Liquid IT : Toward a better compromise between development scalability and performance scalability not definitive

Etienne Brodu

October 5, 2015

Abstract

TODO translate from below when ready

Résumé

Internet étend nos moyens de communications, et réduit leur latence ce qui permet de développer l'économie à l'échelle planétaire. Il permet à chacun de mettre un service à disposition de milliards d'utilisateurs, en seulement quelques heures. La plupart des grands services actuels ont commencé comme de simples applications créées dans un garage par une poignée de personnes. C'est cette facilité à l'entrée qui a permis jusqu'à maintenant une telle croissance sur le web. Google, Facebook ou Twitter en sont quelques exemples. Au cours du développement d'une application, il est important de suivre cette croissance, au risque de se faire rattraper par la concurrence. Le développement est guidé par les besoins en terme de fonctionnalités, afin de vérifier rapidement si le service peut satisfaire l'audience. On parle d'approche modulaire des fonctionnalités. Des langages tel que Ruby ou Java se sont imposés comme les langages du web, justement parce qu'ils suivent cette approche qui permet d'intégrer facilement de nouvelles fonctionnalités.

Si une application répond correctement aux besoins, elle atteindra de manière virale un nombre important d'utilisateurs. Son audience peut prendre plusieurs ordres de grandeurs en quelques jours seulement, ou même en quelques heures suivant comment elle est relayée. Une application est dite *scalable* si elle peut absorber ces augmentations d'audience. Or il est difficile pour une application suivant l'approche modulaire d'être *scalable*.

Au moment où l'audience commence à devenir trop importante, il est nécessaire de modifier l'approche de développement de l'application. Le plus souvent cela implique de la réécrire complètement en utilisant des infrastructures *scalables* qui imposent des modèles de programmation et des API spécifiques, qui représentent une charge de travail conséquente et incertaine. De plus, l'équipe de développement dois concilier cette nouvelle approche de développement *scalable*, avec la demande en fonctionnalités. Aucun langage n'a clairement réussi le compromis entre ces deux objectifs.

Pour ces raisons, ce changement est un risque pour la pérennité de l'application. D'autant plus que le cadre économique accorde peu de marges d'erreurs, comme c'est le cas dans la plupart des start-up, mais également dans de plus grandes structures.

Cette thèse est source de propositions pour écarter ce risque. Elle repose sur les deux observations suivantes. D'une part, Javascript est un langage

qui a gagné en popularité ces dernières années. Il est omniprésent sur les clients, et commence à s'imposer également sur les serveurs avec Node.js. Il a accumulé une communauté de développeurs importante, et constitue l'environnement d'exécution le plus largement déployé. De ce fait, il se place maintenant de plus en plus comme le langage principal du web, détrônant Ruby ou Java. D'autre part, l'exécution de Javascript s'assimile à un pipeline. La boucle événementielle de Javascript exécute une suite de fonctions dont l'exécution est indépendante, mais qui s'exécutent sur un seul cœur pour profiter d'une mémoire globale.

L'objectif de cette thèse est de maintenir une double représentation d'un code Javascript grâce à une équivalence entre l'approche modulaire, et l'approche pipeline d'un même programme. La première répondant aux besoins en fonctionnalités, et favorisant les bonnes pratiques de développement pour une meilleure maintenabilité. La seconde proposant une exécution plus efficace que la première en permettant de rendre certaines parties du code relocalisables en cours d'exécution.

Nous étudions la possibilité pour cette équivalence de transformer un code d'une approche vers l'autre. Grâce à cette transition, l'équipe de développement peut continuellement itérer le développement de l'application en suivant les deux approches à la fois, sans être cloisonné dans une, et coupé de l'autre.

Nous construisons un compilateur permettant d'identifier les fonctions de Javascript et de les isoler dans ce que nous appelons des Fluxions, contraction entre fonctions et flux. Un conteneur qui peut exécuter une fonction à la réception d'un message, et envoyer des messages pour continuer le flux vers d'autres fluxions. Les fluxions sont indépendantes, elles peuvent être déplacées d'une machine à l'autre.

Nous montrons qu'il existe une correspondance entre le programme initial, purement fonctionnel, et le programme pivot fluxionnel afin de maintenir deux versions équivalentes du code source. En ajoutant à un programme écrit en Javascript son expression en Fluxions, l'équipe de développent peut le rendre *scalable* sans effort, tout en étant capable de répondre à la demande en fonctionnalités.

Ce travail s'est fait dans le cadre d'une thèse CIFRE dans la société Worldline. L'objectif pour Worldline est de se maintenir à la pointe dans le domaine du développement et de l'hébergement logiciel à travers une activité de recherche. L'objectif pour l'équipe Dice est de conduire une activité de recherche en partenariat avec un acteur industriel.

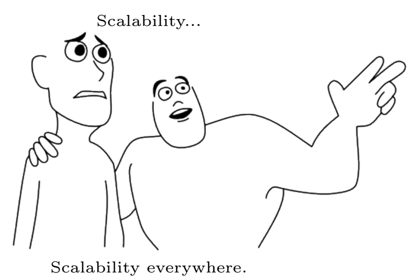
Contents

Chapter 1

Introduction

A few words about my PhD, and its context. How I approached it ? What were the motivations for me ? Worldline ? DICE ?

During this thesis, We studied scalability in the domain of computer sciences. Scalability of the performance of an application over resources, as is often suggested when refering to scalability. But as well as scalability of the development project of such application. It now seems to me as if scalability is a crucial component of interacting and evolving in the world. From space exploration, to economical market ...



What I take for scalability, might be overlapping with marginal increase, or incremental development

1.1 Web development

The growth of web platforms is partially due to Internet's capacity to allow very quick releases of a minimal viable product (MVP). In a matter of hours, it is possible to release a prototype and start gathering a user community around. *"Release early, release often"*, and *"Fail fast"* are the punchlines of the web entrepreneurial community. It is crucial for the prosperity of such project to quickly validate that the proposed solution meets the needs of its users. Indeed, the lack of market need is the number one reason for startup failure.[1] That is why the development team quickly concretises an MVP and

[1] https://www.cbinsights.com/blog/startup-failure-post-mortem/

iterates on it using a feature-driven, monolithic approach. Such as proposed by imperative languages like Java or Ruby.

1.2 Performance requirements

If the service successfully complies with users requirements, its community might grow with its popularity. If the service can quickly respond to this growth, it is scalable. However, it is difficult to develop scalable applications with the feature-driven approach mentioned above. Eventually this growth requires to discard the initial monolithic approach to adopt a more efficient processing model instead. Many of the most efficient models distribute the system on a cluster of commodity machines.

Once split, the service parts are connected by an asynchronous messaging system. Many tools have been developed to express and manage these service parts and their communications. However, these tools impose specific interfaces and languages, different from the initial monolithic approach. It requires the development team either to be trained or to hire experts, and to start over the initial code base. This shift causes the development team to spend development resources in background without adding visible value for the users. It is a risk for the evolution of the project as the number two and three reasons for startup failures are running out of cash, and missing the right competences.

1.3 Problematic and proposal

A web application fails to develop incrementally, and it is a risk for its economical evolution. The main question I address in this thesis is how to reconcile the reactivity required in the early stage of development and the performance increasingly required with the growth of popularity, while avoiding the risks of shifting technology during the evolution ? In this thesis, I study Javascript, as it seems to be increasingly used in web development, both client- and server-side, to start projects. Moreover, Javascript has the particularity to be implemented on top of an event-loop. This design is highly efficient and easy to develop with. But it remains limited to one core of a processor. Therefore, the problem of switching to a distributed approach remains.

To give a first answer to this question, I limited this question to a problematic of compilation from Javascript to a distributed approach. More specifically, to a problem of memory analysis. In this thesis, I make two main contributions. The first contribution is a language and its execution engine to support the distributed approach. The second contribution is an equivalence between Javascript, or any similar language, and the language of the first contribution.

1.4 Thesis organization

This thesis is organized in four main chapters. Chapter 2 introduce the context and the objectives I set for this work.

Chapter 3 presents the bibliography.

Chapter 4 presents the first contribution.

Chapter 5 presents the second contribution.

Chapter 2

Context and objectives

Contents

2.1 Introduction

This chapter presents the general context for this work, and leads to a definition of the problematic addressed in this thesis. Section 2.2 presents the context for the web development, and the motivation that led the web to become a software platform. It presents briefly the main languages available for initiating a web application, and a great section is dedicated to Javascript, as it is increasingly gained popularity these past few years. Section 2.3 presents the problematic for developping web server for large audiences. It explains why the languages presented in the previous section often fail to grow with the project they initially supported very efficiently. It conclude that this rupture is an economical risk for young projects. Finally, section 2.4 presents the goal of this work. That is to reconcile the technologies used in the initial phase of a project, with the ones used to grow the project in performance.

2.2 The Web as a platform

2.2.1 From operating systems to the web

With the invention of electronic computing machine, appeared the market for software applications. This market is not limited by marginal production cost ; software being a virtual product, the production and distribution cost for another unit is virtually null. The market is limited by the platform a software can be deployed on. The bigger the platform, the wider the market. There is an economically incentive to standardize and widen the platform, both for the provider, and for the consumer. The first platforms started as products, in competition with other products. Their manufacturers had economical incentive to increase their market share. Microsoft successfully took over the market of operating system in the 90s, and was on the edge of monopoly more than once. But eventually, the product is standardized, and becomes the platform.

Before the internet, this market was limited for distribution by the physical medium. It takes time to burn a CD, or a floppy, and to bring it to the consummer's home. Sir Tim Berners Lee invented the world wide web in 1989. It was initially intended to share scientific documents and results. And it eventually became the distribution medium

of choice for every virtual products, software included. It pushed the scalability of software distribution.

Similarly to operating systems, Web browsers started as software products. They exposed innovative features to try to increase their market share. Among others is the ability to run scripts. It allows to deploy and run software at unprecedented scales. The web became the platform. Now, with web services, or Software as a Service (SaaS), the distribution medium of software is so transparent that owning a software product to have an easier access is no longer relevant. We explore now the different languages to write and deploy applications on the web.

2.2.2 The languages of the web

In the early 90's, during the web early development, most of the now popular programming languages were released. Python(1991), Ruby(1993), Java(1994), PHP(1995) and Javascript(1995). With Moore's law predicting exponential increase in hardware performance, the industry realized that development time is more expensive than hardware. Low-level languages were replaced by higher-level language, trading performance for accessibility. The economical gain in development time compensated the worsen performances of these languages.

Java, developed by Sun Microsystems, imposes itself early as a language of choice and never really decreased. The language is executed on a virtual machine, allowing to write an application once, and to deploy it on heterogeneous machines. The software industry quickly adopted it as its main development language. It is currently the second most cited language on StackOverflow, and used on Github. And is in the first place of many language popularity indexes. However, the software industry wants stable and safe solutions. This prudence generally slows down Java evolution. The language struggled to keep up with the latest trends in software development.

Python is the second best language for everything. It is a general purpose language, currently popular for data science. In 2003, the release of the Django web framworks brought the language to the web development scene.

Ruby was confined in Japan and almost unknown to the world until the release of Rails in 2005. With the release of this web framework, Ruby took-off and is still in active use. It meets the latest trends in software development. And it might had replaced Java if the latter had not been so well adopted in the software industry.

PHP stands for Personal Home Page Tools. It was initially designed to build personal web pages. It might be one of the easiest language to start web development. However, according to several language popularity indexes, it is on a slow decline since a few years. It is generally unfit to grow projects to industrial size.

Since a few years, Javascript is slowly becoming the main language for web development. It is the only choice in the browser. Because of this unavoidable position, it became fast (V8, ASM.js) and usable (ES6, ES7). And since 2009, it is present on the server as well with Node.js This omnipresence became an advantage. It allows to develop and maintain the whole application with the same language. I argue in this thesis, that Javascript is the language of choice to bring a prototype to industrial standards.

2.2.3 Explosion of Javascript popularity

2.2.3.1 In the beginning

Javascript was created by Brendan Eich at Netscape around May 1995, and released to the public in September. At the time, Java was quickly adopted as default language for web servers development, and everybody was betting on pushing Java to the client as well. The history proved them wrong.

When Javascript was released in 1995, the world wide web was on the rise.[1] Browsers were emerging, and started a battle to show off the best features and user experience to attract the wider public.[2] Javascript was released to be one of these features on Netscape navigator. Microsoft released their browser Internet Explorer 3 in June 1996 with a concurrent implementation of Javascript. At the time, because of the differences between the two implementations, web pages had to be designed for a specific browser. This competition was fragmenting the web.

Netscape submitted Javascript to Ecma International for standardization in November 1996 to stop this fragmentation. In June 1997, ECMA International released ECMA-262, the first specification of ECMAScript, the standard for Javascript. A standard to which all browser should refer for their implementations.

The initial release of Javascript was designed in a rush. The version released in 1995 was finished within 10 days. And, it was intended to be

[1] `http://www.internetlivestats.com/internet-users/`

[2] to get an idea of the web in 1997 : `http://1x-upon.com/`

simple enough to attract unexperienced developers. For these reasons, the language was considered poorly designed and unattractive by the developer community.

But things evolved drastically since. The success of Javascript is due to many factors ; maybe the most important of all is the *View Source* menu that reveals the complete source code of any web application. *The view source menu is the ultimate form of open source*[3]. It is the vector of the quick dissemination of source code to the community, which picks, emphasizes and reproduces the best techniques. It brought open source and collaborative development to the web. Moreover, all web browsers include a Javascript interpreter, making Javascript the most ubiquitous runtime in history [18].

When such a language is distributed freely with the tools to reproduce and experiment on every piece of code. And its distribution is carried during the expansion of the largest communication network in history. Then an entire generation seizes this opportunity to incrementally build and share the best tools they can. This collaboration is the reason for the popularity of Javascript on the Web.

2.2.3.2 Rising of the unpopular language

Why does Javascript suck?[4]
Is Javascript here to stay?[5]
Why Javascript Is Doomed.[6]
Why JavaScript Makes Bad Developers.[7]
JavaScript: The World's Most Misunderstood Programming Language[8]
Why Javascript Still Sucks[9]
10 things we hate about JavaScript[10]
Why do so many people seem to hate Javascript?[11]

[3] http://blog.codinghorror.com/the-power-of-view-source/
[4] http://whydoesitsuck.com/why-does-javascript-suck/
[5] http://www.javaworld.com/article/2077224/learn-java/is-javascript-here-to-stay-.html
[6] http://simpleprogrammer.com/2013/05/06/why-javascript-is-doomed/
[7] https://thorprojects.com/blog/Lists/Posts/Post.aspx?ID=1646
[8] http://www.crockford.com/javascript/javascript.html
[9] http://www.boronine.com/2012/12/14/Why-JavaScript-Still-Sucks/
[10] http://www.infoworld.com/article/2606605/javascript/146732-10-things-we-hate-about-JavaScript.html
[11] https://www.quora.com/Why-do-so-many-people-seem-to-hate-JavaScript

Javascript started as a programming language to implement short interactions on web pages. The best usage example was to validate some forms on the client before sending the request to the server. This situation hugely improved since the beginning of the language. Nowadays, there is a lot of web-based application replacing desktop applications, like mail client, word processor, music player, graphics editor...

ECMA International released several version in the few years following the creation of Javascript. The first and second version, released in 1997 and 1998, brought minor revisions to the initial draft. The third version, released in the late 1999, contributed to give Javascript a more complete and solid base as a programming language. From this point on, the consideration for Javascript kept improving.

