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

Etienne Brodu

September 25, 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'echelle 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 commencés 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 les exemples les plus flagrants. 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 languages 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 dites 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. Cela représente 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 consiste dans une certaine mesure à tenter d'écarter ce risque. Elle repose sur les deux observations suivantes. D'une part, Javascript est un

langage qui a énormément 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 développeur importante, et est 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 ressemble à un pipeline. La boucle événementielle de Javascript est un pipeline de fonctions qui s'exécutent sur un seul cœur pour profiter d'une mémoire globale.

L'objectif de cette thèse est d'étudier 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 permettant une exécution plus efficace que la première.

Nous étudions la possibilité pour cette équivalence de transformer un code d'une approche vers l'autre. Grace à cette transition, l'équipe de développement peut continuellement itérer le développement de l'application en suivant les deux approches à la fois, et ne pas se cloisonner dans une, et se coupant 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. 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 étant indépendantes, elles peuvent être déplacées d'une machine à l'autre. En ajoutant à un programme écrit en Javascript son expression en Fluxions, l'équipe de dévelopment peut le rendre scalable sans effort, tout en étant capable de répondre à la demande en fonctionnalités.

Contents

1 Introduction **4**
- 1.1 Web development 4
- 1.2 Performance requirements 5
- 1.3 Problematic and proposal 5
- 1.4 Thesis organization 6

2 Context and objectives **7**
- 2.1 Introduction 9
- 2.2 The Web as a platform 9
  - 2.2.1 From operating systems to the web 9
  - 2.2.2 The languages of the web 10
  - 2.2.3 Explosion of Javascript popularity 11
    - 2.2.3.1 In the beginning 11
    - 2.2.3.2 Rising of the unpopular language 12
    - 2.2.3.3 Current situation 14
- 2.3 Highly concurrent web servers 18
  - 2.3.1 Concurrency 18
    - 2.3.1.1 Scalability 19
    - 2.3.1.2 Time-slicing and parallelism 19
  - 2.3.2 Interdependencies 20
    - 2.3.2.1 State coordination 20
    - 2.3.2.2 Task scheduling 21
    - 2.3.2.3 Invariance 21
  - 2.3.3 Disrupted development 23
    - 2.3.3.1 Scalable concurrency 23
    - 2.3.3.2 The case for global memory 23
    - 2.3.3.3 Technological shift 24
- 2.4 Equivalence 25

2.4.1 Architecture of web applications . . . 25
2.4.1.1 Real-time streaming web services . . . 25
2.4.1.2 Event-loop . . . 25
2.4.1.3 Pipeline . . . 26
2.4.2 Equivalence . . . 27
2.4.2.1 Rupture point . . . 27
2.4.2.2 State coordination . . . 27
2.4.2.3 Transformation . . . 28

3 Different software systems modularities 29
3.1 Introduction . . . 30
3.2 Development growth . . . 31
3.3 Parallel execution . . . 31
3.3.1 Concurrency Theory . . . 31
3.3.1.1 Models . . . 31
3.3.1.2 Determinism and Non-determinism . . . 32
3.3.2 Message passing concurrency . . . 33
3.3.2.1 Programming models . . . 33
3.3.2.2 Scalability law . . . 34
3.3.2.3 Programming languages . . . 34
3.3.3 Stream Processing . . . 35
3.4 Reconciliations . . . 35

4 Pipeline parallelism for Javascript 36
4.1 Callback identification . . . 36
4.1.1 TODO . . . 36
4.2 Callback isolation . . . 36
4.2.1 Propagation of variables . . . 36
4.2.1.1 Scope identification . . . 36
4.2.1.2 Break the lexical scope . . . 37
4.2.1.3 Scope Leaking . . . 38
4.2.1.4 Propagation of execution and variables . . . 38

5 Conclusion 40

A Language popularity 41
A.1 PopularitY of Programming Languages (PYPL) . . . 41
A.2 TIOBE . . . 42

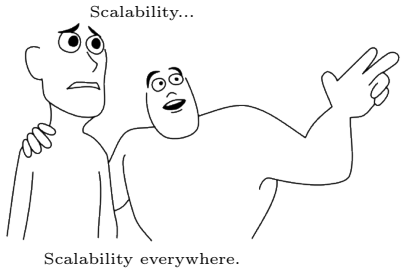
A.3 Programming Language Popularity Chart 43
A.4 Black Duck Knowledge 43
A.5 Github 45
A.6 HackerNews Poll 45

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 **??** presents the second contribution.

