.g. *book_available* or *borrow_book*).
A message therefore has three properties:

- **Selector.** As mentioned before, is the name of the event.
- **Data.** Is the data that the event transports in JSON string format, an easy way of transporting data in string format.
- **Meta-data.** Note that having only the selector of the event and the data it transports is not enough for executing the default behaviour of the message.
  One example is the DOM channel, which produces different types of events (mutation or clicks), so, when the message arrives to the default meta-channel, the behaviour is different depending on the kind of event. And the default meta-channel cannot obtain this information from the selector or from the data.
  This meta-data is stored in key-value format.

In order to deal with these properties the following methods are provided:

| | |
|---|---|
| **Message (selector,data,isSer)** | Creates a message from the selector, data, and if the parameter isSer is true, the selector is deserialized |
| **addMetaData(keyValueList)** | Saves the meta-data of a given event to the message |
| **getMetaData(key)** | Returns the value of the field in the meta-data with the given key |
| **serialize()** | Serialize a message, setting the meta-data in the selector |
| **deserialize(selector)** | Deserializes a selector, setting the meta-data in the message structure |

Table 4: Message API overview

- **AddMetaData**. Adds meta-data to the message from a key-value array.
- **GetMetaData.** Returns the corresponding meta-data from a given key.
- **Serialize.** We use the selector for transporting the meta-data in distributed channels. Therefore we say that a message is serialized when the selector contains the arguments in a certain format. We use the following format: *selector@key:val@key1:val1@keyn:valn.*
- **Deserialize**: When the data received is serialized, it must be deserialized and saved into the message structure as meta-data.
- **Constructor**. Creates a message from a selector, and the data that transport the event. Also receives an additional parameter for informing whether selector must be deserialized.

## 4.4 Implementation

In this section we are going to show the implementation of the channels supported by default in MIAJ and the meta-channel Tracer introduced in the previous sections. The implementation of MIAJ and Message can be found in Appendix A.

### 4.4.1 Message Reification

Every channel produces different kinds of events; even the same channel can produce more than one type of event such as DOM. In order to have the same structure for all the events no matter the channels that produce it, the events must be reified into messages. Thus, allowing MIAJ to reuse the implementation of the meta-channel debugger in all the channels.

In this section we are going to describe how these events are reified into messages.

#### 4.4.1.1 JQueryChannel

We identified two sources of events: clicks on elements and DOM mutation, moreover these two are incoming events (the DOM receives these events), hence, these should be delegated to the *receive* operation of its first meta-channel.

**Click on elements**. The selector of the message is the name of the DOM event (click in this case), and the data is the link information. If the link has a callback associated, the link information is that callback as string, otherwise, if the link does not have a callback associated, the link information is the target of the link (or href attribute). We can observe the two cases in the figure below.

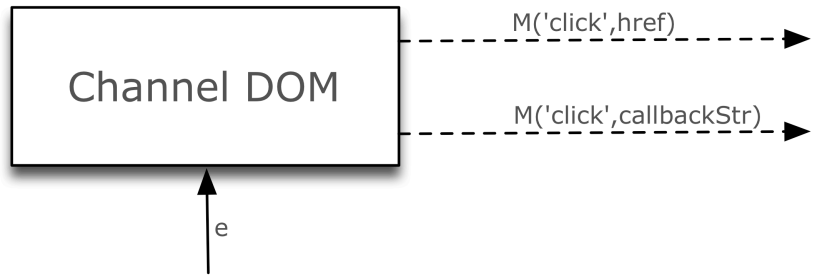


Figure 14: jQuery click events reification

Moreover, the default meta-channel will need additional information for each message in order to execute its default behaviour. That information is passed as meta-data in the message and is as follows:

- Event: type of event, only the channels have knowledge of this information. We defined two, *domin* (for incoming DOM operations) and *domout* (four outgoing DOM operations).
- IdCallback: identification of the callback for a concrete element, if the element has one or more associated.
- LinkId: if it is different than -1, determines that the link has one or more callbacks associated, when a programmer adds a callback to an element this is marked creating a new attribute in it.
- Type: type of link. Link with callback or normal link. Note that when the message arrives to the default meta-channel, it must not always do the same.
- Target: target of the link element: _blank, _self, _parent, etc.

**DOM mutation**. The selector of the message is the name of the method (html), and the data is the html string used for modifying the DOM.

As for the needed meta-data:

- Event: type of event, *domin* in this case.
- Reference: context of the concrete execution of the method, necessary for evaluating and executing the original wrapper in the default meta-channel.
- Arguments: arguments received in the method. Are necessary for evaluating the original wrapper.
- Type: identifies this event as a mutation.

#### 4.4.1.2 SocketIOChannel

The selector of the message is the event name, and the data, is the information transported by the event.

This channel only has two sources of events because it is distributed, thereby, two types of events are identified: incoming and outgoing. Both create an identical message changing the value of its only entry in the meta-data:

- Event: *emit* for outgoing events and *on* for incoming events.

#### 4.4.1.3 MySQLChannel

This channel has two sources of events that must be reified, the first, when a query is sent to the MySQL event loop, and the second when the result of a query is received.

**Queries sent**. The message created receives as arguments the identifier *query+idQuery* (selector), and the query string (data). The meta-data is:

- Event: type of event. We defined two: *emit* for the queries sent to the event-loop, and *on* for the result of those queries.

- IdQuery: unique identifier for a query; is used for matching a response when ready with the query that provokes it.
- Arguments: arguments of the method that sends a query to be processed. The default meta-channel will need it.
- Reference: context of a concrete execution of the method that sends a query to be processed. The default meta-channel will evaluate and execute the original wrapper and needs the arguments and the context for it.

**Query result**. The message created receives as arguments the identifier *queryResponse+idQuery* (selector) and the result of the query (data). The meta-data is:

- Event: type of event, this case is the type *on* because it is an incoming one.
- Lambda: when a query is sent to the MySQL event loop, a callback is passed for capturing the return value. This is a callback. Note that the only the default meta-channel should execute this callback.
- Arguments: arguments of the callback mentioned before are needed for its evaluation and execution.
- Reference: context of the callback needed for its evaluation and execution.

### 4.4.2 Channels

For showing the implementation, we will follow the steps defined by the API: override the methods that represent the *send* and *receive* operations, implement the two operations that represent the default behaviour (*send* and *receive*), and implement the operations *setContext, getContext* and *setMetaChannels*. The implementation in the third step is exactly the same for all channels and will not be shown.

In some parts of the code we use some methods such as *Array.next* or *Array.getPending*. For this reason, take into account that the method *Arrat.prototype.next()* returns the first element of an array, and the method *Array.prototype.getPending()* returns the same array without the first element.

#### 4.4.2.1 JQueryChannel

This channel is responsible for intercepting the click events and the DOM mutations.
Three kinds of behaviours can be identified: link clicks, callback clicks, and DOM mutations.

**Link clicks**. Occurs every time that a user clicks on a link, a new page is loaded. Its implementation does not require JavaScript because it is a behaviour directly implemented by the HTML code:

```
<a target="_blank" href="book_list.html">Book list</a>
```

For intercepting this kind of event, it is enough to intercept every click event and check if the element clicked is a link. We can observe in the fragment of code below that first, the default behaviour is prevented (line 2), then, the event is reified as a new message with all the meta-data described in 4.4.1, and finally, it is delegated to the first meta-channel.

```
1 else if(isLink){ //A element
2       e.preventDefault();
3       var message = new Message(type,isLink);
4       message.addMetaData({'event':'domin','idcb':-1, 'lid':null, 't':'li',
'tg':target });
5       MOS.next().receive(message,MOS.getPending(),private);
6 }
```

**Callback clicks.** The click on a certain element has a callback associated, so, when a click occurs, a callback is evaluated and executed. Performing this behaviour implies both JavaScript and HTML implementation. The code below shows an example.

```
<button id="borrow_book">Borrow book</button>
```

```
$('#borrow_book').click(function(){ //adding a callback to the button
       sendRequests();
});
```

Intercepting this event requiers two additional steps; intercepting at the moment when a programmer adds a callbak to an element, and intercepting when the programmer removes the callback added. In the code below, we can observe how calls are intercepted with the method *jQuery.fn.click* [10], which adds a callback to an element for the event click.

The method is proxied, changing in behaviour. A new attribute *localId* is added to the element that will have a callback associated (line 2) ,and the callback is saved in a global list (line 4).

```
1 jQuery.fn[info.method] = function(){//info.method = click
2       if(!this[0].localId) this[0].localId = uniqueId();
3       if(!private.callbackList[this[0].localId])
private.callbackList[this[0].localId] = new Array();
4       private.callbackList[this[0].localId].push({'ev':info.method,
'cb':arguments[0]}); //callback saved in a global list
5 }
```

After executing this, in clicking an element, if the element has the new attribute *localId*, it means that it has a callback associated, and the value of the attribute is the ID of the callback in the callback list.

Note that the element could have more than one callback associated (e.g. click, double click, right click, etc.), for this reason it is necessary to seek the callback for the corresponding event (lines 3,4,5,6 in the code below). If a callback is found, the message is created and delegated as a callback link message.

```
1 var indexCallback = -1;
2 if(element.localId){
3       var callbacks  = private.callbackList[element.localId];
4       for(ind in callbacks){
5               if(callbacks[ind].ev === type){ indexCallback = ind; break; }
6       }
7 }
8 if(indexCallback !== -1){
9       var idm = uniqueId();
10      var message = new
Message(type,private.callbackList[element.localId][indexCallback].cb.toString())
;
11      message.addMetaData({'event':'domin','idcb':indexCallback,
'lid':element.localId, 't':'cb' });
12      MOS.next().receive(message,MOS.getPending(),private);
13 }
```

**DOM mutations.** DOM mutations are executed from the JavaScript, implying a HTML modification:

```
$('#book_list').html(book_list_html);
```

The html mutation is intercepted, reifying those jQuery methods that modify the DOM. The method *jQuery.fn.html* [10] is an example. In this case, the method is proxied, changing its behaviour.

The method *jQuery.fn.html* has two behaviours; modify the DOM of an element, and examine the DOM element returning its HTML representation. We only want to reify the first type of event, and for this reason in line 3 of the code below, the default behaviour is directly executed when the second behaviour is detected.

So, if the behaviour is to modify the DOM of an element, a new message is created with the information described in 4.4.1, and delegated to the first meta-channel.

```
1 var info = {method:'html',type:'receive',cb:false};
2 jQuery.fn[info.method] = function(){
3       if(!arguments.length)  return
private.orWrappers[info.method].apply(this,arguments);
4       var message = new Message(info.method,arguments[0]);
5       var event = info.type === 'send' ? 'domout' : 'domin';
6       message.addMetaData({'event':event,'ref':this, 'args':arguments,
't':'mu'});
7       switch(info.type){
8               case 'send': MOS.next().send(message,MOS.getPending(),private);
break;
9               case 'receive':
MOS.next().receive(message,MOS.getPending(),private); break;
                default: break;
10      }
11 }
```

The *send* and *receive* operations that represents the default behaviour of the channel, should execute three different behaviours depending on the type of event, which can be DOM mutations, links or callbacks.