In 2005, James Jesse Garrett released *Ajax: A New Approach to Web Applications*, a white paper coining the term Ajax [20]. This paper points the trend in using this technique, and explain the consequences on user experience. Ajax stands for Asynchronous Javascript And XML. It consists of using Javascript to dynamically request and refresh the content inside a web page. It has the advantage to avoid requesting a full page from the server. Javascript is not anymore confined to the realm of small user interactions on a terminal. It can be proactive and responsible for a bigger part in the whole system spanning from the server to the client. Indeed, this ability to react instantly to the user gave to developer the feature to develop richer applications inside the browser. At the time, the first web applications to use Ajax were Gmail, and Google maps[12].

The third version of ECMAScript had been released, and it was homogeneously supported in the browsers. However, the DOM, and the `XMLHttpRequest` method, two components on which AJAX relies, still present heterogeneous interfaces among browsers. Around this time, the Javascript community started to emerge. Javascript framework were released with the goal to straighten the differences between browsers implementations. Prototype[13] and DOJO[14] are early famous examples, and later jQuery[15] and underscore[16]. These frameworks are responsible in great part to the large success

[12] A more in-depth analysis of the history of Ajax, given by late Aaron Swartz `http://www.aaronsw.com/weblog/ajaxhistory`

[13] `http://prototypejs.org/`

[14] `https://dojotoolkit.org/`

[15] `https://jquery.com/`

[16] `http://underscorejs.org/`

of Javascript and of the web technologies.

In the meantime, in 2004, the Web Hypertext Application Technology Working Group[17] was formed to work on the fifth version of the HTML standard. This new version provide new capabilities to web browsers, and a better integration with the native environment. It features geolocation, file API, web storage, canvas drawing element, audio and video capabilities, drag and drop, browser history manipulation, and many mores. It gave Javascript the missing interfaces to become the environment required to develop rich application in the browser. The first public draft of HTML 5 was released in 2008, and the fifth version of ECMAScript was released in 2009. These two releases, ECMAScript 5 and HTML5, represent a mile-stone in the development of web-based applications. Javascript became the programming language of this rising application platform.

In 2008* Google released V8 for its browser Chrome. It is a Javascript interpreter improving drastically the execution performance with a just-in-time compiler. This increase in performance allowed to push Javascript to the server, with the release of Node.js in 2009. Javascript was initially proposed to develop user interfaces. It is implemented around an event-based paradigm to react to concurrent user interactions. Because of the performance increase, this event-based paradigm proved to be also very efficient to react to concurrent requests of a web server. I will present this event-based paradigm in the next section.

⚠ TODO verify date

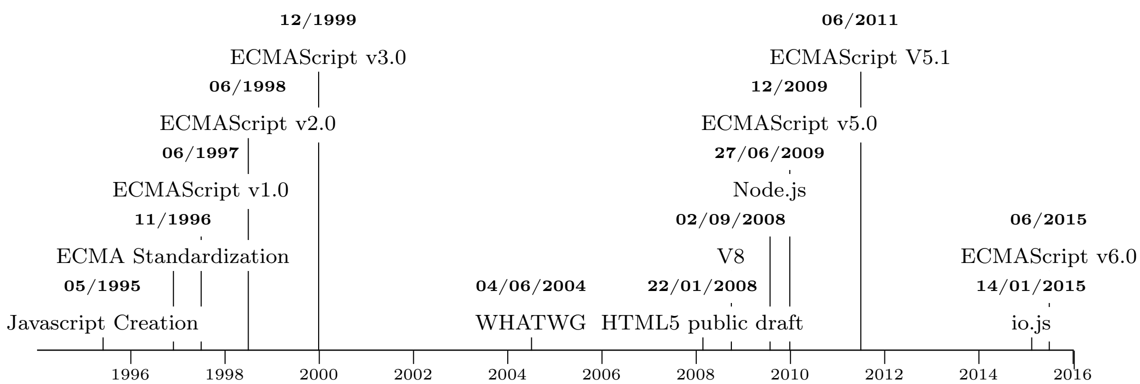
2.2.3.3 Current situation

"When JavaScript was first introduced, I dismissed it as being not worth my attention. Much later, I took another look at it and discovered that hidden in the browser was an excellent programming language."

—Douglas Crockford

The rise of Javascript is obvious on the web and particularly the open source communities. It also seems to be rising in the software industry. It is difficult to give an accurate representation of the situation because the software industry is opaque for economical reason. In the following paragraphs, I report some indexes that represent the situation globally, both in the open source community and in the more opaque software industry.

[17] `https://whatwg.org/`


12/1999
ECMAScript v3.0
06/2011
ECMAScript V5.1
06/1998
ECMAScript v2.0
12/2009
ECMAScript v5.0
06/1997
ECMAScript v1.0
27/06/2009
Node.js
11/1996
ECMA Standardization
02/09/2008
V8
06/2015
ECMAScript v6.0
05/1995
Javascript Creation
04/06/2004
WHATWG
22/01/2008
HTML5 public draft
14/01/2015
io.js
1996
1998
2000
2002
2004
2006
2008
2010
2012
2014
2016

Available resources The TIOBE Programming Community index is a monthly indicator of the popularity of programming languages. Javascript ranks 6th on this index, as of April 2015, and it was the most rising language in 2014. It uses the number of results on many search engines about a certain language. The results contains the resources and traces of the activity around the language that are used as a measure of the popularity of a programming language. However, the number of pages doesn't represent the number of readers. The measure used by the TIOBE is controversial, and might not be representative.

Alternatively, the PYPL index is based on Google trends to measure the number of requests on a programming language. Javascript ranks 7th on this index, as of May 2015. This index seems to be more accurate, as it depicts the actual interest of the community for a language. However, it is not representative as it only takes Google search into account.

From these indexes, the major programming languages are Java, C/C++ and C#. The three languages are still the most widely taught, and used to write softwares.

Developers collaboration platforms An indicator of the popularity and usage of a language is the number of developers and projects using it.

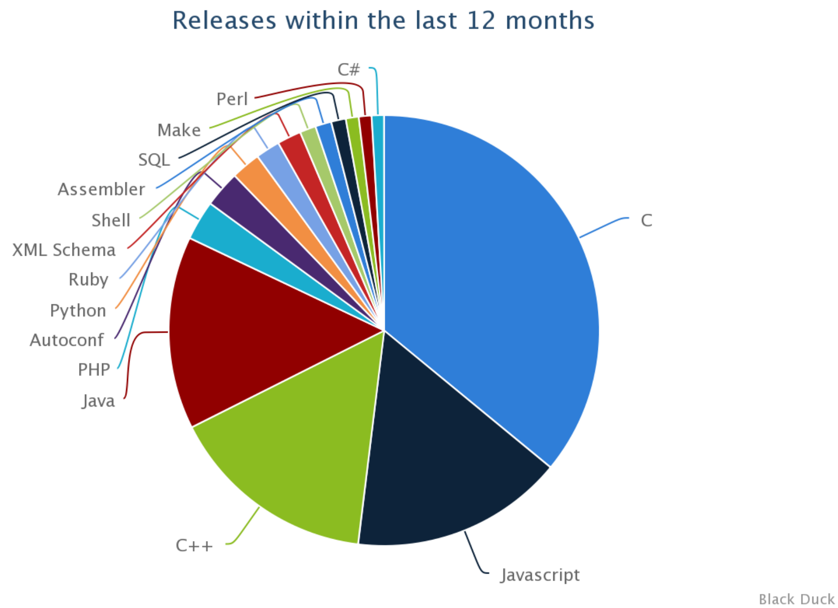
Github is the most important collaborative development platform, with around 9 millions users. Javascript is the most used language on github since mid-2011, with more than 320 000 repositories. The second language is Java with more than 220 000 repositories.

*

TODO : graph of Github repositories by languages

StackOverflow is the most important Q&A platform for developers. It is a good representation of the activity around a language. Javascript is the language the most cited on StackOverflow, with more than 890 000 questions. The second is Java with around 880 000 questions.

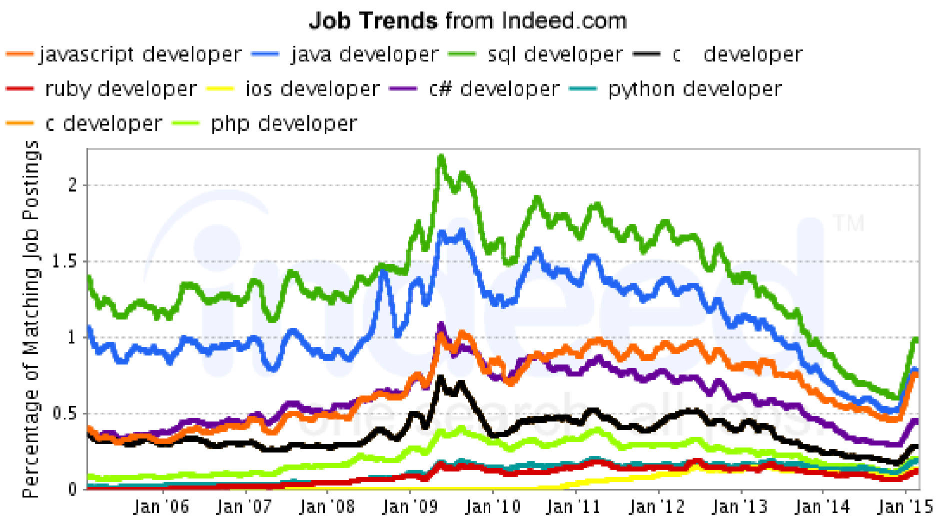
Black Duck Software helps companies streamline, safeguard, and manage their use of open source. For its activity, it analyzes 1 million repositories over various forges, and collaborative platforms to produce an index of the usage of programming language in open source communities. Javascript ranks second. C is first, C++ third and Java fourth. These four languages represent about 80% of all programming language usage in open source communities.



*

TODO redo this graph, it is ugly.

Jobs The software industry is rather closed sourced, and its activity is rather opaque. All these previous metrics are representing the visible activity about programming language, but are not representative of the software industry. The trends on job openings gives an hint of the direction the software industry is heading towards. The job searching platform *Indeed* provides some trends over its database of job propositions. Javascript developers ranked at the third position, right after SQL and Java developers. Over the 5 last years, the number of job position for Javascript developers increased so as to almost close the gap with Java. This position indicate that Javascript is increasingly adopted in the software industry.



*

⚠ TODO redo this graph, it is ugly.

All these metrics represent different faces of the current situation of the Javascript adoption in the developer community. With the evolution of web applications development and increased interest in this domain, Javascript is assuredly one of most important language of this decade. It is widely used in open source projects, and everywhere on the web, as well as in the software industry.

2.3 Highly concurrent web servers

*

⚠ This section needs review

2.3.1 Concurrency

The Internet allows interconnection at an unprecedented scale. There is currently more than 16 billions devices connected to the internet, and it is growing exponentially[18] [**Hilbert2011a**]. This massively interconnected network gives the ability for a web applications to be reached at the largest scale. A large web application like google search receives about 40 000 requests per seconds. That is about 3.5 billions requests a day[19]. Such a web application needs to be highly concurrent to manage a large number of simultaneous request. Concurrency is the ability for an application to make

[18] http://blogs.cisco.com/news/cisco-connections-counter

[19] http://www.internetlivestats.com/google-search-statistics/

progress on several tasks at the same time. For example to respond to several simultaneous requests, a task is a part in the response to a request. *

TOOD define more clearly what is a task

In the 2000s, the limit to reach was to process 10 thousands simultaneous connections with a single commodity machine[20]. Nowadays, in the 2010s, the limit is set at 10 millions simultaneous connections at roughly the same price[21]. With the growing number of connected devices on the internet, concurrency is a very important property in the design of web applications.

2.3.1.1 Scalability

The traffic of a popular web application such as Google search is huge, and it remains roughly stable because of this popularity. There is no apparent spikes in the traffic, because of the importance of the average traffic. However, the traffic of a less popular web application is much more uncertain. If the web application fits the market need, it might become viral when it is efficiently relayed in the media. For example, when a web application appears in the evening news, it expects a huge spike in traffic. With the growth of audience, the number of simultaneous requests obviously increases, and the load of the web application on the available resources increases as well. The available resources needs to increase to meet the load. This growth might be steady and predictable enough to plan the increase of resources ahead of time, or it might be erratic and challenging. The spikes of a less popular web application are unpredictable. Therefore, the concurrency needs to be expressed in a scalable fashion. An application is scalable, if the growth of its audience is proportional to the increase of its load on the resources. For example, if a scalable application uses one resource to handle n simultaneous requests, it will use k resource to handle two times n simultaneous requests. With k being constant, for n ranging from tens to millions of simultaneous requests. Scalability assures that the resource usage is not increasing exponentially in function of the audience increase ; it increases roughly linearly.

2.3.1.2 Time-slicing and parallelism

Concurrency can be achieved on hardware with either a single or several processing units. On a single processing unit, the tasks are executed sequentially ; their executions are interleaved in time. On several processing unit,

[20] http://www.kegel.com/c10k.html

[21] http://c10m.robertgraham.com/p/manifesto.html

the tasks are executed in parallel. Parallel executions reduce computing time over sequential execution, as it uses more processing units.

If the tasks are completely independent, they can be executed in parallel as well as sequentially. This parallelism is a form of scalable concurrency, as it allows to stretch the computation on available hardware to meet the required performance, at the required cost. This parallelism is used in operating system to execute several applications concurrently to allow multi-tasking.

However, the tasks of an application are rarely independent. The tasks need to coordinate their dependencies to modify the global state of the application. This coordination limits the possible parallelism between the tasks, and might impose to execute them sequentially. The type of possible concurrency, sequential or parallel, is defined by the required coordination between the tasks. Either the tasks are independents and they can be executed in parallel, or the tasks need to coordinate a common state, and they need to be executed sequentially to avoid conflicting accesses to the state.

2.3.2 Interdependencies

It is easier to understand the possible parallelism of a cooking recipe than an application. That is because the modifications to the state are trivial in the cooking recipe, hence the interdependencies between operations. It is easy to understand that preheating the oven is independent from whipping up egg whites. While the interdependencies are not immediately obvious in an application. *

⚠ TODO is this metaphor useful here ? if yes, continue to a transition

2.3.2.1 State coordination

The interdependencies between the tasks impose the coordination of the global application state. This coordination happen either by sending events from one task to another, or by modifying a shared memory.

If the tasks are independent enough, they never need access to a state at the same time. The coordination of the state of the application can be done with message passing. They pass the states from one task to another so as to always have an exclusive access on the state. As example, applications built around a pipeline architecture define independent tasks arranged to be executed one after the other. The tasks pass the result of their computation to the next. These tasks never share a state.

If the tasks need concurrent accesses to a state, they cannot efficiently pass the state from one to the other repeatedly. They need to share and to coordinate their accesses to this state. Each access needs to be exclusive to avoid corruption. This exclusivity is assured differently depending on the scheduling strategy.

2.3.2.2 Task scheduling

There is roughly two main scheduling strategy to execute tasks sequentially on a single processing unit : preemptive scheduling and cooperative scheduling. The state coordination presented previously is highly depending on the scheduling strategy.

Preemptive scheduling is used in most execution environment in conjunction with multi-threading. The scheduler allows each task to execute for a limited time before preempting it to let another task execute. It is a fair and pessimistic scheduling, as it grant the same amount of computing time to each task. However, as the preemption might happen at any point in the execution, it is important for the developer to lock the shared state before access, so as to assure exclusivity. This protection is known to be hard to manage.

In cooperative scheduling, the scheduler allows a task to run until the task yield the execution back. Each task is an atomic execution ; it is never be preempted, and have an exclusive access on the memory. It gives back to the developer the control over the preemption. It seems to be the easiest way for developers to write concurrent programs efficiently. Indeed, I presented in the previous section the popularity of Javascript, which is often implemented on top of this scheduling strategy (DOM, Node.js).