Chapter 2

Context and objectives

Contents

2.1 Introduction 9

2.2 The Web as a platform 9

2.2.1 From operating systems to the web 9

2.2.2 The languages of the web 10

2.2.3 Explosion of Javascript popularity 11

2.2.3.1 In the beginning 11

2.2.3.2 Rising of the unpopular language 12

2.2.3.3 Current situation 14

2.3 Highly concurrent web servers 18

2.3.1 Concurrency 18

2.3.1.1 Scalability 19

2.3.1.2 Time-slicing and parallelism 19

2.3.2 Interdependencies 20

2.3.2.1 State coordination 20

2.3.2.2 Task scheduling 21

2.3.2.3 Invariance 21

2.3.3 Disrupted development 23

2.3.3.1 Scalable concurrency 23

2.3.3.2 The case for global memory 23

2.3.3.3 Technological shift 24
2.4 Equivalence 25
2.4.1 Architecture of web applications 25
2.4.1.1 Real-time streaming web services 25
2.4.1.2 Event-loop 25
2.4.1.3 Pipeline 26
2.4.2 Equivalence 27
2.4.2.1 Rupture point 27
2.4.2.2 State coordination 27
2.4.2.3 Transformation 28

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 [8].

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 [9]. 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

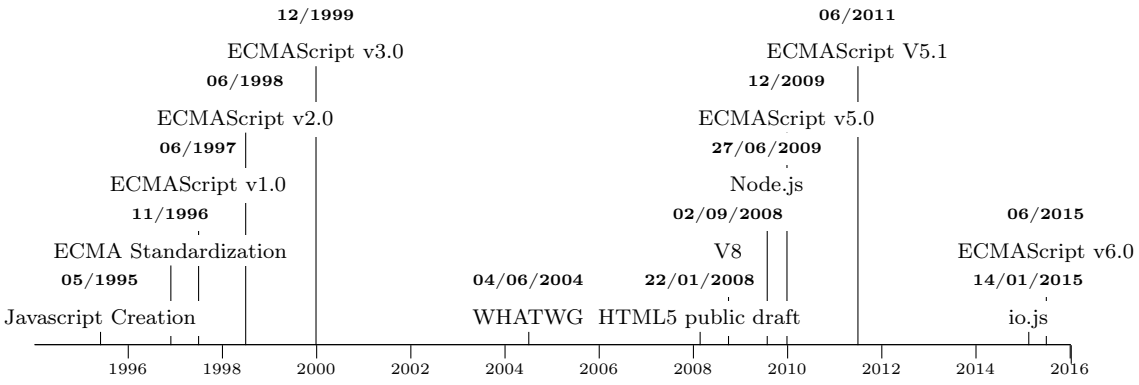
2.2.3.3 Current situation

"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
06/1998
ECMAScript v2.0
06/1997
ECMAScript v1.0
11/1996
ECMA Standardization
05/1995
Javascript Creation
04/06/2004
WHATWG
06/2011
ECMAScript V5.1
12/2009
ECMAScript v5.0
27/06/2009
Node.js
02/09/2008
V8
22/01/2008
HTML5 public draft
06/2015
ECMAScript v6.0
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.

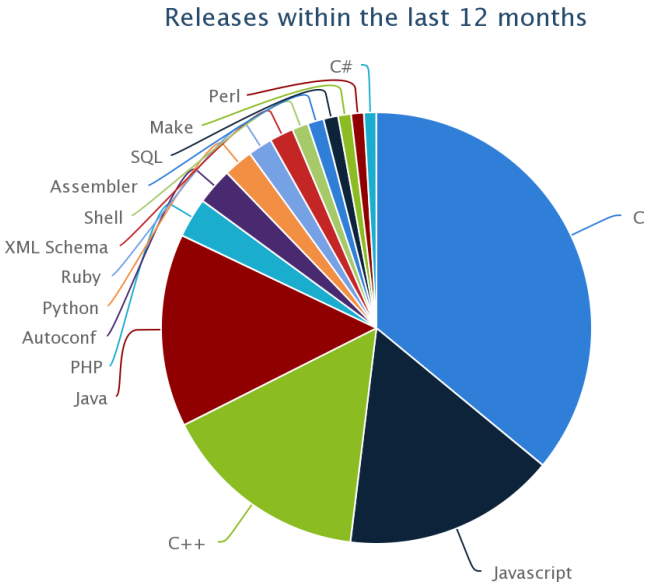
Github is the most important collaborative development platform, with around 9 millions users. Javascript is the most used language on github since mid-2011, with more than 320 000 repositories. The second language is Java with more than 220 000 repositories.

