Drizzt: Dynamic Analysis of Distributed JavaScript Applications

Laurent Christophe, Coen De Roover, Wolfgang De Meuter
Software Languages Lab, Vrije Universiteit Brussel, Brussels, Belgium
Email: lachrist—cderoove—wdmeuter@vub.ac.be

I. Introduction

JavaScript has become ubiquitous on servers and client tiers of contemporary web applications. Accordingly, the research community has shown an increasing interest in helping web developers to understand and maintain JavaScript programs. The study of a program's execution, known as dynamic program analysis, has become a common technique in this respect [2].

Since the first apparition of JavaScript on the client side for form validations, the communication between a web application's tiers continuously increased to meet industrial standard in responsiveness. Yet, web developers still have little choices but to analyze each application tiers separately. The result of every separated analysis must then be recomposed to draw a complete picture of the system. Techniques have been proposed to automate this process for so called "lightweight dynamic analyses" such as tracing and profiling.

However it is not yet the case for so called "heavyweight dynamic analyses" such as taint analysis [4]. Taint analysis is a form of information flow analysis which aims at detecting flows of data that violate program integrity by marking or tainting program values. Today, web developers can taint values on each tier separately, but there is no approach to preserve the taint during cross tiers communication. Losing a value's taint every time it crosses tier boundaries may no longer be acceptable in today's application landscape.

In this position paper we introduce Drizzt an instrumentation platform enabling a wide variety of dynamic analyses for distributed JavaScript applications. Analyzing a distributed system requires an analysis which is itself distributed. However, asking the users of our approach to write distributed code bears a heavy burden on them as it adds a lot of accidental complexity. To factor the burden of orchestrating different parts of the analysis into the approach, we made the key design decision to provide a non-distributed interface. In Drizzt, to analyze distributed JavaScript programs, users write non-distributed code which can be either synchronous or asynchronous. The synchronous interface of Drizzt is actually identical to the interface of Aran which is a JavaScript instrumenter for non-distributed JavaScript programs [1]. In other words analyses developed for Aran can be directly plugged into Drizzt and be deployed in an distributed context "for free".

II. Design decision of the approach

We now present the main design decisions of our approach which are motivated by the three important criteria below:

- *Reliability*: We want the analyses built on top of our platform to produce reliable results.
- *Applicability*: We want our approach to be applicable in a large range of situations.
- *Expressiveness*: Building new dynamic analyses should be as easy as possible.

A. Source Code Instrumentation

In general, to perform dynamic analysis one is left with little choice but to either modify the execution environment or the program under analysis. On the one hand, instrumenting runtimes is a powerful approach because it enables analyses to break barriers imposed by the language specification. On the other hand, instrumenting source code links the approach to language specifications which are much more stable than runtimes. When targeting JavaScript, one must be aware that there exists many fast evolving engines which are all deployed on multiple operating systems. To maximize the applicability of our approach we should support all of these combinations and not force our users to use one particular version of an engine for one particular operating system. Opting for runtime instrumentation in this landscape would poses a severe maintenance challenge. Therefore, for the rest of this paper we focus on source code instrumentation which only requires maintenance against the ECMAScript specification.

B. Centralized Remote Advice

Before moving forward, we introduce two important concepts borrowed from the aspect-oriented terminology: "advice" and "poincut" [3]. We call *advice* the code written by the user of our approach whose execution is interleaved with the instrumented code. In our approach, the advice is composed of functions called *traps* which are called by the instrumented code. We call *poincut* the specification that dictates how the advice must be interleaved with the code of the application under analysis.