As I explained the different paradigms for writing concurrent program in this subsection, it appears that the main problem is to assure to the developer for each task the exclusive access to the state of its application. This assurance is called invariance.

2.3.2.3 Invariance

I call invariance the assurance given that the state accessible from a task will remain unchanged during its access to avoid corruption, and more generally to allow the developer to perform atomic modifications on the state. This assurance allows the developer to regroup operations logically so as to perform all the operations without interference from concurrent executions. The same concept is found in transactional memory.

In a multi-process application, there is no risk of corrupted state by simultaneous, conflicting accesses. The invariance is made explicit by the developer as the memory needs to be isolated inside each process. The invariance is assured at any point in time because the process remains isolated.

In a cooperative scheduling application, the developer is aware of the points in the code where the scheduler switches from one concurrent execution to the other, so it can manage its state in atomic modification. The invariance is assured, because any region in the memory can be accessed only by one task at a time.

Between these two invariances, the locking mechanisms seems to be a promising compromise. The developer defines only the shared states, and these are locked only when needed. However, it increases the complexity of the possible locked combination, leading to unpredictable situations, such as deadlock, and so on. The locking mechanisms are known to be difficult to manage, and sub-optimal. Indeed, they are eventually as efficient as a queue to share resources.

For the rest of this thesis, I focus only on the invariances provided by the multi-process paradigm and the cooperative scheduling. * They are similar, because the developer defines sequence of instructions with atomic access to the memory. And in both paradigms, these sequences communicate by sending messages to each other. The difference is that in the multi-process paradigm, the developer defines the region and the isolated memory, while in the cooperative scheduling, the developer defines only the region, and the memory is isolated by the exclusivity in the execution.

⚠ In the state of the art, we probably cannot reduce the analyze to these two paradigms.

This difference seems to be crucial in the adoption of the technology by the developer community. As we will see in the next subsection, the parallelism of multi-process is difficult to develop, but provides good performances, while the sequentiality of the cooperative scheduling is easier to develop, but provides poor performances compared to parallelism.

*

⚠ TODO this paragraph needs review

2.3.3 Disrupted development

2.3.3.1 Scalable concurrency

Around 2004, the so-called Power Wall was reached. The clock of CPU is stuck at 3GHz because of the inability to dissipate the heat generated at higher frequencies. Additionally, the instruction-level parallelism is limited. Because of these limitations, a processor is limited in the number of instruction per second it can execute. Therefore, a coarser level of parallelism, like the task-level, multi-processes parallelism previously presented is the only option to achieve high concurrency and scalability. But as I presented previously, this parallelism requires the isolation of the memory of each independent task. This isolation is in contradiction with the best practices of software development, hence, is difficult to develop for common developers. It creates a rupture between performance and development accessibility.

2.3.3.2 The case for global memory

The best practices in software development advocate to design a software into isolated modules. This modularity allows to understand each module by itself, without an understanding of the whole application. The understanding of the whole application emerges from the interconnections between the different modules. A developer need only to understand a few modules to contribute to an application of hundreds or thousands of modules.

Modularity advocates three principles : encapsulation, a module contains the data, as well as the functions to manipulate this data ; separation of concerns, each module should have a clear scope of action, and this scope should not overlap with the scope of other modules ; and loose coupling, each module should require no, or as little knowledge as possible about the definition of other modules. The main goal followed by these principles, is to help the developer to develop and maintain a large code-base.

Modularity is intended to avoid a different problem than the isolation required by parallelism. The former intends to avoid unintelligible spaghetti code ; while the latter avoids conflicting memory accesses resulting in corrupted state. The two goals are overlapping in the design of the application. * Therefore, every language needs to provide a compromise between these two goals, and specialized in specific type of applications. I argue that the more accessible, hence popular programming languages choose to provide modularity over isolation. They provide a global memory at the sacrifice

⚠ TODO needs more explanations -> so it is hard for dev to do both ? Why exactly ?

of the performance provided by parallelism. On the other hand, the more efficient languages sacrifice the readability and maintainability, to provide a model closer to parallelism, to allow better performances. * *

TODO instead of language, use a more generic term to refer to language or infrastructure

2.3.3.3 Technological shift

Between the early development, and the maturation of a web application, the development needs are radically different. In its early development, a web application needs to quickly iterate over feedback from its users. *"Release early, release often"*, and *"Fail fast"* are the punchlines of the web entrepreneurial community. The development team quickly releases a Minimum Viable Product as to get these feedbacks. The development reactivity is crucial. The first reason of startup failures is the lack of market need[22]. Therefore, the development team opt for a popular, and accessible language.

TODO justification and examples. What are modular application, or parallel applications ?

As the application matures and its audience grows, the focus shift from the development speed to the scalability of the application. The development team shift from a modular language, to a language providing parallelism.

This shift brings two problems. First, the development team needs to take a risk to be able to grow the application. This risk usually implies for the development team to rewrite the code base to adapt it to a completely different paradigm, with imposed interfaces. It is hard for the development team to find the time, hence the money, or the competences to deploy this new paradigm. Indeed, the number two and three reasons for startup failures are running out of cash, and missing the right competences. Second, after this shift the development pace is different. Parallel languages are incompatible with the commonly learned design principles. The development team cannot react as quickly to user feedbacks as with the first paradigm.

This technological rupture proves that there is economically a need for a a more sustainable solution to follow the evolution of a web application. A paradigm that it is easy to develop with, as needed in the beginning of a web application development, and yet scalable, so as to be highly concurrent when the application matures.

[22] https://www.cbinsights.com/blog/startup-failure-post-mortem/

2.4 Equivalence

I argue that the language should propose to the developer an abstraction to encourage the best practices of software development. Then a compiler, or the execution engine, can adapt this abstraction so as to leverage parallel architectures. So as to provide to the developer a usable, yet efficient compromise. We propose to find an equivalence between the invariance proposed by the cooperative scheduling paradigm and the invariance proposed by the multi-processes paradigm in the case of web applications.

2.4.1 Architecture of web applications

2.4.1.1 Real-time streaming web services

*

⚠

The need for invariance in the streaming applications : it can be emulated by message passing. Indeed the data flows from one processing step to the other, with few retro-progation of state (don't mention retro-propagation yet)

This equivalence intends not to be universal. It focuses on a precise class of applications : web applications processing stream of requests from users in soft real-time.

Such applications are organized in sequences of concurrent tasks to modify the input stream of requests to produce the output stream of responses. This stream of data stand out from the pure state of the application. The data flows in a communication channel between different concurrent tasks, and is never stored on any task. The state represents a communication channel between different instant in time, it remains in the memory to impact the future behaviors of the application. The state might be shared by several tasks of the application, and result in the needs for coordination presented in the previous section. In this thesis I study two programming paradigm derived directly form the cooperative scheduling and the multi-process paradigms presented in previous sections to be applied in the case of real-time web applications. The event-loop execution engine is a direct application of the cooperative scheduling, and the pipeline architecture is a direct application of the multi-process paradigm.

2.4.1.2 Event-loop

The event-loop is an execution model using asynchronous communication and cooperative scheduling to allow efficent execution of concurrent tasks on a single processing unit. It relies on a queue of event, and a loop to process each event one after the other. The communications are asynchronous to let the application use the processor instead of waiting for a slow response. When the response of a communication is available, it queues an event. This event is composed of the result of the communication, and of a function previously defined at the communication initiation, to continue the execution with the result. In the Javascript even-loop, this function is defined following the continuation passing style, and is named a callback. After processing the result, this callback can initiate communications, resulting in the queuing of more events.

In this model, the data is the result of every communication operations - starting with the received user request - flowing through a sequence of callbacks, one after the other. The state contains all the variables remaining in memory from one request to the other, and from one callback to the other. In Javascript, it includes the closures.

*

⚠ TODO schema of an event-loop

2.4.1.3 Pipeline

The pipeline software architecture uses the multi-process paradigm and message passing to leverage the parallelism of a multi-core hardware architectures for streaming application. It consists of many processes treating and carrying the flow of data from stage to stage. This flow of data consist roughly of the requests, and associated data from the user, as well as the necessary state coordination between the stages. Each stage has its independent memory to hold its own state from one request to another.

*

*

⚠ TODO it is not universal, but multi-process paradigms are also oriented around event-loops. An Event-loop is a multi-process on one machine. A multi-process is multiple event-loop running different part of the same program.

The pipeline architecture and the event-loop model present similar execution model. Both paradigms encapsulate the execution, in callbacks or processes. Those containers are assured to have an exclusive access to the memory. However, they provide two different memory models to provide this exclusivity. It results in two distinct ways for the developer to assure the invariance, and to manage the global state of the application. The event-loop shares the memory globally through the application, allowing the best practice of software development. It is possibly the reason of the wide adoption of this programming model by the community of developers.

I argue in this thesis that it is possible to provide an equivalence between the two memory models for streaming web application. In the next subsection, I present the similarity in the execution model, and the differences in

⚠ TODO schema of a pipeline

the memory model for which an equivalence is necessary. Such equivalence would allow to transform an application following the event-loop model to be compatible with the pipeline architecture. This transformation would allow the development of an application following a programming model allowing the best practices of software development, while leverageing the parallelism of multi-core hardware architecture.

2.4.2 Equivalence

2.4.2.1 Rupture point

The execution of the pipeline architecture is well delimited in isolated stages. Each stage has its own thread of execution, and is independent of the others. On the other hand, the execution of the event-loop seems pretty linear to the developer. The continuation passing style nest callbacks linearly inside each others. The message passing linking the callbacks is transparently handled by the event-loop. However, the execution of the different callbacks are as distinct as the execution of the different stages of a pipeline. Precisely, the call stack is as distinct between two callbacks, as between two stages. Therefore, in the event-loop, an asynchronous function call represents the end of the call stack of the current callback, and the beginning of the call stack of the next. It represents what I call a rupture point. It is the equivalent to a data stream between two stages in the pipeline architecture.

Both the pipeline architecture and the event-loop present these ruptures points. To allow the transformation from the event-loop model to the pipeline architecture in the case of real-time web applications, I study in this thesis the possibility to transform the global memory of the event-loop into isolated memory to be able to execute the application on a pipeline architecture.

2.4.2.2 State coordination

The global memory used by the event-loop holds both the state and the data of the application. The invariance holds for the whole memory during the execution of each callback. As I explained in the previous section, this invariance is required to allow the concurrent execution of the different tasks. On the other hand, the invariance is explicit in the pipeline architecture, as all the stages have isolated memories. The coordination between these isolated process is made explicit by the developer through message passing.

I argue that the state coordination between the callbacks requireing a global memory could be replaced by the message passing coordination used manually in the pipeline architecture. I argue that not all applications need concurrent access on the state, and therefore, need a shared memory. Specifically, I argue that each state region remains roughly local to a stage during its modification. *

⚠ TODO review that, I don't know how to formulate these paragraphs. Identify the state and the data in the global memory.

2.4.2.3 Transformation

This equivalence should allow the transformation of an event loop into several parallel processes communicating by messages. In this thesis, I study the static transformation of a program, but the equivalence should also hold for a dynamic transformation. I present the analyzis tools I developed to identify the state and the data from the global memory.

With this compiler, it would be possible to express an application with a global memory, so as to follow the design principles of software development. And yet, the execution engine could adapt itself to any parallelism of the computing machine, from a single core, to a distributed cluster.

TODO too fast on the end of this section

TODO Transition to the chapter State of the Art

Chapter 3

Software Design

Contents

3.1 Introduction

Computer applications are economically constrained by the cost of both development, and exploitation. These constraints became even more important, and the costs increased for web applications because with the growth of the web. In this chapter, we draw a broad view of this duality in software systems projects, to finally refine the scope on our subject of interest, and to define the problematic of this thesis.

Since the early days of software development as a discipline, the best practices advocate to organize the code into independent units to decompose a problem into many subproblems. It was called modular programming, structured design [52], hierarchical structure [15] and object-oriented programming among other approaches. These approaches focus on improving the readability, the maintainability, the comprehensibility ... We say that they intend to assure the evolution of the software system.

These approaches assure the evolution of the implementation of software systems. The Moore's law [42] assured an exponential evolution of the processing power, hence the software industry could always rely on the hardware to increase the execution speed. Eventually, the clock speed of processors plateaued. The increasing number of transistors predicted by Moore's law needed to be reorganized as several execution units into the same processor.

The best practices of software development inherited two goals : to assure the evolution of implementation by decomposing it into subproblems, as well as to decompose the execution on the several execution units. As D. L. Parnas showed in 1972 [48], these two decompositions are incompatible. It seems impossible to develop a software following a decomposition that satisfies both the evolution, and an efficient parallel execution.

With the incentive to leverage the execution power of parallel architectures, intensive work was done to provide tools and model to organize the execution on multiple execution units. Though, these works often completely discard the evolution of implementation, and are often hard to use, and to maintain.

There has been many attempts at reconciling the two goals into a single approach. But none seems really convincing enough to be widely adopted. Throughout this chapter, I will classify different works from the community into three categories : focus on implementation evolution, focus on parallel execution, or reconciliation of the two.

3.2 Software Design

In order to improve and maintain a software system, one needs the mental representation behind its conception. Architects, and mechanical engineer draw codified plans to share their mental representations with other architects and building teams. Similarly software developers write source codes. But because the source code represents both the plan and its execution, the second aspect tends to shadow the first, and the mental representation is lost in technical the details and optimizations of the implementation. It then becomes hard or even impossible to quickly grasp without the associated mental representation. Newcomers, or even the initial author after several weeks, would have difficulties to understand the system. This problem becomes even more critical as the size of the system grows in size. Therefore, it is important to decompose the system into smaller subsystem easier to grasp individually. Such decomposition, improve the readability and comprehensibility hence maintainability of the implementation of a software system. In this section, we show the theoretical tools for this decomposition, and their application in programming languages.

3.2.1 Modularity

3.2.1.1 Structured Programming

The growing size and complexity of software systems eventually urges the developers to split the problem into isolated subproblems. To respond to this problem, Dijkstra developed the concept of Structured Programming [**Dijkstra1970**]. D. Knuth cited C. Hoare to define Structured Programming as *the systematic use of abstraction to control a mass of details, and also a means of documentation which aids program design* [**Knuth1974**]. Dijkstra formalized this procedure on two levels, at a fine grain and at a coarse grain [14, 15].

The `goto` statement makes the flow of control very hard to follow and understand. It is called spaghetti code. Dijkstra advocated instead to decompose the problem into subproblems encapsulated into structures and reusable functions [14]. It impacts the development at a fine grain.

He also proposed to design complex systems with a hierarchical structure [15]. It decomposes a bigger problem at a coarser grain into subproblems encapsulated into layers. Each layer would abstract a design problem for the upper layers. This work established grounds for what is know called modular programming.

3.2.1.2 Modular Programming

Modular programming advocates to design a software system as an assembly of modules communicating with each other. The goal of using modular programming is twofold. It allows a developer to limit its understanding only to the features isolated inside a module, instead of understanding the whole problem [52]. And it reduces development time by allowing several developers to implement simultaneously different modules [**Cataldo2006**, 58].

The criterion to decompose the system into modules are coupling and cohesion [52]. The coupling defines the strength of the interdependence between modules. It is opposed to cohesion which defines how strongly the features inside a module are related. Low coupling between modules and high cohesion inside modules imply a better readability and comprehensibility, hence a better maintainability of the implementation of the system.

These two criterion defines how modular is the implementation. However, it doesn't define how well this organization will stand against the evolution of the implementation.

3.2.2 Design Choices

It is important that the modular organization stand against the evolutions in the specification of the problem, and their consequences in the implementation. The interfaces between modules, and the contents of these modules need to be well thought. The information hiding principle, and the separation of concerns are two similar approach to do so.