*

⚠ 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.

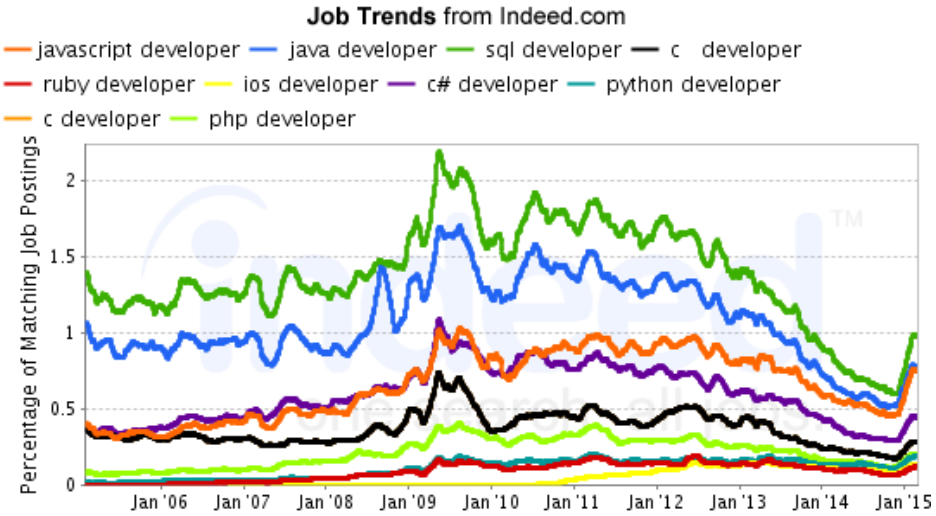
Black Duck Software helps companies streamline, safeguard, and manage their use of open source. For its activity, it analyzes 1 million repositories over various forges, and collaborative platforms to produce an index of the usage of programming language in open source communities. Javascript ranks second. 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

Different software systems modularities

Contents

3.1 Introduction 30
3.2 Development growth 31
3.3 Parallel execution 31
3.3.1 Concurrency Theory 31
3.3.1.1 Models 31
3.3.1.2 Determinism and Non-determinism 32
3.3.2 Message passing concurrency 33
3.3.2.1 Programming models 33
3.3.2.2 Scalability law 34
3.3.2.3 Programming languages 34
3.3.3 Stream Processing 35
3.4 Reconciliations 35

3.1 Introduction

In this thesis, we are interested in the evolution of the development of web applications, constrained by an economical context. In this chapter, we present a broader view on the development of software systems. We present different fields of work related to ours, to finally refine the scope of our research on the subject of our interest.

Since the beginning of the software development discipline, the best practices advocate to organize the code into independent units. It was called modular programming[27], structured design[29], hierarchical structure[6], object-oriented programming among other names. Each of these names represent slightly different approaches on the construction and composition of these units. One recurring principle, though, is that the units are to be as loosely coupled as possible. We are going to explain these approaches focused on improving the maintainability, and the growth of the software.

The focus of the software industry was to improve the development process. The Moore's law[25] assured an exponential evolution of the processing power, hence the execution speed was never to be a problem. Until the clock speed of processors could not be increased. The increasing number of transistors predicted by Moore's law were to be reorganized as several execution units into the same processor.

With multiple execution unit to coordinate, the decomposition of software system in modules have now two goals. The growth of the software, as well as the decomposition of execution on the several execution units. As Parnas famously showed in 1972[27], these two approaches to decomposition are incompatible. It seems impossible to develop a software following a decomposition that satisfies both the growth, and an efficient parallel execution. *

⚠ TODO more on that

With the incentive to leverage the execution power of parallel architecture, many work followed aimed at providing tools and model to organize the execution on multiple execution units. For example distributed systems, single instruction multiple data, among others.*

⚠ TODO not really good examples

There has been many attempts at reconciling the two approaches. But none seems really convincing enough to be widely adopted. Throughout this chapter, I will classify different works from the community in one of these three category : focus on development growth, focus on parallel execution, or reconciliation of the two.

My objective in this thesis, as previously stated, is to find an equivalence

between these two approaches in the case of streaming web applications.