For mutation events, the original wrapper is called with all the meta-data provided by the channel at the moment of creating the message (line 7 of the code below). For the links event, a new window is opened with the corresponding information (line 9), and finally for the links events with a callback, the corresponding callback is evaluated and executed (lines 11 and 12).

```
this.send = function(message){
      return this.orWrappers[message.selector].apply(message.metaData.ref,
message.metaData.args);
}

this.receive = function(message){
      switch(message.metaData.t){
            case 'mu':
      this.orWrappers[message.selector].apply(message.metaData.ref,
message.metaData.args);
                  break;
            case 'li':
                  window.open(message.data,message.metaData.tg ?
message.metaData.tg : '_self');
                  break;
            case 'cb':
                  var callback =
eval(this.callbackList[message.metaData.lid][message.metaData.idcb].cb);
                  callback();
                  break;
            default: break;
      }
}
```

#### 4.4.2.2 SocketIOChannel

This channel intercepts the events produced by Socket.io on both sides of the communication. In doing so, it is enough to intercept using the *emit* and *$emit (or on)* methods of Socket.io [24].

```
1  function SocketIOChannel(socket){
2
3       this.socket = socket;
4       this.emitOr = socket.emit;
5       this.onOr = socket.$emit;
6
7       this.context = null;
8
9       var private = this;
10      var MOS = [];
11
12      socket.emit = function(){
13             var msg = new Message(arguments[0],arguments[1]),;
14             msg.addMetaData({'event':'emit'});
15             MOS.next().send(msg,MOS.getPending(),private);
16      };
17
18      socket.$emit = function(){ //$emit -> on
19             var msg = new Message(arguments[0],arguments[1],true);
20             msg.addMetaData({'event':'on'});
21             MOS.next().receive(msg,MOS.getPending(),private);
22      }
23
24      this.emit = function(selector,data){
25             socket.emit(selector,data);
26      }
27      this.on = function(selector,data){
28             socket.on(selector,data);
29      }
30
31      ...
32 }
```

First, all the references of the original socket object are saved because they are needed in the moment of executing the original behaviour (lines 3,4 and 5 in the code above). Then the methods *emit* and *on* are overridden. The method *on* for example, is overridden in line 18. The message is created with the third parameter as true, because being a distributed communication means that the selector could contain meta-data and must be decoded.

With regards to the default *send* and *receive* operations, we can see their implementation as follows:

```
this.send = function(msg){
       this.emitOr.apply(this.socket,msg.serialize());
};
this.receive = function(msg){
       this.onOr.apply(this.socket,[msg.selector,msg.data]);
};
```

In this case, executing the default behaviour consists of calling the original handlers saved before overriding the original behaviour. Note that the message is serialized before being sent in order to set the meta-data in the selector (as explained in the prior section).

#### 4.4.2.3 MySQLChannel

For executing a query with this library the method *connection.query* is used by receiving two parameters, the query string and the callback for evaluating the return value.

As Figure 15 shows, the *send* events are detected when the method *connection.query* is called the receive events, when the callback received as parameter is executed. Note that the callbacks are provided by the programmers, and for this reason all of them must be proxied for intercepting every processed query.

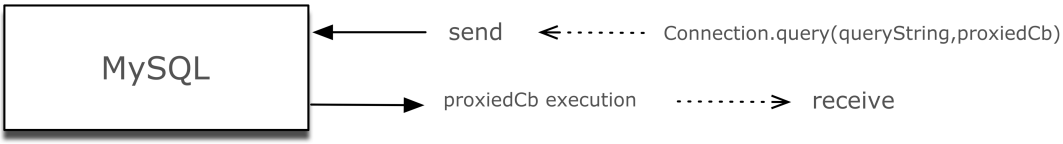


Figure 15: MySQL channel

```
1  connection.query = function(){
2
3       var idq = uniqueId();
4       var message = new Message('query '+idq,arguments[0]);
5       message.addMetaData({'event': emit,idq,'idq':
idq,'args':arguments,'ref':this});
6       MOS.next().send(message,MOS.getPending(),private);
7
8  }
9
10 this.on = function(idq,args,ref,lambda){
11      var message = new Message('queryResponse '+idq,args[1]);
12      message.addMetaData({'event': on,'lambda':lambda, 'args':args,'ref':ref});
13      MOS.next().receive(message,MOS.getPending(),private);
14 }
```

On the one hand, when the event *query* occurs a new message is created with meta-information that the default meta-channel will need. As mentioned in section 4.4.1, the parameter *idq* will be used for identifying the response of the query. We can observe in the code above that the same *idq* used for creating the identifier of the query (line 4) is the same one used for creating the identifier of the response (line 11).

On the other hand, when the method *on* is executed, it means that there is a query already processed, the parameter *idq* added in the *connection.query* identifies the source query of the response.

With regards to the methods *send* and *receive* that implement the default behaviour: the method *send* modifies the callback in order to detect when a query has been processed (line 5 and 6 in the code below) and executes the original handler of the method *connection.query* (line 9).
And the method *receive* executes the original callback received as a parameter in the moment of executing the method *connection.query*.

```
1  function MySQLChannelDefaultMO(channel){
2      this.send = function(message){
3              if(typeof message.metaData.args[1] === 'function'){
4                      var lambda = eval(message.metaData.args[1]);
5                      message.metaData.args[1] = function(){
6                      channel.on(message.metaData.idq,arguments,this,lambda);
7                      }
8              }
9      channel.connectionQuery.apply(message.metaData.ref,message.metaData.args);
10     };
11
12     this.receive = function(msg){
13             msg.params.lambda.apply(msg.metaData.ref,msg.metaData.args);
14     }
15 }
```

### 4.4.3 Meta-channels

There are two types of meta-channels, those that implement behaviours for communication and the default meta-channel, which executes the default behaviour of a channel.

Starting with the meta-channel used in the previous sections, the meta-channel Tracer, and following the steps defined by the API:

1. Implementing the custom behaviour of the meta-channel for the *send* and *receive* operations.
2. Delegating the message to the next meta-channel on the list. Except the default meta-channel.

The resulting implementation is as follows:

```
function Tracer (){
	this.send = function(message,mos,channel){
		console.log('send: '+message.selector); //behaviour
		mos.next().send(message,mos.getPending(),channel);
	}

	this.receive = function(message,mos,channel){
		console.log('receive: '+message.selector); //behaviour
		mos.next().receive(message,mos.getPending(),channel);
	}
}
```

We can observe in the code above that before delegating the message to the next meta-channel, the selector of the message is printed in both operations.

With regards to the default meta-channel, the implementation is as follows and is used in the same way for all channels. Note that each operation finally executes the *send* or *receive* operation of the channel received as a parameter (lines 5 and 12). This meta-channel also sets the given channel as active, using the API of MIAJ.

```
1  function Default(){
2  	this.send = function(msg,mos,channel){
3  		if(channel.context)
4  			channel.context.setChannelTurn(channel,msg);
5  		channel.send(msg);
6
7  	}
8
9  	this.receive = function(msg,mos,channel){
10 		if(channel.context)
11 			channel.context.setChannelTurn(channel,msg);
12 		channel.receive(msg);
13 	}
14 }
```

The same implementation is used for all the channels, this sets the channel turn because it is the active channel when it arrives to this point.

## 4.5 Deployment

### 4.5.1 Deploying MIAJ

It is enough to instantiate MIAJ it, both on the client and server-side.

```
var miaj = new MIAJ();
```

### 4.5.2 Deploying Channels on MIAJ

Once the channel is implemented and instantiated, it must be installed in MIAJ. Recall that in distributed communications such as this, the channels must be installed in pairs, one for the client and another for the server.

#### Client

```
var meta_channels = [new Tracer(),new Default()];
var socket = new SocketIOChannel(io.connect('http://127.0.0.1:8124'));
socket.setMetachannels(meta_channels);
miaj.subscribe(socket);
```

#### Server

```
var meta_channels = [new Tracer(),new Default()];
io.sockets.on('connection', function (socket) {
        socket = new SocketIOChannel(socket);
        socket.setMetachannels(meta_channels);
        miaj.subscribe(socket);
}
```

On the other hand, creating meta-channels in pairs is not mandatory because they depend directly on the behaviour of the meta-channel. For example, if the behaviour of this is tracing only the communication of the client, we do not need to install the meta-channel on the server-side.

## 4.6 Summary

The following tables show the API summary of MIAJ and the API of the elements involved in the infrastructure. The first table defines the methods that MIAJ provides in order to deal with channels and the methods for sharing information between them.

The second table shows the methods that a channel must implement, the two more complex are *send* and *receive* because these execute the default behaviour of the channel. The others are also important but their implementation is trivial and it is similar for all channels.

The third table shows the two methods (send and receive) that a meta-channel must implement; recall that the meta-channels are reifications of both ends of a communication and implement isolated behaviours.

And finally, the fourth table shows the Message API, necessary to homogenize the events produced by libraries or web-abstractions, because all of them can be different.

### MIAJ

| | |
|---|---|
| **Subscribe (channel)** | Subscribes a channel to the MIAJ context. The channel can share information with other channels. |
| **Unsubscribe (channel)** | Unsubscribes a channel from the context. The channel loses contact with other channels. |
| **setChannelTurn (channel,message)** | When a channel executes its default behaviour, is said that is the turn of the given message. |
| **getLastTurn ()** | Gets the last turn of the last active channel |

### Channel

| | |
|---|---|
| **setMetaChannels(mchArray)** | Sets the meta-channels of the channel from a given list of channels |
| **setContext (context)** | Sets the context of the channel |
| **getContext ()** | Gets the context of the channel |
| **send(message)** | Implementation of the default behaviour of the channel for the send operation |
| **reveive(message)** | Implementation of the default behaviour of the channel for the receive operation |

### Meta-channel

| | |
|---|---|
| **send(message,mos,channel)** | Implements the behaviour for the send operation of the meta-channel |
| **receive(message,mos,channel)** | Implements the behaviour for the receive operation of the meta-channel |

### Message

| | |
|---|---|
| **Message (selector,data,isSer)** | Creates a message from the selector, data, and if the parameter isSer is true, the selector is deserialized |
| **addMetaData(keyValueList)** | Saves the meta-data of a given event to the message |
| **getMetaData(key)** | Returns the value of the field in the meta-data with the given key |
| **serialize()** | Serialize a message, setting the meta-data in the selector |
| **deserialize(selector)** | Deserializes a selector, setting the meta-data in the message structure |

## 4.7 Case Study: Causeway on Top of MIAJ

In order to show the details of MIAJ, we implemented Causeway for JS programs using our framework. Recall that Causeway [25] is a message-oriented distributed debugger that provides an offline view from trace files generated by the debugging process. Basically we employ meta-channels in order to generate traces which can be visualized in the Causeway debugger.

The traces will be generated for all the events exchanged between a server and its clients using the library Socket.io. The channel used for intercepting all those asynchronous events is Socket.io implemented in section 4.4.2. We then implement a meta-channel that we call CAUSEWAY for this channel for generating the traces.

### 4.7.1 Causeway Meta-channel

The behaviour of the meta-channel Causeway generates JSON traces for each message sent and received by Socket.io channel. Implementing Socket.io-a distributed communication, the channel should be deployed both on the client and server side.

For guarantying the consistency between the distributed messages, the debugger Causeway uses an ID for identifying the same message in two different points (the message sent by a client is the same as the message received by the server) [25]. So a message sent must be marked with an ID, and messages at two different points with the same ID are thus the same message.

The code below shows the implementation of the meta-channel; following the steps defined by the meta-channel API:

1. Implement the custom behaviour: for the *send* operation, the meta-data *idc* is added to the message (line 8) and then, the corresponding trace is generated using the method *generateTrace* (line 10). This method generates a JSON with the given information.
   As for the *receive* operation, the meta-data *idc* added in the *send* operation is used for generating the *receive* trace of the same message (lines 15 and 16).
2. Delegating to the next meta-channel: once the message is processed, this should be delegated to the next meta-channel on the list (lines 11 and 17).

```
function CAUSEWAY (debugMode){
      ...
      this.generateTrace = function(selector,idc,typeOp){
            ...
      }

      this.send = function(message,mos,channel){
            var metaData = {'idc':uniqueId()};
            message.addMetaData(metaData);
            this.generateTrace(message.selector,metaData.idc,'sent');
            mos.next().send(message,mos.getPending(),channel);
      }

      this.receive = function(message,mos,channel){
            var idc = message.metaData.idc ? message.metaData.idc :
uniqueId();
            this.generateTrace(message.selector,idc,'got');
            mos.next().receive(message,mos.getPending(),channel);
      }
}
```

### 4.7.2 Applying Causeway to the Library Application

The traces were generated for the page that shows the information of a given book. These can be found in appendix B.