3.2.2.1 Information Hiding Principle

The information hiding principle helps define the content of modules so as to limit the impact of the evolution to a small portion of the implementation [48]. It advocates to encapsulate a specific design choice in each module to isolate the evolution on this choice from impacting the rest of the implementation. In this article [48], D. Parnas clearly opposes the organization of modules following the information hiding principle from the one following a pipeline approach to parallelize the execution. The former organization supports the development evolution, while the latter is more favorable to parallel execution and to performance. This opposition shows that a program cannot follow an organization that support both development evolution, and performance.

3.2.2.2 Separation of Concerns

The Separation of Concern is a design principle advocating that each module is responsible for one and only one specific concern [**Tarr1999**, **Hursch1995**]. For example, the separation of the form and the content in HTML / CSS, or the OSI model for the network stack, are example of separation of concerns.

However, this definition is orthogonal to the original meaning coined by Dijkstra [**Dijkstra1982**]. It is interesting to note this difference, as it is related directly to this thesis. The initial definition was about analyzing independently how a system meets different concerns. Dijkstra gives the example of analyzing independently correctness and efficiency. It is impossible to encapsulate correctness, or efficiency in a module, they concern the whole system. In this respect, this thesis is oriented towards separating the concern of development evolution and the concern of performance. That is to be able to reason on the maintainability of a program, independently than of its performance, and vice versa. This seems challenging as D. Parnas opposed these two concerns.

In this thesis, we investigate further this opposition to separate the concern of evolution and the concern of performance in the case of a web application. In the next subsection we investigate the first concern, we present the major programming models used to improve the evolution of an application.

3.2.3 Programming Models

Programming languages are designed for developers to follow the best practices mentioned above. We present two programming models : object oriented programming and functional programming.

3.2.3.1 Object Oriented Programming

Alan Kay, who coined the term, states that Object Oriented Programming (OOP) is about message-passing, encapsulation and late binding. (There is no scholar reference for that, only a public mail exchange[1].) This original definition is strongly related to modular programming. It helps encapsulate both the data, and the functions to process this data in an isolated, loosely coupled module. The very first OOP language was Smalltalk [**Goldberg1984**]. It defined the core concept of OOP, and is inspired by LISP and by the definition of the Actor Model, which we will define in the next section.

Object-Oriented Programming evolved to adopt as well the concepts of class, inheritance and polymorphism. The major languages of the software industry feature this Object-Oriented approach. We can cite C++ and Java as the emblematic figures of OOP.

Though, the field test seems to have had reason of this strict version of OOP. The trends in programming language seems to digress from the pure Object-Oriented approach to evolve toward an approach closer to Functional Programming. Indeed Javascript, Ruby and Python adopt functional features such as dynamic typing and higher-order functions.

3.2.3.2 Functional Programming

Functional programming is often associated to its purest form, manipulating only expressions - in place of operation statements - and forbidding state mutability. However, the essence of functional programming is not as strict, it resides in higher-order functions and lazy evaluation. Two features that major programming languages now commonly adopt.

Higher-Order Function Languages providing higher-order functions allows to manipulate functions like any other primary value : to store them in variables, or to pass them as arguments. Higher-order functions replace the needs for most modern object oriented programming design patterns [2]. Higher-order functions and lazy evaluation help loosen the couple between modules, and improve their re-usability. *In fine*, it helps developers to write applications that are more maintainable, and upgradeable [**Hughes1989**].

[1] `http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en`
[2] `http://stackoverflow.com/a/5797892/933670`

Closures Most languages use closures to implement lexical scope with higher-order functions [53]. A closure is the association of a function and the data context from its creation. It allows this function to access variable from this context, even when invoked outside their scope, for example when passed as an argument to another module.

It loosen the couple between modules, and helps define more generic and reusable modules. However, it increase their dependencies during the execution. Indeed, by exchanging closures, two modules intricately share their contexts of execution.

Functional programming greatly improves the resilience of implementation to the evolution of their specification. However, it requires a global memory to share the context of execution among modules. As we will see in the next section, sharing memory makes parallelism difficult. In this regard, the concern of evolution and the concern of performance seem incompatible.

3.3 Software Efficiency

Programming started with a very sequential nature, as Moore's law [42] was wrongly interpreted as an exponential evolution in the sequential performance of the processing unit. But it eventually evolved toward concurrency to make advantage of parallel architectures [**Flynn1972**].

The first models of computation, like the Turing machine and lambda-calculus, were sequential and based on a global memory state. A formalism was missing to represent concurrent computations. We present the most important works on formalisms for parallel computation. They first tackled the problems of determinacy, communication and state synchronization. The answer to this problems seems to be in a formalism based on a network of concurrent processes, asynchronously communicating via messages. We present the works on the programming models based on this formalism. Recently, with the need of performance from the web to process stream of requests, we see huge improvements in the field of distributed stream processing.

3.3.1 Concurrency Theory

The mathematical models are a ground for all following work on concurrent programming, we briefly explain them in the next paragraphs. There are two main formal models for concurrent computations. The Actor Model of C. Hewitt, the Pi-calculus of R. Milner. Based on these definitions, we explain the importance of determinism, and the reason that made asynchronous message-passing prevail.

3.3.1.1 Models

Actor Model The Actor model allows to express the computation as a set of communicating actors [**Clinger1981**, 27, 26]. In reaction to a received message, an actor can create actors, send messages, and choose how to respond to the next message. All actors are executed concurrently, and communicate asynchronously. Asynchronous communication means that the sender continues its execution immediately after sending the message, before receiving the result of the initiated communication.

The Actor model was presented as a highly parallel programming model, but intended for Artificial Intelligence purposes. Its success spread way out of this first scope, and it became a general reference and influence.

Π-calculus R. Milner presented a process calculus to describe concurrent computation : the Calculus of Communicating Systems (CCS) [38, 41]. It is an algebraic notation to express identified processes communicating through synchronous labeled channels. The π-calculus improved upon this earlier work to allow processes to be communicated as values, hence to become mobile [16, 40, 39]. Therefore, similarly to Actors, in Pi-calculus processes can dynamically modify the topology. However, contrary to the Actor model, communications in Pi-calculus are based on simultaneous execution of complementary actions, they are synchronous.

3.3.1.2 Determinism and Non-determinism

The Actor Model uses asynchronous communications, while π-calculus uses synchronous communications. Because the concurrent executions and the communications in such system are both deterministic, the result of the concurrent system is assured to be deterministic. The correctness of the execution of deterministic systems is guaranteed.

On the other hand, asynchronous communications are non-deterministic. The message sent can take an infinite time to be received. Therefore, the result of the concurrent system is not assured to be deterministic.

But the communication in reality are subject to various fault and attacks [**Lamport1982**]. And the wait required by synchronous communication negatively impact performances of the system because of the difference between communication latency, and execution latency. The Actor model was explicitly designed to take these physical limitations in account [**Hewitt1977a**].

Moreover, the total ordering of messages is only local to an actor, while between actors, messages are causally ordered. As Lamport showed [34], and Reed related later [51], causal order is sufficient to build a correct distributed system. The non-determinism in the asynchronous communications is hidden by the organization of the system. The execution will either terminate correctly, or not terminate at all.

Eventually, following works adopted asynchronous communications. Indeed, it is not realistic to build a distributed system based on synchronous communications.

3.3.2 Concurrent Programming

As demonstrated by the theory, concurrency basically boils down to message passing. However, there exist several programming model abstracting more or less this theoretical view.

3.3.2.1 Independent Processes

The theory advocates asynchronous message-passing, but it doesn't precise the granularity of the actors. In the Actor Model, everything is an actor, even the simplest types, like numbers. In practice, contrary to OOP, this level of granularity is unachievable due to the asynchronous communication overhead. Most implementations adopt a granularity on the process or function level.

The first concurrent programming concept using message passing was the coroutine. It influenced many following works. Conway defines coroutines as an autonomous program which communicate with adjacent modules as if they were input and output subroutines.[12] It is the first definition of a pipeline to implement multi-pass algorithms. Similar works include the Communicating Sequential Processes (CSP) [**Brookes1984**, 28], and the Kahn Networks [31, 32].

These programming models differ from the Actor Model, because they don't allow to dynamically modify the topology of the application. Coroutines and processes are defined statically in the source of the application. We shall come back to this limitation later in this thesis.

3.3.2.2 Synchronization

These programming model allowed parallel execution, but at the time, the machine featured a single execution unit or shared resources among execution units, like a common memory store, or network interface. Multiprogramming was used to allow different programs to be executed concurrently in isolated processes, and to share resources [15]. To synchronize the different processes over these resources, and avoid conflicting accesses, it is crucial to assure the mutual exclusion. For this purpose, Djikstra introduced the Semaphore [**Dijkstra**]. Similar works include guarded commands [13], guarded region [**Hansen1978a**] and monitors [29].

Multi-Threading Multi-threading programming make use of synchronization within isolated processes. Threads are light processes sharing the same memory execution context within an isolated process. It seems to be an easy solution to parallelize sequential execution on parallel execution units with a common memory store. But because of the preemptive scheduling, threads require to synchronize over each and every shared memory cell. It is known that this heavy synchronization leads to bad performances, and is difficult to develop with [4].

PGAS The Partitioned Global Address Space (PGAS) model is another approach to provide a uniform memory access for the developer. Each computing node executes the same program, and provide its local memory to be shared with all the other nodes. The PGAS programming model assure the remote accesses and synchronization of memory across nodes, and enforces locality of reference, to reduce the communication overhead. Known implementation of the PGAS model are Chapel[**Chamberlain2007**], X10 [**Charles2005**]. Unified Parallel C [**El-Ghazawi2006**], CoArray Fortran [**Numrich1998**] and OpenSHMEM [**Chapman2010**].

Scalability Limitation Amdahl and later Ghunter theorized the speedup gains with parallelism for a sequential program [7]. Their conclusion is that sharing resources protected by mutual exclusion eventually decreases performances when increasing parallelism [24, 22, 23, 46, 21].

The execution regions requiring the same resource needs to execute sequentially. This wait impacts performances negatively because of contention. Therefore, to increase parallelism one needs to increase the number of independent processes, and to ensure their communicate to be solely by asynchronous messages without waiting.

3.3.2.3 Programming languages

Some programming languages features message-passing and isolation of actors directly. To some extent, these languages succeeded in industrial contexts. However, they largely remain elitist solutions for specific problems more than a general, and accessible tool.

Scala is an attempt at unifying the object model and functional programming [**Odersky2004**]. Akka[3] is a framework based on Scala, to build higly scalable and resilient applications.

Erlang is a functional concurrent language designed by Ericsson to operate telecommunication devices [**JoeArmstrong**]. *

TODO extends these paragraphs

3.3.3 Stream Processing Systems

All the solutions previously presented are generally designed to build distributed systems. We focus on real-time applications as defined by [25]. A real-time application must respond to a variety of simultaneous requests within a certain time. Otherwise, input data may be lost or output data may lose their significance. Such applications nowadays are often connected to the internet, which implies to process high volumes streams of requests. Moreover, because these systems are key to business, their reliability and latency are of critical importance. These requirements are challenging to meet in the design of such system. We present the state of the art.

3.3.3.1 Data-stream management systems

The processing of large volume of data was historically handled by Database management systems. These systems naturally evolved to manage data-streams as well. They continuously run SQL-like requests on data streams. The computation of these requests spread over a distributed architecture. Among the early works, we can cite NiagaraCQ [11, 45], Aurora [1, 3, 8] which

[3] http://akka.io/

evolved into Borealis [2], AQuery [**Lerner2003**], STREAM [**Arasu2003**, **Arasu2005**] and TelegraphCQ [33, 10]. More recently, we can cite DryadLINQ [30, 59], Timestream [50] and Shark [**Xin2013**].

3.3.3.2 Dataflow pipeline

Another model successful model to process data stream is the pipeline architecture.

SEDA is a precursor in the design of pipeline-based architecture for real-time applications for the internet [57]. It organizes an application as a network of event-driven stages connected by explicit queues. Several projects followed and adapted the principles in this work.

StreaMIT is a language to help the programming of large streaming application [54].

Storm [55] is designed by and used at Twitter to process the flow of tweets, and calculate metrics such as the trending topics. It is only one example of industrial practical application, among many others. We can cite CBP [36] and S4 [47], that were designed at Yahoo, Millwheel [5] designed at Google and Naiad [**Murray2013**] designed at Microsoft.

3.4 Reconciliations

3.4.1 Design patterns

3.4.1.1 Algorithmic Skeletons

[37] Mc Cool, Structured Parallel Programmin with Deterministic Patterns
The general idea is that specific combinations of computation and data access recur in many different algorithms.

3.4.1.2 Accelerators

-> Annotations CUDA, OpenCL

Difference between skeletons and accelerators ?

Imperative Piccolo Parallel in-memory [49] CIEL Stateless dataflow [44] Statdeful Dataflow Graph (SDG) Stateful dataflow [17]

3.4.1.3 Lock-free Algorithm

Lock-free algorithm are highly concurrent, as they can be replicated, however, they are limited, and really hard to develop. `https://en.wikipedia.org/wiki/Non-blocking_algorithm`

3.4.1.4 Microservices & SOA

Microservices are in the reconciliation category, I think. It is an attempt at reconciling the two organization. They advocate that software developers can manage the two organizations at a sufficiently fine level. However, it doesn't support growth as well as sequential programming.

3.4.2 Compilation

Parnas already advocated conciling the two methods in its Information Hiding paper. Using an assembler to transform the high-level, development, vision into the low-level, execution, vision.

3.4.2.1 Cyclic parallelism

Data parallelism / Vectorization

3.4.2.2 Pipeline parallelism

Task parallelism / Pipeline parallelism

An Overview of the SUIF Compiler for Scalable Parallel Machines. [6]

Interesting articles :

http://comjnl.oxfordjournals.org/content/early/2015/09/15/comjnl.bxv077.abstract

-> THIS, to read Load balanced pipeline parallelism **[Kamruzzaman2013]**

3.4.2.3 Static analysis

Javascript static analysis here

3.5 Proposed work

My objective in this thesis is to find a reconciliation of the two goals, by finding an equivalence between two approaches with different goals, in the case of streaming web applications.

We show that there is no languages that features higher-order functions to improve modularity, a common memory store easy to develop with, but at the same time provides scalable concurrency.

We aim at filling this gap, and for a concrete example, focus our work on the Javascript programing language. Indeed, Javascript features higher-order functions, is highly-used in concurrent context, but lacks scalable concurrency.

Our work is divided into two contributions: Due and Fluxions.

-> Schema roadmap.

Chapter 4

Due

4.1 Introduction

Javascript was originally designed for the manipulation of a graphical interface, the Document Object Model (DOM[1]). Functions are first-class citizens ; it allows to manipulate them like any object, and to link them to react to asynchronous events, *e.g.* user inputs and remote requests. These asynchronously triggered functions are named callbacks, and allow to efficiently cope with the distributed and inherently asynchronous architecture of the Internet. This made Javascript a language of choice to develop both client and, more recently, server applications for the web.

Callbacks are well-suited for small interactive scripts. But in a complete application, they are ill-suited to control the larger asynchronous execution flow. Their use leads to intricate imbrications of function calls and callbacks, commonly presented as *callback hell*[2], or *pyramid of doom*. This is widely recognized as a bad practice and reflects the unsuitability of callbacks in complete applications. Eventually, developers enhanced callbacks to meet their needs with the concept of Promise [35].