3.2 Development growth

TODO

3.3 Parallel execution

Programming started with a very sequential nature, as we saw in the last chapter. But it eventually evolved toward concurrency to make advantage of the parallel architecture.

The first models of computation, like the Turing machine and lambda-calculus, were inherently sequential and based on a global state. A formalism was lacking 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 lie 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 is two main formal models for concurrent computations. The Actor Model of C. Hewitt, the Pi-calculus of R. Milner and the Communicating Sequential Processes of T. Hoare. Based on these definition, we explain the importance of determinism, and the reason that asynchronous message-passing prevailed.

3.3.1.1 Models

Actor Model The Actor model allows to express the computation as a set of communicating actors [**Clinger1981**, 15, 14]. 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.

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 on message passing parallel programming. For example, the Scala language advertises its use of an actor approach of concurrency.

The Actor model uses an asynchronous message-passing communication paradigm. The communication between two actors, the sender and the receiver, is a stream of discrete messages. It is asynchronous because, contrary to invocation, the sender doesn't wait for the result of the initiated communication.

Π-calculus R. Milner presented a process calculus to describe concurrent computation : the Calculus of Communicating Systems (CCS) [21, 24]. It is an algebraic notation to express identified processes communicating through synchronous labeled channels. The Pi-calculus improved upon this earlier work to allow processes to be communicated as values, hence to become mobile [7, 23, 22]. 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. Synchronous communications are deterministic. The message sent needs to be received to continue the execution on both ends. Because the concurrent executions and the communications in such system are both deterministic, the result of the concurrent system is assured to be deterministic. Determinism is a wanted property to assure the correctness of the execution.

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, called Byzantin fault [**Lamport1982**]. It makes the real communications means unable to provide the determinism required by the deterministic

models. The Actor model, on the other hands, was explicitly designed to take physical limitations in account [**Hewitt1977a**]. Eventually, All these works evolved to adopt asynchronous communications. Indeed, it is not realistic to build a distributed system based on synchronous communications.

Moreover, the total ordering of messages is only local to an actor, while between actors, messages are causally ordered. As Lamport showed [20], and Reed related later [28], causal order is sufficient to build a correct distributed system. The non-determinism in communications is hidden by the organization of the system. The execution will either terminate correctly, or not terminate at all.

3.3.2 Message passing concurrency

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 this level of asynchronous communication is unachievable due to overhead. In practice, most implementations feature independent synchronous processes communicating by messages.

3.3.2.1 Programming models

Conway defines coroutines as an autonomous program which communicate with adjacent modules as if they were input and output subroutines.[4] It seems to be the first definition of a parallel pipeline. *When coroutines A and B are connected so that A sends items to B, B runs for a while until it encounters a a read command, which means it needs something from A. The control is then transfered to A until it wants to write, whereupon the control is returned to B at the point where it left off.*

Hoare presented the Communicating Sequential Processes (CSP) [**Brookes1984**, 16]. These processes are executed concurrently, and communicates events via named channels. The evolutions of this model were influenced by, and influenced the work of Milner that led to π-calculus.

Similarly, Kahn developed the Kahn Networks [18, 19], following the work of Conway on coroutines. They are explicitly parallel coroutines separated by bounded FIFO streams for communication.

These programming models are highly similar, and differs only in details irrelevant for this thesis. However, it is interesting to note that they don't

correctly inherit from the Actor Model, as it is generally impossible 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.

3.3.2.2 Scalability law

These programming model were applied to run programs concurrently in machines providing a single processor, or shared resources among processors, like a common memory store, or network interface. To manage these resources, and avoid conflicting accesses, it is crucial to assure the mutual exclusion. For this purpose, Djikstra invented the Semaphore [**Dijkstra**].

Following this work, he also introduced guarded commands [5] and Hansen introduced guarded region [**Hansen1978a**]. Both assure the execution of a set of instructions to be exclusive to only one process.

Hoare introduced the monitor following the work of Hansen [17]. A monitor is an extension of a class, it regroups data and procedures, except that it assures its procedures to be entered only once at a time. With this restrictions, it guards against race condition on the access of a shared resource. Modula [30] and Concurrent Pascal [**Hansen1975**] uses Monitors.

Multi-threading programming extensively uses these concepts, because of the preemptive scheduling and the common storage. It is known to lead to bad performances, and difficulties in the development [1].