The following image shows the interpretation of the server messages:

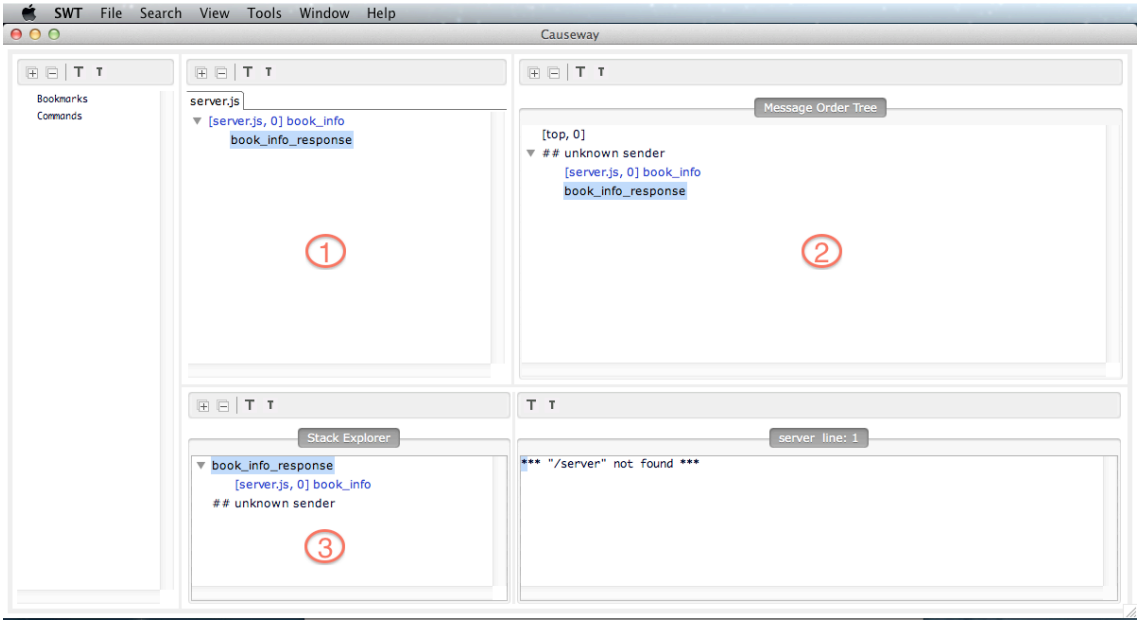


Figure 16: Using Causeway debugger, book_info_response

Selecting the message *book_info_response* in view 1 of the figure above, we can observe the message order in view 2. The message *book_info* arrived first.

On the other hand we can also observe the stack information in view 3, which is the message that triggered the selected message.

## 4.8 Conclusions

This chapter presented MIAJ, the meta-level infrastructure for asynchronous JavaScript applications, which provides the appropriate basis for reifying communication traces in programs in a simple and extensible way.

This infrastructure allows us to intercept and manipulate the distributed and non-distributed events of all the abstractions that represent communication channels such as DOM or Socket.io. By default, it supports three channels: jQueryDOM, Socket.io and MySQL for node.js applications, and two meta-channels: Causeway and Debugger (this implements the debugging features described in the next chapter).

MIAJ is important for the remainder of the thesis because apart from this element, it is possible to build tools as a debugger. Having a well-designed meta-level infrastructure becomes easier to build this kind of tool. In addition, it is an extensible framework that can deal with heterogenic debugging.

# 5 JAD: a JavaScript Asynchronous Debugger

This chapter will be focused on designing JAD – online JavaScript Asynchronous Debugger – a tool besides MIAJ for debugging asynchronous JavaScript applications. This tool is built around one central idea: to adapt the features from REME-D because both are built on an event loop concurrency model.

Despite the meta-level architecture being designed for intercepting any kind of asynchronous communication and for working in any web server, the tool, as a first prototype, has been implemented for Node.js servers. And for intercepting only three types of asynchronous events: jQuery DOM events [10]; in particular DOM mutations and click events. Socket.io events; both on the client and server side, and MySQL events.

## 5.1 Architecture

MIAJ provides, by default, implementation of four channels: the first for reifying the communication in the DOM, the following two for reifying the socket.io communications on the client and server side, and the last for reifying the MySQL asynchronous operations on the server side. JAD uses these four channels as debugging targets (Note in Figure 17 that all the channels are associated to a new meta-channel called Debugger).

JAD, therefore, is divided into two elements; two meta-channels implementing all the debugging features on the client and on the server side, and the debugger manager, which manages the debugging features of the participants within the debugging session by means of its user interface.

As we can see in Figure 17 the communication between the meta-channels and the debugger manager is bidirectional. On the one hand the actors send information to the debugger manager, and on the other hand, the debugger manager sends commands to the actors. This communication is by means of sockets.

In this implementation, since Node.js is used as a web server and MIAJ allows us to reuse meta-channels; the same implementation of the meta-channel debugger will be used on both sides of the communication. Take into account that the debugging features are implemented for an actor, and not for a specific channel, for this reason all channels must share the same meta-channel debugger on each side of the communication, otherwise we could have for example, one mailbox (messages paused in the actor) for each channel.

Accordingly, the manager is implemented as a JavaScript application. Note that the manager could be implemented for example in Visual Basic or even as a mobile application.

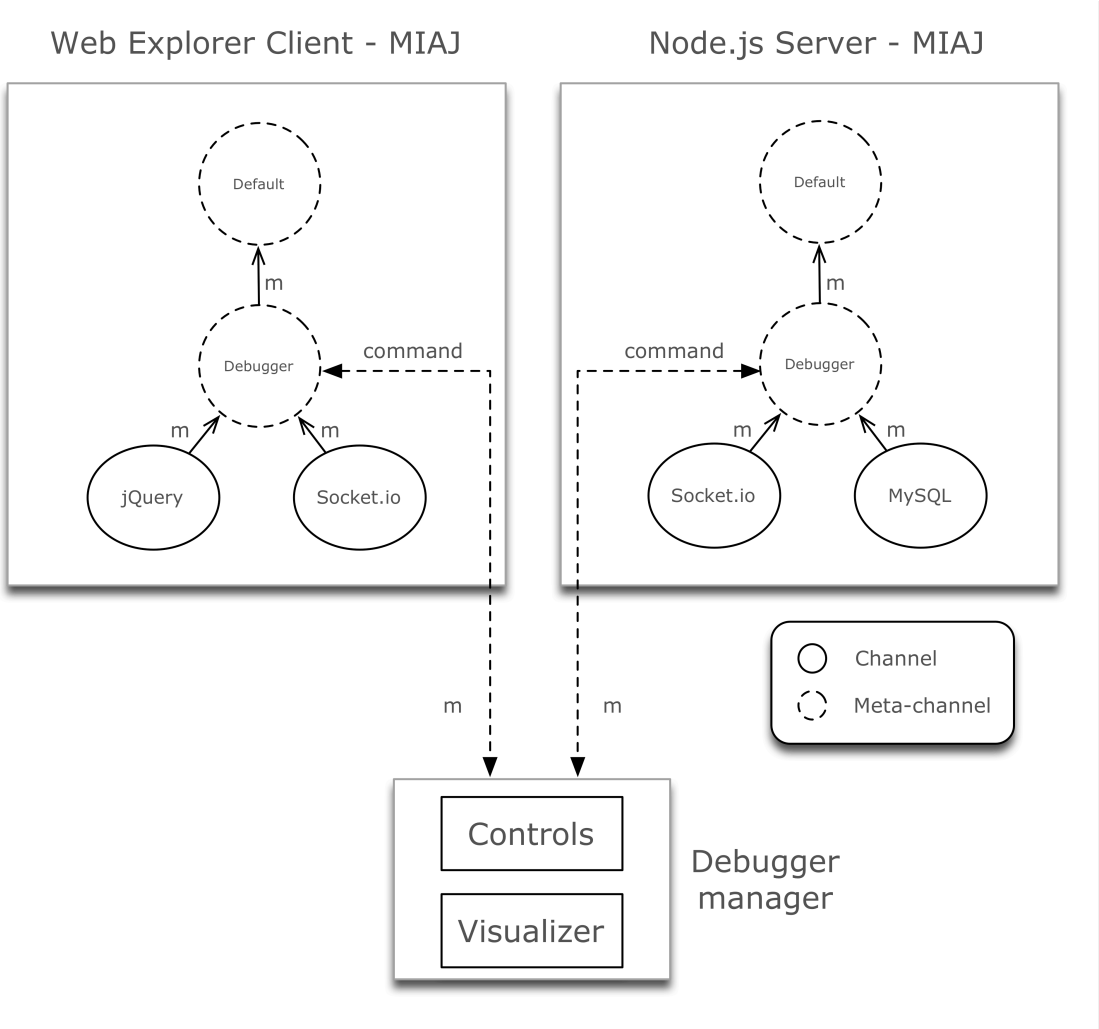


Figure 17: JAD Architecture

## 5.2 Features

In Chapter 3 we discussed and analysed which features could be useful for a tool such as JAD and which make sense or not in the context of JavaScript applications. The resulting features of this analysis are: state introspection, stepping, causal link browsing and epidemic debugging. It is known that the latter is a concept unique to ambient-oriented debuggers, but can be applied to JavaScript applications as well, as explained in chapter 1.

### State inspection

State inspection allows us to examine the actor's mailbox while the execution is paused. The actor's mailboxes are the messages that have yet to be processed by them. In distributed web applications, the client actors execute a part of the program while another part, by the server. The communication between them is by asynchronous events as in ambient talk applications, but in JavaScript there is more that one kind of asynchronous event. This does not appears to be a problem because it is possible with MIAJ to use several channels.

Even though there is only one application, for this feature the clients and servers will be treated separately because of the use different spaces of memory. Moreover, both client and server have only one single mailbox although each side can run different event-loops (or kind of events).

### Stepping

Stepping implies executing a turn of a message that has not been processed while the actor is paused or suspended. In this implementation, it is possible to execute the messages one by one (or step-by-step) in the same order of their arrival, or it is possible to select whichever and process it no matter the order of their arrival. For instance, the client mailbox could contain four suspended messages, the tool allows selecting one of those four messages and processing it. We call this last feature *step-to-message.*

### Causal link browsing

JAD is able to establish a happened-before relationship between all the events reified with channels. An example of this feature is the case of two asynchronous requests that use MySQL in the server. Those two requests are sent from a client to the server and processed one by one by the server (event-loop model), but in the moment of performing the MySQL operations, this sequencing disappears because MySQL calls are asynchronous.
In Figure 18 we can observe how it is possible to establish the relationship between events if the MySQL operations are not asynchronous.

The two messages (A and B) are queued up to the server event-loop and then processed one by one. In this instance, making the happened-before relationship would be easy because all the existing events in the server between the *send* and *receive* of the client belong to the same request (marked in grey in Figure 18).