Promises bring a different way to control the asynchronous execution flow, better suited for large applications. They fulfill this task well enough to be part of the next version of the Javascript language, ECMAScript 6[3]. However, because Javascript started as a scripting language, beginners are

[1] `http://www.w3.org/DOM/`

[2] `http://maxogden.github.io/callback-hell/`

[3] `http://people.mozilla.org/~jorendorff/es6-draft.html`

often not introduced to Promises early enough. Most APIs use the classical callback approach encouraging beginner in this practice. Moreover, despite its benefits, the concept of Promise is not yet widely acknowledged. Developers may implement their own library for asynchronous flow control before discovering existing ones. There is such a disparity between the needs for and the adoption of Promises libraries, that there are almost 40 different implementations[4].

With the upcoming introduction of Promise as a language feature, we expect an increase of interest, and believe that many developers will shift to this better practice. In this paper, we propose a compiler to automate this shift in existing code bases. We present the transformation from an imbrication of callbacks to a sequence of Promise operations, while preserving the semantic.

Promises bring a better way to control the asynchronous execution flow, but they also impose a conditional control over the result of the execution. Callbacks, on the other hand, leave this conditional control to the developer. This paper focuses on the transformation from imbrication of callbacks to chain of Promises. To avoid unnecessary modifications on this conditional control, we introduce an alternative to Promises leaving this conditional control to the developer, like callbacks. We call this simpler specification Dues. Our approach enables us to compile legacy Javascript code and brings a first automated step toward full Promises integration. This simple and pragmatic compiler has been tested over 64 *Node.js* packages from the node package manager (npm[5]), 9 of them with success.

In section 4.2 we define callbacks, Promises and Dues. In section 4.3, we explain the transformation from imbrications of callbacks to sequences of Dues. In section 5.3, we present a compiler to automate the application of this equivalence. In section 5.4, we evaluate the developed compiler.

[4]`https://github.com/promises-aplus/promises-spec/blob/master/implementations.md`

[5]`https://www.npmjs.com/`

4.2 Definitions

4.2.1 Callback

A callback is a function passed as a parameter to a function call. It is invoked by the callee to continue the execution with data not available in the caller context. We distinguish three kinds of callbacks.

Iterators are functions called for each item in a set, often synchronously.

Listeners are functions called asynchronously for each event in a stream.

Continuations are functions called asynchronously once a result is available.

As we will see later, Promises are designed as placeholders for a unique outcome. Iterators and Listeners are invoked multiple times resulting in multiple outcomes. Only continuations are equivalent to Promises. Therefore, we focus on continuations in this paper.

Callbacks are often mistaken for continuations; the former are not inherently asynchronous while the latter are. In a synchronous paradigm, the sequentiality of the execution flow is trivial. An operation needs to complete before executing the next one. In an asynchronous paradigm, parallelism is trivial, but the sequentiality of operations needs to be explicit. Continuations are the functional way of providing this control over the sequentiality of the asynchronous execution flow.

A continuation is a function passed as an argument to allow the callee not to block the caller until its completion. The caller is able to continue the execution while the callee runs in background. The continuation is invoked later, at the termination of the callee to continue the execution as soon as possible and process the result; hence the name continuation. It provides a necessary control over the asynchronous execution flow. It also brings a control over the data flow which essentially replaces the `return` statement at the end of a synchronous function. At its invocation, the continuation retrieves both the execution flow and the result.

The convention on how to hand back the result must be common for both the callee and the continuation. For example, in *Node.js*, the signature of a continuation uses the *error-first* convention. The first argument contains an error or `null` if no error occurred; then follows the result. Listing 4.1 is a pattern of such a continuation. However, continuations don't impose any conventions; indeed, other conventions are used in the browser.

```
my_fn(input, function continuation(error, result) {
  if (!error) {
    console.log(result);
  } else {
    throw error;
  }
});
```

Listing 4.1 – Example of a continuation

The callback hell occurs when many asynchronous calls are arranged to be executed sequentially. Each consecutive operation adds an indentation level, because it is nested inside the continuation of the previous operation. It produces an imbrication of calls and function definitions, as shown in listing 4.2. We say that continuations lack the chained composition of multiple asynchronous operations. Promises allow to arrange such a sequence of asynchronous operations in a more concise and readable way.

```
my_fn_1(input, function cont(error, result) {
  if (!error) {
    my_fn_2(result, function cont(error, result) {
      if (!error) {
        my_fn_3(result, function cont(error, result) {
          if (!error) {
            console.log(result);
          } else {
            throw error;
          }
        });
      } else {
        throw error;
      }
    });
  } else {
    throw error;
  }
});
```

Listing 4.2 – Example of a sequence of continuations

4.2.2 Promise

In a synchronous paradigm, the sequentiality of the execution flow is trivial. While in an asynchronous paradigm, this control is provided by continuations. Promises provide a unified control over the execution flow for both paradigms. The ECMAScript 6 specification[6] defines a Promise as an object that is used as a placeholder for the eventual outcome of a deferred (and possibly asynchronous) operation. Promises expose a `then` method which expects a continuation to continue with the result; this result being synchronously or asynchronously available.

Promises force another control over the execution flow. According to the outcome of the operation, they call one function to continue the execution with the result, or another to handle errors. This conditional execution is indivisible from the Promise structure. As a result, Promises impose a convention on how to hand back the outcome of the deferred computation, while classic continuations leave this conditional execution to the developer.

```
var promise = my_fn_pr(input)

promise.then(function onSuccess(result) {
  console.log(result);
}, function onError(error) {
  throw error;
});
```

Listing 4.3 – Example of a promise

Promises are designed to fill the lack of chained composition from continuations. They allow to arrange successions of asynchronous operations as a chain of continuations, by opposition to the imbrication of continuations illustrated in listing 4.2. That is to arrange them, one operation after the other, in the same indentation level.

The listing 4.4 illustrates this chained composition. The functions `my_fn_pr_2` and `my_fn_pr_3` return promises when they are executed, asynchronously. Because these promises are not available synchronously, the method `then` synchronously returns intermediary Promises. The latter resolve only when the former resolve. This behavior allows to arrange the continuations as a flat chain of calls, instead of an imbrication of continuations.

```
my_fn_pr_1(input)
.then(my_fn_pr_2, onError)
.then(my_fn_pr_3, onError)
.then(console.log, onError);

function onError(error) {
  throw error;
```

[6] `https://people.mozilla.org/~jorendorff/es6-draft.html#sec-promise-objects`

```
}
```

Listing 4.4 – A chain of Promises is more concise than an imbrication of continuations

The Promises syntax is more concise, and also more readable because it is closer to the familiar synchronous paradigm. Indeed, Promises allow to arrange both the synchronous and asynchronous execution flow with the same syntax. It allows to easily arrange the execution flow in parallel or in sequence according to the required causality. This control over the execution leads to a modification of the control over the data flow. Programmers are encouraged to arrange the computation as series of coarse-grained steps to carry over inputs. In this sense, Promises are comparable to some coarse-grained data-flow programming paradigms, such as Flow-based programming [43].

4.2.3 From continuations to Promises

As detailed in the previous sections, continuations provide the control over the sequentiality of the asynchronous execution flow. Promises improve this control to allow chained compositions, and unify the syntax for the synchronous and asynchronous paradigm. This chained composition brings a greater clarity and expressiveness to source codes. At the light of these insights, it makes sense for a developer to switch from continuations to Promises. However, the refactoring of existing code bases might be an operation impossible to carry manually within reasonable time. We want to automatically transform an imbrication of continuations into a chained composition of Promises.

We identify two steps in this transformation. The first is to provide an equivalence between a continuation and a Promise. The second is the composition of this equivalence. Both steps are required to transform imbrications of continuations into chains of Promises.

Because Promises bring chained composition, the first step might seem trivial as it does not imply any imbrication to transform into chain. However, as explained in section 4.2.2, Promises impose a control over the execution flow that continuations leave free. This control induces a common convention to hand back the outcome to the continuation.

In the Javascript landscape, there is no dominant convention for handing back outcomes to continuations. In the browser, many conventions coexist. For example, *jQuery*'s `ajax`[7] method expects an object with different continuations for success, errors and various other events during the asynchronous operation. *Q*[8], a popular library to control the asynchronous flow, exposes two methods to define continuations: `then` for successes, and `catch` for errors. On the other hand, the *Node.js* API always used the *error-first* convention, encouraging developers to provide libraries using the same convention. In this large ecosystem the *error-first* convention is predominant. All these examples use different conventions than the Promise specification detailed in section 4.2.2. They present strong semantic differences, despite small syntactic differences.

To translate these different conventions into the Promises one, the compiler would need to identify them. Such an identification might be possible with static analysis methods such as the points-to analysis [56], or a program logic [19, 9]. However, it seems impracticable because of the number and semantical heterogeneity of these conventions. Indeed, in the browser, each library seems to provide its own convention.

In this paper, we are interested in the transformation from imbrications to chains, not from one convention to another. The *error-first* convention, used in *Node.js*, is likely to represent a large, coherent code base to test the equivalence. Indeed contains currently more than 125 000 packages. For this reason, we focus only on the *error-first* convention. Thus, our compiler is only able to compile code that follows this convention. The convention used by Promises is incompatible. We propose an alternative specification to Promise following the *error-first* convention. In the next section we present this specification called Due.

The choice to focus on *Node.js* is also motivated by our intention to compare later the chained sequentiality of Promises with the data-flow paradigm. *Node.js* allows to manipulate streams of messages. This proved to be efficient for real-time web applications manipulating streams of user requests. Both Promises and data-flow arrange the computation in chains of independent operations.

[7] `http://api.jquery.com/jquery.ajax/`

[8] `http://documentup.com/kriskowal/q/`

4.2.4 Due

A Due is an object used as placeholder for the eventual outcome of a deferred operation. Dues are a simplification of the Promise specification. They are essentially similar to Promises, except for the convention to hand back outcomes. They use the *error-first* convention, like *Node.js*, as illustrated in listing 4.5. The implementation of Dues and its tests are available online[9]. A more in-depth description of Dues and their creation follows in the next paragraphs.

```
1 var my_fn_due = require('due').mock(my_fn);
2
3 var due = my_fn_due(input);
4
5 due.then(function continuation(error, result) {
6   if (!error) {
7     console.log(result);
8   } else {
9     throw error;
10  }
11 });
```

Listing 4.5 – Example of a due

A due is typically created inside the function which returns it. In listing 4.5, line 1, the `mock` method wraps `my_fn` in a Due-compatible function. The rest of this code is similar to the Promise example, listing 4.3.

We illustrate in listing 4.6 the creation of a Due through the `mock` method. At its creation, line 6, the Due expects a callback containing the deferred operation, which is `my_fn` here. This callback is executed synchronously with the function `settle` as argument to settle the Due, synchronously or asynchronously. The `settle` function is pushed at the end of the list of arguments. The callback invokes the deferred operation with this list of arguments, and the current context, line 8. When finished, the latter calls `settle` to settle the Due and save the outcome. Settled or not, the created Due is always synchronously returned. Its `then` method allows to define a continuation to retrieve the saved outcome, and continue the execution after its settlement. If the deferred operation is synchronous, the Due settles during its creation and the `then` method immediately calls this continuation. If the deferred operation is asynchronous, this continuation is called during the Due settlement.

```
1 Due.mock = function(my_fn) {
```

[9] https://www.npmjs.com/package/due

```
  return function mocked_fn() {
    var _args = Array.prototype.slice.call(arguments),
        _this = this;

    return new Due(function(settle) {
      _args.push(settle);
      my_fn.apply(_this, _args);
    })
  }
}
```

Listing 4.6 – Creation of a due

The composition of Dues is the same than for Promises (see section 4.2.2). Through this chained composition, Dues arrange the execution flow as a sequence of actions to carry on inputs.

This simplified specification adopts the same convention than *Node.js* for continuations to hand back outcomes. Therefore, the equivalence between a continuation and a Due is trivial. Dues are admittedly tailored for this paper, hence, they are not designed to be written by developers, like Promises are. They are an intermediary step between classical continuations and Promises. We present in section 4.3 the equivalence between continuations and Dues.

4.3 Equivalence

We present the transformation from a nested imbrication of continuations into a chain of Dues. We explain the three limitations imposed by our compiler for this transformation to preserve the semantic. They preserve the execution order, the execution linearity and the scopes of the variables used in the operations.

4.3.1 Execution order

Our compiler spots function calls with a continuation, which are similar to the abstraction in (4.1). It wraps the function fn into the function $fn_{\mathbf{due}}$ to return a Due. And it relocates the continuation in a call to the method **then**, which references the Due previously returned. The result should be similar to (4.2). The differences are highlighted in bold font.

$$fn([arguments], continuation) \tag{4.1}$$

$$fn_{\mathbf{due}}([arguments])\mathbf{.then}(continuation) \tag{4.2}$$

The execution order is different whether *continuation* is called synchronously, or asynchronously. If fn is synchronous, it calls the *continuation* within its execution. It might execute *statements* after executing *continuation*, before returning. If fn is asynchronous, the continuation is called after the end of the current execution, after fn. The transformation erases this difference in the execution order. In both cases, the transformation relocates the execution of *continuation* after the execution of fn. For synchronous fn, the execution order changes ; the execution of *statements* at the end of fn and the continuation switch. The latter must be asynchronous to preserve the execution order.

4.3.2 Execution linearity

Our compiler transforms a nested imbrication of continuations, which is similar to the abstraction in (4.3) into a flatten chain of calls encapsulating them, like in (4.4).

$$\begin{aligned}
&fn1([arguments], cont1\{\\
&\quad declare\ variable \leftarrow result\\
&\quad fn2([arguments], cont2\{\\
&\qquad print\ variable\\
&\quad \})\\
&\})
\end{aligned} \tag{4.3}$$

$$\begin{aligned}
&\textbf{declare variable}\\
&fn1_{\textbf{due}}([arguments])\\
&\textbf{.then}(cont1\{\\
&\quad variable \leftarrow result\\
&\quad fn2_{\textbf{due}}([arguments])\\
&\})\\
&\textbf{.then}(cont2\{\\
&\quad print\ variable\\
&\})
\end{aligned} \tag{4.4}$$

An imbrication of continuations must not contain any loop, nor function definition that is not a continuation. Both modify the linearity of the execution flow which is required for the equivalence to keep the semantic. A call nested inside a loop returns multiple Dues, while only one is returned to continue the chain. A function definition breaks the execution linearity. It prevent the nested call to return the Due expected to continue the chain. On the other hand, conditional branching leaves the execution linearity and the semantic intact. If the nested asynchronous function is not called, the execution of the chain stops as expected.

4.3.3 Variable scope

In (4.3), the definitions of $cont1$ and $cont2$ are overlapping. The *variable* declared in $cont1$ is accessible in $cont2$ to be printed. In (4.4), however, definitions of $cont1$ and $cont2$ are not overlapping, they are siblings. The *variable* is not accessible to $cont2$. It must be relocated in a parent function to be accessible by both $cont1$ and $cont2$. To detect such variables, the compiler must infer their scope statically. Languages with a lexical scope define the scope of a variable statically. Most imperative languages present a lexical scope, like C/C++, Python, Ruby or Java. The subset of Javascript excluding the built-in functions `with` and `eval` is also lexically scoped. To compile Javascript, the compiler must exclude programs using these two statements.

4.4 Compiler

We build a compiler to automate the application of this equivalence on existing Javascript projects. The compilation process contains two important steps, the identification of the continuations, and the generation of chains.

4.4.1 Identification of continuations

The first compilation step is to identify the continuations and their imbrications. The nested imbrication of callbacks only occurs when they are defined *in situ*. The compiler detects a function definition within the arguments of a function call. This detection is based on the syntax, and is trivial.

