# Operating Systems mutations, how and why integrate the user in the digital era ?

Etienne Brodu

April 29, 2015

## Abstract

Abstract

# Contents

# Chapter 1

# Introduction

# Chapter 2

# Context and Objectives

## 2.1 Javascript

### 2.1.1 History

Javascript was created by Brendan Eich in 10 days in May 1995. It was designed as a script for web pages. The language was taken in 1996 by the ECMA association under the name ECMAScript, and referenced by the specification ECMA-262.

### 2.1.2 Explosion of Javascript popularity

#### In the beginning

Javascript started as scripting language for web pages. But extended past this initial usage. It is now used in very different ways from embedded systems to desktop applications.

#### Rising of the unpopular language

Language popularity

Indexes : PYPL PopularitY of Programming Languages - Trends on Language + tutorial on google trends http://pypl.github.io/PYPL.html

TIOBE - trends on Language + programming on a lot of search engine. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html (Javascript is on the wall of fame for 2014, it has the most important rising in the given year)

Stack overflow tags http://makingdataeasy.com/stackoverflow-trends?tc=relative-button

Github repo http://adambard.com/blog/top-github-languages-2014/ https://bigquery.cloud.goo

**Current situation : complete world domination**

https://www.destroyallsoftware.com/talks/the-birth-and-death-of-javascript

The Atom editor is written in Javascript node.js.

Now, major PaaS (which one) support node.js by default.

Heroku support Python, Java, Ruby, Node.js, PHP, Clojure and Scala

Amazon Lambda Web service support node.js in priority.

»> News :

npm raises 8m. http://techcrunch.com/2015/04/14/popular-javascript-package-manager-npm-raises-8m-launches-private-modules/

**Isomorphic Javascript** https://www.meteor.com/

https://facebook.github.io/flux/

**Reactive** http://facebook.github.io/react/

Code reuse. Why it never worked ?

### 2.1.3 Overview of the language

#### Higher-order functions

#### Closures

A closure is a function associated with the environment in which it was declared. It is used in languages with higher-order functions, such as Javascript, to have lexical scoping.

### 2.1.4 Turn-based programming

#### the Javascript event-loop

Web pages are graphical environment offering multiple area of interaction for the user. Because of this multiplicity, the traditional linear programing model doesn't hold anymore. Graphical systems switched from this linear programming model to a different programming model focused on events.

Javascript uses higher-order functions. It is the ability for a language to manipulate functions like any other value. This ability is used to register a function to trigger after an event occurred. An event might be the click on an element of the page, for example.

Such a function is named a callback, a handler, a listener ... And it shift the programming paradigm from synchronous to asynchronous, which is a big deal.

In synchronous programming, the computation step are executed sequentially, one after the other. The program execution follows perfectly the program layout written in a linear textual file.

On the other hand, asynchronous programming allows a step back from this linearity.

A multi-threaded system allows the developer to explicitly express the parallelism in the application. A GOTO statement allows the developer to explicitly express the control flow in the application.

Asynchronous programming allows the program to manage the concurrency of the execution. Unlike a linear layout of an imperative program, it allows to express more finely the dependencies between instructions.

#### Promises

## 2.2 Scalability

We define a web service as a computer program whose main interface is based on web protocols, such as HTTP. Such a service uses resources allocated on a network of computers. Scalability defines the ability of the service to use a certain quantity of resource to meet a desired performance. We call system the association of the computer program and the available resources. The performance of this system is measured by its latency and throughput.

### 2.2.1 Latency and throughput

Latency is the time elapsed between the reception of a request, and the sent of the reply. It includes the time waiting for resources to be free to process the request, and the time to process the request.

Throughput is the number of requests processed by the system by unit of time.

Latency and throughput are linked in a certain way. If a modification of the web service reduces its mean latency to a half, then the throughput doubles immediately. It takes half the time to process a request, therefore, the service can process more requests in the same time. However, if throughput augment, the latency doesn't necessarily decrease.