Amdahl warned against sequential portion of a program impacting the performance gained with parallelism [2]. Ghunter extended Amdahl's law to show that sharing resources, even protected, leads to bad, and even decreases performances with increasing parallelism [13, 11, 12, 26, 10]. Therefore, the only way to do an efficient concurrent application is to isolate the parallel processes in such way to limit the communication to the minimum. It is important the processes to be independent, and communicate solely by messages.

3.3.2.3 Programming languages

Scala / Akka / Erlang

3.3.3 Stream Processing

Nowadays, we see more industrial platforms for stream processing.

3.4 Reconciliations

TODO

Chapter 4

Pipeline parallelism for Javascript

From here, the reader should be confortable with the event-loop, and the analogy we drawn between the event-loop and a pipeline. The problematic is now clear : how to split the heap so that each asynchronous callback has its own exclusive heap ?

4.1 Callback identification

4.1.1 TODO

4.2 Callback isolation

We explain in this section the compilation process we developped to isolate the memory access for each callbacks. The result of this process should be two-fold. First each callback should have an exclusive access on a region of the memory. So that two different callback can be executed in parallel. And it should be clear for each callback, what are the variable needed from upstream callbacks, and what are the variable to send downstream.

4.2.1 Propagation of variables

4.2.1.1 Scope identification

In section **??**, we explained that Javascript is roughly lexically scoped. A consequence is that the declaration of contexts can be inferred statically. For example, in a lexically scoped, strongly typed, compiled language, the

compiler know the content of each scope during compile time, and can prepare the memory stack to store the variables in each scope.

In most languages, the memory is in two parts : the stack, and the heap. The stack is statically scoped, and its layout is known at compile time. The heap, on the other hand is dynamically allocated. Its layout is built at run time.

But Javascript is a dynamic language, perhaps the most dynamic of all languages. It doesn't have this distinction between stack and heap. Every variable is dynamically allocated on the heap. That induce two consequences. The first is that Javascript provides two statements to dynamically modify the lexial scope : `eval` and `with`. The second is that to know the layout of the heap, we need to use static analysis tools. In the next two sections, we adress these two consequences.

4.2.1.2 Break the lexical scope

Without these statements, `eval` and `with`, Javascript is lexically scoped. It is possible to infer the scope of each variable at compile time.

The `with` statement continue the execution using an expression as the lexical scope. As the provided expression is dynamically evaluated, it is possible to dynamically modify the lexical scope. The code snippet below show an example of such a situation.

```
1 var aliveCat = {isAlive: true};
2 var deadCat = {isDead: true}
3
4 with (Math.random() > 0.5 ? aliveCat : deadCat) {
5   isAlive;
6   // Half the time -> ReferenceError: isAlive is not defined
7   // Half the time -> true;
8 }
```

The variable `isAlive` is defined only in the object `aliveCat`. The presence of the variable `isAlive` in the lexical environment within the `with` statement cannot be determined statically, as the lexical environment is dynamically linked to either `aliveCat` or `deadCat`.

Note that the MDN reference page on `with`[1] says that *Using* `with` *is not recommended, and is forbidden in ECMAScript 5 strict mode.*

The

[1] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/with`

Not to be mistaken with the `this` operator. It is possible to dynamically change the content of an object,

```
function stuff() {
  this.x = 42;
}

stuff.call({})
```

However, even if Javascript is lexically scoped, the memory is still dynamicall allocated and manipualeted, so that it is not possible to actually infer the memory layout at compiler time only with lexical scope analysis, and without deeper static analysis.

4.2.1.3 Scope Leaking

To infer the layout of the heap at compile time, static analysis tools are used, like the points-to analysis, developped by Andersen in its PhD thesis [3]. For such analysis, the memory is splitted at the access scale. In low-level languages, like C/C++, the memory is mainly managed by the developer. Allowing access to the memory at a small grained scale : up to the address. It impose the analysis to split the memory to the adress scale in some cases. In higher-level languages, like Javascript, the developer cannot access the memory to the adress scale. The memory is accessed at a coarser scale : the property scale. (At the exception of some arrays and buffers, that mimic, and are mapped to actual memory adresses for performance reasons.)

4.2.1.4 Propagation of execution and variables

For the execution of each callback / stage, the corresponding part of the state is local, and the rest is remote, and inaccessible. We are going to explain why it must remain inaccessible.