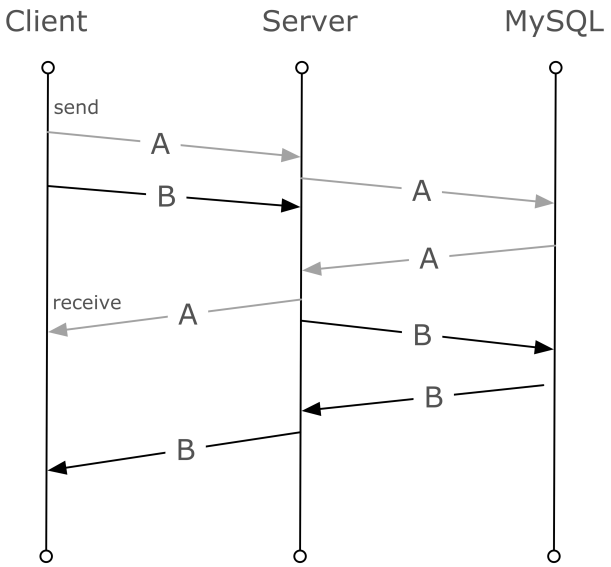


Figure 18: Link causality, one event-loop in the server

Figure 19 shows how that strategy cannot be applied when the MySQL operations are asynchronous, because of the moment the two events are queued up to the server event-loop, then immediately queued up to the MySQL event-loop, consequently losing the sequentially of the previous example. Using the same strategy, the event B marked in grey in Figure 19 would not belong to the turn of request A.

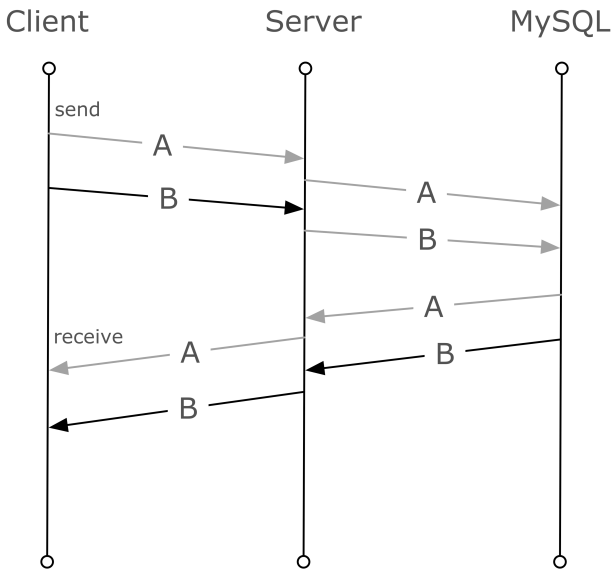


Figure 19: Link causality, two event-loops in the server

JAD can deal with both types of implementations using the methods provided by MIAJ for sharing information between channels.

#### Epidemic debugging

Epidemic debugging consists in allowing the clients to join the debugging session dynamically. In Web applications the level of volatility of the clients is quite high. With JAD it is possible to debug actors no matter if these have been disconnected several times during the debugging session. Furthermore, no special setup is needed for joining the debugging session. All the clients use the same web application.

## 5.3 Debugging JavaScript Applications

When debugging JavaScript applications, the communication between the actors that participate in the debugging session and the debugger manager must be bidirectional because on the one hand, the debugger manager sends commands to the actors, and on the other hand, the actors send information to the debugger manager in order to be managed and visualized.
The server manager, as its name indicates, is just a manager because all the features are implemented in the meta-channels. Thereby, the manager distributes the debug commands to the clients or to the server.
Two cases can be concluded with an idea offered of what each implies:

#### Sending a debug command to a client

For instance, this action could come from an event triggered by a button in the client manager; the corresponding command is generated and sent to all the actors in the debugging session using sockets. The clients or the servers are responsible for filtering whether or not they are the addressees of these commands. Thereby, every command is sent with the actor ID in order to be filtered by them.

#### Sending information from an actor to the Debugger's user interface

The meta-channel debugger sends the information to the debugger manager using sockets, and the debugger manager uses that information conveniently for being visualised. For instance, for showing the happened-before relationship between messages.

## 5.4 Implementation

MIAJ provides four channels by default, and JAD uses them as a target. We can find its implementation in section 4.4.

This section shows the implementation of the meta-channel Debugger, which implements all the debugging features, and finally the implementation of the debugger manager, which controls all those features.

### 5.4.1 Debugger Meta-channel

This meta-channel implements all the debug features on the client and server side. The same implementation is used in both sides of the communication.

The first characteristic of this debugger is the state. In JavaScript, every time a certain page is reloaded, all the values of all the variables return to its initial state, so, any local change made in variables disappears. This means that the debugger would not be able to maintain its state after reloading a page.

For resolving this problem we use LocalStorage API [19], so, for instance if a client is suspended and a certain page is reloaded, the client will remain suspended.

When a page is reloaded it is not necessary saving the mailbox of the actors because every page load implies a new context (or new start). The mailbox contains the pending messages within a context and if the page is reloaded the context changes, and the messages do not make any sense in this other context.

This meta-channel adds meta-data to every message redirected to the client manager. The information needed for knowing for example the message status, or for generating the happened-before relationship between messages. This information is:

**Client ID**. Is used for filtering the client's information in the client manager. The debugger user interface separates all the clients connected to the debugging session in different tabs. The client ID is unique for each client and server and never changes.

**Sender**. Is used for identifying who sent a request. For example when a client sends a request to the server, the server can identify which client sent that request. Needed for separating the

different requests and responses within the server context. All the requests and responses processed by the server event loop do not belong to the same client.

**Message status.** There are two possible statuses in this tool, suspended and not suspended. If the actor is suspended, all the messages sent to the client manager must be differentiated because they are the representation of the actor mailbox.

**Timestamp.** The timestamp is necessary for constructing the causality message relationship. This is irrefutable because it shows the exact moment when a message is sent.

**Message ID**. Giving to each message a unique ID enables the debugger manager to identify unequivocally a message in the mailbox. This is how the step-to-message feature is implemented; the debugger manager sends a command with the ID of the message to be processed. Can be useful as well for setting breakpoints in messages.

**Previous ID**. Unique ID used for marking all the messages that belong to a same request (or turn). For example, a DOM click implies an asynchronous request using Socket.io that causes a MySQL operation in the server. All these messages generated by all the channels, responses included, must have the same Previous ID. The request A of Figure 20 illustrates the example.

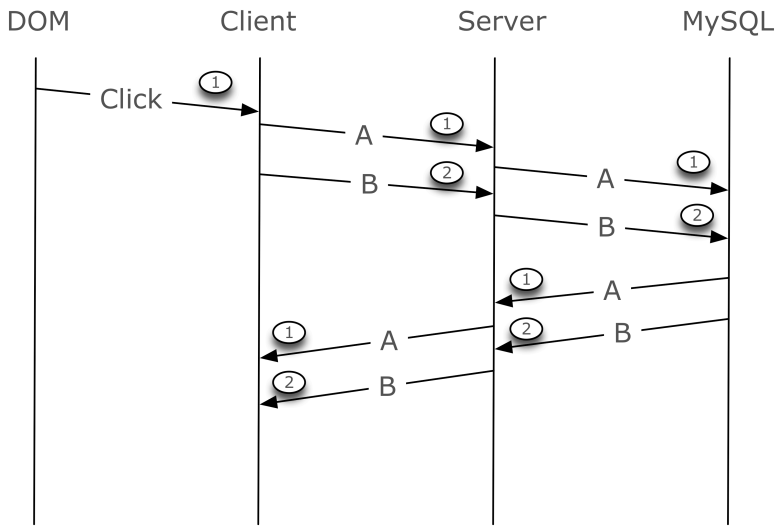


Figure 20: Establishing a turn for a message

#### 5.4.1.1 Operations send & receive

In this meta-channel, there are two differences between the *send* and *receive* operations; how the turn is established and the delegation operation (*send* delegates to *send* of the next meta-channel, and *receive* to the *receive* operation of the next meta-channel).

Therefore, in order to explain the implementation we are going to use the *send* operation as example, and explain how the turn is established in the next section.

```
1 this.send = function(message,mos,channel){
2
3
4       var validSelector = this.notSuspend.indexOf(message.selector) === -1;
5       if(message.metaData.mstatus && (message.metaData.mstatus === 'p')){
6               message.metaData['mstatus'] = 's';
7       }
8       else{
9               var idm = uniqueId();
10              this.establishTurn(message,channel,idm,serverMode,'send');
11              message.addMetaData({'cid': getClientId(), 'mstatus':'s',
'idm':idm,'ts':Date.now(),'sender': getClientId()});
12      }
13
14      var paused = false;
15
16      if(!message.metaData.mbox && (this.getStatus() ===
this.statusName.pause) && validSelector){
17              mos.unshift(this);
18              message.metaData['mstatus'] = 'p';
19              message.addMetaData({'mbox':true});
20              var pendObj = {'msg': message, 'mos': mos,
'act':'send','ch':channel};
21              this.mailbox.push(pendObj);
22              paused = true;
23      }
24
25      this.debugSocket.emit(message.serialize()[0],message.serialize()[1]);
26      if(!paused){
27              mos.next().send(message,mos.getPending(),channel);
28      }
29 }
```

Starting from the bottom in the code above (line 26), if the state of the meta-channel is suspended (or paused) the message is not delegated to the next meta-channel, but sent to the debugger manager using the *debugSocket* (line 25). Moreover, it is added to the actor's mailbox along with the meta-channels that must process that specific message and the reference to the channel that produced it.

Before adding a message to the mailbox, it is marked as *not sent* (line 18). In this way the debugger manager can differentiate the messages status. Moreover, new meta-data called *mbox* is added to the message for identifying the actors and a message from the mailbox before being finally sent.

If a message arrives at line 16, and has the parameter *mbox*, the message should not be paused in any way because that means that the debugger manager resumed it.

In the moment of adding a message to the mailbox, note that even though the meta-channel debugger has already processed the message, the same meta-channel is added to the pending list of meta-channels that must process the message (line 17 of the code). This is because the debugger manager must know when a message is added to the mailbox and when it is processed from the mailbox. In the figure below, we can observe the representation of this process.

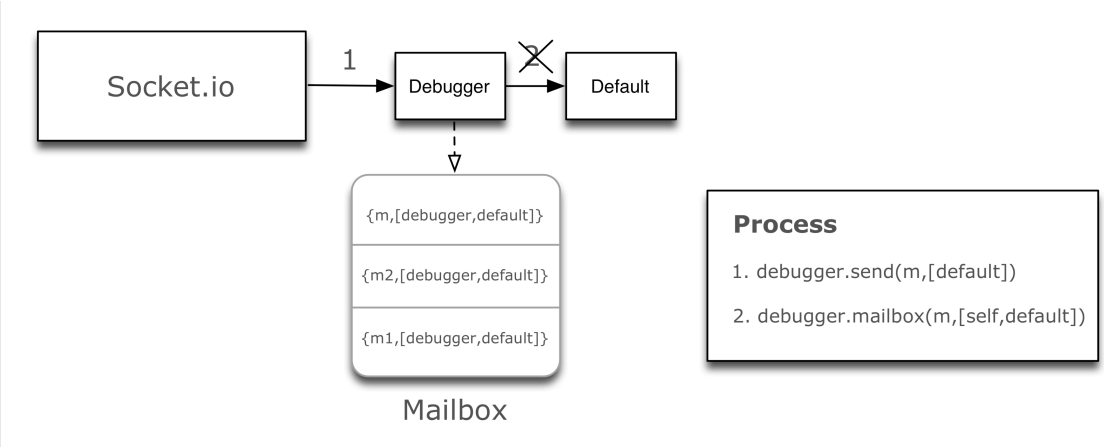


Figure 21: Pause process

Therefore, when a message arrives to this method, *send* (or *receive*) can come from two sources: the channel, which is the original event emitter and trapper or from the same meta-channel because the message was previously added to the mailbox and must be sent. If a message arrives to this point and contains the data *mstatus*, the value *p* comes from the mailbox. This data must be changed to the value *s* for informing the debugger manager that it will not be paused.

On the other hand, this debugger should provide enough meta-information for creating the happened-before relationship between messages in the debugger manager. All the messages on both sides of the communication with the same value in the meta-data *Previous ID* belong to the same turn. Along with the meta-data, *Timestamp* enables the debugger manager for establishing the causality relationship. We are going describe how the ID of the turn is generated further on (method *establishTurn* in line 10).