### 2.2.2 Scalability granularity

We define a computer program as a set of operations. In the case of a web service, these operations can be directly requested by the user through the interface. An operation can cause any other operation to execute.

Because both the resources used and the operations executed are discrete : not infinitely divisible, scalability is inherently discrete.

Scalability granularity is the increment of resources. How the input data can be split up ? How the program can be deployed on many machines ?

We call system the association of the computer program with the resources

### 2.2.3 Horizontal and vertical scaling

There is two ways to augment the resources of the system. Enhance the nodes in the computer network - vertical scaling. Or add more nodes to the computer network - horizontal scaling.

There are three theories, from the most restrictive, to the most general.

Scalability is the property of a computer program to occupy available resources to meet a needed performance. EIther in Latency, or in throughput.

### 2.2.4 Linear scalability

Clements et. al. [**Clements2013a** ] prove that a computer program scale linearly if all its operations are commutative. Two operations are said to be commutative if they can be executed in any orders, and the same initial state will result in the same final state. Commutativity implies the two operations to be memory-conflict free, or independent, which is equivalent to say that they can be executed in parallel.

Therefore, to achieve linear scalability, a computer program must be composed of a set of operations that commutes. Thus, all the operations are parallel, they can be executed simultaneously, on any number of machines as required.

The size of the operations sets the scalability granularity.

However, commutativity is not achievable in real applications. Even sv6, the operating system resulting from the work on commutative scalability only has 99% commutativity. For real application, in the best case, the granularity is coarse, in the worst case, there is no possible commutativity because of shared resources (like a product inventory, or a friend graph).

### 2.2.5 Limited scalability

Amdahl introduced in 1967 a law to predict the limitation of speedup a computer program can achieve if a fraction of its code is sequential. Amdahl worked at increasing the speed of computer clock, while the scientific community was working on improving parallelism of computing machines.

In a set of operations, even if one is non-commutative, it cannot be executed in parallel of any others, the scalability is limited by this operation.

There is a difference if the operation is non-commutative with itself, or only with others. In the first case, it impose a queuing, while in the second case, it only increase the granularity : you can regroup the non-commutative operation with its subsequents, and form a bigger commutative operation.

### 2.2.6 Negative scalability

Gunther generalized Amdahl's law into the Universal Scalability Law. It includes the parallelization of non-independent operations with the use of synchronization.

It models the negative return on scalability from sharing resources observed in many real world applications.

### 2.2.7 Eventual Consistency

To overpass the scalability limits set by the previous rules, it is possible to abandon consistency. It simply tolerate incoherences between multiple replicas. The output of an operation can be false while its state is synchronized with the other replicas.

## 2.3 Concurrency

### 2.3.1 About concurrent systems

This demonstration focus on application depending on long waiting operations like I/O operations. Particularly, this demonstration focus on real-time web services. It is irrelevant for application heavily relying on CPU operations, like scientific applications.

Threads-based system and event-based system evolved significantly over the last half century. These evolutions were fueled by the long-running debate about which design is better. We try to succinctly and roughly retrace theses evolutions to understand the positions of each community. This demonstration show that thread and events are two faces of the same reality.

Lauer and Needham [**Lauer1979** ] presented an equivalence between Procedure-oriented Systems and a Message-oriented Systems.

Adya *et. al.* analyzed this debate and presented fives categories through which to present the problem [**Adya2002** ]. These two categories were often associated with thread-based systems and event-based systems. Their advantages and drawbacks were mistaken with those of thread and events. Adya *et. al.* explain in details two of these categories that are most representative, Task management and Stack management. We paraphrase these explanations.

#### Task management

Consider a task as an encapsulation of part of the logic of a complete application. All the task access the same shared state. The Task management is the strategy chosen to arrange the task executions in available space and time.