Analyses for distributed programs can directly be built on top of instrumenters for non-distributed programs. Indeed, as depicted in Figure 1 (left), such instrumenters can be used to implement an analysis for distributed programs which is itself distributed. In this setting, each *tier process* would run instrumented code and a local advice written by the user of our

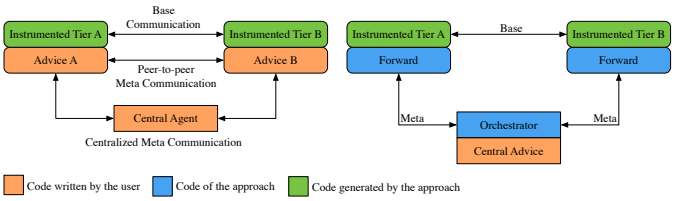

Fig. 1. Two distribution models: distributed local advices (left) and centralized remote advice (right).

approach. To draw a complete picture of the distributed system theses local advices would have to communicate either peer-to-peer or in a centralized manner. However, this contradicts the expressiveness criteria because orchestrating these distributed local advices bears a heavy load for the user of our approach as it adds a lot of accidental complexity. In short, this distribution model provides an interface which is too low-level to easily build dynamic analyses for distributed programs.

As it is often the case when designing frameworks, the challenge lies in balancing expressiveness and applicability. We want to factor as much complexity as possible in our approach yet it should be generic enough to implement many different dynamic analyses. A radical way of lifting the burden of analysis orchestration into the approach would be to require the user to write code that would only be executed on a single process. This is what depicts Figure 1 (right): the tier processes still execute instrumented code but, the local advices have been replaced by code forwarding everything to a *central process*. This central process runs an *orchestrator* who is responsible for providing a high-level interface to the centralized remote advice. To validate the applicability of this model we should make sure that the interesting dynamic analyses that motivated our work can still be implemented in it.

C. Synchronous Communication for Tier Processes

We now focus on how tier processes should communicate with the central process. We show in this section through an example that if this communication is asynchronous it necessarily introduces so called "heisenbugs". The word "Heisenbug" is a pun made from Werner Heisenberg's name who discovered that observing a quantum system necessarily alters its state. In the context of dynamic analysis, a heisenbug arises when the analysis alters the original program in an undesired and/or unintended way. So although asynchronous communication is encouraged by event-driven languages such as JavaScript, we opted for synchronous communication to preserve the reliability of our approach.

For the sake of the argument the example is based on ARAN-style instrumentation but the discussion holds for any style of instrumentations. We first start in a non-distributed context with the advice of Listing 1 being applied on the target program of Listing 2. The advice logs every program branching while the target program sets a timeout for an exception to be thrown, performs an if test and finally clears the timeout before it had the chance to be triggered. Applying the advice on the target program with the pointcut `["test"]` would involve executing the instrumented code of Listing 3. We note that the analyzed program behaves as the original program at the exception of the analysis log which is the only deviation we should tolerate.

```
1 var advice = {};
2 advice.test = function (x, i) {
3   console.log("TEST", x, i);
4   return x;
5 };
```

Listing 1. Advice for logging branching events.

```
1 function boum () { throw "BOUM" }
2 var id = setTimeout(boum, 0);
3 function clear () {
4   clearTimeout(id);
5   console.log("cleared");
6 }
7 if ("foo")
8   console.log("bar");
9 clear();
```

Listing 2. `boum.js`

```
bar
cleared
```

```
1 function boum () { throw "BOUM" }
2 var id = setTimeout(boum, 0);
3 function clear () {
4   clearTimeout(id);
5   console.log("cleared");
6 }
7 if ($traps.test("foo", 19))
8   console.log("bar");
9 clear();
```

Listing 3. `boum.js` instrumented with the pointcut `["test"]`.

```
TEST foo 19
bar
cleared
```