This meta-channel also needs to receive commands for changing its state and for performing operations related to the mailbox. In the code below we can see the implementation of the *pause* command and the *step* command.

The pause command only has to change the state of the meta-channel. Note that the same meta-channel is responsible for determining if the debug command is directed to it. Every command sent by the debugger manager contains the corresponding information.

The *step* command takes the first element from the mailbox and delegates the message to its corresponding first meta-channel. Observing this code, it is easy to imagine how the *proceed* operation is implemented; first, changing the state of the debugger to normal, then doing step on all the messages in the mailbox.

```
this.debugSocket.on('debug@pause',function(info){
       if(info.client !== getClientId()) return;
       private.setStatus(private.statusName.pause);
});

this.debugSocket.on('debug@step,function(info){
       if((info.client !== getClientId()) || !private.mailbox.length ||
(private.getStatus() !== private.statusName.pause))  return;
       var pendObj = private.next()
       private.mailbox = private.mailbox.getPending();
       if(pendObj['act'] === 'send')
pendObj['mos'].next().send(pendObj['msg'],pendObj['mos'].getPending(),pendObj['ch
']);
       else
pendObj['mos'].next().receive(pendObj['msg'],pendObj['mos'].getPending(),pendObj[
'ch']);

});
```

#### 5.4.1.2 Establishing the turn of a message

Basically consists of setting the meta-data Previous ID of the message with an ID. Messages from the same turn must have the same Previous ID as shown in Figure 20. We use the API of MIAJ for determining the last channel active and its processed turns.

Every time that the meta-channel reifies the communication, it asks MIAJ (by means of the method *getLastTurn*) which was the last active channel, and the youngest turn processed by this channel (recall that once a turn is returned, it will be not available anymore). By doing so, the message that is being processed by the meta-channel can set the meta-data Previous ID with the same Previous ID of the turn returned by MIAJ. We can observe this in lines 5, 16 and 22 for the different cases in the code below.

Distributed received messages do not need to obtain the turn because a message sent is the same message received in the other point of the communication. We can observe this case in line 27, the turn is asked, but not used for setting the Previous ID of the message.

As we can observe, the process is always the same depending on the case (i.e. if the message is distributed or if there was no active channel before). In the code below, we can observe the full implementation:

```
1 this.establishTurn =
function(message,channel,initialIdm,server_mode,operation){
2       var prev = initialIdm;
3       if(operation === 'send'){
4               if(server_mode){
5                       var tmp = channel.getContext().getLastTurn();
6                       prev = tmp ? tmp.metaData.prev : initialIdm;
7               }
8               else channel.getContext().getLastTurn();
9               message.metaData['prev'] = prev;
10      }
11      else if(operation === 'receive'){
12              if(!message.metaData['prev']){
13                      var prev = message.metaData['idm'];
14                      if(server_mode){
15                              if(message.metaData['sender'] === 'server'){
16                                      var tmp =
channel.getContext().getLastTurn();
17                                      prev = (tmp ? tmp.metaData.prev :
message.metaData['idm'])
18                              }
19
20                      }
21                      else{
22                              var tmp = channel.getContext().getLastTurn();
23                              prev = (tmp ? tmp.metaData.prev :
message.metaData['idm'])
24                      }
25                      message.metaData['prev'] = prev;
26              }
27              else channel.getContext().getLastTurn();
28      }
29 }
```

**Send operation.** On the server side, the *send* operation can be executed in the main event-loop (e.g. Socket.io operations) or in the event-loop for processing I/O operations (e.g. MySQL operations). For instance, at the moment of processing a query (second event-loop), the turn of the last active channel must be examined (which asynchronous operation triggered that query) for setting the turn of the query (must be the same). In this case the turn before comes from the main event-loop, which used the *receive* operation of Socket.io. This behaviour is implemented from line 3 to 10 of the code above.

Note that when a *send* operation does not detect a previous active channel from which to obtain the turn, the Previous ID is created for the first time.

**Receive operation**. On the one hand if the message received already contains the parameter Previous ID, it means that the message is distributed (it is not necessary to establish the turn of the message). Otherwise, if the message arrives to the server, it could come from MySQL, and the last turn must be examined (line 16). And if the message arrives to the client, it could come from the Socket.io channel, the last turn must be examined as well (line 22).

### 5.4.2 JAD from a User's Perspective

The debugger manager is divided into two elements: a *controller*, which sends all the commands to the actors within the debugging session and the *user interface,* which has two views, one to show all the actors within the debugging session and its corresponding messages, and the second for viewing the happened-before relationship of the messages of a concrete actor in live.

#### 5.4.2.1 Debugger User Interface

The first view shows all the clients connected and the server by means tabs. These tabs are created dynamically when the clients join the debugging session. The messages can be observed along with timestamp and by clicking on one of them; it is possible to observe its data. This is possible because the meta-channel debugger sends the information described in section 5.4.1 in every message. With this information the controller of the application has only to present these messages in the user interface (no special computation is needed).

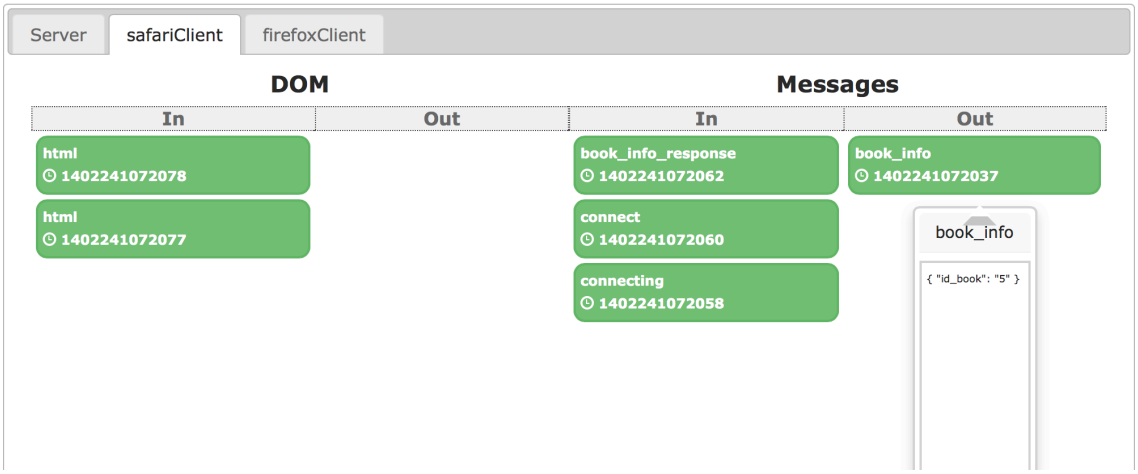


Figure 22: Debugger manager default client view

The first time that a client joins the debugging session, the name of the tab is an ID of 21 bytes-not that easy to remember; for this reason the name of the tabs can be modified at any moment.

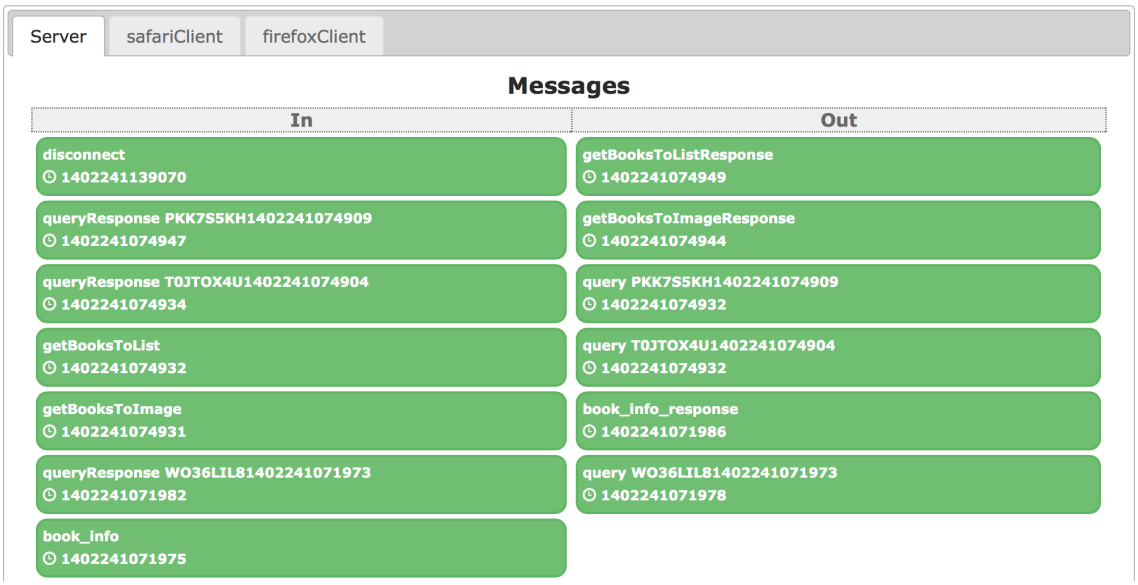

Figure 23: Debugger manager default server view

In the server tab we can observe all messages from all the clients within the debugging session.

On the other hand, double clicking on any message, the second view will be shown creating the happened-before relationship between messages for the actor selected.

In this view the messages from the client and server are not separated, and only the server messages related with the client selected are shown. The timeline is from top to the bottom.

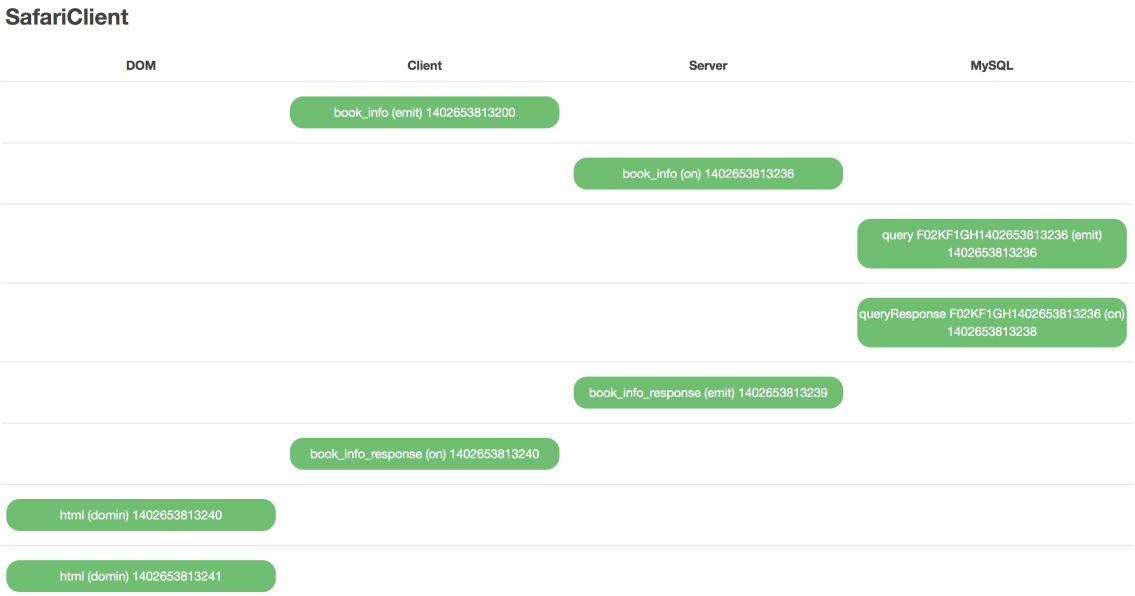

Figure 24: Debugger manager happened-before relationship between messages

#### 5.4.2.2 Controller

The debugger manager is able to process four kinds of commands listed and described in the table below:

| | |
|---|---|
| **Pause** | Sets the actor state as suspended, the event loop will not process any message, queuing it to the mailbox. |
| **Proceed** | Sets the actor state as normal, the event loop will process all its messages normally. And if there are messages in the mailbox, these are also processed. |
| **Step** | Processes the older message in the mailbox when the actor is suspended. |
| **Resend** | Processes a selected message of the mailbox without taking into account the order that arrived. |

Table 5: Debugger manager commands

As we can observe in the implementation, all the features are implemented in the meta-channel. The debugger manager only commands what the meta-channels have to do by means of the debugging socket.

```
function pause (){
        socket.emit('debug@pause',{'client': clientSelected});
        setPannelInfo(clientSelected);
}
function resume(){
        socket.emit('debug@proceed',{'client': clientSelected});
        setPannelInfo(clientSelected);
}

function next(){
        socket.emit('debug@next',{'client': clientSelected});
}

function resumeMessage(idm){
        socket.emit('debug@resend',{'client': clientSelected, 'message':
        cleanMessageId(clientSelected,idm)});
}
```

When the command pause is sent, this is notified in the panel of the actor (dark grey panel) and all the messages are shown in orange colour for identifying the messages in the actor mailbox.