While a callback is executing a request, the previous callback (the up stream callback) is executing the next request. The next request will arrive at the current callback some time in the future. The modification done in the state of the upstream callback will propagate only later in the current callback. The state of the upstream callback is in a different time frame than the state of the current callback.

To really understand that, we need to compare this execution with the execution on a unique event-loop. If the current callback executes, then the upstream callback might have, or might not have started to execute the next

request. But as soon as the current callback executes, the modifications done on the states, are immediatly propagated, so that the upstream callback can take them into account for the next request.

However, if the two callbacks are distant, then the modification of the current callback will not immediatly propagate to the upstream callback. During the propagation, the upstream callback might execute requests than would not be aware of the state modification from the current callback (from downstream). That is why we say the upstream callback and the current callback are in two different time frame. Propagating the state modification upstream is like going backward in time, it is impossible. That is why the execution, and the state modification propagation must always flow downstream.

As a note, I must add that if an upstream and a downstream callbacks are on the same event-loop, then this doesn't apply. it is like a loop in the time : the modification immediatly propagate from downstream to upstream.

Chapter 5

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	↑	Java	16.041%	-1.31%
2	1	↓	C	15.745%	-1.89%
3	4	↑	C++	6.962%	+0.83%
4	3	↓	Objective-C	5.890%	-6.99%
5	5		C#	4.947%	+0.13%
6	9	↑	JavaScript	3.297%	+1.55%
7	7		PHP	3.009%	+0.24%
8	8		Python	2.690%	+0.70%
9	-	2× ↑	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]

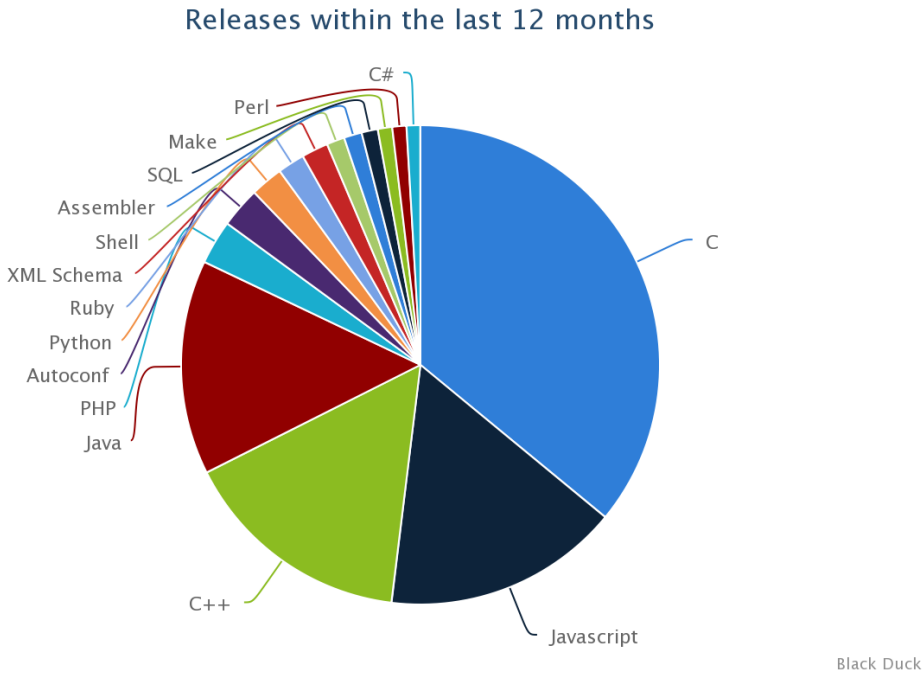
The black-duck, which analyze the usage of language on many forges, and collaborative hosts, rank Javascript number 2, after C, and with about the same usage as C++.

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

[2] 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).

[3] LO Andersen. "Program analysis and specialization for the C programming language". In: (1994).

[4] 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.

[5] 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.

[6] 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.

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

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

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

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

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

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

[13] JL Gustafson. “Reevaluating Amdahl’s law”. In: *Communications of the ACM* (1988).

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

[15] 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).

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

[17] 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.

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

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

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

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

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

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

[24] 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.

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

[26] R Nelson. “Including queueing effects in Amdahl’s law”. In: *Communications of the ACM* (1996).

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

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

[29] 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.

[30] N. Wirth. “Modula: A language for modular multiprogramming”. In: *Software: Practice and Experience* 7.1 (Jan. 1977), pp. 1–35. DOI: 10.1002/spe.4380070102.