Not all detected callbacks are continuations, but the equivalence is applicable only on the latter. A continuation is a callback invoked only once, asynchronously. Spotting a continuation implies to identify these two conditions. There is no syntactical difference between a synchronous and an asynchronous callee. And it is impossible to assure a callback to be invoked only once, because the implementation of the callee is often statically unavailable. Therefore, the identification of continuations is necessarily based on semantical differences. To recognize these differences, the compiler would need to have a deep understanding of the control and data flows of the program. Because of the highly dynamic nature of Javascript, this understanding is either unsound, limited, or complex. Instead, we choose to leave to the developer the identification of compatible continuations among the identified callbacks. They are expected to understand the limitations of this compiler, and the semantic of the code to compile.

We provide a simple interface for developers to interact with the compiler. We built this interface around the compiler in a web page available online[10] to reproduce the tests. The web technologies allow to quickly build an interface for a wide variety of computing devices.

This interaction prevents the complete automation of the individual compilation process. However, we are working on an automation at a global scale. We expect to be able to identify a continuation only based on the name of its callee, *e.g.* `fs.readFile`. We built a service to gather these names along with their identification. The compiler queries this service to present to the developer an estimated identification. After the compilation, it sends back the identification corrected by the developer to refine the future estimations. In future works, we would like to study the possibility for such a service to assist, and ease the compilation process.

4.4.2 Generation of chains

The compositions of continuations and Dues are arranged differently. Continuations structure the execution flow as a tree, while a chain of Dues imposes to arrange it sequentially. A parent continuation can execute several children, while a Due allow to chain only one. The second compilation step is to identify the imbrications of continuations, and trim the extra branches to transform them into chains.

If a continuation has more than one child, the compiler tries to find a single legitimate child to form the longest chain possible. This legitimate child is the only parent among its siblings. If there are several parents among the children, none are the legitimate child. The non legitimate children start a new tree. This step transform each tree of continuations into several chains

[10] `compiler-due.apps.zone52.org`

of continuations that translate into sequences of Dues. The code generation from these chains is straightforward from the equivalence.

4.5 Evaluation

To validate our compiler, we compile several Javascript projects likely to contain continuations. We present the results of these tests.

The compilation of a project requires user interaction. To conduct the test in a reasonable time, we limit the test set to a minimum. We search the *Node Package Manager* database to restrict the set to *Node.js* projects. We refine the selection to web applications depending on the web framework *express*, but not on the most common Promises libraries such as *Q* and *Async*. We refine further the selection to projects using the test frameworks *mocha* in its default configuration. We use these tests to validate the compiler. The test set contains 64 projects. This subset is very small, and cannot represent the wide possibilities of Javascript. However, we believe it is sufficient to represent a majority of common cases.

For each project, we verify that is is correctly tested, and passes the tests. During the compilation, we identify the compatible continuations among the detected callbacks. We apply the unmodified test on the compilation result. The compilation result should pass the tests as well. This is not a strong validation, but it assures the compiler to work as expected in most common cases.

Of the 64 projects tested, almost a half, does not contain any compatible continuations. We reckon that these projects use continuations the compiler is unable to detect. The other projects were rejected by the compiler because they contain `with` or `eval` statements, they use Promises libraries we didn't filter previously. 9 projects compiled successfully. The compiler did not fail to compile any project of the initial test set.

Over the 9 successfully compiled projects, the compiler detected 172 callbacks. We manually identified 56 of them to be compatible continuations. The false positives are mainly the listeners that the web applications register to react to user requests.

One project contains 20 continuations, the others contains between 1 and 9 continuations each. On the 56 continuations, 36 are single. The others 20 continuations belong to imbrications of 2 to 4 continuations. The result of this evaluation prove the compiler to be able to successfully transform imbrications of continuations.

On the 64 projects composing the test set

29 (45.3%) do not contain any compatible continuations,

10 (15.6%) are not compilable because they contain `with` or `eval` statements,

5 (7.8%) use less common asynchronous libraries we didn't filter previously,

4 (6.3%) are not syntactically correct,

4 (6.3%) fail their tests before the compilation,

3 (4.7%) are not tested, and

10 (14.0%) compile successfully.

The compiler do not fail to compile any project. The details of these projects are available in Appendix **??**.

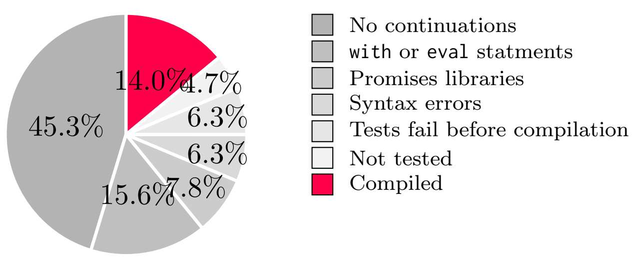


Figure 4.1 – Compilation results distribution

Chapter 5

Fluxion

5.1 Introduction

5.2 Fluxional execution model

In this section, we present an execution model to provide scalability to web applications. To achieve this, the execution model provides a granularity of parallelism at the function level. Functions are encapsulated in autonomous execution containers with their state, so as to be reallocated and executed in parallel. This execution model is close to the actors model, as the execution containers are independent and communicate by messages. The communications are assimilated to stream of messages, similarly to the dataflow programming model. It allows to reason on the throughput of these streams, and to react to load increases.

The fluxional execution model executes programs written in our high-level fluxionnal language, whose grammar is presented in figure 5.1. An application ⟨program⟩ is partitioned into parts encapsulated in autonomous execution containers named *fluxions* ⟨flx⟩. In the following paragraphs, we present the *fluxions*. Then we present the messaging system to carry the communications between *fluxions*. Finally, we present an example application using this execution model.

$$
\begin{array}{rcl}
\langle\text{program}\rangle & \models & \langle\text{flx}\rangle \;|\; \langle\text{flx}\rangle\; \textit{eol}\; \langle\text{program}\rangle \\
\langle\text{flx}\rangle & \models & \texttt{flx}\; \langle\text{id}\rangle\; \langle\text{tags}\rangle\; \langle\text{ctx}\rangle\; \textit{eol}\; \langle\text{streams}\rangle\; \textit{eol}\; \langle\text{fn}\rangle \\
\langle\text{tags}\rangle & \models & \texttt{\&}\; \langle\text{list}\rangle \;|\; \textit{empty string} \\
\langle\text{streams}\rangle & \models & \texttt{null} \;|\; \langle\text{stream}\rangle \;|\; \langle\text{stream}\rangle\; \textit{eol}\; \langle\text{streams}\rangle \\
\langle\text{stream}\rangle & \models & \langle\text{type}\rangle\; \langle\text{dest}\rangle\; [\langle\text{msg}\rangle] \\
\langle\text{dest}\rangle & \models & \langle\text{list}\rangle \\
\langle\text{ctx}\rangle & \models & \texttt{\{}\langle\text{list}\rangle\texttt{\}} \\
\langle\text{msg}\rangle & \models & \texttt{[}\langle\text{list}\rangle\texttt{]} \\
\langle\text{list}\rangle & \models & \langle\text{id}\rangle \;|\; \langle\text{id}\rangle\; ,\; \langle\text{list}\rangle \\
\langle\text{type}\rangle & \models & \texttt{>>} \;|\; \texttt{->} \\
\langle\text{id}\rangle & \models & \textit{Identifier} \\
\langle\text{fn}\rangle & \models & \textit{imperative language and stream syntax}
\end{array}
$$

Figure 5.1 – Syntax of a high-level language to represent a program in the fluxionnal form

5.2.1 Fluxions

A *fluxion* ⟨flx⟩ is named by a unique identifier ⟨id⟩ to receive messages, and might be part of one or more groups indicated by tags ⟨tags⟩. A *fluxion* is composed of a processing function ⟨fn⟩, and a local memory called a *context* ⟨ctx⟩. At a message reception, the *fluxion* modifies its *context*, and sends messages on its output streams ⟨streams⟩ to downstream *fluxions*. The *context* handles the state on which a *fluxion* relies between two message receptions. In addition to message passing, the execution model allows *fluxions* to communicate by sharing state between their *contexts*. The fluxions that need to synchronize together are grouped with the same tag, and loose their independence.

There are two types of streams, *start* and *post*, which correspond to the nature of the rupture point yielding the stream. We differentiate the two types with two different arrows, double arrow (`>>`) for *start* rupture points and simple arrow (`->`) for *post* rupture points. The two types of rupture points are further detailed in section 5.3.1.1.

5.2.2 Messaging system

The messaging system assures the stream communications between fluxions. It carries messages based on the names of the recipient fluxions. After the execution of a fluxion, it queues the resulting messages for the event loop to process.

The execution cycle of an example fluxional application is illustrated in figure 5.2. Circles represent registered fluxions. The source code for this application is in listing 5.1 and the fluxional code for this application is in listing 5.2. The fluxion *reply* has a context containing the variable `count` and `template`. The plain arrows represent the actual message paths in the messaging system, while the dashed arrows between fluxions represent the message streams as seen in the fluxionnal application.

The *main* fluxion is the first fluxion in the flow. When the application receives a request, this fluxion triggers the flow with a `start` message containing the request, ②. This first message is to be received by the next fluxion *handler*, ③ and ④. The fluxion *handler* sends back a message, ⑤, to be enqueued, ⑥. The system loops through steps ③ through ⑥ until the queue is empty. This cycle starts again for each new incoming request causing another `start` message.

5.2.3 Service example

To illustrate the fluxional execution model, and the compiler, we present in listing 5.1 an example of a simple web application. This application reads a file, and sends it back along with a request counter.

```
1  var app = require('express')(),
2      fs = require('fs'),
3      count = 0;
4
5  app.get('/', function handler(req, res){
6    fs.readFile(__filename, function reply(err, data) {
7      count += 1;
8      res.send(err || template(count, data));
9    });
10 });
11
12 app.listen(8080);
```

Listing 5.1 – Example web application

The `handler` function, line 5 to 11, receives the input stream of request. The `count` variable at line 3 increments the request counter. This object

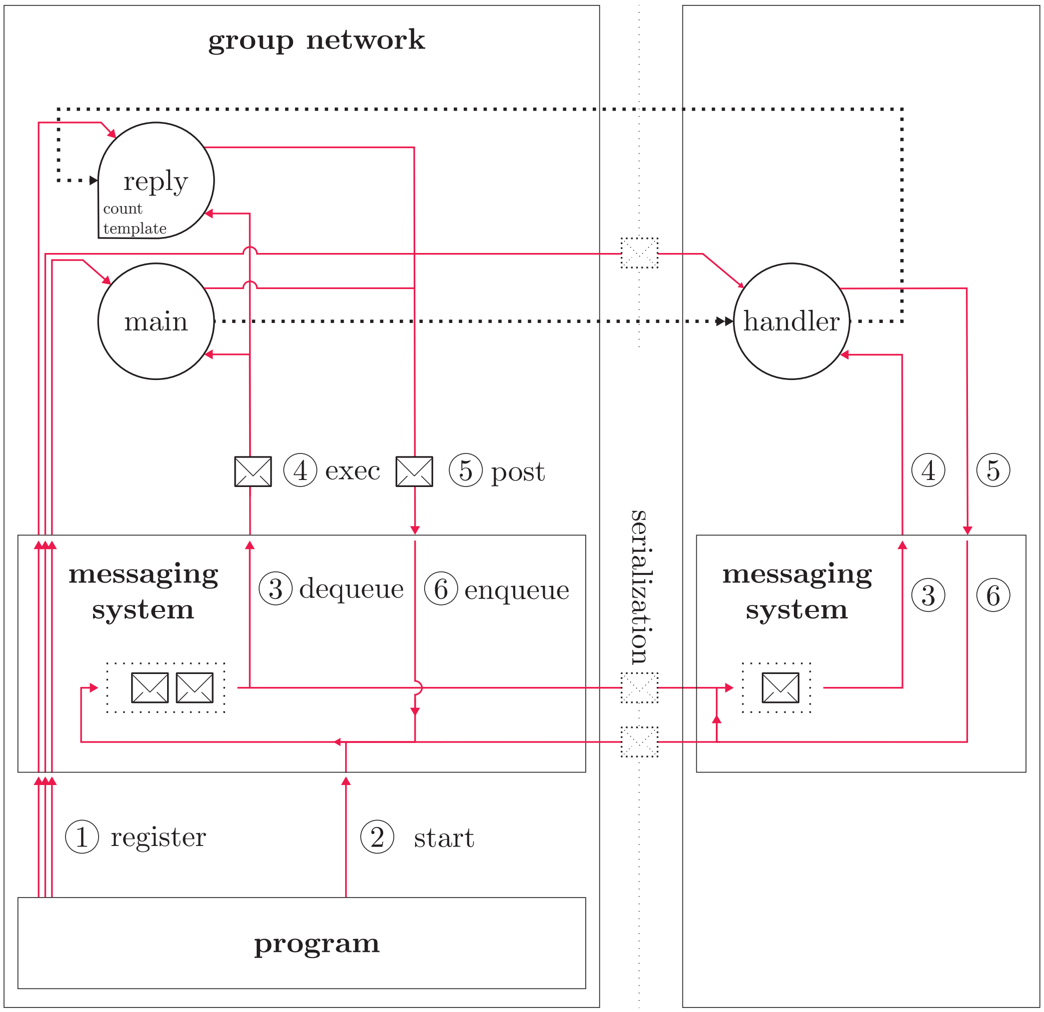


Figure 5.2 – The fluxionnal execution model in details

needs to be persisted in the fluxion *context*. The `template` function formats the output stream to be sent back to the client. The `app.get` and `res.send` functions, respectively line 5 and 8, interface the application with the clients. And between these two interface functions is a chain of three functions to process the client requests : `app.get` $\twoheadrightarrow$ `handler` $\rightarrow$ `reply`. This application is transformed into the high-level fluxionnal language in listing 5.2 which is illustred in Figure 5.2.

```
flx main & network
>> handler [res]
```

```
  var app = require('express')(),
      fs = require('fs'),
      count = 0;

  app.get('/', >> handler); //
  app.listen(8080);

flx handler
-> reply [res]
  function handler(req, res) {
    fs.readFile(__filename, -> reply); //
  }

flx reply & network {count, template}
-> null
  function reply(error, data) {
    count += 1; //
    res.send(err || template(count, data)); //
  }
```

Listing 5.2 – Example application expressed in the high-level fluxional language

The application is organized as follow. The flow of requests is received from the clients by the fluxion `main`, it continues in the fluxion `handler`, and finally goes through the fluxion `reply` to be sent back to the clients. The fluxions `main` and `reply` have the tag `network`. This tag indicates their dependency over the network interface, because they received the response from and send it back to the clients. The fluxion `handler` doesn't have any dependencies, hence it can be executed in parallel.

The last fluxion, `reply`, depends on its context to holds the variable `count` and the function `template`. It also depends on the variable `res` created by the first fluxion, `main`. This variable is carried by the stream through the chain of fluxion to the fluxion `reply` that depends on it. This variable holds the references to the network sockets. It is the variable the group `network` depends on.

Moreover, if the last fluxion, `reply`, did not relied on the variable `count`, the group `network` would be stateless. The whole group could be replicated as many time as needed.

This execution model allows to parallelize the execution of an application. Some parts are arranged in pipeline, like the fluxion `handler`, some other parts are replicated, as could be the group `network`. This parallelization improves the scalability of the application. Indeed, as a fluxion contains its state and expresses its dependencies, it can be migrated. It allows to adapt the number of fluxions per core to adjust the resource usage in function of the desired throughput.

Our goal, as described in the introduction, is not to propose a new programming paradigm with this high-level language but to automate the architecture shift. We present the compiler to automate this architecture shift in the next section.

5.3 Fluxionnal compiler

The source languages we focus on should present higher-order functions and be implemented as an event-loop with a global memory. Javascript is such a language : it doesn't require an event-loop, but it is often implemented on top of an event-loop. *Node.js* is an example of such an implementation. We developed a compiler that transforms a *Node.js* application into a fluxional application compliant with the execution model described in section 5.2.