There are three ways for processing the messages in the actor mailbox, clicking on the button *resume*, clicking on the button *next* (the first message in the mailbox will be processed) and clicking on the button inside the message (then no matter its position in the mailbox, it will be processed). Every time that a message from the mailbox is processed, the colour changes from orange to green because those are not in the actor mailbox anymore.

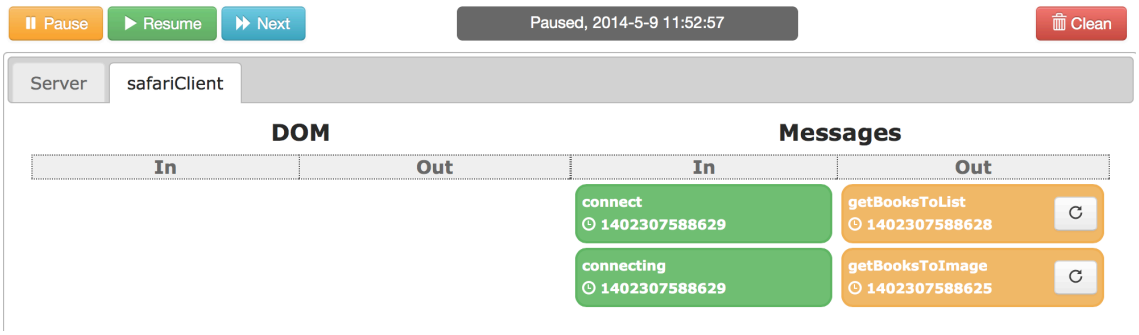


Figure 25: Debugger manager controls

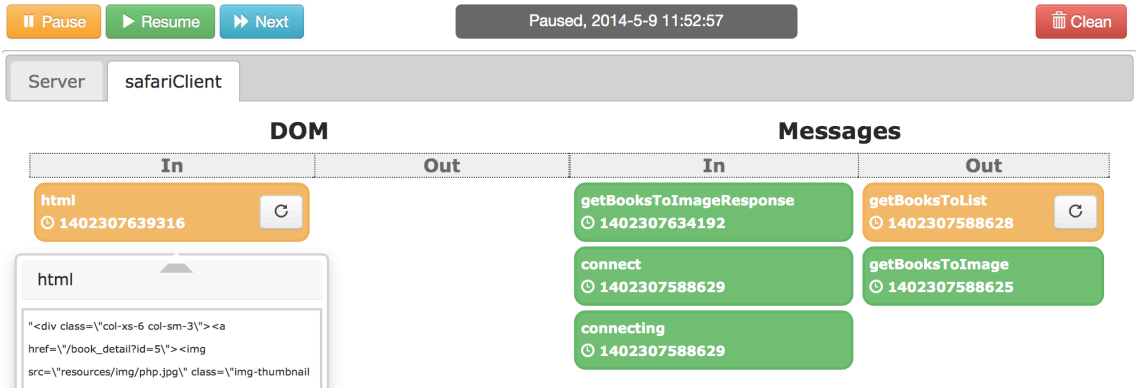


Figure 26: Debugger manager controls part II

Finally the red button with the text “Clean” cleans the messages of the selected tab.

## 5.5 Employing JAD in the Library Application

### 5.5.1 Scenario

A web for lending books is manifesting an error in the main page. The page shows a spinner at the top and no JavaScript errors are reported. We can see a representation in the figure below.

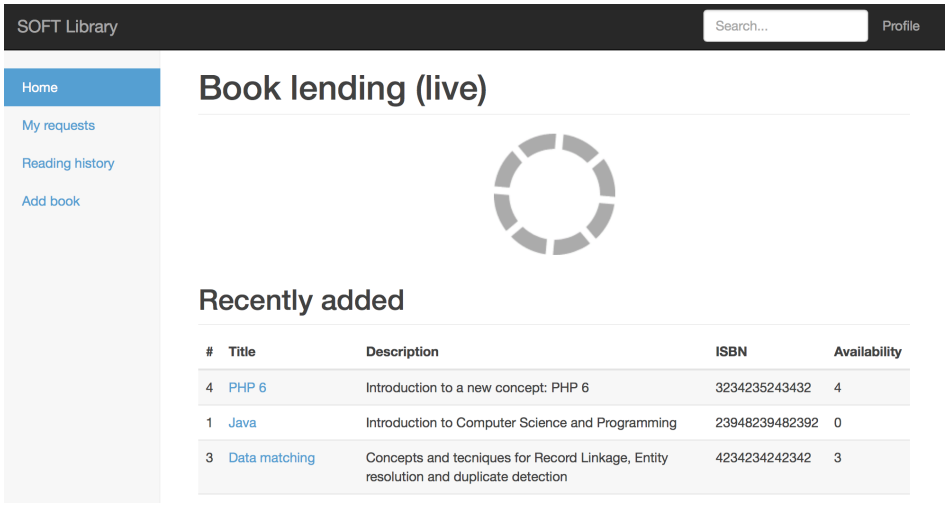


Figure 27: Library web application, bug detected

We decide to use JAD for trying to locate and solve the detected bug.

### 5.5.2 Debugging Process

The first step for debugging the application is installing the meta-level infrastructure. For the clients, it is necessary to add the next setup to all the pages that will be debugged:

```
var miaj = new MIAJ();

var metaChannels = [new Debugger(false),new Default()];

var jquerych = new jQueryChannel();
jquerych.setMetaChannels(metaChannels);
miaj.subscribe(jquerych);

var socket = new SocketIOChannel(io.connect('http://127.0.0.1:8124'));
socket.setMetaChannels(metaChannels);
miaj.subscribe(socket);
```

The meta-channel Debugger receives a parameter for instantiating it. If the value of the parameter is true, it means that it will be an instance used on the server side. There are some details of implementation that are not alike in both sides, such as the local storage.

And for the servers:

```
var miaj = new MIAJ();

var metaChannels = [new Debugger(true),new Default()];

var mysql = new MySQLChannel(connection);
mysql.metaChannels(metaChannels);
miaj.subscribe(mysql);

io.sockets.on('connection', function (socket) {
        socket = new SocketIOChannel(socket);
        socket.metaChannels(metaChannels);
        miaj.subscribe(socket);
        //now is possible to use socket
});
```

With this installation, the server and every client connected will join the debugging session dynamically. Now, in order to run the debugger manager that we implemented as a web application, it is necessary to execute the following command in the folder where the debugger manager application is located.

```
> node debuggerManager.js
```

We decide to pause the execution of the application, and start the application again for debugging the message *getBooksToImage* (this is the message that requests the information not shown) in order to find the bug. The following image shows what we see in the debugger manager default view.

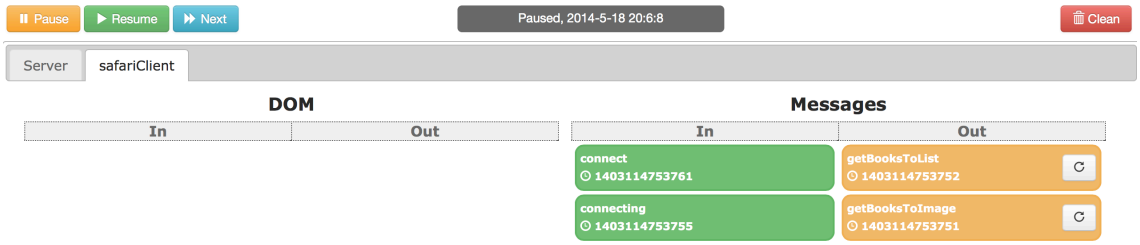


Figure 28: JAD in action. Debugging a client message

Using the feature step-to-message we process first the *getBooksToImage* request, and the response of the server was *getBooksToListResponse* (note that the responses can have any name, so, it is possible that *getBooksToListResponse* is the corresponding response of the request *getBooksToImage*).

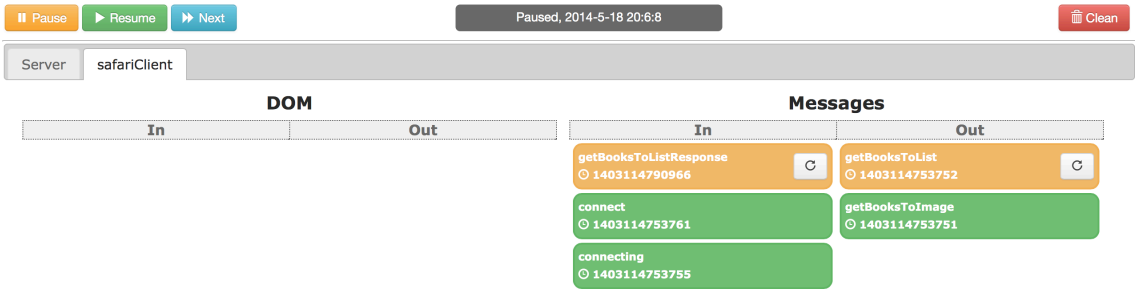


Figure 29: JAD in action. Debugging a client message part II

Processing this last response (shown in Figure 29) and examining the html that this generates (shown in Figure 30), we realize that the html seems to be the body of a table.

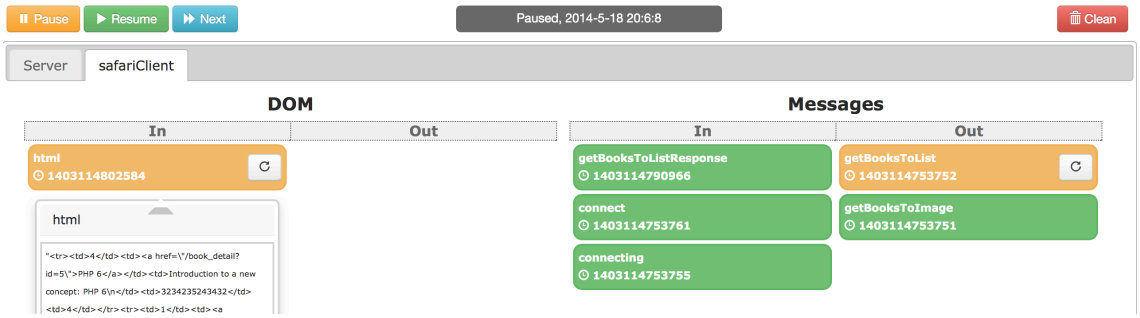


Figure 30: JAD in action. Debugging a client message part III

We know, as programmers of the application that the not shown part of the page is not a table. So far, the problem can come from the client who is generating the html incorrectly or from the response received by the client, this being incorrect.

We decided to go beyond and debug on the server side. We change the state of the client to normal and we pause the server for debugging the message *getBooksToImage*.