Transposing this situation in the centralized remote advice model would involve executing code similar to Listing 1 in an other process and possibly on a remote host. The traps invoked by the instrumented code would then have to perform some sort of remote communication with this central process. A priori, event-driven languages such as JavaScript would invite for asynchronous communication. This can be achieved by performing a *continuation passing style* (CPS) transformation to the program under analysis. In JavaScript specifically, the only other alternative would be to use the keyword `await` but it would have the same outcomes as Listing 4. For ARAN-style instrumentation, CPS transformation would involve passing an additional callback parameter which would represent their *continuation* i.e., the instructions to be executed after it returns. However, such transformation may lead to instructions interleaving that would never happen while executing the original program. Indeed, as depicted in Listing 4 and Listing 5, regardless which instructions we decide to embed in the continuation we cannot avoid heisenbugs.

```
function boum () { throw "BOUM" }
var id = setTimeout(boum, 0);
function clear () {
  clearTimeout(id);
  console.log("cleared");
}
$traps.test("foo", 19, function (res) {
  if (res)
    console.log("bar");
  clear();
});
```

Listing 4. `boum.js` CPS instrumented with the pointcut `["test"]` (`clear()` inside the continuation)

```
BOUM
```

```
function boum () { throw "BOUM" }
var id = setTimeout(boum, 0);
function clear () {
  clearTimeout(id);
  console.log("cleared");
}
$traps.test("foo", 19, function (res) {
  if (res)
    console.log("bar");
});
clear();
```

Listing 5. `boum.js` CPS instrumented with the pointcut `["test"]` (`clear()` outside the continuation)

```
cleared
"bar"
```

In Listing 4, the timeout having zero delay was triggered before the traps completion which led to an exception being thrown. In Listing 5, the function `clear` was invoked before returning to the event loop but the logging order was inversed. In both cases the CPS transformation introduced a heisenbug. It is important to underline that using other instrumentation styles would still require some sort of CPS transformation and eventually would introduce the same heisenbugs.

A principle of importance amongst event-driven program is *run-to-completion*, it states that an event must be completely processed before processing the next one. In other words, no code can interrupt the current event processing nor can it resume it later. However this is precisely what CPS transformation simulates by wrapping subsequent instructions inside a callback. For our approach to be reliable, it is important that the analyses built on top of it preserve this principle from the point of view of the original program. This is why we opted for synchronous communication on tier processes even though it introduces waiting time and performance overhead.

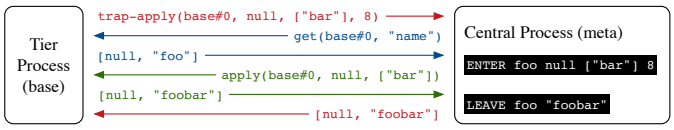

Fig. 2. Communication resulting from applying the advice of the Listing 6 to the program of Listing 7 with the pointcut `["apply"]`.

III. Presentation of the Prototype

We now briefly present our prototype named DRIZZT. First, we provide an example of communication between the central process and a tier process then we depict the architectural overview of our approach.

A. Communication Example

To illustrate the kind of communication carried out by our approach, we study how the advices of Listing 6 behave when applied to the program of Listing 7 with the pointcut `["apply"]`. This involves executing the instrumented code of Listing 8. The resulting communication between the central process and the tier process is depicted in Figure 2. First the `apply` trap sends a request to the central process indicating a function application. This initial request contains the context of the function application. Of interest is the parameter `base#0` which will be used to create a representation of `foo` within the central process. The first action of the remote advice is to retrieve the name of `f` by sending a request back to the tier process. Upon which the tier process replies with the array `[null,"foo"]`. The first element of that array is a potential error, its second element is the result of the operation should it succeeds. Upon receiving this response, the central process logs a message indicating the beginning of a function application. It then forwards the application to the actual function owned by the tier process. The tier process performs the function application and returns the array `[null,"foobar"]`. Upon receiving this response, the central process logs a message indicating the end of a function application. Finally the central process returns the same array as the final result of the `apply` trap.

```
var advice = {};
advice.apply = function (f, t, xs, i) {
  var n = f.name;
  console.log("ENTER", n, t, xs, i);
  var r = Reflect.apply(f, t, xs);
  console.log("LEAVE", r);
  return r;
};
```

Listing 6. Advice for logging function application

```
function foo (x) {
  return "foo" + x;
}
foo("bar");
```

Listing 7. `foobar.js`

```
function foo (x) {
  return "foo" + x;
}
$traps.apply(foo, null, ["bar"], 8);
```

Listing 8. `foobar.js` instrumented with the pointcut `["apply"]`.