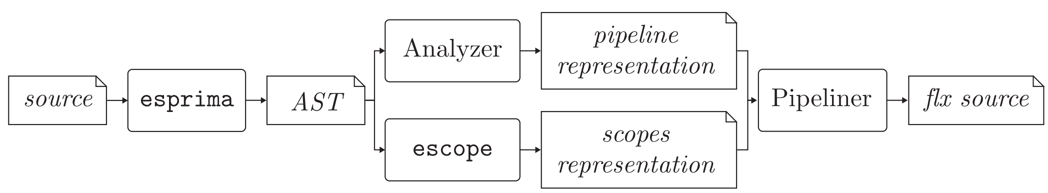


Figure 5.3 – Compilation chain

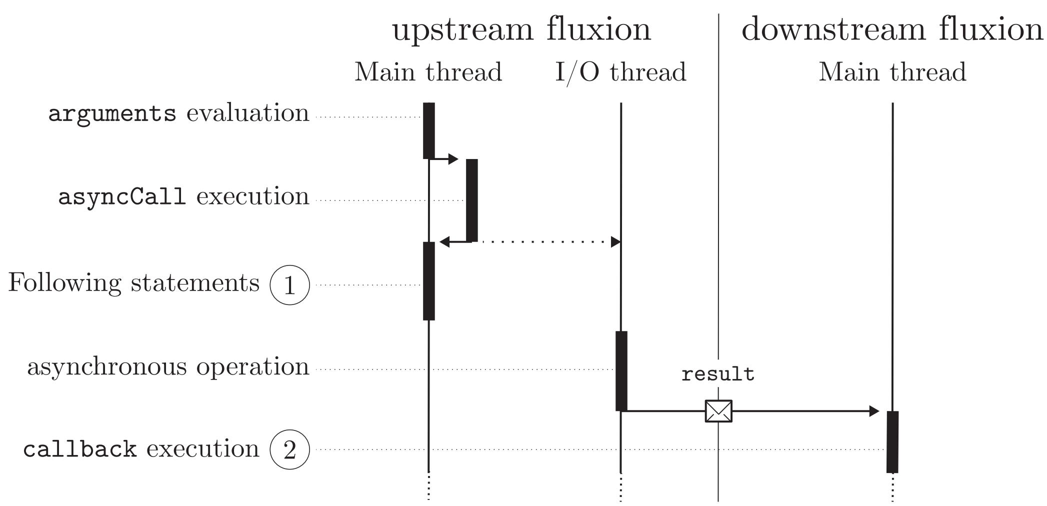
The chain of compilation is described in figure 5.3. From the source of a *Node.js* application, the compiler extracts an Abstract Syntax Tree (AST) with `esprima`. From this AST, the analyzer step identifies the limits of the different application parts and how they relate to form a pipeline. This first step outputs a pipeline representation of the application. Section 5.3.1 explains this first compilation step. In the pipeline representation, the stages are not yet independent and encapsulated into fluxions. From the AST, `escope` produces a representation of the memory scopes. The pipeliner step analyzes the pipeline representation and the scopes representation to distribute the shared memory into independent groups of fluxions. Section 5.3.2 explains this second compilation step.

5.3.1 Analyzer step

The limit between two application parts is defined by a rupture point. The analyzer identifies these rupture points, and outputs a representation of the application in a pipeline form, with application parts as the stages, and rupture points as the message streams of this pipeline.

5.3.1.1 Rupture points

A rupture point is a call of a loosely coupled function. It is an asynchronous call without subsequent synchronization with the caller. In *Node.js*, I/O operations are asynchronous functions and indicate such rupture point between two application parts. Figure 5.4 shows an example of a rupture point with the execution of the two application parts isolated into fluxions. The two application parts are the caller of the asynchronous function call on one hand, and the callback provided to the asynchronous function call on the other hand.




```
asyncCall(arguments, function callback(result){ 2 });
// Following statements 1
```

Figure 5.4 – Rupture point interface

A callback is a function passed as a parameter to a function call. It is invoked by the callee to continue the execution with data not available in the caller context. We distinguish three kinds of callbacks, but only two are asynchronous : listeners and continuations. Similarly, there are two types of rupture points, respectively *start* and *post*.

Start rupture points are indicated by listeners. They are on the border between the application and the outside, continuously receiving incoming user requests. An example of a start rupture point is in listing 5.1, between the call to `app.get()`, and its listener `handler`. These rupture points indicate the input of a data stream in the program, and the beginning of a chain of fluxions to process this stream.

Post rupture points are indicated by continuations. They represent a continuity in the execution flow after an asynchronous operation yielding a unique result, such as reading a file, or querying a database. An example of a post rupture points is in listing 5.1, between the call to `fs.readFile()`, and its continuation `reply`.

5.3.1.2 Detection

The compiler uses a list of common asynchronous callees, like the `express` and file system methods. This list can be augmented to match asynchronous callees individually for any application. To identify the callee, the analyzer walks the AST to find a call expression matching this list.

After the identification of the callee, the callback needs to be identified as well to be encapsulated in the downstream fluxion. For each asynchronous call detected, the compiler test if one of the arguments is of type `function`. Some callback functions are declared *in situ*, and are trivially detected. For variable identifier, and other expressions, the analyzer tries to detect their type. To do so, the analyzer walks back the AST to track their assignations and modifications, and to determine their last value.

5.3.2 Pipeliner step

A rupture point eventually breaks the chain of scopes between the upstream and downstream fluxion. The closure in the downstream fluxion cannot access the scope in the upstream fluxion as expected. The pipeliner step replaces the need for this closure, allowing application parts to rely only on independent memory stores and message passing. It determines the distribution using the scope representation, which represents the variables' dependencies between application parts. Depending on this representation, the compiler can replace the broken closures in three different ways. We present these three alternatives with the example figure 5.5.

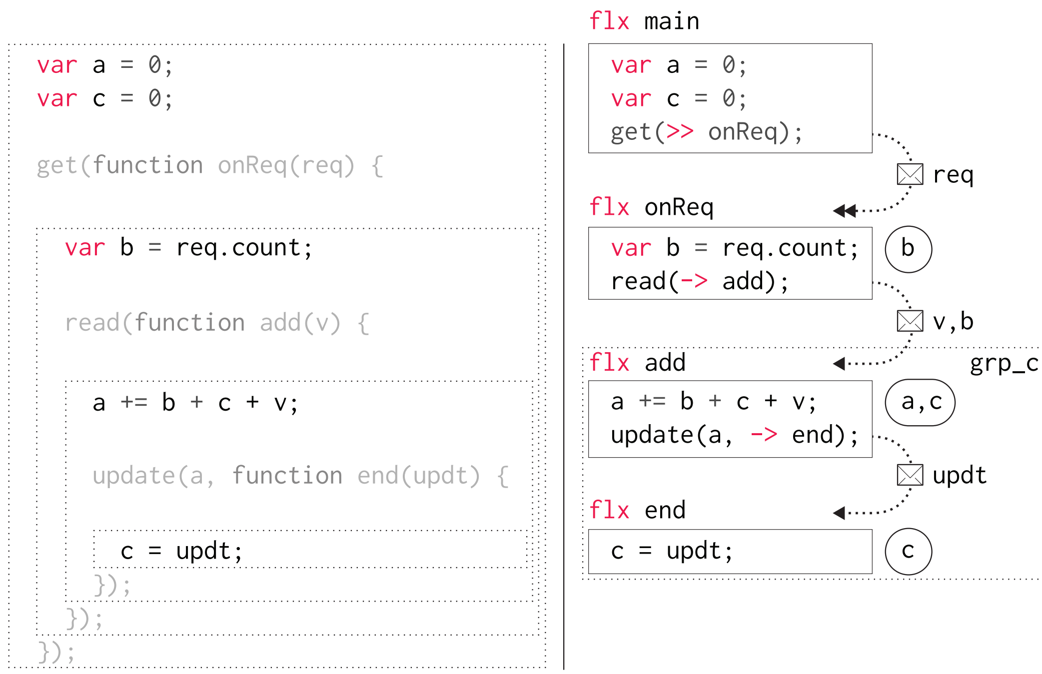


Figure 5.5 – Variable management from Javascript to the high-level fluxionnal language

Scope If a variable is modified inside only one application part in the current *post* chain, then the pipeliner adds it to the context of its fluxion.

In figure 5.5, the variable `a` is updated in the function `add`. The pipeliner step stores this variable in the context of the fluxion `add`.

Stream If a variable is modified inside an application part, and read inside downstream application parts, then the pipeliner makes the upstream fluxion add this variable to the message stream to be sent to the downstream fluxions. It is impossible to send variables to upstream fluxions, without race conditions. If the fluxion retro propagates the variable for an upstream fluxion to read, the upstream fluxion might use the old version while the new version is on its way.

In figure 5.5, the variable `b` is set in the function `onReq`, and read in the function `add`. The pipeliner step makes the fluxion `onReq` send the updated variable `b`, in addition to the variable `v`, in the message sent to the fluxion `add`.

Exceptionally, if a variable is defined inside a *post* chain, like `b`, then this variable can be streamed inside this *post* chain without restriction on the order of modification and read. Indeed, the execution of the upstream fluxion for the current *post* chain is assured to end before the execution of the downstream fluxion. Therefore, no reading of the variable by the upstream fluxion happens after the modification by the downstream fluxion.

Share If a variable is needed for modification by several application parts, or is read by an upstream application part, then it needs to be synchronized between the fluxions. To respect the semantics of the source application, we cannot tolerate inconsistencies. Therefore, the pipeliner groups all the fluxions sharing this variable within a same tag. And it adds this variable to the contexts of each fluxions.

In figure 5.5, the variable `c` is set in the function `end`, and read in the function `add`. As the fluxion `add` is upstream of `end`, the pipeliner step groups the fluxion `add` and `end` with the tag `grp_c` to allow the two fluxions to share this variable.

5.4 Real case test

The goal of this test is to prove the possibility for an application to be compiled into a network of independent parts. We want to show the current limitations of this isolation and the modifications needed on the application to circumvent these limitations.

We present a test of our compiler on a real application, gifsockets-server[1]. This application was selected from the `npm` registry because it depends on `express`, it is tested, working, and simple enough to illustrate this evaluation. It is part of the selection from a previous work.

[1]`https://github.com/twolfson/gifsockets-server`

This application is a real-time chat using gif-based communication channels. The server transforms the received text into a gif frame, and pushes it back to a never-ending gif to be displayed on the client. Listing 5.3 is a simplified version of this application.

```
1 var express = require('express'),
2     app = express(),
3     routes = require('gifsockets-middleware'),
4     getRawBody = require('raw-body');
5
6 function bodyParser(limit) {
7   return function saveBody(req, res, next) {
8     getRawBody(req, {
9       expected: req.headers['content-length'],
10      limit: limit
11    }, function (err, buffer) {
12      req.body = buffer;
13      next();
14    });
15  };
16 }
17
18 app.post('/image/text', bodyParser(1 * 1024 * 1024), routes.writeTextToImages);
19 app.listen(8000);
```

Listing 5.3 – Simplified version of gifsockets-server

On line 18, the application registers two functions to process the requests received on the url `/image/text`. The closure `saveBody`, line 7, returned by `bodyParser`, line 6, and the method `routes.writeTextToImages` from the external module `gifsockets-middleware`, line 3. The closure `saveBody` calls the asynchronous function `getRawBody` to get the request body. Its callback handles the errors, and calls `next` to continue processing the request with the next function, `routes.writeTextToImages`.

5.4.1 Compilation

We compile this application with the compiler detailed in section 5.3. The function call `app.post`, line 18, is a rupture point. However, its callbacks, `bodyParser` and `routes.writeTextToImages` are evaluated as functions only at runtime. For this reason, the compiler ignores this rupture point, to avoid interfering with the evaluation.

The compilation result is in listing 5.4. The compiler detects a rupture point : the function `getRawBody` and its anonymous callback, line 11. It encapsulates this callback in a fluxion named `anonymous_1000`. The callback is replaced with a stream placeholder to send the message stream to this downstream fluxion. The variables `req`, and `next` are appended to this message stream, to propagate their value from the `main` fluxion to the `anonymous_1000` fluxion.

When `anonymous_1000` is not isolated from the `main` fluxion, the compilation result works as expected. The variables used in the fluxion, `req` and `next`, are still shared between the two fluxions. Our goal is to isolate the two fluxions, to be able to safely parallelize their executions.

```
1  flx main
2  >> anonymous_1000 [req, next]
3    var express = require('express'),
4        app = express(),
5        routes = require('gifsockets-middleware'), //
6        getRawBody = require('raw-body');
7
8    function bodyParser(limit) { //
9      return function saveBody(req, res, next) { //
10       getRawBody(req, { //
11         expected: req.headers['content-length'], //
12         limit: limit
13       }, >> anonymous_1000);
14     };
15   }
16
17   app.post('/image/text', bodyParser(1 * 1024 * 1024), routes.writeTextToImages);
         //
18   app.listen(8000);
19
20 flx anonymous_1000
21 -> null
22   function (err, buffer) { //
23     req.body = buffer; //
24     next(); //
25   }
```

Listing 5.4 – Compilation result of gifsockets-server

5.4.2 Isolation

In listing 5.4, the fluxion `anonymous_1000` modifies the object `req`, line 23, to store the text of the received request, and it calls `next` to continue the execution, line 24. These operations produce side-effects that should propagate in the whole application, but the isolation prevents this propagation. Isolating the fluxion `anonymous_1000` produces runtime exceptions. We detail in the next paragraph, how we handle this situation to allow the application to be parallelized. This test highlights the current limitations of the compiler, and presents future works to circumvent them.

5.4.2.1 Variable `req`

The variable `req` is read in fluxion `main`, lines 10 and 11. Then it is associated in fluxion `anonymous_1000` to `buffer`, line 23. The compiler is unable to identify further usages of this variable. However, the side effect resulting from this association impacts a variable in the scope of the next callback, `routes.writeTextToImages`. We modified the application to explicitly propagate this side-effect to the next callback through the function `next`. We explain further modification of this function in the next paragraph.

5.4.2.2 Closure `next`

The function `next` is a closure provided by the `express` `Router` to continue the execution with the next function to handle the client request. Because it indirectly relies on network sockets, it is impossible to isolate its execution with the `anonymous_1000` fluxion. Instead, we modify `express`, so as to be compatible with the fluxionnal execution model. We explain the modification below.

```
1  flx main & express
2  >> anonymous_1000 [req, next]
3    var express = require('express'),
4        app = express(),
5        routes = require('gifsockets-middleware'), //
6        getRawBody = require('raw-body');
7
8    function bodyParser(limit) { //
9      return function saveBody(req, res, next) { //
10       getRawBody(req, { //
11         expected: req.headers['content-length'], //
12         limit: limit
13       }, >> anonymous_1000);
14     };
15   }
16
17   app.post('/image/text', bodyParser(1 * 1024 * 1024), routes.writeTextToImages);
       //
18   app.listen(8000);
19
20 flx anonymous_1000
21 -> express_dispatcher
22   function (err, buffer) { //
23     req.body = buffer; //
24     next_placeholder(req, -> express_dispatcher); //
25   }
26
27 flx express_dispatcher & express //
28 -> null
29   merge(req, msg.req);
```

```
30    next(); //
```

Listing 5.5 – Simplified modification on the compiled result

Originally, the function `next` is the continuation to allow the anonymous callback on line 11, to continue the execution with the next function to handle the request. To isolate the anonymous callback, this function is replaced on both ends. The result of this replacement is illustrated in listing 5.5. The `express Router` registers a fluxion named `express_dispatcher`, line 27, to continue the execution after the fluxion `anonymous_1000`. This fluxion is in the same group `express` as the `main` fluxion, hence it has access to network sockets, to the original variable `req`, and to the original function `next`. The call to the original `next` function in the anonymous callback is replaced by a placeholder to push the stream to the fluxion `express_dispatcher`, line 24. The fluxion `express_dispatcher` receives the stream from the upstream fluxion `anonymous_1000`, merges back the modification in the variable `req` to propagate the side effects, before calling the original function `next` to continue the execution, line 30.