Preemptive task management executes each task concurrently. Their executions interleave on a single core, or overlap on multiple cores. It allows to leverage the parallelism of modern architectures. This parallelism has a cost however, developers are responsible for the synchronization of the shared memory. While accessing a memory cell, it must be locked so that no other task can modify it. Synchronization mechanism impose the developer to be especially aware of race condition, and deadlocks. These synchronization problems make concurrency hard to program with preemptive task management.

The opposite approach, Serial task management, executes each task to completion before starting the next. The exclusivity of execution assures an exclusive access on the memory. Therefore, it removes the need for synchronization mechanism. However, this approach is ill-fitted for modern applications, where concurrency is needed.

A compromise approach, Cooperative task management, allows tasks to yield voluntarily. A task may yield to avoid monopolizing the core for too long. Typically, it yields to avoid waiting on long I/O operations. It merges the concurrency of the preemptive task management, and the exclusive memory access. Thus, it relieves the developer from synchronization problems. But at the cost of dropping parallel execution.

Threads are associated with preemptive task management, and events with Cooperative task management. For this reason, it is commonly believed that synchronization mechanisms make threads hard to program [**Ousterhout1996** ]. While it is really Preemptive task management that is responsible for these synchronization problems [**Adya2002** ].

### Stack management

Consider a task is composed of several subtasks interleaved with I/O operations. Each I/O operation signal its completion with an event. The task stops at each I/O operation, and must wait the event to continue the execution. The stack management is the strategy chosen to express the sequentiality of the subtasks.

The automatic stack management is what is mostly used in imperative programming. The execution seems to wait the end of the operations to continue with the next instruction. The call stack is kept intact. This is what is commonly called synchronous programming.

In the manual stack management, developers need to manually register the handlers to continue the execution after the operation. The execution immediately continues with the next instruction, without waiting the completion of the operation. It implies to rip the call stack in two functions; one to initiate the operation, and another to retrieve the result. This is what is commonly called asynchronously programming.

What we argue is that synchronous is good because it is linear, it avoids stack ripping. But asynchronous is good because it allows parallelism by default.

Threads are associated with the automatic stack management, and events with manual stack management. For this reason, it is commonly believed that threads are easier to program. [**Thread systems allow programmers to express control flow** ] [**Behren2003** ]. However, the automatic stack management is not exclusive to threads. Fibers, presented by Adya *et. al.* is an example of cooperative task management with automatic stack management [**Adya2002** ] . Fibers present the advantage of cooperative task management, without the disadvantage of stack ripping. That is the ease of programming because of the absence of synchronization, without the difficulty of stack ripping.

We argue that the advantages of manual stack management outweigh its drawbacks for web services. Because of the numerous I/O operations, parallelism is

But what is actually highlighted is the automatic state management provided by threads. And with lighter context change, threads are a good choice which provide parallelism.

Historically, events-based system are associated with manual state management, while threads-based systems are associated with automatic state management. Manual state management imposed stack ripping [**Adya2002** ]. With closure, it is not the case anymore. Events now have automatic state management as well [**Krohn2007** ].

Now, there is implementation of thread model with cooperative management, with context-switch overhead improved enough to fill the gap with events model. And there is implementation of event model with automatic state management filling the gap with thread model. In this condition, we ask, what really is the difference between thread and events. We argue there is none. Except the isolation, versus sharing of the memory, which, again is not significant of either. In the first case, the different execution threads exchange messages, while in the second, they use synchronization mechanism to assure invariants in their states

For a single thread of execution, both model could avoid synchronization through cooperative task management, which assure invariants. Or avoid procedure slicing (if any) using synchronization. These are the two ends of a design spectrum. One end (cooperative task management) fits better for small processing with heavy use of shared resources. While the other end (synchronization) fits better for long processing with small use of shared resources. When one end of the design spectrum is used while the other should be used, one might expect unresponsiveness because of too heavy events, or performance fall due to interlocking.

Scalability is achieved through parallelism, which is itself achieved in our case (web servers) through cluster of commodity machines.