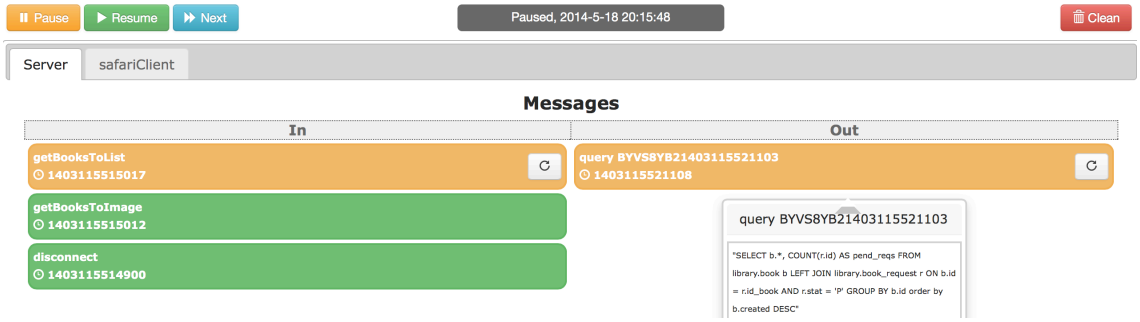


Figure 31: JAD in action. Debugging a server message

Doing once again step-to-message, we can observe that the message executes a query in the server, but we realize, as programmers of the application that this is not the proper query, actually when the message *getBooksToList* is processed the two queries are the same. Then, processing all the messages in the mailbox we can observe as well, that the server generates the same response twice (*getBooksToListResponse*).

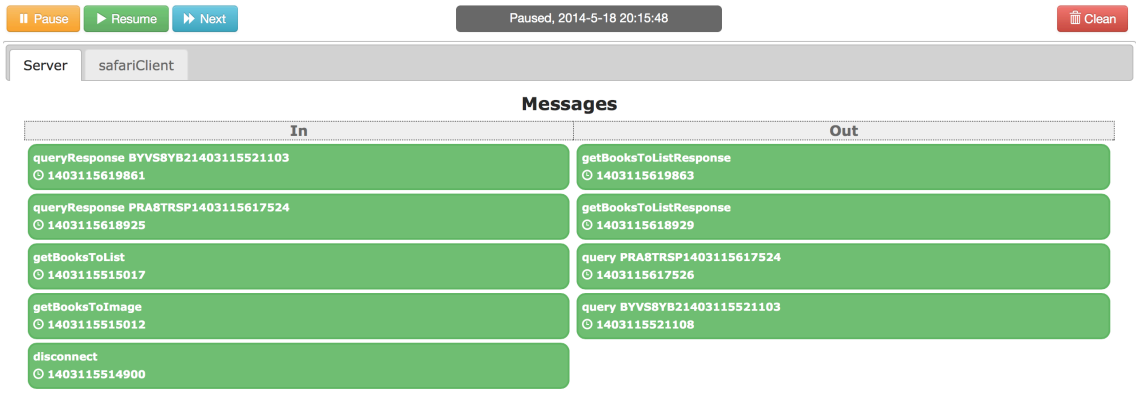


Figure 32: JAD in action. Debugging a server message part II

So far, with the information provided by the debugger, we can conclude where is exactly the bug. The bug is located in the server because this emits two responses similar to receiving two different requests. More precisely, this is exactly located in the part of the code when the server receives the request *getBooksToImage,* because it is the request that generates the wrong response.

We check this part of code and indeed, when the server receives the request; it executes the behaviour of the other message (*updateIndexListBySocket),* being the correct *updateIndexImageBySocket* behaviour.

```
socket.on('getBooksToList', function (data) {
        updateIndexListBySocket(socket,data,false);
});

socket.on('getBooksToImage', function (data) {
        updateIndexListBySocket(socket,data,false);
});
```

Finally, fixing the code in the server and executing the application again, we observe that the page works perfectly.

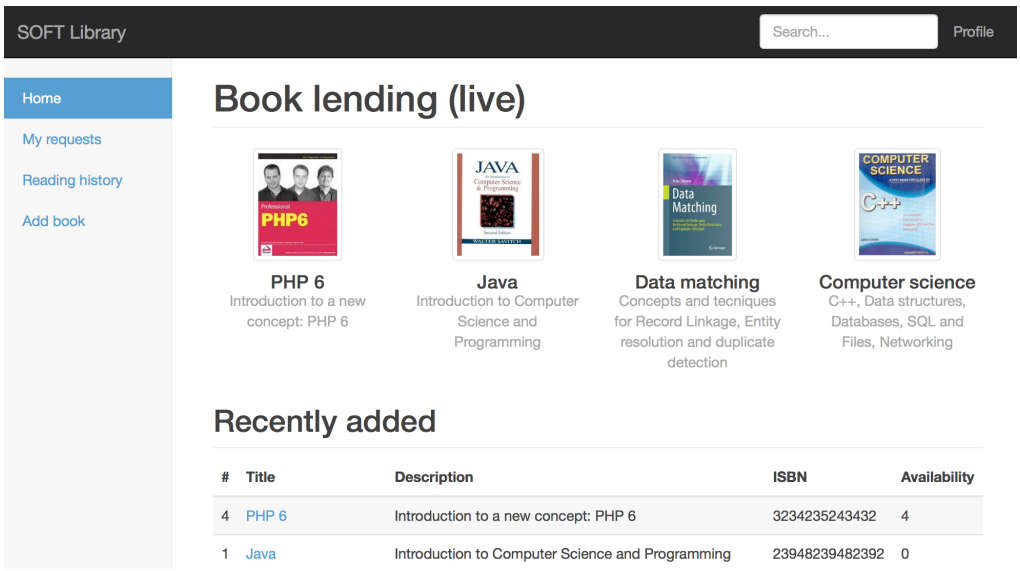


Figure 33: Library web application, bug found and fixed

Note that directly using the detail view; we can arrive at the same conclusion, even without pausing the application.

In the representation of the view shown in the figure below, we can observe that the two requests have two identical responses, even in the HTML generated for modifying DOM.

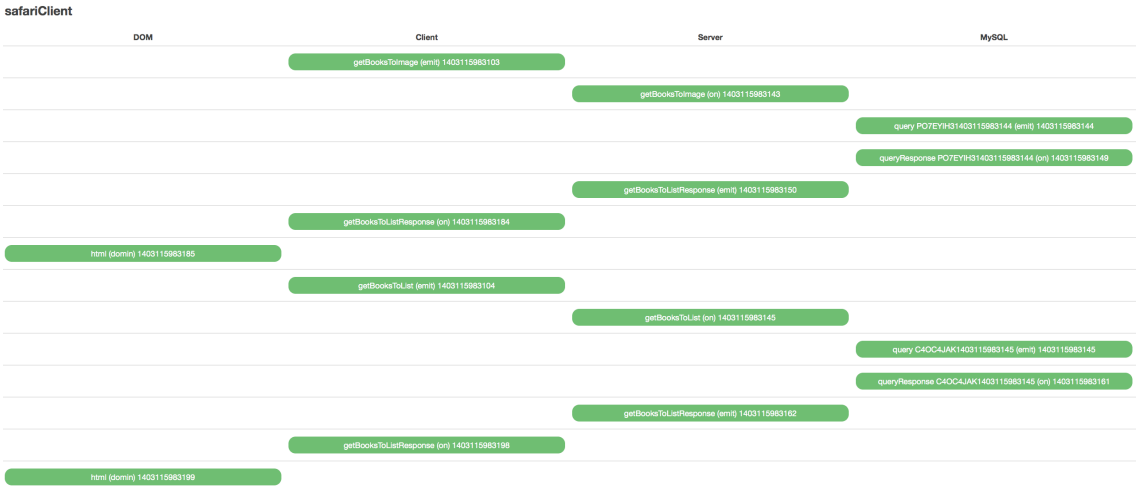


Figure 34: JAD in action. Detail view

## 5.6 Conclusions

This chapter presented the first prototype of the debugging tool implemented on top of MIAJ. The tool meets all the criteria we discussed in chapter 1.

- It is an online debugger, allowing developers to debug when the application is running.
- Is message-oriented, allowing us to generate the happened-before relationship between the messages.
- Supports distributed messaging.
- Is web-oriented.
- Works on servers based on the event-loop concurrency model.

This is the first implementation that meets all these criteria; it is a first step for creating a tool even more versatile in ending the problem of debugging asynchronous applications in JavaScript and its communicating event-loops.

# 6 Conclusion

## 6.1 Summary

The importance of debugging support in asynchronous JavaScript applications is self-evident. With the emergence of AJAX, JavaScript applications based on the event-loop concurrency model introduced at the same time new solutions for implementing web applications. Some problems arose because the applications became more difficult to understand and maintain. Moreover, web-based applications introduce different event loops processing different types of events. By analysing these problems, we could determine the challenges of debugging this kind of applications and we identified three: message-oriented, heterogeneity and open debugging.

**Message-oriented.** In asynchronous applications the control flow is divided into two parts, the request process and the reception process. Establishing the happened-before relationship between these two parts is not that easy as in sequential programming, because the stack of the program is always empty when processing a message (or turn), which means that every message is treated independently when processed in an event loop.

**Heterogeneity**. Web-based applications use different types of events generated by different technologies such as DOM events (e.g. clicks, DOM modification) or asynchronous remote events, all of them being processed by the JavaScript event-loop. Having different types of events makes the debugging process difficult because each one has a different structure

**Open Debugging**. The frequent disconnections of web applications can complicate the debugging process because it could cause a loss of messages. In such cases, the debugger must be able to keep control of all the messages processed by an actor while disconnected from the debugging session.

We also analysed the current tools for debugging JavaScript and none of them overcome such challenges. Tools that only allow debugging in the client side, such as Firebug or FireDetective, which allows us debugging on both sides of the communication are only for AJAX-based communication and always after the program execution.

## 6.2 Our Approach

With the aim of overcoming the aforementioned challenges, we present two levels of solutions, a reflective model that we call MIAJ, and JAD, a debugger for distributed asynchronous JavaScript.

### 6.2.1 Meta-level Engineering in JavaScript

Based on the ideas of the classic channel-oriented reflective frameworks [1] and transmitter-receptor model from AmbientTalk's/M language [3], we present MIAJ, a Meta-level Infrastructure for Asynchronous JavaScript.

This infrastructure reifies the asynchronous communications based on communicating event-loops present in JavaScript applications. This infrastructure deals with the heterogeneity of JS applications, reifying all types of events as first-class objects called Messages, and reifying the extremes of the communications (send and receive) by means of the use of meta-objects called meta-channels. By manipulating these meta-channels or creating news, developers can modify and specialize the behaviour of the events.

MIAJ also provides support for establishing the happened-before relationship between messages, by means of sharing information between the channels. For example, a given channel could know which was the last active channel, and the different turns processed.

By default MIAJ supports the following channels: DOM communication based on jQuery, distributed asynchronous communication based on Socket.io, and distributed communication based on MySQL.

### 6.2.2 An Online Message-oriented Debugger for JavaScript

JAD. Is a debugger built on top of MIAJ that adapts the features of the REME-D debugger because both are built on an event loop concurrency model. This implements the traditional debugging features such as state inspection, stepping, and causality link browsing and open debugging.

- **State inspection**. Enables developers to pause the execution of a program, and allows examining the actor's mailboxes, which are the messages that are paused. For this feature the clients and server use different mailboxes because they remain in different spaces of memory. Moreover each actor has only one mailbox although the application processes different kind of events.
- **Stepping**. This tool provides two features for processing a message from the mailbox, step-by-step (processes the older message in the mailbox), and step-to-message (processes a message selected).
- **Open debugging**. Can deal dynamically with disconnections and reconnections of the actors to the debugging session.
- **Causality link browsing**. The debugger allows us to establish the happened-before relationship between all the events reified as messages.