After the modifications detailed above, the server works as expected for the subset of functionalities we modified. The isolated fluxion correctly receives, and returns its serialized messages. The client successfully receives a gif frame containing the text.

5.4.3 Future works

We intend to implement the compilation process presented into the runtime. A just-in-time compiler would allow to identify callbacks dynamically evaluated, and to analyze the memory to identify side-effects propagations instead of relying only on the source code. Moreover, this memory analysis would allow the closure serialization required to compile application using higher-order functions.

Chapter 6

Conclusion

Appendix A

Language popularity

A.1 PopularitY of Programming Languages (PYPL)

[1] The PYPL index uses Google trends[2] as a leading indicator of the popularity of a programming language. It search for the trend for each programming language by counting the number of searches of this language and the word "tutorial".

PYPL for May 2015

[1] http://pypl.github.io/PYPL.html

[2] https://www.google.com/trends/

Rank	Change	Language	Share	Trend
1		Java	24.1%	-0.9%
2		PHP	11.4%	-1.6%
3		Python	10.9%	+1.3%
4		C#	8.9%	-0.7%
5		C++	8.0%	-0.2%
6		C	7.6%	+0.2%
7		Javascript	7.1%	-0.6%
8		Objective-C	5.7%	-0.2%
9		Matlab	3.1%	+0.1%
10	2× ↑	R	2.8%	+0.7%
11	5× ↑	Swift	2.6%	+2.9%
12	1× ↓	Ruby	2.5%	+0.0%
13	3× ↓	Visual Basic	2.2%	-0.6%
14	1× ↓	VBA	1.5%	-0.1%
15	1× ↓	Perl	1.2%	-0.3%
16	1× ↓	lua	0.5%	-0.1%

A.2 TIOBE

[3]

The TIOBE index uses many search engines as an indicator of the current popularity of programming languages. It count the number of pages each search engine find when queried with the language name and the word "programming". This indicator indicates the number of resources available, and the discussions about a given programming language.

Javascript was the most rising language of 2014 in the TIOBE index.

TIOBE for April 2015

[3] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

Apr 2015	Apr 2014	Change	Programming Language	Ratings	Change
1	2	$\uparrow$	Java	16.041%	-1.31%
2	1	$\downarrow$	C	15.745%	-1.89%
3	4	$\uparrow$	C++	6.962%	+0.83%
4	3	$\downarrow$	Objective-C	5.890%	-6.99%
5	5		C#	4.947%	+0.13%
6	9	$\uparrow$	JavaScript	3.297%	+1.55%
7	7		PHP	3.009%	+0.24%
8	8		Python	2.690%	+0.70%
9	-	$2\times\uparrow$	Visual Basic	2.199%	+2.20%

A.3 Programming Language Popularity Chart

4

The programming language popularity chart indicates the activity of a given language in the online communities. It uses two indicators to rank languages : the number of line changed in github of, and the number of questions tagged with a certain language.

Javascript is ranked number one in this index. The Javascript community is particularly active online, and in the open source.

indeed.com

A.4 Black Duck Knowledge

5

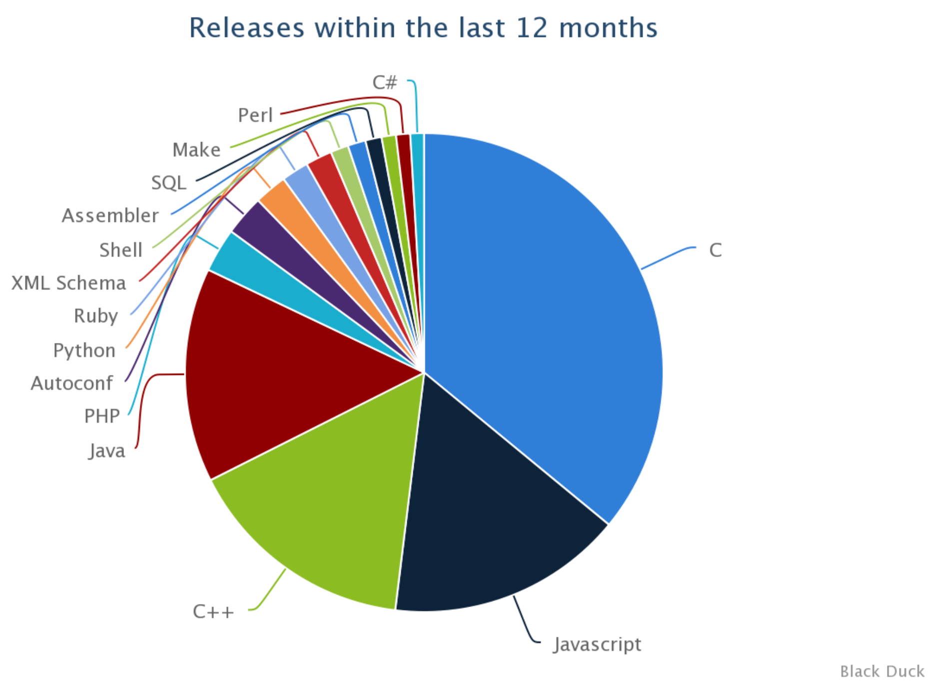
The black-duck, which analyze the usage of language on many forges, and collaborative hosts, rank Javascript number 2, after C, and with about the same usage as C++.

github.com sourceforge.net cpan.org rubyforge7.org planetsourcecode.com ddj.com

[4] http://langpop.corger.nl

[5] https://www.blackducksoftware.com/resources/data/this-years-language-use

Language	%
C	34.80
Javascript	15.45
C++	15.13
Java	14.02
PHP	2.87
Autoconf	2.65
Python	2.15
Ruby	1.77
XML Schema	1.73
Shell	1.18
Assembler	1.16
SQL	1.07
Make	0.94
Perl	0.92
C#	0.90



A.5 Github

http://githut.info/

A.6 HackerNews Poll

https://news.ycombinator.com/item?id=3746692

Language	Count
Python	3335
Ruby	1852
JavaScript	1530
C	1064
C#	907
PHP	719
Java	603
C++	587
Haskell	575
Clojure	480
CoffeeScript	381
Lisp	348
Objective C	341
Perl	341
Scala	255
Scheme	202
Other	195
Erlang	171
Lua	150
Smalltalk	130
Assembly	116
SQL	112
Actionscript	109
OCaml	88
Groovy	83
D	79
Shell	76
ColdFusion	51
Visual Basic	47
Delphi	45
Forth	41
Tcl	34
Ada	29
Pascal	28
Fortran	26
Rexx	13
Cobol	12

Bibliography

[1] D Abadi, D Carney, and U Cetintemel. "Aurora: a data stream management system". In: *Proceedings of the ...* (2003).

[2] DJ Abadi, Y Ahmad, and M Balazinska. "The Design of the Borealis Stream Processing Engine." In: *CIDR* (2005).

[3] DJ Abadi and D Carney. "Aurora: a new model and architecture for data stream management". In: *The VLDB Journal— ...* (2003).

[4] A Adya, J Howell, and M Theimer. "Cooperative Task Management Without Manual Stack Management." In: *USENIX Annual Technical Conference* (2002).

[5] T Akidau and A Balikov. "MillWheel: Fault-Tolerant Stream Processing at Internet Scale". In: *Proceedings of the VLDB Endowment 6.11* (2013).

[6] SP Amarasinghe, JAM Anderson, MS Lam, and CW Tseng. "An Overview of the SUIF Compiler for Scalable Parallel Machines." In: *PPSC* (1995).

[7] GM Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, spring joint ...* (1967).

[8] H Balakrishnan and M Balazinska. "Retrospective on aurora". In: *The VLDB Journal* (2004).

[9] M Bodin and A Charguéraud. "A trusted mechanised JavaSript specification". In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2014).

[10] S Chandrasekaran and O Cooper. "TelegraphCQ: continuous dataflow processing". In: *Proceedings of the ...* (2003).

[11] J Chen, DJ DeWitt, F Tian, and Y Wang. "NiagaraCQ: A scalable continuous query system for internet databases". In: *ACM SIGMOD Record* (2000).

[12] Melvin E. Conway. "Design of a separable transition-diagram compiler". In: *Communications of the ACM* 6.7 (July 1963), pp. 396–408. DOI: 10.1145/366663.366704.

[13] Edsger W. Dijkstra. "Guarded commands, nondeterminacy and formal derivation of programs". In: *Communications of the ACM* 18.8 (Aug. 1975), pp. 453–457. DOI: 10.1145/360933.360975.

[14] Edsger W. Dijkstra. "Letters to the editor: go to statement considered harmful". In: *Communications of the ACM* 11.3 (Mar. 1968), pp. 147–148. DOI: 10.1145/362929.362947.

[15] Edsger W. Dijkstra. "The structure of the "THE"-multiprogramming system". In: *Communications of the ACM* 11.5 (May 1968), pp. 341–346. DOI: 10.1145/363095.363143.

[16] Uffe Engberg and Mogens Nielsen. *A Calculus of Communicating Systems with Label Passing*. en. May 1986.

[17] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. "Making state explicit for imperative big data processing". In: *USENIX ATC* (2014).

[18] D Flanagan. *JavaScript: the definitive guide*. 2006.

[19] P Gardner and G Smith. "JuS: Squeezing the sense out of javascript programs". In: *JSTools@ ECOOP* (2013).

[20] JJ Garrett. "Ajax: A new approach to web applications". In: (2005).

[21] NJ Gunther. "A New Interpretation of Amdahl's Law and Geometric Scalability". In: *arXiv preprint cs/0210017* (2002).

[22] NJ Gunther. "A simple capacity model of massively parallel transaction systems". In: *CMG-CONFERENCE-* (1993).

[23] NJ Gunther. "Understanding the MP effect: Multiprocessing in pictures". In: *In other words* (1996).

[24] JL Gustafson. "Reevaluating Amdahl's law". In: *Communications of the ACM* (1988).

[25] Per Brinch Hansen. “Distributed processes: a concurrent programming concept”. In: *Communications of the ACM* 21.11 (Nov. 1978), pp. 934–941. DOI: 10.1145/359642.359651.

[26] C Hewitt. “Viewing control structures as patterns of passing messages”. In: *Artificial intelligence* (1977).

[27] C Hewitt, P Bishop, and R Steiger. “A universal modular actor formalism for artificial intelligence”. In: *Proceedings of the 3rd international joint conference on Artificial intelligence* (1973).

[28] C. A. R. Hoare. “Communicating sequential processes”. In: *Communications of the ACM* 21.8 (Aug. 1978), pp. 666–677. DOI: 10.1145/359576.359585.

[29] C. A. R. Hoare. “Monitors: an operating system structuring concept”. In: *Communications of the ACM* 17.10 (Oct. 1974), pp. 549–557. DOI: 10.1145/355620.361161.

[30] M Isard, M Budiu, Y Yu, A Birrell, and D Fetterly. “Dryad: distributed data-parallel programs from sequential building blocks”. In: *ACM SIGOPS Operating ...* (2007).

[31] Gilles Kahn. “The semantics of a simple language for parallel programming”. In: (1974).

[32] Gilles Kahn and David Macqueen. *Coroutines and Networks of Parallel Processes*. en. Tech. rep. 1976, p. 20.

[33] S Krishnamurthy and S Chandrasekaran. “TelegraphCQ: An architectural status report”. In: *IEEE Data Eng. ...* (2003).

[34] L Lamport. “Time, clocks, and the ordering of events in a distributed system”. In: *Communications of the ACM* (1978).

[35] B Liskov and L Shrira. *Promises: linguistic support for efficient asynchronous procedure calls in distributed systems*. 1988.

[36] D Logothetis, C Olston, and B Reed. “Stateful bulk processing for incremental analytics”. In: *Proceedings of the 1st ...* (2010).

[37] MD McCool. “Structured parallel programming with deterministic patterns”. In: *Proceedings of the 2nd USENIX conference on Hot ...* (2010).

[38] R Milner. “Processes: A mathematical model of computing agents”. In: *Studies in Logic and the Foundations of Mathematics* (1975).

[39] R Milner, J Parrow, and D Walker. "A calculus of mobile processes, I". In: *Information and computation* (1992).

[40] R Milner, J Parrow, and D Walker. "A calculus of mobile processes, II". In: *Information and computation* (1992).

[41] Robin Milner. "A calculus of communicating systems". In: *LFCS Report Series* 86.7 (1986), pp. 1–171. DOI: 10.1007/3-540-15670-4_10.

[42] G Moore. "Cramming More Components Onto Integrated Circuits". In: *Electronics* 38 (1965), p. 8.

[43] JP Morrison. *Flow-Based Programming*. 1994, pp. 1–377.

[44] DG Murray and M Schwarzkopf. "CIEL: a universal execution engine for distributed data-flow computing". In: *Proceedings of the ...* (2011).

[45] JF Naughton, DJ DeWitt, and D Maier. "The Niagara internet query system". In: *IEEE Data Eng. ...* (2001).

[46] R Nelson. "Including queueing effects in Amdahl's law". In: *Communications of the ACM* (1996).

[47] L Neumeyer and B Robbins. "S4: Distributed stream computing platform". In: *Data Mining Workshops ...* (2010).

[48] DL Parnas. "On the criteria to be used in decomposing systems into modules". In: *Communications of the ACM* (1972).

[49] R Power and J Li. "Piccolo: Building Fast, Distributed Programs with Partitioned Tables." In: *OSDI* (2010).

[50] Z Qian, Y He, C Su, Z Wu, and H Zhu. "Timestream: Reliable stream computation in the cloud". In: *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)* (2013).

[51] DP Reed. "" Simultaneous" Considered Harmful: Modular Parallelism." In: *HotPar* (2012).

[52] W. P. Stevens, G. J. Myers, and L. L. Constantine. "Structured design". English. In: *IBM Systems Journal* 13.2 (1974), pp. 115–139. DOI: 10.1147/sj.132.0115.

[53] GJ Sussman and GL Steele Jr. "Scheme: A interpreter for extended lambda calculus". In: *Higher-Order and Symbolic Computation* (1998).

[54] W Thies, M Karczmarek, and S Amarasinghe. "StreamIt: A language for streaming applications". In: *Compiler Construction* (2002).

[55] A Toshniwal and S Taneja. “Storm@ twitter”. In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data - SIGMOD '14* (2014).

[56] S Wei and BG Ryder. “State-sensitive points-to analysis for the dynamic behavior of JavaScript objects”. In: *ECOOP 2014–Object-Oriented Programming* (2014).

[57] M Welsh, D Culler, and E Brewer. “SEDA: an architecture for well-conditioned, scalable internet services”. In: *ACM SIGOPS Operating Systems Review* (2001).

[58] Sunny Wong, Yuanfang Cai, Giuseppe Valetto, Georgi Simeonov, and Kanwarpreet Sethi. “Design Rule Hierarchies and Parallelism in Software Development Tasks”. In: *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, Nov. 2009, pp. 197–208. DOI: 10.1109/ASE.2009.53.

[59] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Ulfar Erlingsson, Pradeep Kumar Gunda, Jon Currey, Frank McSherry, Kannan Achan, and Christophe Poulain. “Some sample programs written in DryadLINQ”. In: *Microsoft Research* (2009).