With distribution, this design spectrum gets a better contrast.

The synchronization of distributed, shared resources is limited through the CAP theorem [**Gilbert2002a** ]. Partition tolerance is a requirement of a distributed system. One needs to choose good latency (availability) or consistency. The CAP theorem is generalized into a broader theroem about[**Gilbert2012** ]

The isolation of resources implies to split the architecture in different stages, like Ninja [**Gribble2001** ], SEDA [**Welsh2000** ], or Flash [**Pai1999** ]. This splitting is difficult for the developer. The splitting which is good for the machine, is not the same as the one good for the design in modules.

The two ends of this design spectrum presented map directly onto the two kinds of parallelism advocated for scalability. That is pipeline parallelism, and data parallelism.

Pipeline parallelism is good for data locality, and important throughput. But each stage adds an overhead in latency.

Data parallelism is good for latency, because one request is processed from beginning to the end without waiting in queues. But it implies that the different machines share a common database. Which is a shared resource, and is limited by the CAP theorem.

Both parallelism have advantages and drawbacks, and both could be combined, like in the SEDA architecture. Ultimately, it would be possible to design a design spectrum to choose which kind of parallelism for a set of requirements. But we leave this for future works.

Splitting an architecture in stages is a difficult process, which prevent future code refactoring, and module modifications. We argue that the design for the technical architecture, and the design for the human minds should not be the same. Threads belong to the mental model, the design granularity Events belong to the execution model, the architecture granularity It is a mistake to attempt high concurrency without help from the compiler [**Behren2003** ]. Through compilation, we want to transform an event-loop based program (cooperative task management, no synchronization) into a pipeline parallelism distributed system. So, basically, we argue that it is possible to distribute one loop event onto multiple execution core.

/! WARNING The paper Why events are a bad idea states that : the control flow patterns used by these applications fell into three simple categories: call/return, parallel calls, and pipelines. Indeed, it is no coincidence that common event patterns map cleanly onto the call/return mechanism of threads. Robust systems need acknowledgements for error handling, for storage deallocation, and for cleanup; thus, they need a "return" even in the event model. » Why is it completly false ? It is crucial to find an answer. Moreover, Ayda et. al. state that : For the classes of applications we reference here [file servers and web servers], processing is often partitioned into stages. Other system designers advocated non-threaded programming models because they observe that for a certain class of high-performance systems [...] substantial performance improvements can be optained by reducing context switching and carefully implementing application-specific cache-conscious task scheduling.

The paper Why events are a bad idea states that : One could argue that instead of switching to thread systems, we should build tools or languages that address the problems with event systems (i.e., reply matching, live state management, and shared state management). However, such tools would effectively duplicate the syntax and run-time behavior of threads. » Well, yes ... With the exception of the stack junction. The paper on Duality had it right, their graph is correct, but for threads, it cannot be distributed because of stacks, while for events, it can.

Software evolution substantially magnifies the problem of function ripping: when a function evolves from being compute-only to potentially yielding, all functions, along every path from the function whose concurrency semantics have changed to the root of the call graph may potentially have to be ripped in two. (More precisely, all functions up a branch of the call graph will have to be ripped until a function is encountered that already makes its call in continuation-passing form.) We call this phenomenon "stack ripping" and see it as the primary drawback to manual stack management. Note that, as with all global evolutions, functions on the call graph may be maintained by different parties, making the change difficult. » Stack ripping is what I am talking about. While the stack are joined, it is not possible to distribute. If they say that stack ripping is necessary, that means it is not possible to encapsulate asynchronous function into synchronous function.

## 2.4 Objectives

### 2.4.1 LiquidIT

#### Development Scalability

#### Performance Scalability

### 2.4.2 Proposal and Hypothesis

# Chapter 3

# State of the Art

## 3.1 Flow-programming

## 3.2 ... ?

# Chapter 4

# Pipeline parallelism for Javascript

# Chapter 5

# Conclusion