B. Architectural Overview

When developing our prototype, considerable efforts were made to properly modularize it. As a result, DRIZZT is built on top of 5 modules which have been decoupled as much as possible. Some of these modules can be used to develop analyses for non-distributed applications while the others orchestrate the distributed part of the analysis. Each

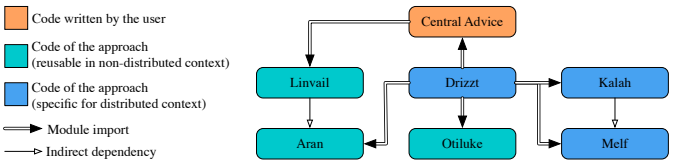

Fig. 3. The dependency links between the components of DRIZZT.

component of our approach is briefly presented below and Figure 3 depicts their dependency links.

- OTILUKE: [https://github.com/lachrist/otiluke]: Platform for developping and deploying JavaScript code instrumenters which are themselves written in JavaScript. OTILUKE provides a uniform interface for deploying instrumenters both on browsers and nodejs.
- ARAN: [https://github.com/lachrist/aran]: JavaScript code instrumenter for building dynamic analyses fully compatible with ECMAScript5. In ARAN, an analysis is implemented as a set of trap functions which are collectively called advice. Analyses written in ARAN can be deployed on modular applications with OTILUKE.
- LINVAIL [https://github.com/lachrist/linvail]: JavaScript shadow-executer built as an ARAN advice which enables tagging runtime values with analysis-related data. Linvail achieves primitives tracking by swapping them with special wrapper objects. The main challenge of this approach lies in preventing wrappers from escaping to the external world which would introduce heisenbugs.
- MELF: [https://github.com/lachrist/melf]: Communication library that supports both non-blocking synchronous requests and asynchronous requests between JavaScript processes.
- KALAH: [https://github.com/lachrist/kalah]: Synchronous implementation of far references based on MELF. Unlike other implementations of far references, KALAH returns actual values instead of promises. Although this makes them easier to use, it comes at the price of performance overhead as only MELF-related event can be processed while manipulating them.
- DRIZZT: [https://github.com/lachrist/drizzt]: Angular stone of our approach, this module imports every other components but LINVAIL which is only needed for value-centric analyses. After launching a DRIZZT orchestrator the application tiers must be launched within a DRIZZT instance to handle code instrumentation and information forwarding.

IV. CONCLUSION

We have detailed a novel approach to build dynamic analyses for distributed programs. To factor as much accidental complexity as possible into the approach we centralized code written by the analysis implementer into a single process. This main design decision imposes synchronous communication on the client side to preserve the "run-to-completion" principle and avoid modifying the behavior of the program under analysis.

The work we presented in this paper is still in progress. The next corner stone of our approach consists in making analyses aware of cross-tier communications. This would enable analyses to keep track of primitive values as they are passed between an application's processes. If tiers are themselves using far references, this value tracking could be extended to references as well. Only then, we will fully benefit from the centralized architecture of our approach.

REFERENCES

[1] Laurent Christophe, Elisa Gonzalez Boix, Wolfgang De Meuter, and Coen De Roover. Linvail: A general-purpose platform for shadow execution of javascript. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 260–270. IEEE, 2016.

[2] Bas Cornelissen, Andy Zaidman, Arie Van Deursen, Leon Moonen, and Rainer Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.

[3] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. *ECOOP'97Object-oriented programming*, pages 220–242, 1997.

[4] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. Taj: effective taint analysis of web applications. In *ACM Sigplan Notices*, volume 44, pages 87–97. ACM, 2009.