The implementation of the debugger consists in a meta-channel implementing all the debugging features previously described, and a debugger manager for controlling and visualizing these features.

## 6.3 Limitations and Future work

This thesis introduces MIAJ, a reflective architecture for dealing with asynchronous JavaScript applications. This architecture provides, by default, support for three channels, jQuery DOM (only clicks and HTML modification), Socket.io (for both sides of the communication), and Node.js for MySQL. However, on the one hand, the idea is to extend the functionalities of the Channel jQuery DOM for reifying a part of the click events and DOM mutations, and other kinds of events such as double click, mouse over, keyboard interactions, append, prepend, etc. [10].

And on the other hand, designing and implementing more channels for reifying web abstractions and libraries such as:

- Page load and reload
- Timers
- Asynchronous messaging: AJAX, Promises, etc.
- Script invocations
- Resources load

The pretention is reifying as much abstractions and libraries as possible in order to provide a complete debugging support.

With regards to the debugger, which is a meta-channel on top of MIAJ, the next step is designing and implementing three more features:

- Setting breakpoints in the messages. While examining the mailbox of a client, should be possible in selecting a message and setting a breakpoint in it, thus, when the execution will be resumed, the execution will stop in the message that contains the breakpoint.
- Modifying messages of the mailbox at run time. Being able to modify the data of a message in the mailbox, for example, to modify the query before executing it with MySQL, whether it is for checking the SQL syntax or for changing the results.
- Modifying messages processed at runtime. The idea is to modify the client DOM by resending messages already processed. For example, to modify the data of a DOM mutation message already processed, and consequently modifying the DOM in the client by resending the given message.

The debugger manager also needs some improvements in the user interface, however, this being a prototype for validating the debugging features, it is not clear how it will finally be implemented. As a Firefox extension? As a plugin for a JavaScript editor such as Eclipse? The response to this question requires a great analysis for validating it such as in [32] or [13].

Finally, the implementation of MIAJ can only be installed in JavaScript applications using Node.js in the server side. One step beyond would be to implement MIAJ for other server-side runtimes.

## 6.4 Contributions

The key contribution of this thesis is the definition of a Meta-level architecture (MIAJ) on top of the web abstractions for asynchronous JavaScript applications, for reifying the communication traces. This model provides support for dealing with the heterogeneity problem of the web applications (where the asynchronous messages can be of different types), and provides support for establishing the causality relationship between messages (or communication channels).

By combining the ideas of the channel reification model [1] and the transmitter/receptor model from AmbientTalk's/M language [3], MIAJ defines three components channels (reifying different kinds of communication (e.g. implicit event loops)), meta-channels (reifying both ends of communication) and messages (reifying the events sent and received in the JS as first-class objects). By modifying the meta-channels or creating new ones, developers can modify or specialize the behaviour of an event.

This first contribution lead us to the second one, which is an application of this model for capturing the interactions of some JavaScript technologies such as Socket.io (for remote communications in both extremes), jQuery DOM (for interactions with the DOM using jQuery), and MySQL for Node.js. These technologies are reified as Channels.

For applying this model, two meta-channels were also created, the first for generating the file communication traces used by the debugger Causeway. And the second implementing debugging features such as state inspection, stepping and link causality. This latter comes along with another application called debugger manager that controls the debugging features in the actors by means of a user interface.

We call JAD the combination of the debugger manager and the meta-channel that implements the debugging features. To the best of our knowledge, JAD becomes the first online debugger for distributed asynchronous JavaScript applications.

# 7 References


[1] Ancona, M., Cazzola, W., Dodero, G., & Gianuzzi, V. (1998). Channel reification: A reflective model for distributed computation.

[2] FireQuery. https://www.binaryage.com [Online; accessed June-2014].

[3] Boix, E. G. (2012). Handling partial failures in mobile ad hoc network applications: From programming language design to tool support., 55-105.

[4] Gonzalez Boix, E., Noguera, C., & De Meuter, W. (2014). Distributed debugging for mobile networks. J.Syst.Softw., 90, 76-90.

[5] B. Liskov and L. Shrira. Promises: linguistic support for efficient asyn- chronous procedure calls in distributed systems. In Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation, pages 260–267. ACM Press, 1988.

[6] B. Liskov, “Distributed programming in Argus,” Communications Of The ACM, vol. 31, no. 3, pp. 300–312, 1988.

[7] Chalk, S., & Donat, M. (2003). Debugging in an asynchronous world.

[8] Chrome DevTools. https://developer.chrome.com/devtools/index [Online; accessed June-2014].

[9] G. Kiczales, J. D. Rivieres, and D. G. Bobrow. The Art of the Metaobject Protocol. MIT Press, Cambridge, MA, USA, 1991.

[10] Jquery. http://jquery.com [Online; accessed June-2014].

[11] K. Zyp. http/::wiki.commonjs.org:wiki:Promises:A, 2009. [Online; accessed June-2014].

[12] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. Commun.ACM, 21(7), 558-565.

[13] Layman, L., Nagppan, M., Singer, J., & DeLine, R. Debugging revisited.

[14] Lerner, B. S., Carroll, M. J., Kimmel, D. P., La Vallee, H. Q., & Krishnamurthi, S. (2012). Modeling and reasoning about DOM events. Proceedings of the 3rd USENIX Conference on Web Application Development, Boston, MA. pp. 1-1.

[15] Lieber, T. (2013). Theseus: Understanding asynchronous code. CHI '13 Extended Abstracts on Human Factors in Computing Systems, Paris, France. pp. 2731-2736.

[16] M. Miller, E. D. Tribble, and J. Shapiro. Concurrency among strangers: Programming in E as plan coordination. In Symposium on Trustworthy Global Computing, volume 3705 of LNCS, pages 195–229. Springer, April 2005.

[17] Xmlhttprequest. https://developer.mozilla.org/es/docs/XMLHttpRequest [Online; accessed June-2014].

[18] Ajax. https://developer.mozilla.org/es/docs/AJAX [Online; accessed May-2014].

[19] Storage. https://developer.mozilla.org/en-US/docs/Web/Guide/API/DOM/Storage [Online; accessed April-2014].

[20] Node.js. http://nodejs.org [Online; accessed February-2014].

[21] Firebug. https://getfirebug.com [Online; accessed May-2014].

[22] Q. http://documentup.com/kriskowal/q [Online; accessed June-2014].

[23] MySQL for node.js. https://github.com/felixge/node-mysql [Online; accessed February-2014].

[24] Socket.io. http://socket.io [Online; accessed February-2014].

[25] Stanley, T., Close, T., & Miller, M. S. (2009). Causeway: A message-oriented distributed debugger (Technical No. 78). HP Laboratories:

[26] Boix, E. G., Noguera, C., Van Cutsem, T., De Meuter, W., & D'Hondt, T. (2011). REME-D: A reflective epidemic message-oriented debugger for ambient-oriented applications. Proceedings of the 2011 ACM Symposium on Applied Computing, TaiChung, Taiwan. pp. 1275-1281

[27] Cutsem, T. V., Mostinckx, S., Boix, E. G., Dedecker, J., & Meuter, W. D. (2007). AmbientTalk: Object-oriented event-driven programming in mobile ad hoc networks. Proceedings of the XXVI International Conference of the Chilean Society of Computer Science, pp. 3-12.

[28] Van Cutsem, T., & Miller, M. S. (2013). Trustworthy proxies: Virtualizing objects with invariants. Proceedings of the 27th European Conference on Object-Oriented Programming, Montpellier, France. pp. 154-178.

[29] Van Cutsem, T., & Miller, M. S. (2010). Proxies: Design principles for robust object-oriented intercession APIs. SIGPLAN Not., 45(12), 59-72.

[30] Waldo, J., Waldo, J., Wyant, G., Wyant, G., Wollrath, A., Wollrath, A., et al. (1994). A note on distributed computingIEEE Micro.

[31] Wang, H. J., Fan, X., Howell, J., & Jackson, C. (2007). Protection and communication abstractions for web browsers in MashupOS. SIGOPS Oper.Syst.Rev., 41(6), 1-16.

[32] Zaidman, A., Matthijssen, N., Storey, M., & Van deursen A. (2012). FireDetective: Understanding ajax Client/Server interactions (Technical No. 4). Delf University of Technology: Software Enginnering Research Group.

# Appendices

## 7.1 Appendix A

### A.1. MIAJ Implementation

```
function MIAJ(){
	var activeChannel = new AssociativeArray();

	this.subscribe = function(channel){
		channel.setContext(this);
		activeChannel[channel.constructor.name] = {'ac':0, m:new Array()};
	}

	this.unsubscribe = function(channel){
		channel.setContext(null);
	}

	this.setChannelTurn = function(ch,msg){
		if(activeChannel[ch.constructor.name].ac === 1){
			activeChannel[ch.constructor.name].m.push(msg);
		}
		else{
			for(i in activeChannel){
				if(activeChannel.hasOwnProperty(i)) {
					activeChannel[i].ac = 0;
					activeChannel[i].m = new Array();
				}
			}
			activeChannel[ch.constructor.name].ac = 1;
			activeChannel[ch.constructor.name].m = [msg];
		}

	}

	this.getLastTurn = function(){
		for(i in activeChannel){
			if(activeChannel.hasOwnProperty(i) && activeChannel[i].ac
=== 1){
				var res = activeChannel[i].m.next();
				activeChannel[i].m =
activeChannel[i].m.getPending();
				return res;
			}
		}
		return null;
	}
}
```

## A.2. Message Implementation

```
function Message(selector,data,rec){
	this.selector = selector;
	this.data = data;
	this.metaData = {};

	this.metaDataSeparator = '@';
	this.keyValueSeparator = ':';
	this.allowedDistributedMetaData = ['string','number','boolean'];

	if(rec && selector) this.deserialize(selector);
}


Message.prototype.addMetaData = function(p){
	for(i in p){
	   this.metaData[i] = p[i];
	}
}

Message.prototype.getMetaData = function(key){
	return this.metaData.hasOwnProperty(key) ? this.metaData[key] : null;
}

Message.prototype.serialize = function(){
	var newArgs = [this.selector,this.data];
	for (var k in this.metaData){
		if (this.metaData.hasOwnProperty(k)) {
			var tParam = typeof this.metaData[k];
			if (this.allowedDistributedMetaData.indexOf(tParam) !== -
1){
				newArgs[0] +=
this.metaDataSeparator+k+this.keyValueSeparator+this.metaData[k];
			}
		}
	}
	return newArgs;
}

Message.prototype.deserialize = function(selector){

	var parts = selector.split(this.metaDataSeparator);
	this.selector = parts[0];
	for(var i = 0; i < parts.length; i++){
		if(i === 0) continue;
		var keyValue = parts[i].split(this.keyValueSeparator);
		if(keyValue.length > 1){
			this.metaData[keyValue[0]] = keyValue[1];
		}
	}
}
```

## 7.2 Appendix B

### B.1. Traces Causeway meta-channel


```
    [
     {
        "class":[
           "org.ref_send.log.Got",
           "org.ref_send.log.Event"
        ],
        "anchor":{
           "number":1,
           "turn":{
              "loop":"server.js",
              "number":0
           }
        },
        "message":"AM7CXPB11402050838179",
        "trace":{
           "calls":[
              {
                 "name":"book_info",
                 "source":"server"
              }
           ]
        }
     },
     {
        "class":[
           "org.ref_send.log.Sent",
           "org.ref_send.log.Event"
        ],
        "anchor":{
           "number":0,
           "turn":{
              "loop":"server.js",
              "number":0
           }
        },
        "message":"MV5YURS11402050838537",
        "trace":{
           "calls":[
              {
                 "name":"book_info_response",
                 "source":"server"
              }
           ]
        }
     }
]
```